Theor Chim Acta (1987) 71: 169-186

Theoretica Chimica Acta

Extracting more than a few eigenvectors from a dense real symmetric matrix: Optimal algorithms versus the architectural constraints of the FPS-X64

Stephen T. Elbert

Ames Laboratory-USDOE*, Iowa State University, Ames, IA 50011

(Received October 6, revised December 4/Accepted December 5, 1986)

Ten widely available sets of routines, including HQRII, QCPE GIVENS and EISPACK 3, were evaluated for reliability, robustness, accuracy, speed, compactness, portability and simplicity. All were found lacking in one or more areas. Modified versions of the EISPACK routines TRED3, TQLRAT, TINVIT and TRBAK3 performed somewhat better. Changes to TINVIT were especially important for improved speed, accuracy and reliability. To achieve the maximum capabilities of the FPS-X64 series of computers access to table memory is required, but since the FORTRAN compiler does not allow this and there is no library support for the required operations, it was necessary to write three routines in APAL. The standard algorithm needs to be modified before full efficiency can be achieved for the back transformation.

Key words: Diagonalization — eigenvector — Inverse — Iteration — Householder — Givens, EISPACK — HQRII

1. Introduction

The need to determine wavefunctions and their associated frequencies (vibrations, energies) is pervasive throughout computational chemistry and physics. The solution of this problem is the solution of the eigenvalue problem. Given a real

* Operated for the US Department of Energy by Iowa State University under contract no. W-74-05-ENG-82. This work was supported by the Office of Basic Energy Sciences

symmetric matrix $\boldsymbol{A}$ of dimension n, a matrix $\boldsymbol{X}$ containing m orthogonal column vectors of length n and a diagonal matrix $\boldsymbol{D}$ whose diagonal element d_i is the eigenvalue of $\boldsymbol{X}_i$, the matrix representation of the problem, $\boldsymbol{AX} = \boldsymbol{XD}$, appears in many guises, not the least of which is the time-independent Schrödinger equation: $\boldsymbol{H\Psi} = \boldsymbol{E\Psi}$.

In *ab initio* quantum chemistry programs, the eigenvalue problem appears in the solution of both self-consistent field (SCF) and configuration interaction (CI) wavefunctions, although in quite different forms. In the SCF problem, eigenvectors for all occupied orbitals must be found, although all vectors, including unoccupied ones, are usually found. The extraction of the vectors in an *ab initio* SCF calculation is not the time-limiting step so the efficiency of the method is not as important as reliability and accuracy. Matrices in SCF problems are memory resident and may range in size from 10^1 to 10^3. CI calculations, on the other hand, typically involve large matrices that are not memory resident (in fact the matrix itself never exists as such in direct CI methods), only a few vectors are needed, typically less than 10, and the vector solution is the time-limiting step. The largest dimension feasible with current computers is on the order of 10^7.

As may be deduced from the disparity between SCF and CI environments, there is a considerable difference in the methods used to solve these two eigenvalue problems, even though they both deal with real symmetric matrices. The fact that the large matrices are usually sparse need not complicate the comparison. The basic properties of the methods are independent of whether the matrix is dense or not. The conventional distinction has been between methods dealing with small (memory resident) and large (non-memory resident) matrices. Small matrix methods usually involve an initial transformation, which becomes impractical if the matrix is not memory resident, before the actual solution is begun. Large matrix methods avoid this step. Given the amount of memory available on current systems however, where one million words (8 MB) may be considered small and the Cray 2 has 256 million words (2048 MB) of memory, this distinction is less appropriate. Indeed, large matrix schemes like the Davidson [1] method can generally find a single eigenvector for a small matrix faster than the standard small matrix methods.

A more appropriate division among methods is not the size of the matrix but the fraction of the eigenvectors computed. The crossover point between large and small matrix methods may be found by comparing the number of multiply and add operations (MAO's), to highest order, for a dense matrix. Large matrix methods require $C_l K_m n^2$ MAO's to obtain m vectors of length n. Here C_l is a proportionality constant and

$$K_m = \sum_{j=1}^{m} K_j = m\bar{K}_l$$

with K_j the number of iterations required to obtain vector j and $\bar{K}_l$ the average number of iterations per vector. Small matrix methods require $C_1 n^3 + C_2 n^2 m + C_3 \bar{K}_s n^2 m$ MAO's. C_1, C_2 and C_3 are again proportionality constants and $\bar{K}_s$ is

analogous to $\bar{K}_l$. When the eigenvectors are found using a QR (or a related method such as QL) the product $C_3\bar{K}_s$ is typically in the range of four to twelve, but since the most efficient algorithms, i.e., those using the inverse iteration technique, have C_3 essentially zero, the following analysis does not involve C_3. Equating the MAO's for the large and small matrix method and solving for m gives

$$m = \frac{C_1}{C_l\bar{K}_l - C_2} \cdot n \approx \frac{n}{\bar{K}_l}.$$

Reasonable values of the constants are one for C_l and C_2 and two-thirds for C_1, so when K_l is eight the small matrix methods are more efficient if at least 10% of the vectors must be computed. If the vectors could be found with an average of four iterations each, a possibility with a quadratically convergent method or a very good initial guess, the breakeven point goes up to about 20%. Such a high breakeven point is probably unrealistic for another reason. Many of the large matrix methods, Davidson [1] and Lanczos [2] in particular, need to find all of the eigenvectors in an expansion basis subspace that start with a dimension equal to the number of basis vectors sought. Convergence is enhanced by adding that number of basis vectors to the sub-space on each iteration. In a hypothetical case seeking 20% of the vectors, by the fourth iteration all the vectors of a matrix of dimension $0.8n$ must be found. It thus appears that when about 10% or more of the eigenvectors are required, a small matrix method should be used. Less than 10% of the dimension of the matrix is considered to be a few eigenvectors.

Although *ab initio* SCF methods may soon need to diagonalize Fock matrices of order 10^3, the construction of the matrix is an n^4 time step, whereas its solution is only n^3. Sufficient parallel computation may be able to reduce the construction phase to n^3, so efficient diagonalization methods may become important. Of more serious concern are semi-empirical SCF methods whose construction phase is already of order n^3. Calculations on biologically active molecules using 10^3 basis functions will soon be common. Purely empirical methods, such as the various Hückel approaches, require only n^2 time in the matrix construct phase. First principle local density function calculations by solid state physicists routinely find ten percent of the vectors of matrices of order 1000 to 2000. These calculations spend fully $\frac{2}{3}$ of their time finding eigenvectors in an algorithm that iterates to self-consistency. Since the matrix needs to be memory resident, the availability of memory has been a constraint in the past. Larger memories are putting an even heavier demand on the eigenvector routines. Problems of this nature involving 20 000 basis functions should be possible on the Cray 2, allowing the first realistic calculations of impurities in metal clusters. Solving large eigenvector problems are also common in computing both vibrational and photoionization spectra. Quantum molecular dynamics is another area requiring efficient diagonalization routines. In his studies of HF-HF, Truhlar [3] has diagonalized 300 matrices of dimension 948. Obviously there is a clear need for an efficient method to find more than a few, i.e., from ten percent to all, of the eigenvectors of a dense real symmetric matrix with dimensions of order 10^3 or 10^4.

2. Basic theory

The methods discussed here can be broken into four parts: (1) reduction to tridiagonal from; (2) evaluation of the eigenvalues; (3) evaluation of the eigenvectors in tridiagonal form and (4) transformation of the vectors back to the original matrix form. The operations are not necessarily carried out in this order. Methods that do not follow this pattern, for example Jacobi [4], are not competitive in solving an arbitrary symmetric matrix.

Householder's method [5] is the most efficient way to form the tridiagonal matrix. A similarity transformation

$$A_{i+1} = P_i A_i P_i, \qquad i = 1, 2, \ldots, n-2$$

is constructed from a set of reflectors,

$$P_i = I - u_i u_i^T / H_i, \qquad H_i = u_i^T u_i / 2$$

each one of which zeros out all elements of a row and column with the exception of the diagonal and codiagonal elements. This is usually implemented as a rank 2 update that first computes

$$p_i = A_i u_i, \qquad (1/3n^3 \text{MAO's})$$

and then

$$A_{i+1} = A_i - u_i q_i^T - q_i u_i^T, \qquad (1/3n^3 \text{MAO's}),$$

where

$$K_i = u_i^T p_i / 2H_i \qquad q_i = p_i / H_i - K_i u_i, \qquad (O(n^2) \text{MAO's}).$$

The total operation count is thus $2/3n^3$ MAO's, which is half the number of operations required using the Givens' plane rotation approach [6].

A wide number of methods have been used to compute the eigenvalues of the tridiagonal matrix. The most efficient are variations on the QR or QL methods. This step requires only order n^2 operations, but the complexity of the algorithms may preclude vectorization. This, and the fact that the eigenvalues are computed by an iterative method, may make the proportionality constant quite high (Parlett [7] uses a value of nine).

There are two common ways to get the eigenvectors of the tridiagonal matrix. The first and most reliable is to accumulate the plane rotations generated by the QR/QL process. This method is of order n^3, but is appropriate only if all the eigenvectors must be found. The second method is inverse iteration and is generally of order n^2. Since it finds the eigenvector of a given eigenvalue, the specification of whether to compute a few or all of the vectors is easily controlled. Much has been written about the stability of inverse iteration [7] and it is generally considered a safe technique. Inverse iteration usually converges in one or two cycles, but it does occasionally fail to converge. For this reason it may be necessary to provide a QR/QL routine as a backup if computing all the eigenvectors is appropriate.

The final step is to recover the eigenvectors of the original matrix. This may be accomplished by accumulating the reflectors

$$Z = \prod_{i=1}^{n-2} (I - u_i u_i^T / H_i) \cdot Z_0$$

either onto a unit matrix, as in the QR/QL variant using $2/3n^3$ MAO's, or onto the tridiagonal vectors using $1n^3$ MAO's.

Using inverse iteration, a total of approximately $5/3n^3$ MAO's is required. In contrast, when a QR/QL method is used to find the vectors, at least $10/3n^3$ MAO's will be required, and frequently two or three times that number.

3. Evaluation procedure

Since some numerical analysts consider the understanding of the basic theory "essentially complete" [7], the choice is really which is the best implementation. As anyone knows who has had one of the standard routines fail, this is not a trivial task. To guide the choice, a list of attributes that can be measured with some objectivity is required. These attributes include speed, compactness, accuracy, reliability, robustness, portability and simplicity.

Speed is an obvious and important attribute and one that is easy to measure. The simplest approach is to make direct measurements of the time required to carry out the various sections of the calculation. Reasonable estimates may also be made based on the formal complexity of the algorithm, the time required to carry out individual operations and, where appropriate, the number of iterations involved. Here, direct measurements were made for each of the four steps. Even when one process was embedded in another, the appropriate sections were timed independently.

Compactness is a measure of the memory required to carry out a task. Methods that store the matrix in packed or symmetric storage mode where $A(1) = A_{11}$, $A(2) = A_{12}$, $A(3) = A_{22}$, $A(4) = A_{13}, \ldots,$ which require only $n(n+1)/2$ words for the input matrix, are usually preferable unless the output vectors can overwrite the input matrix. No local storage should ever be necessary; array storage should be provided by the calling routine.

Accuracy means producing the correct results with sufficient precision. Some of the methods tested require the calling routine to specify a level of accuracy by giving a threshold value for convergence. Most of the time the algorithms used by the tested routines were capable of producing results with near full machine precision with no significant increase in time. For this reason it is preferable to have a routine that handles accuracy automatically and thus avoids a possible source of error by specifying an inadequate amount of precision in the calling routine. Measuring accuracy is not difficult but it is time consuming. Sometimes accuracy is measured by comparing the results with those produced by the same routine (or a more reliable one) in extended precision. This approach is not generally useful, as some machines do not support precision beyond the working

precision (generally a 64 bit word) and it may be very slow and expensive to use on machines that do support it. Such absolute error methods also make it difficult to compare the same routine on different machines. Accuracy is measured here by computing the normalized relative residual [8]

$$\rho = \max_{1 \le i \le n} \frac{\|AX_i - \lambda_i X_i\|}{10 \cdot n \cdot \varepsilon \cdot \|A\| \cdot \|X_i\|}.$$

The norms are 1-norms and ε is defined as the smallest value that may be added to one such that the result is different from one. The value of ten in the expression for ρ was chosen empirically so the following statements would hold: (1) a value less than one indicates satisfactory performance; (2) a value greater than 100 indicates poor performance and (3) a value between one and 100 indicates a progressively marginal performance.

Reliability refers to the probability of failure and as such is difficult to measure. The method should produce accurate results most of the time and signal when it cannot. Producing no result is much better than producing erroneous results. For this reason the routines were tested with pathological matrices to see how well they handled themselves in difficult situations. Failure to produce results should not be considered a deficiency in a high performance routine if the failures are infrequent and the failure is communicated to the driving routine, which can then call a slower, more reliable, routine.

Robustness is closely related to reliability. A robust routine fails gracefully and without surprises. A robust routine does not work well with one matrix and fail completely on a similar one. It takes more time or produces less accurate results in a predictable manner as the problem becomes more difficult. Robust routines do not abort in the middle of a calculation because of arithmetic exceptions.

To measure the reliability and robustness of the routines, several test matrices were used. Each of the matrices was defined in a manner that allows it to be generated for any given dimension. All the eigenvectors for each matrix were calculated using the following 22 dimensions: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50, 75, 100, 125 and 500. Note that an equal number of even and odd dimensions were tested. The first few values check special cases, while the final value was used for speed comparisons. Most of the matrices were chosen to present a variety of difficult situations so that a qualitative assessment of reliability and robustness could be made. Matrices representing real problems were also included to assess performance in a more normal environment. The following test matrices [9] were used:

1) NULL. This matrix is all zeroes. A good routine will set the eigenvalues to zero and return a unit matrix while doing very little work. A mediocre routine will return an error code. A bad routine will abort when it attempts to divide by zero.

2) DIAGDN. This is a diagonal matrix with descending values along the diagonal, i.e., $A_{ii} = n - i + 1$.

3) WILKWP. The Wilkinson W^+ matrix is a tridiagonal matrix whose diagonal elements are defined as $A_{ii} = [n/2] + 1 - \min(i, n-i+1)$ where $[b]$ is the largest integer less than or equal to b. The codiagonal elements all have a value of one. The separation of the two largest eigenvalues is $n!^{-2}$. A good method will spot this as a tridiagonal matrix.

4) WILKWM. The Wilkinson W^- matrix has the same form as the W^+ matrix except $A_{ii} = [n/2] + 1 - i$. For odd order, this matrix has pairs of eigenvalues that are equal in magnitude but opposite in sign. The magnitudes are close to some of those of the corresponding W^+ matrix.

5) ONES. This matrix, consisting entirely of ones, is of rank 1 (only one non-zero eigenvalue) and singular. Its solution in exact arithmetic is trivial, but rounding errors on some machines can result in disaster.

6) BORDER. This highly degenerate matrix ($n-2$ eigenvalues equal one) is zero everywhere except on the diagonal, where $A_{ii} = 1$, and in the last row and column, where $A_{in} = A_{ni} = 2^{1-i}$.

7) FRANK. This matrix, with $A_{ij} = \min(i, j)$, is reasonably well behaved.

8) MOLER. This matrix has diagonal elements $A_{ii} = i$ and off-diagonal elements $A_{ij} = \min(i, j) - 2$. It too is reasonably well behaved, although it has one small eigenvalue.

9) NESBET. All off-diagonal elements of this matrix are one and the diagonal elements are split into two ranges. For $i \leq 5$, $A_{ii} = 1 + 0.1(i-1)$. For $i > 5$, $A_{ii} = 2i - 1$. This matrix has been frequently used in testing large matrix methods.

10) HRING2. This matrix is the Hückel representation for two independent rings where one ring has even indices and the other odd indices. The matrix elements are zero except $A_{i,i-2} = A_{i-2,i} = -1$, $A_{1,n-1} = A_{n-1,1} = -1$ and $A_{2,n} = A_{n,2} = -1$. The resulting eigenvectors preserve this structure, i.e., except for the two lowest eigenvectors, every other element of the vector is zero. Methods that do not provide adequate orthogonality fail to do this.

11) DINDON. The ding dong matrix is defined as $A_{ij} = 0.5/(n-i-j+1.5)$. This matrix is always represented inexactly and has clusters of eigenvalues near $\pm\pi/2$.

12) HILSEG. The Hilbert segment of order n is defined as $A_{ij} = 1/(i+j-1)$. This matrix is notorious for its logarithmically distributed eigenvalues. Although positive definite, it is so ill-conditioned that Nash [9] claims most eigenvalue algorithms will fail for some value of $n < 20$.

Portability means a routine requires no changes other than those related to precision when changing machines. Portable routines to evaluate machine dependent constants are readily available. To check portability, each routine was run on a DEC VAX-11/780 and an NAS 9160 in addition to the FPS-164. The 9160 is an IBM compatible machine with a vector processing facility. A few selected routines were also run on a Cray X-MP/24 and on its compatible the SCS-40/14.

Finally there is *simplicity*. Simplicity means that a routine is easy to use and easy to modify. A routine is easy to use when it has a well-documented set of calling parameters, no local storage and no COMMON blocks. Since modifications at some point are inevitable, be they for tuning purposes or because a better algorithm has been found, routines must be modular. Modularity helps clarify the structure of the algorithm and insure that changes to one section do not affect others. Small, modular routines are much easier to tune and thus encourage more efficient utilization of resources.

4. Routines evaluated and results

Although it has not received the intensive development effort of LINPACK [10], EISPACK [8] is the standard for comparison when dealing with routines to solve eigenvalue equations. EISPACK was originally a collection of FORTRAN routines that were direct translations of Algol procedures developed in the 1960's by nineteen different authors [11]. One of the side effects of the translation was decreased performance due to inefficient memory access resulting from the different way arrays are stored in Algol (by row) and FORTRAN (by column). The latest edition, version 3, has resolved this problem, at least as far as symmetric matrices are concerned. Other improvements in version 3 include improved portability and a new driver, RSM, that uses inverse iteration. In earlier editions users needed to supply their own drivers to use the inverse iteration routines in EISPACK. Unfortunately, RSM does not use the more memory efficient packed form of the input matrix. The RS routine does not use packed form either but, because it uses a QL method to obtain vectors, it can accumulate the vectors on top of the input matrix. RSM can't do this because the reflectors need to be preserved until after the tridiagonal vectors are generated, resulting in twice the storage requirements of RS. The only driver to handle the input matrix in packed form is RSP, which also uses a QL method for the vectors but does not do the back transformation until after the tridiagonal vectors have been found. This makes it slower and less compact than RS. Since all the vectors must be found when using the QL approach, RSM is the only driver offering the choice of finding the m lowest vectors. Several improvements to this situation are immediately obvious. One would be to modify RSP to be as efficient as RS. Another would be to allow RSP access to the inverse iteration routines as RSM does. This latter change is very easy to do and is called RSPII here. RSPII has the same performance characteristics as RSM but uses less memory; in fact, if less than half the vectors are needed, RSPII uses less memory than RS.

The only routine to produce satisfactory results for all matrices on all three machines was RS (calling TRED2 and TQL2). From this standpoint it is reliable and accurate. Since there were no failures, graceful or otherwise, it appears to be reasonably robust, although the times produced by TRED2 for the ONES matrix, shown in Table 1, do not scale correctly, indicating room for improvement. This problem was caused by an inconsistent ability to finish the tridiagonalization after the first similarity transformation, a problem TRED3 also suffers from. Since it accumulates vectors from QL rotations, it is not one of the faster routines. RS

Table 1. Milliseconds to tridiagonalize ONES matrix for FPS-164

n	TRED2[a]	TRED3	ETRED3
10	0.65	0.51	0.56
15	1.61	1.34	1.43
20	4.73	3.41	2.11
25	6.26	5.82	2.94
30	9.62	13.17	3.80
35	7.84	7.06	5.01
40	19.40	19.42	6.24
45	33.66	32.13	7.59
50	50.81	50.78	9.14
75	79.01	84.46	18.80
100	95.21	100.76	31.90
125	366.08	416.37	48.49

[a] Tridiagonalization time only; reflector accumulation not included

is very compact, requiring only n^2+2n words of memory and very portable. Improvements to the clarity and structure would improve its simplicity. In particular, TRED2 could be made more modular by splitting it into separate routines to generate and accumulate the transformations. The version of RS supplied by FPS in their subroutine library did not do as well as the EISPACK RS. The FPS version indicated a failure to converge the first eigenvalue for any NULL matrix, for ONES with $n=5$, 7, 9, 15 and HRING2 with $n=4$. The APMATH64 Manual [12] indicates that IMTQL2 is used instead of TQL2.

As expected, RSP (calling TRED3, TQL2 and TRBAK3) generally produced results comparable to, but slower than, RS; requiring 6% more time on FPS, 10% on VAX and 21% on NAS. RSP is not as robust as RS, however, with TRED3 generating two overflow conditions on the NAS machine (ONES, $n=25$, 35) and one on the VAX (BORDER, $n=40$). The FPS is normally run with arithmetic exceptions disabled because the compiler will sometimes process undefined values at the higher optimization levels. The results indicated satisfactory operation for FPS. As previously noted, RSP is not a compact routine, requiring $(3n^2+7n)/2$ words of memory.

RSPII (calling TRED3, IMTQLV, TINVIT and TRBAK3) encountered the same overflows as RSP. In addition, TINVIT failed to converge for several cases. When it did converge, the values of the residual were generally larger than with RS or RSP. Values of ρ up to 6.2 were observed and the trend was towards larger values with larger matrices. RSPII thus had problems with reliability, robustness and accuracy. For $n=500$ it required only 26% of the time needed by RS on the FPS, 35% on the VAX and 50% on the NAS. It requires $n^2/2+nm+21/2$ words of memory, so its compactness depends on how many vectors are required. Its portability and simplicity are the same as the other EISPACK codes, which is to say the clarity and structure of the code could be improved. RSM (calling TRED1, IMTQLV, TINVIT and TRBAK3) should exhibit the same performance characteristics as RSPII except for not being as compact.

A common way to speed up codes to use the LINPACK BLAS [10]. These routines are usually specially coded in an assembly language for improved vector performance. A modification of RSPII, referred to as RSPIIB, uses S/DDOT and S/DAXPY in TRED3 and TRBAK3 and S/DNRM2 in place of PYTHAG in TINVIT. The effect on the time is shown in Table 2. The most dramatic change is in TINVIT, where PYTHAG is a very inefficient way to compute the Euclidean norm of a vector. S/DNRM2 is used instead of S/DDOT because of the potential for overflow here. Otherwise, the BLAS are a mixed blessing. The additional calling overhead slows down scalar machines and even vector machines when short vectors are used. Vector machines with good compilers may produce machine code as good as that available in the BLAS, but without the overhead of a subroutine call. TRED3 processes many short vectors, even for large matrices, thereby raising the payback level substantially when the BLAS are called.

There are a number of routines in common use that are similar to the EISPACK routines. One is DIAGD from the Gaussian 82 [13] set of programs. For comparison purposes DIAGD was modified to make it portable and to treat only real matrices. It will be called GDIAGD here. These routines appear to be an independent translation of the Algol routines that are the basis for EISPACK. It is similar to RSP, but with a different technique for handling tolerances. Satisfactory results were obtained for all matrices on the FPS and NAS machines. The VAX produced some very poor results for HILSEG ($n \geq 10$) and DINDON ($n = 25$). DIAGD uses G floating arithmetic on the VAX while GDIAGD uses the default D floating arithmetic, which has a smaller dynamic range. GDIAGD uses the older Algol-influenced addressing scheme in the equivalent of TRED3, so it is slower than RSP in that section and as fast as RSP elsewhere. The tridiagonalization times scaled correctly for the ONES matrix on the NAS machine, unlike RSP.

In the NRCC program GAMESS [14] there is a routine called GIVEIS written by Cleve Moler and Dale Spangler. It uses modified EISPACK2 versions of TRED3, IMTQLV, TINVIT and TRBAK3. The modifications consisted of improving the addressing scheme and calling the LINPACK BLAS. GIVEIS requires an explicit machine epsilon value, which has been left at 2^{-50} for all the machines used. It experienced the same overflow and other robustness problems

Table 2. Ratio of execution times RSPIIB/RSPII to diagonalize NESBET

Dimension	Machine	TRED3	TINVIT	TRBAK3
$n = 50$	FPS-164	1.41	0.32	1.03
	VAX-11/780	2.24	0.49	1.11
	NAS 9160	0.95	0.41	0.53
$n = 500$	FPS-164	0.90	0.31	0.75
	VAX-11/780	1.24	0.58	1.08
	NAS 9160	0.30	0.45	0.20
	Cray X-MP	1.27	0.34	0.82

as RSP. If TINVIT fails to converge, which happened (though not as often as it did for RSPII), TQL2 is called if all eigenvectors are being computed. The version of TINVIT used by GIVEIS suffers from another problem that was remedied in version 3, namely a grouping tolerance for degenerate vectors that was too right. This resulted in excessive effort going into orthogonalizing the vectors. This can be seen in the $O(n^3)$ time spent in TINVIT for the NESBET matrix (Table 3). The major flaw is its unreliability. Completely wrong results were produced without warning for the DINDON matrix around $n = 50$ on all machines and for NESBET ($n = 500$) on the NAS.

Several routines whose development was unrelated to EISPACK were also evaluated. The GAMESS program had another routine named LIGENB, which uses a packed input matrix, that is a modified version of the Gaussian 82 routine EIGEN, which does not use a packed input matrix. The version used here was further modified to remove local storage and to use the BLAS S/DROT to perform the rotations. LIGENB accumulates vector rotations and is similar to RS in speed, but it is very unreliable. The FPS version doesn't work at all and there were overflows and poor results produced frequently on the other two machines.

Long a favorite in many programs because it was faster than the standard EISPACK drivers is the QCPE routine GIVENS written by Franklin Prosser [15]. This routine uses a Sturm sequence method to find the eigenvalues and inverse iteration to find the vectors. As with other older code, machine dependent constants need to be set. This version (62.3) was modified to call the EISPACK3 routine EPSLON to obtain a value of the machine epsilon. The value of THETA, specifying the dynamic range, was set to 10^{37} for all machines. For most cases the routine produced satisfactory results, certainly better than routines calling TINVIT, but there were several cases of poor results for BORDER and HRING2. There is no error return code to warn of these cases. It is certainly not a simple routine.

Another recently popular routine that has spawned many local versions is HQRII [16]. The original version was similar to RSM (the unpacked form of the matrix was used) except that a QR method was used instead of QL. The version used here has been adapted for packed matrices and was obtained from QCPE. QCPE supplies another routine, DIAHOU, which is yet another version of HQRII that uses a Givens' bisection method when $m/n < 1/4$. There were minor, but not notable, differences in the performance of the two routines, so only HQRII is discussed here. The eigenvalue convergence tolerance, which is user specified, was set to 16·EPSLON(1.0). The routine produced wrong results for ONES, BORDER, HRING2, DINDON and HILBERT and has no error reporting mechanism. Divide checks occurred for NULL. When it worked, the results were generally better than TINVIT. The QR and inverse iteration sections were slightly faster than RSPII equivalents, but the back transformation was noticeably slower, making the total slower than the EISPACK3 routines called by RSPII. The routine is not modular, indeed the back transformation and the inverse iteration sections are intermixed.

Table 3. Seconds to diagonalize NESBET for $n = 500$

Driver	3-Diag	Values	3d-Vec	Backtr	Total	ρ
A. FPS-164						
RS	45.764	23.719	535.893	56.044	661.419	3.97E−02
RS/FPS[a]	—	—	—	—	238.492	3.86E−02
RSP	54.378	23.717	535.934	84.893	699.018	3.88E−02
RSPII	54.378	18.130	17.625	84.893	175.027	3.68E+00
RSPIIB	49.183	18.130	5.524	63.836	136.675	3.68E+00
GDIAGD	207.143	12.681	535.853	84.944	840.724	1.01E−01
GIVEIS	53.251	7.950	67.135	64.115	192.452	1.35E+01
EVVRSP	54.112	2.409	7.491	63.954	127.966	3.81E−02
EVVRSPTM[a]	16.493	2.409	7.492	36.224	62.618	3.88E−02
GIVENS	221.086	109.438	15.304	85.418	431.545	1.01E−02
HQRII	54.488	6.171	10.863	134.541	206.064	1.98E−02
SHQRII	145.324	8.844	159.256	87.079	400.506	5.43E+03
B. VAX-11/780						
RS	987.39	453.31	5857.23	1025.76	8323.72	1.94E−02
RSP	1051.55	486.63	5896.26	1687.75	9125.04	2.17E−02
RSPII	1016.25	73.04	76.73	1713.74	2879.81	5.54E+00
RSPIIB	1261.58	72.87	44.33	1851.91	3230.75	5.54E+00
GDIAGD	1199.87	462.98	5280.64	1554.52	8502.85	4.29E−02
GIVEIS	1013.67	51.13	1863.59	1825.51	4753.92	7.82E+00
EVVRSP	1019.88	23.28	60.30	1866.61	2970.13	1.04E−02
LIGENB	1341.73	461.95	5541.59	2528.20	9873.50	4.64E−01
GIVENS	1366.75	385.99	117.65	1637.31	3509.68	3.51E−02
HQRII	1196.81	49.25	75.39	2640.34	3961.84	2.76E−02
SHQRII	1501.18	70.43	2073.42	2089.92	5734.98	7.08E+03
C. NAS 9160						
RS	26.316	14.076	127.654	32.809	200.857	1.89E−01
RSP	31.042	14.273	130.146	66.795	242.316	1.93E−01
RSPII	30.927	3.495	3.510	62.616	100.549	7.48E+00
RSPIIB	9.283	3.544	1.583	12.277	26.687	7.48E+00
GDIAGD	45.607	13.068	140.855	49.576	249.159	1.01E−01
GIVEIS	28.039	2.311	12.158	12.349	54.858	3.90E+10
EVVRSP	30.714	0.920	1.869	11.729	45.232	8.27E−02
LIGENB	40.955	13.207	50.592	77.252	182.007	2.30E+00
GIVENS	50.977	16.819	3.820	49.254	120.926	6.40E−02
HQRII	34.409	2.157	2.237	77.381	116.185	8.51E−02
SHQRII	39.157	3.728	38.762	45.567	127.214	1.58E+04
D. Cray X-MP/24 (1 proc.)						
RS	2.434	5.416	9.900	2.901	20.651	5.53E−02
RSP	2.375	5.416	9.905	4.767	22.469	5.55E−02
RSPII	2.374	2.060	1.988	4.734	11.156	9.02E+00
RSPIIB	3.016	2.060	0.672	3.875	9.622	9.09E+00
EVVRSP	2.354	0.332	0.868	3.830	7.384	2.31E−02
E. SCS-40/14						
RS	7.411	22.418	42.518	7.668	80.016	5.53E−02
RSP	7.250	22.422	42.522	13.177	85.397	5.55E−02
RSPII	7.250	5.395	5.707	13.177	31.530	9.02E+00
RSPIIB	9.388	5.393	2.338	12.503	29.623	9.09E+00
EVVRSP	7.167	1.034	3.159	12.497	23.857	2.31E−02

[a] Uses APAL routines other than BLAS

The most recent proposed improvement on HQRII is called SHQRII [17]. This is a good example of how not to improve a program. SHQRII still produces the wrong results for ONES, BORDER, HRING2, DINDON and HILBERT observed for other versions of HQRII. Although the reporting of some trivial input errors is improved, there is still no warning the routine has failed. The divide check in NULL is avoided, but new ones occur on the VAX for BORDER and HRING2. The NAS generated an overflow on BORDER. Many additional cases of poor and marginal performance showed up. It was slower (by nearly a factor of two on the FPS) on all three machines. It uses a non-standard form of packed storage (by rows instead of columns) that complicated the addressing. The routine is much more complicated since it uses loop unrolling and jamming, which defeat many vectorizing compilers. It is not modular, so the complexity is harder to see through. A complicated quicksort using static local storage, with no checks for stack overruns, was added with essentially no improvement in total time (the change of an $O(n^2)$ step to $O(n\log_2(n))$ in an $O(n^3)$ algorithm is nearly negligible). An inflexible COMMON block is used to transfer parameters and data (what per cent of the time is saved this way in an $O(n^3)$ routine?).

EVVRSP is an attempt to produce a routine that gets high ratings for each of the seven desirable attributes; combining the reliability and accuracy of RS with the speed of RSPII, or rather RSPIIB since the replacement of PYTHAG with S/DNRM2 in TINVIT has already been shown to be advantageous. It should be clear by now that one wants to use EISPACK as a starting point because, even though not perfect, it has the most carefully thought out codes available so far.

There are two key changes that improve TINVIT's reliability and accuracy. The first is to scale up the value of EPS3. This increases the magnitude of the initial guess vector and reduces the number of situations where TINVIT fails to converge. Unfortunately it also reduces the accuracy of the vector, but that may be partially offset by the second change, which is to force one more iteration after convergence. This is actually nothing more than following Wilkinson's original suggestion [18]. It is a good idea even if the value of EPS3 is not increased. TINVIT is the only inverse iteration routine examined that allowed an exit after only one iteration. The result of this can be disastrous and lead to the inclusion of HRING2 as a test matrix. The extra iteration does not take that much of the total time. The problem of accuracy and convergence seems to call into equation the meaning of the current convergence test. The amount to scale EPS3 is an empirical choice. Too small a value does not aid convergence and too large hurts accuracy. With a scale factor of 16, satisfactory converged results were obtained for all matrices except DINDON and HILSEG. Of the nonconverged cases, only one result was poor (DINDON, $n=45$, on VAX), two were marginal (DINDON, $n=40$; HILSEG, $n=45$, on NAS) and the rest were satisfactory (TINVIT was further modified to allow processing to continue even when not converged). This suggests the possibility of proceeding, albeit with caution, even after TINVIT warns of non-convergence. The fewest cases of non-convergence occurred with a scale factor of 64, but at the cost of an increase in the number of marginal results.

Instead of using IMTQLV to find the eigenvectors, as RSM and RSPII do, TQLRAT was modified to support the sub-matrix blocking that is an advantageous feature of TINVIT. The inner loop of TQLRAT was modified to remove an IF statement, permitting the loop to vectorize. The inner loop of IMTQLV computes a square root that inhibits vectorization on most machines, but there is no square root in the inner loop of TQLRAT. The modified TQLRAT is generally three or more times faster than IMTQLV. EISPACK warns that TQLRAT may not be accurate enough for TINVIT, but so far that does not seem to be the case. There are faster and more reliable algorithms than TQLRAT available [7] and they will be implemented as time permits.

TRED3 was modified so that the vectors p_i and q_i are constructed in a separate subroutine as is the rank 2 update. Each of these new, easy to tune, routines carries out $1/3n^3$ MAO's after being called n times. The new routines are similar in function to two of the recently proposed level 2 BLAS [19] (S/DSPMV and S/DSPR2). The modified routine, ETRED3, makes more reasonable choices about when a row and column are already in correct form to machine accuracy. It can therefore skip matrices that are already tridiagonal and does not have the robustness problems seen in TRED2 and TRED3 when dealing with the ONES matrix (Table 1).

5. FPS-X64 implementation

The FPS-X64 series of computers is perhaps unique in the way that it compiles FORTRAN code directly into microcode, with all the concurrency control of the functional units that implies. Hardware pipelines are common now in everything from microprocessors to supercomputers, but rarely is the user able to control these resources on a cycle by a cycle basis. Each 64-bit instruction can control a variety of independent functions simultaneously and one instruction may be issued every cycle. The functional units include a three stage multiply pipeline, a two stage add pipeline, a three stage main memory pipeline, a two stage table memory pipeline, an integer arithmetic unit and a branch control unit.

The power of the instruction set may be seen in the fact that the loop needed to carry out the dot product of two vectors may be expressed as a single instruction. This is accomplished with a technique called software pipelining that rolls the elements of the operation as they would be performed sequentially into a set of overlapped microinstructions. In the case of the dot product, the operations that would take seven cycles if done sequentially may be treated as a seven stage pipeline consisting of a single instruction, producing a floating point multiply and add (2 FLOPS) every cycle. This is the full theoretical speed of the machine and it can be sustained with these limitations: versions of the machine with dynamic memory will lose about 4% of their throughput when the machine stops to refresh the contents of memory (static memory does not need refreshing), the addressing must be such that successive vector elements are not in the same memory bank (bank conflict), and the length of the vectors is limited by the amount of table memory.

Table memory is a smaller, faster memory that provides an additional data path to the arithmetic pipelines. These pipelines run at only half speed when all the data must come from main memory because only one word is fetched or stored in a single cycle. With table memory, one vector element of the dot product may come from each memory to keep the multiply pipeline going full speed while the adder accumulates the result. A serious deficiency of the FPS-X64 FORTRAN compiler is that it never uses table memory, so for most applications it can achieve only half of the machine's capacity at best. There are library routines that use table memory and these may be adequate in many cases, but there are restrictions that limit their usefulness. One restriction is that, since the data can't normally be generated in table memory, the values must be copied there at a cost of at least two cycles per element. If the vector is only used once, the throughput has dropped below the level that would have been achieved without table memory. The cost of loading the vector into table memory may be amortized over several uses, as in a matrix-vector product, just as the startup cost of the pipelines may be amortized over long vectors.

If all code could be written in terms of dot products, it would not be so difficult to make full use of the machine. Unfortunately the vector outer product, as performed by the BLAS SAXPY, is an operation that occurs frequently and this is more difficult for the X64 to handle. In the vector outer product, each element of a vector is multiplied by a vector and added to a second vector that may be stored as a third vector or back onto the second vector, an operation that needs to access three vectors simultaneously in order to operate at full speed. SAXPY thus requires two cycles using table memory and runs at half the maximum FLOP rate; three cycles are necessary without table memory. The table memory version of SAXPY, which can be expressed as a four stage, two cycle loop, is not normally supplied by FPS. The use of specially coded versions of SDOT and SAXPY is usually justified because the compiler generates an extra cycle for each of these operations, yielding three cycle loops for SDOT and four cycle loops for SAXPY.

The two BLAS, SDOT and SAXPY, are sufficient to vectorize the key $O(n^3)$ loops in TRED3 and TRBAK3. In TRED3 the formation of $p_i = A_i u_i$ consists of a loop with both an SDOT and a SAXPY and the rank 2 update $A_{i+1} = A_i - u_i q_i^T - q_i u_i^T$ can be expressed, somewhat less effectively, as two SAXPY's. In TRBAK3 a single SDOT followed by a SAXPY is all that is needed. The FPS-164 running at 5.5 MHz is capable of 11 MFLOPS. Without table memory and ignoring overhead, the limit should be eight FLOPS in eleven cycles or four MFLOPS for TRED3 and four FLOPS in five cycles or 4.4 MFLOPS for TRBAK3. For $n = 500$, the FORTRAN versions of TRED3 and TRBAK3 achieved 3.1 and 2.9 MFLOPS or 77 and 66% of the limit. For TRED3 the compiler actually did better than would have been possible with FORTRAN BLAS, which would have yielded 2.9 MFLOPS. The compiler was able to exploit redundant memory references in the loops that disappear when the BLAS are used. Nevertheless, using the BLAS changed the performance to 3.4 and 3.9 MFLOPS and 85 and 89% of the no table memory limit. If table memory versions of SDOT and SAXPY were used, TRED3 would use only seven cycles for 6.3 MFLOPS and TRBAK3 three cycles

for 7.3 MFLOPS. The architecture thus places a significant constraint on the straightforward implementation of the algorithm.

It is possible to overlap the memory fetches of the first inner loop in TRED3 because elements of A_i appear in both the inner and outer product expressions. The outer product portion of the loop then has only two memory references to contend with instead of three. The result is four FLOPS in two cycles instead of three cycles and maximum efficiency is achieved. To accomplish this, p_i should be in table memory, because both its input and output elements have to be in the same memory and the overhead to put A_i in table memory would be too large. This approach is unworkable because of another unfortunate constraint; a store to table memory and an add can't be performed in the same cycle. That means three of the four memory references must be to main memory, which unbalances the situation again. Unrolling the inner loop to a depth of two would use four instructions involving two references to A_i but still only two references to p_i. All the addressing can't be handled in four instructions however. It was finally necessary to unroll to a depth of 8, producing a two stage, 16 cycle loop performing 32 FLOPS. It should be noted that 16 is the maximum number of instructions the loop branch could handle given everything else that was going on. Even at this level of complexity it was not possible to guarantee an absence of bank conflicts. If the first elements of A_i and p_i are in the same bank, the loop takes an extra cycle to complete. With a large A_i, arranging p_i to be stored after A_i should avoid this situation. Handling the special cases required by such extensive unrolling resulted in a routine with nearly 500 instructions (the source code has over 4000 lines of text). This much code adds somewhat to the overhead because now the machine must stop occasionally to reload the instruction cache from memory.

Coding the rank 2 update in TRED3 directly saves a memory store, resulting in four FLOPS and four memory accesses for two instructions. Again, to gain enough instructions to do the addressing, the loop was unrolled to a depth of two resulting in a three stage pipeline of four cycles. To reduce indexing this time, the vectors p_i and q_i were interleaved in table memory. By calling these two APAL routines, TRED3 achieves 10.1 MFLOPS on the 164 for $n = 500$, or 92% of the limit.

The back transformation, a relatively simple routine, was coded entirely in APAL to avoid the overhead of n^2 calls to the level 1 BLAS. It achieved 6.9 MFLOPS, or 94% of the 7.3 MFLOPS possible with the straight use of table memory versions of SDOT and SAXPY. If two reflectors are accumulated at a time, the vector matrix update can be expressed as

$$Z_{j+1} = Z_j - h_i u_i v_i^T - h_{i+1} u_{i+1} w_{i+1}^T$$

at a cost of $O(n^2)$ additional MAO's. Here $h_i = 1/H_i$, $v_i^T = u_i^T Z$ and

$$w_{i+1} = v_{i+1}^T - h_i (u_{i+1}^T u_i) v_i^T.$$

This is now similar to the rank 2 update in TRED3 and could be implemented with either v_i and w_{i+1} or u_i and u_{i+1} in table memory. The latter has the disadvantage of requiring a copy from main memory to table memory, but the

advantage or sequential addressing in Z. The former could be created in table memory but would require careful coding to avoid bank conflicts in Z, which is addressed by row. In either case, performance near the machine's theoretical limit should be possible for TRBAK3 just as it is for TRED3.

6. Conclusion

Choosing the best routine to extract from 10–100% of the eigenvectors from a dense real symmetric matrix is not a simple matter. If not all the vectors are needed, the choice is narrowed to a routine using the inverse iteration method. None of the routines tested converged in all cases, so only routines that warn of a failure to converge should be considered. The EISPACK 3 inverse iteration routine TINVIT, although returning convergence information, was not as accurate as some other inverse iteration routines. EINVIT (a modification of TINVIT and one of a set of routines driven by EVVRSP) converges more often, with greater accuracy and in less time than the original.

If all the vectors are needed, the choice is more complicated. The speed and accuracy for each of the routines to find all the eigenvectors of a NESBET matrix of dimension 500 is given in Table 3 for each of the three machines used to measure portability. Similar results for the EISPACK 3 based routines is also given for a Cray X-MP/24 (using only one processor) and an SCS-40/14 which is compatible with the Cray machine. The extent of the compatibility is reflected in the complete agreement of the values of ρ. If accuracy, reliability and compactness are more important than speed, then RS is the best choice. If speed is the major consideration, then inverse iteration routines should be considered. Although EVVRSP was among the fastest routines on all the machines tested, it is relatively easy to tune the key $O(n^3)$ parts of any well-written, modular routine to make it competitive on a given machine. Unfortunately, the architectural features of the fast machines currently available make them sensitive to different tuning techniques. What works well for the FPS may be detrimental for other machines. On vector machines, it is likely that the nominally $O(n^2)$ sections will impact, or even dominate, performance if they are not vectorized, except for very large values of n. This is particularly true of inverse iteration methods, where partial pivoting in the Gaussian elimination is not readily vectorizable. Some consideration of these points was made in EVVRSP, where the QL section and the orthogonalization process in the inverse iteration section now vectorize; but more work needs to be done.

High memory bandwidth (with interleave levels adequate to avoid bank conflicts) would make the coding of these algorithms much easier and more general while maintaining high levels of theoretical capacity. What is really needed is multiple (at least three) data paths and compilers that know how to use them. In the meantime, tuning general routines like EISPACK to specific architectures can achieve significant increases in throughput. The FORTRAN version of EVVRSP is available upon request through BITNET. The requests should be sent to ELBERT@ALISUVAX.

Acknowledgement. Time on the NAS 9160 provided by the Iowa State University Computation Center is gratefully acknowledged.

References

1. Davidson ER (1975) J Comput Phys 17:87
2. Lanczos C (1950) J Res Nat Bur Stand Sect B 45:225
3. Schwenke DW, Truhlar DG (1986) Theor Chim Acta 69:175
4. Jacobi CGJ (1846) J Reine Angew Math 30:51 (see [7] for a review of modern methods)
5. Householder AS (1958) J Soc Ind Appl Math 6:6
6. Givens W (1954) Numerical Computation of the Characteristic Values of a Real Symmetric Matrix, ORNL-1574, Oak Ridge National Laboratory, Oak Ridge, Tenn
7. Parlett BN (1980) The Symmetric Eigenvalue Problem, Prentice-Hall, Englewood Cliffs, NJ
8. Smith BT, Boyle JM, Garbow BS, Ikebe Y, Klema VC, Moler CB (1976) Matrix Eigensystem Routines, Lecture Notes in Computer Science 6, 2nd ed., Springer, Berlin Heidelberg New York
9. (a) DIAGDN, WILKWP, WILKWM, ONES, BORDER, FRANK, MOLER, DINDON and HILSEG are described in Nash JC (1978) Compact Numerical Methods for Computers: linear algebra and function minimisation, Appendix 1, John Wiley and Sons, New York; (b) WILKWP and WILKWM are also described in [17]; (c) NESBET is described in Moler CB, Shavitt I (eds) (1978) Numerical Algorithms in Chemistry: Algebraic Methods-LBL 8158. Lawrence Berkeley Laboratory, University of California, Berkeley, CA
10. Dongarra JJ, Moler CB, Bunch JR, Stewart GW (1979) LINPACK Users Guide, SIAM, Philadelphia
11. Wilkinson JH, Reinsch C (1971) Handbook for Automatic Computation, Volume II-Linear Algebra, Springer, Berlin Heidelberg New York
12. APMATH64 Manual, Floating Point Systems, Portland (1985)
13. Hehre WJ, Random L, Schleyer P von R, Pople JA (1986) Ab Initio Molecular Orbital Theory, Wiley-Interscience, New York
14. (a) Dupuis M, Sprangler D, Wendolowski JJ (1980) NRCC Software Catalog 1, Program No. QG01 (GAMESS), Lawrence Berkeley Laboratory, University of California, Berkeley, CA; (b) Moler CB, Spangler D (1980) NRCC Software Catalog 1, Program No. ND03 (GIVEIS), Lawrence Berkeley Laboratory, University of California, Berkeley, CA
15. Quantum Chemistry Program Exchange, Chemistry Department, Indiana University, Bloomington
16. (a) Beppu Y, Ninomiya I, (1982) Comput Chem 6:87 (b) Ramek M (1984) Comput Chem 8:227
17. Tomašić ZA (1985) Comput Chem 9:123
18. Wilkinson JH (1965) The Algebraic Eigenvalue Problem, Clarendon Press, Oxford
19. Dongarra, JJ, Du Croz J, Hammarling S, Hanson RJ (1986) Mathematics and Computer Science Division Technical Memorandum No 41, Argonne National Laboratory, Argonne, IL