An Algorithm for the Fast Hartley Transform

Ronald F. Ullmann

Abstract

The fast Fourier transform algorithm uses two properties of the discrete Fourier transform to decrease the number of computations. The first property is that the kernel for the Fourier transform is periodic. The second property is the shift rule. The shift rule of the Fourier transform states that shifting a function in the time domain corresponds to multiplying the function in the frequency domain by an complex exponential factor. The Hartley transform has similar properties to the Fourier transform. The Hartley kernel is periodic, and shifting a function in the time domain corresponds to two multiplications in the frequency domain. Due to the properties of the discrete Hartley transform, a fast Hartley transform algorithm can be based on the algorithm for the fast Fourier transform. Since the fast Hartley transform can work with only real numbers, the transform requires half of the multiplications of a similar FFT. The transform and other operations in Hartley space take less computing time than corresponding operations in Fourier space.

Introduction to the Fourier and Hartley Transform

The discrete Fourier transform (DFT) transforms a sequence of real numbers from the time domain into a sequence of complex numbers in the frequency domain. Half of the numbers in the frequency domain are redundant because the information in the negative frequencies repeat the information in the positive frequencies. In other words, if $X_f(k)$ is the Fourier transform of a real sequence of real numbers, than

$$\textbf{Real}\,(X_f(k)) \;=\; \textbf{Real}\,(X_f(-k))$$

and

$$\textbf{Imag}\,(X_f(k)) \;=\; \textbf{Imag}\,(-X_f(-k))\,.$$

The multiplication of complex numbers in a computer require four floating point multiplications and two floating point additions. Due to the amount of memory required, the redundant information, and the number of computations needed, the Fourier transform is not the most efficient method of transforming real numbers to the frequency domain. Unless the input sequence consists of complex numbers, the Fourier transform is not efficient. Some DFT programs use an algorithm that repacks the numbers to eliminate the redundancy. For example, the negative real frequency values are replaced by the positive imaginary values, and all the rest of the imaginary values are ignored.

In another article in this report, Ottolini introduces the Hartley transform. The discrete Hartley transform (DHT) transforms a sequence of real numbers in the time domain into a sequence of real numbers in the frequency domain. The Hartley transform can be viewed as an algorithm that removes redundancy in the Fourier domain by repacking the numbers through the relation

$$X_h(k) = \textbf{REAL}[X_f(k)] - \textbf{IMAG}[X_f(k)] \tag{1}$$

where $X_h(k)$ is the Hartley transform of a sequence, and $X_f(k)$ is the Fourier transform of the same sequence. As can seen in equation (1), the Hartley transform results in real numbers. No information is lost because data in the Hartley domain can be converted back to the Fourier domain by the relations

$$\textbf{REAL}[X_f(k)] = [X_h(k) + X_h(-k)]/2 \tag{2a}$$

$$\textbf{IMAG}[X_f(k)] = [X_h(k) - X_h(-k)]/2 \tag{2b}$$

Since no information gets lost in the transform, the real sequence in the frequency domain uses the minimum required memory space. The frequency representation is real, which means the Hartley transform requires fewer multiplications. Based on these primary criteria, the Hartley transform is more efficient than the straight Fourier transform.

In his article, Ottolini covers some of the geophysical applications of the Hartley transform, but he did not include a method of calculating the Hartley transform of an input sequence. The Fourier and Hartley transforms are very similar and share many properties. Among these properties are the shift rule and the periodic properties of their kernels. Both of these properties are useful in constructing the fast Fourier transform (FFT). This article contains the derivation of a computer algorithm that computes a fast Hartley transform (FHT). The derivation of the FHT algorithm is based on the FFT algorithm. This article also includes some discussion of the advantages and disadvantages of using the Hartley transform.

A Review on How the FFT Works

The N-point discrete Fourier transform of a time sequence, $x(n)$, has the formula

$$X_f(k) = \sum_{n=0}^{N-1} x(n)\, e^{-i\,(2\pi/N)\,nk} ,$$

where $X_f(k)$ is the DFT of $x(n)$. The FFT algorithm makes use of the symmetry present in the DFT equation above. First, the FFT algorithm breaks the original N-point sequence into two smaller $(N/2)$-point sequences. For example, the FFT breaks the original N-point sequence $x(n)$ into the $(N/2)$-point sequences

$$x_1(n) = x(2n) \qquad n = 0, 1, \ldots, \frac{N}{2} - 1$$

and

$$x_2(n) = x(2n+1) \qquad n = 0, 1, \ldots, \frac{N}{2} - 1$$

The first sequence, $x_1(n)$, contains the even members of $x(n)$, and $x_2(n)$ contains the odd members of $x(n)$. The two $(N/2)$-point sequences are Fourier transformed and combined into a N-point DFT of $x(n)$ by the formula

$$X_f(k) = \sum_{n=0}^{N/2-1} x(2n)\, e^{-i\,(2\pi/N)\,2nk} + \sum_{n=0}^{N/2-1} x(2n+1)\, e^{-i\,(2\pi/N)\,(2n+1)k} \qquad (3)$$

The second term in equation (3) is not a true DFT because the kernel in the second term, $e^{-i\,(2\pi/N)\,(2n+1)k}$, is not the DFT kernel. Using the relation

$$e^{-i\,(2\pi/N)\,(2n+1)k} = e^{-i\,(2\pi/N)\,k}\, e^{-i\,(2\pi/N)\,2nk} ,$$

equation (3) becomes

$$X_f(k) = \sum_{n=0}^{N/2-1} x(2n)\, e^{-i(2\pi/N)\,2nk} + e^{-i\,(2\pi/N)\,k} \sum_{n=0}^{N/2-1} x(2n+1)\, e^{-i(2\pi/N)\,2nk} \qquad (4a)$$

The second term of equation (4a) is now a true DFT. Equation (4a) can be rewritten as

$$X_f(k) = X_{f1}(k) + e^{-i\,(2\pi/N)\,k}\, X_{f2}(k) \qquad (4b)$$

where $X_{f1}(k)$ is the discrete Fourier transform of $x_1(n)$, and $X_{f2}(k)$ is the discrete Fourier transform of $x_2(n)$. The complex exponential in the second term of equation (4b) comes from the shift rule of the Fourier transform, which is given by

$$X_f(k+c) = e^{-i\,(2\pi/N)\,c}\, X_f(k)$$

where c is some constant. When the term $e^{-i\,(2\pi/N)\,k}$ was factored out of the second term in equation (3), the time sequence $x_2(n)$ was shifted to the left by one. The shifted $x_2(n)$ was Fourier transformed to become $X_{f2}(k)$. The shift term $e^{-i\,(2\pi/N)\,k}$ in equation (4b) re-shifts $x_2(n)$ to its original location in the time domain. Once $X_{f2}(k)$ is shifted back to the right, the sum of the two terms in equation (4a) can take place. The shift rule allows the FFT algorithm to take the DFT of two $(N/2)$-point sequences and combine them into one N-point sequence.

The terms $X_{f1}(k)$ and $X_{f2}(k)$ in equation (4b) are defined for $0 \le k \le N/2 - 1$. Since $X_f(k)$ is defined for $0 \le k \le N - 1$, a rule is needed to obtain the values of $X_f(k)$ for $k \ge N/2$. In order to find the rule, the periodic property of the DFT is used. The kernel of the DFT, $e^{-i\,(2\pi/N)}$, is periodic and has a period of N. For example, the term $e^{-i\,(2\pi/N)\,2nk}$ in equation (4a) has a period of $(N/2)$. When k gets larger than $(N/2 - 1)$, then $(N/2)$ can be subtracted from k to get

$$e^{-i\,(2\pi/N)\,2(k\,-\,N/2)} \;=\; e^{-i\,(2\pi/N)2k}\,. \tag{5}$$

The two terms in equation (4b), $X_{f1}(k)$ and $X_{f2}(k)$, have a period of $N/2$. Combining equations (4a) and (5), $X_f(k)$ for all k becomes

$$X_f(k) \;=\; \begin{cases} X_{f1}(k) + e^{-i\,(2\pi/N)\,k}\,X_{f2}(k) & 0 \le k \le \dfrac{N}{2} - 1 \\ X_{f1}(k\,-\,N/2) + e^{-i\,(2\pi/N)\,k}\,X_{f2}(k\,-\,N/2) & \dfrac{N}{2} \le k \le N - 1 \end{cases} \tag{6}$$

Using the fact that the term $e^{-i\,(2\pi/N)\,k}$ in equation (4a) has a period of N, and

$$e^{-i\,(2\pi/N)\,(k\,+\,N/2)} \;=\; -\,e^{-i\,(2\pi/N)k}\,,$$

equation (6) can be rewritten as

$$X_f(k) \;=\; \begin{cases} X_{f1}(k) + e^{-i\,(2\pi/N)\,k}\,X_{f2}(k) & 0 \le k \le \dfrac{N}{2} - 1 \\ X_{f1}(k\,-\,N/2) - e^{-i\,(2\pi/N)\,(k\,-\,N/2)}\,X_{f2}(k\,-\,N/2) & \dfrac{N}{2} \le k \le N - 1 \end{cases} \tag{7}$$

The periodic property of the DFT allows the FFT algorithm to compute all of the values of a N-point sequence from two $(N/2)$-point sequences.

The top part of figure 1 shows the flow graphs for computing an eight-point DFT from two four-point transforms. The input sequences on the left are $X_{f1}(k)$ and $X_{f2}(k)$, and the output sequence on the right is $X_f(k)$. The lines in figure 1 show the flow of data from the left side to the right side. The bottom part of figure 1 shows the general FFT butterfly flow graph. If a and b are inputs on the left side of the flow graph, then the output on the right

side is $(a + wb)$ and $(a - wb)$. In the top part of figure 1, $X_{f1}(k)$ corresponds to a, and $X_{f2}(k)$ corresponds to b. The w corresponds to the factor $e^{-i(2\pi/N)k}$. The flow graph in the bottom part of figure 1 is called the FFT butterfly due to its shape. The FFT butterfly is used recursively to combine shorter DFT sequences into longer DFT sequences.

Figure 1 just shows the transform for an eight-point transform. The FFT butterfly works on any sequence whose length is a power of two. The FFT starts by performing two-point transforms of the input sequence. The FFT takes the output from the two-point transforms and repeats the process, but with a four-point transform. The FFT continues in this way until N is reached. Several books list a Fortran program that performs the FFT (Claerbout, 1976). Appendix A contains a subroutine written in Ratfor that performs the FFT algorithm.

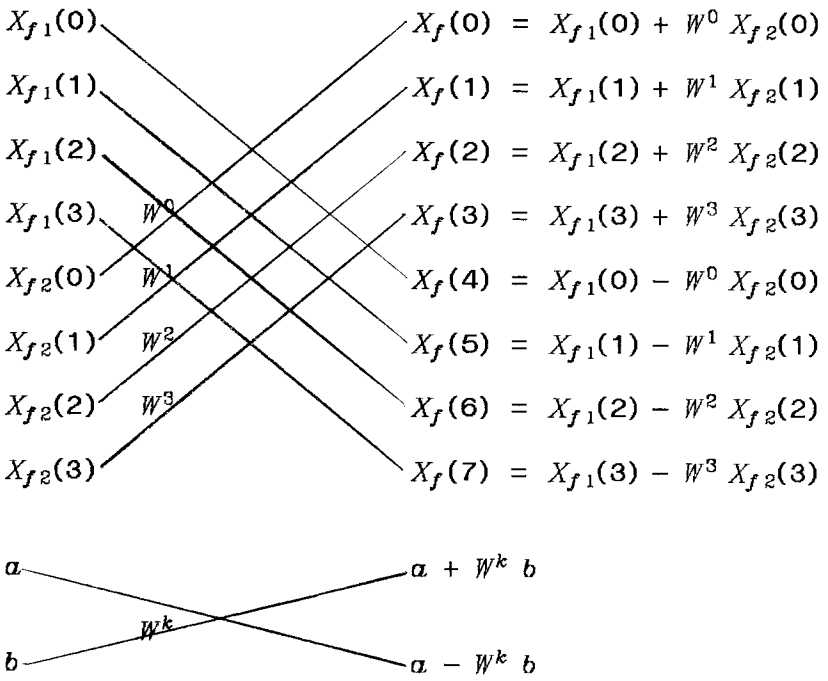

FIG. 1. Butterfly flow graph for the FFT. Note that $W = e^{-i(2\pi/N)}$, and that data flows from the left to the right.

Deriving the Fast Hartley Transform

The N-point discrete Hartley transform (DHT) is given by the formula

$$X_h(k) = \sum_{n=0}^{N-1} x(n) \operatorname{cas}((2\pi/N)nk)$$

where

$$\text{cas}\,((2\pi/N)\,nk) = \cos((2\pi/N)\,nk) + \sin((2\pi/N)\,nk)\,.$$

The FHT algorithm starts the same as the FFT. First, the FHT splits the N-point sequence $x(n)$ into two smaller $(N/2)$-point sequences $x_1(n)$ and $x_2(n)$. The two sequences are Hartley transformed into $X_{h1}(k)$ and $X_{h2}(k)$ and combined into $X_h(k)$, as the FFT does in equation (3). However, the Hartley transform combines $X_{h1}(k)$ and $X_{h2}(k)$ with the equation

$$X_h(k) = \sum_{n=0}^{N/2-1} x(2n)\,\text{cas}\,((2\pi/N)\,2nk) + \sum_{n=0}^{N/2-1} x(2n+1)\,\text{cas}\,((2\pi/N)\,(2n+1)k)\,. \tag{8}$$

Just as for the equation (3), the second term in equation (8) is not a DHT because the time sequence $x_2(n)$ is shifted to the left. The time sequence $x_2(n)$ needs to be shifted to the left by one by the shift rule of the DHT, which is given by the formula

$$X_h(k+c) = X_h(k)\cos(c) + X_h(-k)\sin(c) \tag{9}$$

where c is some constant. After shifting $x_2(n)$ by using equation (9), equation (8) becomes

$$X_h(k) = X_{h1}(k) + \cos((2\pi/N)\,k)\,X_{h2}(k) + \sin((2\pi/N)\,k)\,X_{h2}(-k)\,. \tag{10}$$

where $X_{h1}(k)$ is the Hartley transform of $x_1(n)$, and $X_{h2}(k)$ is the Hartley transform of $x_2(n)$. By using this shift rule, two $(N/2)$-point DHTs can be combined into one N-point DHT.

The rule for calculating the values of $X_h(k)$ when $k \geq N/2$ is identical to the rule for the Fourier transform. The kernel of the DHT has the same periodic properties as the kernel of the DFT. For example,

$$\text{cas}\,((2\pi/N)\,(n+N)) = \text{cas}\,((2\pi/N)\,n)$$

and

$$\text{cas}\,((2\pi/N)\,(n+N/2)) = -\text{cas}\,((2\pi/N)\,n)\,.$$

Using the periodic properties shown in above equations, the values of $X_h(k)$ for all k are given by

$$X_h(k) = \begin{cases} X_{h1}(k) + \cos((2\pi/N)\,k)\,X_{h2}(k) + \sin((2\pi/N)\,k)\,X_{h2}(-k) & 0 \leq k \leq \dfrac{N}{2}-1 \\ X_{h1}(k-N/2) - \cos((2\pi/N)\,(k-N/2))\,X_{h2}(k-N/2) - \sin((2\pi/N)\,(k-N/2))\,X_{h2}(-k+N/2) & \dfrac{N}{2} \leq k \leq N-1 \end{cases} \tag{11}$$

Equation (11) gives the basic formula for the FHT algorithm. The main difference between equation (11) for the FHT and equation (7) for the FFT is that for each k, $X_h(k)$ requires three values and $X_f(k)$ requires only two values. Since equation (11) requires three values, the standard FFT butterfly flow graph shown in figure 1 cannot be used for the FHT. Instead, the FHT requires a double butterfly flow graph to implement equation (11). Figure 2 shows one of the double butterflies used for evaluating a eight-point transform from two four-point transforms. The double butterfly uses four inputs and produces four outputs. The inputs on the left side are $X_{h1}(1)$, $X_{h1}(3)$, $X_{h2}(1)$, and $X_{h2}(3)$, and the outputs on the right side are $X_h(1)$, $X_h(3)$, $X_h(5)$, and $X_h(7)$. There is a reason why the double butterfly needs four inputs when equation (11) only calls for three. According to equation (11), the output $X_h(1)$ requires the inputs $X_{h1}(1)$, $X_{h2}(1)$, and $X_{h2}(-1)$. Since $X_{h2}(k)$ is not defined for $k < 0$, a corresponding entry is required. The negative frequency components of a N-point sequence are stored after the $(N/2)$. A general formula for computing the location of a negative frequency component is

$$X_h(-k) = X_h(N-k) \qquad -\frac{N}{2} \leq k < 0 .$$

In the case of $X_{h2}(-1)$,

$$X_{h2}(-1) = X_{h2}(4-1) = X_{h2}(3) .$$

Therefore, the inputs for $X_h(1)$ are $X_{h1}(1)$, $X_{h2}(1)$, and $X_{h2}(3)$. The output $X_h(3)$ also requires the inputs $X_{h2}(1)$ and $X_{h2}(3)$. Since $X_{h2}(1)$ and $X_{h2}(3)$ are needed, the double butterfly calculates $X_h(3)$ at the same time it calculates $X_h(1)$.

The outputs for $X_h(0)$, $X_h(2)$, $X_h(4)$, and $X_h(6)$ in figure 2 each require only two inputs. The reason is that the inputs for these values do not have corresponding negative frequencies. For example, $X_h(2)$ requires $X_{h1}(2)$, $X_{h2}(2)$, and $X_{h2}(-2)$ for inputs. Both $X_{h2}(2)$ and $X_{h2}(-2)$ are the Nyquist frequencies of a four-point sequence, which means $X_{h2}(2) = X_{h2}(-2)$. Appendix B has a Ratfor listing of a subroutine that calculates the FHT.

An Examination of the FHT Subroutine

The FFT listing in Appendix A and the FHT listing in Appendix B both have very similar structures. Both algorithms have the same first *do* loop. This loop re-orders the input sequence $x(n)$ according to bit-reversed order. At the same time, the *do* loop scales the numbers in the input sequence. The data re-ordering loop is a standard part of most FFT algorithms. See Rabiner and Gold (1975) for a good explanation of bit-reversed order.

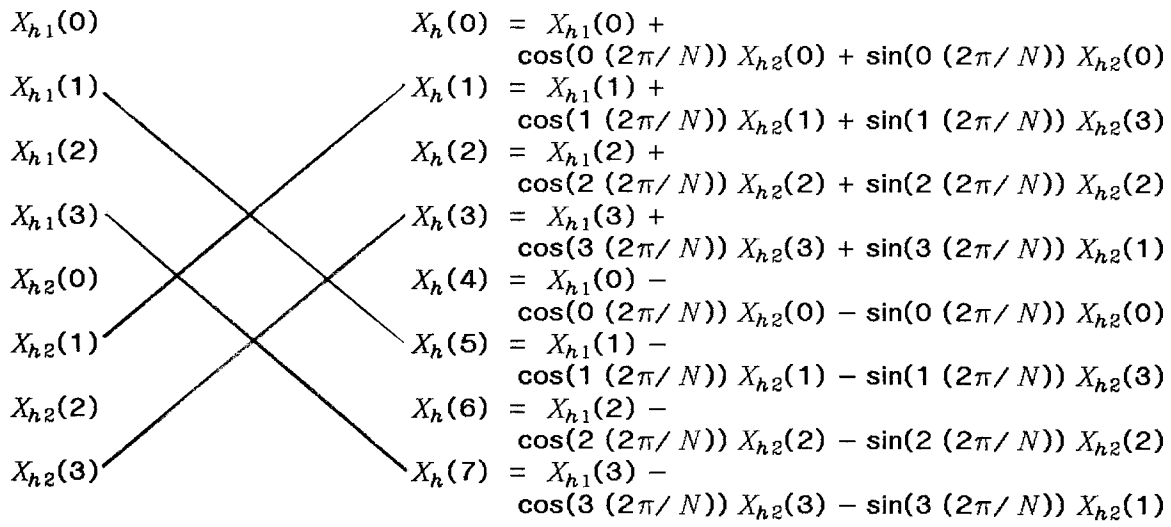

FIG. 2. A FHT double butterfly for an eight-point sequence. Data flows from the left side to the right.

The actual transforms are done within the *repeat-until* loops. Appendix C provides a list of the major variables in the *repeat-until* loop and explains their purpose in the loop. The transform loop of the FHT subroutine is divided into three different *do* loops. The first and last *do* loops over the variable k handle the transform for the zero frequencies or the Nyquist frequencies, respectively. For both of these two cases, the FHT double butterfly becomes a single FFT butterfly. The middle *do* loop over the variable *butloc* computes the double butterfly for the other frequencies. The double butterfly equations located in the inner *do* loop over the variable k are the following:

$$X_h(k) = X_{h1}(k) + \cos((2\pi/N)\,k)\,X_{h2}(k) + \sin((2\pi/N)\,k)\,X_{h2}(N-k) \quad (12a)$$

$$X_h(N-k) = X_{h1}(N-k) - \cos((2\pi/N)\,k)\,X_{h2}(N-k) + \sin((2\pi/N)\,k)\,X_{h2}(k) \quad (12b)$$

$$X_h(k+N) = X_{h1}(k) - \cos((2\pi/N)\,k)\,X_{h2}(k) - \sin((2\pi/N)\,k)\,X_{h2}(N-k) \quad (12c)$$

$$X_h(2N-k) = X_{h1}(N-k) + \cos((2\pi/N)\,k)\,X_{h2}(N-k) - \sin((2\pi/N)\,k)\,X_{h2}(k) \quad (12d)$$

where N is the length of the sequences $X_{h1}(k)$ and $X_{h2}(k)$, and k varies as

$$k = 1, 2, \cdots, N/2 - 1 .$$

When $k = 0$ or $k = N/2$, the equations of the double butterfly become

$$X_h(k) = X_{h1}(k) + X_{h2}(k) \quad (13a)$$

$$X_h(k) = X_{h1}(k) - X_{h2}(k) . \qquad (13b)$$

Note that the values of k in the above equations are consistent with the values of k used in equation (11) and are not consistent with Fortran subscripts. In other words, k in equations (12a-d) and (13a,b) starts at zero and goes up to $(N/2 - 1)$, while k in the FHT listing starts at one and goes up to $(N/2)$. In order to get the equivalent k subscripts for the program listings, add 1 to the values of k in the ranges for equations (12a-d) and equations (13a,b). The variable listings in Appendix C use the Fortran method of subscripting arrays.

Comparison Between the FFT and the FHT

There are two criteria by which the advantages and disadvantages of the FFT and the FHT are compared: execution speed and memory requirements. Clearly, the FHT algorithm requires less memory because all of the data are stored in arrays of real numbers. The FFT uses complex arrays, which require twice the memory space of a real array. In most geophysical processing systems, the computer has a large core memory, so computing speed is the most important factor. Most of the computations of the FFT and FHT transforms are done within the butterfly loops. For the listings in the Appendices, these loops are marked by comments. Any algorithm that reduces the number of additions and multiplications in these loops will reduce the overall computation speed. In the FFT butterfly loop, there is one multiplication and two additions of complex numbers, which adds up to four multiplications and six additions of floating point numbers for each iteration. The FHT butterfly loop has four multiplications and six additions of floating point numbers for each iteration. However, the butterfly loop for the FHT loops from 2 to $N/2$, while the butterfly loop for the FFT loops from 1 to N. Since the FHT loops half the number of times as the FFT, the FHT algorithm has two multiplications and three additions for every four multiplications and six additions of the FFT. The FHT has no multiplications for the zero and Nyquist frequencies, which may appear to be a savings. However, the FFT subroutine can be rewritten to eliminate multiplications for the zero and Nyquist frequencies. Based on the program listings, the FHT algorithm uses half of the floating point operations the FFT algorithm.

There are some FFT algorithms that are about the same speed as the FHT algorithm listed in the appendix. These FFT algorithms are fast because they pre-process the data by repacking the real numbers into complex numbers, thus cutting the length of the input sequence in half. The faster FFT algorithm then does a regular FFT over the shorter complex sequence and unpacks the result. A FFT will always do a certain number of complex multiplications and additions. When the real data compacted to complex form, the algorithm utilizes the required multiplications and additions more efficiently. Rick Ottolini and I did some time

comparisons of the Hartley transform and a fast FFT. In each case, the Hartley transform was faster by about five percent. When the FHT was compared to the FFT in Appendix A, the FHT was about fifty percent faster than the FFT.

Some Frequency Operations in Hartley Space

One of the advantages of computing in the frequency domain is that that the fewer operations are needed. For example, convolution is faster in the frequency domain than in the time domain. If the Hartley transform is to replace the Fourier transform, computations in Hartley space should take less time then in Fourier space.

Some of the important operations in the frequency domain involves either an even or an odd function. For example, a simple low pass filter is an even function in the frequency domain. The symmetry of even and odd functions will eliminate the number of floating point operations. In the comparisons below, the number of floating point operations for an even function and a general function are considered. The equations given below for even functions are the same equations for the odd function, so the number of computations are the same for even and odd functions.

In the following comparisons, two definitions are used. First, a complex multiplication involves four multiplications and two additions. Second, a memory access involves using computing an array index, locating a floating point number in an array, and then fetch the number from that location. This operation will involve at least one integer multiplication and addition as the computer calculates the address in an array. Once the address in an array is calculated, the time needed to access the next number in memory is very small. Therefore, a memory access of a complex number takes about the same amount of time as a memory access of a real number.

Convolution. The convolution formula in Fourier space is

$$W_f(k) = X_f(k)\, Y_f(k)\,. \tag{14}$$

Equation (14) involves four multiplications, two additions, and two memory accesses for each k. The convolution formula in Hartley space is

$$W_h(k) = 0.5 * \left[X_h(k)\,(Y_h(k) + Y_h(-k)) + X_h(-k)\,(Y_h(k) - Y_h(-k)) \right] \tag{15}$$

Equation (15) involves three multiplications, three additions, and four memory accesses for each k. Suppose that $Y_f(k)$ and $Y_h(k)$ are even functions. If $Y_f(k)$ is even, $Y_f(k)$ is a real function. So when $Y_f(k)$ is even, equation (14) involves two multiplications and two memory accesses. If $Y_h(k)$ is even, equation (15) becomes

$$W_h(k) = X_h(k)\, Y_h(k) \tag{15a}$$

which involves only one multiplication and two memory accesses. If memory access time is short, convolution of two sequences is faster in Hartley space.

Auto-correlation. The formula for computing the auto-correlation of a sequence in Fourier space is

$$W_f(k) = X_f(k)\, X_f^*(k) \tag{16}$$

where X^* is the complex conjugate of X. Equation (16) involves two multiplications, one addition, and one memory accesses for each k, where $k = 0, 1, \ldots, (N/2 - 1)$. The formula for auto-correlation in Hartley space is

$$W_h(k) = 0.5 * \left[X_h^2(k) + X_h^2(-k)\right] \tag{17}$$

Equation (17) involves three multiplications, one addition, and two memory accesses for each k, where $k = 0, 1, \ldots, (N/2 - 1)$. If $X_f(k)$ is even, then equation (16) involves one multiplication and one memory access. If $X_h(k)$ is an even sequence, then equation (17) becomes

$$W_h(k) = X_h^2(k) \tag{17a}$$

which requires one multiplication and one memory access. Therefore, auto-correlation is faster in Fourier space. If the input sequence is even, then auto-correlation takes the same amount of computation time in both spaces.

Cross-correlation. The formula for computing the cross-correlation of two sequences in Fourier space is given by

$$W_f(k) = X_f(k)\, Y_f^*(k)\,. \tag{18}$$

Equation (18) involves four multiplications, two additions, and two memory accesses for each k. The formula for cross-correlation in Hartley space is

$$W_h(k) = 0.5 * \left[X_h(k)\,(Y_h(k) + Y_h(-k)) - X_h(-k)\,(Y_h(k) - Y_h(-k)) \right] \tag{19}$$

Equation (19) involves three multiplications, three additions, and four memory accesses for each k. If $Y_f(k)$ in equation (18) is even, then equation (18) involves two multiplications and two memory accesses. If $Y_h(k)$ is even, then equation (19) becomes

$$W_h(k) = X_h(k)\, Y_h(k) \tag{19a}$$

which involves one multiplication and two memory accesses. Therefore, cross-correlation is

faster in Hartley space.

Hilbert Transform. The formula for computing the Hilbert transform in Fourier space is given by

$$W_f(k) = \begin{cases} i\, X_f(k) & k \geq 0 \\ -i\, X_f(k) & k < 0 \end{cases} \tag{20}$$

Equation (20) involves one memory access and an exchange of two numbers for each k. The formula for computing the Hilbert transform in Hartley space is given by

$$W_h(k) = \begin{cases} -X_f(-k) & k \geq 0 \\ X_f(-k) & k < 0 \end{cases} \tag{21}$$

Equation (21) involves two memory accesses and an exchange of two numbers for each k, where $k < 0$. Therefore, the Hilbert transform is faster in Hartley space than in Fourier space because Hartley space requires half of the exchanges needed in Fourier space.

All of the above operations counts are based on the equations. Actual computer subroutines might involve more or fewer multiplications or memory accesses, depending on the method. Based on the equations alone, operations in Hartley space are faster or use the same time as operations in Fourier space. The only exception is auto-correlation of non-even functions, which is faster in Fourier space.

The FHT Algorithms in a Paging Operating System

Another factor which affects the performance of an algorithm is the size of the data array that must be in the computer's memory at once. Many modern computer operating systems use paging from the disk system to allow more separate processes to reside in memory. When a program is executing, the program source and its variables are stored in units of memory called pages. During execution, the program may not need all of its variables stored at once in memory. The operating system detects this and may store some of the pages of memory that the program is not using on disk. When program tries to access an address of a page that is on the disk, the program generates a page fault. The operating system detects the page fault, reads the missing page from disk into memory, and lets the program continue from where it stopped. The operating system starts to read and write pages when a large number of processes are running at the same time in computer's memory. When there is a high demand for memory, the operating system will record memory pages that have not been access recently onto the disk. In other words, the operating system will write the oldest memory page that has not been accessed by any program. If the program needs that page,

the operating system has to read the page back from the disk. If a program can keep its memory access confined to a few pages for long periods of time, the program will be less susceptible to page faults.

Consider two programs that are running on a busy system. Both programs have to process a large array of numbers that require three pages of memory storage. Program A has an algorithm that starts at one end of its array and processes each number serially until it reaches the other end. Program A does not need the entire array in memory at once, so two pages of its array can be kept on disk. During program A's execution, at most two disk reads and writes are needed by program A. Program B has an algorithm that uses numbers from all parts of its array. Therefore, program B needs at all three pages of its array in memory at once. In a very busy system, the operating system may remove one of program B's pages from memory if program B has not accessed the page very recently. When program B needs to access a number in a page that is on disk, program B has to wait until the system reads the page back in. Therefore, program B is more likely to require more disk reads and writes to finish its job.

Operations like convolution in Hartley space behave like program B because both the positive and negative frequencies are accessed from opposite ends of the array at the same time. Convolution in Fourier space behaves like program A because only works on one section of an array at a time. A convolution program that uses the algorithm for Hartley space is more likely to generate page faults than a program using the convolution algorithm in Fourier space. Most programs that use any routines in Hartley space will require more disk read and write time if the computer is very busy and there is a high demand for memory. One method to prevent page faults is to interleave the positive and negative frequency components in the array. A convolution program would not have to access both end of the array since the negative frequency component is next to the positive component. However, reordering the array requires some extra computation time.

Two Dimensional Hartley Transforms

Clayton (1978) outlined an algorithm for calculating the two-dimensional Fourier transform of a data set that is too large to fit in the computer's memory and too large to transpose easily. For example, consider a two-dimensional data set that has 256 columns with 2048 elements in each column. The first part of Clayton's algorithm read in a column, did the FFT, and then wrote the column back out. This first part of the algorithm would be repeated for each column, or 256 times. Clayton's algorithm would then start Fourier transforming over the columns by reading in the columns necessary to make a FFT butterfly,

apply a FFT butterfly to each element in the columns, and then write the resulting columns. The second part would be repeated until all of the FFT butterflies are done. Appendix D has a Ratfor program that follows Clayton's two-dimensional FFT algorithm for a large data set.

A similar algorithm can be used to form a two-dimensional Hartley transform. The algorithm would read in a column, do a FHT, and write the column back to disk. The second part of the algorithm would read in the columns needed for a double butterfly, apply a FHT butterfly to each element in the columns, and write the resulting columns. The second part would be repeated until the FHT over the columns is done. Appendix E has a Ratfor program that does the two-dimensional FHT in a similar manner to the routine in Appendix D.

Both the 2D-FFT and the 2D-FHT algorithms require the same amount of memory. The 2D-FFT requires two complex arrays to store its columns, while the 2D-FFT requires four real arrays to store its columns. Both algorithms used the same number of reads and writes from disk. For a 16-point transform, the butterfly stage of both algorithms requires a total of 128 reads and writes. However, the 2D-FHT only reads in or writes half of the amount of data as the 2D-FFT. Since the disk reads and writes will take the most time for both algorithms, the 2D-FHT is faster than the 2D-FFT.

Overall Rating of the Hartley Transform

The Hartley transform offers the following advantages:

1. The fast Hartley transform needs fewer multiplications and additions than the corresponding fast Fourier transform. Therefore, the fast Hartley transform takes less computing time than the fast Fourier transform.
2. Convolution, cross-correlation, and the Hilbert transform require fewer multiplications and additions in Hartley space than in Fourier space.
3. The Hartley transform needs less memory to store numbers than the Fourier transform because the Hartley transform does not use complex numbers. The results of the Hartley transform can be stored in the same amount of space as the original data set, thus eliminating the need to allocate more disk or tape space.
4. Deriving operations for Hartley space is simple. Equations (1) and (2) can convert formulas from Fourier space to Hartley space and back. Any operations that are formulated for Fourier space can be translated directly into Hartley space using equation (1).

5. Numbers can be switched from Hartley space to Fourier space and back. Equations (1) and (2) require one addition, two memory access, and maybe one multiplication to convert data from one space to the other. Therefore, operations which are faster in Fourier space, like the auto-correlation, can be done in Fourier space. A program converts the numbers from Hartley space to Fourier space, applies the operation, and converts the numbers back to Hartley space.
6. Several users here at the SEP have commented that they find frequency domain operations easier to think about in Hartley space. For example, they do not have to worry about the sign of the Fourier kernel in the FFT. Program coding is also easier because all of the numbers are real instead of complex, and no complicated indexing scheme is needed.
7. The two dimensional fast Hartley transform needs the same number of disk reads and writes as the two dimensional fast Fourier transform. However, the Hartley transform requires shorter reads and writes, which may save time.

The Hartley transform has the following disadvantages:

1. Any time and memory savings is gone when the Hartley transform is used on complex data. If the time domain data is complex, then the Hartley domain data is complex also. Complex data is transformed to Hartley space by running the Hartley transform twice, once for the real numbers and once for the complex numbers. The time required to do two Hartley transforms on complex data is the same as doing one Fourier transform on the same data.
2. Some fast Fourier transform algorithms use special packing techniques to run at the same speed and use the same amount of memory as the fast Hartley transform. These algorithms repack the complex data by removing the redundant information, resulting in real numbers. The packing usually involves ignoring the negative frequencies and keeping only the positive frequencies. Like the Hartley transform, the faster Fourier transforms can only transform real data. Any time savings of the faster Fourier transforms is gone when the input data is complex.
3. Auto-correlation is slower in Hartley space than in Fourier space.
4. Operations in Hartley space will take more time for memory accesses in a computer operating environment where there is a great deal of paging. If there is not a large demand for core memory, paging will not affect the memory access time.

Based on the above observations and comparisons, the Hartley transform appears to be a very good method of transforming data into the frequency domain.

REFERENCES

Claerbout, Jon, 1976, *Fundamentals of geophysical data processing:* San Francisco, McGraw-Hill Book Company.

Clayton, Robert, 1978, Two-Dimensional Fourier transforms without transposing, *SEP Report #15,* p. 247.

Ottolini, Richard, 1984, Migration by Hartley transform, *SEP Report #39.*

Rabiner, Lawrence R. and Gold, Bernard, 1975, *Theory and application of digital signal processing:* Englewood Cliffs, New Jersey, Prentice-Hall, Inc.

Appendix A - Ratfor Listing of Fast Fourier Transform Subroutine

```
      # A ratfor subroutine to do the one dimensional fast Fourier transform
#     len - length of input array
#     cx - array containing input data (complex)
#     signi - indicates whether the transform is forward or not
#
subroutine fft(len, cx, signi)
complex cx(len), carg, ctemp, cu, cw
integer len, k, jj, butloc, n, istep
real signi, scale
#
scale = sqrt (1./len)
jj = 1
do k = 1, len {                                          # Bit Reversal Loop
            if (k <= jj) {                               #
                        ctemp = cx(jj)*scale             #
                        cx(jj) = cx(k)*scale             #
                        cx(k) = ctemp                    #
            }                                            #
            butloc = len/2                               #
            while ((jj > butloc) && (butloc >= 1))       #
            {                                            #
                        jj = jj - butloc                 #
                        butloc = butloc/2                #
            }                                            #
            jj = jj + butloc                             #
}                                                        #
#
n = 1
carg = (0., 1.)*3.14159265*signi
repeat {
            istep = 2*n
            cu = cexp (carg)
            carg = carg/2.
            cw = (1., 0.)
            do butloc =  1, n {
                        do k = butloc, len, istep {              # FFT butterfly
                                    ctemp = cw*cx(k + n)         #
                                    cx(k + n) = cx(k) - ctemp    #
                                    cx(k) = cx(k) + ctemp        #
                        }                                        #
                        cw = cw*cu
            }
            n = istep
} until (n >= len)
return
end
```

Appendix B - Ratfor Listing of Fast Hartley Transform Subroutine

```
      # Subroutine to do fast Hartley transform of the array of real numbers fx.
# len is the length of the array. Note that len should be a power of two.
#
subroutine fht(len, fx)
integer len, k, butdis, butloc, n, istep
real fx(len), scale, temp1, temp2, fcos, fsin, dsin, dcos, arg
#
arg = 1./len
scale = sqrt(arg)
butdis = 0
do k = 1, len {                                                    # Bit Reversal Loop
        if (k <= butdis) {                                         #
                temp1 = fx(butdis)*scale                           #
                fx(butdis) = fx(k)*scale                           #
                fx(k) = temp1                                      #
        }                                                          #
        butloc = len/2                                             #
        while (butdis > butloc && butloc >= 1) {                   #
                butdis = butdis - butloc                           #
                butloc = butloc/2                                  #
        }                                                          #
        butdis = butdis + butloc                                   #
}                                                                  #
arg = 3.14159265359
n = 1
repeat {
        istep = 2*n
        dcos = cos (arg);               dsin = sin (arg)
        arg = arg/2.
        fcos = dcos;            fsin = dsin
        do k = 1, len, istep {                                     # Zero Frequency Loop
                temp1 = fx(k + n)                                  #
                fx(k + n) = fx(k) - temp1                          #
                fx(k) = fx(k) + temp1                              #
        }                                                          #
        if (n > 2) {
                butdis = n - 2
                do butloc = 2, n/2 {
                        do k = butloc, len, istep {                # Double Butterfly
                                temp1 = fcos*fx(k + n) + fsin*fx(k + n + butdis)
                                temp2 = fsin*fx(k + n) - fcos*fx(k + n + butdis)
                                fx(k + n) = fx(k) - temp1
                                fx(k + n + butdis) = fx(k + butdis) - temp2
                                fx(k) = fx(k) + temp1
                                fx(k + butdis) = fx(k + butdis) + temp2
                        }                                          #
                        temp1 = fcos*dcos - fsin*dsin
                        fsin = fsin*dcos + fcos*dsin
                        fcos = temp1
                        butdis = butdis - 2
                }
```

```
        }
        if (n > 1)
                do k = n/2 + 1, len, istep {                    # Nyquist Frequency Loop
                        temp1 = fx(k + n)                       #
                        fx(k + n) = fx(k) - temp1               #
                        fx(k) = fx(k) + temp1                   #
                }                                               #
        n = istep
} until (n >= len)
return
end
```

Appendix C - List of Variables in Fast Hartley Transform Subroutine

The Ratfor variables defined in this appendix refer to the FHT listing in Appendix B. Figure C-1 shows two of the double butterflies used in a 16-point FHT. The figure shows the stage when the algorithm is combining four 4-point sequences into two 8-point sequences. The terms defined below use figure C-1 for an example. Note that X_h in figure C-1 starts at 1 instead of 0. This difference reflects the fact that the variables below are from a Fortran program, where arrays start at 1.

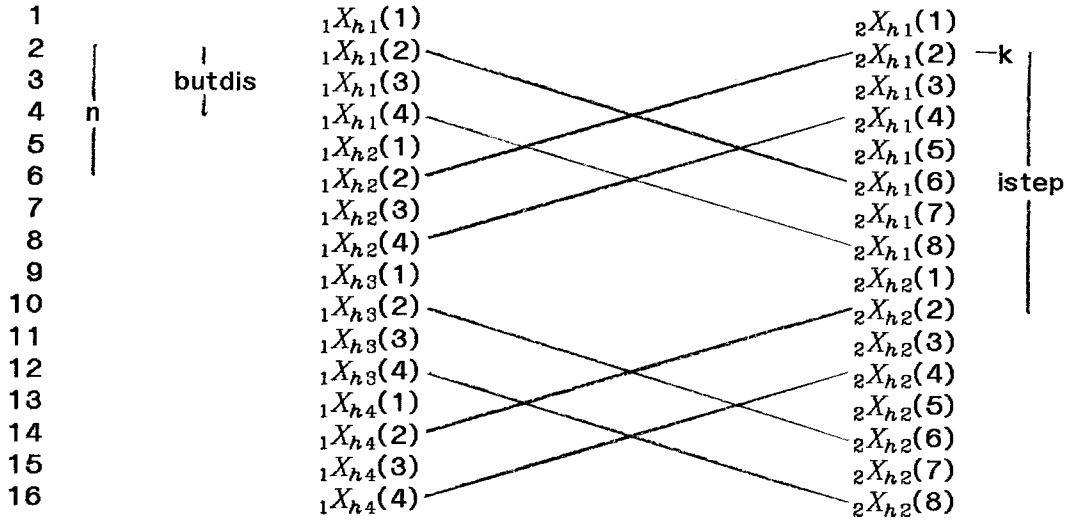

FIG. C-1. This figure shows two intermediate double butterflies for a 16-point Hartley transform. Data flows from the left side to the right.

n. This variable is the length of the two sequences that are combined into one sequence. In figure C-1, four 4-point sequences are combined into two 8-point sequences. For this case, n is equal to 4. n is initialized to 1, and is doubled after each pass through the *repeat* loop.

istep. This variable is the distance between groups of butterflies and is always twice the value of n. In figure C-1, the top of the top double butterfly is 8 points away from the top of the bottom butterfly. Therefore, *istep* in figure C-1 is 8.

k. This variable gives the array address of the top of a double butterfly. k starts at 1 and increments by *istep* until it reaches the end of the sequence. In figure C-1, k would have two values. For the top butterfly, k is 2, while for the bottom butterfly, k is 10.

butdis. A double butterfly consists of two butterflies. *butdis* contains the distance between these individual butterflies within a double butterfly. In figure C-1, *butdis* is 2 for both double butterflies. *butdis* is initialized to n -2 and is decremented by two until it reaches zero. *butdis* reaches zero at the Nyquist frequency.

butloc. A block is the longer sequence that the transform is combining data for. In figure C-1, the FHT is combining two 4-point sequences into one 8-point sequence or block. On the right side of figure C-1, there are two blocks; each block contains a 8-point sequence. On the left side of figure C-1, there are also two block, but each block contains two 4-point sequences. *istep* contains the length of a block. *butloc* gives the location of the top of the double butterfly with respect to the top of a block. In figure C-1, *butloc* is 2 because the top of butterfly starts at the second element of each block. *butloc* starts at 2 and increments by one up to $n/2$.

The rest of the variables in the *repeat* loop keep track of the sin and cos factors used in the double butterfly.

Appendix D - Ratfor Listing of a Two-Dimensional FFT

The subroutine below calls two subroutines called *cread* and *cwrite*. These two subroutines read or write complex vectors and are site dependent. The two subroutines should have the ability to go to any location in a file and read or write data from that location. These two subroutines have four arguments:

1. *dfile* is the file descriptor of the file that contains the data. This is the number returned by the system when the file is opened. The program should be able to read and write to this file.
2. *ca* and *cb* are the complex arrays of length *leny*. The data is stored in these two arrays after reading or before writing.
3. The third argument gives the vector number that will be accessed next in the file. The subroutine should compute the address in the file from the third argument.
4. *leny* is the length of the vector to read in. This value is used in computing the location of the vector in the file.

```
      # A ratfor subroutine to do two-dimensional fast Fourier transforms without
# transposing the data.
#    dfile - file descriptor
#    leny - length of columns (fast direction)
#    lenx - length of rows (slow direction)
#    signx   -1 for a reverse Fourier transform
#    signy    1 for a forward Fourier transform
#    ca, cb - two scratch arrays that can hold a column
#
subroutine fft2d(dfile, leny, lenx, signy, signx, ca, cb)
complex ca(leny), cb(leny), carg, ctemp, cu, cw
integer dfile, k, ii, jj, butloc, n, istep
real signy, signx, scale
#
scale = sqrt(1./lenx)
# Do the transform over the columns first
do ii = 1, lenx {
            call cread(dfile, ca, ii, leny)
            call fft(leny, ca, signy)
            do jj = 1, leny
                        ca(jj) = ca(jj)*scale
            call cwrite(dfile, ca, ii, leny)
}
# Do the row FFTs
jj = 1
do k = 1, lenx {                                          # Bit Reversal Loop
            if (k <= jj) {                                #
                        call cread(dfile, ca, jj, leny)   #
                        call cread(dfile, cb, k, leny)    #
                        call cwrite(dfile, ca, k, leny)   #
```

```
                call cwrite(dfile, cb, jj, leny)          #
        }                                                 #
        butloc = lenx/2                                   #
        while ((jj > butloc) && (butloc >= 1))            #
        {                                                 #
                jj = jj - butloc                          #
                butloc = butloc/2                         #
        }                                                 #
        jj = jj + butloc                                  #
}                                                         #
#
n = 1
carg = (0., 1.)*3.14159265*signx
repeat {
        istep = 2*n
        cu = cexp(carg)
        carg = carg/2.
        cw = (1., 0.)
        do butloc =  1, n {
                do k = butloc, lenx, istep {              # FFT butterfly
                        call cread(dfile, ca, k, leny)
                        call cread(dfile, cb, k + n, leny)
                        do ii = 1, leny {
                                ctemp = cb(ii)*cw
                                cb(ii) = ca(ii) - ctemp
                                ca(ii) = ca(ii) + ctemp
                        }
                        call cwrite(dfile, ca, k, leny)
                        call cwrite(dfile, cb, k + n, leny)
                }                                         #
                cw = cw*cu
        }
        n = istep
} until (n >= lenx)
return
end
```

Appendix E - Ratfor Listing of a Two-Dimensional FHT

The subroutine below calls two subroutines called *rread* and *rwrite*. These two subroutines read or write real vectors and are site dependent. The two subroutines should have the ability to go to any location in a file and read or write data from that location. These two subroutines have four arguments:

1. *dfile* is the file descriptor of the file that contains the data. This is the number returned by the system when the file is opened. The program should be able to read and write to this file.
2. *a*, *b*, *c*, and *d* are the real arrays of length *leny*. The data is stored in these four arrays after reading or before writing.
3. The third argument gives the vector number that will be accessed next in the file. The subroutine should compute the address in the file from the third argument.
4. *leny* is the length of the vector to read in. This value is used in computing the location of the vector in the file.

```
# Subroutine to do the two-dimensional fast Hartley transform without
# transposing the data.
#       dfile - file descriptor
#       leny - length of columns (fast direction)
#       lenx - length of rows (slow direction)
#       a, b, c, d - scratch arrays that can hold a column.
#
subroutine fht2d(dfile, leny, lenx, a, b)
integer ii, k, butdis, butloc, n, istep, dfile, lenx, leny
real a(leny), b(leny), c(leny), d(leny)
real scale, temp1, temp2, fcos, fsin, dsin, dcos, arg
#
arg = 1./lenx
scale = sqrt(arg)
# Do the column transforms first
do ii = 1, lenx {
        call rread(infile, a, ii, leny)
        call fht(leny, a)
        do k = 1, leny
                a(k) = a(k)*scale
        call rwrite(dfile, a, ii, leny)
}
# Do the row transforms now
butdis = 0
do k = 1, len {                                         # Bit Reversal Loop
        if (k <= butdis) {                              #
                call rread(dfile, a, butdis, leny)      #
                call rread(dfile, b, k, leny)           #
                call rwrite(dfile, a, k, leny)          #
                call rwrite(dfile, b, butdis, leny)     #
```

```
            }                                                   #
            butloc = len/2                                      #
            while (butdis > butloc && butloc >= 1) {            #
                        butdis = butdis - butloc                #
                        butloc = butloc/2                       #
            }                                                   #
            butdis = butdis + butloc                            #
}                                                               #
arg = 3.14159265359
n = 1
repeat {
            istep = 2*n
            dcos = cos (arg);               dsin = sin (arg)
            arg = arg/2.
            fcos = dcos;            fsin = dsin
#
            do k = 1, len, istep {                              # Zero Frequency Loop
                        call rread(dfile, a, k, leny)
                        call rread(dfile, b, k + n, leny)
                        do ii = 1, leny {
                                    temp1 = b(ii)
                                    b(ii) = a(ii) - temp1
                                    a(ii) = a(ii) + temp1
                        }
                        call rwrite(dfile, a, k, leny)
                        call rwrite(dfile, b, k + n, leny)
            }                                                   #
#
            if (n > 2) {
                        butdis = n - 2
                        do butloc = 2, n/2 {
                                    do k = butloc, len, istep {             # Double Butterfly
                                                call rread(dfile, a, k, leny)
                                                call rread(dfile, b, k + butdis, leny)
                                                call rread(dfile, c, k + n, leny)
                                                call rread(dfile, d, k + n + butdis, leny)
                                                do ii = 1, leny {
                                                            temp1 = fcos*c(ii) + fsin*d(ii)
                                                            temp1 = fsin*c(ii) - fcos*d(ii)
                                                            c(ii) = a(ii) - temp1
                                                            d(ii) = b(ii) - temp2
                                                            a(ii) = a(ii) + temp1
                                                            b(ii) = b(ii) + temp2
                                                }
                                                call rwrite(dfile, a, k, leny)
                                                call rwrite(dfile, b, k + butdis, leny)
                                                call rwrite(dfile, c, k + n, leny)
                                                call rwrite(dfile, d, k + n + butdis, leny)
                                    }                                       #
                                    temp1 = fcos*dcos - fsin*dsin
                                    fsin = fsin*dcos + fcos*dsin
                                    fcos = temp1
                                    butdis = butdis - 2
                        }
```

```
        }
#
        if (n > 1)
                do k = n/2 + 1, len, istep {                    # Nyquist Frequency Loop
                        call rread(dfile, a, k, leny)
                        call rread(dfile, b, k + n, leny)
                        do ii = 1, leny {
                                temp1 = b(ii)
                                b(ii) = a(ii) - temp1
                                a(ii) = a(ii) + temp1
                        }
                        call rwrite(dfile, a, k, leny)
                        call rwrite(dfile, b, k + n, leny)
                }                                               #
        n = istep
} until (n >= len)
return
end
```