Fourier transform and convolution subroutines for the IBM 3090 Vector Facility

by Ramesh C. Agarwal
James W. Cooley

A set of highly optimized subroutines for digital signal processing has been included in the Engineering and Scientific Subroutine Library (ESSL) for the IBM 3090 Vector Facility. These include FORTRAN-callable subroutines for Fourier transforms, convolution, and correlation. The subroutines are carefully designed and tuned for optimal vector and cache performance. Speedups of up to 9½ times over scalar performance on the 3090 have been obtained.

1. Introduction

In 1977, Carl H. Savit, Vice President for Data Processing of Western Geophysical Company of America, estimated [1] that by 1985, seismic exploration for oil and gas would require an increase in computer memory and speed by a factor of 3×10^6. Furthermore, he stated that the rate of increase in computer power is much lower than the rate of increase in computer requirements for seismic exploration alone. The crossover point was placed by him at about 1972.

Much of the computation in seismic exploration is referred to as "digital signal processing" and is characterized by the fact that the data originate as analog signals which are converted into a digital representation. As far as the computer is concerned, the important characteristics of this class of problems are that the amount of data is huge and must be processed economically within a certain limited amount of time, and that the mathematical and computational processes are dominated by the calculation of Fourier transforms, correlations, convolutions, and spectral analysis. Other areas of application of these calculations which share many of these characteristics include geophysical research, crystallography, vibration analysis, radar and sonar signal processing, communications, speech recognition and synthesis, and the processing of weather data for analysis and prediction.

If the figures cited above are compared with the rate of increase in the raw speed of computer hardware, it is inevitable that intensive work must be done in producing parallel and vector processing capabilities along with algorithmic procedures for using them efficiently. The response to these demands has been that a large number of parallel and vector processors have come upon the market and are in use today. However, these fall far short of filling the requirements that Savit described. Experience with these machines and the demands put upon them has given rise to many new problems in algorithm design and program generation and in the design and use of compilers and other programming tools. Typically, it is found that the problem of fully utilizing the facilities of a parallel or vector processor is many orders of magnitude greater than for a serial machine. This means that the difference between optimized

©**Copyright** 1986 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

hand-coded programs and programs generated by compilers or mediocre programming is far greater than it is in serial machines.

In response to this situation, the set of "signal processing" programs included in the Engineering and Scientific Subroutine Library (ESSL) package [2, 3] contain very efficient hand-coded inner subroutines which do most of the computing. The innermost subroutines, forming the core of all of the Fourier transform and the large convolution and correlation calculations, use the methods of Agarwal [4] to do all calculations with full vector registers and to maintain high cache performance. Programs which do the "controlling" and take up insignificant amounts of time are written in FORTRAN. The program logic is designed and finely tuned to yield the best vector and cache performance. To the user, these form an integrated package of FORTRAN-callable subroutines. It is expected that in the typical applications mentioned above, one will be able to write FORTRAN calling programs in such a way that the dominant portions of calculations will be done by the optimized vector subroutines.

For a general text on digital signal processing, the reader is referred to Oppenheim and Schafer [5]. For general references on the fast Fourier transform (FFT) algorithm, see [6–11]. For applications of the FFT algorithm, see [7, 12–14]. Since there was very little written on the finite discrete Fourier transform before the advent of digital computers and the FFT algorithm, [10] was written and is referred to here. Reference [13] describes the revisions of traditional power spectrum estimation methods which came about as a result of the advent of digital computers and the FFT algorithm.

2. Contents of the signal processing package

The signal processing package of ESSL consists of Fourier transform subprograms and convolution and correlation subprograms. The Fourier transform subprograms discussed are

- SCFT: Single-precision complex Fourier transform.
- DCFT: Double-precision complex Fourier transform.
- SRCFT: Single-precision real-to-complex Fourier transform.
- SCRFT: Single-precision complex-to-real Fourier transform.
- SRCFT2: Single-precision real-to-complex Fourier transform in two dimensions.
- SCRFT2: Single-precision complex-to-real Fourier transform in two dimensions.

The convolution and correlation subprograms discussed are

- SCON: Single-precision convolution of one sequence with several sequences.
- SCOR: Single-precision correlation of one sequence with many sequences.
- SACOR: Single-precision autocorrelation of several sequences.

- *Fourier transforms—General discussion*

The N-point discrete Fourier transform (DFT) of a vector, $x(n)$, $n = 0, \cdots, N-1$, is the vector defined by

$$y(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad k = 0, \cdots, N-1. \tag{1}$$

Here and in what follows,

$$W_N = e^{-2\pi i/N}, \tag{2}$$

in which we define $i = \sqrt{-1}$. The inverse transform, which gives back $x(n)$ as a function of $y(k)$, is

$$x(n) = (1/N) \sum_{k=0}^{N-1} y(k) W_N^{-nk}. \tag{3}$$

References [7–10, 12, 13] give important relationships satisfied by the DFT pair which are used in what follows.

- *Fourier transform subprograms—Detailed description*

SCFT: Complex single-precision discrete Fourier transform and DCFT: Complex double-precision discrete Fourier transform

The subprograms SCFT and DCFT compute the discrete Fourier transforms, in single and double precision, respectively, of a set of M sequences. For a given complex input sequence, $x(n)$, $n = 0, \cdots, N-1$, the subprograms compute the complex sequence defined by

$$y(k) = SCALE \times \sum_{n=0}^{N-1} x(n) W_N^{ISIGNnk} \qquad k = 0, \cdots, N-1, \tag{4}$$

where $SCALE$ and $ISIGN = \pm 1$ are arguments to the subroutine.

The internal components of the SCFT subprogram are used in all of the Fourier transform programs.

SRCFT: Real-to-complex single-precision discrete Fourier transform

The subprogram SRCFT computes the complex discrete Fourier transforms, in single precision, of a set of M real sequences. For a given real single-precision input sequence, $x(n)$, $n = 0, \cdots, N-1$, the subprograms compute the complex sequence defined by (4). Since the input is real, the output will be complex conjugate even, meaning that

$$y(k) = y^*(N-k), \tag{5}$$

where * denotes the complex conjugate. Therefore, results are given only for $k = 0, \cdots, N/2$.

SCRFT: Complex-to-real single-precision discrete Fourier transform

The subprogram SCRFT computes the real discrete Fourier transforms, in single precision, of a set of M complex sequences. For a given complex conjugate even single-precision input sequence, $x(n)$, $n = 0, \cdots, N-1$, the subprograms compute the real sequence defined by (4). Since the input is assumed to be complex conjugate even, meaning that

$$x(n) = x^*(N - n), \tag{6}$$

the output will be real. Therefore, the input is used only for $n = 0, \cdots, N/2$. With input parameters $SCALE = 1/N$, $ISIGN = -1$, and with $x(n)$ a function of frequency, one obtains what is generally referred to as the inverse Fourier transform, usually a function of time or distance. If the output of subprogram SRCFT described above is supplied as input to SCRFT with the parameters cited, the original input to SRCFT will be given as the output of SCRFT.

SRCFT2: Real-to-complex single-precision discrete Fourier transform in two dimensions

The subprogram SRCFT2 computes the two-dimensional discrete Fourier transform, in single precision, of an N_1 by N_2 array. For a given real single-precision input array, $x(n_1, n_2)$, $n_1 = 0, \cdots, N_1 - 1$, $n_2 = 0, \cdots, N_2 - 1$, the subprograms compute the complex array defined by

$$y(k_1, k_2) = SCALE \times \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_1}^{ISIGNk_1n_1} W_{N_2}^{ISIGNk_2n_2}, \tag{7}$$

where $k_1 = 0, \cdots, N_1 - 1$, and $k_2 = 0, \cdots, N_2 - 1$. With input parameters $SCALE = 1$ and $ISIGN = +1$, one obtains what is generally referred to as the two-dimensional Fourier transform, a function of frequency. Since the input is real, the output will be complex conjugate even, meaning that

$$y(k_1, k_2) = y^*(N_1 - k_1, N_2 - k_2). \tag{8}$$

Therefore, results are given for all k_2 but only for $k_1 = 0, \cdots, N_1/2$.

SCRFT2: Complex-to-real single-precision discrete Fourier transform in two dimensions

The subprogram SCRFT2 computes the two-dimensional discrete Fourier transform, in single precision, of an N_1 by N_2 complex conjugate array. For a given complex conjugate even single-precision input array, $x(n_1, n_2)$, $n_1 = 0, \cdots, N_1 - 1$, $n_2 = 0, \cdots, N_2 - 1$, the subprograms compute the real sequence defined by (7). Since the input is assumed complex conjugate even, meaning that

$$x(n_1, n_2) = x^*(N_1 - n_1, N_2 - n_2), \tag{9}$$

the input is used for all n_2, but only for $n_1 = 0, \cdots, N_1/2$. With input parameters $SCALE = 1/(N_1 \times N_2)$, $ISIGN = -1$, and with $x(n_1, n_2)$ a function of frequency, one obtains what is generally referred to as the inverse Fourier transform, usually a function of distance. If the output of subprogram SRCFT2, described above, is supplied as input to SCRFT2 with the parameters cited, the original input to SRCFT2 will be given as the output of SCRFT2.

• *Convolution and correlation subprograms—General discussion*

The relation between convolution and correlation integrals and their discrete representation and the use of the convolution theorem for Fourier transforms is described in Ref. [8]. In what follows, discrete, finite convolutions and correlations are considered.

The convolution of a sequence $h(j)$, $j = 0, 1, \cdots, N_h - 1$ with another sequence $x(j)$, $j = 0, 1, \cdots, N_x - 1$ is defined by

$$y(n) = C_{hx}(n) = \sum_{j=\max(0,n-N_x+1)}^{\min(n,N_h-1)} h(j)x(n - j). \tag{10}$$

Defining the limits of summation is equivalent to saying that the sum is over all j and that the data outside the interval of definition are zero. The range of indices of possibly nonzero values of $y(k)$ is $k = 0, \cdots, N_h + N_x - 2$. From the definition, one can easily show the symmetry

$$C_{hx}(n) = C_{xh}(n). \tag{11}$$

Under the same assumptions as above, the crosscorrelation of two sequences $h(i)$ and $x(i)$ is defined by

$$y(n) = \overline{C}_{hx}(n) = \sum_{j=\max(0,-n)}^{\min(N_h-1,N_x-1)} h(j)x(n + j). \tag{12}$$

As in the previous paragraphs, defining the limits of summation is equivalent to saying that the sum is over all j and that the data outside the interval of definition are zero. The possible nonzero values of $\overline{C}_{hx}(n)$ are for $n = -N_h + 1, \cdots, N_x - 1$. Calling sequences will give the lengths of the input and output sequences. If $h(n)$ and $x(n)$ are the same, (12) is known as the *autocorrelation* function. The symmetry condition is

$$\overline{C}_{hx}(n) = \overline{C}_{xh}(-n). \tag{13}$$

Therefore, the crosscorrelation programs can give results for $n < 0$.

An important class of problems consists in solving a set of normal equations

$$\sum_i a_i \overline{C}_{xx}(n - i) = \overline{C}_{hx}(n). \tag{14}$$

In Linear Predictive Coding (LPC) models of speech [15], $h(i)$ and $x(i)$ are the same; in many system identification problems, they are different. The Levinson algorithm (see, for example, [15]) is often used to solve for the a_i's in Eq. (14).

SCON: Convolution of one h with many x's
This subroutine computes the convolutions

$$y(k, m) = C_{hx}(k, m) \tag{15}$$

of one vector $h(n)$ with many vectors $x(n, m)$, with all vector elements spaced in memory with given strides. (The term *stride* indicates the distance in storage between elements of a sequence.)

SCOR: Correlation of one h with many x's
This subroutine computes the correlations

$$y(k, m) = \overline{C}_{hx}(k, m) \tag{15'}$$

of one vector $h(n)$ with many vectors $x(n, m)$, with all vector elements spaced in memory with given strides.

SACOR: Autocorrelations of many sequences
This subroutine computes the autocorrelations

$$y(k, m) = \overline{C}_{xx}(k, m) \tag{16}$$

of a set of vectors $x(n, m)$, $m = 1, \cdots, M$.

3. FFT computing using the 3090 Vector Facility

One of the approaches which could have been taken was to use a state-of-the-art scalar FFT program and vectorize it using the vectorizing compiler. A preliminary analysis indicated that such an exercise would result in programs giving far less than the peak expected performance. It became clear that to fully exploit the Vector Facility, a restructuring of the FFT algorithm was necessary so that the algorithm would match the architecture. In this section, we give justifications for this decision.

The 3090 Vector Facility has a vector length or vector section size (*VSS*) of 128. Most vector instructions such as load, store, multiply, add, multiply-add, etc. produce one result every cycle, after the initial start-up delay, which could be as much as 30 cycles or more. Thus, to efficiently utilize the Vector Facility we must work with long vector lengths, preferably with the full vector length (128). Therefore, the FFT algorithm needs to be reformulated such that we work with a vector length of 128 as far as possible.

The above assumption of one result every cycle is valid only if all the operands are in cache, which is of size 64K bytes for this machine. If any of the operands are not in cache, data must be fetched from the main memory in units of 128 bytes (a line of cache). This introduces a sizable further delay. The cost of bringing a line into cache is fixed. Therefore, the algorithm must be restructured such that if a line (128 bytes = 8 double words = 16 single words) is brought into cache, all of it gets utilized. Cache has a structure such that only four lines, with identical seven low-order address bits, can reside there at the same time. If one more line, with the same seven low-order address bits, is needed, one of the previous lines has to be cast out. This implies that it is very difficult to bring an array into cache with a power of 2 stride. The conventional FFT algorithm requires strides which are powers of 2 in the innermost loop. Therefore, a restructuring of the FFT algorithm is necessary to avoid this problem.

The cache considerations are much more important for the Vector Facility *vis à vis* the scalar processor. In the scalar processor, the arithmetic is slow, and therefore the cost of bringing the data into cache is a small fraction of the total cost. But this is *not* the case for the Vector Facility, where the arithmetic unit is very fast and the cost of data transfers between main memory and cache becomes *very* important. To reduce this data traffic, we have to restructure the algorithm in such a way that we break a large problem into many subproblems, where each subproblem fits in cache. Most of our effort in this project has been directed towards these two problems:

1. How to achieve full vector length for all phases of the computation.
2. How to work around the cache.

We have restructured the FFT algorithm with these two goals in mind, and this has resulted in many different cases depending on the problem size.

We found some of the architectural features of the machine to be very useful in this regard:

- We used a long-precision load/store to load/store two short-precision vectors.
- The multiply-add instruction performs two floating-point operations per cycle. We have used this instruction wherever possible; sometimes this has meant a slight restructuring of the algorithm.
- The vector processor has 16 short-precision vector registers, and we have attempted to utilize all of them in such a way that we do as much computing as possible in registers between load and store operations. Thus, we have tried to reduce the ratio of load/store operations to arithmetic operations.
- Making use of the three-operand architecture of the machine, we avoid loads as far as possible. Now we try to pick up one of the operands from memory as part of the arithmetic instruction. Here is where (2) comes into play, as we have structured the algorithm to reference these operands repeatedly, implying that they will become cache-resident. In effect, cache becomes an extended memory of the vector registers.

4. Methodology in Fourier transforms

The radix-2 FFT algorithm for length $N = 2^m$ consists of m radix-2 stages. At each stage, $N/2$ radix-2 butterflies [5, p. 296] are computed. Thus, it appears to be vectorizable, with a vector length of $N/2$. But, unfortunately, data indexing for these butterflies is not uniform, thereby requiring indirect addressing of data, which on the 3090

Vector Facility is somewhat slow. Scalar FFT algorithms do "in-place" computation, with each radix-2 stage broken into two nested DO-LOOPS, with uniform indexing within these. Depending on the stage of the FFT computation, the inner DO-LOOP length varies from $N/2$ to 1, with an average length of only $O(\log N)$. Thus a straightforward vectorization of a scalar FFT code would give an average vector length of only $O(\log N)$, which is too small for efficient implementation on a vector architecture.

Pease [16] had developed an FFT algorithm for a parallel machine. Korn and Lambiotte [17] adopted the Pease algorithm for implementation on the STAR-100 vector computer. Their algorithm works with vectors of length $N/2$, which is the longest possible vector, but it requires special vector operations generally referred to as GATHER or COMPRESS performed under a bit-vector control. By using a bit-vector of identical length, the data vector is compressed, selecting only those elements which have a one bit in the bit-vector. For the FFT, two GATHER instructions are required to form one vector. This is repeated for every stage of the FFT computation. The COMPRESS instruction is available on the 3090 Vector Facility, but an FFT implementation using it would be inefficient because it doubles the number of vector store instructions.

The CRAY-1 computer does not have a COMPRESS instruction; therefore, Swarztrauber [18] and Petersen [19] adopted a variation of the scalar algorithm which increases the average vector length from $O(\log N)$ to $O[(\sqrt{N}/4) \log N]$. This is a considerable improvement but it is still not the best possible.

In [4], a mixed-radix FFT algorithm is presented which is fully vectorized and requires all loads/stores with only a small stride, for all intermediate FFT stages. Indirect addressing is used only for the initial "index-reversal." For the special case of the radix-2 FFT, the vector length is always $N/2$. In addition to the basic arithmetic operations, the radix-2 stages require load/store operations with a stride of 1 or 2. For a radix-r stage, the following steps are performed with a vector length of N/r:

- Load r vector registers with stride r.
- Do $r - 1$ complex vector multiplications using the $r - 1$ pre-computed (pre-permuted) twiddle-factor vectors (twiddle factor = multiplication by a complex number to accomplish phase shift).
- Compute the radix-r vector DFT.
- Store r vectors with stride one.

The algorithm works "in place"; i.e., the output vectors replace the input vectors in storage.

- *Case A*

For the 3090 Vector Facility, considering the number of vector registers available (16 in short precision or, equivalently, 8 in long precision), the radix-4 seems to be the best radix for the vector FFT implementation. Depending on the ratio of N and VSS, the FFT computation is done by one of the four routines. First, we describe Case A, which is for $N \leq 4 \times VSS$. For this case, the vector length of the machine is sufficient to accommodate all the data in vector registers. For shorter transforms, the possibility of simultaneously computing several transforms exists. In this implementation, the first stage is always a radix-4 stage implemented by the F4F$ routine. All the intermediate stages are also radix-4 stages implemented by the F4I$ routine. The last stage could be either a radix-4 stage (if N is an even power of 2) implemented by the F4L$ routine, or a radix-2 stage (if N is an odd power of 2) implemented by the F2L$ routine.

We consider the case where several transforms are to be computed simultaneously. In computing m transforms simultaneously, for a radix-4 stage, the required vector length is $m \times N/4$. Therefore, in a vector block $m = 4 \times VSS/N$ transforms could be computed simultaneously. This utilizes the full vector length of the machine which, as discussed earlier, is most efficient. This is the first level of segmentation. Additional implementation efficiency is achieved by processing several vector blocks simultaneously. For each FFT stage, the twiddle-factor vectors are the same for all the vector blocks and therefore are loaded outside the loop on vector blocks. They remain in vector registers throughout the loop. Thus the cost of loading the twiddle-factor vectors is shared by many vector blocks, leading to improved efficiency. The cache size limits the number of vector blocks that are processed simultaneously. The aim is to make sure that all of the data work area (the number of vector blocks being processed) and the vector twiddle factors for all the stages remain in cache with room to spare for input and output arrays. This decides the number of vector blocks which can be most efficiently processed together. This collection of vector blocks is called a cache block.

- *Case B*

A radix-r step of the FFT algorithm requires working with r vector registers of length N/r each; when N/r is longer than the vector length of the machine, the algorithm of [4] creates many problems. In that situation, we cannot do in-place computation, and therefore two work arrays of size N each are required for data. It also leads to a large storage requirement for twiddle-factor arrays, which are separately pre-permuted for each FFT stage. Thus, even for moderate-size transforms, a straightforward application of ideas of [4] may lead to a storage requirement which may be larger than the cache size of the machine. This would create cache misses and thereby degrade performance. This leads us to Case B, which is for transform lengths of size $8 \times VSS$ and $16 \times VSS$. The algorithm of [4] is restructured so that all

FFT stages, except the last radix-4 stage, require working with vectors of length only VSS, and thus require twiddle-factor vectors also of size VSS only. This leads to a significant reduction in the storage space for twiddle-factor vectors. Except for the first radix-4 stage, all other stages do in-place computing; i.e., the output array overwrites the input array. This also reduces the working-array size, which has to be kept in cache.

- *Cases C and D*

Yet another problem arises when the transform length N is so large that the working array and the associated twiddle-factor arrays do not fit in cache. In that situation, the cache-miss ratio increases significantly, leading to a significant degradation in performance. This brings us to Case C, where we do yet another reformulation of the FFT algorithm in which the data are blocked in cache blocks which fit in cache. The algorithm is restructured such that a fairly large amount of processing is done on a cache block before the next one is processed. This keeps data in cache for longer periods, reducing the cache miss ratio.

We reformulate a one-dimensional transform of length N as a two-dimensional transform of size N_1 by N_2 with a twiddle-factor multiplication stage between, where $N = N_1N_2$. By using this technique, the computation of the length-N DFT can be done in the following four steps:

1. Performing N_2 row transforms of length N_1.
2. Twiddle-factor multiplication by powers of W_N.
3. Performing N_1 column transforms of length N_2.
4. A final two-dimensional array transposition.

For the present ESSL design, considering the cache size (64K bytes) and the vector section size of the 3090 Vector Facility ($VSS = 128$), N_1 was chosen as 32 for the short-precision routine (SCFT) and 16 for the long-precision routine (DCFT). Here we do not give the implementation details except to point out some of the salient features. For both the row and the column transform computations, all the vectors are formed with stride one. An auxiliary array $AUX2$, of size slightly larger than N, is used for processing. All the processing takes place in the $AUX2$ array, which can be thought of as a two-dimensional array of size $(N_2 + 16)$ by N_1 (for the short-precision routine). The last 16 rows of the $AUX2$ array are not used, but are provided to improve the cache performance. In computing the length-N_1 row transforms (Step 1), the vectorization is done along the N_2 dimension. Therefore, during this step, the vectors are formed with stride one, and the required twiddle factors are only scalars. All these things help in achieving better cache performance. Step 2 is incorporated with the last radix-4 stage of Step 1. This eliminates loads/stores for Step 2. Column transforms of length N_2 are computed using the technique described earlier in Cases A and B. These, being column transforms, are also computed with stride one. In this approach, the only additional cost is in the two-dimensional array transposition (Step 4), which is also implemented by blocking the data in cache blocks. This more than pays for itself by improving the overall cache performance.

The above technique extends efficient FFT computation by a factor of N_1, with the full level of vectorization and efficient cache management. The technique can be used recursively (Case D) to compute even longer transforms. In ESSL, we have used it twice (Cases C and D), resulting in the maximum transform length of $16 \times VSS \times N_1 \times N_2$, which for the present design is 2097152 for SCFT and 524288 for DCFT.

- *Summary of the one-dimensional complex Fourier transform*

The main computing kernels for all short-precision Fourier transform routines are the complex-to-complex FFT routines described above. There are four cases, depending on the transform length N:

- Case A: $8 \le N \le 4 \times VSS$.
- Case B: $8 \times VSS \le N \le 16 \times VSS$.
- Case C: $32 \times VSS \le N \le 512 \times VSS$.
- Case D: $1024 \times VSS \le N \le 16384 \times VSS$.

- *Real-to-complex and complex-to-real transforms*

For the complex-to-real Fourier transform routine (SCRFT), we first do a special radix-2 FFT routine. This is followed by a length-$N/2$ complex-to-complex Fourier transform, using the above kernels. This results in a complex sequence of length $N/2$, which, when interpreted as an equivalent real sequence of length N, is the desired output. Similarly, for the real-to-complex Fourier transform routine (SRCFT), the above two steps are done in the reverse order. First, a length-$N/2$ complex-to-complex Fourier transform is computed, using the above kernels. For this purpose, we treat the given input real sequence of length N as an equivalent complex sequence of length $N/2$. This is followed by a special radix-2 FFT routine, which gives the final result as $N/2 + 1$ complex values, of which the first and the last values have zero imaginary parts. Because of this structure, the maximum transform length for SRCFT and SCRFT is twice that for SCFT.

- *Two-dimensional transforms*

For the two-dimensional real-to-complex Fourier transform (SRCFT2) of dimension N_1 by N_2, N_2 column transforms of length N_1 each are first computed as real-to-complex transforms, as in SRCFT above. This results in a complex array of size $(N_1/2 + 1)$ by N_2. Next, we compute $(N_1/2 + 1)$ complex-to-complex Fourier transforms of length N_2 each, using the above kernels. For the two-dimensional complex-to-real Fourier transform (SCRFT2), the above two steps are reversed.

5. Methodology in convolutions and correlations

• *Direct methods*

The subroutines SCOR, SCON, and SACOR compute correlations, convolutions, and autocorrelations, respectively, by essentially the same methods. When one of the convolved sequences or the output sequence is short, one of several direct methods is used. These use vector operations to accumulate sums of products. The products are computed and accumulated in double precision, meaning that fairly high accuracy is obtained in the final results. If input data consist only of integers and if no numbers become too large (larger than $2^{24} - 1$), the results will be exact.

• *Fourier methods when all sequences are long*

When all sequences are long, Fourier methods are used. This means that Fourier transforms of input data are computed and multiplied element-by-element in single-precision arithmetic. The inverse Fourier transform is then computed. There are internally generated rounding errors in the Fourier transforms. It has been shown [20] that in the case of white noise data, the relative RMS (root-mean-square) error is proportional to $\log_2 N$ with a very small proportionality factor. Therefore, while direct methods with integer data are exact, the Fourier methods are not. In particular, when the input data consist of integers, the results may be close but not equal to the correct integer results. However, the generated relative error is not great enough to cause difficulties in the type of calculations for which these subroutines will be used. In fact, these results show that the RMS error for $N = 1000$ is only a few units in the last position, which is less than the error which may be expected to result from rounding of the input data.

6. Performance measurements with Fourier transforms

We ran all ESSL Fourier transform routines for various transform lengths. These routines have an initialization phase, which needs to be done only once for a set of parameters. During initialization, we set up all the twiddle-factor arrays, permutation indices, etc. In many applications, one typically initializes the routine once and then it is called many times for the actual FFT computations. Therefore, our aim has been to minimize the run time. For short-length transforms, as mentioned in Section 4, significantly better performance is achieved by computing several transforms simultaneously. In this context, we have the concept of the vector block (implies that the number of transforms is such that all the computing is done with the full vector length of the machine) and the cache block (implies that the number of transforms fill up the cache block). The full cache size of the machine is 64K bytes (8K double words), but only a part of it is utilized for data; the rest of it is used to store the twiddle-factor vectors, permutation indices, other constants, and the programs. Thus, depending on the routine, the actual cache-block size is a tuned parameter in the range 2K to 4K double words. Compared to a single transform computation, better performance is achieved at the vector-block level, and the best performance is achieved at the cache-block level. We have run the one-dimensional Fourier transform routines for one transform, for the number of transforms needed to fill a vector block, and for the number of transforms needed to fill a cache block. Below, we give detailed performance results for various FFT routines. All timings are the virtual CPU timings in microseconds. Depending on the actual computing environment, the timings may vary to some extent.

• *SCFT*

Table 1 gives SCFT performance on the 3090 Vector Facility for various values of N and M. For short-length transforms, we have performance numbers for $M = 1$, $M = 512/N$ (vector block), and $M = 4096/N$ (cache block). Columns 6 and 7 give normalized run time and total time (including the initialization time), respectively. These are normalized by $M \times N \times \log_2 (N)$, to reflect the relative performance level. For the applications requiring repeated calls to SCFT, Column 6 numbers are meaningful numbers (as the initialization is done only once) for comparison with other programs. The initialization time is not significant for long transforms or when many transforms are being computed simultaneously. As mentioned before, for short-length transforms, the best performance is achieved for a full cache block. There is no particular advantage to be gained by using a larger M value. And for the same reason, for longer transforms ($N > 2048$) no advantage is to be gained by using $M > 1$. Examination of Columns 6 and 7 reveals a slight jump in times at $N = 4096$ and 131072, which reflects the additional computing cost for the array transposition required in Cases C and D, as discussed in Section 4.

We have compared SCFT against two scalar FFT routines which are considered to be among the best available. **Table 2** gives performance numbers (similar to Columns 6 and 7 of Table 1), for Singleton's mixed-radix FFT routine [21], and for Bergland's radix 8-4-2 FFT routine [22]. Note that Bergland's routine works only for N up to 32768. For these scalar FFT programs, there is no advantage to be gained by computing several transforms. These also do not have an initialization phase. A comparison of Columns 2 and 3 reveals that Bergland's program runs faster, for N up to 16384, on the 3090. Also note the effect of cache for $N > 8192$. In the same table, we compare these against SCFT performance. In Column 4, we give low ratio, which is the ratio of the worst vector performance (values from Column 7 of Table 1, for $M = 1$, which include the initialization time) to the best scalar performance (better of Columns 2 and 3). In Column 5, we give the high ratio,

Table 1 Performance of the SCFT routine.

Transform length N	*No. of transforms M*	*Initialization time (μs)*	*Run time (μs)*	*Total time (μs)*	*Run time* / $M \times N \times \log_2(N)$	*Total time* / $M \times N \times \log_2(N)$
64	1	169	136	305	0.354	0.794
64	8	353	373	726	0.121	0.236
64	64	382	2329	2711	0.095	0.110
128	1	227	198	425	0.221	0.474
128	4	385	449	834	0.125	0.233
128	32	416	2841	3257	0.099	0.114
256	1	288	296	584	0.145	0.285
256	2	370	474	844	0.116	0.206
256	16	421	3104	3525	0.095	0.108
512	1	528	549	1077	0.119	0.234
512	8	532	3608	4140	0.098	0.112
1024	1	411	1084	1495	0.106	0.146
1024	4	505	4054	4559	0.099	0.111
2048	1	634	2357	2991	0.105	0.133
2048	2	704	4678	5382	0.104	0.119
4096	1	1360	5746	7106	0.117	0.145
8192	1	2225	12122	14347	0.114	0.135
16384	1	4049	25973	30022	0.113	0.131
32768	1	7264	54565	61829	0.111	0.126
65536	1	14187	120187	134374	0.115	0.128
131072	1	28417	271681	300098	0.122	0.135
262144	1	56637	575150	631787	0.122	0.134
524288	1	114969	1203786	1318755	0.121	0.132
1048576	1	224112	2487562	2711674	0.119	0.129

Table 2 Performance for "best" scalar FFT and comparison against SCFT.

Transform length N	*Singleton time* / $N \times \log_2(N)$	*Bergland time* / $N \times \log_2(N)$	*Vector-scalar comparison*	
			Low ratio	*High ratio*
64	1.002	0.914	1.15	9.62
128	0.941	0.770	1.62	7.77
256	0.780	0.670	2.35	7.05
512	0.754	0.633	2.70	6.45
1024	0.710	0.614	4.20	6.20
2048	0.722	0.586	4.40	5.63
4096	0.667	0.580	4.00	4.95
8192	0.692	0.600	4.44	5.26
16384	0.876	0.714	5.45	6.31
32768	0.971	1.012	7.70	8.74
65536	0.888		6.93	7.72
131072	0.912		6.75	7.47
262144	0.928		6.92	7.60
524288	1.052		7.96	8.69
1048576	0.996		7.72	8.36

which is the ratio of the best vector performance (best values from Column 6 of Table 1, which do not include the initialization time) to the best scalar performance. The results are self-explanatory. Also note the effect of better cache management (in SCFT) for longer transforms.

• *DCFT*

DCFT performance on the 3090 Vector Facility for various values of N and M is found in **Table 3.** For short-length transforms, we have performance numbers for $M = 1$ and $M = 512/N$ (vector block). It turns out that for DCFT no advantage is to be gained by processing more than one vector block at a time. Columns 6 and 7 give normalized run time and total time (including the initialization time), respectively. These are normalized by $M \times N \times \log_2(N)$ to reflect the relative performance level. For the applications requiring repeated calls to DCFT, Column 6 numbers are meaningful numbers (as the initialization is done only once) for comparison with other programs. The initialization time is not significant for long transforms or when many transforms are being computed simultaneously. For short-length transforms, the best performance is achieved for a full vector block. There is no particular advantage to be gained by using a larger M value. And for the same reason, for longer transforms ($N > 2048$) no advantage is to be gained by using $M > 1$. Examination of Columns 6 and 7 reveals a slight jump in times at $N = 4096$ and 65536, which reflects

Table 3 Performance of the DCFT routine.

Transform length N	*No. of transforms M*	*Initialization time (μs)*	*Run time (μs)*	*Total time (μs)*	*Run time* / $M \times N \times \log_2(N)$	*Total time* / $M \times N \times \log_2(N)$
64	1	187	161	348	0.419	0.906
64	8	463	521	984	0.170	0.320
128	1	237	240	477	0.268	0.532
128	4	523	585	1108	0.163	0.309
256	1	325	372	697	0.182	0.340
256	2	627	635	1262	0.155	0.308
512	1	762	721	1483	0.156	0.322
1024	1	727	1563	2290	0.153	0.224
2048	1	1058	3513	4571	0.156	0.203
4096	1	3095	8515	11610	0.173	0.236
8192	1	5433	18442	23875	0.173	0.224
16384	1	9847	40302	50149	0.176	0.219
32768	1	19072	90683	109755	0.184	0.223
65536	1	39122	204941	244063	0.195	0.233
131072	1	76668	442615	519283	0.199	0.233
262144	1	151135	936279	1087414	0.198	0.230
524288	1	301734	2031161	2332895	0.204	0.234

the additional computing cost for the array transposition required in Cases C and D, as discussed in Section 4.

• *SRCFT and SCRFT*

Tables 4 and **5,** respectively, give SRCFT and SCRFT performance on the 3090 Vector Facility for various values of N and M. For short-length transforms, we have performance numbers for $M = 1$, $M = 1024/N$ (vector block), and $M = 4096/N$ (cache block). Columns 6 and 7 give normalized run time and total time (including the initialization time), respectively. These are normalized by $M \times N \times \log_2(N)$ to reflect the relative performance level. For the applications requiring repeated calls to SRCFT/SCRFT, Column 6 numbers are meaningful numbers (as the initialization is done only once) for comparison with other programs. The initialization time is not significant for long transforms or when many transforms are being computed simultaneously. As mentioned before, for short-length transforms, the best performance is achieved for a full cache block. There is no particular advantage to be gained by using a larger M value. And for the same reason, for longer transforms ($N > 2048$), no advantage is to be gained by using $M > 1$. Examination of Columns 6 and 7 reveals a slight jump in times at $N = 8192$ and 262144, which reflects the additional computing cost for the array transposition required in Cases C and D, as discussed in Section 4.

• *SRCFT2 and SCRFT2*

For square arrays of various sizes, SRCFT2 and SCRFT2 performance on the 3090 Vector Facility is shown respectively in **Tables 6** and 7. Column 7 gives normalized run time and Column 8 gives total time (including the initialization time). These are normalized by $M \times N \times \log_2(N)$ to reflect the relative performance level. For large transforms, the initialization time is a very small fraction of the total time, and therefore values of Columns 7 and 8 are almost identical. For large transforms, timings also depend on INC2Y (stride between first elements of the columns or, equivalently, the leading dimension of the output array). If $N_1 > 128$, the recommended values of INC2Y are $N_1 + 32$ for SCRFT2 and $N_1/2 + 16$ for SRCFT2. The minimum required values of INC2Y are $N_1 + 2$ for SCRFT2 and $N_1/2 + 1$ for SRCFT2. In Tables 6 and 7 we have given performance numbers for both of these choices of INC2Y. It can be observed that, for $N_1 > 128$, better performance is obtained with the values of INC2Y recommended above. For $N_2 > 256$, to improve the cache performance in computing the row transforms, we transfer the data into a temporary array, where row transforms are computed. After the computation, the data are transferred back to the output array. Because of this additional cost of the two data transfers, we note a slight drop in performance for $N_2 > 256$.

7. Performance of convolution and correlation subroutines

• *The direct-method subroutines*

Two assembly language subroutines, CORSH and CORSY, compute the correlation function

$$y(j) = \sum_{i=0}^{\min(N_h-1, N_x-1-j)} h(i)x(i+j) \qquad (17)$$

Table 4 Performance of the SRCFT routine.

Transform length N	*No. of transforms* M	*Initialization time* (μs)	*Run time* (μs)	*Total time* (μs)	*Run time* $\frac{}{M \times N \times \log_2(N)}$	*Total time* $\frac{}{M \times N \times \log_2(N)}$
64	1	161	134	295	0.349	0.768
64	16	550	467	1017	0.076	0.166
64	64	558	1589	2147	0.065	0.087
128	1	192	165	357	0.184	0.398
128	8	537	497	1034	0.069	0.144
128	32	523	1747	2270	0.061	0.079
256	1	285	240	525	0.117	0.256
256	4	578	579	1157	0.071	0.141
256	16	570	1966	2536	0.060	0.077
512	1	323	371	694	0.081	0.151
512	2	490	604	1094	0.066	0.119
512	8	486	2066	2552	0.056	0.069
1024	1	559	670	1229	0.065	0.120
1024	4	603	2329	2932	0.057	0.072
2048	1	544	1329	1873	0.059	0.083
2048	2	593	2484	3077	0.055	0.068
4096	1	1064	2781	3845	0.057	0.078
8192	1	1703	6523	8226	0.061	0.077
16384	1	2733	13744	16477	0.060	0.072
32768	1	4924	29692	34616	0.060	0.070
65536	1	9041	62213	71254	0.059	0.068
131072	1	17543	134845	152388	0.061	0.068
262144	1	34422	301498	335920	0.064	0.071
524288	1	69834	626547	696381	0.063	0.070
1048576	1	135693	1325572	1461265	0.063	0.070
2097152	1	268410	2729993	2998403	0.062	0.068

for $j = 0, 1, \cdots, N_y - 1$. The lengths of the h, x, and y sequences are N_h, N_x, and N_y, respectively. For negative j and for convolution, CORSH and CORSY are called with reversed and truncated sequences. The subroutine CORSH uses an algorithm which is designed for efficiency when the h sequence is short; CORSY is designed for use when y is short. Before the subroutines are called, the sequences are shortened, if possible, to the number of elements of h and x which will actually enter the calculation, and y is reduced to those elements which can have nonzero values. For this, the subroutines make the replacements

$$N_h \leftarrow \min(N_h, N_x),$$

$$N_x \leftarrow \min(N_x, N_h + N_y),$$

$$N_y \leftarrow \min(N_y, N_x). \tag{18}$$

In what follows, we use these new definitions of sequence lengths so that within the subroutines we have

$$N_h, N_y \leq N_x \leq N_h + N_y. \tag{19}$$

The subroutines are written so that no arithmetic is done with elements beyond those defined by the length N_x of x. Therefore, the formula for the number of multiply-adds in (17) has two parts: If $N_y \leq N_x - N_h$,

$$M(N_h, N_x, N_y) = N_y N_h, \tag{20}$$

and, for the "tail" of the correlation, where $N_y > N_x - N_h$,

$$M(N_h, N_x, N_y) = (N_x - N_h)N_h + (N_h - N_x + N_y)(N_h + N_x - N_y + 1)/2. \tag{21}$$

• *The direct method with scalar operations*

For comparing methods we use data with $N_x \geq N_h + N_y$. As a basis of comparison, timing was done for the simple FORTRAN program

```
    DO 10 J=0,NY-1
    Y(J)=0
    DO 10 I=0,NH-1
10  Y(J)=Y(J)+H(I)*X(I+J)
```

for computing the correlation defined in (17). The timing for this program can be expressed in the form

$$T_s(N_h, N_y) = AN_hN_y + BN_y + D. \tag{22}$$

Actual timing of this as an in-line program yielded the following values, in microseconds, for the coefficients in (22): $A = 0.2375$, $B = 0.4555$, and $D = 34.9598$, which may be taken as a basis of comparison with the vector subroutines described below.

Table 5 Performance of the SCRFT routine.

Transform length N	*No. of transforms* M	*Initialization time* (μs)	*Run time* (μs)	*Total time* (μs)	*Run time* / $M \times N \times \log_2(N)$	*Total time* / $M \times N \times \log_2(N)$
64	1	164	141	305	0.367	0.794
64	16	481	609	1090	0.099	0.177
64	64	485	2144	2629	0.087	0.107
128	1	175	169	344	0.189	0.384
128	8	418	560	978	0.078	0.136
128	32	427	1886	2313	0.066	0.081
256	1	243	239	482	0.117	0.235
256	4	445	596	1041	0.073	0.127
256	16	462	2017	2479	0.062	0.076
512	1	298	368	666	0.080	0.145
512	2	448	611	1059	0.066	0.115
512	8	482	2103	2585	0.057	0.070
1024	1	604	670	1274	0.065	0.124
1024	4	618	2298	2916	0.056	0.071
2048	1	545	1303	1848	0.058	0.082
2048	2	515	2451	2966	0.054	0.066
4096	1	916	2749	3665	0.056	0.075
8192	1	1709	6490	8199	0.061	0.077
16384	1	2750	13612	16362	0.059	0.071
32768	1	4918	29142	34060	0.059	0.069
65536	1	8902	61035	69937	0.058	0.067
131072	1	17321	133314	150635	0.060	0.068
262144	1	34581	298955	333536	0.063	0.071
524288	1	68282	621029	689311	0.062	0.069
1048576	1	135737	1311152	1446889	0.063	0.069
2097152	1	268970	2708759	2977729	0.062	0.068

• *Timing of CORSH*

The basic algorithm is as described by Gazdag et al. [23] in this issue as Algorithm 2. Let $AM(N_h, N_x, N_y)$ be the time taken for performing the multiplications and additions. Since CORSH does a multiplication and an addition with the single-cycle multiply-add instruction (VMAE), one may expect A to be close to the cycle time of the machine. The time for all other operations is derived as follows: The innermost loop divides the y and x sequences into vector-length segments. For each segment of y, it loops over the elements of h, multiplying each scalar $h(i)$ by a segment of x starting at $x(i + j)$ and adding it to an accumulated segment of the sequence y. Using notation from [23], let the number of segments of y be denoted by

$$S(N_y) = [(N_y - 1)/128] + 1, \tag{23}$$

where $[\cdot]$ denotes the integer part of the bracketed expression. The time for the loop over the h's plus the time for clearing and storing the vector register for accumulating a segment of y is of the form

$$BN_h + C. \tag{24}$$

The time for clearing and storing the vector register for segments of y has a component of the form EN_y. Allowing an overall start-up time D, the formula for the time of the calculation is

$$T_h(N_h, N_x, N_y) = AM(N_h, N_x, N_y) + (BN_h + C)S(N_y) + D + EN_y. \tag{25}$$

A good fit to the actual running time in microseconds for the program was obtained with $A = 0.0191$, $B = 0.816$, $C = 2.61$, $D = 22.49$, and $E = 0.022$. This may be compared with [23, Eq. (24)], where the same formula is given in terms of estimates of machine cycles. The coefficient A comes out just a little over the cycle time of the machine. The incremental start-up time, for each additional $h(i)$ within each vector segment is $B = 0.816$, which is 34% of the time, $128A = 2.44$, for doing the 128 multiply-additions of the vector of x's times $h(i)$. Hockney and Jesshope [24] define their $n_{1/2}$ parameter as the number of vector elements which can be processed in the start-up time for the vector. In this case, we get

$$n_{1/2} = B/A = 43.5. \tag{26}$$

This means that if the vector length were about 43, the machine would process the vector operation at half of full

Table 6 Performance of the SRCFT2 routine for two-dimensional square arrays.

N_1, N_2	*INC2Y*	*Total array size* $N = N_1 \times N_2$	*Initialization time* (μs)	*Run time* (μs)	*Total time* (μs)	$\frac{\textit{Run time}}{N \times \log_2(N)}$	$\frac{\textit{Total time}}{N \times \log_2(N)}$
64	33	4096	789	2862	3651	0.058	0.074
128	80	16384	1145	12905	14050	0.056	0.061
128	65	16384	1059	12925	13984	0.056	0.061
256	144	65536	1176	57922	59098	0.055	0.056
256	129	65536	1169	60775	61944	0.058	0.059
512	272	262144	1056	283555	284611	0.060	0.060
512	257	262144	1052	290695	291747	0.062	0.062
1024	528	1048576	1161	1237136	1238297	0.059	0.059
1024	513	1048576	1167	1329611	1330778	0.063	0.063
2048	1040	4194304	1350	5434115	5435465	0.059	0.059
2048	1025	4194304	1390	6730500	6731890	0.073	0.073

Table 7 Performance of the SCRFT2 routine for two-dimensional square arrays.

N_1, N_2	*INC2Y*	*Total array size* $N = N_1 \times N_2$	*Initialization time* (μs)	*Run time* (μs)	*Total time* (μs)	$\frac{\textit{Run time}}{N \times \log_2(N)}$	$\frac{\textit{Total time}}{N \times \log_2(N)}$
64	66	4096	881	3317	4198	0.067	0.085
128	130	16384	943	13493	14436	0.059	0.063
128	160	16384	1020	13614	14634	0.059	0.064
256	288	65536	1050	58245	59295	0.056	0.057
256	258	65536	1032	60466	61498	0.058	0.059
512	544	262144	1012	280961	281973	0.060	0.060
512	514	262144	1010	288087	289097	0.061	0.061
1024	1056	1048576	1123	1226252	1227375	0.058	0.059
1024	1026	1048576	1141	1319782	1320923	0.063	0.063
2048	2080	4194304	1287	5421507	5422794	0.059	0.059
2048	2050	4194304	1396	6804578	6805974	0.074	0.074

speed, the speed at which operations are processed after vector start-up. The value 43.5 obtained here for the VMAE instruction is fairly good, being well below the vector length of 128. There is a rather large amount of time, $C = 2.61$, for starting the loop over the h's. It is almost the same as the time $128A = 2.44$ for doing an entire vector of 128 multiply-adds. Finally, the call statement and overall start-up time amount to $D = 22.5$, about nine times as long as the 128 multiply-adds.

• *Timing of CORSY*

The algorithm for CORSY is designed for efficiency when N_y is small. It is as described by Gazdag et al. [23] in this issue as Algorithm 1. However, the timing formula for CORSY differs from the one in [23, Eq. (23)] in that the last term there, containing a start-up time for segmenting the y sequence, is missing here. CORSY does not have this term, since it does not have to clear storage for y. Again, let $AM(N_h, N_x, N_y)$ be the time taken for performing the multiplications and additions. For each output $y(i)$, CORSY does a multiply and accumulate (VMCE) with a segment of h's, which is kept in a vector register for all subsequent operations which use it. The loop over the $y(i)$'s is performed for each segment of h taking an amount of time given by a term of the form

$$(BN_y + C) \times S(N_h). \tag{27}$$

When the VMCE instruction multiplies this segment of h by a segment of x, it accumulates the products, in double precision, in four partial sums. Then, in a separate operation, it accumulates the four partial sums of these products. The time for this is proportional to N_y and the

Table 8 Time (μs) for convolution and correlation by the Fourier method in terms of N, the Fourier transform size.

N	*Initial*	*Run*	*Total*
32	477	231	708
64	555	300	855
128	675	364	1039
256	997	525	1522
512	1204	796	2000
1024	1610	1490	3700
2048	2855	2900	5755
4096	5679	6095	11774

number of segments of h and will therefore contribute to the term (27). Since full speed for the VMCE instruction is one cycle, one may expect the value of A for CORSY to be approximately equal to the cycle time of the machine. Forming each resulting $y(j)$ requires time for clearing a storage location, adding its contents to the partial sums, and storing the result. This produces a term EN_y in the timing formula. As mentioned above, a segment of h is loaded only once in a vector register. This is done outside the loop over $y(i)$'s, so it makes a contribution to C in (27). This also takes a small amount of time which depends on the number of elements in h, which we designate by FN_h. Allowing for an overall subroutine call and start-up time, D, the formula for the time of the calculation is given by

$$T_y(N_h, N_x, N_y) = AM(N_h, N_x, N_y) + (BN_y + C)S(N_h) + D + EN_y + FN_h. \quad (28)$$

A very good fit to the actual running time in microseconds for the program was obtained with $A = 0.0189$, $B = 1.84$, $C = 1.97$, $D = 22.43$, $E = 0.02$, and $F = 0.0157$. This may be compared with [23, Eq. (23)], where the same formula is given in terms of estimated machine cycles. The coefficient A comes out to be about the same as the A for CORSH. The time for the loop over the $y(i)$'s is seen in terms of B to be more than twice as large as the corresponding terms for CORSH due to the accumulation of partial sums and some clerical operations. If the $n_{1/2}$ of [24] is evaluated as for CORSH, above, one obtains

$$n_{1/2} = B/A = 97.3, \quad (29)$$

which is quite close to the vector length for the machine. This result indicates that the amount of parallelism for the VMCE instruction is too small, or, in other words, that the vector register should be longer to use the VMCE instruction efficiently.

• *Timing of the Fourier method*

The timing of the Fourier method depends entirely on the size N of the Fourier transform which must be computed. For the simple parameters considered above, N will be the next higher power of 2 above $N_h + N_y - 1$. Formally, this may be expressed

$$N = 2^{\lceil \log_2(N_h+N_y-1) \rceil}, \quad (30)$$

where $\lceil \cdot \rceil$ denotes the ceiling of the expression—i.e., the next higher integer. All of the vector subroutines require an initialization time which is negligible for CORSH and CORSY but is large enough in the Fourier method subroutines to require some consideration. Therefore, the initialization time is given with the run and total time in **Table 8** as functions of N.

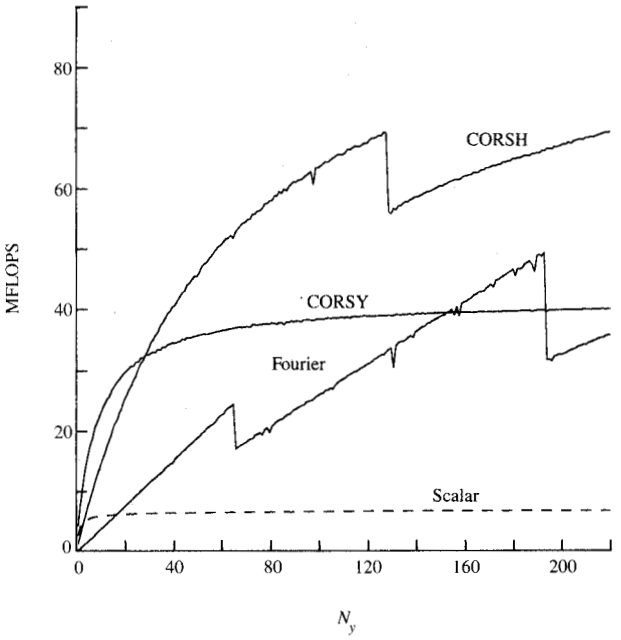

Figure 1

For $N_h = 64$ and $0 < N_y < 220$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

• *Discussion of timing for convolution/correlation*

In this section, performance is described in terms of computing speed by dividing the number of multiplications and additions, $M(N_h, N_x, N_y)$ in Eq. (20) by the time required for the calculation. For the direct methods, this gives the rate at which the machine actually performs FLOPs (floating-point multiplications and additions), but this is not the case for the Fourier methods, where the number of FLOPs is in general lower.

In the comparisons of timing which follow, we continue to assume, as in the above discussions, that the input x sequence is indefinitely long. The effect is that $N_x = N_y + N_h$ and the number of operations is N_hN_y. The plots in **Figure 1** show the MFLOP rates for $N_h = 64$ as functions of N_y for

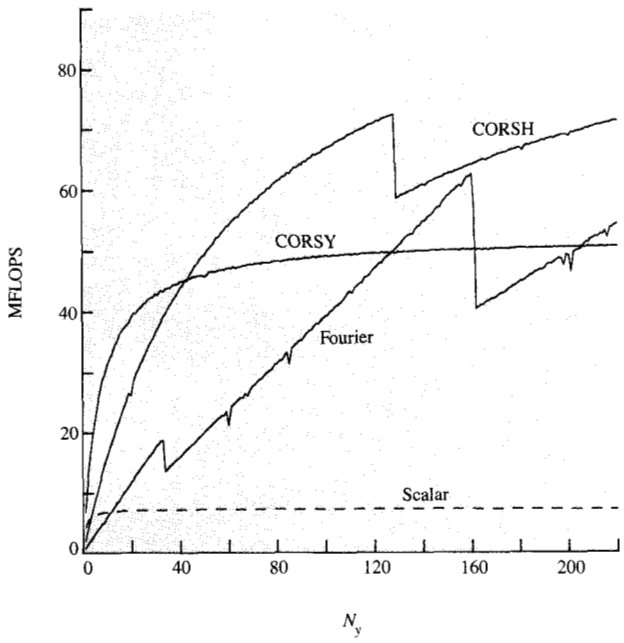

Figure 2

For $N_h = 96$ and $0 < N_y < 220$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

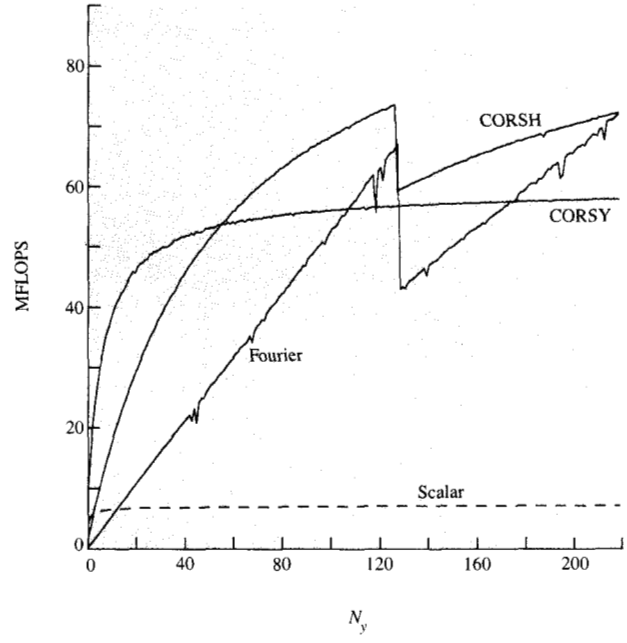

Figure 3

For $N_h = 128$ and $0 < N_y < 220$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

1. The in-line scalar program for the direct method.
2. CORSH, which uses the best strategy for short h.
3. CORSY, which uses the best strategy for short y.
4. The Fourier transform method which is used in the subroutines SCON, SCOR, and SACOR.

Small glitches appear in some plots, showing a drop in speed at isolated points. These result from interrupts caused by multitask operation of the machine. They are left in the plots to show their relative effect on performance in actual calculations.

For a very small range of parameters which is of little interest here, the direct in-line method is best. In Fig. 1, showing rates for $N_h = 64$, CORSY shows the best performance up to about $N_y = 30$. Increasing N_h to 96 (as shown in **Figure 2**), which for CORSY increases the number of elements in the vector register, increases CORSY's speed. (The Fourier method shows an expected improvement in performance also.)

Results for $N_h = 128$ are shown in **Figure 3.** Here, h just fills a vector register, giving CORSY the greatest advantage and increasing its range to $N_y = 55$. In general, since CORSH uses the better strategy for small N_h, we expect to see this crossover point at a higher N_y for higher N_h, as shown in Fig. 2. In the plots for CORSH, one sees the discontinuity at multiples of $N_y = 128$ where the time increases by the start-up time for a new vector segment (divided by N_hN_y).

Figures 4–6 show the performance plots for a wider range of N_y (up to 2200) with the MFLOPS going twice as high, up to 180. In Fig. 4, the $N_h = 64$ plot demonstrates that CORSH is better for most of the range shown.

Figure 5 displays an unexpected result: There are many crossings between the performance curves for the Fourier method and CORSH. In Fig. 6, the $N_h = 128$ plot shows the Fourier method to be best above $N_y = 210$. For $N_y < 210$, there are many crossings between the performance curves.

The locations of the crossover points in the timing curves are important in choosing which of the programs to use. For the sake of comparing CORSY and CORSH, **Figure 7** shows a plot giving the value of N_y at the crossover point for each N_h. Thus, for all N_h, N_y below the graph, CORSY is faster than CORSH. For changes in N_h within a segment length, the graph is a straight line which, for the first segment, has a slope of 0.42. It is interesting to note that the observation made above that n_{v_2} is too high for CORSY is supported by the data plotted in Fig. 7. The drop in the performance curve caused each time a segment of h is filled causes the equal-performance graph to stay below $N_y = 57$; thus, CORSY will never be better for more than 57.

8. Conclusions

The general methods for scheduling the FFT algorithm described by Agarwal [4] offered great advantages in

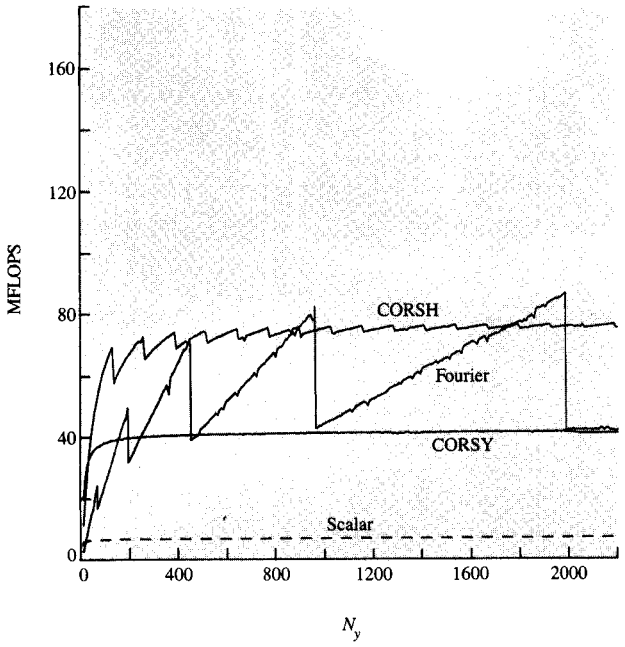

Figure 4

For $N_h = 64$ and $0 < N_y < 2200$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

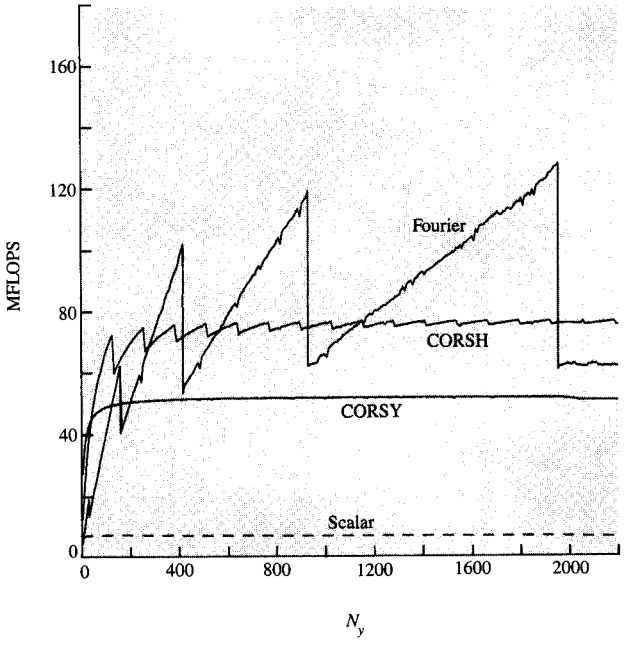

Figure 5

For $N_h = 96$ and $0 < N_y < 2200$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

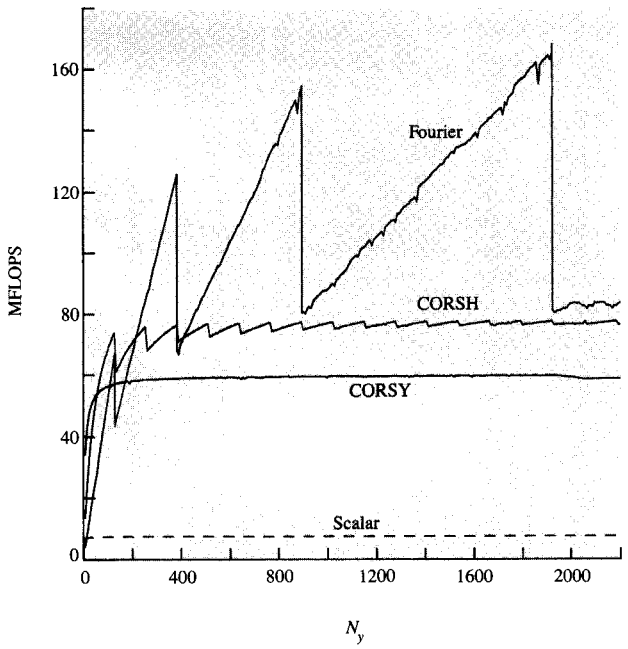

Figure 6

For $N_h = 128$ and $0 < N_y < 2200$, plot of computational speed = $2N_hN_y$/time for the in-line FORTRAN scalar program, CORSH, CORSY, and the Fourier method as functions of N_y. ($N_x = N_y + N_h$.)

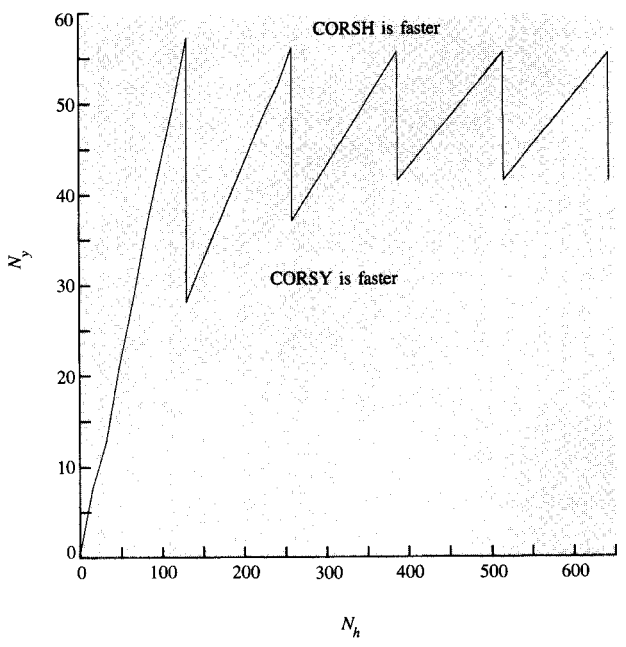

Figure 7

Plot of values of N_h and N_y where the computational speeds of CORSH and CORSY are equal. In the region below the graph, CORSY is faster; above the graph, CORSH is faster. ($N_x = N_y + N_h$.)

vectorizing the FFT algorithm for the IBM 3090 Vector Facility. However, further details of the structuring, depending on the number of vectors and their size, had to be devised. The radix-4 algorithm gave some savings in the number of multiplications and permitted a large number of operations within vector registers between accesses to storage. The relation between vector and cache sizes made it necessary to schedule the algorithm in such a way as to keep as much computing within a cache block as possible. Where the algorithm called for accessing data with strides equal to powers of 2, redimensioning of data arrays was shown to make great improvements in cache performance. In Tables 6 and 7, it is shown how run times vary considerably by changing the INC2Y parameter. This is an example where the user may, by simply altering a stride, reduce repeated accesses to the same cache lines, with a consequent improvement in running time. Programming experience has shown that special formulation of the algorithms can be very important and has led to some fairly general programming principles and techniques which have yielded significant improvements in performance.

To achieve high performance, it was very important to divide the calculations into vector-block and cache-block units. In large arrays special techniques had to be used to transpose data without producing too many "cache misses."

To do the above and maintain efficiency, it was necessary to have the subroutines do preprocessing, and when sequences were short it was efficient to have them compute several transforms at once.

Comparisons with the best scalar programs, run on the IBM 3090, showed the vector program to run from 1.2 to 8.0 times as fast, allowing for initialization (see Table 2). However, for full-speed operation, i.e, not counting initialization, speedups of from 5 to 9½ times were achieved.

The formulation of the FFT algorithm and the design of the programs made it possible to keep all vector registers filled at every iteration. The number of vector registers and the three-address operation code made it possible to use the radix-4 FFT algorithm so that large amounts of computing could be done within registers with relatively few storage references. The permutations of data in the FFT algorithm could be performed economically by the efficient use of strides and the indexed load/store operations.

During the planning stages of these programs, it was expected that for the majority of problem parameters, it would be most efficient to compute convolutions with Fourier transforms. In fact, estimates of the numbers of arithmetic operations and results on conventional scalar machines showed that the Fourier methods were better for sequence lengths of more than 16 to 32. The Fourier transform methods require fewer arithmetic operations than the direct methods. Nevertheless, the crossover points for direct methods are higher than those for scalar machines. Each of these methods depends upon a vector operation which does a multiplication and an addition in a single instruction—that is, in a single cycle.

One direct method uses an operation (VMAES) which multiplies a scalar by a vector and adds the result to a vector. This was used in the internal subroutine CORSH, which was designed to be efficient when one of the input sequences is short. Here we have found out that "short" may be as long as 210. This subroutine will be very useful in many large problems where one is doing digital filtering on a digitized signal or, in other words, computing a moving average over a long signal with a fixed short sequence of weights.

The second vector operation referred to above multiplies a vector by a vector and accumulates the products (VMCE). It only accumulates partial sums so that additional overhead is required. The result is that it is never superior for an output sequence length greater than 57. However, there are many situations where long input sequences of samples of stochastic variables are used and relatively few values of the computed covariance function are desired.

It may be seen in Fig. 7 that examining the computing speed of these two direct methods as a function of input sequence length N_h and output sequence length N_y shows that the N_h, N_y plane is divided into two disjoint regions where one or the other method is superior. Thus, it is important that both be available.

Data in the figures and in Table 8 show that for large ranges of parameters, the performance curves for the three methods make many crossings. This is caused by the vector segmentation and by the fact that the present Fourier transform subroutines apply only to lengths equal to powers of 2. Therefore, there is no simple test of parameters to determine the best method. Instead, timing formulas must be used.

It is well known that vector machines make the task of program planning and writing far more critical than do scalar machines. Therefore, the problem of making the full capabilities of vector machines available can, in part, be solved by identifying computational kernels and making an intensive effort to plan and program subroutines for them. This paper describes a contribution towards that goal.

Acknowledgments

R. C. A. would like to thank G. Paul of the IBM Thomas J. Watson Research Center for familiarizing him with vector architecture and for suggesting the study of the vectorization of the FFT algorithm. Discussions with S. Winograd of the Department of Mathematical Sciences at the Research Center were informative and stimulating. F. Gustavson, manager of the project, gave valuable support and maintained close contact and interest in all phases of the work. B. Tuckerman wrote and made available to us his vector simulator, with its program debugging facilities, which was indispensable in getting programs developed long before

the machine was available. The authors wish to express their gratitude to G. Radicati of the IBM Rome Scientific Center. He graciously sent us an early version of the paper he coauthored (Ref. [23]) and had previously given us the program which became the basis for the subroutine CORSH. He also performed the timing and testing of our programs before the 3090 Vector Facility became available to us. G. Slishman gave valuable system support which facilitated the use of the machine in Kingston and, later, the machine at the Research Center. We are also grateful to the ESSL software development group at IBM Kingston under S. Schmidt for performing final testing and for preparing the programs for acceptance as IBM products.

References

1. Carl H. Savit, "Geophysical DP Requirements Could Exceed the World's GP Capacity by 1985," *AFIPS Conf. Proc.* **47,** 63–66 (1978).
2. *Engineering and Scientific Subroutine Library Guide and Reference,* Order No. SC23-0184, available through IBM branch offices.
3. *Engineering and Scientific Subroutine Library General Information,* Order No. SC23-0182, available through IBM branch offices.
4. Ramesh C. Agarwal, "An Efficient Formulation of the Mixed-Radix FFT Algorithm," *Proceedings of the International Conference on Computers, Systems, and Signal Processing,* Bangalore, India, December 10–12, 1984, pp. 769–772.
5. A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.
6. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comp.* **19,** 297–301 (April 1965).
7. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform Algorithm and Its Applications," *IEEE Trans. Education* **E-12,** No. 1, 27–34 (March 1969).
8. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "Application of the Fast Fourier Transform to Computation of Fourier Integrals, Fourier Series, and Convolution Integrals," *IEEE Trans. Audio Electroacoust.* **AU-15,** No. 2, 79–84 (June 1967).
9. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "Historical Notes on the Fast Fourier Transform," *IEEE Trans. Audio Electroacoust.* **AU-15,** No. 2, 76–79 (June 1967). See also *Proc. IEEE* **55,** No. 10, 1675–1677 (October 1967).
10. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Finite Fast Fourier Transform," *IEEE Trans. Audio Electroacoust.* **AU-17,** No. 2, 77–85 (June 1969).
11. J. W. Cooley, "Fast Fourier Transform," *Encyclopedia of Computer Sciences,* A. Ralston, Ed., Mason Charter Publishers Inc., New York, 1976.
12. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform: Programming Considerations in the Calculation of Sine, Cosine and Laplace Transforms," *J. Sound Vibr. Anal.* (University of Southampton, England) **12,** No. 3, 315–337 (July 1970).
13. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Application of the Fast Fourier Transform Algorithm to the Estimation of Spectra and Cross-Spectra," *J. Sound Vibr. Anal.* (University of Southampton, England) **12,** No. 3, 339–352 (July 1970).
14. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform and Its Application to Time Series Analysis," Ch. 14 of *Statistical Methods for Digital Computers,* Vol. III of *Mathematical Methods for Digital Computers,* K. Enslein, A. Ralston, and H. Wilf, Eds., Wiley-Interscience, New York, 1977.
15. J. D. Markel and A. H. Gray, *Linear Prediction of Speech,* Springer-Verlag New York, 1976.
16. M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *J. ACM* **15,** 253–264 (1968).
17. David G. Korn and J. Lambiotte, Jr., "Computing the Fast Fourier Transform on a Vector Computer," *Math. Comp.* **33,** 977–992 (July 1979).
18. P. N. Swarztrauber, "Vectorizing the FFTs," *Parallel Computations,* G. Rodrique, Ed., Academic Press, Inc., New York, 1982.
19. W. P. Petersen, "Vector Fortran for Numerical Problems on CRAY-1," *Commun. ACM* **26,** 1008–1021 (November 1983).
20. A. Oppenheim and C. Weinstein, "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform," *Proc. IEEE* **60,** No. 8, 957–976 (August 1972).
21. R. C. Singleton, "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Trans. Audio Electroacoust.* **AU-17,** 93–103 (June 1969). See also *Digital Signal Processing,* L. Rabiner and C. Rader, Eds., IEEE Press, New York, 1975.
22. G. D. Bergland, "A Fast Fourier Transform Algorithm Using Base 8 Iterations," *Math. Comp.* **22,** 275–279 (April 1968). See also *Digital Signal Processing,* L. Rabiner and C. Rader, Eds., IEEE Press, New York, 1975.
23. J. Gazdag, G. Radicati, P. Sguazzero, and H. H. Wang, "Seismic Migration on the IBM 3090 Vector Facility," *IBM J. Res. Develop.* **30,** No. 2, 172–183 (1986, this issue).
24. R. W. Hockney and C. R. Jesshope, *Parallel Computers* Adam Hilger Ltd., Bristol, England, 1981.

Received November 5, 1985; accepted for publication November 20, 1985

Ramesh C. Agarwal *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Agarwal received his B.Tech. degree (with honors) from the Indian Institute of Technology (IIT), Bombay, India, and the M.S. and Ph.D. degrees from Rice University, Houston, Texas, all in electrical engineering, in 1968, 1970, and 1974, respectively. During 1971–72, he was an Associate Lecturer at the School of Radar Studies, IIT Delhi, India; from 1974 to 1977 he was with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. He spent the period 1977–1981 as a Principal Scientific Officer at the Centre for Applied Research in Electronics at IIT Delhi, India, and returned to IBM in 1982. His research interests have included network synthesis, information theory and coding, number theoretic transforms, fast algorithms for computing convolution and DFT, application of digital signal processing to structure refinement of large biological molecules using X-ray diffraction data, sonar signal processing, architecture for special-purpose signal processors, digital DTMF/MF receivers, filter structures, analysis of Kennedy assassination tapes, computation of elementary functions, and vectorization for engineering/scientific computations. Dr. Agarwal received the 1974 Acoustics, Speech, and Signal Processing Senior Award from the Institute of Electrical and Electronics Engineers for papers on number theoretic transforms, an IBM Outstanding Contribution Award in 1979 for work on crystallographic refinement of biological molecules, an IBM Outstanding Technical Achievement Award in 1984 for elementary functions work, and an IBM Outstanding Innovation Award in 1985 for his work in vectorizing the FFT algorithm.

James W. Cooley *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Cooley received his B.A. from Manhattan College, New York City, in 1949 and his M.A. and Ph.D. in applied mathematics from Columbia University, New

York, in 1951 and 1961, respectively. He was a programmer on John von Neumann's electronic computer at the Institute for Advanced Study, Princeton, New Jersey, beginning in 1953; in 1956 he became a research assistant at the Computing Center of the Courant Institute at New York University, where he worked on numerical methods for quantum mechanical calculations. Since 1962, he has been on the Research staff of the Thomas J. Watson Research Center in Yorktown Heights, except for a one-year sabbatical, 1973–1974, which he spent at the Royal Institute of Technology, Stockholm, Sweden. At IBM he has worked on computational methods for solving diffusion and transport equations in applications to transistor and ionic flow problems. He has assisted in the development and use of mathematical models of the electrical activity in nerve and muscle membranes and in assorted eigenvalue problems and numerical methods for solving ordinary and partial differential equations. Dr. Cooley has been involved in the development of numerical methods for computers, including the fast Fourier transform and convolution algorithms. In recent years, he has participated in the development of software for elementary functions and signal processing programs for the IBM 3090 Vector Facility. He is a Fellow of the Institute of Electrical and Electronics Engineers. Dr. Cooley holds five IBM awards and four IEEE awards.