Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers

Jack J. Dongarra*
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Linda Kaufman
AT & T Bell Laboratories
Murray Hill, New Jersey 07974

and

Sven Hammarling
Numerical Algorithms Group Ltd.
NAG Central Office, Mayfield House
256 Banbury Road, Oxford OX2 7DE, England

Submitted by J. Alan George

ABSTRACT

This paper describes modifications to many of the standard algorithms used in computing eigenvalues and eigenvectors of matrices. These modifications can dramatically increase the performance of the underlying software on high-performance computers without resorting to assembler language, without significantly influencing the floating-point operation count, and without affecting the roundoff-error properties of the algorithms. The techniques are applied to a wide variety of algorithms and are beneficial in various architectural settings.

INTRODUCTION

On high-performance vector computers like the CRAY-1, CRAY X-MP, Fujitsu VP, Hitachi S-810, and Amdahl 1200, there are three basic performance levels—*scalar*, *vector*, and *supervector*. For example, on the CRAY-1 [5, 7, 10],

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

these levels produce the following execution rates:

Performance level	Rate of execution (MFLOPS)[1]
Scalar	0–4
Vector	4–50
Supervector	50–160

Scalar performance is obtained when no advantage is taken of the special features of the machine architecture. Vector performance is obtained by using the vector instructions to eliminate loop overhead and to take full advantage of the pipelined functional units. Supervector performance is obtained by using vector registers to reduce the number of memory references and thus avoid letting the paths to and from memory become a bottleneck.

Typically, programs written in FORTRAN run at scalar or vector speeds, so that one must resort to assembler language (or assembler-language kernels) to improve performance. But in [2], Dongarra and Eisenstat describe a technique for attaining supervector speeds *from* FORTRAN for certain algorithms in numerical linear algebra. They notice that many algorithms had the basic form

```
ALGORITHM A.

For $i = 1$ to $m$
    $y \leftarrow \alpha_i x_i + y$
End
```

where α_i is a scalar and x_i and y are vectors. Unfortunately, when this algorithm is implemented in a straightforward way, the CRAY, Fujitsu, and Hitachi FORTRAN compliers do not recognize that it is the "same y" acted upon every time, and issue a store vector y and a load vector y command between each vector additions. Thus the path to and from memory becomes the bottleneck. The compliers generate vector code of the general form

```
Load vector Y
Load scalar $\alpha_i$
Load vector X(I)
Multiply scalar $\alpha_i$ times vector X(I)
Add result to vector Y
Store result in Y
```

[1] MFLOPS is an actonym for million FLoating-point OPerations (additions or multiplications) per second.

This gives 2 vector operations to 3 vector memory references. Moreover because of the concept called "chaining" on the CRAY, Fujitsu and Hitachi, the time for the vector multiply and add is practically insignificant. In most circumstances these may be initiated soon after the loading of the vector X(I) has begun, and for vectors of significant length the load, multiply, and add may be thought of as practically simultaneous operations.

Dongarra and Eisenstat showed that if one unrolled the loop several times[2], the number of memory references could be reduced and execution times often decreased by a factor of 2 or 3. For example unrolling Algorithm A to a depth of two gives:

```
ALGORITHM A.2.
For $i = 2$ to $m$ in steps of 2
    $y \leftarrow \alpha_{i-1}x_{i-1} + \alpha_i x_i + y$
End
if ($m$ is odd) $y \leftarrow \alpha_m x_m + y$
```

The compliers generate vector code of the general form

```
    Load vector Y
    Load scalar $\alpha_{i-1}$
    Load vector X(I − 1)
    Multiply scalar $\alpha_{i-1}$ times vector X(I − 1)
    Add result to vector Y
    Load scalar $\alpha_i$
    Load vector X(I)
    Multiply scalar $\alpha_i$ times vector X(I)
    Add result to vector Y
    Store result in Y
```

This gives 4 vector operations to 4 vector memory references. The larger the ratio of vector operations to vector memory references becomes, the better the performance of the program segment. This is the result of vector memory operations, i.e. loads and stores, costing as much as other vector operations. When the loop is unrolled to a depth of 8 there are 16 vector operations to 10 vector memory references. Dongarra and Eisenstat incorporated this idea into two "kernal" subroutines: SMXPY, which added a matrix times a vector to

[2]The loops have been unrolled to different depths on different machines, depending on the effect; on the CRAY the depth is 16, and on the Fujitsu and Hitachi the depth is 8.

another vector ($Ax + y$), and SXMPY, which added a vector times a matrix to another vector ($x^TA + y^T$). They showed that several linear system solvers could be rewritten using these kernel subroutines.

In this paper we try to apply the same concept to algorithms used in solving the eigenvalue problem. Normally these problems are solved in several steps:

(1) Reduce the problem to a simpler problem (e.g., a tridiagonal matrix if the matrix was symmetric),

(2) Solve the eigenproblem for the simpler problem,

(3) If eigenvectors are requested, transform the eigenvectors of the simplified problem to those of the original problem.

For symmetric problems, step (2) usually has the fewest floating-point operations, while for nonsymmetric matrices step (2) has the most floating-point operations. Because steps (1) and (3) often involve transformations that can be forced into the form of Algorithm A, we will concentrate our efforts on these steps. In certain cases speeding up these steps will not significantly affect the overall time required to solve the eigenproblem, but in other cases, such as the symmetric generalized eigenvalue problem, we will be speeding up the most time-consuming portion of the whole operation. Sometimes part of the algorithm simply has a matrix-by-vector multiplication; then application of Dongarra and Eisenstat's idea is straightforward. At other times, the code will need to be radically transformed to fit the form of Algorithm A.

In Section 2 we describe some underlying ideas that can be used to decrease memory references in various subroutines in the matrix eigenvalue package EISPACK [6, 11, 13]. In Section 3 we apply the concepts of Section 2 to specific subroutines in EISPACK and provide execution-timing information on the CRAY-1, the current version of EISPACK [3]. The appendix contains execution-timing information on the Hitachi S-810/20 and Fujitsu VP-200 (Amdahl 1200). (In [4] we presented reprogramming of selected subroutines that are radically different from the original or representative of a class of changes that might be applied to several subroutines.)

2. UNDERLYING IDEAS

In this section we outline some of the underlying methods that occur throughout the algorithms used in the EISPACK package. We also discuss how they can be implemented to decrease vector memory references, without significantly increasing the number of floating-point operations.

2.1. *Transformations*

Many of the algorithms implemented in EISPACK have the following form:

ALGORITHM B.

```
For $i, \ldots$
    Generate matrix $T_i$
    Perform transformation $A_{i+1} \leftarrow T_i A_i T_i^{-1}$
End
```

Because we are applying similarity transformations, the eigenvalues of A_{i+1} are those of A_i. In this section we examine various types of transformation matrices T_i.

2.1.1. Stabilized Elementary Transformations. Stabilized elementary transformation matrices have the form $T = PL$, where P is a permutation matrix, required to maintain numerical stability [12], and L has the form

$$\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & * & 1 & & \\ & & & \vdots & & \ddots & \\ & & & * & & & 1 \end{pmatrix}.$$

The inverse of L has the same structure as L. When L^{-1} is applied on the right of a matrix, one has a subalgorithm with the exact form of Algorithm A, which can be implemented using SMXPY. Unfortunately, when applying L on the left as in Algorithm B, one does not get the same situation. The vector y changes, but the vector x remains the same. However, in the sequence of transformations in Algorithm B, T_i consists of a matrix L_i whose off-diagonals are nonzero, only in the ith column, and at the ith step, one might apply transformations T_1 through T_i only to the ith row of the matrix. Subsequent row transformations from the left will not affect this row, and one can implement this part of the algorithm using SXMPY.

This idea was incorporated in the subroutine ELMHES, which will be discussed in Section 3.

2.1.2. *Householder Transformations.* In most of the algorithms the transformation matrices T_i are Householder matrices of the form

$$Q = I - \beta u u^T, \qquad \text{where} \quad \beta u^T u = 2,$$

so that Q is orthogonal. To apply Q from the left to a matrix A, one would proceed as follows:

ALGORITHM C.

```
1.  $v^T = u^T A$
2.  Replace $A$ by $A - \beta u v^T$
```

Naturally the first step in Algorithm C can be implemented using SXMPY, but the second step, the rank-one update, does not fall into the form of Algorithm A. However, when applying a sequence of Householder transformations, one may mitigate the circumstances somewhat by combining more than one transformation and thus performing a higher than rank-one update on A. This is somewhat akin to the technique of loop unrolling discussed earlier. We give two illustrative examples.

Firstly suppose that we wish to form $(I - \alpha w w^T)A(I - \beta u u^T)$, where for a similarity transformation $\alpha = \beta$ and $w = u$. This is normally formed by first applying the left-hand transformation as in Algorithm C, and then similarly applying the right-hand transformation. But we may replace the two rank-one updates by a single rank-two update using the following algorithm:

ALGORITHM D.1.

```
1.  $v^T = w^T A$
2.  $x = Au$
3.  $y^T + v^T - (\beta w^T x) u^T$
4.  Replace $A$ by $A - \beta x u^T - \alpha w y^T$
```

As a second example suppose that we wish to form $(I - \alpha w w^T)(I - \beta u u^T)A$; then as with Algorithm D.1 we might proceed as follows:

ALGORITHM D.2.

```
1.  $v^T = w^T A$
2.  $x^T = u^T A$
3.  $y^T = v^T - (\beta w^T u) x^T$
4.  Replace $A$ by $A - \beta u x^T - \alpha w y^T$
```

In both cases we can see that steps 1 and 2 can be achieved by calls to SXMPY and SMXPY. Step 3 is a simple vector operation and step 4 is now a rank-two correction, and one gets 4 vector memory references for each 4 vector floating-point operations (rather than the 3 vector memory references for every 2 vector floating-point operations, as in step 2 Algorithm C). The increased saving is not as much as is realized with the initial substitution of SXMPY for the inner products in step 1 of Algorithm C, but it more than pays for the additional $2n$ operations incurred at step 3 and exemplifies a technique that might pay off in certain situations. This technique was used to speed up a number of routines that require Householder transformations.

2.1.3. Plane Rotations. Some of the most time-consuming subroutines in EISPACK, e.g. HQR2, QZIT, IMTQL2, TQL2, spend most of their time applying transformations in 2 or 3 planes to rows or columns of matrices. We have been able to speed up the application of these transformations by only about 15%, but if one is spending 90% of one's computation time here, the total effect is greater than that of improving the part which only contributes 10% of the total computation time.

First of all we should mention that on the CRAY-1 the time required by a 3-multiply Householder transformation in 2 planes is hardly less than that required by a 4-multiply Givens transformation [12]. Thus once again the computation time is influenced more by the number of vector memory references than by the number of floating-point operations. We were able to eliminate several vector loads and stores by noticing that one of the planes used in one transformation is usually present in the next. Thus a typical Givens code which originally looked like

```
For $i = 1$ to $n - 1$
   Compute $c_i$ and $s_i$
   For $j = 1$ to $n$
      $t \leftarrow h_{ji}$
      $h_{ji} \leftarrow c_i t + s_i h_{j,i+1}$
      $h_{j,i+1} \leftarrow s_i t - c_i h_{j,i+1}$
   End
End
```

would become

```
For $j = 1$ to $n$
   $t_j \leftarrow h_{j1}$
End
For $i = 1$ to $n - 1$
```

```
        Compute $c_i$ and $s_i$
        For $j = 1$ to $n$
            $h_{ji} \leftarrow c_i t_j + s_i h_{j,i+1}$
            $t_j \leftarrow s_i t_j - c_i h_{j,i+1}$
        End
    End
    For $j = 1$ to $n$
        $h_{jn} \leftarrow t_j$
    End
```

and a typical Householder code which looked like

```
For $i = 1$ to $n - 1$
    Compute $q_i, x_i$ and $y_i$
    For $j = 1$ to $n$
        $p \leftarrow h_{ji} + q_i h_{j,i+1}$
        $h_{ji} \leftarrow h_{ji} + p x_i$
        $h_{j,i+1} \leftarrow h_{j,i+1} + p y_i$
    End
End
```

would become

```
For $i = 1$ to $n - 2$ in steps of 2
    Compute $q_i, x_i, y_i, q_{i+1}, x_{i+1}$ and $y_{i+1}$
    For $j = 1$ to $n$
        $p \leftarrow h_{ji} + q_i h_{j,i+1}$
        $r \leftarrow h_{j,i+1} + q_{i+1} h_{j,i+2} + y_i p$
        $h_{ji} \leftarrow h_{ji} + p x_i$
        $h_{j,i+1} \leftarrow h_{j,i+1} + p y_i + r x_{i+1}$
        $h_{j,i+2} \leftarrow h_{j,i+2} + r y_{i+1}$
    End
End
```

Notice that for the Householder transformations we have actually increased the number of multiplications in total but still the amount of time has decreased. For a 3-plane Householder transformation, like that found in HQR2, unrolling the loop twice causes about a 10% drop in execution time.

Inserting the modified Givens into a code like TQL2 is an easy task. Changing codes like HQR2 to use the unrolled Householders is rather unpleasant.

2.2. *Triangular Solvers*

Assume one has an $n \times n$ nonsingular lower triangular matrix L and an $n \times m$ matrix, B, and wishes to solve

$$LY = B. \tag{2.1}$$

If $m = 1$ one might normally proceed as follows:

ALGORITHM E.

```
$y \leftarrow b$
For $i = 1$ to $n$
   $y_i \leftarrow y_i / l_{ii}$
   For $j = i+1$ to $n$
      $y_j \leftarrow y_j - l_{ji} y_i$                                        (2.2)
   End
End
```

Equation (2.2) almost looks like Algorithm A, but the length of the vector y decreases. Unrolling the i loop once decreases the number of vector memory references from 3 for every 2 vector floating-point operations to 4 for every 4 vector floating-point operations. The unrolled code would be of the following form:

ALGORITHM F.

```
$y \leftarrow b$
For $i = 1$ to $n-1$ in steps of 2
   $y_i \leftarrow y_i / l_{ii}$
   $y_{i+1} \leftarrow (y_{i+1} - l_{i+1,i} y_i) / l_{i+1,i+1}$
   For $j = i+2, \cdots, n$
      $y_j \leftarrow y_j - y_i l_{ji} - y_{i+1} l_{j,i+1}$
   End
End
If $(n \bmod 2 \neq 0)$ $y_n \leftarrow y_n / l_{nn}$
```

On the CRAY-1 the ratio of execution times of Algorithm F to Algorithm E is 1.5, as Table 1 indicates.

However, when m is sufficiently large that it makes computational sense to treat vectors of length m, one can do much, much better by computing Y by rows rather than repeating either Algorithm E or Algorithm F for each column. Let Y_j denote the first j rows of the matrix Y, y_j^T denote its jth row,

TABLE 1
CRAY-1 TIMES (IN 10^{-3} SECONDS) FOR TRIANGULAR SOLVERS

n	m	Algorithm E	Algorithm F	Algorithm G
100	1	.506	.340	5.14
	25	12.5	8.32	6.92
	100	49.9	33.3	14.4
200	1	1.55	1.02	17.2
	25	38.6	25.5	24.1
	100	151	102	52.2
	200	308	202	93.7
300	1	3.15	2.05	35.9
	25	78.5	51.1	51.8
	150	472	306	162
	300	940	613	290

and l_j^T denote the jth row of L. Then one might proceed as follows:

```
ALGORITHM G.

$Y \leftarrow B$
For $j = 1$ to $n$
    $y_j^T \leftarrow b_j^T - l_j^T Y_{j-1}$                    (2.3)
    $y_j^T \leftarrow y_j^T / l_{jj}$
End
```

The step (2.3) can be implemented using SXMPY. Obviously, working by rows is superior if m is sufficiently large. Since Algorithm G uses vectors of length m, when m is small one should use Algorithm F. We have discussed triangular solvers using a lower triangular matrix. One can implement the last three algorithms for an upper triangular matrix, and Algorithm G would determine the last row first and work backwards. Triangular solvers occur in the EISPACK subroutines REDUC and REBAK used in the solution of the symmetric generalized eigenvalue problem $Ax = \lambda Bx$. Inserting calls to SXMPY and SMXPY decreases the time required by these subroutines to such an extent that the time required for the generalized eigenvalue problem is not appreciably more than that required for the standard eigenvalue problem on the high-performance computers under discussion.

2.3. *Matrix Multiplication with Symmetric Packed Storage*

The algorithms in EISPACK that deal with symmetric matrices permit the user to specify only the lower triangular part of the matrix. There are routines requiring that a two-dimensional array be provided, using only the information in the lower portion and routines accommodating the matrix packed into a one-dimensional array. The normal scheme for doing this matrix-vector product would be

ALGORITHM H.

```
For $j = 1$ to $n$
    $t \leftarrow y_j$
    For $i = j + 1$ to $n$
        $y_i \leftarrow y_i + a_{ij}x_j$
        $t \leftarrow t + a_{ij}x_i$
    End
    $y_j \leftarrow t + a_{jj}x_j$
End
```

Certainly one might consider stepping the outer loop by 2, doing two inner products followed by a rank-two correction. Another alternative in the same vein, which unfortunately would not be amenable to a subroutine that packed the symmetric matrix into a one-dimensional array, is the following:

ALGORITHM I.

```
For $i = 1$ to $n - 1$ in steps of 2
    For $j = 1$ to $i - 1$
        $y_j \leftarrow y_j + a_{ij}x_i + a_{i+1,j}x_{i+1}$
    End
    For $j = i + 1$ to $n$
        $y_j \leftarrow y_j + a_{ji}x_i + a_{j,i+1}x_{i+1}$
    End
    $y_i \leftarrow y_i + a_{i+1,i}x_{i+1} + a_{ii}x_i$
End
```

A less obvious technique is a divide-and-conquer approach. If we consider referencing a symmetric matrix in a matrix-vector product where the matrix is specified in the lower triangular matrix, we have

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + \begin{pmatrix} T_1 & B^T \\ B & T_2 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

where T_1 and T_2 are symmetric matrices stored in the lower portion and B is

TABLE 2
COMPARISON OF EXECUTION TIMES ON THE CRAY-1
FOR SYMMETRIC MATRIX MULTIPLY

	Ratio of execution times		
Order	Alg. H/Alg. I	Alg. H/Alg. HJ	Alg. H/Alg. IJ
100	2.14	1.24	2.13
200	1.87	1.43	2.12
300	1.78	1.53	2.08

full. This can be written as

```
ALGORITHM J.

Set $y_1 = T_1x_1 + B^Tx_2$
Set $y_2 = Bx_1 + T_2x_2$
```

In writing the matrix multiply this way, two things should be noted. There are two square $(n/2)\times(n/2)$ full matrix-vector multiplications, and two symmetrix matrix-vector products.

Table 2 gives a comparison on the CRAY-1 of Algorithm H (a standard approach) with Algorithm I, Algorithm HJ (where T_1x_1 and T_2x_2 of Algorithm J are done according to Algorithm H), and Algorithm IJ (where these are done according to Algorithm I).

When the matrix is packed in a one-dimensional array stored by column, the same divide-and-conquer approach can be applied.

3. SUBROUTINES IN EISPACK

3.1. *The Unsymmetric Eigenvalue Problem*

In this section we investigate methods for the efficient implementation of the algorithms that deal with the standard eigenvalue problem

$$Ax = \lambda x,$$

where A is a real general matrix. The algorithms for dealing with this problem follow the form:

(1) Reduce A to upper Hessenberg form (ELMHES or ORTHES).

(2) Find the eigensystem of the upper Hessenberg matrix.

(3) If eigenvectors are requested, back-transform the eigenvectors of the Hessenberg matrix to form the eigenvectors of the original system (ELMBAK or ORTBAK).

For this particular problem, most of the time is spent in the second step, but as was discussed in Section 2.1.3, this part is not easy to vectorize, so we concentrate our discussion on steps (1) and (3).

3.1.1. ELMHES *and* ORTHES. In the subroutine ELMHES, which reduces a matrix to upper Hessenberg form while preserving the eigenvalues of the original matrix, a sequence of stabilized elementary transformations are used. The transformations are of the form

$$T_k \cdots T_2 T_1 A T_1^{-1} T_2^{-1} \cdots T_k^{-1}.$$

This set of transformations has the effect of reducing the first k columns of A to upper Hessenberg form.

The transformations can be applied in such a way that matrix-vector operations are used in the time-consuming part. At the kth stage of the reduction we apply the previous $k-1$ transformations on the left side of the reduced A to the last $n-k$ elements of the $k+1$st row. Then, on the right the inverse of the kth transformation is applied to the reduced matrix A, followed by the application on the left of all k transformations to the elements below the diagonal of the kth column. Because of the structure of the transformations (see Section 2.1.1), both these steps are simple matrix-vector multiplication. The application of transformations from the left follows essentially the algorithm given in Dongarra and Eisenstat [2] for the LU decomposition of a matrix. In the original EISPACK codes, at the kth stage permutations from the left are applied to only the last $n-k+1$ columns of the matrix. In our new code, in order to use SMXPY and SXMPY, we must apply these permutations to the whole matrix. Thus the elements below the subdiagonal of the matrix A which are necessary for finding the eigenvectors might be slightly scrambled and hence the user must use the modified ELMBAK given in Section 3.1.3.

The subroutine ORTHES uses Householder orthogonal similarity transformations to reduce A to upper Hessenberg form. At the kth stage we perform the operation

$$Q_k A Q_k^T,$$

where $Q_k = I - \beta u u^T$. As shown in Algorithm D.1, the usual two rank-one updates may be replaced by a rank-one update to the first k rows of A followed by a rank-two update to rows $k+1$ through n. In this case

TABLE 3
COMPARISON OF EXECUTION ON THE CRAY-1
FOR ROUTINE ELMHES AND ORTHES

	Ratio of execution times: (EISPACK/MV)		
		orthes	
Order	ELMHES	Rank 1 only	rank 2
50	1.5	2.0	2.5
100	2.2	1.9	2.5
150	2.4	1.8	2.4

Algorithm D.1 becomes

```
1. $v^T = u^T A$
2. $x = Au$
3. $y^T = v^T - (\beta u^T x) u^T$
4. Replace $A$ by $A - \beta(xu^T + uy^T)$
```

Seeing the transformations applied in this way leads to a straightforward matrix-vector implementation. Table 3 reports the comparison between the EISPACK implementations and the ones just described. Significant speedups are accomplished using these constructs.

3.1.2. ELTRAN *and* ORTRAN. If all the eigenvectors are requested, one might choose to use either ELTRAN or ORTRAN (depending on whether one used ELMHES or ORTHES) followed by a call to HQR2, rather than finding the eigenvectors using INVIT and then back transforming using ELMBAK or ORTBAK. ELTRAN requires no floating-point operations, but because of the use of stabilized elementary transformations in ELMHES, it may require swapping of various rows of the partial eigenvector matrix being constructed. Because ELMHES has changed, the swapping in ELTRAN is slightly different. ORTRAN applies the Householder transformations determined in ORTHES to the identity matrix. By combining two Householder transformations we can perform a rank-two update to I using the technique described in Section 2.1.2, and this realizes a cut in the execution time for this routine by a factor of two.

3.1.3. ELMBAK *and* ORTBAK. Both ELMBAK and ORTBAK compute the eigenvectors of the original matrix given the eigenvectors of the upper Hessenberg matrix and the transformations used to reduce the original matrix. This requires that a set of transformations be applied on the left to the matrix of eigenvectors in reverse order. The reduction is of the form $TAT^{-1}TX = \lambda TX$, where $T = T_{n-1} - T_2 T_1$. The eigenvectors, say Y, of the reduced matrix H are found using

$$H = TAT^{-1} \quad \text{and} \quad HY = \lambda Y;$$

then the eigenvectors for the original problem are computed as

$$X = T^{-1}Y = T_1^{-1}T_2^{-1} \cdots T_n^{-1}Y.$$

The original EISPACK subroutines use T as a product of transformations as given above. For ELMBAK we use a slightly different approach. As in Section 2.1.1, each T_i may be written as $L_i P_i$, where P_i is a permutation matrix and L_i is a lower triangular matrix. On output from the new ELMHES, let B be the $(n-1)\times(n-1)$ lower triangular matrix below the subdiagonal of the reduced A. Let C be the unit lower triangular matrix

$$C = \begin{pmatrix} 1 & & & \\ 0 & 1 & & \\ \vdots & B & \ddots & \\ 0 & & & 1 \end{pmatrix}.$$

Then one can show that $T^{-1} = P_1 P_2 \ldots \ldots P_{n-1} C$.

Since ORTBAK involves a product of Householder transformations, reducing the number of vector memory references is again a straightforward task. Dramatic improvements are seen in these back-transformation routines, as shown in Table 4. Originally ELMBAK was 2.4 times faster than ORTBAK; in the MV version it only enjoys an advantage of 1.9 over ORTBAK using $(n-1)/2$ rank-2 changes.

TABLE 4
COMPARISON OF EXECUTION ON THE CRAY-1 FOR EISPACK ROUTINES ORTBAK AND ELMBAK

	Ratio of execution times (EISPACK/MV)		
		ORTBAK	
Order	ELMBAK	Rank 1	Rank 2
50	2.2	2.8	3.6
100	2.6	2.5	3.3
150	2.7	2.3	3.0

3.2. *The Symmetric Eigenvalue Problem*

In this section we look at the methods for efficient implementation of the algorithms that deal with the symmetric eigenvalue problem

$$Ax = \lambda x,$$

where A is a real symmetric matrix. The algorithms for dealing with this problem have two possible paths:

Path 1.

(1) Transform A to tridiagonal form (TRED1).

(2) Find the eigenvalues of the tridiagonal matrix (IMTQLV).

(3) If the eigenvectors are requested, find the eigenvectors of the tridiagonal matrix by inverse iteration (TINVIT).

(4) If eigenvectors are requested, back-transform the vectors of the tridiagonal matrix to form the eigenvectors of the original system (TRBAK1).

Path 2.

(1) Transform A to tridiagonal form, accumulating the transformations (TRED2).

(2) Find the eigenvalues of the tridiagonal matrix and accumulate the transformations to give the eigenvectors of the original matrix (IMTQL2).

On conventional serial machines, Path 2 typically requires nearly twice as much time as Path 1. On vector machines however we do not see this relationship. For EISPACK, Path 2 is slightly faster and after the modification described below requires roughly the same amount of time. This is the result of two problems in routine TINVIT. First, TINVIT has not been modified to induce vectorization at any level. One can achieve an increase in performance by vectorizing across the eigenvectors being computed. We have not presented an algorithm of this form, since it requires a different technique to achieve performance and cannot run at supervector rates. The time spent in TINVIT on serial machines is inconsequential with respect to the total time to execute Path 1. However, on vector machines TINVIT becomes a significant contributor to the total execution time of the path. The second factor influencing performance for Path 1 is that the current version of TINVIT has a call to an auxiliary routine, PYTHAG, in an inner loop of the algorithm. PYTHAG is used to safely and portably compute the square root of the sum of squares. If TINVIT is modified to replace the call to PYTHAG by a simple square root, the time for TINVIT becomes more attractive by about 30%.

We note that the advantage of path 2 is that near-orthogonality of the eigenvectors is guaranteed, while with Path 1 one may see some degradation in this property for eigenvectors corresponding to close eigenvalues. Both paths give excellent eigensystems in the sense that they are eigensystems of a problem close to the original problem [12, 13].

We will now describe the implementation of routines TRED1 and TRED2 using matrix-vector operations.

3.2.*1*. TRED1, TRED2, and TRBAK1. Routine TRED1 or TRED2 reduces a real symmetric tridiagonal matrix using orthogonal similarity transformations. An $n \times n$ matrix requires $n-2$ transformations, each of which introduces zeros into a particular row and column of the matrix, while preserving symmetry and preserving the zeros introduced by previous transformations. TRED1 is used to just compute the tridiagonal matrix, while TRED2, in addition to computing the tridiagonal matrix, also returns the orthogonal matrix which would transform the original matrix to this tridiagonal matrix. TRBAK1 forms the eigenvectors of the real symmetric matrix from the eigenvectors of the symmetric tridiagonal matrix determined by TRED1. This orthogonal matrix will later be used in computing the eigenvectors of the original matrix. These subroutines deal with the real symmetric matrix as stored in the lower triangle of an array.

The sequence of transformations applied to the matrix A is of the form

$$A_{i+1} \leftarrow Q_i A_i Q_i, \qquad i = 1, 2, \cdots, n-2,$$

where Q is a Householder matrix of the form described in Section 2.1.2. Each of the similarity transformations is applied as in Algorithm D.1 with the simplification that w and u are the same, so that application becomes

1. $x = Au$
2. $y^T = x^T - (\beta u^T x)u^T$
3. Replace A by $A - \beta x u^T - \beta u y^T$

Since the matrix A is symmetric and stored in the lower triangle of the array, the matrix-vector operation in step 1 follows the form described in Section 2.3 as implemented in Algorithm IJ.

TRED2 differs from TRED1 in that the transformation matrices are accumulated in an array Z. The sequence of transformations applied to the matrix Z is of the form

$$Z_{n-2} = Q_{n-2},$$

$$Z_i \leftarrow Q_i Z_{i+1}, \qquad i = n-3, \cdots, 2, 1.$$

This can be implemented in a straightforward manner as in Algorithm C of Section 2.1.2 using matrix-vector multiply and a rank-one update. Since all transformations are available at the time they are to be accumulated, more than one transformation can be accumulated at a time, say two at a time, thus giving a rank-two update. This then gives an implementation that has the form of Algorithm D.2 in Section 2.1.2.

When TRED1 and TRED2 are implemented as described, significant improvements in the execution time can be realized on vector machines. Table 5 displays the execution time for the current EISPACK versions of TRED1 and TRED2 as well as the modified matrix vector implementations, referred to as TRED1V and TRED2V.

TRBAK1 applies the transformations used by TRED1 to reduce the matrix to tridiagonal form. This can be organized as in TRED2, by matrix-vector multipli-

TABLE 5

CRAY-1 TIMES (IN 10^{-2} SEC) FOR THE SYMMETRIC GENERALIZED EIGENVALUE PROBLEM

Subroutine	$n = 100$	$n = 200$
REDUC	16.9	85.5
REDUC3	4.26	23.6
REDUCV	3.62	19.5
REDUC4	3.00	16.1
TRED1	6.94	38.5
TRED1V	4.95	29.7
TRED2	14.3	84.5
TRED2V	8.31	51.3
TQL1	7.58	29.1
TQL2	19.8	117
REBAK	9.79	52.5
REBAKV	2.20	15.3
No vectors:		
total old	32.92	165.4
total new	16.15	78.3
Vectors:		
total old	60.8	339.6
total new	33.9	203.1

cation and a rank-2 update. The table below shows the improvement in performance when this is implemented:

COMPARISON OF EXECUTION ON THE CRAY-1
FOR EISPACK ROUTINE TRBAK1

Order	Ratio of execution times (EISPACK/MV version)
50	4.20
100	3.66

3.3. *The Symmetric Generalized Eigenvalue Problem*

In this section we consider methods for increasing the efficiency of the subroutines in EISPACK for solving the generalized eigenvalue problem

$$Ax = \lambda Bx,$$

where A and B are symmetric matrices and B is positive definite. In EISPACK this problem is solved in the following steps with the name of the corresponding subroutine in the package given in parenthesis:

(1) Factor B into LL^T, and form $C = L^{-1}AL^{-T}$ (REDUC).
(2) Solve the symmetric eigenvalue problem $Cy = \lambda y$.
(3) If eigenvectors are requested, transform the eigenvectors of C into those of the original system (REBAK).

In general the majority of the execution time is spent in REDUC and REBAK, and it will be these routines on which we will concentrate.

3.3.1. REDUC.

REDUC has three main sections:

1. Find the Cholesky factors of B, i.e., find lower triangular L such that $B = LL^T$
2. Find the upper triangle of $E = L^{-1}A$
3. Find the lower triangle of $C = L^{-1}E^T$

Step 1, the Cholesky factorization, was discussed in Dongarra and Eisenstat [2]; its inner loop can be replaced by the call to SMXPY. Step 2 is a lower triangular solve. The original code in EISPACK follows the suggestion in Section 2.2 and computes E by rows. Thus it is a simple matter to replace the inner loop by a call to SMXPY. Step 3 is another lower triangular solve. The EISPACK encoding computes C by columns and uses the fact that C is symmetric. Thus the first $i-1$ elements of the ith column of C are already known before the code commences to work on the ith column. For the ith column REDUC has two inner loops. The first updates the last $n-i$ elements with the previous known $i-1$ elements. The second does a lower triangular solve with an $(n-1)\times(n-1)$ matrix as in Algorithm E of Section 2.2. The first loop can be easily implemented using a SMXPY; the only hope for easily increasing the efficiency of the second loop is using Algorithm F of Section 2.2.

Thus it is straightforward to replace three of the four inner loops of REDUC by SMXPY, and this is accomplished by REDUC3 listed in Table 5. The decrease in execution time of REDUC3 from REDUC is quite surprising, considering that the changes being made affect only how the matrix is accessed. REDUCV replaces the fourth loop of REDUC by Algorithm F of Section 2.2. It produces a further modest saving.

REDUC4 replaces the two inner loops of step 3 of REDUC by a modification of Algorithm G which computes only the first i elements of the ith row of C rather than the whole row. Because C is symmetric, these first i elements are also the first i elements of the ith column of C. Thus by the end of the ith stage of step 3 of REDUC4, the top $i\times i$ submatrix of C has been computed while at the same stage of REDUC and REDUC3, the first i columns of C have been computed. REDUC4, as Table 5 indicates, is the least expensive of the subroutines, but it has one major drawback. REDUC, REDUC3, and REDUCV overwrite only the lower-triangular portions of the matrices A and B while forming L, E, and C. REDUC4 overwrites the whole matrix A.

3.3.2. REBAK. The subroutine REBAK takes the eigenvectors Y of the standard symmetric eigenproblem and forms those of the original problem X by multiplying Y by L^{-T}. Thus it is an upper-triangular solve with many right-hand sides. The original REBAK computes X one column at a time using inner products. REBAKV uses the upper-triangular version of Algorithm G to compute X. The difference is computation times given in Table 5 for REBAK and REBAKV is really remarkable considering that they require the same number of floating-point operations.

3.4 *The Singular-Value Decomposition*

The singular-value decomposition (SVD) of an $m\times n$ matrix A is given by

$$A=U\Sigma V^T,$$

where U is an $M\times n$ orthogonal matrix, V is an $n\times n$ orthogonal matrix, and Σ is an $m\times n$ diagonal matrix containing the singular values of A, which are the nonnegative square roots or the eigenvalues of A^TA. Amongst the many applications of the SVD algorithm is the solution of least-squares problems and the determination of the condition number of matrix.

The algorithm implemented in EISPACK's SVD is usually considered to have two stages:

(1) Determine Q and Z such that $J = QAZ$ is bidiagonal.
(2) Iteratively reduce J to a diagonal matrix.

In typical applications where $m \gg n$ the first stage of the SVD calculation is the most time consuming. The matrices Q and Z are the product of Householder transformations, and, as described in Section 2.1, the number of vector memory references can be reduced by replacing all vector matrix multiplications by calls to SXMPY, as is done in the subroutine SVDI given in Table 6. Moreover, since each Householder transformation from the left is followed by Householder transformation from the right, one may use Algorithm D.1, and this is implemented in subroutine SVDV in Table 6. The second stage of the SVD calculation involves plane rotations which, when only the singular values are requested, do not involve any operations.

When the singular vectors are requested, the Householder transformations which form Q and Z are accumulated in reverse order. Here again we can use the techniques described earlier. In the second stage, plane rotations are applied to vectors, and it is not easy to decrease the number of vector memory references as was discussed in Section 2.1.3.

Chan [1] has described a modification of SVD which, when $m \gg n$, requires up to 50% fewer floating-point operations. Chan suggests first applying Householder transformations from the left to reduce A to triangular

TABLE 6
CRAY-1 TIMES (IN 10^{-2} SEC) FOR THE SINGULAR-VALUE DECOMPOSITION

$m =$	100			200		
$n =$	10	50	100	10	50	100
No singular vectors:						
SVD	1.02	10.8	30.9	1.92	18.4	54.7
SVDI	0.57	7.57	23.1	0.94	11.5	37.3
SVDV	0.38	6.1	19.4	0.55	8.2	28.0
With singular vectors:						
SVD	1.31	19.8	70.1	2.41	32.6	115
SVDI	0.85	16.5	61.8	1.43	25.6	97.3
SVDV	0.68	13.5	51.0	1.09	20.6	79.5

form before applying the traditional SVD algorithm. Thus the Householder transformations applied from the right, which are designed to annihilate elements above the superdiagonal, would be applied to vectors of length n rather than to vectors of length m. Unfortunately, on the CRAY-1 Chan's suggestion seems to produce at most a 10% speedup in execution time. When the inner product loops in all Householder transformation applications are replaced by calls to SMXPY, the execution times are still about the same as for SVDV.

4. CONCLUSIONS

In this paper we have shown how to make some of the subroutines in EISPACK more efficient on vector machines such as the CRAY-1. We have concentrated our efforts on speeding up programs that already run at vector speed but because of bottlenecks caused by referencing vectors in memory do not run at supervector speeds. We have not considered subroutines which currently run at scalar speeds, like BANDR [8] on small-bandwidth problems, TINVIT, and BISECT, which can all be vectorized.

Our techniques for speeding up the eigenvalue solvers do not significantly change the number of floating-point operations, only the number of vector loads and stores. Since we have been able to obtain speedups in the range of 2 to 5, vector loads and stores seem to be the dominant factor in determining the time required by an algorithm. Thus the traditional merit function, the number of floating-point operations, seems to be not as relevant for these machines as for the scalar machines.

For the most part we have been able to isolate computationally intense sections of codes into well-defined modules. This has made some of the programs shorter and has made their mathematical function clearer. Some of the techniques used to gain better performance could be done by an extremely clever vectorization compiler. However, this is not usually the case. Certainly a clever compiler would not know that one could delay transformations as is done in the new ELMHES.

Our techniques will always produce faster code, even on machines with conventional architecture. We have never resorted to assembly language. Thus our programs are transportable. Moreover there is still room for some improvement by using some assembly-language modules in critical places.

APPENDIX

Table 7 contains timing informations in the form of ratios of increased performance over existing EISPACK routines on the Fujitsu VP-200 and Hitachi

TABLE 7
EISPACK/MV RATIOS

ELMHES

	Hitachi S-810/20	Fujitsu VP-200
n	ratio	ratio
50	1.1	1.1
100	1.6	1.6
150	1.9	1.8
200	2.0	1.8
250	2.1	1.8
300	2.2	1.9

ORTHES ORTBAK[a]

	Hitachi S-810/20		Fujitsu VP-200	
	ORTHES	ORTBAK	ORTHES	ORTBAK
n	ratio	ratio	ratio	ratio
50	1.8	3.6	1.9	3.2
100	2.1	4.6	2.3	3.2
150	2.2	4.9	2.5	3.6
200	2.2	4.6	2.7	3.8
250	2.2	4.0	2.8	3.9
300	2.2	3.8	2.9	4.0

REDUC

	Hitachi S-810/20	Fujitsu VP-200
n	ratio	ratio
50	1.7	1.8
100	2.1	2.2
150	2.3	2.4
200	2.4	2.5
250	2.5	2.6
300	2.5	2.6

SVD, Hitachi S-810/20[b]

	$m = 100$		$m = 200$			
	$n = 50$	$n = 100$	$n = 50$	$n = 100$	$n = 150$	$n = 200$
novect	1.7	1.6	2.0	1.9	1.8	1.7
vect	2.0	1.7	3.0	2.5	2.2	2.0

SVD, Fujitsu VP-200[b]

	$m = 100$		$m = 200$			
	$n = 50$	$n = 100$	$n = 50$	$n = 100$	$n = 150$	$n = 200$
novect	1.6	1.5	1.9	1.8	1.7	1.7
vect	1.9	1.6	2.7	2.4	2.1	1.7

[a] Routines ORTHES and ORTBAK here are implemented using rank-1 updates only.
[b] "novect" refers to computing just the singular values and "vect" refers to computing both the singular values and left and right singular vectors. m is the number of rows and n the number of columns in the matrix.

810/20 as they existed in September 1984. The matrix-vector multiply routines, SMXPY and SXMPY, have been unrolled to a depth of eight for both the Fujitsu and Hitachi machines. A depth of eight gives the best performance on these machines. Subsequent hardware and software changes made affect the timing information to some extent. n refers to the order of the matrix; "ratio" is the execution time for the current version of the EISPACK routine divided by the time for the modified version.

REFERENCES

1 T. Chan, An improved algorithm for computing the singular values decomposition, *ACM Trans. Math. Software* 8:72–89 (1982).

2 J. J. Dongarra and S. C. Eisenstat, Squeezing the most out of an algorithm in CRAY Fortran, *ACM Trans. Math. Software* (3):221–230 (1984).

3 J. J. Dongarra and C. B. Moler, EISPACK—A package for solving matrix eigenvalue problems, in *Sources and Development of Mathematical Software* (W. R. Cowell, Ed.), Prentice-Hall, (1984).

4 J. J. Dongarra, Linda Kaufman, and Sven Hammarling, Squeezing the most out of eigenvalue solvers on high-performance computers, ANL-MCSD-TM/46, Argonne National Lab. Jan. 1985.

5 Kirby Fong and Thomas L. Jordan, Some linear algebra algoritms and their performance on the CRAY-1, Report UC-32, Los Alamos Scientific Lab. June 1977.

6 B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines—*EISPACK *Guide Extension*, Lecture Notes in Computer Science, Vol. 51, Springer, Berlin, 1977.

7 R. W. Hockney and C. R. Jesshope, *Parallel Computers*, J. W. Arrowsmith, Bristol, Great Britain, 1981.

8 L. Kaufman, Banded eigenvalue solvers on vector machines, *ACM Trans. Math. Software* 10(1):73–86 (1984).

9 C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Software* 5:308–371 (1979).

10 R. M. Russell, The CRAY-1 computer system, *Comm. ACM* 21(1):63–72 Jan. 1978.

11 B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines—*EISPACK *Guide*, Lecture Notes in Computer Science, Vol. 6, 2nd ed., Springer, Berlin, 1976.

12 J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford U. P., London, 1965.

13 J. H. Wilkinson and C. Reinsch (Eds.) *Handbook for Automatic Computation, Vol. II, Linear Algebra*, Springer, New York, 1971.

Received 18 December 1984; revised 23 September 1985