J. W. Hartwell

A Procedure for Implementing the Fast Fourier Transform on Small Computers*

Abstract: A technique has been developed that adapts the Fast Fourier Transform algorithm for implementation on computers having relatively long multiplication times. The technique is particularly well suited to real-time processing on a small data acquisition computer such as the IBM System/7.

Four basic ideas are utilized to improve the performance of the original Cooley-Tukey algorithm on such a machine:

1) The real-valued nature of the input data is exploited.
2) The number of multiplications that must be carried out is minimized at the expense of additions.
3) The calculations are performed in a carefully ordered sequence.
4) Special multiplication algorithms are used.

This technique has reduced by more than an order of magnitude the time required to carry out 1024-point transformations on a small computer. A program is developed for calculating these transforms in real time on an IBM System/7 computer. With this program, a maximum sampling rate in excess of 10 kHz is obtained.

Introduction

The Fast Fourier Transform[1] is a method for efficiently computing the discrete Fourier transform of a sequence of data samples. This technique greatly reduces the number of computations required to calculate such a transform on a digital computer. Consequently, it has made feasible the use of Fourier transforms in the analysis of many problems that were previously approached by other methods. Fourier transforms are now routinely used in such diverse areas as seismic exploration, speech analysis, echo-ranging systems, vibration analysis, image processing, and many others.

Recent years have also witnessed a great increase in the availability of small, relatively inexpensive computers, which are capable of performing most general purpose calculations quite rapidly. In addition, many of them are structured so as to facilitate the acquisition of data from external sources. Because of these capabilities, small computers are now popularly employed in process control and other real-time environments. Such computers may be used to sample incoming signals and to perform certain calculations using these data so that the results become known as the process continues.

In this paper, procedures are developed for implementing fast Fourier transforms on small computers. Special algorithms are devised that adapt this powerful computational technique to the special circumstances presented by these computers. Several ideas for improving the performance of the basic Cooley-Tukey algorithm are presented. When they are combined, an algorithm is obtained which decreases, by more than an order of magnitude, the time required to calculate a Fast Fourier Transform.

After the Fast Fourier Transform and its properties are briefly introduced, each of the five techniques contributing to the improved algorithm is described in detail. The performance of an algorithm for the forward transformation incorporating these techniques is examined in terms of computation time. In addition, a method for performing the inverse transformation is presented. Finally, the results of testing the algorithm are given.

Definition and properties

The discrete (or finite) Fourier transform of a set of N numbers $f(k)$, $k = 0, 1, \cdots, N-1$ is given by a set of N Fourier coefficients $A(n)$, $n = 0, 1, \cdots, N-1$. The coefficients are defined by the relation[2]:

*This paper summarizes the author's Ph.D. dissertation, Department of Electrical Engineering, Duke University, Durham, North Carolina, 1971.

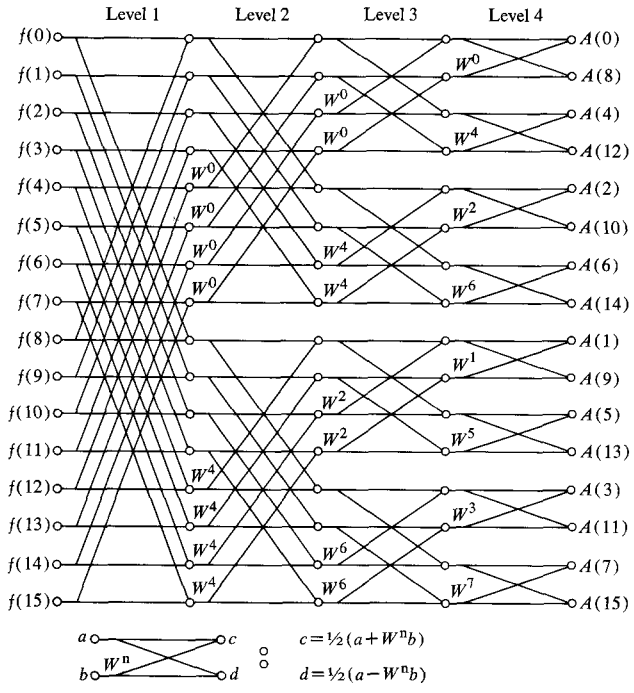

Figure 1 FFT algorithm expressed as a butterfly diagram for $N = 16$.

$$A(n) = 1/N \sum_{k=0}^{N-1} f(k) \exp(-i2\pi nk/N), \tag{1}$$

where $i = (-1)^{\frac{1}{2}}$.

The inverse transform is given by

$$f(k) = \sum_{n=0}^{N-1} A(n) \exp(i2\pi nk/N). \tag{2}$$

An important special case of the transform arises when $f(k)$ is a real-valued sequence. Some of the coefficients may then be obtained directly from others. Replacing n with $N - n$ in Eq. (1) yields the result

$$\begin{aligned} A(N-n) &= 1/N \sum_{k=0}^{N-1} f(k) \exp[-i2\pi(N-n)k/N] \\ &= 1/N \sum_{k=0}^{N-1} f(k) \exp(+i2\pi nk/N) \\ &= \text{conjg}[A(n)]. \end{aligned} \tag{4}$$

The conjugate symmetry expressed here removes the need to calculate the coefficients beyond $N/2$. This result may also be understood from an information theory point of view. Only N real numbers are required to specify the original sequence, so N real numbers also suffice to express the coefficients. $A(0)$ and $A(N/2)$ are real. Each of the intervening $N/2 - 1$ coefficients requires two real numbers to specify its complex value. As a corollary of this result, it is also possible to synthesize a real-valued sequence from half of the coefficients. Using Eq. (4) to eliminate those coefficients in Eq. (2) with subscripts greater than $N/2$, one obtains

$$f(k) = A(0) + (-1)^k A(N/2) + 2 \sum_{n=1}^{N/2-1} \text{Re}[A(n) \exp(i2\pi nk/N)]. \tag{5}$$

• *The Fast Fourier Transform*

The Fast Fourier Transform (FFT) algorithm[1] greatly speeds the calculation of the discrete Fourier coefficients. If the A's are calculated directly from Eq. (1), N^2 operations are required. Here an operation consists of a complex multiplication followed by a complex addition. When N is factorable, however, the A's may be obtained in many fewer operations. The algorithm is particularly simple when N is equal to a radix raised to an integer power. Only the case for N equal to a power of two will be considered here.

When $N = 2^m$, the FFT algorithm can be executed by a repetition of one simple process. The total number of repetitions required to calculate the discrete transform is $mN/2$. For the typical value $N = 1024 = 2^{10}$, this method requires fewer than one percent of the operations needed for a direct application of Eq. (1).

There are several systematic ways[3] to implement the FFT algorithm. They share a repetition of the basic technique on each of m levels, but they differ in the order in which intermediate results are tabulated. One popular implementation is illustrated in the "butterfly diagram" of Fig. 1. The modular nature of the calculations, suggested by the butterflies, is an important property of the FFT algorithm. Each butterfly may be understood as representing one basic operation (i.e., one complex multiplication followed by a complex addition, by a complex subtraction, and by a division by two), where W is the weighting factor $\exp(-i2\pi/N)$. The "levels" of the transform are shown by the vertical columns. This method has the advantage of permitting in-place calculations, i.e., intermediate results can be stored in the same locations on each successive level. Its disadvantage is that the Fourier coefficients as finally computed are not stored in the normal sequence. One additional procedure is required to rearrange them.

The presence of the normalization factor $1/N$ in the forward transformation permits the algorithm to be written with a division by two included in every operation, as shown in Fig. 1. This is advantageous if the calculations are to be done in fixed-point arithmetic. The effect is to constrain the magnitudes of the operands at each successive level so as to form a nonincreasing sequence.

In the inverse transformation, the unwanted normalization factor $1/N$ can be removed by eliminating the division by two that was included in each butterfly. The decision to alter the algorithm in this way, however,

must be based upon evidence that overflow conditions will not result. A sufficient condition is that the transform being inverted consists of unamplified Fourier coefficients derived from a previous forward transformation.

• *Computation time*

When the number of data samples is given by $N = 2^m$, the FFT is calculated by performing $N/2$ butterflies on each of m successive levels of computation. Thus, a total of $mN/2$ complex multiplications, complex additions, and complex subtractions is required.

Each complex multiplication can be computed with four real multiplications and two real additions. Each complex addition or subtraction requires two real additions. Then the entire FFT calculation requires $2mN$ real multiplications and $2mN$ real additions.

Since the time required for multiplication is assumed here to be much greater than that for addition, the total time T required for an N-point transform is approximately

$$T = 2mNt, \tag{6}$$

where t is the time for one real multiplication. This value is a lower bound, because housekeeping, input/output operations, and the reordering of the Fourier coefficients have not been included in the estimate.

Implementation techniques

As shown in Eq. (6), the multiplication time is a significant factor in determining the performance of a small computer in the execution of fast Fourier transforms. In the procedures described here, particular emphasis is placed on efficient multiplication at the expense of increased additions and storage requirements. The first technique employed involves factoring a complex multiply operation to reduce the number of real multiplications by 25 percent. The second technique is based on a characteristic of the algorithm, as depicted in Fig. 1, which allows trivial multiplications to be predicted and thereby eliminated. Two other techniques greatly reduce the multiplication time by constructing special purpose algorithms. Each of these techniques is discussed in the subsequent sections. Finally, a specialization of the basic FFT algorithm to the case of real-valued data is presented. This method is useful in many circumstances because it permits a reduction in storage requirements as well as in execution time.

• *Golub's method for complex multiplication*

Golub's method is mentioned in a footnote by Singleton[4]. Although it is not of much value when the FFT algorithm is coded in a high level language such as FORTRAN, it is an important factor in the present effort. The method factors a complex multiplication into three real multiplications and five real additions. (The conventional evaluation requires four real multiplications and two real additions.) The method is based on the following identity:

$$(a + ib)(c + id) = [(a + b)(c - d) + ad - bc] + i(ad + bc). \tag{7}$$

When this technique is coded directly in machine language, it provides an immediate savings of nearly 25 percent in computation time if the multiplication time is assumed to be much greater than the addition time. Reducing the number of multiplications from four to three far more than offsets the increase in the number of additions under this assumption.

• *Ordering the butterflies*

The following procedure, which groups together operations requiring a common set of multipliers, was suggested to the author by N. M. Brenner (IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y.), who credits Charles Rader (MIT Lincoln Laboratory, 1966) with the basic idea. The idea is not known to have been implemented previously. The additional housekeeping required to use it would normally more than offset its computational advantages. In the present setting, however, it is combined with assembly language programming techniques and with the dynamic compilation procedure described below to produce significant savings in the computation time for an FFT.

In this scheme the butterflies are not executed in the level-by-level sequence implicit in Fig. 1. Instead the operations are performed in an order that depends upon the complex multipliers used. First, all butterflies requiring $W^0 = 1$ are calculated, beginning with those on level 1, then doing those on level 2, and so forth. Next, the butterflies using $W^{N/4} = -i$ are computed. Again they are executed by levels reading from left to right in Fig. 1. The number x of butterflies using W^0 can be found by summing those from each of the levels

$$x_0 = N/2 + N/4 + N/8 + \cdots + 1 = N - 1. \tag{8}$$

In the same way the number of butterflies x requiring $W^{N/4}$ is found to be

$$x_{N/4} = N/4 + N/8 + \cdots + 1 = N/2 - 1. \tag{9}$$

It should be noted that Eqs. (8) and (9) count all of the trivial operations. Those butterflies are executed without actually multiplying. The significance of this result can be appreciated when their sum $[3(N/2) - 2]$ is compared to the total number of butterflies $(mN/2)$. For $N = 1024$, this reduces the number of multiplications by almost 30 percent.

Those butterflies that use certain special multipliers may be executed with only two real multiplications. These

are $W^{N/8} = (\sqrt{2}/2)(1 - i)$ and $W^{3N/8} = (\sqrt{2}/2)(-1 - i)$. The remaining butterflies require all three real multiplications given by Golub's method. The combination of Golub's method and the ordering technique significantly reduces the number of real multiplications required to implement a Fast Fourier Transform. For a 1024-point transform ($m = 10$), the number of multiplications is approximately halved.

In addition to the direct savings in the number of real multiplications, this ordering procedure yields two other benefits. It collects together all the butterflies requiring common sets of multipliers, and it expedites counting the number of times each factor is used. It can be concluded that relatively few multipliers are involved in carrying out most of the operations. This fact is exploited in the multiplication techniques introduced below.

• *Special-purpose multiplication*

A general-purpose software multiplication routine was made available near the beginning of this project for the IBM System/7 computer. Execution time for this routine averages about 65 μsec, and 58 words of storage are used. The routine uses 16-bit (1-word) operands and provides a 2-word product. The special-purpose multiplication resulted from efforts to improve this time.

Substantial savings were found to be possible owing to the special circumstances of the Fast Fourier Transform. First, it was concluded that a single precision result (16 bits) was acceptable. This shortened the general purpose routine by eliminating double shifts. Second, it was decided that the multiplication procedure would not be accessed from more than one interrupt level. This eliminated those steps required for reentrant capability. Third, it was noticed that considerable time and storage were used in the housekeeping necessary to execute the routine as a loop. These steps were deleted by writing the TEST, ADD, and SHIFT statements for each multiplier bit separately. Finally, it was decided to treat every multiplier as a positive number. (The sign of the resultant product must then be established in a separate procedure that takes into account the proper quadrant for the weighting factor).

The effect of these changes is to shorten the multiplication time to 24 μsec and to lengthen the storage requirements to 61 words. This procedure is used in the FFT when the number of times a given multiplier appears is insufficient to justify the dynamic compilation described below. It is also recommended for FFT calculations on computers with very limited storage capabilities.

• *Dynamic compilation*

It has been found possible to carry out fixed-point multiplications very rapidly by specifically tailoring an algorithm to a multiplier. This technique should be valuable in any setting that requires repeated multiplication by a single number. Its advantage in FFT calculations is significant because there are many multiplications with relatively few multipliers. This technique was formulated in consultation with W. D. Modlin (IBM General Systems Division, Boca Raton, Florida). The basic idea is his, and he participated throughout its implementation in the FFT algorithm.

When a multiplier is to be used more than once, certain steps in the ADD-and-SHIFT routine are observed to be redundant. These are the steps that constitute an examination of the multiplier bits. The dynamic compilation technique avoids this redundancy by examining the multiplier only once and compiling a list of instructions for execution specifically tailored to it.

A simplified version of this procedure could be implemented as follows: First, a block of storage in the main program sequence is reserved for installing the tailored multiply algorithm. Second, the machine instruction for adding a number (the multiplicand) to the accumulator, and the machine instruction for shifting the accumulator one binary position to the right, are placed in known storage locations. Third, the multiplier is examined bit by bit, beginning with the least significant bit. The appropriate ADD and SHIFT instructions are installed in the reserved space. If the multiplier bit is a ONE, the ADD instruction is stored, followed by the SHIFT instruction. If the multiplier bit is a ZERO, only the SHIFT instruction is installed. Finally, when each bit has been examined and the appropriate instructions stored, a BRANCH instruction is installed at the end of the sequence. This branch routes the execution around any unused locations and back to the main program sequence. Each operation requiring this multiplier can now be executed without the redundancy of additional steps to examine the multiplier.

The coding for this compilation procedure requires 68 words of storage. It has an average execution time of about 65 μsec. As shown below, however, it results in an average multiplication time of about 4.8 μsec.

Figure 2 is a flow diagram for the dynamic compilation recommended for FFT applications. It differs from the simplified case just discussed in two important respects: When successive shifts are required, they are accomplished by installing a single instruction specifying the appropriate number of shifts. This is easily accomplished through the System/7 instruction set since the number of shifts is given by the final field in the instruction. A look-ahead feature is also provided for successive ONE bits. This feature speeds the execution process by subtracting the multiplicand when a string of ONES is encountered, and by adding one to the multiplier so that the string of ONES is transformed to a string of ZEROS preceded by a ONE. Then an instruction for the appropriate number of

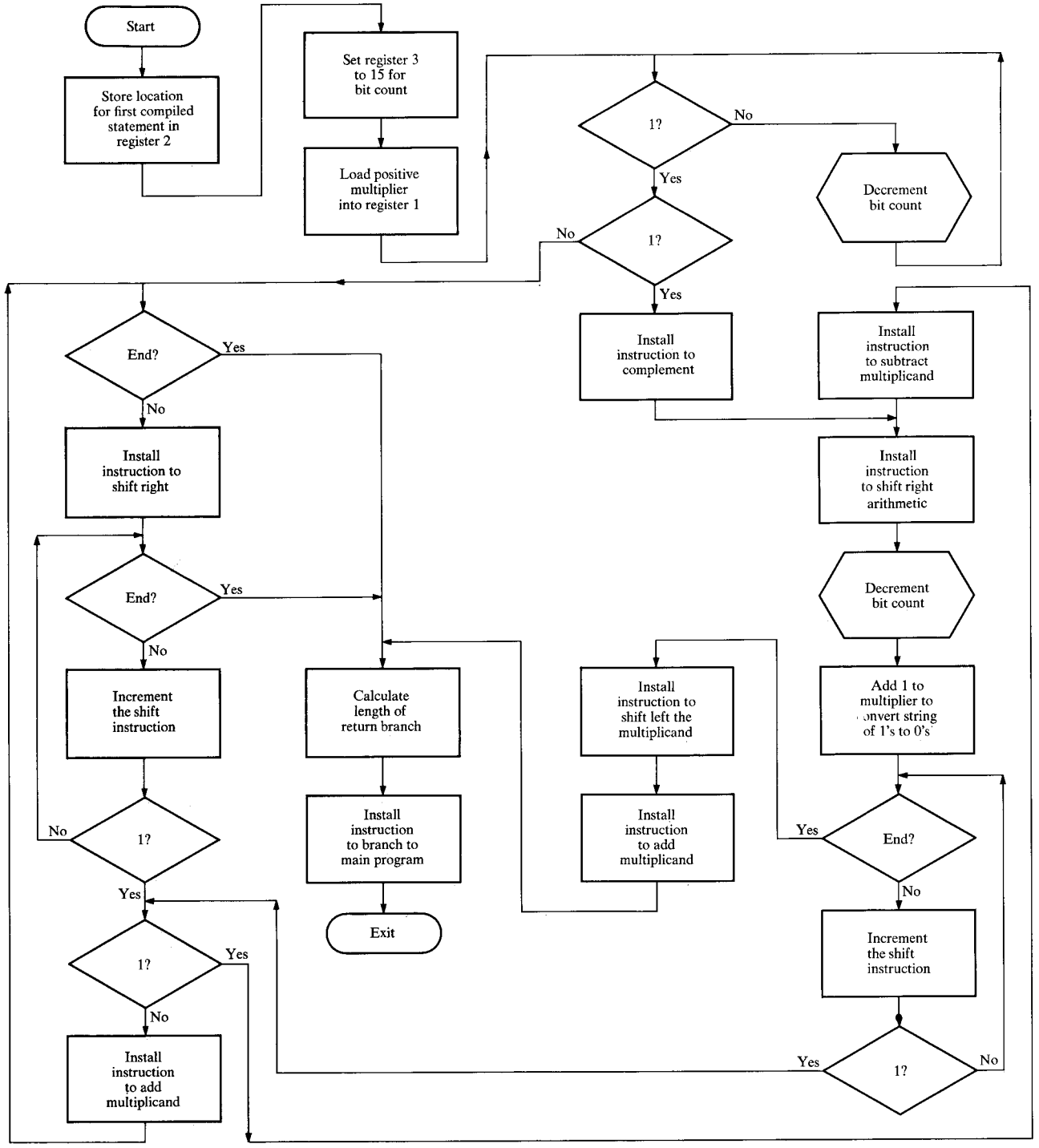

Figure 2 Flow diagram for dynamic compilation procedure.

arithmetic shifts is installed, followed by an ADD-MULTIPLICAND instruction when the next ONE bit is encountered.

The flow diagram of Fig. 2 has been kept as brief as possible by lumping several instructions in the decision block labeled "1?" This block includes the instruction required to shift the next multiplier bit into position after each examination.

Since the multiplier is assumed positive, only 15 of its 16 bits are checked in the procedure shown in Fig. 2 An exception to this can be caused by a string of ONE bits in the most significant positions operated on by the look-

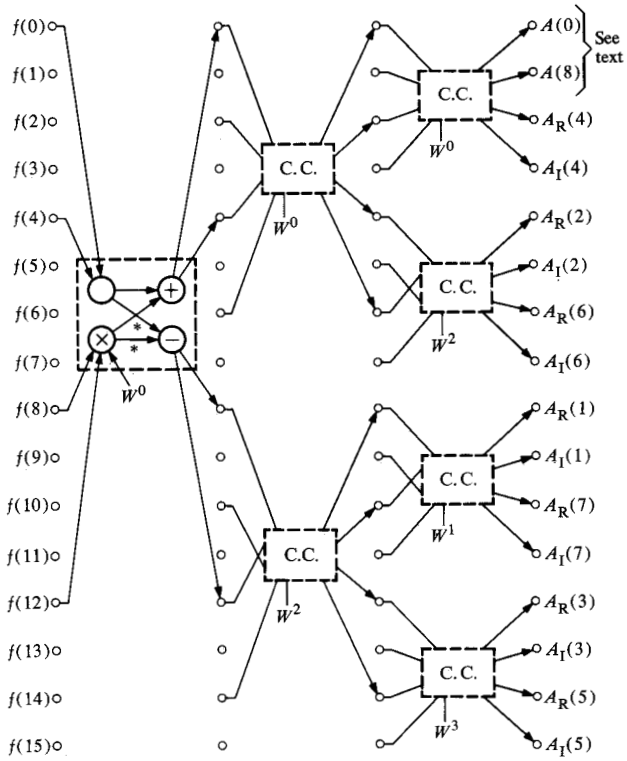

Figure 3 FFT algorithm for real-valued data from Bergland, (Ref. 5).

ahead procedure. This necessitates an additional route to the exit from the right-hand loop. A bit counter is initialized to 15 at the outset and is decremented as the bits are examined. This decrementing is accomplished as part of the decision block labeled "End?" A branch to the EXIT routine is taken when the count has reached zero.

It should be noted that no "End?" block is included in the top loop, which counts low-order ZEROS. This is consistent with the assumption that the multiplier is strictly positive: eventually a ONE bit must be found. This fact also allows the compiled procedure to begin with the multiplicand in the accumulator, since it would be added to an empty accumulator when the first ONE bit was found. In the event, however, that the first ONE bit encountered in the multiplier is the start of a string, then the requisite subtraction is performed by complementing.

The final process blocks in Fig. 2 relate to the branch instruction that must be installed at the end of the compiled procedure. Each time an instruction is placed in the sequence, an index containing this address is incremented. The length of the branch required can be computed by subtracting this address from the address at which execution is to be resumed.

This procedure results in an execution sequence in which the time depends upon the multiplier's specific bit configuration. The maximum execution time on System/7 is 6.8 μsec. The average time required for a multiplication was estimated with the aid of a PL/1 program. The program calculated each of the multipliers needed for a 1024-point transform and expressed them in binary form. A simple procedure was established to count the number of instructions that would be compiled for each of these multipliers. From this information, an average time of 4.8 μsec was deduced.

Storage must be reserved to accommodate the maximum length compilation. Space for 27 words is required.

The decision to use the dynamic compilation technique depends upon the number of times a multiplier is used. The 24-μsec procedure is more advantageous if the multiplier is needed three or fewer times. For multipliers needed four or more times, the 65-μsec dynamic compilation and the resulting 4.8-μsec multiplications are faster.

This method provides greatly improved performance in FFT calculations where some multipliers are used on the order of N times for an N-point transform.

Real-valued data

The butterfly diagram of Fig. 1 shows a method for computing the discrete Fourier transform for the general case of complex-valued data points. A substantial shortening of the procedure is possible when the data are restricted to real values. This case would seem to be of particular interest to the real-time processing contemplated in this paper.

It was shown in Eq. (4) that only half of the coefficients need to be calculated when the data are real-valued. The others can be inferred from the relation $A(N-n) = \text{conjg}[A(n)]$. Bergland[5] has developed an algorithm that exploits this fact. In it the conjugate symmetry of the coefficients for real-valued data is applied on every level. Each level of the butterfly diagram of Fig. 1 may be understood as representing a combination of two transforms into a longer one. Each of these subtransforms has the property of conjugate symmetry. By including the requisite conjugations at each stage, it is possible to eliminate one level of operations. The resulting algorithm is depicted schematically in Fig. 3.

The boxes labeled "C.C." (for complex calculation) are analogous to the butterflies of Fig. 1. The drawing is kept as simple as possible by showing only one box for each complex multiplier on a given level. It is to be understood that this calculation is applied sequentially to all of the operands in its group.

Only the two real-valued coefficients, $A(0)$ and $A(N/2)$, are not correctly computed by the procedure shown in Fig. 3. One final butterfly, a trivial one involving W^0, is

required to complete the calculation. An advantage of this scheme is that the real and imaginary parts of the other coefficients are stored in adjacent locations. These are indicated by the subscripts R and I in the figure.

The benefits gained from the use of the techniques discussed above are shown in Table 1.

Table 1 Reduction in multiplication time.

Technique	*Number of real multiplications*	*Approximate multiplication time for $N = 1024$*
Basic Cooley-Tukey FFT	$2mN$	100% (reference)
Golub's method for complex multiplication	$6mN/4$	75%
Ordering technique to expose simple multipliers	$(4m - 13)N/2 + 10$	68%
Special purpose multiplication	Reduces real multiplication time	35%
Dynamic compilation (where beneficial)	Reduces real multiplication time	17%
Bergland's FFT for real-valued data	$(m - 1)N$	45%
Composite algorithm	$(3m - 10)N/4 + 4$	4%

Real-time FFT for a small computer

In some circumstances it may be advantageous to calculate FFT's in real time, that is, to compute the FFT for one set of samples while the succeeding set is being collected. For these applications, it might be desirable to entrust to a single machine the actual transform computations as well as the data sampling procedures.

Small computers appear to provide the requisite capabilities for this task at moderate cost. They can be equipped with adequate storage which is accessible in times on the order of one microsecond. Unless hardware multiply and divide features are provided, however, these machines suffer the disadvantage of relatively long multiplication times. A typical fixed-point software multiply requires one to two hundred storage cycle intervals, whereas additions or subtractions may customarily be carried out in only a few machine cycles. As was shown in Eq. (6), this long multiplication time severely limits the small computer's ability to perform real-time FFT calculations by conventional methods.

In a real-time application, the computation time must not exceed the time required to collect the N data points for the next transform. Thus, from Eq. (6), the sampling rate S is limited by

$$S < N/T = 1/(2mt). \tag{10}$$

An initial estimate for the performance of the IBM System/7 computer was made using Eq. (10). Taking $m = 10$, and using the general-purpose software multiplication time of 65 μsec, this relation shows the sampling rate limited to about 750 samples per second.

• *Algorithm for the forward transformation*

A scheme has been devised that improves the performance predicted by Eq. (10) by more than an order of magnitude. This increase is achieved by combining all of the techniques heretofore considered. By combining these techniques, an algorithm is obtained that is suitable for real-time processing on a small computer.

The complex calculations required by the algorithm for real-valued data are ordered according to the multipliers used. First, the trivial calculations requiring W^0 are carried out proceeding level by level from left to right. Counting by levels, the number of these boxes is

$$x_0 = N/4 + N/8 + \cdots + 1 = N/2 - 1. \tag{11}$$

Then those boxes involving $W^{N/8} = (\sqrt{2}/2)(1 - i)$ are done in the same sequence. Each of these requires two real multiplications. Counting as before, their number is found to be

$$x_{N/8} = N/4 - 1. \tag{12}$$

Hereafter the complex multipliers are introduced in pairs, $W^{N/4-n}$ accompanying W^n. The elements of each pair are related by the interchange of real and imaginary parts. Three real multipliers, as required by Golub's method, are associated with each pair. In general, the Kth level requires the introduction of 2^{K-3} pairs, each of which will be used in a total of $N/2^K - 1$ operations. The final level $(m - 1)$ calls for $N/16$ new pairs, each used once.

The total number of real multiplications can be found by noting that there are $(m - 1)\,N/4$ complex calculation boxes in all. Each requires three real multiplications, except for the $N/4 - 1$ needing only two, and the $N/2 - 1$ trivial ones:

$$\begin{aligned} x_{\text{real mult}} &= 2(N/4 - 1) + 3[(m - 1)N/4 - 3N/4 + 2] \\ &= (3m - 10)N/4 + 4. \end{aligned} \tag{13}$$

Of this total, some derive from those complex multipliers used only once. They are carried out by the "slow multiply" procedure that does not use dynamic compilation. As previously pointed out, there are $N/16$ such pairs. The number of multiplications to be done this way is

$$x_{\text{slow mult}} = 3 \times 2N/16 = 3N/8. \tag{14}$$

The remaining operations involve multipliers used six or more times. These are accomplished by dynamic

Table 2 System/7 FFT time requirements.

Operation	*No. of times used*	*Execution time in μsec*	*Time required for $N = 2^{10} = 1024$, in sec*
Real addition	$(16m - 39)N/8 + 9$	2	0.031
"Slow multiply"	$3N/8$	24	0.009
"Fast multiply"	$(6m - 23)N/8 + 4$	4.8 (avg)	0.024
Dynamic compilation	$3N/16 - 2$	65 (avg)	0.012
I/O operations	N	5.8	0.006
Housekeeping/ reordering			0.017
Total processing time for 1024-point real-valued transform			0.099

compilation. The number of "fast multiplies" is obtained by subtracting Eq. (14) from Eq. (13):

$$x_{\text{fast mult}} = (6m - 23)N/8 + 4. \tag{15}$$

The number of multipliers for which a dynamic compilation must be carried out can be found by a similar procedure. Only one real multiplier is required for $W^{N/8}$. Thereafter, new multipliers are introduced at the rate of three per pair, and 2^{K-3} pairs for each level from the third through the last. Those introduced on the final level, however, are treated by the "slow multiply" technique. Summing over level three through level $(m - 2)$ gives:

$$x_{\text{compiles}} = 1 + 3(1 + 2 + \cdots + N/32) = 3N/16 - 2. \tag{16}$$

The foregoing procedures greatly shorten the time consumed by multiplication. An accurate estimate of computation time must now include factors previously neglected. The times required for real additions, housekeeping chores, rearranging the Fourier coefficients to their normal order, and input/output operations also must be included in the total. These times are shown in Table 2 for System/7. Other small computers may display significantly different times for the I/O operations.

The total time for processing a 1024-point transform in this manner on System/7 is about 0.099 sec. In a real-time application, this would permit a data sampling rate in excess of 10,300 samples per second. This improves the corresponding figure for the original Cooley-Tukey algorithm by more than an order of magnitude.

• *Inverse transform for real-valued data*

Bergland[5] has suggested an algorithm for inverting transforms of real-valued data that is similar to the forward transformation shown in Fig. 3. It is not possible with this inverse algorithm to utilize the ordering technique. For this reason, its execution time on small general purpose computers is much longer than the corresponding time for the forward transformation.

Another technique has been developed for computing this inverse transformation. The algorithm for the forward transformation is used without alteration. It is, however, both preceded and followed by a simple modification of the data. Let $B(k)$, $k = 0, 1, \cdots, N/2$ represent the discrete Fourier coefficients for the real-valued sequence of samples $g(n)$, $n = 0, 1, \cdots, N - 1$. As was pointed out in Eqs. (4) and (5), $B(0)$ and $B(N/2)$ are real, and each $g(n)$ can be calculated from the formula

$$g(n) = B(0) + (-1)^n B(N/2) + 2 \sum_{k=1}^{N/2-1} \text{Re}[B(k) \exp(i2\pi nk/N)]. \tag{17}$$

The first step in computing this function is to construct a new sequence of real numbers $f(k), k = 0, 1, \cdots, N - 1$ from the Fourier coefficients

$$f(k) = \text{Re}[B(k)] + \text{Im}[B(k)] \qquad k = 0, 1, \cdots, N/2$$
$$f(N - k) = \text{Re}[B(k)] - \text{Im}[B(k)]. \tag{18}$$

In order to see the effect of this change on Eq. (17), it is necessary to write the original coefficients in terms of the new sequence:

$$B(k) = \tfrac{1}{2}[f(k) + f(N - k)] + i\tfrac{1}{2}[f(k) - f(N - k)]. \tag{19}$$

Then these values can be inserted in Eq. (17) to obtain

$$g(n) = \sum_{k=0}^{N-1} f(k) \cos(2\pi nk/N) - f(k) \sin(2\pi nk/N). \tag{20}$$

These two sums can be computed using the forward transformation for real-valued data. If the results of this algorithm are denoted as in Eqs. (1) and (4) by $A(n)$, $n = 0, 1, \cdots, N/2$, then Eq. (20) simply states that the desired results are to be obtained from the relations

$$g(n)/N = \text{Re}[A(n)] + \text{Im}[A(n)] \qquad n = 0, 1, \cdots, N/2$$
$$g(N - n)/N = \text{Re}[A(n)] - \text{Im}[A(n)]. \tag{21}$$

It is to be noted that the data modifications required by Eqs. (18) and (21) are identical. Thus, the inverse transformation can be implemented by executing the same procedure both before and after the forward algorithm is applied. On System/7, these additional steps require approximately 7 msec for a 1024-point transformation. The unwanted normalization factor $1/N$ was discussed above for the more general case of complex data.

Results

A program for the forward transformation was executed on a System/360 Model 25 which emulates System/7.

The input data was restricted to the format used by the medium speed analog-to-digital converter: 14 bits plus sign. The maximum size of the real or imaginary part of any Fourier coefficient is thus also restricted to be less than 2^{14}.

As the program was being debugged, transforms were calculated for data consisting of simple impulse functions. No computational errors were observed. These transforms may be considered as trivial, however, owing to the very large number of ZEROS. To provide a better measure of the accuracy of the algorithm, a ramp function was transformed. The correct results for this input can be calculated in closed form in terms of some trigonometric functions. The maximum error found in any coefficient (real or imaginary part) was ± 2, where fullscale is the maximum permitted amplitude of 2^{14}. Of the 1024 values computed, only seven were in error by this amount. Errors of ± 1 were found for 530 of the numbers computed, while 487 were calculated correctly.

These results may be summarized by stating that the maximum error found was $\pm 0.012\%$ fullscale, with a standard deviation of $\pm 0.004\%$ fullscale. These values should be viewed as only rough estimates of the computational error to be expected for an arbitrary input sequence. No attempt was made to transform random input data. Efforts to obtain a theoretical error analysis were ultimately abandoned owing to the complexities of the program and the twos-complement arithmetic used in System/7.

The algorithms developed in this paper provide a substantial gain in performance when they are implemented on machines having relatively long multiplication times. They are particularly well suited to real-time applications on small computers with high speed data acquisition capabilities. The storage requirements are not excessive, and the accuracy should be acceptable for many purposes.

References

1. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comp.* **19,** 297–301 (1965).
2. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Finite Fourier Transform," *IEEE Trans. A-E,* **AU-17,** 77–85 (1969).
3. G-AE Subcommittee on Measurement Concepts, "What is the Fast Fourier Transform?" *IEEE Trans. A-E,* **AU-15,** 45–55 (1967).
4. R. C. Singleton, "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Trans. A-E,* **AU-17,** 93–103 (1969).
5. G. D. Bergland, "A Fast Fourier Transform Algorithm for Real-Valued Series," *Comm. ACM,* **11,** 703–710 (1968).

Received March 18, 1971

Dr. Hartwell is Assistant Professor of Electrical Engineering at the College of Engineering, Florida Atlantic University, Boca Raton, Fla.