On Computing The Fast Fourier Transform

RICHARD C. SINGLETON
Stanford Research Institute, Menlo Park, Calif.

Cooley and Tukey have proposed a fast algorithm for computing the complex Fourier transform

$$x_j = \sum_{k=0}^{n-1} \alpha_k \exp(i2\pi jk/n) \quad \text{for } j = 0, 1, \cdots, n-1,$$

and have shown major time savings in using it to compute large transforms on a digital computer. With n a power of two, computing time for this algorithm is proportional to $n \log_2 n$, a major improvement over other methods with computing time proportional to n^2. In this paper, the fast Fourier transform algorithm is briefly reviewed and fast difference equation methods for accurately computing the needed trigonometric function values are given. The problem of computing a large Fourier transform on a system with virtual memory is considered, and a solution is proposed. This method has been used to compute complex Fourier transforms of size $n = 2^{16}$ on a computer with 2^{15} words of core storage; this exceeds by a factor of eight the maximum radix two transform size with fixed allocation of this amount of core storage. The method has also been used to compute large mixed radix transforms. A scaling plan for computing the fast Fourier transform with fixed-point arithmetic is also given.

Introduction

The transformation of a time series to an amplitude spectrum, or in other words to a set of Fourier series coefficients, is a familiar task for a digital computer. Also, the inverse operation of evaluating a Fourier series is a similarly common task performed. Given a real sequence $\{x_j\}$ for $j = 0, 1, \cdots n-1$, with n even, the Fourier cosine and sine coefficients, respectively can be represented as

$$a_k = \frac{2}{n} \sum_{j=0}^{n-1} x_j \cos(2\pi jk/n) \quad \text{for } k = 0, 1, \cdots, n/2,$$

$$b_k = \frac{2}{n} \sum_{j=0}^{n-1} x_j \sin(2\pi jk/n) \quad \text{for } k = 1, 2, \cdots, n/2 - 1.$$

The original sequence can be recovered by the inverse relationship

$$x_j = \frac{a_0}{2} \sum_{k=1}^{n/2-1} [a_k \cos(2\pi jk/n) + b_k \sin(2\pi jk/n)] + \frac{a_{n/2}}{2} \cos(\pi j) \quad \text{for } j = 0, 1, \cdots, n-1.$$

A proof of this relationship can be found in any standard text on the subject, for example, Hamming [1]. Hamming also describes Goertzel's method [2] for computing the Fourier coefficients, a method in which cos $(2\pi jk/n)$ and sin $(2\pi jk/n)$ are computed by a multiple angle formula embedded in the summation to evaluate each pair a_k, b_k of coefficients. The method can also be adapted for Fourier series evaluation. The Goertzel algorithm is among the fastest methods in general use before the fast transform, but computing time increases as n^2 and roundoff errors grow rapidly with n.

The Cooley-Tukey fast Fourier transform algorithm is, except when n is a prime number, considerably faster and more accurate than the Goertzel method when compared on the same computer. This algorithm was preceded by several others of comparable efficiency, but less generality [3]. Danielson and Lanczos [4] showed how to compute a transform of dimension $2n$ with only slightly more than double the number of operations for a transform of length n and proposed using a sequence of doublings to compute larger transforms. Although their work was oriented toward simplified hand calculation of the coefficients of a real Fourier series, Rudnick [5] has reported use of the method on a digital computer. The fast Fourier transform for complex-valued (and multivariate) data was first formulated by Good [6], apparently without knowledge of the previous work of Danielson and Lanczos. In Good's solution, there remained one important limitation: in computing single-variate transforms, n was decomposed into mutually prime factors. Thus $n = 36$ could be factored as 4×9, but not $2 \times 2 \times 3 \times 3$. Cooley and Tukey

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

[7] generalized Good's algorithm to allow arbitrary factorization of the dimension of a single-variate transform. Their report of dramatic time savings has stimulated widespread interest in the fast Fourier transform.

The Cooley-Tukey fast Fourier transform algorithm can be used with any value of n, and the required number of arithmetic operations is proportional to n times the sum of the factors of n. The case of $n = 2^m$ is of special interest because of the ease of programming and is sufficiently general for most applications in which the user is free to choose n. Computing time for n a power of two increases as

$$n \log_2 n = nm$$

and roundoff errors increase slowly. Even for small n, this algorithm is faster and more accurate than previous methods, and it permits solution of large problems that would before have required an impractical amount of computing time.

In this paper, we consider several topics related to the fast Fourier transform. Reviewed first is the fast transform algorithm for n a power of two, giving shortcuts for computing the needed trigonometric function values, and showing a convenient method of transforming real-valued data. We then consider computing the fast transform on a system with virtual memory and rearrange the transform steps to reduce the number of memory overlay operations. The approach given is applicable to the mixed radix transform as well as the radix two transform. A scaling plan for computing the fast transform with fixed-point arithmetic is also given.

α_0	$\alpha_0 + \alpha_4$	$(\alpha_0+\alpha_4) + (\alpha_2+\alpha_6)$	$[(\alpha_0+\alpha_4) + (\alpha_2+\alpha_6)] + [(\alpha_1+\alpha_5) + (\alpha_3+\alpha_7)]$	$= x_0$
α_1	$\alpha_1 + \alpha_5$	$(\alpha_1+\alpha_5) + (\alpha_3+\alpha_7)$	$[(\alpha_0 + \alpha_4) + (\alpha_2+\alpha_6)] - [(\alpha_1+\alpha_5) + (\alpha_3+\alpha_7)]$	$= x_4$
α_2	$\alpha_2 + \alpha_6$	$(\alpha_0+\alpha_4) - (\alpha_2+\alpha_6)$	$[(\alpha_0+\alpha_4) - (\alpha_2+\alpha_6)] + c^2[(\alpha_1+\alpha_5) - (\alpha_3+\alpha_7)]$	$= x_2$
α_3	$\alpha_3 + \alpha_7$	$(\alpha_1+\alpha_5) - (\alpha_3+\alpha_7)$	$[(\alpha_0+\alpha_4) - (\alpha_2+\alpha_6)] - c^2[(\alpha_1+\alpha_5) - (\alpha_3+\alpha_7)]$	$= x_6$
α_4	$\alpha_0 - \alpha_4$	$(\alpha_0-\alpha_4) + c^2(\alpha_2-\alpha_6)$	$[(\alpha_0-\alpha_4) + c^2(\alpha_2-\alpha_6)] + c^1[(\alpha_1-\alpha_5) + c^2(\alpha_3-\alpha_7)]$	$= x_1$
α_5	$\alpha_1 - \alpha_5$	$(\alpha_1-\alpha_5) + c^2(\alpha_3-\alpha_7)$	$[(\alpha_0-\alpha_4) + c^2(\alpha_2-\alpha_6)] - c^1[(\alpha_1-\alpha_5) + c^2(\alpha_3-\alpha_7)]$	$= x_5$
α_6	$\alpha_2 - \alpha_6$	$(\alpha_0-\alpha_4) - c^2(\alpha_2-\alpha_6)$	$[(\alpha_0-\alpha_4) - c^2(\alpha_2-\alpha_6)] + c^3[(\alpha_1-\alpha_5) - c^2(\alpha_3-\alpha_7)]$	$= x_3$
α_7	$\alpha_3 - \alpha_7$	$(\alpha_1-\alpha_5) - c^2(\alpha_3-\alpha_7)$	$[(\alpha_0-\alpha_4) - c^2(\alpha_2-\alpha_6)] - c^3[(\alpha_1-\alpha_5) - c^2(\alpha_3-\alpha_7)]$	$= x_7$
	$\vert$— S_1 →$\vert$	— S_2 →$\vert$	— S_3 →$\vert$	

$c^k \equiv \exp(i2\pi k/8)$

FIG. 1. Example of fast Fourier transform algorithm for $n = 8$

The Fast Fourier Transform

The fast Fourier transform algorithm is a method of computing the finite complex Fourier transform

$$x_j = \sum_{k=0}^{n-1} \alpha_k \exp(i2\pi jk/n) \quad \text{for } j = 0, 1, \cdots, n-1,$$

where α and x are complex-valued vectors of length n, and i represents the imaginary unit $\sqrt{-1}$. In matrix notation, we can represent this transformation by

$$x = T\alpha$$

where T is the $n \times n$ matrix of complex-valued exponentials

$$\begin{aligned} t_{jk} &= \exp(i2\pi jk/n) \\ &= \cos(2\pi jk/n) + i \sin(2\pi jk/n) \\ &\qquad \text{for } j, k = 0, 1, \cdots, n-1. \end{aligned}$$

The inverse transform

$$\alpha_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j \exp(-i2\pi jk/n) \quad \text{for } k = 0, 1, \cdots, n-1$$

can similarly be represented by

$$\alpha = \frac{1}{n} T^* x,$$

where T^* is the complex conjugate of T; i.e.,

$$\begin{aligned} t^*_{jk} &= \exp(-i2\pi jk/n) \\ &= \cos(2\pi jk/n) - i \sin(2\pi jk/n) \\ &\qquad \text{for } j, k = 0, 1, \cdots, n-1. \end{aligned}$$

The matrices T and T^* have the relation

$$TT^* = T^*T = nI$$

where I is the $n \times n$ identity matrix.[1]

If n has m factors $n_1, n_2, \cdots n_m$, where $n = \prod_j n_j$, the Cooley-Tukey fast transform algorithm decomposes the transformation T into the product of m elementary

[1] When using a program not having the complex conjugate transform as an option, one can compute T^*x by forming the complex conjugate of x, transforming by T, then taking the complex conjugate of the result, i.e., $(Tx^*)^*$.

transformations, followed by a permutation P of the result.[2] In matrix notation, T can be represented by

$$T = PS$$
$$= PS_mS_{m-1} \cdots S_2S_1 .$$

Each step S_j is in turn composed of n/n_j transforms of dimension n_j. Since the number of arithmetic operations for a transform of dimension n_j is of the order of n_j^2, the total number of operations for the transformation S is of the order of $n \sum_j n_j$. The permutation P adds comparatively few additional operations.

Consider now the case $n = 2^m$. Each step S_j consists of computing $n/2$ transforms of dimension 2. In the transformation S_1, pairs of elements $n/2$ apart in the complex data vector are transformed. In S_2, pairs of elements $n/2$ apart in the upper and lower halves of the data vector are selected. In S_3, pairs of elements $n/4$ apart within each quarter of the data vector are selected, and so on until pairs of adjacent elements are selected in the transformation S_m. These steps are illustrated in Figure 1 for $n = 8$. We defer for the moment discussion of the sequence of exponential multipliers used and consider first the permutation P of the transform result.

To bring the results into normal binary order, we must interchange pairs of entries according to the following rule: We represent an index j as

$$j = j_{m-1}2^{m-1} + j_{m-2}2^{m-2} + \cdots + j_12^1 + j_0$$

and associate with it the reverse binary representation

$$r_j = j_02^{m-1} + j_12^{m-2} + \cdots + j_{m-2}2^1 + j_{m-1} ;$$

if $r_j > j$, we interchange entries r_j and j. Note that if $r_j = j$, as is true, for example, for $j = 0, n - 1, n/2 - 2, n/2 + 1, \cdots$, the jth entry remains fixed. The permutation matrix P of this interchange is symmetric and $PP = I$, i.e., permuting twice gives an identity permutation. Thus, a computing procedure for permuting from reverse binary to normal order can also be used for permuting data from normal to reverse binary order.

In computing the fast Fourier transform, we can alternatively permute the data first, then compute the transform. Thus

$$\alpha = Tx = PSx$$
$$= (PSP)Px$$
$$= (PS_mP)(PS_{m-1}P), \cdots, (PS_1P)Px.$$

Whereas in S_j, we selected pairs of elements 2^{m-j} apart, in (PS_jP), we select pairs 2^{j-1} apart. Thus in (PS_1P), we select adjacent pairs of data elements and in PS_mP we select pairs of elements $n/2$ apart. In some applications, we will wish to have computing procedures for both transformations S and (PSP) to avoid the need for permutation. We can then transform a data vector with S, modify the result, and transform back with (PS^*P).

In computing the radix 2 transform in place, the schedule of pairs of data elements used depends on the original ordering of the data. If the data are in normal order, we start with pairs $n/2$ apart in the first step and end with adjacent pairs in the last step; the result is left in reverse binary order. If the data are in reverse binary order, we start with adjacent pairs in the first step and end with pairs $n/2$ apart in the last step; the result is then in normal order.

We now consider the sequence of complex exponential multipliers used in the radix 2 transform,[3] or, equivalently, the required sequence of pairs of sine and cosine values. The example in Figure 1 is based on the form of the fast transform algorithm proposed by Cooley and Tukey [7]. If we consider the $n = 16$ case, the required multiples of $2\pi/16$, if we take the eight pairs of data elements in order according to the lower index member of each pair, are:

0, 0, 0, 0, 0, 0, 0, 0	1st step
0, 0, 0, 0, 4, 4, 4, 4	2nd step
0, 0, 4, 4, 2, 2, 6, 6	3rd step
0, 4, 2, 6, 1, 5, 3, 7	4th step.

We note that if the data elements are indexed in normal order, the trigonometric function values are used in reverse binary order. For the corresponding transform (PSP) with the data originally in reverse binary order, the required multiples of $2\pi/16$ are:

0, 0, 0, 0, 0, 0, 0, 0	1st step
0, 4, 0, 4, 0, 4, 0, 4	2nd step
0, 2, 4, 6, 0, 2, 4, 6	3rd step
0, 1, 2, 3, 4, 5, 6, 7	4th step.

In this case, trigonometric function values can be generated in normal sequence as needed. However, for efficient use of these values, we need to step through the data array more than once in all but the first and last steps. This can be a disadvantage if we are using a computing system allowing automatic overlay of data storage.

We can invert the transformation S by using the transformation $1/n(PS^*P)$, since

$$I = \frac{1}{n} T^*T = \frac{1}{n} (PS^*)(PS) = \frac{1}{n} (PS^*P)S.$$

[2] We assume throughout this paper that the transform is computed in place with the results of each transform of dimension n_j replacing the previous values.

[3] This topic is discussed by Gentleman and Sande [8] for the mixed radix case.

Alternatively, we can express the inverse as:

$$S^{-1} = S_1^{-1} S_2^{-1} \cdots S_m^{-1},$$

the product of the inverses of the m steps S_j in the transform S. Considering an elementary dimension 2 transform

$$y_j = x_j + x_k \exp(i\theta)$$
$$y_k = x_j - x_k \exp(i\theta)$$

in S, we see that its inverse is

$$x_j = \tfrac{1}{2}(y_j + y_k)$$
$$x_k = \tfrac{1}{2}(y_j - y_k) \exp(-i\theta).$$

Using this modified arithmetic, we can backtrack through the steps of transform S. Considering again the first $n = 16$ example above, we now use the trigonometric function sequence of the 4th step, the 3rd step, the 2nd step, then finally the 1st step.

Similarly if our data are in normal order, we can backtrack through the steps of the transform (PSP). Considering the second $n = 16$ example above, we now use the trigonometric function sequence of the 4th step, the 3rd step, the 2nd step, and finally the 1st step. Sande [8] has proposed using this organization of the fast transform and points to the advantage of using the trigonometric function values in normal order.

The author has coded ALGOL radix 2 transforms based on the $(PSP)^{-1}$ organization for data in normal order and based on the (PSP) organization for data in reverse binary order [9] for use on a system with fixed memory allocation. The S and S^{-1} organizations are used in a later section of this paper as the basis of a method to be used on a computer system with memory overlay; the advantage of being able to use each pair of trigonometric values on a single sequence of consecutive data points outweighs the disadvantage of generating them in reverse binary order.

Computing Trigonometric Function Values

Counting one sine and cosine value for each 2×2 transform, the fast Fourier transform of $n = 2^m$ data points requires nearly $2n$ trigonometric function values. In most methods of computing the fast transform, we can reduce this number to n by using the relations:

$$\cos\left(\frac{\pi}{2} + \theta\right) = -\sin(\theta)$$
$$\sin\left(\frac{\pi}{2} + \theta\right) = \cos(\theta).$$

We may further reduce the number to $n/2$ by using similar relations for the angles $\pi/2 - \theta$ and $\pi - \theta$. If we complete each of the m transform steps before going on to the next, further reduction in the number of trigonometric values requires storing some values for later use.

If sufficient high speed storage is available, we can table the values

$$\sin(2\pi j/n) \quad \text{for } j = 0, 1, \cdots, n/4;$$

TABLE 1. EXTRAPOLATED VALUES OF COS $\pi/2$ AND $1 - \sin \pi/2$ FOR TWO DIFFERENCE EQUATION METHODS

Number of extrapolations	First method (value in units of 10^{-12})		Second method (value in units of 10^{-12})	
	cos $\pi/2$	$1 - \sin \pi/2$	cos $\pi/2$	$1 - \sin \pi/2$
2^4	0	0	−5.91	0
2^5	−11.60	3.64	0.68	0
2^6	6.59	−29.10	−13.87	0
2^7	−14.81	0	−8.44	0
2^8	−0.91	−14.55	−11.60	0
2^9	−5.74	0	−21.00	0
2^{10}	−1.62	1.82	16.79	5.46
2^{11}	−14.41	−29.10	−60.25	−14.55
2^{12}	114.18	34.56	88.36	12.73
2^{13}	31.61	1.82	−2.38	0
2^{14}	−3.91	23.65	53.18	9.09
2^{15}	243.07	−494.77	−0.62	0
2^{16}	127.04	−29.10	119.22	14.55
2^{17}	−60.92	50.93	149.99	1.82

this table contains all of the values needed in the transform computation. The entries can be quickly filled in by a binary process. We first enter the values for 0, $\pi/4$, and $\pi/2$. On the next step, the values for $\pi/8$ and $3\pi/8$ are computed, using the relation

$$\sin((k+1)\theta) = \frac{\sin(k\theta) + \sin((k+2)\theta)}{2\cos\theta},$$

where $\theta = \pi/8$. At each succeeding step the number of values is doubled until the table is complete. If the values are used in reverse binary sequence in computing the fast transform the table may then be permuted for convenience in indexing.

Another approach is to generate the trigonometric function values for successive multiples of $2\pi/n$ by difference equation. Since we extrapolate from initial values, care in selecting a method is needed. We show two methods that give satisfactory accuracy. In the first, we generate cosine and sine values as independent sequences, using the second difference relations

$$C_{k+1} = R \times \cos(k\theta) + C_k$$
$$\cos((k+1)\theta) = \cos(k\theta) + C_{k+1}$$
$$S_{k+1} = R \times \sin(k\theta) + S_k$$
$$\sin((k+1)\theta) = \sin(k\theta) + S_{k+1},$$

where the constant multiplier is

$$R = -4\sin^2(\theta/2)$$

and the initial values are

$$C_0 = 2\sin^2(\theta/2)$$
$$S_0 = \sin(\theta)$$

$$\cos(0) = 1$$
$$\sin(0) = 0.$$

Another method, in which the sine and cosine are computed as a pair with four multiplications instead of two, is as follows:

$$\cos((k+1)\theta) = [C \times \cos(k\theta) - S \times \sin(k\theta)] + \cos(k\theta)$$
$$\sin((k+1)\theta) = [C \times \sin(k\theta) + S \times \cos(k\theta)] + \sin(k\theta),$$

where

$$C = -2\sin^2(\theta/2)$$
$$S = \sin(\theta).$$

The two methods were tested using initial values calculated by the library trigonometric function procedure on a Burroughs B5500 computer, then values for $\cos \pi/2$ and $\sin \pi/2$ were computed by extrapolation. Results are given in Table I. The computer used has a 39-bit floating-point mantissa, with octal exponent; thus for a number between 1/8 and 1, a change in the low order bit gives a difference of 1.8×10^{-12}. Both methods result in a low level of errors, but when used in a Fourier transform program, the second method often yields about one binary place more accuracy than the first.

Transforming Real Data

If the data vector to be transformed is real-valued, one can augment the data with an imaginary component vector of zeros and then use a subroutine for computing the complex Fourier transform. The resulting Fourier coefficients α_k for $k = 0, 1, \cdots n-1$ have a complex conjugate symmetry about $n/2$, thus

$$\alpha_{n-k} = \alpha_k^* \quad \text{for } k = 1, 2, \cdots n-1.$$

As pointed out by Cooley [10], this symmetry allows us to transform simultaneously two real vectors x and y and separate the results. We compute the transform

$$\lambda = \frac{1}{n} T(x + iy)$$
$$= \frac{1}{n} Tx + i\frac{1}{n} Ty$$
$$= \frac{1}{2}(\alpha + i\beta)$$

where α and β are the real Fourier transforms of x and y, and then recover α and β by

$$\alpha_k = \begin{cases} \lambda_0 + \lambda_0^* & \text{for } k = 0 \\ \lambda_k + \lambda_{n-k}^* & \text{for } k = 1, 2, \cdots, n/2, \end{cases}$$
$$\beta_k = \begin{cases} i(\lambda_0^* - \lambda_0) & \text{for } k = 0 \\ i(\lambda_{n-k}^* - \lambda_k) & \text{for } k = 1, 2, \cdots, n/2. \end{cases}$$

Bingham, Godfrey, and Tukey [11] point out that this technique can be used to transform a single sequence of real data points if n is divisible by two. Alternate data points are stored in the real and imaginary components of the complex vector to be transformed. After transforming, permuting to normal order, and separating the Fourier coefficients of the two sequences, the two coefficient sets are combined in a final step. In using this method, two reorderings are needed if the real data are initially in a single sequence, one before the transform and another after. For n a power of two, the final permutation from reverse binary to normal order can be done in place in one step by pair interchanges. However, the initial permutation is not as easily done in place.[4] Here we propose a simple solution to this problem.

Let us suppose that the original data are stored in normal sequence, with the first $n/2$ entries in the real vector A and the second $n/2$ entries in the imaginary vector B of the transform. We then in two steps permute the data so that the even numbered entries are in A and the odd-numbered entries in B, both in reverse binary order. If we then compute the transform with a procedure operating on complex data originally in reverse binary order, the transformed result is in normal order with no further permutation needed. In the first step, we interchange entries

$$A_{j+1} \text{ and } B_j \quad \text{for } j = 0, 2, \cdots, (n/2) - 2,$$

which leaves the even-numbered entries in A and the odd-numbered entries in B. In the second step, we permute A and B independently to reverse binary order, ignoring the low-order bit, as follows: for $j = 0, 2, \cdots (n/2) - 2$, let k be the reverse binary value of $j/2$; then if $k > (j/2)$, interchange A_{2k} and A_j and interchange A_{2k+1} and A_{j+1}. The same permutation is applied to the B vector. The two steps for $N = 16$ are shown in Figure 2, where the numbers listed are the original indices of the data. It should be noted that the order of doing the two steps of the permutation may be interchanged if desired.

A	B	A	B	A	B
0	8	0	1	0	1
1	9	8	9	8	9
2	10	2	3	4	5
3	11	10	11	12	13
4	12	4	5	2	3
5	13	12	13	10	11
6	14	6	7	6	7
7	15	14	15	14	15
Original order		First permutation		Second permutation	

FIG. 2. Permutation of real data before Fourier transform

The inverse operation of evaluating a real Fourier series, given a set of cosine and sine coefficients, can be done by

[4] We assume that the real and imaginary components of complex data are stored in separate arrays. If interleaved, as with some FORTRAN compilers, the problem considered here does not exist.

backtracking through the steps described above, computing the inverse of each. While this is usually the most convenient method, another method offers advantages in computing convolutions. If we transform two real sequences of length n to the frequency domain and form the product $\alpha\beta^*$ of the two transforms, the resulting n complex values are the Fourier coefficients of the convolution, a real sequence of length n. Thus, the real component a is an even function, the imaginary component ib is an odd function, and Ta and $T(ib)$ are both real. Since

$$T(a + ib) = \text{Re}\,[T(a - b)] + \text{Im}\,[T(a - b)],$$

we can subtract b from a, transform, then add the real and imaginary components to get the convolution. The advantage of this approach is that we can perform all of the frequency domain operations with the data in reverse binary order, reducing the intermediate result to a complex vector of length $n/2$ at this stage before transforming to the time domain. With the result in normal order, we then separate the transforms of the even- and odd-numbered entries and combine them, using the algorithm for computing Fourier coefficients described above. A final addition then gives the convolution. An ALGOL convolution procedure based on this approach is available [9].

Methods for Use on a Virtual Memory System

In using the fast transform algorithm on the Burroughs B5500 computer, we have frequent need to transform data sets of a size exceeding the 2^{15} words of core storage. One solution, using auxiliary memory for data storage, has been explored previously [12]. Here we consider use of the computer's virtual core memory feature.

Virtual memory and multiprogramming are standard features of the B5500 computer system. Data and programs are stored in variable-length segments of up to 1023 words. In ALGOL, array rows correspond to data segments, and for large radix two transforms, we store data in two-dimensional arrays with rows of 512 words in length. When a program refers to an element of an absent array row, the operating system takes control and finds core space for the row. Space occupied by program segments not currently in use is taken first, then data segments are copied to disk storage generally on a first-in, first-out basis, without regard to the program to which they belong. Then an input operation reads the desired row into core and control is returned to the user program. This organization places a premium on doing as much computing as possible with a small set of array rows before going on to others. Memory overlay operations begin when the total storage requirement of the programs in the "mix" exceeds core capacity.

The Sande version of the fast transform, while very efficient for transforming a single segment of data, is highly vulnerable to slowdown through memory overlay. A complex transform of size $n = 2^{13}$ can be computed without difficulty on the B5500 if it is run without multiprogramming, but the addition of another program of comparable size will slow the transform program almost to a halt. Fortunately, the Cooley version will run under the same conditions with only a moderate increase in computing and input-output channel time. In this section, we show a rearrangement of the Cooley version giving a more efficient schedule of data accesses. With this arrangement, input-output channel time caused by memory overlay increases

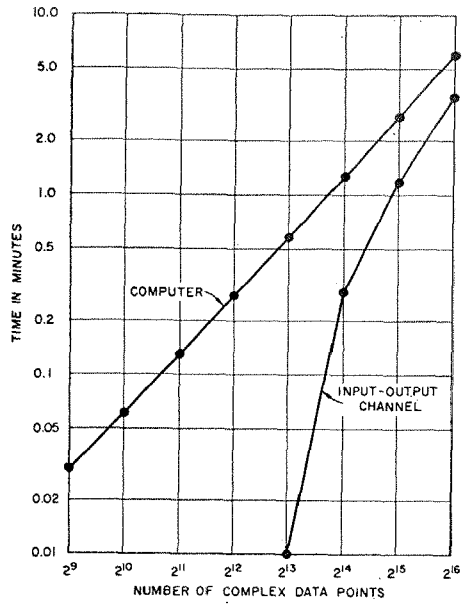

FIG. 3. Computer and input-output channel times for the fast Fourier transform on a virtual memory system with 2^{15} words of core storage

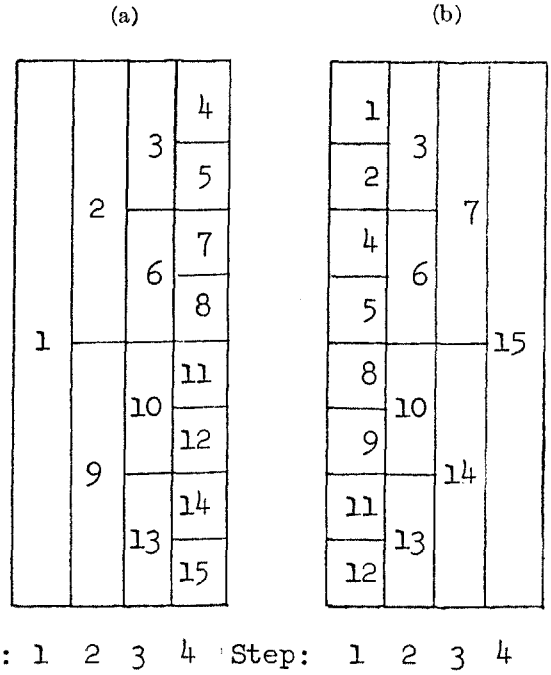

FIG. 4. Computing sequence for Fourier transform with virtual memory, $n = 16$; (a) data in normal order; (b) data in reverse binary order

slowly as core capacity is exceeded. In Figure 3, we show a plot of computer and input-output channel time for a radix two transform based on this arrangement.[5]

In the first step of the radix two fast transform algorithm, data pairs are taken in serial order from the full data set, one element from the lower half and the other from the upper half of the data vector. However, after the first step in the transform, the two halves of the complex data arrays are independent. Similarly, after the second step, the four quarters are independent, and so on. Thus, after the first step, we can complete the remaining $m - 1$ steps in the first half of the data before going on to the second half. And after the second step in the first half, we can complete the remaining $m - 2$ steps on the first quarter of the data before going on to the second quarter. Continuing in this manner, we can compute the first pair of entries in the final result, having computed only the necessary intermediate results, then compute the second pair of entries in the final result, and so on. The sequence of steps for the $n = 16$ case is shown in Figure 4(a). For most of the computing, the needed subset of the data is small enough to fit within the high speed memory of the computer.

In this method, we use the arithmetic of the Cooley version of the fast transform, thus the angle θ of the exponential multiplier remains constant within each of the numbered sections in Figure 4. When a segment splits into two halves in moving to the adjacent transform step, the angle for the lower half is $\theta/2$ and the angle for the upper half is $(\theta + \pi)/2$; a single sine-cosine pair serves for the two halves. We can use the library trigonometric function procedure to compute sin $(\theta/2)$, then calculate cos $(\theta/2)$ by the relation

$$\cos(\theta/2) = \frac{\sin(\theta)}{2\sin(\theta/2)} \quad \text{for } 0 < \theta < \pi,$$

where sin (θ) is remembered from the previous segment. In this way, the trigonometric function library procedure is called only $(n/2) - m$ times.

0	0	0
8	1	1
4	4	2
12	5	3
2	2	4
10	3	5
6	6	6
14	7	7
1	8	8
9	9	9
5	12	10
13	13	11
3	10	12
11	11	13
7	14	14
15	15	15
	P_1	P_2

FIG. 5. Steps of reverse binary to binary permutation

[5] In an alternate version of this program, using radix eight arithmetic for the final steps within a single real-imaginary pair of array rows, computing time is reduced by one-third, while input-output channel time remains unchanged.

When we start with data in reverse binary order, a similar method can be used. We do the first step of the transform on the first pair of data entries, then on the second pair. Before going on to the third pair, we first do the second step of the transform on the first four entries. After each initial pair is transformed by the first step, we proceed to all possible subsequent steps. The sequence of steps for $n = 16$ is shown in Figure 4(b). The trigonometric functions are computed as above, except that we store a list of the next sine-cosine pair for each of the m steps.

The author has written ALGOL procedures for computing the fast Fourier transform on a virtual memory system, both for the radix two case [13] and for the mixed radix case [14]. In the mixed radix case, after doing a step of the transform with a factor n_j on a portion of the data array, when transforming data originally in normal order, that portion of the data can be subdivided into n_j independent sections for further transform steps.

For the $n = 2^m$ case, the reverse binary-to-binary permutation can also be organized to reduce time loss through memory overlay. In this case, the permutation is done in several steps. In the first step, entries in the lower and upper halves of the complex vector are interchanged so that each entry is then in its correct half. In the second step, entries within each half are interchanged so that each entry is in its correct quarter. This process continues until the desired permutation has been completed. An example for $n = 2^4$ is given in Figure 5. In the first step, single entries in each vector are interchanged, and in the second step, pairs of entries are interchanged. For larger n, the process continues with entries interchanged 2^{k-1} at a time in the kth step. For $n = 2^m$, $[m/2]$ steps are required, and at each step half the entries in each vector are interchanged and the other half remain fixed. On the B5500 computer, this permutation method can be speeded up by doing the block memory interchanges in character rather than word mode. Similar speed-ups can be achieved on other computers with block-memory transfer instructions.

The permutation procedure has a partial generalization in the mixed radix case. If we factor n by arranging pairs of identical factors symmetrically about the factors of the square-free portion of n, we can then do one reordering step for each pair of identical factors, using a simple sequence of pair interchanges. After the first step, each result entry is in the correct section of length n/n_1. After the kth step, each result entry is in the correct section of length $n/(n_1 \times n_2 \times \cdots n_k)$. If the square-free portion of n contains at most one factor, the reordering is completed by this procedure. Otherwise a final step is required in which we compute the permutation cycles of the square-free portion of n and use this schedule to complete the reordering.

Scaling for Fixed-Point Arithmetic

The fast Fourier transform can be computed with good accuracy, using fixed-point arithmetic. On a computer without floating-point hardware, the saving in time can be

large. When a FORTRAN subroutine for the SDS 930 computer was translated to a fixed-point machine language program, computing time was reduced by a factor of about ten. In the scaling used, the binary point is assumed to be just to the right of the sign bit, and the initial complex data values and all intermediate results are scaled to be less than $\frac{1}{2}$ in magnitude. A stored table of sine values is used with sin $(\pi/2)$ set at the largest positive value less than one.

Assuming that the complex data points x_j initially have magnitude less than one, we scale by an additional factor two. Thus

$$|x_j| < \tfrac{1}{2} \qquad \text{for } j = 0, 1, \cdots, n-1,$$

$$|x_j \pm x_k \exp(i\theta)| < 1$$

in the computation of each elementary 2×2 transform. If we then scale each intermediate result by $\frac{1}{2}$ before storing, which can be done by shifting right by one place, the values used in computing the second step of the transform are again $< \frac{1}{2}$ in magnitude. Continuing in this way, we scale each result but the final one by $\frac{1}{2}$, giving an overall scale factor of $1/n$ for a transform of $n = 2^m$ data values. The scaling is gradual, and satisfactory accuracy is preserved.

This scaling is easily generalized to the mixed radix case. Assuming that initially

$$|x_j| < 1 \quad \text{for } j = 0, 1, \cdots, n-1$$

we scale the data before each step by $1/n_k$, where n_k is the factor of n used in the kth step of the transform. The overall scaling is again $1/n$.

Conclusion

We have considered several techniques associated with computing the fast Fourier transform. First, we proposed ways of reducing time spent in computing trigonometric function values. One approach is to compute a table of $n/4 + 1$ sine values using an interpolation method. Another approach is to compute the values as needed in the transform calculation, using a difference equation extrapolation method. For the case of transforming the even- and odd-numbered entries of a set of $n = 2^m$ real data points, it is shown that there is an advantage in permuting the data to reverse binary order before the transform rather than computing the transform and then reordering afterward.

In presenting his paper [8] at the 1966 Fall Joint Computer Conference, Gentleman included some discussion of the impracticality of making use of virtual memory to compute the fast Fourier transform in problems exceeding core storage capacity. If we transform with the Sande version of the algorithm and permute in the usual way, his view is correct. However we have shown here that it is possible to rearrange both the transform and permutation to make practical computing large transforms on a virtual memory system.

Finally, we give a scaling for computing the fast transform with fixed-point arithmetic. The fast transform is well suited to fixed-point computing, and this approach should be considered if a significant reduction in computing time can be gained.

Acknowledgments. The author thanks Dr. Samuel Schechter for helpful discussions of methods of computing trigonometric function values and Dr. Harold Stone for his comments and suggestions for revision of this paper.

RECEIVED NOVEMBER, 1966; REVISED JULY, 1967

REFERENCES

1. HAMMING, R. W. *Numerical Methods for Scientists and Engineers.* McGraw Hill Book Company, Inc., New York, 1962.
2. GOERTZEL, G. An algorithm for the evaluation of finite trigonometric series. *Am. Math. Monthly 65* (Jan. 1958), 34–35.
3. COOLEY, J. W., LEWIS, P. A. W., AND WELCH, P. D. Historical notes on the fast Fourier transform. *IEEE Trans. Audio Electroacoustics AU-15*, 2 (June 1967), 76–79.
4. DANIELSON, G. C., AND LANCZOS, C. Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. *J. Franklin Inst. 233* (1942), 365–380; 435–452.
5. RUDNICK, P. Note on the calculation of Fourier series. *Math. Comput. 20*, 95 (July 1966), 429–430.
6. GOOD, I. J. The interaction algorithm and practical Fourier series. *J. Roy. Statist. Soc. Ser. B, 20* (1958), 361–372; Addendum, *22* (1960), 372–375.
7. COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90 (April 1965), 297–301.
8. GENTLEMAN, W. M., AND SANDE, G. Fast Fourier transforms—for fun and profit. Proc. AFIPS 1966 Fall Joint Comput. Conf., Vol. 29, pp. 563–578,
9. SINGLETON, R. C. An ALGOL convolution procedure based on the fast Fourier transform. SRI Project 181531-132, Stanford Res. Inst., Menlo Park, Calif., Jan. 1967, Defense Doc. Ctr. AD-646 628.
10. COOLEY, J. W. Harmonic analysis complex Fourier series. SHARE Program library No. SDA 3425, Feb. 7, 1966.
11. BINGHAM, C., GODFREY, M. D., AND TUKEY, J. W. Modern techniques of power spectral estimation. *IEEE Trans. Audio Electroacoustics AU-15*, 2 (June 1967), 56–66.
12. SINGLETON, R. C. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Trans. Audio Electroacoustics AU-15*, 2 (June 1967) 91–98.
13. SINGLETON, R. C. ALGOL procedures for the fast Fourier transform. SRI Project 181531-132, Stanford Res. Inst., Menlo Park, Calif., Nov. 1966, Defense Doc. Ctr. AD-643 996.
14. SINGLETON, R. C. An ALGOL procedure for the fast Fourier transform with arbitrary factors. SRI Project 181531-132, Stanford Res. Inst., Menlo Park, Calif., Dec. 1966, Defense Doc. Ctr. AD-643 997.