Matrix transposition and the 2-D FFT revisited

Stewart A. Levin

Introduction

A simple, direct way to write an F-K filter program is to take an input nx by nt matrix and Fourier transform (FFT) each trace to produce an nx by nf matrix. Then transform each monofrequency slice to produce an nk by nf matrix that you filter by, for example, zeroing the values in a quadrant. Now reverse your steps and inverse transform to an nx by nf and then to an nx by nt matrix to obtain the filtered result.

This was done a few years ago at a contracting company I know and worked fine until one day it was applied to a stacked section (a single 1000 trace file) instead of unstacked CMP gathers (e.g. a thousand 24 trace records). For twelve hours, until the job was cancelled, system throughput came to a standstill!

Analysis of system activity records and core dumps showed the cause to be extremely high paging[1] activity during the x to k transform. A bit of thought showed why: the nx samples of a monofrequency slice were internally separated by nf words of computer memory rather than being adjacent. This forced the system to fetch a new page for each sample, about a *thousand times* more often than if the samples had been consecutive!

So here is the problem - even though the matrix fit in *virtual* core it didn't fit in the available *real* computer memory.

[1] In today's modern computer an executing program resides primarily on disk and only a limited number of segments or *pages* of it are in the computer's memory at any one time. The operating system automatically updates old pages on disk and *swaps in* other pages from disk as the program references different data areas of its "virtual core".

Matrix transposition

One solution to this problem is to draw on sorting methods used to reorder extremely large data files. Typically one sorts subsets of the data and then successively merges the subsets to finish the sort. In seismic data, each trace is already sorted in the order each element should appear in the output matrix and we need only implement a merging phase. This is precisely the idea presented by Claerbout (1977) where repeated merging is used to "magically" shuffle (sort) a card deck.

To use this "trick" in a general framework, we first have to understand how a merge operates on data matrices. So suppose we have two traces of data and we interleave (merge) them. We have, in effect, *transposed* a $2 \times n$ array to an $n \times 2$ array.

$$\begin{array}{cccccc|cccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 6 & 1 & 7 & 2 & 8 & 3 & 9 & 4 & 10 & 5 & 11 \end{array}$$

If we had, instead, six traces, we could interleave six ways and convert a $6 \times n$ matrix to an $n \times 6$ transpose. Alternatively, we could first merge traces 0 & 3, 1 & 4, and 2 & 5 to form three double length traces and then merge these together to get the same final result. In this way we have treated the $6 \times n$ input as a $3 \times 2 \times n$ array which is successively transformed to a $2 \times n \times 3$ array and finally to an $n \times 3 \times 2$ or $n \times 6$ output.

Here we see how *factoring* one of the matrix dimensions can reduce a large transpose to a succession of smaller merges. In general, if we factor the number of traces nx into $n_1 \times n_2 \times \ldots n_k$ we can transpose an $nx \times nt$ matrix in k stages or passes:

$$\begin{array}{l} \quad nx \qquad \times nt \\ n_1 \times n_2 \times \ldots \times n_k \times nt \\ n_2 \times n_3 \times \ldots \times n_k \times nt \times n_1 \\ n_3 \times n_4 \times \ldots \times n_k \times nt \times n_1 \times n_2 \\ \qquad \vdots \\ n_k \times nt \times n_1 \times \ldots \times n_{k-1} \\ nt \times n_1 \times n_2 \times \ldots \times n_k \\ nt \times \qquad nx \end{array}$$

Reversing the data flow, we discover how a sequence of "unmerges" can also transpose a large data array. The one to choose in any given application is the one that minimizes the number of I/O operations. With a bit of care, we can make the number of I/O operations per pass stay within a few percent of the desired minimum value of the total number of data bytes divided by the physical blocking factor of the device on which the data is stored.

Thus minimizing the number of I/O operations is equivalent to minimizing the number of passes over the data.

So, to design an optimal transpose we should select the largest factors possible in order to get the fewest number of them. If we find that nx has the fewest factors we would use our original successive merge scheme. Otherwise an unmerging algorithm is more appropriate.

What limits the size of factors? Our F-K filter example tells us immediately that it is the amount of real core available to store I/O blocks (or pages) of data to be merged. So, if we have enough area for m physical blocks (typically of length 512 bytes or some multiple thereof), our factors cannot be chosen larger than m (actually $m-1$ since at least one block is reserved for output and so cannot be used for an input factor and vice-versa).

Now we can describe out current transpose program, due to Rob Clayton, and see how close to optimal it is. Rob, depending upon the dimensions of the matrix, uses either the sequence

Algorithm 1

$$
\begin{aligned}
&nx \qquad \times nt \\
&n_1 \times n_2 \times \ldots \times n_k \times nt \\
&n_k \times n_1 \times n_2 \times \ldots \times n_{k-1} \times nt \\
&n_{k-1} \times n_1 \times n_2 \times \ldots \times n_{k-2} \times nt \times n_k \\
&n_{k-2} \times n_1 \times n_2 \times \ldots \times n_{k-3} \times nt \times n_{k-1} \times n_k \\
&\qquad\qquad \vdots \\
&n_2 \times n_1 \times nt \times n_3 \times \ldots \times n_k \\
&n_1 \times nt \times n_2 \times n_3 \times \ldots \times n_k \\
&nt \times n_1 \times n_2 \times \ldots \times n_k \\
&nt \times \qquad nx
\end{aligned}
$$

or the sequence

Algorithm 2

$$
\begin{aligned}
&nx \times \qquad nt \\
&nx \times n_1 \times n_2 \times \ldots \times n_k \\
&n_k \times nx \times n_1 \times n_2 \times \ldots \times n_{k-1} \\
&n_{k-1} \times n_k \times nx \times n_1 \times \ldots \times n_{k-2} \\
&\qquad\qquad \vdots \\
&n_2 \times n_3 \times \ldots \times n_k \times nx \times n_1 \\
&n_1 \times n_2 \times n_3 \times \ldots \times n_k \times nx \\
&\qquad nt \qquad \times nx
\end{aligned}
$$

Both are implemented with n_{k-j} input buffers and n_{k-j-1} output buffers at the j'th stage ($n_{k+1}=n_0=1$). This scheme, therefore, requires that successive factors sum to no more than the maximum number, MAXBLK (set to 300 in the program), of physical (512 byte) blocks. Elementary calculus will show that the product of these factors will be maximized when n_j = n_{j-1} = MAXBLK/2. This is just over half of the MAXBLK-1 theoretical limit per factor previously derived and so generally one unnecessary pass over the data for each $\log_2$ MAXBLK number of factors is incurred. (This inefficiency was later corrected by changing the first algorithm to be the reverse of the second and assigning only one buffer to the output (resp. input) file during each pass.)

Another question of interest is how much auxiliary space is needed to transpose the data? With the present scheme we need enough room to hold the original data plus up to MAXF additional partial blocks where MAXF is the largest factor in the transpose sequence. Because we flip the data back and forth between two work units we require, in total, twice this amount of scratch work disk space. When would it be possible to transpose "in place" and overwrite output blocks on previously read input blocks? When would it be desirable?

Clearly it is possible to overwrite input with output. Suppose we have $n \times m$ blocks of data input to the pass for factor n. Then we can treat the input blocks as elements of an $n \times m$ matrix of blocks accessed sequentially. The output is an $m \times n$ matrix of merged blocks to be overwritten on the input. Thus block $I=i_1+i_2 n$ of the output will be overwritten on block $I'=i_2+i_1 m$ of the input. But I' = mI mod$(mn-1)$ as mI = $i_1 m+i_2 mn$ = $i_2+i_1 m+i_2(mn-1)$. Thus to find where to store successive output blocks we need only increment a counter by m mod$(mn-1)$. The only ambiguities are the first and last blocks which stay where they are. Of course I' itself is in some scrambled order so we would follow the chain back through each previous stage to find where to read and write blocks with respect to the original sequential file.

One special case which greatly simplifies in-place transposition is when the dimension not factored is divisible into an integral number of blocks, e.g. a multiple of $512/ulen$ on our system where $ulen$ is the number of bytes per element of the data. In this case there are no partial blocks to worry about and the number of blocks, D, of data stays the same in every pass. Here, then, the input to the j'th stage is indexed by $m_1 \times m_2 \times \cdots \times m_{j-1}$ mod $(D-1)$ and the output by $m_1 \times m_2 \times \cdots \times m_j$ mod $(D-1)$. Thus we need only keep $m_1 \times m_2 \times \cdots \times m_{j-1}$ mod $(D-1)$ as the increment of a counter modulo $D-1$. This is the idea behind a clever field demultiplexing system patented not long ago by Carl Savit of Western Geophysical.

On the other hand it may be needless to transpose in-place. Typically one has the input data on one disk file and the output on another. Thus we already have two data-sized

areas with which we can work. I can think of only two situations in which we wouldn't use the input/output areas for intermediate results:

1. If one or both files are tape devices.
2. If raw disk I/O [2] is desired.

In both these cases it would be advantageous to transpose in-place *if there were insufficient work space for two copies of the data.*

Another interesting question is whether, by padding additional dummy traces and/or samples, the transposition can be shortened? What is required is the number of passes to be reduced by one (or more) without increasing (by padding) the total number of blocks read/written in each pass so much as to offset the savings. We can set approximate bounds for padding by assuming D blocks are read/written in each of nf passes before padding and $D+E$ read and written in $nf-1$ passes after padding. This leads to the condition $D>E\times(nf-1)$.

Finally, a couple of random notes. On some machines memory is *interleaved* perhaps 2,4 or 8 ways. In this case it is fastest to use odd increments (1,3,5, etc.) in moving data to minimize memory conflict which slows access time. In our program this might be done by offsetting I/O buffers by 1 doubleword. For our machine this is not done because it has the drawback of reducing the number of 512 byte blocks the high speed cache will hold.

Another note is that the Unix® *popen* and *pclose* subroutines permit us to transpose data already in core without having to first write it to disk, transpose it separately, and then read it back from disk. This also allows us to transpose more than one set of data simultaneously by opening two or more pipes to our *transp* command.

Another approach

Matrix transposition using the optimized methods just described is not the only solution to Fourier transforming large data sets efficiently. However, in order to optimize Fast Fourier Transform (FFT) algorithms for large amounts of data, such as two and three dimensional seismic datasets, an intimate understanding of the internal computations and data shuffling of the basic FFT algorithms is required. The author knows from experience that this can be a trial and so we digress to explicate the method based upon a unified approach to the FFT

[2] Our operating system permits circumvention of normal system buffering and device access. Such "raw I/O" is less flexible but more efficient than normal "cooked I/O".

found in Rabiner and Gold (1975).

The 1-D FFT

The FFT algorithm (actually there are several very closely related ones) uses a method of divide and conquer to compute the **discrete Fourier transform** (DFT)

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-2\pi i k n / N} \qquad k = 0, 1, \ldots, N-1 \tag{1}$$

of an N point sequence $x(n)$. The most common case is when N is a power of two. Here let W_N represent the phase rotation $exp(-2\pi i/N)$ and write, following Oppenheim and Schafer (1975),

$$X(k) = \sum_{n \text{ even}} x(n) W_N^{nk} + \sum_{n \text{ odd}} x(n) W_N^{nk} \tag{2}$$

$$= \sum_{r=0}^{N/2-1} x(2r) W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1) W_N^{(2r+1)k} \tag{3}$$

$$= \sum_{r=0}^{N/2-1} x(2r) W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1) W_{N/2}^{rk} \tag{4}$$

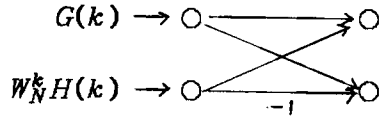
$$= G(k) + W_N^k H(k) \tag{5}$$

where we recognize $G(k)$ and $H(k)$ as $N/2$ point DFT's. Actually these shorter DFT's are defined for the range $k = 0, 1, \ldots, N/2-1$ but we leave it to the interested reader to verify that they extend by periodicity to other values of k. Therefore, noting $W_N^{N/2} = -1$, we have the following prescription

$$X(k) = G(k) + W_N^k H(k) \qquad k = 0, 1, \ldots, N/2-1 \tag{6a}$$

$$X(k+N/2) = G(k) - W_N^k H(k) \qquad k = 0, 1, \ldots, N/2-1 \tag{6b}$$

This is called a *butterfly* because a signal flow diagram



takes the form of a (stylized) butterfly.

The FFT algorithm applies this splitting recursively to generate the two $N/2$ point

DFT's from 4 $N/4$ point DFT's, etc. For $N = 8$ the signal flow diagram takes the form

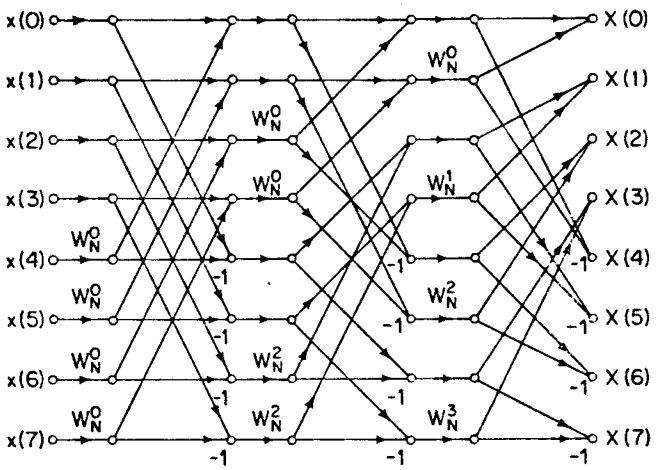


Fig. 6.13 Rearrangement of Fig. 6.10 with both input and output in normal order.

which is often rearranged to an "in-place" form such as

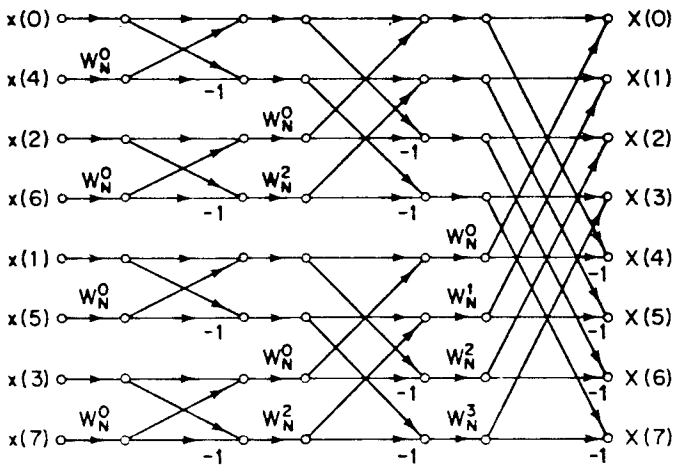


Fig. 6.10 Flow graph of eight-point DFT using the butterfly computation of Fig. 6.9.

or

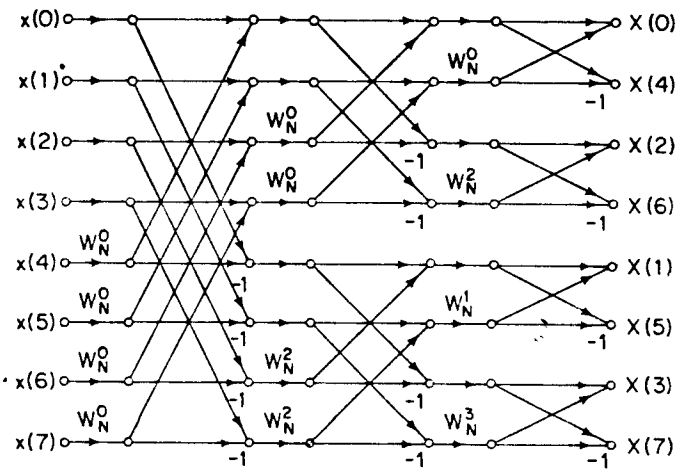


Fig. 6.12 Rearrangement of Fig. 6.10 with input in normal order and output in bit-reversed order

These flow diagrams point up an interesting complexity to in-place processing: the output sequence is scrambled with respect to the input order. This reordering is known in the trade as *bit reversal.* Furthermore the W_N^k *twiddle factors* are also in a similarly scrambled order. Conversely, the in-place diagram highlights a desirable feature of the algorithm we will want to exploit later: where the algorithm has two inputs to one stage of the computation, the right choice of four inputs can be carried through two stages; by extension the right choice of eight inputs cascades through three stages, etc.

This successive subdivision offers significant computational savings with $O(N \log N)$ computations as compared to $O(N^2)$ for direct DFT evaluation. Rabiner and Gold cite a two order of magnitude improvement for $N = 1024$.

The 2-D cross-vector FFT

Two (and higher) dimensional discrete Fourier transforms can also be evaluated with the aid of the FFT algorithm by transforming all rows and then transforming the resulting columns (or vice versa). As Oppenheim and Schafer point out, a fundamental difficulty with multidimensional transforms is that the amount of data generally exceeds available random-access memory. Indeed this is why we just looked at matrix transposition - one solution is to transform the columns of a data array on disk, transpose the data using the (clever) algorithm I outlined, and then transform the new columns (which are the rows of the untransposed matrix). A different method is presented in SEP-15 by Clayton (1978) in which the (harder) row transforms are performed by transposing the FFT algorithm rather than the data matrix. In other words, one places a column vector at each input node of the butterfly, rather than a single number, and obtains a corresponding column vector at each output node.

It is the implementation of this second method that we now address. As most array processors have highly efficient microcoded 1-D FFT algorithms, our goals are to answer the following questions:

1. How can we minimize I/O load in the algorithm?
2. How can we increase parallelism in the approach to offset the speed advantage of array processor FFT's?
3. Having optimized as much as possible, does the vector FFT emulation save enough time by avoiding matrix transposition to make up the difference?

To put these questions in perspective, we emphasize that both FFT via matrix transposition and the transposed FFT algorithm require, for a *square* $N \times N$ matrix, $O(N^2 \log N)$ I/O

operations to implement the same arithmetic computations (albeit in different order). Therefore careful attention to constants of proportionality is needed to compare such algorithms. For this reason, the first two questions are of extreme practical importance. The matrix transpose algorithm has to move elements around to resort them - a liability - but, for non-square arrays, reduces the number of I/O passes over the data by using the smaller of the matrix dimensions; the transposed FFT reckons only on the number of columns. Also, there may be overriding requirements, such as a need to have the transform output in a transposed format for later calculations, which make matrix transposition the better choice regardless of the relative efficiency of the two approaches.

Let us first consider the speed advantage of array processor FFT's and whether it can be overcome. This speed arises from pipeline processing wherein arithmetic computations and data access are all done simultaneously. To simplify the discussion assume an FPS array processor in which data access can, like additions and multiplications, be initiated every machine cycle. (This is not the case for our FPS - we can only fetch/store every other cycle.) For this FPS, the 1-D FFT algorithm is essentially compute bound, that is there is almost no waiting for data to be fetched before computations can proceed, whereas the individual complex vector multiply-add algorithms are not and therefore spend a good part of their time sitting idle. This leads us to consider whether a bit of microcode of our own can reduce or eliminate such unused time.

If the basic vector butterfly were coded we would have 8 data transfers, 4 real multiplies, and 6 real additions per loop, still 33% inefficient. (A multiply and add can be done simultaneously and so the 8 data transfers are the limiting factor.) However, if we microcode *two* stages of the butterfly with *four* input vectors and *four* output vectors we turn the corner and do become compute bound. This still holds true if we use the simplified rearrangement known as a radix 4 butterfly (Oppenheim and Schafer pp. 314-317) reproduced below. Therefore we can achieve parity in computational speed by writing a bit of our own microcode. (For our FPS a radix 8 butterfly, which uses 32 real multiplications, 66 real additions, and 32 data transfers, is needed to avoid waiting on memory.) Here at SEP Rob Clayton has written microcode to handle an arbitrary power of two radix by decomposing it internally into a sequence of radix two passes.

To take full advantage of computational parity requires corresponding savings in disk and array processor I/O. Here we want to proceed through as many FFT stages as possible before "coming up for air"; i.e., having to transfer data vectors between core and array processor or disk. Naturally we need to know how to select the right sets of column vectors and which twiddle factors to use with them. To grasp this we can relate the FFT to the factorization approach to matrix transposition. For this I borrow a page (371) from Rabiner and

Gold.

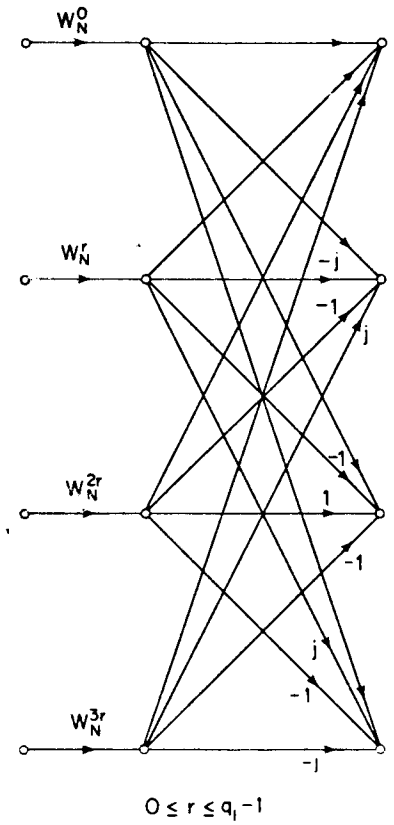


Fig. 6.29 Alternative arrangement of Fig. 6.28 resulting in savings of multiplications.

Suppose we have a sequence $L \times M$ elements long; i.e. write $x(n) = x(l,m)$ and let the transform index k likewise index an $M \times L$ (i.e. "transposed") N point DFT. Then we can write

$$X(k) = X(s,r) = \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} x(l,m)\, W_N^{(Ml+m)(Lr+s)} \quad 0 \le s \le L-1 \quad 0 \le r \le M-1 \tag{7}$$

$$= \sum_{m=0}^{M-1} W_M^{mr}\, W_N^{ms} \sum_{l=0}^{L-1} x(l,m)\, W_L^{ls} \tag{8}$$

which has the interpretation

1. Compute the L-point DFT of each column.
2. Obtain a new array $h(s,m)$ by multiplying by W_N^{ms}.
3. Compute the M-point DFT of each row.

Here we see a 1-D DFT as similar to a 2-D DFT with the addition of an intermediate "twiddle" step and the final transposed output. More generally a k-fold factorization

$$n_1 \times n_2 \times \ldots \times n_k \tag{9}$$

transposes under the algorithm into

$$n_k \times \ldots \times n_2 \times n_1 \tag{10}$$

which, for factors of two, gives rise to bit reversal.

For optimization of the cross-vector transform, then, we would like to decompose N into a product with the largest factors available storage will allow. Therefore, just as in matrix transposition, we factor the (row) dimension into the product (9) such that n_j blocks (512 bytes for example) fit in the array processor. (Depending upon I/O configurations, it can be advantageous to stage additional data in core so as to feed the AP new blocks more rapidly). Transforms then proceed via Rob Clayton's microcode for n_j using appropriate twiddle factors and the result is placed back on disk.

Separating the forest from the trees

Assume now we are in the typical case of having to compute FFT's for data with dimensions exactly a power of two. From our discussion above we might draw the following principle: With suitable attention to indices, a *2**M* by *2**N* submatrix held in core can be advanced through M column stages and N row stages of computation before writing them back to disk.

We see that the individual dimensions are immaterial: *2**K* elements held in core can be advanced through K computational passes. Therefore we might well expect that a dataset with *2**L* elements can be transformed with $(L+1)/K$ (rounded up to the nearest integer) I/O passes. (The $+1$ assumes one pass of bit-reverse reordering.) Researchers have proven this statement to be essentially correct. They show that because we access data in blocks it is the number of *blocks* rather than the number of *elements* in the data that in general determines the number of passes required for the transform. Fraser (1979) gives the formula $(L+1-B)/(K-B)$, where blocksize is *2**B* elements, and a computer program implementing such an algorithm. It is a straightforward, albeit lengthy, process to adapt it to perform all computations in the array processor. Microcode similar to that described above would need to be written to take advantage of parallelism.

Conclusions

When the number of rows exceeds the number of columns, the transposed FFT algorithm can be implemented using the same number of I/O operations as the alternative matrix transpose approach and can be made just as computationally efficient in the FPS array processor. The transposed FFT algorithm is in this case superior because it avoids the in-core data shuffling of the matrix transposition algorithm. When the number of rows is significantly less than the number of columns, matrix transposition becomes the preferred method, using fewer I/O operations. Both methods are limiting cases of the optimized mass storage FFT of Fraser which results in fewer I/O passes for large, approximately square data matrices.

REFERENCES

Claerbout, J.F., 1977, How to transpose a big matrix, SEP-11, p. 211-212.

Clayton, R., 1978, Two-dimensional Fourier transforms without transposing, SEP-15, p. 247-250.

Fraser, D., 1979, An optimized mass storage FFT, ACM Transactions on Mathematical Software, Vol.5, No. 5, p. 500-517.

Oppenheim, A.V. and Schafer, R.W., 1975, Digital signal processing, Prentice-Hall, New Jersey.

Rabiner, L.R. and Gold, B., 1975, Theory and application of digital signal processing, Prentice-Hall, New Jersey.