A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem

J. Rutter

Report No. UCB/CSD 94/799

February 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem

Jeffery D Rutter †
Department of Mathematics
University of California at Berkeley
Berkeley, CA 94720

February, 1994

This also appears as
Computer Science Division Report UCB//CSD-94-799,
University of California, Berkeley.

---

†The author acknowledges the support by subcontract no. 20552402 to Argonne National Laboratory at the University of Chicago, under Department of Energy contract W-31-109-Eng-38, as well as NSF grant ASC-9005933. This thesis was submitted in partial satisfaction of the requirements for the degree Master of Arts in Mathematics in February 1993. The committee in charge consisted of Profs. J. Demmel, B. Parlett, and R. Fateman.

Abstract

This report discusses a serial implementation of Cuppen's divide and conquer algorithm for computing all eigenvalues and eigenvectors of a real symmetric matrix $T = Q\Lambda Q^T$. This method is compared with the LAPACK implementations of QR, bisection/inverse iteration, and root-free QR/inverse iteration to find all of the eigenvalues and eigenvectors.

On a DEC Alpha using optimized Basic Linear Algebra Subroutines (BLAS), divide and conquer was uniformly the fastest algorithm by a large margin for large tridiagonal eigenproblems. When Fortran BLAS were used, bisection/inverse iteration was somewhat faster (up to a factor of 2) for very large matrices ($n \geq 500$) without clustered eigenvalues. When eigenvalues were clustered, divide and conquer was up to 80 times faster. The speedups over QR were so large in the tridiagonal case that the overall problem, including reduction to tridiagonal form, sped up by a factor of 2.5 over QR for $n \geq 500$.

Nearly universally, the matrix of eigenvectors generated by divide and conquer suffered the least loss of orthogonality. The smallest eigensystem residual usually came from the eigensystem generated by bisection/inverse iteration, with divide and conquer coming a close second.

Contents

List of Figures

Chapter 1

Introduction

A broad range of applications involve real symmetric eigenproblems [18]. Matrices arising in such applications may be dense, irregularly sparse, banded, or even tridiagonal. The typical method for solving non-tridiagonal symmetric eigenproblems first involves reduction to a tridiagonal matrix T which is similar to (a projection of) the original matrix: $U^TAU = T$ where U is orthogonal (for simplicity of exposition, we will assume that these matrices are all $n \times n$). This is accomplished by applying a series of Givens rotations or Householder reflections or by the Lanczos method.

There are many ways to compute the eigendecomposition $T = Q\Lambda Q^T$ of the symmetric tridiagonal matrix T. Once this has been done, the eigendecomposition of the full matrix can be computed by multiplying the two resulting orthogonal matrices: $A = UTU^T = UQ\Lambda Q^TU^T = Z\Lambda Z^T$. Thus the eigenvalues Λ of A are the same as the eigenvalues Λ of T and the eigenvectors Z of A are simply related to the eigenvectors Q of T.

The symmetric tridiagonal eigenproblem has been attacked in many different ways, including QR and QL methods, divide and conquer algorithms such as Cuppen's method, Laguerre iteration, Toda flow, Rayleigh quotient iteration, Jacobi's method, homotopy method, and bisection (or another method for find eigenvalues only) with inverse iteration [18, 6, 4, 7, 5, 17, 13, 15, 16]. QR is the most commonly used method.

In this report we will be comparing Cuppen's divide and conquer method, bisection with inverse iteration, the QR method, and root-free QR with inverse iteration.

1.1. Results

This report discusses a serial implementation of Cuppen's divide and conquer algorithm for computing all the eigenvalues and all the eigenvectors of a real symmetric tridiagonal matrix [4]. Our implementation incorporates recent improvements of Li [14], Gu and Eisenstat [10], and Kahan [12]. In contrast to earlier implementations, ours is designed to work correctly on the Cray XMP, Cray YMP, Cray C90, Cray 2, and other machines which similarly lack guard digits in their floating point arithmetic.

The method is compared to the LAPACK implementations of QR iteration (`ssteqr`), bisection followed by inverse iteration (`sstebz` and `sstein`), and root-free QR followed by inverse iteration (`ssterf` and `sstein`). Although we did not directly test the algorithms based on homotopies and Laguerre iteration in [13, 16], we believe our testing of both bisection with inverse iteration, as well as root-free QR followed by inverse iteration, will bracket the behavior of these algorithms (this is suggested but not guaranteed by the data in [13, 16], which used different test matrices, so further testing would be of interest). All implementations use real single precision IEEE arithmetic [2]. The algorithms are compared for speed, accuracy of the eigenvalues and eigenvectors (measured by residuals), and orthogonality of the eigenvectors. We use a variety of dense and tridiagonal test matrices of dimensions from 5 to 1000. The tests were run on a DEC Alpha (DEC 3000/500X), using both Fortran BLAS (Basic Linear Algebra Subroutines), and highly optimized BLAS. They were also run on a DEC 5000 and Sparc 2 using Fortran BLAS; the relative performance of the algorithms was essentially the same as on the DEC Alpha with Fortran BLAS.

The results are summarized as follows. When using optimized BLAS, divide and conquer is uniformly fastest for matrices of dimension $n \geq 30$, and essentially identical in speed to QR for $n \leq 30$. Speedups over QR increase with increasing dimension, ranging up to 15 for tridiagonals resulting from reducing uniformly random dense matrices to tridiagonal form, and up to 54 for tridiagonals with geometrically distributed eigenvalues. Speedups over bisection/inverse iteration depend even more strongly on test matrix type, decreasing from 5 to 1.5 for tridiagonals from uniformly random dense matrices, and increasing to over 70 for tridiagonals with geometrically distributed eigenvalues. The speedup over QR for the tridiagonal problem is so large that the overall dense symmetric eigenproblem speeds up by a factor of 2.5 for $n \geq 500$ — QR takes up to 3 times the time of the reduction to tridiagonal form whereas divide and conquer takes only about half the time of the reduction to tridiagonal form. When Fortran BLAS are used, the relative advantage of divide and conquer is smaller, and in fact for reduced random dense matrices, bisection/inverse iteration is fastest for $n \geq 500$, by up to a factor of 2. With Fortran BLAS and geometrically distributed eigenvalues, divide and conquer is again much faster.

As for accuracy, divide and conquer nearly always computes the most orthogonal eigenvectors of any of these algorithms; in all cases orthogonality is close to full machine precision. Bisection/inverse iteration usually produces the smallest eigensystem residuals, with divide and conquer a close second (and again always good to nearly full machine precision). In contrast, in some cases with tightly clustered eigenvalues, bisection/inverse iteration completely loses orthogonality.

The only drawback of divide and conquer for the problem of computing all eigenvalues and all eigenvectors is its need for much more workspace than either QR or bisection/inverse iteration: either $2n^2$ or $3n^2$. If this space is available, if the full eigendecomposition is desired, and if either highly optimized BLAS are available or highly clustered eigenvalues are possible, we recommend divide and conquer as the algorithm of choice. The software will soon be available as part of the LAPACK library [1].

1.2. Notation

In general, upper case Roman letters will denote matrices, lower case Roman letters will denote column vectors, lower case Roman letters that are italicized and subscripted will denote the entries in a vector (eg. $\mathbf{x} = (x_1, \ldots, x_n)^T$), and lower case Greek letters will denote scalar quantities. A symmetric tridiagonal $n \times n$ matrix will be denoted T, with entries $a_1, \ldots, a_n$ along the diagonal and $b_1, \ldots, b_{n-1}$ along the super- and subdiagonals. A symmetric dense $n \times n$ matrix will be denoted A. An eigendecomposition of A or T will be denoted $Q\Lambda Q^T$, where Λ is an $n \times n$ diagonal matrix with the eigenvalues as its diagonal entries. The $n \times n$ matrix Q is orthogonal and its columns are the eigenvectors corresponding to the eigenvalues in Λ.

A unit vector in the i^{th} dimension will be represented by $\mathbf{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$—the 1 occurs in the i^{th} position.

At times it will be necessary to distinguish between actual values and computed values. A circumflex over a symbol will denote a computed quantity, eg. $\hat{\mathbf{x}}$.

The eigenvalue of a matrix T which is largest in magnitude will be denoted $|\lambda|_{max}$. The corresponding computed value is $|\hat{\lambda}|_{max} \approx \|T\|_2$.

1.3. Outline of Report

The report is organized as follows: Chapter 2 reviews the bisection, inverse iteration, QR, and root-free QR algorithms. Chapter 3 gives a mathematical presentation of Cuppen's method. Chapter 4 describes the software in more detail. Chapter 5 presents numerical results including a theoretical performance analysis in section 5.6. The final chapter mentions several possible directions for extension of this research.

The two appendices contain information which may be helpful in understanding some portions of this document. Appendix A contains the calling sequence as it appears in the actual `sstedc` code. Appendix B roughly details how the code developed from its first incarnation to its final form.

Chapter 2

Bisection, Inverse Iteration, and QR

We will now briefly describe the competing algorithms.

2.1. Bisection Inverse Iteration

Bisection is based on Sylvester's Law of Inertia; specifically, this is used to compute the number of eigenvalues in a particular interval. First, the Gershgorin Disk Theorem is used to determine a finite interval in which all of the eigenvalues lie.

A fairly simple calculation can be performed to tell us how many eigenvalues lie to the left (or right) of a particular value σ. We use this calculation to repeatedly bisect the interval until there is just one eigenvalue per interval. After that, we continue to apply bisection to isolate each eigenvalue to machine precision. The LAPACK bisection routine is `sstebz`.

Algorithm 2.1.1 The Bisection Algorithm

1. *Determine a region which contains all of the eigenvalues: compute all Gershgorin disks to find an interval containing all n eigenvalues.*

2. *Compute all eigenvalues to desired accuracy using bisection method.*

For details on the bisection algorithm, see [18].

Inverse iteration is an application of the power method to $(A - \hat{\lambda} I)^{-1}$, where $\hat{\lambda}$ is one of the computed eigenvalues. The eigenvectors computed by inverse iteration may not be orthogonal if the eigenvalues are close together, and may need to be reorthogonalized by using modified Gram-Schmidt. This reorthogonalization can dominate the overall cost if many eigenvalues are close together. Despite these precautions, there are still cases in which orthogonality can be lost (see chapter 5).

Algorithm 2.1.2 The Inverse Iteration Algorithm

1. *Sort the eigenvalues and slightly perturb any which are too close for inverse iteration to function properly.*
2. *Compute the eigenvectors using inverse iteration.*
3. *Orthogonalize the eigenvectors corresponding to close eigenvalues.*

This algorithm is implemented in LAPACK routine `sstein`.

2.2. The QR Method

Every real square matrix A has a factorization of the form $A = QR$ where Q is an orthogonal matrix and R is upper triangular. The QR algorithm uses this factorization to produce a sequence of symmetric matrices T_i:

$$T_i - \sigma I = Q_i R_i \qquad T_{i+1} = R_i Q_i + \sigma I$$

Note that $T_{i+1} = Q_i^T T_i Q_i$.

If σ, which is called the shift, is chosen appropriately (as an approximate eigenvalue), the T_i will converge globally and usually cubically to the diagonal matrix of eigenvalues, and the product $Q_0 Q_1 \cdots Q_i$ will converge to the matrix of eigenvectors. QR is the standard algorithm for computing all eigenvalues and all eigenvectors. See [18] for details.

The LAPACK routine which implements the QR algorithm on symmetric tridiagonal matrices is `ssteqr`.

2.3. Root-Free QR

This algorithm is the Pal-Walker-Kahan variant of the QR algorithm [18]. It computes all of the eigenvalues of a symmetric tridiagonal matrix using a QR algorithm which does not require the use of square roots. Root-free QR is implemented in the LAPACK routine `ssterf` which is by far the fastest serial routine for the problem of finding all the eigenvalues of a symmetric tridiagonal matrix. We combine it with inverse iteration to find the full eigensystem. This has not been done traditionally, but proves to be fairly efficient here — if the computed eigenvalues are sufficiently accurate.

Chapter 3

Mathematical Formulation of Cuppen's Divide and Conquer Algorithm

The primary focus of this report is on how Cuppen's divide and conquer algorithm was implemented. In order to understand this, one should first understand how the mathematics underlying the algorithm work.

A tridiagonal matrix T can be decomposed into the sum of two matrices. One of the matrices is composed of two smaller, symmetric, tridiagonal matrices and the other is a very simple rank-one corrector for the off-diagonals which were removed:

$$T = \left[\begin{array}{ccc|ccc} \ddots & \ddots & & & & \\ \ddots & a_{m-1} & b_{m-1} & & & \\ & b_{m-1} & a_m & b_m & & \\ \hline & & b_m & a_{m+1} & b_{m+1} & \\ & & & b_{m+1} & a_{m+2} & \ddots \\ & & & & \ddots & \ddots \end{array}\right]$$

$$= \left[\begin{array}{cccc|ccc} \ddots & \ddots & & & & & \\ \ddots & a_{m-1} & b_{m-1} & & & & \\ & b_{m-1} & a_m - |b_m| & & & & \\ \hline & & & a_{m+1} - |b_m| & b_{m+1} & & \\ & & & & b_{m+1} & a_{m+2} & \ddots \\ & & & & & \ddots & \ddots \end{array}\right] + \left[\begin{array}{c|c} |b_m| & b_m \\ \hline b_m & |b_m| \end{array}\right]$$

$$\text{where } m = \lfloor \frac{n}{2} \rfloor \tag{3.0.1}$$

So, we have

$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + \rho \mathbf{v}\mathbf{v}^{\mathbf{T}}$, where $\rho = |b_m|$ and

$$\mathbf{v} = \begin{cases} \begin{bmatrix} \mathbf{e}_m \\ \mathbf{e}_1 \end{bmatrix} & \text{when } b_m > 0 \\ \begin{bmatrix} -\mathbf{e}_m \\ \mathbf{e}_1 \end{bmatrix} & \text{when } b_m < 0 \end{cases}$$

where $\mathbf{e}_m$ has length m and $\mathbf{e}_1$ has length $n - m$.

Suppose we know the eigendecompositions of T_1 and T_2: $T_i = Q_i \Lambda_i Q_i^T$. We relate the eigenvalues of T to those of T_1 and T_2 as follows.

$$\begin{aligned} T &= \begin{bmatrix} Q_1 \Lambda_1 Q_1^T & \\ & Q_2 \Lambda_2 Q_2^T \end{bmatrix} + \rho \mathbf{v}\mathbf{v}^T \\ &= \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left\{ \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} + \rho \mathbf{z}\mathbf{z}^T \right\} \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix} \end{aligned}$$

where

$$\mathbf{z} = \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix} \mathbf{v} = \begin{bmatrix} \pm\text{last column of } Q_1^T \\ \text{first column of } Q_2^T \end{bmatrix}$$

since $\mathbf{v} = [0, \ldots, 0, 1, 1, 0, \ldots, 0]^T$. Therefore, the eigenvalues of T are the same as those of

$$D + \rho \mathbf{z}\mathbf{z}^T \quad \textit{where } D = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} \tag{3.0.2}$$

D is diagonal and $\rho \mathbf{z}\mathbf{z}^T$ is rank-1. We apply a permutation P to $\{D + \rho \mathbf{z}\mathbf{z}^T\}$ so that $d_1, \ldots, d_n$ (the diagonal of D) is sorted: $d_{P(1)} \leq \cdots \leq d_{P(n)}$:

$$P^T P \left\{ D + \rho \mathbf{z}\mathbf{z}^T \right\} P^T P.$$

To find the eigenvalues of $D + \rho \mathbf{z}\mathbf{z}^T$, assume $D - \lambda I$ is nonsingular and compute the characteristic polynomial as follows:

$$\det(D + \rho \mathbf{z}\mathbf{z}^T - \lambda I) = \det((D - \lambda I)(I + \rho(D - \lambda)^{-1}\mathbf{z}\mathbf{z}^T))$$

Since $D - \lambda I$ is nonsingular, $\det(I + \rho(D - \lambda)^{-1}\mathbf{z}\mathbf{z}^T) = 0$ whenever λ is an eigenvalue. Note that $I + \rho(D - \lambda)^{-1}\mathbf{z}\mathbf{z}^T$ is the identity plus a rank-1 matrix; the determinant of such a matrix is easy to compute:

Lemma 3.0.1 *If* $\mathbf{x}$ *and* $\mathbf{y}$ *are vectors,* $\det(I + \mathbf{x}\mathbf{y}^T) = 1 + \mathbf{y}^T\mathbf{x}$.

Proof: Let $X = \left[\frac{\mathbf{x}}{\|\mathbf{x}\|_2}, X_2\right]$ be an orthogonal matrix with first column $\frac{1}{\|\mathbf{x}\|_2}\mathbf{x}$. Then

$$\det(I + \mathbf{x}\mathbf{y}^T) = \det(X^T(I + \mathbf{x}\mathbf{y}^T)X) = \det(I + X^T\mathbf{x}\mathbf{y}^T X)$$

We have $X^T\mathbf{x} = [\|\mathbf{x}\|, 0, \ldots, 0]^T$ and $\mathbf{y}^T X = [\frac{\mathbf{y}^T\mathbf{x}}{\|\mathbf{x}\|_2}, \mathbf{y}^T X_2]$, so

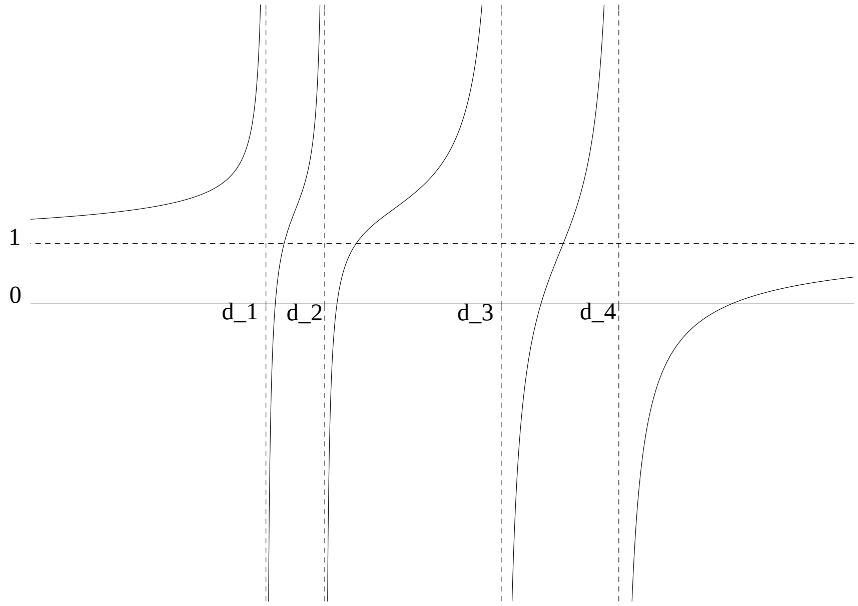
$$I + X^T\mathbf{x}\mathbf{y}^T X = \begin{bmatrix} 1 + \|\mathbf{x}\|_2 \cdot \frac{\mathbf{y}^T\mathbf{x}}{\|\mathbf{x}\|_2} & \mathbf{y}^T X \\ 0 & I \end{bmatrix}$$

which is upper triangular with determinate $1 + \mathbf{y}^T\mathbf{x}$. $\square$

Therefore

$$\det(I + \rho(D - \lambda)^{-1}\mathbf{z}\mathbf{z}^T) = 1 + \rho\mathbf{z}^T(D - \lambda)^{-1}\mathbf{z} = 1 + \rho\sum_{i=1}^{n}\frac{z_i^2}{d_i - \lambda} \equiv f(\lambda) \quad (3.0.3)$$

and the eigenvalues of T are the roots of the **secular equation** $f(\lambda) = 0$. If all d_i are distinct and all $z_i \neq 0$, the function $f(\lambda)$ has the graph shown here (for $n = 4$ and $\rho > 0$).



As we can see, the line $y = 1$ is a horizontal asymptote, the lines $x = d_i$ are vertical asymptotes, and the roots of $f(\lambda)$ are interlaced by the d_i. Since $f(\lambda)$ is monotonic on the intervals (d_i, d_{i+1}) it is possible to find a version of Newton's method that converges fast and monotonically to each root with a starting point in (d_i, d_{i+1}). We use the zero-finder in [14] which is faster and more reliable than the original one in [7] or in [3]; see [14] for details.

It is also easy to derive an expression for the eigenvectors of T:

Lemma 3.0.2 *If λ is an eigenvalue of $D + \rho\mathbf{z}\mathbf{z}^T$ then $(D - \lambda I)^{-1}\mathbf{z}$ is its eigenvector.*

Proof:

$$\begin{aligned}(D + \rho\mathbf{z}\mathbf{z}^T)[(D - \lambda I)^{-1}\mathbf{z}] &= (D - \lambda I + \lambda I + \rho\mathbf{z}\mathbf{z}^T)(D - \lambda I)^{-1}\mathbf{z} \\ &= \mathbf{z} + \lambda(D - \lambda I)^{-1}\mathbf{z} + \mathbf{z}[\rho\mathbf{z}^T(D - \lambda I)^{-1}\mathbf{z}] \\ &= \mathbf{z} + \lambda(D - \lambda I)^{-1}\mathbf{z} - \mathbf{z}, \text{ since } \lambda \text{ is a root of (3.0.3)} \\ &= \lambda[(D - \lambda I)^{-1}\mathbf{z}]\end{aligned}$$

as needed. □

Unfortunately, this simple formula for the eigenvectors is not numerically stable; this is the primary reason the algorithm took over a decade from its formulation until it became sufficiently reliable to use. The trick which makes the algorithm work is an added step once the roots of the secular function are found. At this point, we have $d_1, \ldots, d_n$ and the newly computed $\lambda_1, \ldots, \lambda_n$. These values are used to recompute $\hat{z}_1, \ldots, \hat{z}_n$ and this $\hat{\mathbf{z}}$ is used to compute the eigenvectors. See [10] for more details.

The overall efficiency of the algorithm is also significantly aided by **deflation**, which means that some roots of the secular equation can be found very cheaply. For example, if d_i and d_{i+1} are nearly the same, we know the eigenvalue which lies in (d_i, d_{i+1}) must also nearly be the same. A similar phenomenon occurs if some z_i is very small. These observations can save half the work for some matrices. There will be further details on deflation in the next chapter.

See [4, 7, 19, 10] for more details.

The overall divide and conquer algorithm can best be expressed recursively:

Algorithm 3.0.1 Divide and Conquer Algorithm *Finding eigenvalues and eigenvectors of a symmetric tridiagonal matrix using divide-and-conquer*

proc dc_eig (T, Q, Λ) ..... from input T compute outputs Q and Λ where $T = Q\Lambda Q^T$
  if T is 1 by 1
    return $Q = 1, \Lambda = T$
  else
    form $T = \left[\begin{array}{c|c} T_1 & \\ \hline & T_2 \end{array}\right] + b_m \mathbf{v}\mathbf{v}^T$, as in 3.0.1
    call dc_eig (T_1, Q_1, Λ_1)
    call dc_eig (T_2, Q_2, Λ_2)
    form $D + \rho \mathbf{z}\mathbf{z}^T$ from $\Lambda_1, \Lambda_2, Q_1, Q_2$, as in 3.0.2
    find eigenvalues Λ and eigenvectors $\tilde{Q}$ of $D + \rho \mathbf{z}\mathbf{z}^T$
      by deflating and solving the secular equation
    form (eigenvectors of T) $= Q = \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] \cdot \tilde{Q}$
    return Q and Λ
  endif

This summary excludes many of the nuances involved in sorting Λ_1 and Λ_2 to get D, deflation, and other considerations which will be dealt with in the next chapter.

Chapter 4

The Divide and Conquer Code

If the mathematical description seemed straightforward, the code is rather complex and requires explanation. The code is composed of 13 different routines with a total of 3863 lines of Fortran, 2009 of which are comments. The code covers two cases not discussed above — finding only the eigenvalues of a symmetric tridiagonal matrix, and finding the eigenvalues and eigenvectors of a dense symmetric matrix which has been reduced to tridiagonal form ($A = UTU^T$) by directly updating U by multiplying the intermediate eigenvector matrices into it as they are generated. Both of these options use exactly the same mathematics as the eigensystem from tridiagonal case, but use additional storage to avoid having to accumulate the eigenvectors of the tridiagonal. The calling sequence for the main routine `sstedc` is given in appendix A. The complete software will be available as part of the LAPACK library [1].

This chapter is organized in the following fashion: Section 4.1 details the structure of the code with respect to the calculation of the eigensystem of a symmetric tridiagonal matrix. Section 4.2 explains the code used to find only the eigenvalues of a symmetric tridiagonal matrix. Section 4.3 tells how the code which finds only the eigenvalues is modified to find the full eigensystem of a reduced symmetric matrix.

4.1. Eigenvalues and Eigenvectors of a Tridiagonal

This is code option `COMPQ` = "I" in `sstedc`. This option requires routines `sstedc`, `slaed0`, `slaed1`, `slaed2`, `slaed3`, `slaed4`, `slaed5`, `slaed6`, and `slamrg`.

1. `sstedc` checks the tridiagonal for off-diagonals which are small enough to be treated as if they were zero (i.e. tries to split the problem). It then scales each submatrix and calls `slaed0` to solve each in turn, scaling the eigenvalues back when finished.

2. `slaed0` is the real driver of the divide and conquer algorithm. Since Fortran77 doesn't support recursion, this routine divides the input matrix into submatrices by repeatedly halving the size of all submatrices until the largest submatrix is at most `SMLSIZ` (a parameter which is currently set to 25). The rank-one perturbations which make the subproblems independent are then performed. The LAPACK QR code `ssteqr` is used to compute the eigensystems of these independent submatrices. Each rank-one perturbation is repaired by a call to `slaed1`. See figure 4.1.1 for a diagram indicating the overall structure of the algorithm.

3. `slaed1` acts primarily as a routing station. There are several steps required to repair one of these rank-one perturbations. First, the two lists of diagonal elements from the previous level need to be combined into a single sorted list. This enables us to efficiently look for elements which are close together (which in turn allows us to deflate the problem). We also test whether the `z`-vector entry corresponding to an element in the sorted diagonal (D) is exceptionally small (this also allows us to deflate the given element from the secular equation problem). Deflated eigenvalues are moved to the end of the list of diagonal elements so that they can be easily ignored by the zero finder. These parts are performed by `slaed2`.

   After the problem is deflated as much as is possible, the zeros of the secular function must be found. These roots are the eigenvalues of the current (sub)problem and are used to compute the associated vectors. The resulting matrix of eigenvectors is multiplied into the larger matrix which holds the collective results of all previous eigenvector calculations. These latter parts are performed by `slaed3`.

   Finally, `slaed1` determines a merge permutation which will reintegrate the deflated values into the full list so that the list will be easily brought to sorted order. This permutation is stored list-wise (rather than as an $n \times n$ matrix) in

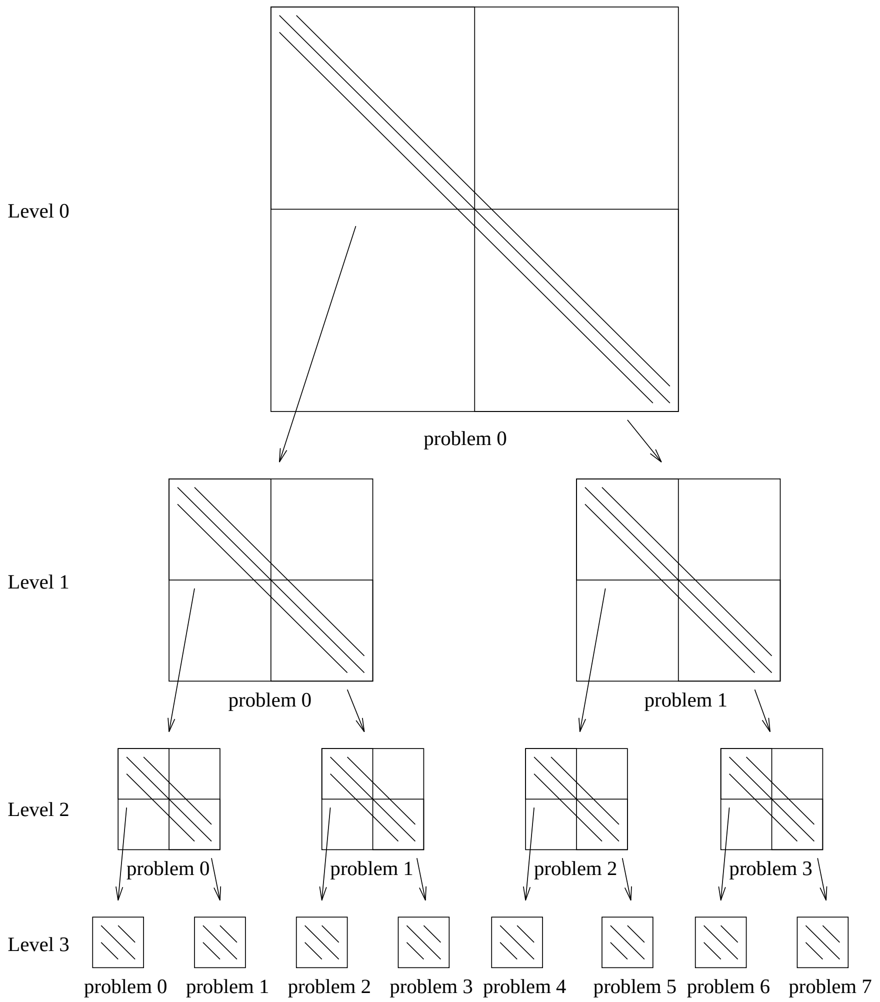


Figure 4.1.1: A divide and conquer tree

This figure shows the organization of the various levels of divide and conquer tree. The horizontal and vertical cuts through a matrix problem indicate the rank-one perturbation which is repaired by solving the problem. On the lowest level, QR is used to find the eigendecomposition of the smallest subproblems. The algorithm then proceeds to work back up the levels, solving (in order) all of the problems on the level before proceeding to the next level.

INDXQ such that

$$\mathtt{D}(\mathtt{INDXQ}(i)) \leq \mathtt{D}(\mathtt{INDXQ}(j)) \text{ whenever } i \leq j.$$

4. slaed2 is the sorting and deflation routine. All sorted lists are in ascending order unless noted otherwise. First, elements which were deflated on the previous level are merged back into each of the two lists using the permutation stored in INDXQ. Mathematically, we insert a permutation $P_{\mathtt{IXQ}}$ to rearrange the diagonal elements and the $\mathbf{z}$-vector:

$$\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_{\mathtt{IXQ}}^T P_{\mathtt{IXQ}} \left(\left[\begin{array}{c|c} \Lambda_1 & \\ \hline & \Lambda_2 \end{array}\right] + \rho \mathbf{z}\mathbf{z}^T \right) P_{\mathtt{IXQ}}^T P_{\mathtt{IXQ}} \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T$$

Thus there are two independent sorted lists. slamrg is called to generate a permutation which will merge these two lists. This permutation is stored in INDX.

$$\begin{aligned}
&\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_{\mathtt{IXQ}}^T P_{\mathtt{IX}}^T P_{\mathtt{IX}} P_{\mathtt{IXQ}} \left(\left[\begin{array}{c|c} \Lambda_1 & \\ \hline & \Lambda_2 \end{array}\right] + \rho \mathbf{z}\mathbf{z}^T \right) P_{\mathtt{IXQ}}^T P_{\mathtt{IX}}^T P_{\mathtt{IX}} P_{\mathtt{IXQ}} \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T \\
= \quad & \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] \underbrace{P_{\mathtt{IXQ}}^T P_{\mathtt{IX}}^T}_{P_1^T} \left(D_{sort} + \rho \mathbf{z}_{sort} \mathbf{z}_{sort}^T \right) P_{\mathtt{IX}} P_{\mathtt{IXQ}} \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T
\end{aligned}$$

The permutations thus generated are not applied to the collection of eigenvectors at this time, but are maintained and will be applied later along with two other permutations. Z and COLTYP are sorted according to the same permutations as D. COLTYP is an array of length n which keeps track of the structure of the eigenvectors:

$$\mathtt{COLTYP}(i) = \begin{cases} 1 \text{ if the vector associated with } \mathtt{D}(i) \\ \quad\quad \text{is nonzero only in the upper half} \\ 2 \text{ if the vector associated with } \mathtt{D}(i) \\ \quad\quad \text{is nonzero only in the lower half} \end{cases}$$

COLTYP tells whether the vector came from Q_1 or Q_2.

Each $\mathtt{D}(j)$ is now checked to see if it can be deflated from the problem. First, $\mathtt{Z}(j)$ is checked to determine if it is exceptionally small. If not, $\mathtt{D}(j)$ is checked to determine if it is exceptionally close to the preceding undeflated value. In the former case, $\mathtt{D}(j)$ is deflated without further calculation. In the latter case, if $\mathtt{D}(i)$ and $\mathtt{D}(j)$ are the two values involved (with $\mathtt{D}(i) < \mathtt{D}(j)$), a Givens rotation is performed with the objective of changing $\mathtt{Z}(i)$ to zero.

$$\left(\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T\right) \left\{G\left(D_{sort} + \rho \mathbf{z}_{sort}\mathbf{z}_{sort}^T\right) G^T\right\} \left(G P_1 \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T\right)$$

Here, $P_1 = P_{\mathtt{IX}} P_{\mathtt{IXQ}}$ and G is the product of all of the Givens rotations performed during deflation. These rotations are also applied to $\mathtt{D}(i)$ and $\mathtt{D}(j)$, as well as the associated vectors, now found in $\mathtt{Q}(1 : n, \mathtt{INDXQ}(\mathtt{INDX}(i)))$ and $\mathtt{Q}(1 : n, \mathtt{INDXQ}(\mathtt{INDX}(j)))$. If $\mathtt{COLTYP}(i) \neq \mathtt{COLTYP}(j)$ then neither of these vectors can be guaranteed to have any sparsity and so we assign $\mathtt{COLTYP}(i) = 4$ to indicate that the related value has been deflated and $\mathtt{COLTYP}(j) = 3$ to indicate that the related vector is dense. Then $\mathtt{D}(i)$ is deflated, just as in the case when $\mathtt{Z}(i)$ was small in the beginning. Any values which are deflated are moved to the end of the list. Note that this means that the deflated values will be arranged in *descending* order. The permutation thus generated is stored in `INDXP` for later use; this permutation is represented below by $P_{\mathtt{IXP}}$. The values in `D` are permuted into `DLAMDA` and likewise `Z` into `W`.

$$\begin{aligned}
&\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T P_{\mathtt{IXP}}^T P_{\mathtt{IXP}} G \{D_{sort} + \rho \mathbf{z}_{sort}\mathbf{z}_{sort}^T\} G^T P_{\mathtt{IXP}}^T P_{\mathtt{IXP}} G P_1 \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T \\
&= \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T P_{\mathtt{IXP}}^T \left[\begin{array}{cc} \breve{D} + \rho \breve{\mathbf{z}}\breve{\mathbf{z}}^T & \\ & \breve{D}_4 \end{array}\right] P_{\mathtt{IXP}} G P_1 \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T
\end{aligned}$$

Once all possible deflation has taken place, a final permutation $P_{\mathtt{IXC}}$ is generated for the eigenvectors. Recall that the final step in the repair process includes a matrix-matrix multiply:

After computing $\breve{D} + \rho \breve{\mathbf{z}}\breve{\mathbf{z}}^T = \tilde{Q}\tilde{D}\tilde{Q}^T$ (computed by `slaed3`) we have

$$\underbrace{\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T P_{\mathtt{IXP}}^T \left[\begin{array}{c|c} \tilde{Q} & \\ \hline & I \end{array}\right]}_{to\ be\ evaluated} \tilde{D} \left[\begin{array}{c|c} \tilde{Q} & \\ \hline & I \end{array}\right]^T P_{\mathtt{IXP}} G P_1 \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right]^T$$

$P_{\mathtt{IXC}}$ is formulated to group the vectors according to their `COLTYP`—in this way we can take maximal advantage of the structure of the matrix:

$$
\begin{aligned}
&\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T P_{\mathtt{IXP}}^T P_{\mathtt{IXC}} P_{\mathtt{IXC}}^T \left[\begin{array}{c|c} \tilde{Q} & \\ \hline & I \end{array}\right] \\
= \quad &\left(\left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] P_1^T G^T P_{\mathtt{IXP}}^T P_{\mathtt{IXC}} \right) \left(P_{\mathtt{IXC}}^T \left[\begin{array}{c|c} \tilde{Q} & \\ \hline & I \end{array}\right] \right) \\
= \quad &\left[\begin{array}{c|c|c|c} \breve{Q}_1 & & & \\ \cline{1-2} & \breve{Q}_2 & \breve{Q}_3 & \breve{Q}_4 \end{array}\right] \left(P_{\mathtt{IXC}}^T \left[\begin{array}{c|c} \tilde{Q} & \\ \hline & I \end{array}\right] \right)
\end{aligned}
$$

The idea of distinguishing the columns of Q according to their sparsity structure appears in [10]. Since the order of the deflated values is irrelevant to further computation on this problem, $P_{\mathtt{IXC}}$ looks like:

$$
P_{\mathtt{IXC}} = \left[\begin{array}{c|c} \grave{P}_{\mathtt{IXC}} & \\ \hline & I \end{array}\right]
$$

The dimensions of I are the same as the number of deflated values (i.e. the number of columns in $\breve{Q}_4$). The matrix is thereby organized in such a way that it is composed of four parts, some of which may be empty. This organization allows us to use the BLAS to perform matrix-matrix multiplies of minimal sizes. The permutation list for $P_{\mathtt{IXC}}$ is stored in `IXC`. The vectors in `Q` are copied into `Q2`, using all four permutations at once.

$$
\mathtt{Q2}(1:n, i) = \mathtt{Q}(1:n, \mathtt{INDXC}(\mathtt{INDXP}(\mathtt{INDX}(\mathtt{INDXQ}(i)))))
$$

5. `slaed3` performs a sequence of calls to `slaed4`. Each such call solves for one root of the secular equation (stored directly into `D`) as well as its associated eigenvector (stored temporarily in `Q`). Once all of the roots have been found, the eigenvectors are modified according to the methods designed by Gu and Eisenstat [10] and the rows of this eigenvector matrix are permuted according to premultiplication by $P_{\mathtt{IXC}}^T$. In the code, we move `Q` into workspace `S`:

$$
\mathtt{S}(i, 1:k) = \mathtt{Q}(\mathtt{INDXC}(i), 1:k)
$$

At this point, the matrix-matrix multiply is performed:

$$\left[\begin{array}{c|c|c|c} \breve{Q}_1 & & & \\ \hline & \breve{Q}_2 & \breve{Q}_3 & \breve{Q}_4 \end{array}\right]\left[\begin{array}{c|c} \grave{P}_{\mathtt{IXC}}\tilde{Q} & \\ \hline & I \end{array}\right] = \left[\left[\begin{array}{c|c|c} \breve{Q}_1 & & \\ \hline & \breve{Q}_2 & \breve{Q}_3 \end{array}\right]\left[\grave{P}_{\mathtt{IXC}}\tilde{Q}\right] \middle| \breve{Q}_4\right]$$

We write

$$\grave{P}_{\mathtt{IXC}}\tilde{Q} = \left[\begin{array}{c} \tilde{Q}_1 \\ \hline \tilde{Q}_2 \\ \hline \tilde{Q}_3 \end{array}\right]$$

($\tilde{Q}$ is $k \times k$, $\tilde{Q}_1$ is $k_1 \times k$, $\tilde{Q}_2$ is $k_2 \times k$, $\tilde{Q}_3$ is $k_3 \times k$, where k_i is the number of columns in $\breve{Q}_i$, $k = k_1 + k_2 + k_3, k + k_4 = n$), so the matrix-matrix multiply is broken into three parts:

$$\left[\begin{array}{c|c|c} \breve{Q}_1 & & \\ \hline & \breve{Q}_2 & \breve{Q}_3 \end{array}\right]\left[\begin{array}{c} \tilde{Q}_1 \\ \hline \tilde{Q}_2 \\ \hline \tilde{Q}_3 \end{array}\right] = \left[\begin{array}{c} \breve{Q}_1\tilde{Q}_1 \\ \hline \\ \end{array}\right] + \left[\begin{array}{c} \\ \hline \breve{Q}_2\tilde{Q}_2 \end{array}\right] + \left[\ \breve{Q}_3\tilde{Q}_3\ \right]$$

Standard BLAS are used for each of these matrix-matrix multiplications which are formulated as follows:

$$\begin{array}{lll}
\mathtt{Q}(1:\lfloor\frac{n}{2}\rfloor, 1:k) & \leftarrow & \mathtt{Q2}(1:\lfloor\frac{n}{2}\rfloor, 1:k_1)\cdot\mathtt{S}(1:k_1, 1:k) \\
\mathtt{Q}(\lfloor\frac{n}{2}\rfloor+1:n, 1:k) & \leftarrow & \mathtt{Q2}(\lfloor\frac{n}{2}\rfloor+1:n, k_1+1:k_1+k_2) \\
 & & \cdot\mathtt{S}(k_1+1:k_1+k_2, 1:k) \\
\mathtt{Q}(1:n, 1:k) & \leftarrow & \mathtt{Q}(1:n, 1:k) + \mathtt{Q2}(1:n, k_1+k_2+1:k) \\
 & & \cdot\mathtt{S}(k_1+k_2+1:k, 1:k) \\
\mathtt{Q}(1:n, k+1:n) & \leftarrow & \mathtt{Q2}(1:n, k+1:n)
\end{array}$$

6. `slaed4` is a routine described in [14]. It calculates a single root of the secular equation and returns the information necessary to compute the eigenvectors. `slaed4` also calls `slaed5` and `slaed6`. `slaed5` handles the solution of the secular equation when $n = 2$, and `slaed6` computes one of the two roots least in magnitude of

$$f(x) = \rho + \frac{z_1}{d_1 - x} + \frac{z_2}{d_2 - x} + \frac{z_3}{d_3 - x}$$

7. `slamrg` takes a vector containing two concatenated lists of floating point numbers which are independently sorted and generates a permutation list which will merge them into a single sorted list.

4.2. Eigenvalues Only of a Tridiagonal

This is code option `COMPQ` = "N" in `sstedc`. This problem requires routines `sstedc`, `slaed0`, `slaed4`, `slaed5`, `slaed6`, `slaed7`, `slaed8`, `slaed9`, `slaeda`, and `slamrg`. Instead of performing many matrix-matrix multiplies to keep the eigenvector matrix of the tridiagonal up to date, this option in the code simply stores all of the intermediate eigenvector matrices. Whenever a **z**-vector is needed, `slaeda` performs a series of matrix-vector multiplies to obtain it.

1. `sstedc` performs exactly the same function as in the previous section.

2. `slaed0` performs exactly the same function as in the previous section. It also sets up a number of workspaces which are required for this option and routes the merge operation to `slaed7`. See the description of `slaeda` below for details on the data structures.

3. `slaed4`, `slaed5`, and `slaed6` perform exactly the same functions as in the previous section.

4. `slaed7` performs the same function as `slaed1` above with some important exceptions. The computation of the **z**-vector is no longer trivial. Since rows from the matrix of eigenvectors are not available, `slaeda` must be called in order to calculate the relevant ones. The same steps are required to repair a rank-one perturbation:

   (a) merge the two lists of diagonal elements

   (b) deflate where possible

   (c) find the roots of the remaining secular equation

   (d) compute the associated eigenvectors, but do *not* multiply them against the previously computed eigenvectors.

   The first two steps are accomplished by `slaed8`, the latter two by `slaed9`.

   Finally, just as in `slaed1`, `slaed7` determines a merge permutation which will reintegrate the deflated values into the full list so that the list can be easily brought to sorted order and stores it in `INDXQ`.

5. `slaed8` is the sorting and deflation routine. First, elements which were deflated on the previous level are merged into each of the two lists by use of `INDXQ`. This results in two independent sorted lists. These are then merged and the permutation used is stored in `INDX`. The permutations which accomplish this sorting are not applied to the collection of eigenvectors but are recorded and will be combined with other permutations and stored for use in computing subsequent $\mathbf{z}$-vectors.

   Deflation is done in exactly the same manner as `slaed2` with the exception that when deflation occurs because two diagonal elements are close to one another, the Givens rotation is not applied to the eigenvectors, but rather the columns involved are stored in `GIVCOL` and the floating point values of the rotation are stored in `GIVNUM` for use in computing subsequent $\mathbf{z}$-vectors. The permutation used for deflation is recorded in `INDXP`.

   Since no matrix-matrix multiplication will occur, no reorganization will take place.

   The combined permutations are recorded in $\texttt{PERM}(1:n)$,

$$\texttt{PERM}(i) = \texttt{INDXQ}(\texttt{INDX}(\texttt{INDXP}(i))).$$

6. `slaed9` performs a sequence of calls to `slaed4`. Each such call solves for one root of the secular equation as well as its associated eigenvector. Once all of the roots have been found, the eigenvectors are modified according to the methods in [10]. This matrix of eigenvectors is stored for use in computing subsequent $\mathbf{z}$-vectors.

7. `slaeda` calculates the $\mathbf{z}$-vector needed for a rank-one repair problem. Understanding this section of code requires an understanding of the data structure used to store the permutations, Givens rotations, and intermediate eigenvector matrices.

   The values which are required to decipher the data structure are `TLVLS`, `CURLVL`, and `CURPBM`. `TLVLS` is the highest numbered level in the overall divide and conquer tree (the root level is numbered zero), i.e. the number of splits required to make all of the submatrices of the problem submitted to `slaed0` no larger than `SMLSIZ`. `TLVLS` is constant over each call to `slaed0`. `CURLVL` is the level of the tree on which the current rank-one repair problem resides. The root of the tree is level 0. There are $2^{\texttt{CURLVL}}$ rank-one repair problems on level CURLVL. CURPBM is the number of the "current" rank-one repair problem on level CURLVL — the one for which $\mathbf{z}$ is to be computed; these are numbered $0, \ldots, 2^{\texttt{CURLVL}} - 1$.

The problems are numbered bottom to top, left to right in the divide and conquer tree, so that the problems on the lowest level of the tree are numbered $0, \ldots, 2^{\texttt{TLVLS}} - 1$. Call this number *prbnum* (which is distinct from CURPBM):

$$prbnum(\texttt{TLVLS}, \texttt{CURLVL}, \texttt{CURPBM}) = 1 + \texttt{CURPBM} \sum_{i=\texttt{CURLVL}+1}^{\texttt{TLVLS}} 2^i$$

Each problem has a permutation associated with it. The permutations are stored end-to-end in the array PERM. If $t = \texttt{TLVLS}$, the permutation which merges the two lists from level $\texttt{CURLVL} = cl$, in problems $2i-1$ and $2i$ ($1 \leq 2i-1, 2i \leq 2^{cl}$) is first used in problem i on level $cl-1$, so it is stored in locations $\texttt{PRMPTR}(prbnum(t, cl-1, i))$ through $\texttt{PRMPTR}(prbnum(t, cl-1, i+1)) - 1$ of PERM. PRMPTR, like the rest of the pointer arrays, indicates the starting point of the data of interest. The length of the permutation for a problem is the same as the size of the problem before deflation:

$$\begin{aligned} &\text{permdim}(prbnum(t, cl-1, i)) \\ &\quad = \quad \texttt{PRMPTR}(prbnum(t, cl-1, i)) - \texttt{PERMPTR}(prbnum(t, cl-1, i+1)) \end{aligned}$$

Similarly, the data necessary to perform the Givens rotations are stored in GIVCOL(2,*) (which stores the column numbers involved in the rotation) and in GIVNUM(2,*) (which stores the matrix elements involved in formulating the rotation). The rotations for the aforementioned problem would be found in locations $\texttt{GIVPTR}(prbnum(t, cl-1, i))$ through $\texttt{GIVPTR}(prbnum(t, cl-1, i+1)) - 1$.

The storage of the intermediate eigenvector matrices is exactly similar. Q holds the matrices which are stored as one long vector by stacking the columns. The matrix for the previous problem would be found in locations $\texttt{QPTR}(prbnum(t, cl-1, i))$ through $\texttt{QPTR}(prbnum(t, cl-1, i+1)) - 1$. The dimensions of the square Q matrix thus found are

$$\begin{aligned} &\text{qdim}(prbnum(t, cl-1, i)) \\ &\quad = \sqrt{\texttt{QPTR}(prbnum(t, cl-1, i)) - \texttt{QPTR}(prbnum(t, cl-1, i+1))} \end{aligned}$$

The computation is fairly straightforward: (a pre-subscript here will designate the level from which a matrix originates)

$$\mathbf{z} = \begin{bmatrix} {}_{cl+1}\tilde{Q}_1 & & \\ & \ddots & \\ & & {}_{cl+1}\tilde{Q}_{2^{cl}} \end{bmatrix}^T \begin{bmatrix} {}_{cl+1}\tilde{G}_1 & & \\ & \ddots & \\ & & {}_{cl+1}\tilde{G}_{2^{cl}} \end{bmatrix}^T \begin{bmatrix} {}_{cl+1}\tilde{P}_1 & & \\ & \ddots & \\ & & {}_{cl+1}\tilde{P}_{2^{cl}} \end{bmatrix}^T$$

$$\cdots \begin{bmatrix} {}_{t-1}\tilde{Q}_1 & & \\ & \ddots & \\ & & {}_{t-1}\tilde{Q}_{2^{cl}} \end{bmatrix}^T \begin{bmatrix} {}_{t-1}\tilde{G}_1 & & \\ & \ddots & \\ & & {}_{t-1}\tilde{G}_{2^{cl}} \end{bmatrix}^T \begin{bmatrix} {}_{t-1}\tilde{P}_1 & & \\ & \ddots & \\ & & {}_{t-1}\tilde{P}_{2^{cl}} \end{bmatrix}^T$$

$$\cdot \begin{bmatrix} {}_{t}\tilde{Q}_1 & & \\ & \ddots & \\ & & {}_{t}\tilde{Q}_{2^{t}} \end{bmatrix}^T \begin{bmatrix} \mathbf{e}_m \\ \mathbf{e}_1 \end{bmatrix}$$

Algorithm 4.2.1 `slaeda` — *computing* $\mathbf{z}$

$t =$ `TLVLS`,

$midprb = 2^{t-1}$,

$midz = \lfloor \frac{n}{2} \rfloor$,

$zsize1 = \text{qdim}(midprb)$,

$zsize2 = \text{qdim}(midprb + 1)$

$\mathbf{z} = \mathbf{0}$

$$\mathbf{z}_{[midz - zsize1 : midz + zsize2]} = \begin{bmatrix} \mathbf{0} \\ \text{last col of } {}_t\tilde{Q}^T_{midprb} \\ \text{last col of } {}_t\tilde{Q}^T_{midprb+1} \\ \mathbf{0} \end{bmatrix}$$

For i = `TLVLS` - *1 downto* `CURLVL` + *1*

> $midprb = 2^{i-1}$,
>
> $zstart = midz - \text{permdim}(midprb) + 1$,
>
> $zfin = midz - \text{permdim}(midprb + 1)$

$$\mathbf{z}_{[zstart:zfin]} = \begin{bmatrix} {}_iP_{midprb} & \\ & {}_iP_{midprb+1} \end{bmatrix}^T \mathbf{z}_{[zstart:zfin]}$$

$$\mathbf{z}_{[zstart:zfin]} = \begin{bmatrix} {}_iG_{midprb} & \\ & {}_iG_{midprb+1} \end{bmatrix}^T \mathbf{z}_{[zstart:zfin]}$$

$$\mathbf{z}_{[zstart:zfin]} = \begin{bmatrix} {}_i\tilde{Q}_{midprb} & \\ & {}_i\tilde{Q}_{midprb+1} \end{bmatrix}^T \mathbf{z}_{[zstart:zfin]}$$

This simple routine computes the same $\mathbf{z}$-vectors as were computed with code option `COMPQ` = "I", but with matrix-vector multiplies instead of matrix-matrix multiplies.

8. `slamrg` performs exactly the same function as in the previous section.

4.3. Eigenvalues and Eigenvectors of a Reduced Matrix

This is code option `COMPQ` = "V". This problem requires routines `sstedc`, `slaed0`, `slaed4`, `slaed5`, `slaed6`, `slaed7`, `slaed8`, `slaed9`, `slaeda`, and `slamrg`. This case is handled almost identically to the previous one. In this case, however, when operations occur which would alter the intermediate eigenvector matrix, they are instead applied to U, the orthogonal matrix supplied to the routine (this is presumably the matrix used to reduce the dense or banded matrix to tridiagonal form).

1. `sstedc` performs exactly the same function as in the previous section.

2. `slaed0` performs exactly the same function as in the previous section. An additional $n \times n$ floating point workspace is required in order to perform matrix-matrix multiplies into U.

3. `slaed4`, `slaed5`, and `slaed6` perform exactly the same functions as in the previous section.

4. `slaed7` performs exactly the same function as in the previous section. It also performs a matrix-matrix multiplication: the newly computed intermediate eigenvectors into U.

5. `slaed8` is the sorting and deflation routine. Sorting happens as always. The sorting permutations are stored as before. The required Givens rotations are stored as before and also applied directly to U. Once the deflation permutation is calculated, the cumulative effect of the three permutations is applied to the relevant columns of U.

6. `slaed9`, `slaeda`, and `slamrg` perform exactly the same functions as in the previous sections.

Chapter 5

Numerical Results

This section compares four methods for solving the symmetric tridiagonal eigenproblem in terms of run-time and accuracy. The methods are divide and conquer (referred to as EDC in the labels in the various figures), QR (EQR), bisection/inverse iteration (EBZ), and root-free QR/inverse iteration (ERI). Reducing a dense matrix to tridiagonal form and forming the corresponding orthogonal matrix is referred to by TRD and the final matrix-matrix multiply for bisection/inverse iteration and divide and conquer is referred to by MM. The `COMPQ` = "I" and `COMPQ` = "V" options were tested. The results in sections 5.4 and 5.5 are just for `COMPQ` = "I". Some performance comparisons between `COMPQ` = "I" and `COMPQ` = "V" are given in section 5.6. Only the `COMPQ` = "I" option (section 4.1) was tested. The test cases and method of error measurement will be defined in this section.

The tests were run on a DEC Alpha (DEC 3000/500X), using both Fortran BLAS (Basic Linear Algebra Subroutines), and highly optimized BLAS. They were also run on a DEC 5000 and Sparc 2 using Fortran BLAS; the relative performance of the algorithms was essentially the same as on the DEC Alpha with Fortran BLAS.

This chapter is organized as follows. Section 5.1 describes how we know when an algorithm is performing well and how we compare the performances of different algorithms. Section 5.2 details the matrices which were used in the testing process. Section 5.3 gives the overall results of the testing procedure. Section 5.4 gives the results for dense matrices. Section 5.5 gives the results for tridiagonal matrices. Section 5.6 gives a theoretical analysis of the performance difference between the options `COMPQ` = "I" and `COMPQ` = "V" with some experimental data.

5.1. Measures of the Quality of a Method

The speed with which a method computes a solution is measured straightforwardly. The accuracy of a method is determined by the residual error in the computed solution and the orthogonality of the computed eigenvectors. For $T = Q\Lambda Q^T$ with computed solution $\hat{Q}\hat{\Lambda}\hat{Q}^T$, these errors are computed by:

$$\text{residual error} = \mathcal{R} = \max_i \frac{\|(T\hat{Q} - \hat{Q}\hat{\Lambda})\mathbf{e}_i\|_2}{|\hat{\lambda}|_{max}}, \text{ and}$$

$$\text{orthogonality error} = \mathcal{O} = \max_i \|(\hat{Q}\hat{Q}^T - I)\mathbf{e}_i\|_2.$$

The errors thus determined are governed by the the largest error for any single computed eigenpair.

The theorem below shows that if the residual and orthogonality errors are small, then the computed eigendecomposition has small absolute error. The proof of this theorem depends on some lemmas about the properties of matrix norms.

Lemma 5.1.1 $\|A\|_2 \leq \sqrt{n}\max_i \|A\mathbf{e}_i\|_2$

Lemma 5.1.2 $\|AA^T\|_2 = \|A\|_2^2$.

Lemma 5.1.3 *If* $\|X\| < 1$*, then*

$$\|(I - X)^{-1}\| \leq \frac{1}{1 - \|X\|}$$

Theorem 5.1.1 *Let* $\hat{Q}\hat{\Lambda}\hat{Q}^T$ *be the computed eigensystem of a symmetric tridiagonal matrix* T*. If* $\mathcal{R} \leq \epsilon_1$ *and* $\mathcal{O} \leq \epsilon_2$*, then there exists a matrix* E *such that*

$$T + E = \hat{Q}\hat{\Lambda}\hat{Q}^T, \text{ and } \|E\|_2 \leq \sqrt{n}\left(|\lambda|_{max}\epsilon_2 + |\hat{\lambda}|_{max}\epsilon_1\sqrt{1 + \sqrt{n}\epsilon_2}\right).$$

Proof: Let

$$E_1 = \delta(T\hat{Q} - \hat{Q}\hat{\Lambda}), \text{ where } \delta = \frac{1}{|\hat{\lambda}|_{max}} \tag{5.1.1}$$

$$E_2 = \hat{Q}\hat{Q}^T - I$$

$$\begin{aligned} \textit{Lemma } 5.1.1 \Rightarrow \quad & \|E_1\|_2 \leq \sqrt{n} \max_i \|E_1 \mathbf{e}_i\|_2 \\ & \mathcal{R} \equiv \delta \max_i \|(T\hat{Q} - \hat{Q}\hat{\Lambda})\mathbf{e}_i\|_2 \leq \epsilon_1 \\ & \Rightarrow \|E_1\|_2 \leq \sqrt{n}\mathcal{R} \leq \sqrt{n}\epsilon_1 \end{aligned}$$

Similarly,

$$\begin{aligned} \textit{Lemma } 5.1.1 \Rightarrow \quad & \|E_2\|_2 \leq \sqrt{n} \max_i \|E_2 \mathbf{e}_i\|_2 \\ & \mathcal{O} \equiv \max_i \|(\hat{Q}\hat{Q}^T - I)\mathbf{e}_i\|_2 \leq \epsilon_2 \\ & \Rightarrow \|E_2\|_2 \leq \sqrt{n}\mathcal{O} \leq \sqrt{n}\epsilon_2 \end{aligned}$$

$$\begin{aligned} \textit{Lemma } 5.1.2 \Rightarrow \sqrt{n}\epsilon_2 & \geq \|E_2\|_2 = \|\hat{Q}\hat{Q}^T - I\|_2 \\ & \geq \|\hat{Q}\hat{Q}^T\|_2 - \|I\|_2 = \|\hat{Q}^T\|_2^2 - 1 \\ \Rightarrow \|\hat{Q}^T\|_2 & \leq \sqrt{1 + \sqrt{n}\epsilon_2} \end{aligned}$$

From (5.1.1), $T\hat{Q}\hat{Q}^T - \hat{Q}\hat{\Lambda}\hat{Q}^T = |\hat{\lambda}|_{max} E_1 \hat{Q}^T$. So,

$$\begin{aligned} \hat{Q}\hat{\Lambda}\hat{Q}^T & = T\hat{Q}\hat{Q}^T - |\hat{\lambda}|_{max} E_1 \hat{Q}^T \\ & = T(I + E_2) - |\hat{\lambda}|_{max} E_1 \hat{Q}^T \\ & = T + E \end{aligned}$$

where $E = TE_2 - |\hat{\lambda}|_{max} E_1 \hat{Q}^T$ and so

$$\begin{aligned} \|E\|_2 & \leq |\lambda|_{max} \|E_2\|_2 + |\hat{\lambda}|_{max} \|E_1\|_2 \|\hat{Q}^T\|_2 \\ & \leq \sqrt{n}\left(|\lambda|_{max}\epsilon_2 + |\hat{\lambda}|_{max}\epsilon_1 \sqrt{1 + \sqrt{n}\epsilon_2}\right) \end{aligned}$$

as desired. □

Note that if we define $E_3 = \hat{Q}^T\hat{Q} - I$ and have $\bar{\mathcal{O}} \equiv \max_i \|E_3 \mathbf{e}_i\|_2 \leq \epsilon_2$ then

$$\|E\|_2 \leq \sqrt{n}\left(|\lambda|_{max}\epsilon_2 \sqrt{\frac{1 + \sqrt{n}\epsilon_2}{1 - \sqrt{n}\epsilon_2}} + |\hat{\lambda}|_{max}\epsilon_1 \sqrt{1 + \sqrt{n}\epsilon_2}\right)$$

Since

$$\|E_2\|_2 \leq \|E_3\|_2 \sqrt{\frac{1 + \sqrt{n}\epsilon_2}{1 - \sqrt{n}\epsilon_2}} \tag{5.1.2}$$

Proof: of (5.1.2)

$$\begin{aligned}
E_2 &= \hat{Q}\hat{Q}^T\hat{Q}\hat{Q}^{-1} - I \\
&= \hat{Q}\left(\hat{Q}^T\hat{Q} - I\right)\hat{Q}^{-1} \\
&= \hat{Q}E_3\hat{Q}^{-1} \\
\|E_2\|_2 &\leq \|\hat{Q}\|_2\|E_3\|_2\|\hat{Q}^{-1}\|_2
\end{aligned} \tag{5.1.3}$$

and

$$\textit{Lemma } 5.1.3 \Rightarrow \underbrace{\|Q^{-1}Q^{-T}\|_2}_{\|} = \|(I+E_3)^{-1}\|_2 \leq \frac{1}{1-\|E_3\|_2}$$
$$\|Q^{-1}\|_2^2 \quad \leq \frac{1}{1-\sqrt{n}\epsilon_2}$$

so, from (5.1.3)

$$\begin{aligned}
\|E_2\|_2 &\leq \sqrt{1+\sqrt{n}\epsilon_2}\|E_3\|_2\sqrt{\frac{1}{1-\sqrt{n}\epsilon_2}} \\
\|E_2\|_2 &\leq \|E_3\|_2\sqrt{\frac{1+\sqrt{n}\epsilon_2}{1-\sqrt{n}\epsilon_2}}
\end{aligned}$$

as stated. □

As long as $\sqrt{n}\epsilon_2 \ll 1$, this bound is meaningful. Our orthogonality residual is defined by $\bar{\mathcal{O}}$; for our purposes it serves to bound the error entirely adequately.

Furthermore, note that when

$$\|T\hat{Q} - \hat{Q}\hat{\Lambda}\|_2 \leq \epsilon_1 \text{ and}$$

$$\|\hat{Q}\hat{Q}^T - I\|_2 \leq \epsilon_2,$$

the error matrix E is bounded by

$$\|E\|_2 \leq |\lambda|_{max}\epsilon_2 + |\lambda|_{max}\epsilon_1\sqrt{1+\epsilon_2},$$

which is independent of the matrix order. The maximum column 2-norm is used in practice because it is less expensive to compute.

5.2. Test Matrices

The following matrix types were used to evaluate the performance of divide and conquer relative to QR and bisection with inverse iteration:

1. Symmetric dense matrices with random entries uniformly distributed on (-1,1)

2. Symmetric tridiagonal matrices with random entries designed to "simulate" a dense matrix as above after it has been reduced to tridiagonal form—the diagonals are given by

$$a_i = rnd(-1, 1)$$

and the off-diagonals are given by

$$b_i = \pm\sqrt{\sum_{k=i}^{n}[rnd(0,1)]^2} \tag{5.2.1}$$

[20].

3. Symmetric dense and tridiagonal matrices generated by the LAPACK test code `slatms` [1]. These matrices are of the form UDU^T where U is a random orthogonal matrix and D is a diagonal matrix chosen according to a "mode" as follows. Three of the possible six modes were tested. In each mode description, $D(1:n)$ is the array of eigenvalues, and eps = machine epsilon $= 2^{-24} \approx 5.96 \times 10^{-8}$ for IEEE single precision.

   - mode = 3: sets $D(i) = eps^{\frac{i-1}{n-1}}$ (eigenvalues geometrically distributed from 1 to eps)

   - mode = 4: sets $D(i) = 1 - \frac{i-1}{n-1}(1 - eps)$ (eigenvalues arithmetically distributed from 1 to eps)

   - mode = 5: sets D to random numbers in the range $(eps, 1)$ such that their logarithms are uniformly distributed.

4. 10 glued 21×21 Wilkinson matrices ($glue = \delta = 10^{-4}$)

$$W = \begin{bmatrix} 10 & 1 & & & & \\ 1 & 9 & 1 & & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & 0 & \ddots & \\ & & & \ddots & \ddots & 1 \\ & & & & 1 & 10 \end{bmatrix}$$

$$T = \begin{bmatrix} W & {\scriptstyle\delta} & & & \\ {\scriptstyle\delta} & W & {\scriptstyle\delta} & & \\ & {\scriptstyle\delta} & \ddots & \ddots & \\ & & \ddots & W & {\scriptstyle\delta} \\ & & & {\scriptstyle\delta} & W \end{bmatrix}$$

5. Tridiagonal matrices T of dimensions 100, 200, 300, 400, and 500 with $a_i = 2$ and $b_i = 1$.

6. The tridiagonal in BCSST08.dat: a 1047×1047 test tridiagonal matrix from running Lanczos on an example from the Boeing-Harwell collection. See figure 5.2.1 for a plot of the elements in this tridiagonal matrix.

7. The tridiagonal in LUND.dat: analogous to BCSST08.dat, 147×147. See figure 5.2.2 for a plot of the elements in this tridiagonal matrix.

5.3. General Results

On a DEC Alpha using optimized Basic Linear Algebra Subroutines (BLAS), divide and conquer was uniformly the fastest algorithm by a large margin for large symmetric tridiagonal eigenproblems. When Fortran BLAS were used, bisection/inverse iteration was somewhat faster (up to a factor of 2) for very large matrices ($n \geq 500$) without clustered eigenvalues. When eigenvalues were clustered, divide and conquer was up to 80 times faster. The speedups over QR were so large in the tridiagonal case that the overall problem, including reduction to tridiagonal form sped up by a factor of 2.5 over QR for $n \geq 500$.

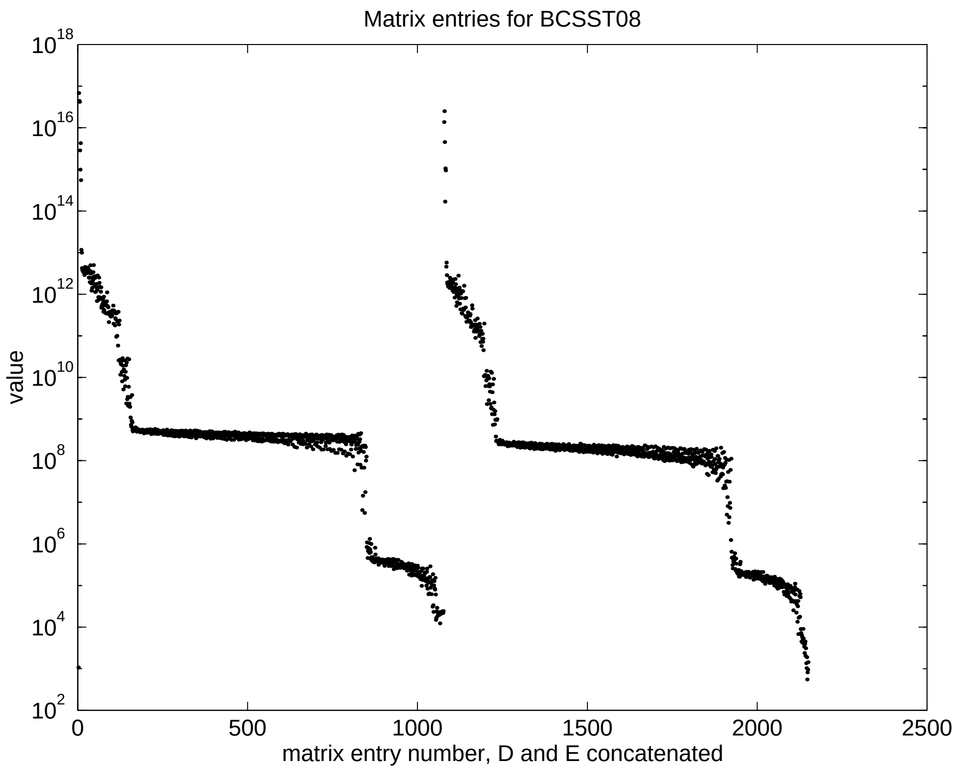


Figure 5.2.1: BCS elements

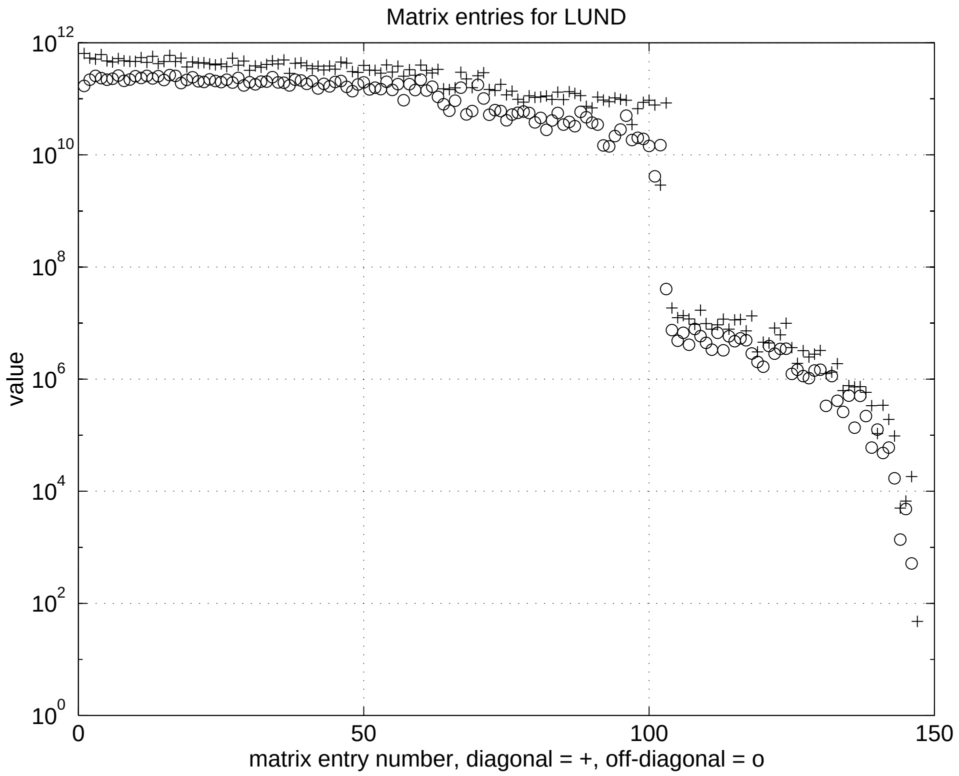


Figure 5.2.2: LUND elements

Nearly universally, the matrix of eigenvectors generated by divide and conquer suffered the least loss of orthogonality. The smallest eigensystem residual ($\|T\hat{Q} - \hat{Q}\hat{\Lambda}\|$) usually came from the eigensystem generated by bisection/inverse iteration, with divide and conquer coming a close second (except on those occasions where inverse iteration failed to compute orthogonal vectors).

The plots will often be labeled with "Exx" where "x" is some other letter. These abbreviations refer to the code which was used to generate the given results. EDC refers to the divide and conquer code (`sstedc`), EBZ is the bisection with inverse iteration code (`sstebz` and `sstein`), EQR is the QR code (`ssteqr`), and ERI refers to the root-free QR code with inverse iteration (a slightly modified `ssterf` and `sstein`).

It is worth noting that all of the codes contain criteria for "splitting" matrices with exceptionally small off-diagonals. These criteria are not the same for different codes. For the purpose of comparison, it was decided that it would be fairest to alter some of the codes so that they all used the same splitting criteria. To this end, the following splitting condition was used:

T splits between a_i and a_{i+1} if

$$|b_i| < eps\sqrt{|a_i|} \cdot \sqrt{|a_{i+1}|}.$$

This is the splitting condition which was coded into `sstedc`.

5.4. Results for Dense Matrices

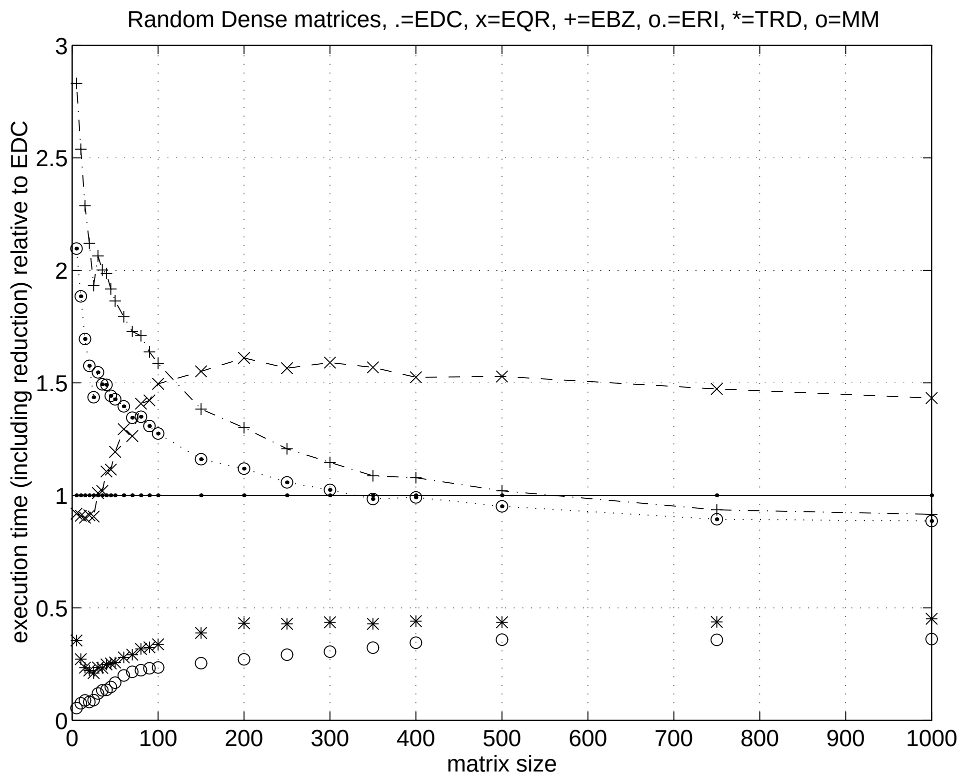


Figure 5.4.1: Relative times on random dense matrices using the DEC Alpha with Fortran BLAS

The random symmetric dense matrices generated have the property that their eigenvalues are fairly evenly distributed. For inverse iteration this is good since less reorthogonalization of eigenvectors is required. For divide and conquer it is bad since this means that there will be little deflation within the intermediate problems. QR and root-free QR seem mostly unaffected by changes in matrix type.

The bisection/inverse iteration combination almost always yields the smallest residual. Divide and conquer is nearly always the next smallest. Sometimes divide and conquer gets a smaller residual than bisection/inverse iteration. This happens when a great deal of deflation takes place within the divide and conquer algorithm—fewer computations mean less loss of accuracy. The root-free QR/inverse iteration combination is fairly unpredictable. In general, QR seems to have the greatest residual of all, but all of the codes are fairly accurate most of the time.

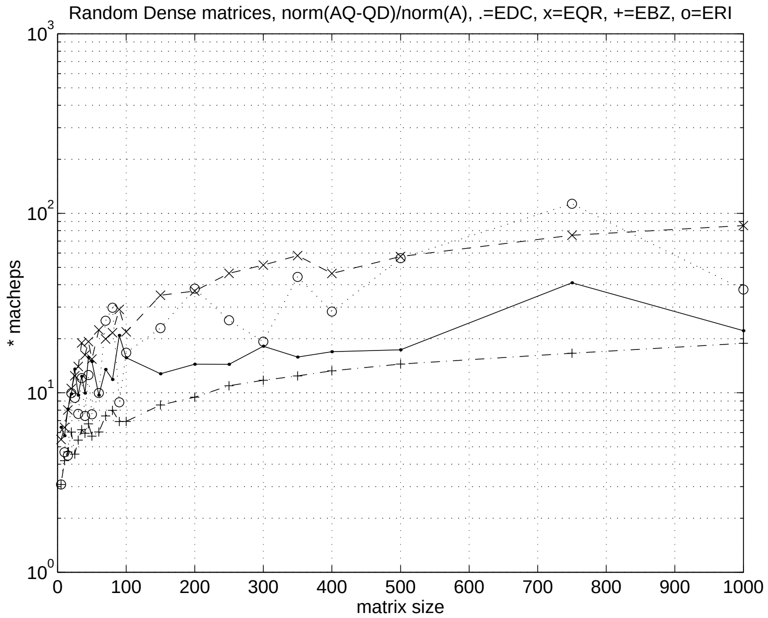


Figure 5.4.2: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on random dense matrices using the DEC Alpha with Fortran BLAS

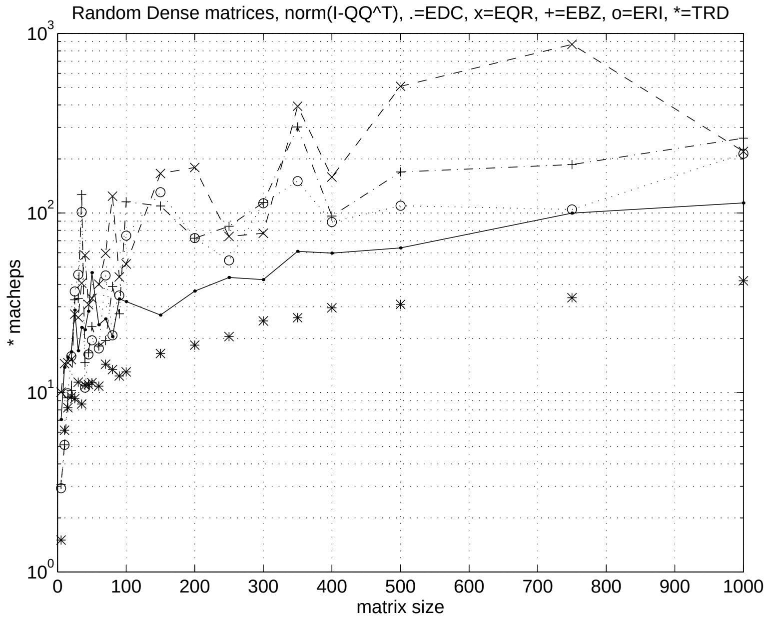


Figure 5.4.3: $\|\hat{Q}^T\hat{Q} - I\|$ on random dense matrices using the DEC Alpha with Fortran BLAS

Divide and conquer nearly always has the smallest error for the orthogonality test; it maintains orthogonal eigenvectors very well at each step, following the orthogonality error from the matrix used to reduce to tridiagonal form (`TRD` above) more closely than any of the other methods. Both of the algorithms making use of inverse iteration generally yield low orthogonality errors, but occasionally inverse iteration will yield vectors which are not orthogonal. When this happens, it often ruins the residual error as well. QR reliably gets decently small residual and orthogonality errors, but it tends to be less accurate than the other codes.

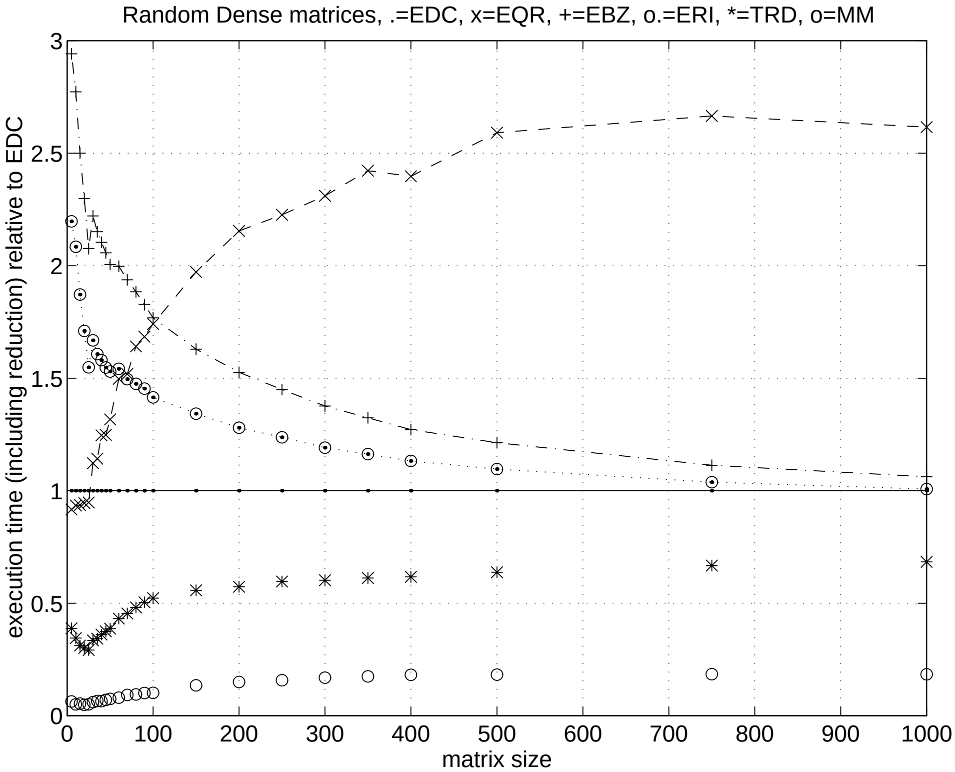


Figure 5.4.4: Relative times on random dense matrices using the DEC Alpha with BLAS from the Digital eXtended Math Library

The optimized BLAS make quite a bit of difference. Since divide and conquer is based on BLAS3 (matrix-matrix multiplies) and bisection/inverse iteration is based on BLAS1 (vector-vector operations), divide and conquer gets a much bigger boost from the use of optimized BLAS. QR, being based on small rotation operations, is not helped at all by the addition of optimized BLAS.

The errors are not independent of the BLAS used. Although the error plots for random dense matrices using DXML are nearly identical to the plots produced with Fortran BLAS, some of the later DXML plots are profoundly different from their Fortran counterparts.

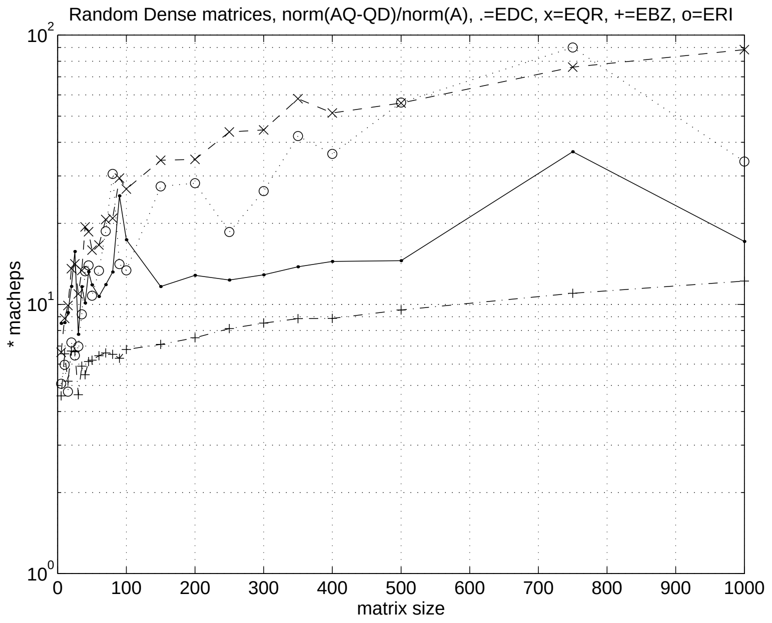


Figure 5.4.5: $\frac{||A\hat{Q}-\hat{Q}\hat{\Lambda}||}{||A||}$ on the DEC Alpha with DXML BLAS

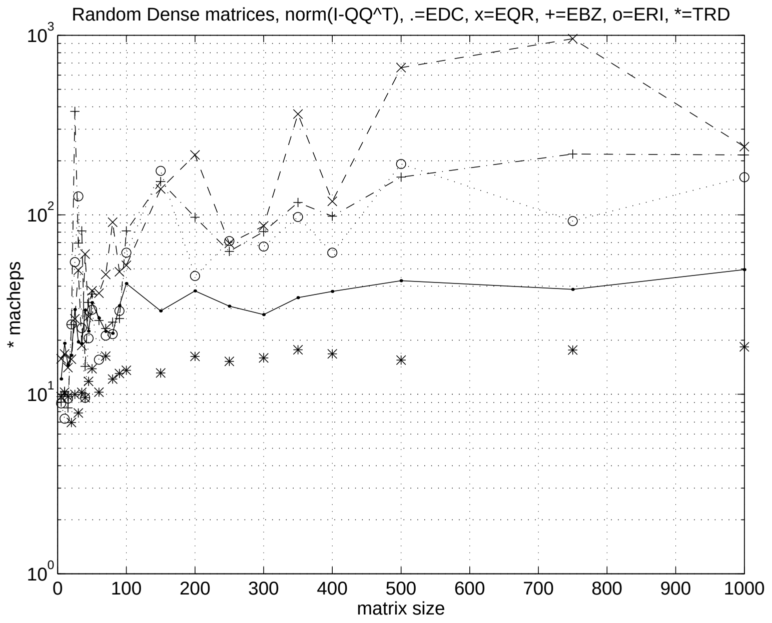


Figure 5.4.6: $\|\hat{Q}^T\hat{Q} - I\|$ on the DEC Alpha with DXML BLAS

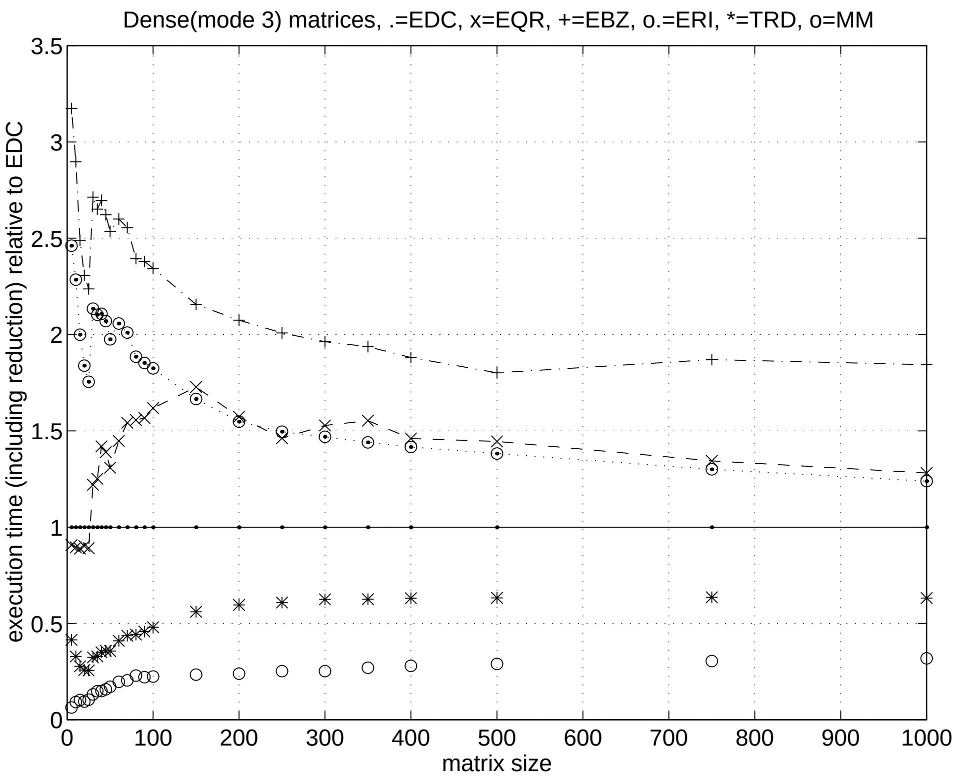


Figure 5.4.7: Relative times on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with Fortran BLAS

The fraction of time spent in the final matrix-matrix multiply is considerably less for these matrices when using Fortran BLAS than it was for the random dense matrices. This is because the matrix of eigenvectors emerging from divide and conquer on the tridiagonal matrix is highly structured due to considerable deflation in the last subproblem in the divide and conquer tree. The Fortran matrix-matrix multiply routine from netlib try to take advantage of matrices with such structure by checking to see if there are zero entries.

The large amount of deflation not only makes the divide and conquer algorithm go very fast, but it also improves its accuracy considerably.

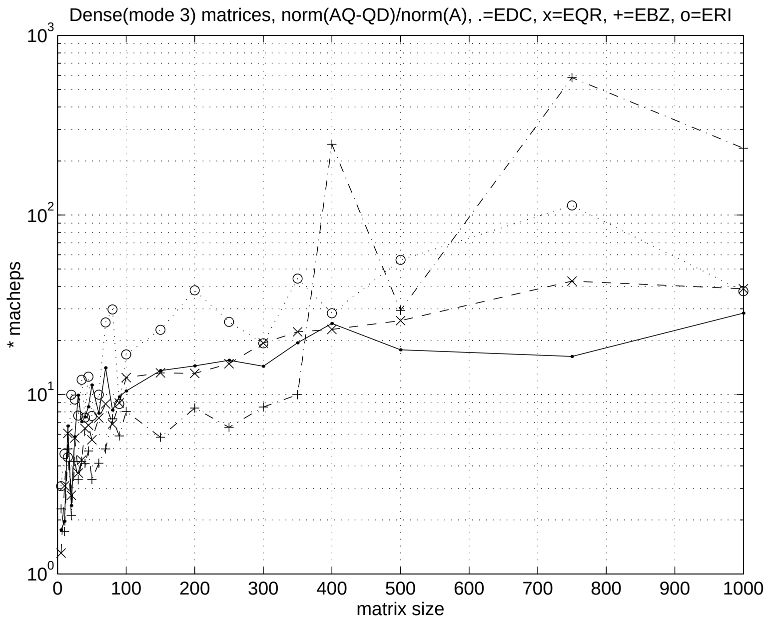


Figure 5.4.8: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with Fortran BLAS

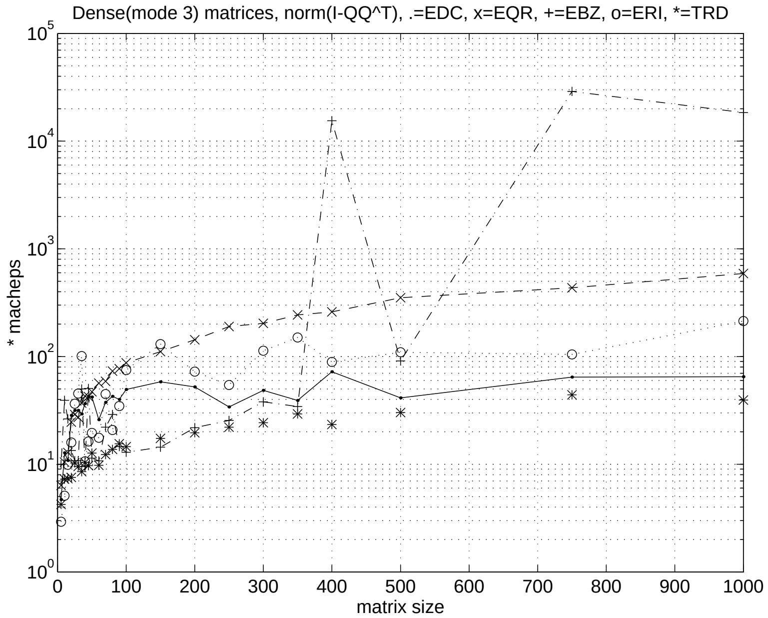


Figure 5.4.9: $\|\hat{Q}^T\hat{Q}-I\|$ on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with Fortran BLAS

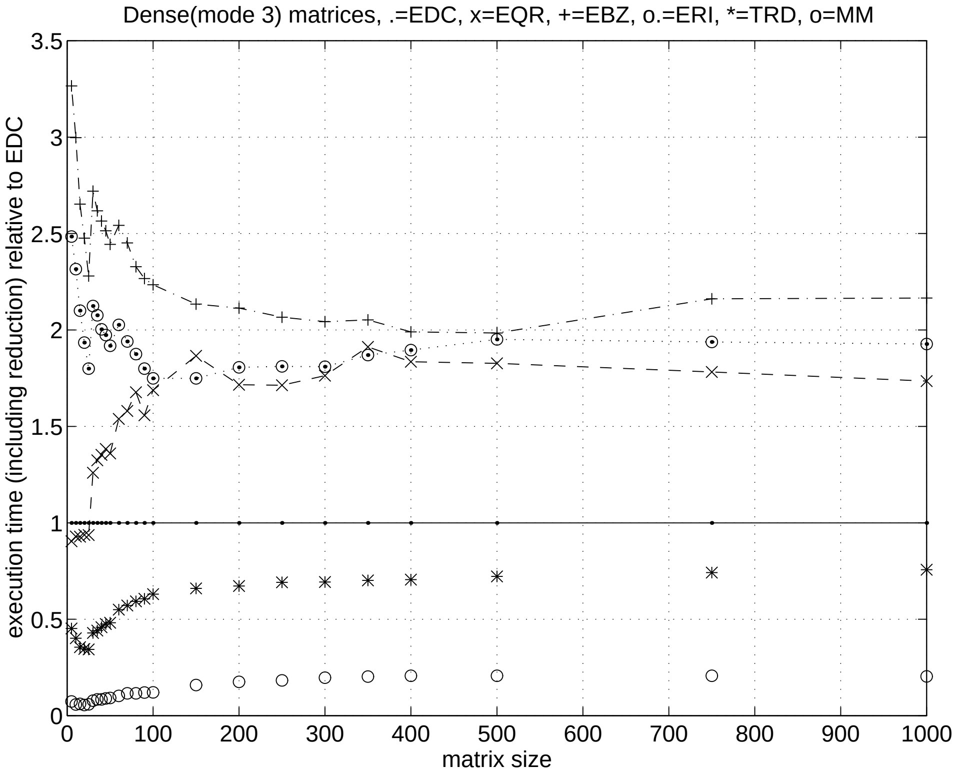


Figure 5.4.10: Relative times on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

The fraction of time spent in the final matrix-matrix multiply appears nearly identical to the same operation on random dense matrices. We can be fairly sure that the highly optimized BLAS do not check for zeros inside the matrices of a matrix-matrix multiplication.

This time the error residuals are significantly different from the same residuals using Fortran BLAS. Notice that QR is unaffected since it does not make use of BLAS.

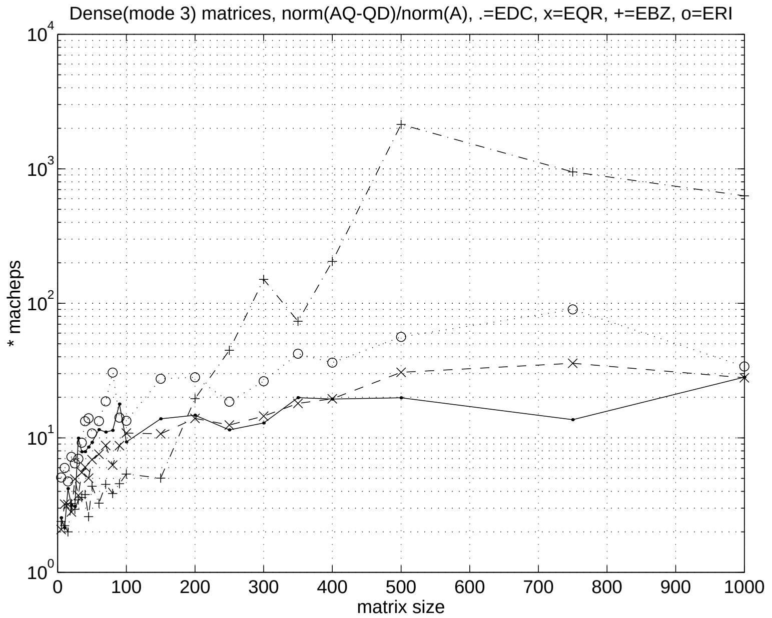


Figure 5.4.11: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

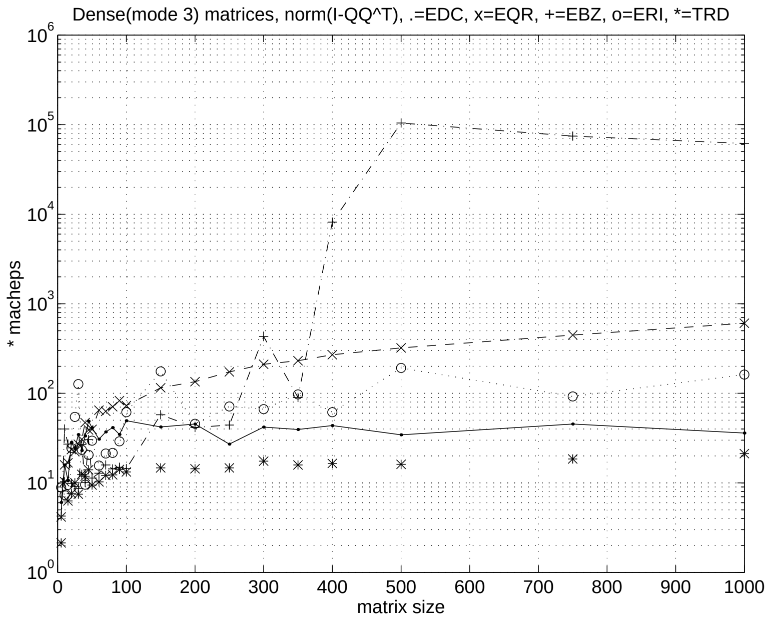


Figure 5.4.12: $\|\hat{Q}^T\hat{Q} - I\|$ on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

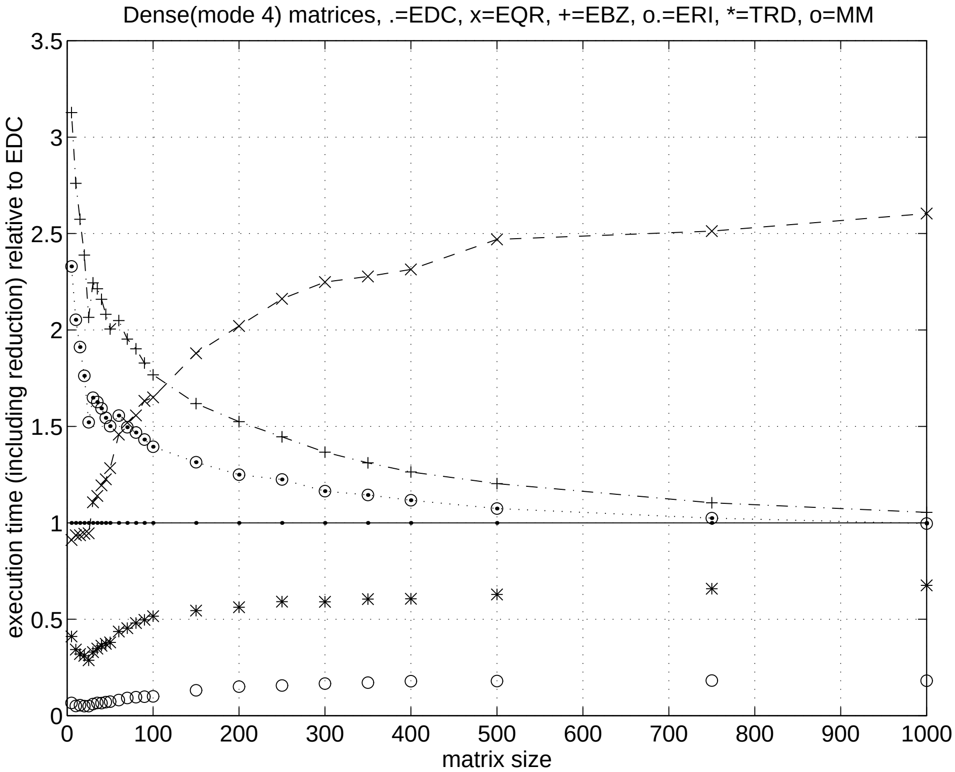


Figure 5.4.13: Relative times on dense matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

This is very similar to the plot for random dense matrices.

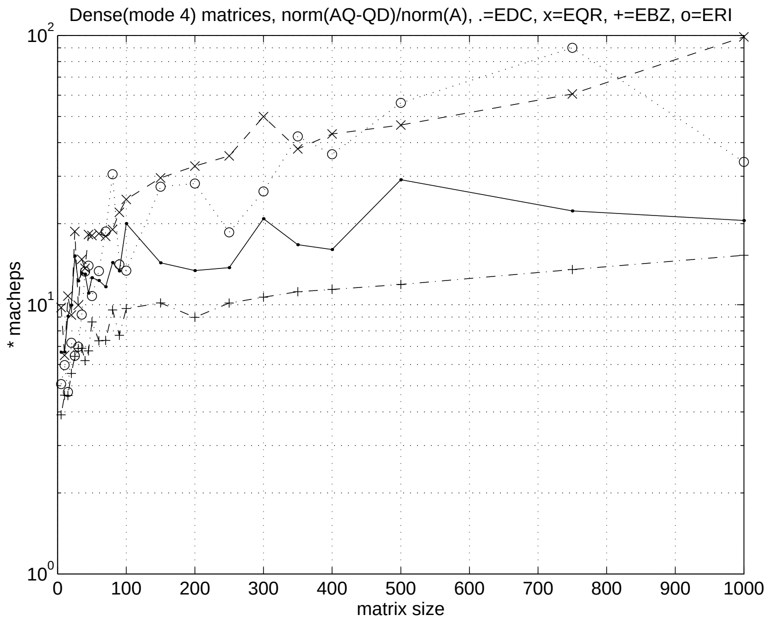


Figure 5.4.14: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on dense matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

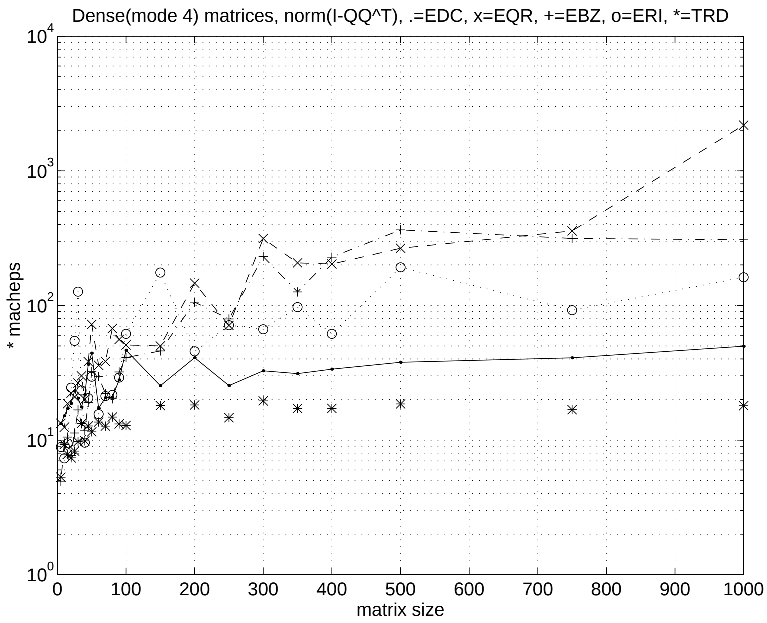


Figure 5.4.15: $\|\hat{Q}^T\hat{Q}-I\|$ on dense matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

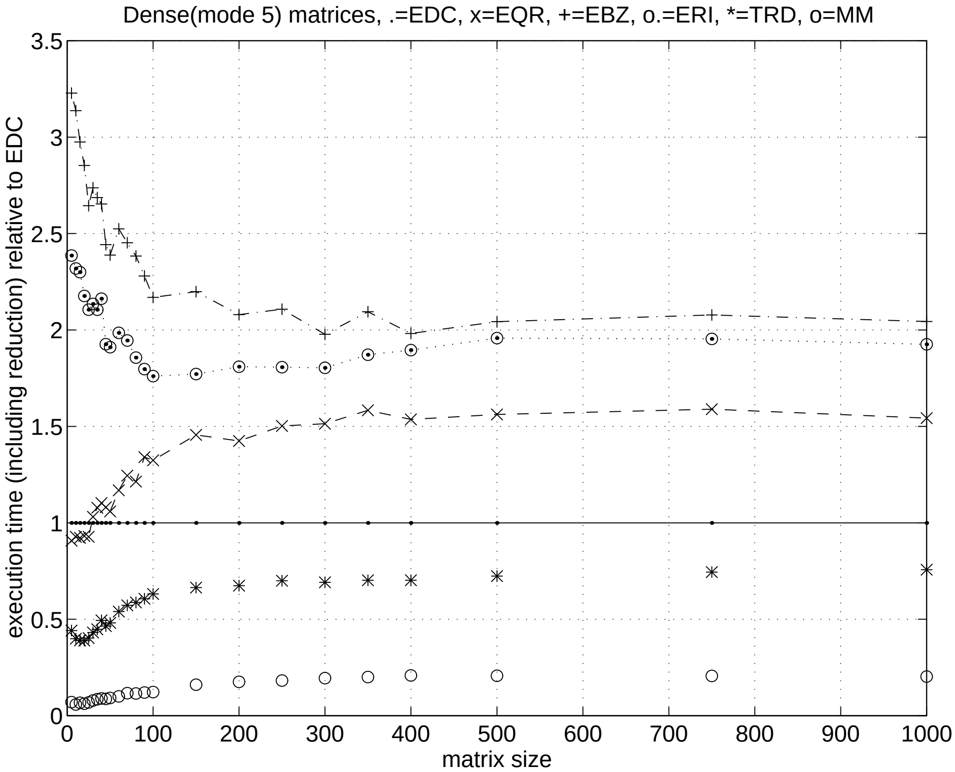


Figure 5.4.16: Relative times on dense matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

This is very similar to the `MODE=3` plot. Both types of matrices have many clustered eigenvalues.

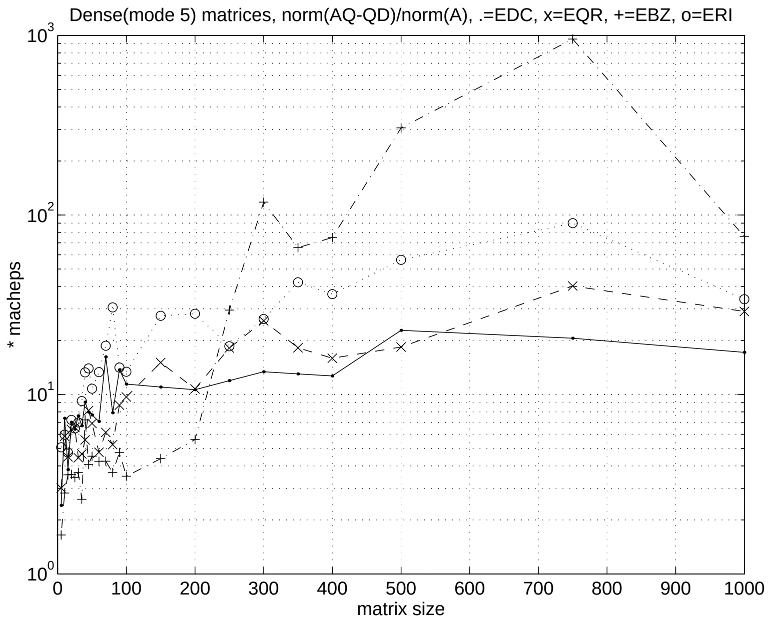


Figure 5.4.17: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on dense matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

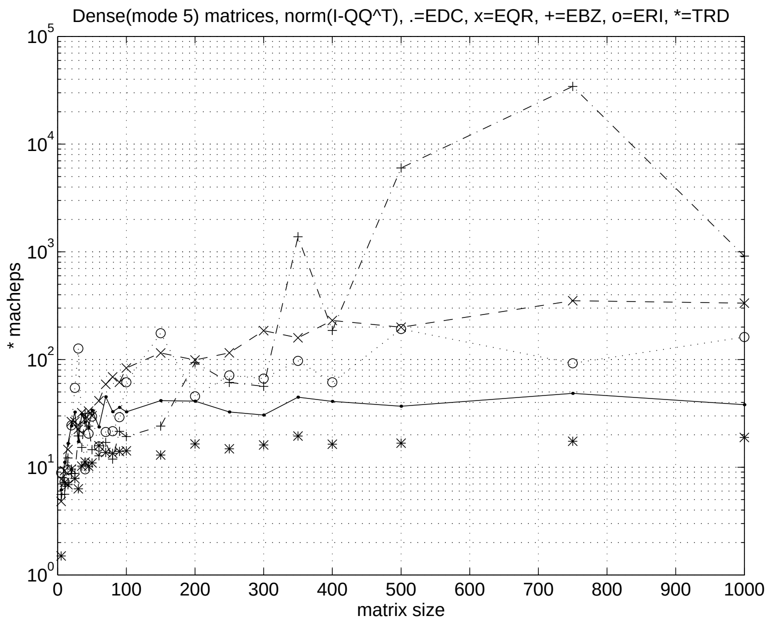


Figure 5.4.18: $\|\hat{Q}^T\hat{Q}-I\|$ on dense matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

5.5. Results for Tridiagonal Matrices

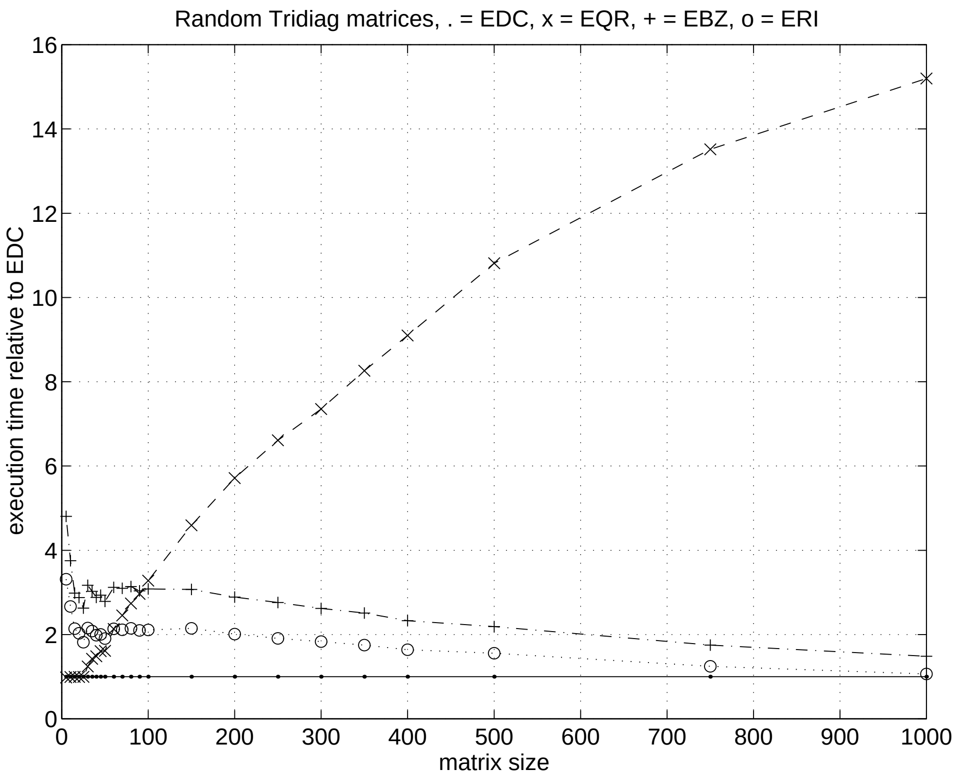


Figure 5.5.1: Relative times on random tridiagonal matrices using the DEC Alpha with DXML BLAS

See 5.2.1 for details on the formulation of these tridiagonal matrices. For the tridiagonal eigenproblem, QR is simply not competitive.

The error residuals are about what we expect.

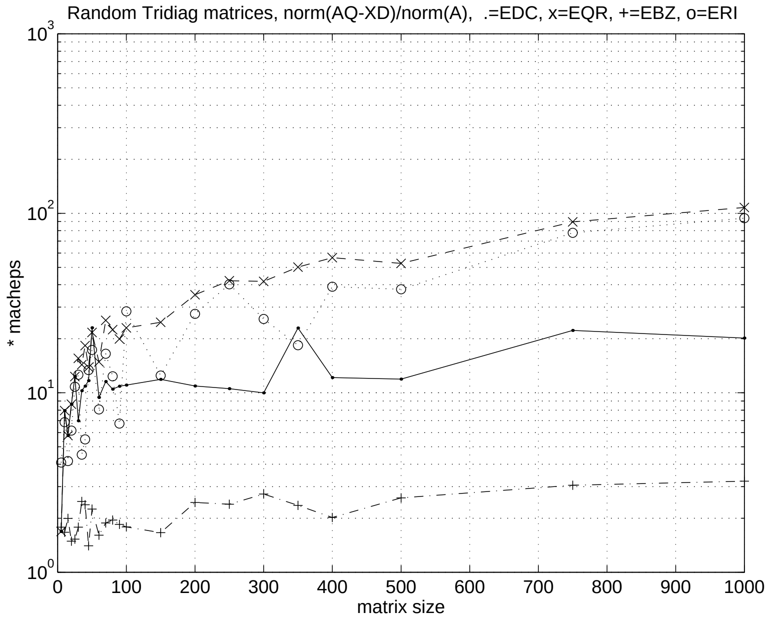


Figure 5.5.2: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on random tridiagonal matrices using the DEC Alpha with DXML BLAS

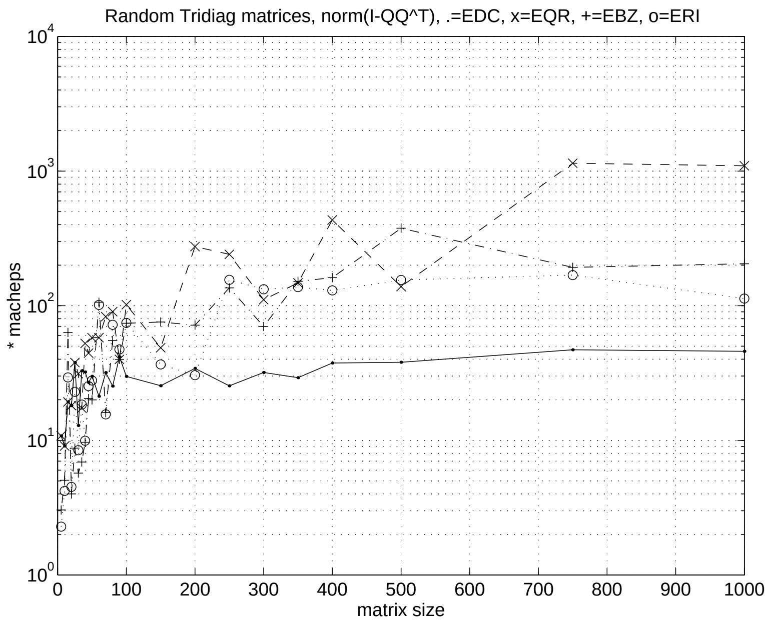


Figure 5.5.3: $\|\hat{Q}^T\hat{Q}-I\|$ on random tridiagonal matrices using the DEC Alpha with DXML BLAS

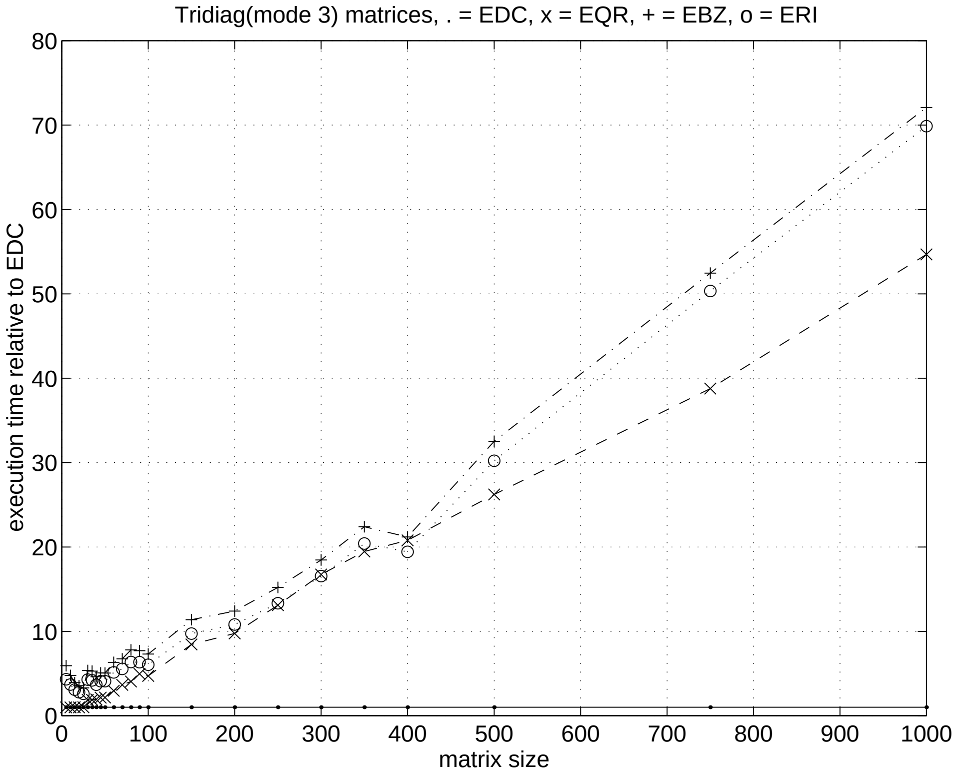


Figure 5.5.4: Relative times on tridiagonal matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

There is a trememdous amount of deflation in this matrix—probably due to a combination of the clustered eigenvalues and the way in which the matrices are constructed (a sequence of Givens rotations applied to a diagonal matrix).

This is a good example of inverse iteration failing to produce orthogonal vectors without adversely affecting the overall eigensystem residual. It is also an excellent example of the exceptional accuracy achieved by divide and conquer when a great deal of deflation occurs.

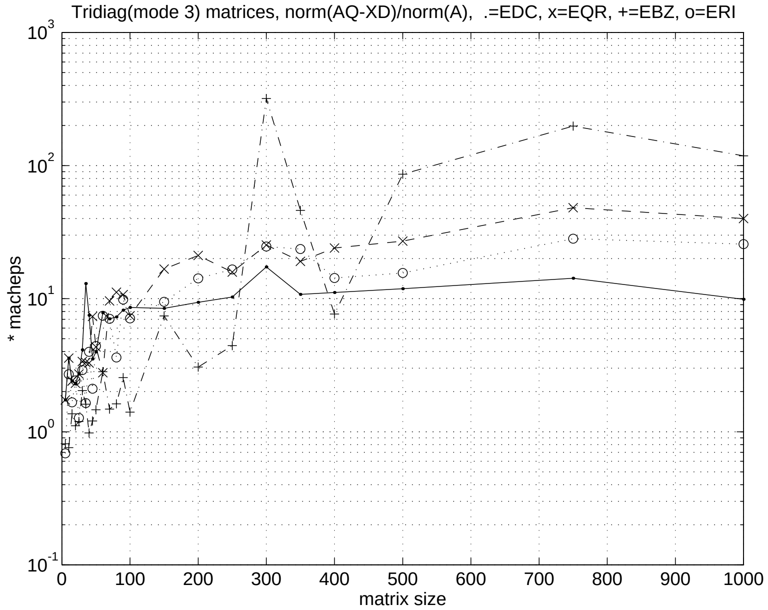


Figure 5.5.5: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on tridiagonal matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

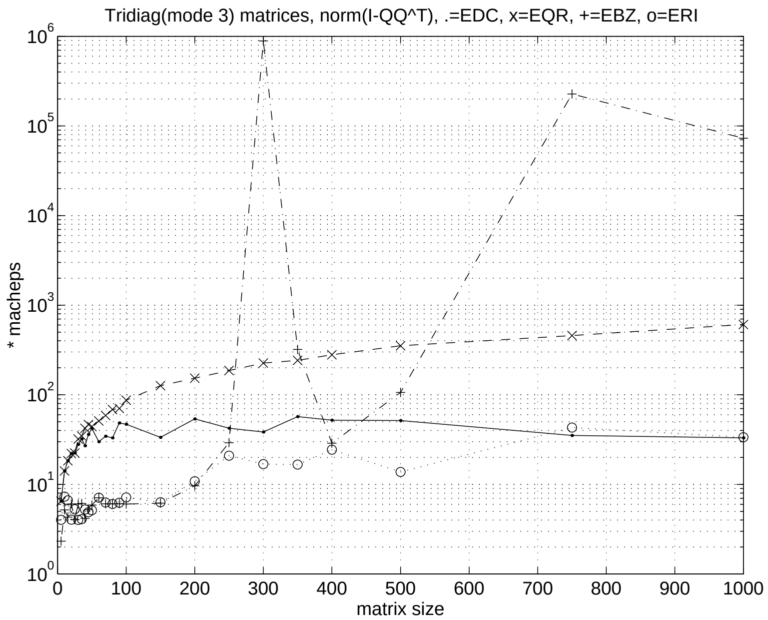


Figure 5.5.6: $\|\hat{Q}^T\hat{Q} - I\|$ on tridiagonal matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with DXML BLAS

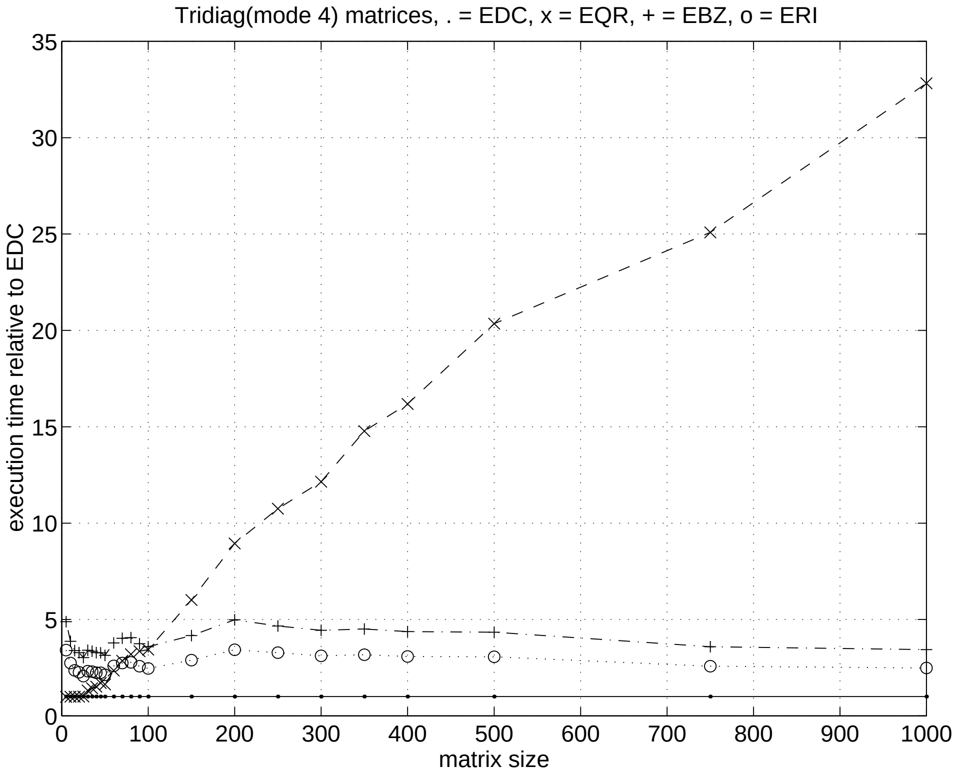


Figure 5.5.7: Relative times on tridiagonal matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

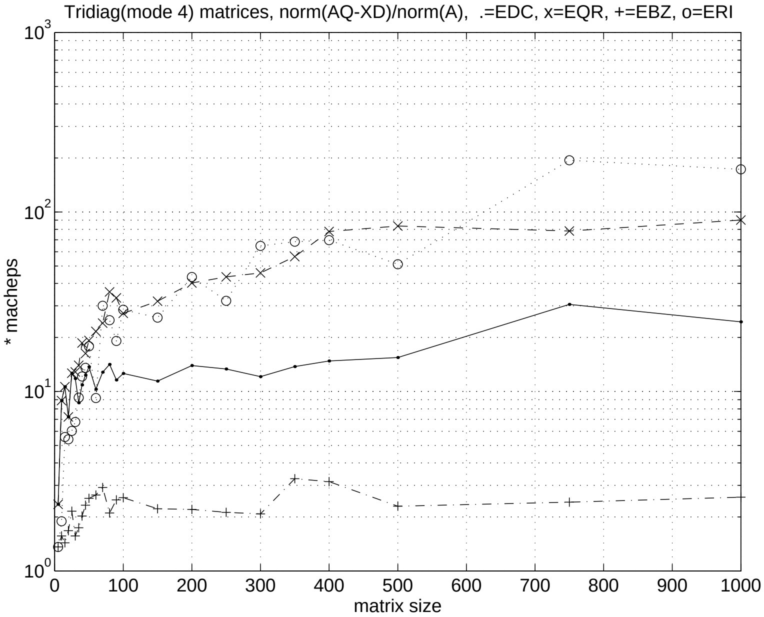


Figure 5.5.8: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on tridiagonal matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

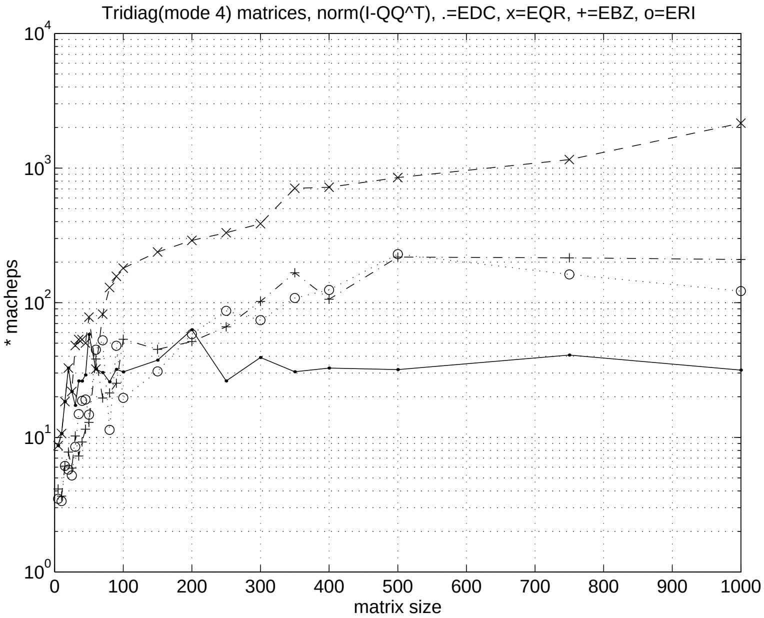


Figure 5.5.9: $\|\hat{Q}^T\hat{Q} - I\|$ on tridiagonal matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with DXML BLAS

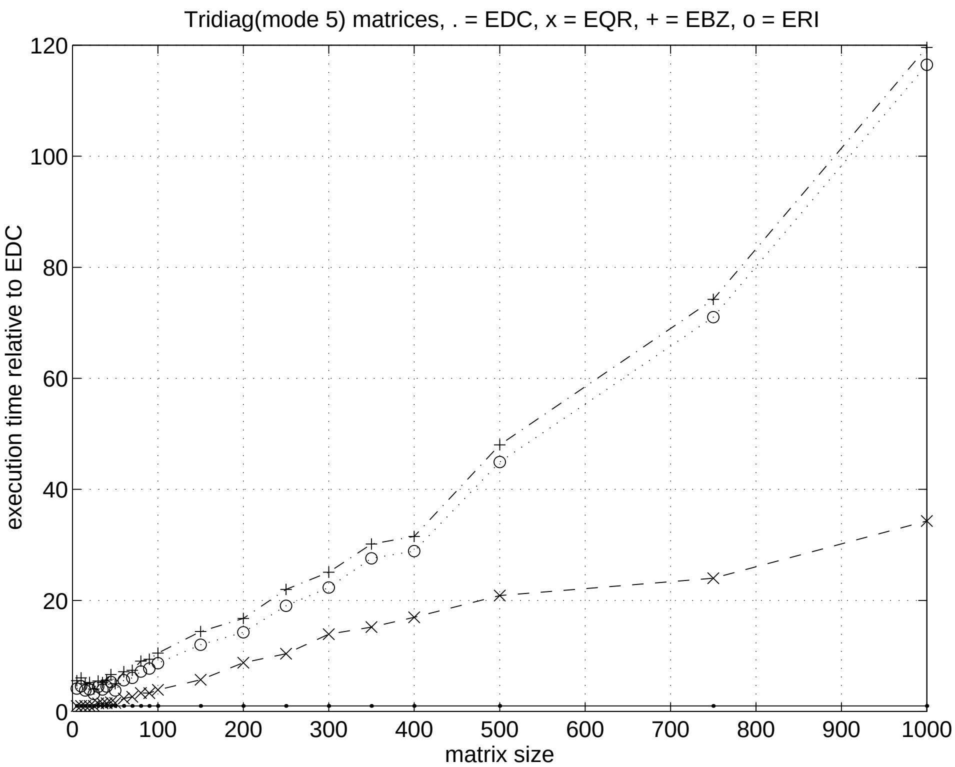


Figure 5.5.10: Relative times on tridiagonal matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

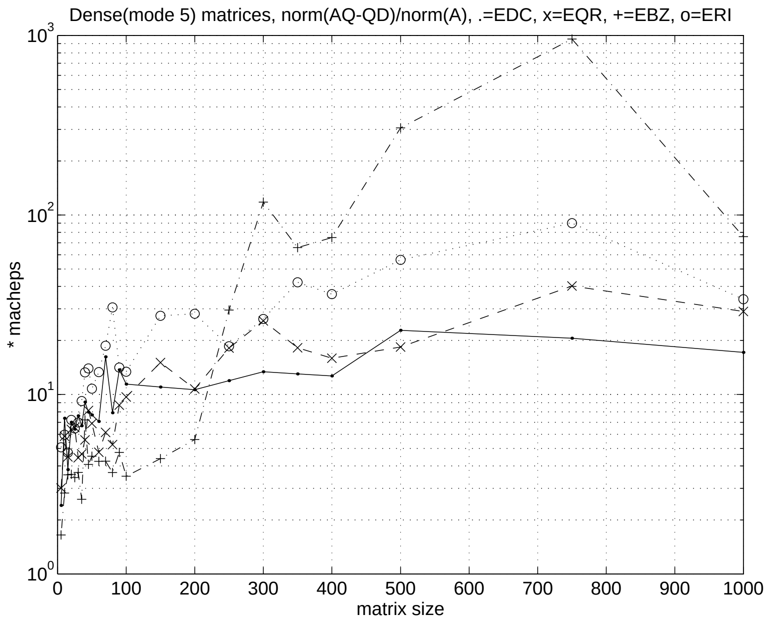


Figure 5.5.11: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on tridiagonal matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

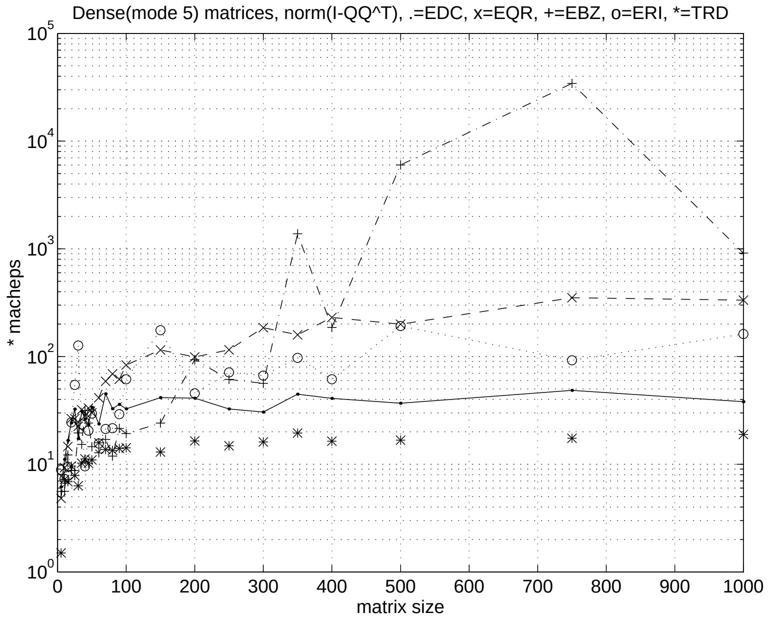


Figure 5.5.12: $\|\hat{Q}^T\hat{Q}-I\|$ on tridiagonal matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with DXML BLAS

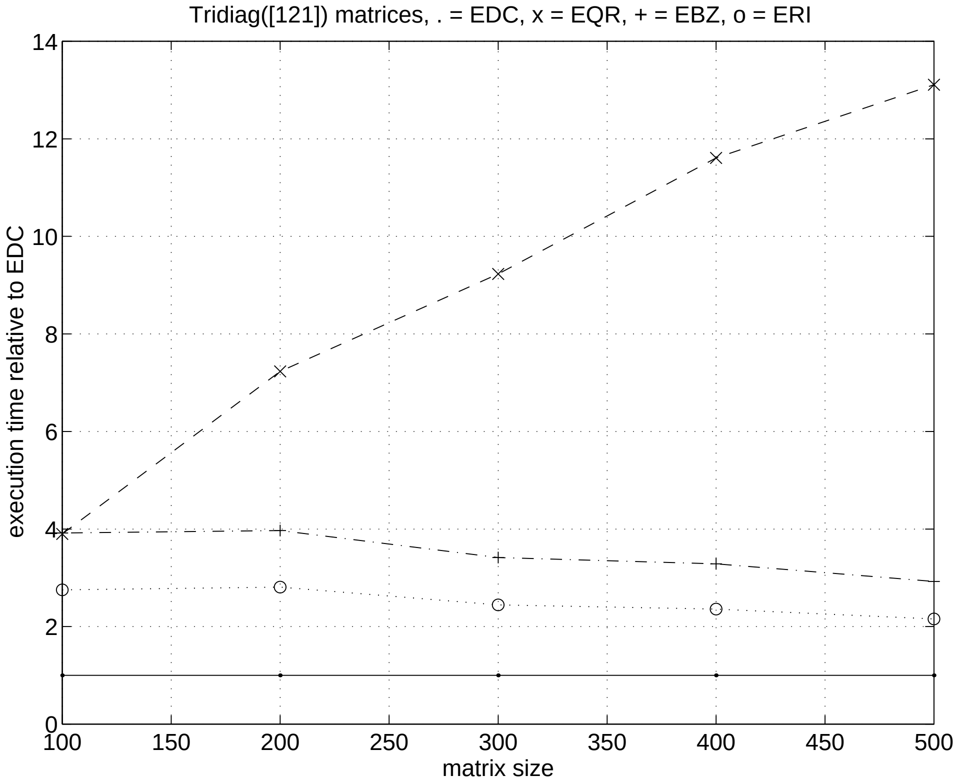


Figure 5.5.13: Relative times on tridiagonal matrices with 2's on the diagonal and 1's on the off-diagonal using the DEC Alpha with DXML BLAS

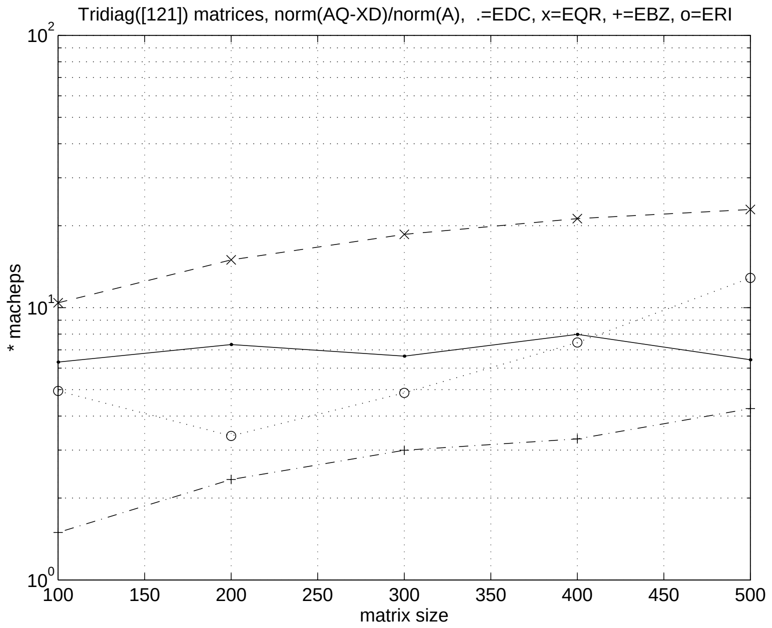


Figure 5.5.14: $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on tridiagonal matrices with 2's on the diagonal and 1's on the off-diagonal using the DEC Alpha with DXML BLAS

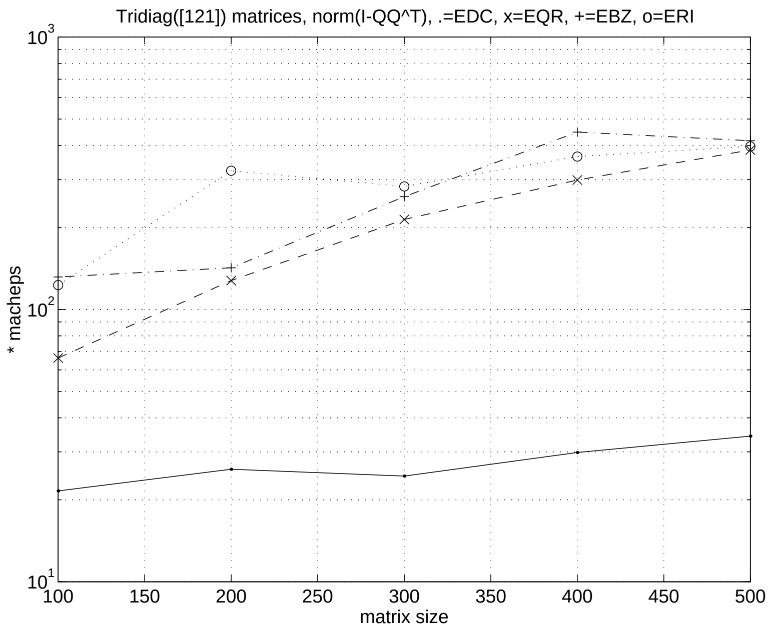


Figure 5.5.15: $\|\hat{Q}^T\hat{Q} - I\|$ on tridiagonal matrices with 2's on the diagonal and 1's on the off-diagonal using the DEC Alpha with DXML BLAS

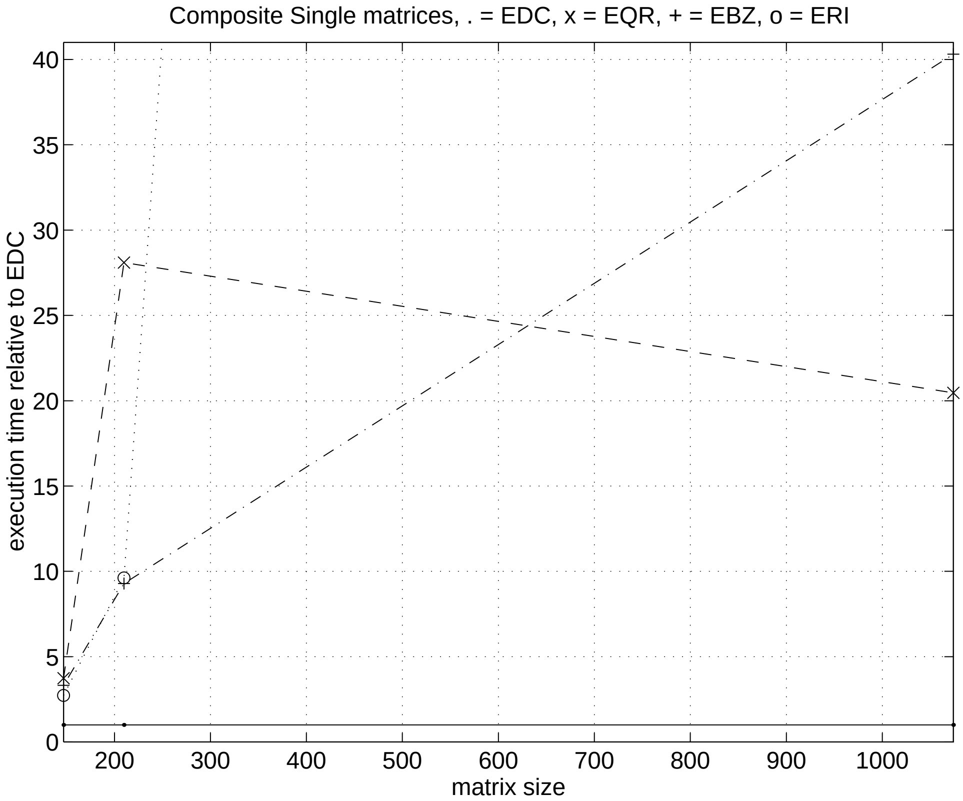


Figure 5.5.16: A composite of relative times on the DEC 5000 with Fortran BLAS

The left-most matrix size is at 147—this is the LUND matrix. Proceeding left to right, the next size is 210—this is for the glued Wilkinson matrices. The final size is for BCSST08 at 1074.

The large BCSST08 matrix has many small eigenvalues. We suspect that root-free QR does not find these values accurately enough for inverse iteration to function quickly. The root-free QR/inverse iteration execution time relative to divide and conquer is far off the scale here, taking close to 700 times longer.

For its additional effort, the eigensystem that root-free QR/inverse iteration computes yields a residual much lower than any other, although all of the algorithms yield residuals very close to machine precision.

The eigenvectors computed by inverse iteration yield errors which are calculated to be zero. That is why those errors do not appear on the semilog plot.

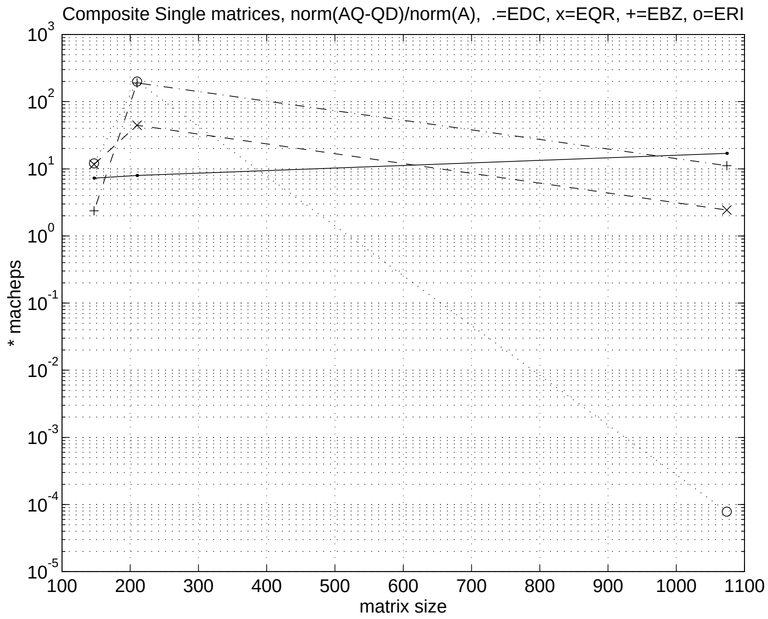


Figure 5.5.17: A composite of $\frac{\|A\hat{Q}-\hat{Q}\hat{\Lambda}\|}{\|A\|}$ on the DEC 5000 with Fortran BLAS

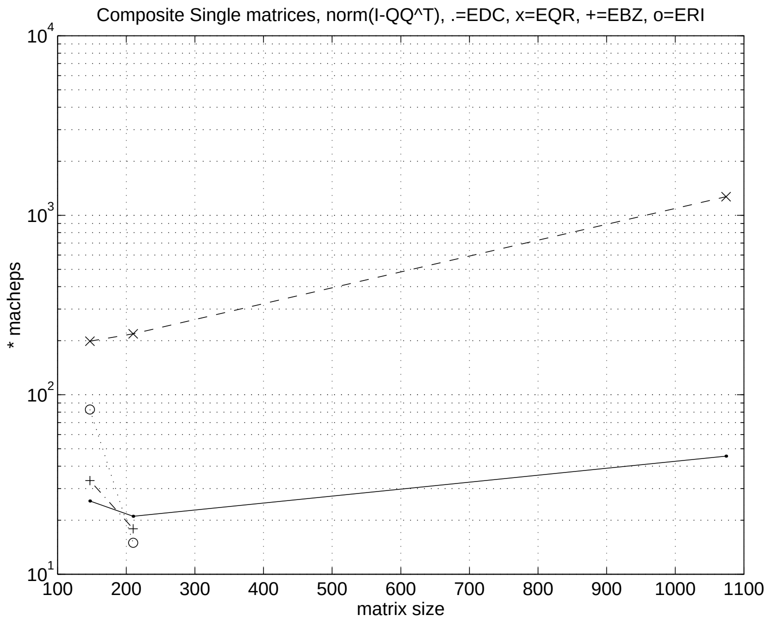


Figure 5.5.18: A composite of $\|\hat{Q}^T\hat{Q} - I\|$ on the DEC 5000 with Fortran BLAS

5.6. Theoretical Performance Analysis for Dense Matrices

There are two possibilities for computing the full eigensystem of a dense matrix using the divide and conquer code. Both begin by finding a tridiagonal factorization: $A = UTU^T$. The subsequent call to `sstedc` could use either option `COMPQ` = "V" or option `COMPQ` = "I". If option "V" is chosen, then the algorithm directly computes $A = Z\Lambda Z^T$ from the input of T and U. If option "I" is chosen, then the algorithm computes $T = Q\Lambda Q^T$ from the input of T. From this it is easy to compute the eigenvectors of A:

$$A = UTU^T = UQ\Lambda Q^T U^T = Z\Lambda Z^T$$

So the eigenvectors of A are the columns of $Z = UQ$; a single large matrix-matrix multiply is needed in addition to the computation of the eigensystem of the tridiagonal matrix. In this section, we will give a theoretical analysis of which algorithm is faster.

The differences in the two methods are in the calculation of $\mathbf{z}$, the nature of the matrix-matrix multiplies into U, and the amount of workspace required. In the following summaries, δ will represent the fraction of eigenvalues on a level which were deflated. For simplicity, I assume here that the overall matrix size $n = 2^{\mathtt{t}+1}$, that the matrix will be split in the divide step until the subproblems are of size 2, that the percentage of deflation is the same for each problem on each level, and that this deflation is all of the sort in which the value in $\mathbf{z}$ is small. $\alpha = 1 - \delta$ will be the fraction of non-deflated values in each problem.

- Using option "I", no computation is needed to form $\mathbf{z}$, for problem j on level i it is simply read from $\begin{bmatrix} {}_{i+1}Q_{2j-1} & \\ & {}_{i+1}Q_{2j} \end{bmatrix}$. Keeping ${}_iQ$ up to date, however, is non-trivial.

  Let `TLVLS` $= \mathtt{t} = \lg n - 1$, this means that the levels are numbered from 0 to $\mathtt{t}$.

  On the i^{th} level, $0 \le i \le \mathtt{t}$, there are 2^i intermediate ${}_iQ_j$ matrices of size $\alpha 2^{\mathtt{t}-i+1}$.

  For problem j on level i, $0 \le i \le \mathtt{t}-1, 1 \le j \le 2^i$, there will be 2 matrix-matrix multiplies of the form

$$\breve{Q}_{[2^{\mathtt{t}-i} \times \alpha 2^{\mathtt{t}-i}]} \tilde{Q}_{[\alpha 2^{\mathtt{t}-i} \times \alpha 2^{\mathtt{t}+1-i}]}$$

requiring

$$2(2^{\mathtt{t}-i} \cdot \alpha 2^{\mathtt{t}-i} \cdot \alpha 2^{\mathtt{t}+1-i}) = 2\alpha^2 2^{3(\mathtt{t}+1)-3i-2} = \frac{1}{2}\frac{\alpha^2 n^3}{2^{3i}}$$

floating point multiplies and the same number of floating point adds. For simplicity, whenever we refer to a number of "floating point multiplies and adds" we mean "floating point multiplies and the same number of floating point adds." All of the matrix-matrix multiplies on level i will cost

$$2^i \frac{1}{2}\frac{\alpha^2 n^3}{2^{3i}} = \frac{1}{2}\frac{\alpha^2 n^3}{2^{2i}} \text{ floating point multiplies and adds.}$$

The entire algorithm will require matrix-matrix multiplies for $0 \le i \le \mathtt{t} - 1$:

$$\begin{aligned}
\sum_{i=0}^{\mathtt{t}-1} \frac{1}{2}\frac{\alpha^2 n^3}{2^{2i}} &= \frac{1}{2}\alpha^2 n^3 \sum_{i=0}^{\infty} \frac{1}{2^{2i}} - \sum_{i=\mathtt{t}}^{\infty} \frac{1}{2^{2i}} \\
&= \frac{1}{2}\alpha^2 n^3 \frac{1}{1-\frac{1}{4}}\left(1 - \frac{1}{4^{\mathtt{t}}}\right) \\
&= \frac{2}{3}\alpha^2 n^3 \left(1 - \frac{1}{4^{\mathtt{t}}}\right) \text{ floating point multiplies and adds.}
\end{aligned}$$

One final full-size matrix-matrix multiply will be required to form $Z = UQ$. This will cost n^3 multiplies and adds for a total of $[1 + \frac{2}{3}\alpha^2\left(1 - \frac{1}{4^{\mathtt{t}}}\right)]n^3$ floating point multiplies and adds which are not also performed by using option "V".

- Using the "V" option, there is some work to be done in calculating $\mathbf{z}$. This is done by a series of matrix-vector multiplies. Also, U must be updated by each intermediate eigenvector matrix.

  - For a problem on level i computation of $\mathbf{z}$ involves a pair of matrix-vector multiplies for each level below i, not including level $\mathtt{t}$. This costs

$$\begin{aligned}
2\sum_{k=i+1}^{\mathtt{t}-1}\left(\alpha 2^{\mathtt{t}+1-k}\right)^2 &= 2\alpha^2 n^2 \sum_{k=i+1}^{\mathtt{t}-1} \frac{1}{2^{2k}} \\
&= \frac{2}{3}\alpha^2 n^2 \frac{1}{2^{2i+2}} \frac{1}{1-\frac{1}{4}}\left(1 - \frac{1}{4^{\mathtt{t}}}\right) \\
&= \frac{2}{3}\frac{\alpha^2 n^2}{2^{2i}}\left(1 - \frac{1}{4^{\mathtt{t}}}\right) \qquad (5.6.1)
\end{aligned}$$

floating point multiplies and adds.

For every problem on level i, that comes to a total of

$$\sum_{j=1}^{2^i} \frac{2}{3}\frac{\alpha^2 n^2}{2^{2i}}\left(1-\frac{1}{4^{\mathtt{t}}}\right) = \frac{2}{3}\frac{\alpha^2 n^2}{2^i}\left(1-\frac{1}{4^{\mathtt{t}}}\right) \text{ floating point multiplies and adds.}$$

For the entire algorithm, we calculate $\mathbf{z}$'s on levels $i = \mathtt{t}-1,\ldots,0$:

$$\sum_{i=0}^{\mathtt{t}-1} \frac{2}{3}\frac{\alpha^2 n^2}{2^i} = \frac{4}{3}\alpha^2 n^2\left(1-\frac{1}{4^{\mathtt{t}}}\right) \text{ floating point multiplies and adds.}$$

– Each intermediate eigenvector matrix must be multiplied into U. For problem j on level i, this means one matrix-matrix multiply:

$$U_{[n\times\alpha 2^{\mathtt{t}+1-i}]}Q_{[\alpha 2^{\mathtt{t}+1-i}\times\alpha 2^{\mathtt{t}+1-i}]}$$

yielding

$$n\alpha^2 2^{2(\mathtt{t}+1)-2i} = \frac{\alpha^2 n^3}{2^{2i}} \text{ floating point multiplies and adds.}$$

Across level i:

$$2^i\frac{\alpha^2 n^3}{2^{2i}} = \frac{\alpha^2 n^3}{2^i}\text{floating point multiplies and adds.}$$

The entire algorithm requires

$$\begin{aligned}\sum_{i=0}^{\mathtt{t}} \alpha^2\frac{n^3}{2^i} &< \alpha n^3 \frac{1}{1-\frac{1}{2}}\left(1-\frac{1}{2^{\mathtt{t}+1}}\right)\\ &= 2\alpha^2 n^3\left(1-\frac{1}{2^{\mathtt{t}+1}}\right) \text{ floating point multiplies and adds.}\end{aligned}$$

So the entire algorithm requires

$$2\alpha^2 n^3\left(1-\frac{1}{2^{\mathtt{t}+1}}\right) + \frac{4}{3}\alpha^2 n^2\left(1-\frac{1}{4^{\mathtt{t}}}\right)$$

which are not also performed by using option "I".

If we compare the two costs to determine when option "V" is more cost-effective than option "I", we see that "V" has a lower cost when

$$2\alpha^2 n^3\left(1-\frac{1}{2^{\mathtt{t}+1}}\right) + \frac{4}{3}\alpha^2 n^2\left(1-\frac{1}{4^{\mathtt{t}}}\right) < [1+\frac{2}{3}\alpha^2\left(1-\frac{1}{4^{\mathtt{t}}}\right)]n^3$$

for large n the n^2 term and the terms involving $\frac{1}{2^{t+1}}$ or $\frac{1}{4^t}$ are negligible, so we compare only the following terms resulting from the use of BLAS3:

$$\left(2\alpha^2 - \frac{2}{3}\alpha^2\right) n^3 < n^3$$

$$\frac{4}{3}\alpha^2 < 1$$

$$\alpha^2 < \frac{3}{4}$$

$$\alpha < \sqrt{\frac{3}{4}} \approx 0.866$$

We conclude that unless there is substantial deflation ($\delta > .14$) there is no appreciable gain in using option "V". There are problems which routinely get deflations as high as this (some as high as 90%!) and others which have a much lower deflation rate.

A sequence of examples follows which show the percentage of deflation occurring in each problem of a sequence of matrices whose eigensystems were solved using divide and conquer. Note every problem of a given matrix type and *all of its subproblems* are included in these graphs. In each case, the plot which follows will show the execution time of the divide and conquer code with `COMPQ` = "V" relative to the execution time of the divide and conquer code with `COMPQ` = "I".

The results are approximately as we predicted. However, it is clear from the plots that the assumptions of the model are not borne out—the fraction of values deflated in a subproblem is *not* constant throughout the whole problem, but seems to vary depending on how high up in the divide and conquer tree the subproblem occurs.

We never seem to save more than 10% of total execution time when using `COMPQ` = "V" and we lose up to 20% (depending on the matrix type). We would therefore recommend using `COMPQ` = "I" unless you are aware that your matrices have large clusters of eigenvalues.

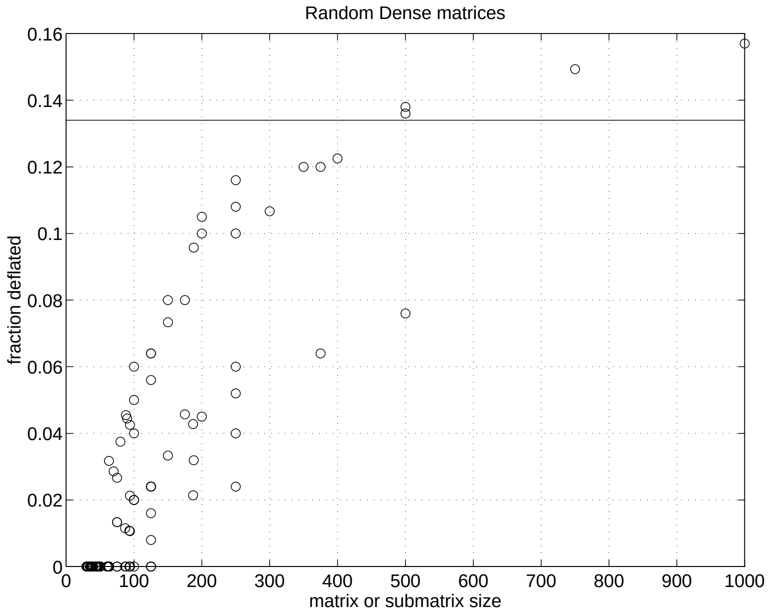


Figure 5.6.1: Deflation on random dense matrices

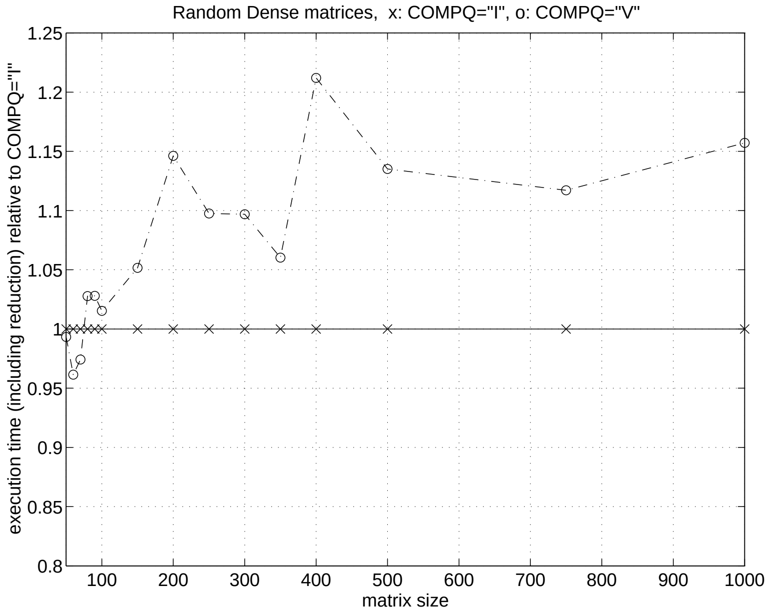


Figure 5.6.2: Option "V" relative to option "I" on random dense matrices using the DEC Alpha with Fortran BLAS

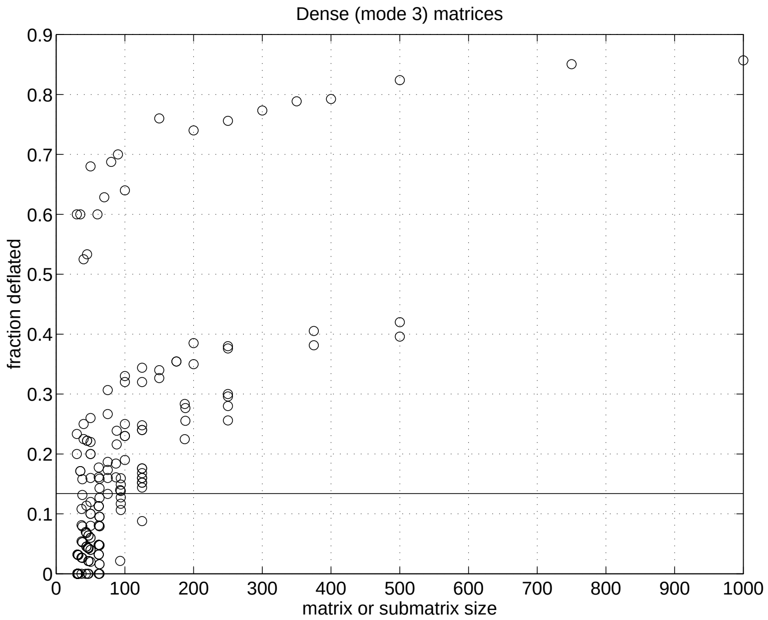


Figure 5.6.3: Deflation on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`)

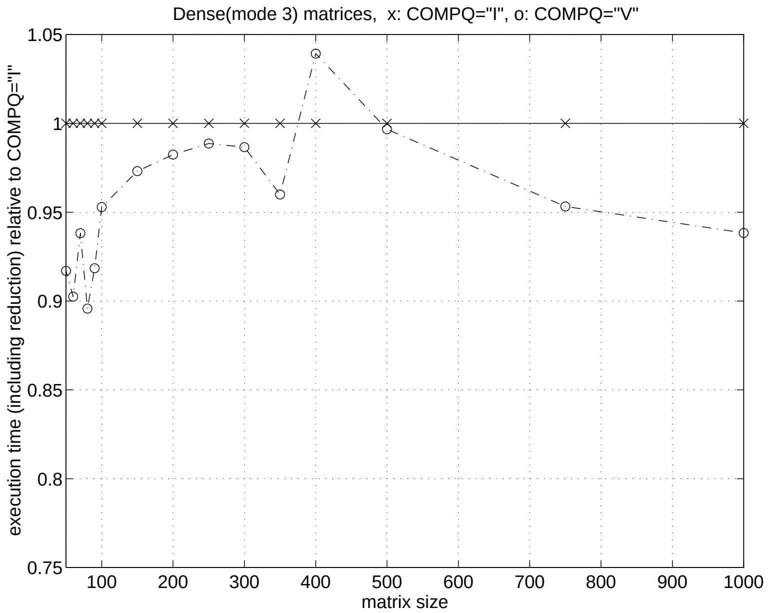


Figure 5.6.4: Option "V" relative to option "I" on dense matrices with geometrically distributed eigenvalues (`slatms, MODE=3`) using the DEC Alpha with Fortran BLAS

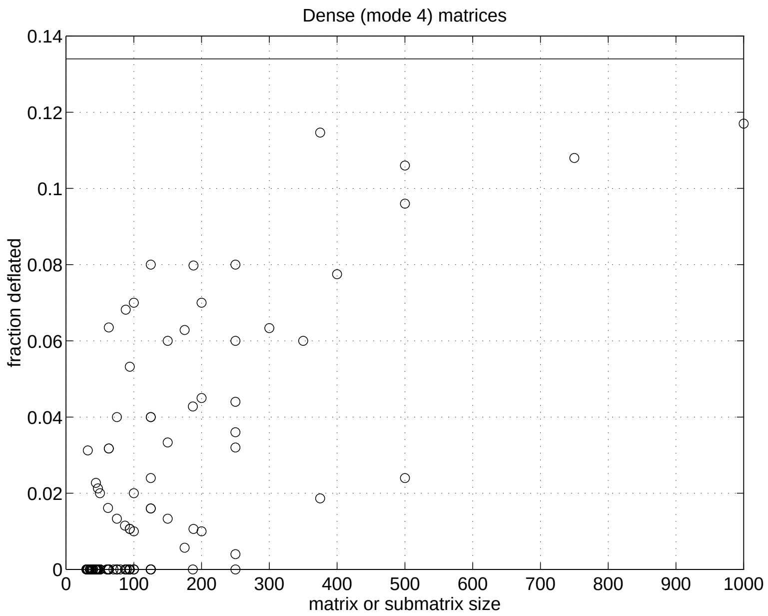


Figure 5.6.5: Deflation on dense matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`)

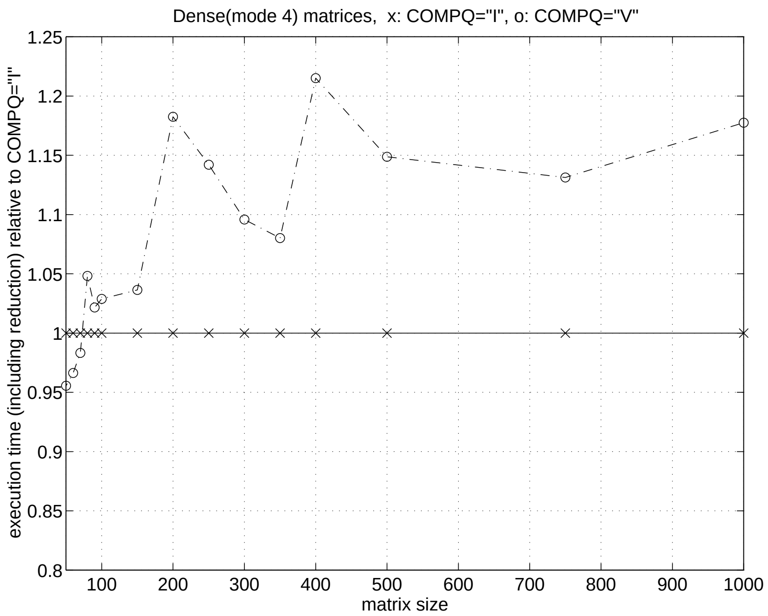


Figure 5.6.6: Option "V" relative to option "I" on dense matrices with arithmetically distributed eigenvalues (`slatms, MODE=4`) using the DEC Alpha with Fortran BLAS

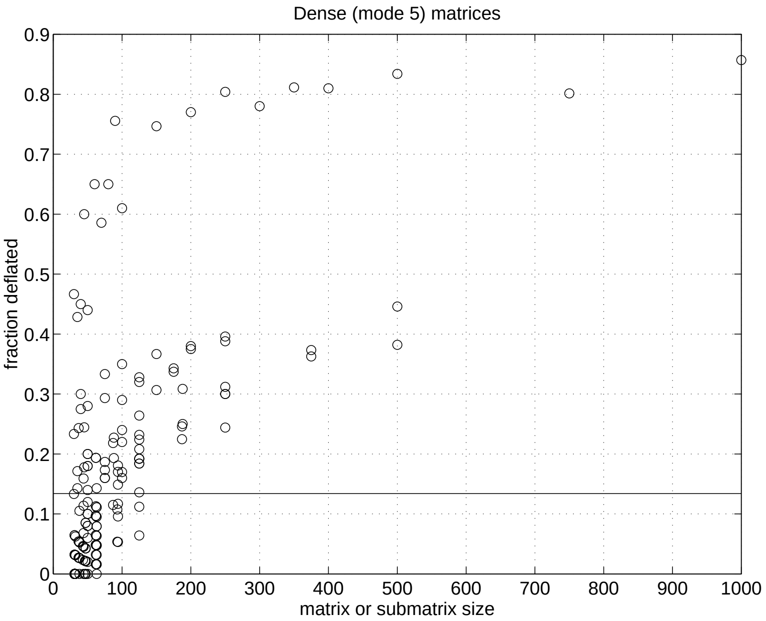


Figure 5.6.7: Deflation on dense matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`)

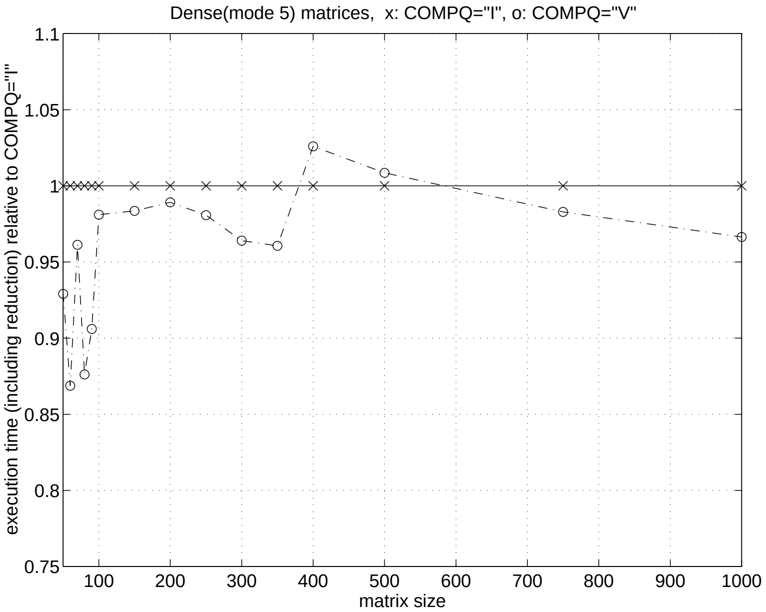


Figure 5.6.8: Option "V" relative to option "I" on dense matrices with random eigenvalues logarithmically distributed (`slatms, MODE=5`) using the DEC Alpha with Fortran BLAS

Chapter 6

Future Work

Ideas for extending the usefulness of the divide and conquer code include:

- **Computing a Subset of the Eigenvectors** A recent idea of Gu [9] suggests it is possible to compute some subset of the eigenvectors rather than all of them, by a modified version of the divide and conquer path which currently computes eigenvalues only. Briefly, we would save the intermediate orthogonal matrices, permutations, and Givens rotations, and then multiply them into k columns of the identity matrix, corresponding to the k eigenvectors we want. This would require $O(kn^2)$ floating point operations, with the constant heavily dependent on the amount of deflation. The main competing algorithm for this task is bisection/inverse iteration which uses $O(kn)$ flops if no reorthogonalization is required or $O(k^2n)$ if maximal reorthogonaliztion is required. We expect that this algorithm would only be faster than bisection/inverse iteration for certain kinds of matrices with a great deal of deflation, or when k is not much smaller than n.

- **Bidiagonal Singular Value Decomposition** An obvious algorithm for the bidiagonal SVD is to find the eigendecomposition of a symmetric tridiagonal matrix with zero diagonal and the bidiagonal matrix entries strung along the sub- and superdiagonals [8]. But this may fail to compute orthogonal singular vectors if the matrix is ill-conditioned. There is an alternative formulation of divide-and-conquer specialized to bidiagonal matrices which is discussed in [11] — one could adapt the the methods in the report to the basic algorithm in [11].

- **Parallelizing Divide and Conquer** There are several challenges to overcome to fully exploit the available parallelism in this algorithm. First, different nodes in the divide and conquer tree in Figure 3.0.1 may all be done independently. But assigning an equal number of processors to each node may lead to poor load balance if there are differing amounts of deflation in each node (it is easy to construct examples where this would happen). Second, as we move up the tree, there are fewer such independent nodes at each level, but there is still a great deal of independent work to do, say in solving for different roots of the secular equation. So a good implemenation would have to gradually shift parallel resources from working on independent nodes, to cooperating to solving the secular equations. Third, much of the time is spent in the data parallel operation of matrix-matrix multiplication, especially near the top of the tree. In the serial code, this is done by reorganizing data structure to permit calls to the the BLAS (see section 4.1, `slaed3`). On a parallel machine, this might require a great deal of expensive communication, which could overwhelm the floating point costs. Also, it require a gradual shifting of resources away from task parallelism near the bottom of the tree (independent nodes) to data parallelism (matrix multiplication). All these are challenging issues in parallel programming.

- **Updating and Downdating the Symmetric Eigenproblem and SVD** If one knows the eigendecomposition of a matrix $A = Q\Lambda Q^T$, then one sometimes wants to know the eigendecomposition of the rank-one perturbed matrix $A + \rho \mathbf{v}\mathbf{v}^{\mathbf{T}}$, where ρ is a scalar and $\mathbf{v}$ is a vector. Similarly, one would like to update (or downdate) the SVD of a general matrix $G = U\Sigma V^T$ when it is modified to $G + \mathbf{u}\mathbf{v}^T$. Both problem involve solving a single secular equation, and so the techniques of this report are applicable.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 1.0.* SIAM, Philadelphia, 1992. 235 pages.

[2] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[3] C. F. Borges and W. B. Gragg. A parallel divide and conquer algorithm for the generalized real symmetric definite tridiagonal eigenproblem. Working Paper, 1992.

[4] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[5] P. Deift, T. Nanda, and C. Tomei. ODEs and the symmetric eigenvalue problem. *SIAM J. Num. Anal.*, 20(1), 1983.

[6] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica, volume 2.* Cambridge University Press, 1993.

[7] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, March 1987.

[8] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[9] M. Gu. personal communication, 1994.

[10] M. Gu and S. Eisenstat. A stable algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, September 1992.

[11] E. Jessup and D. Sorensen. A divide and conquer algorithm for computing the singular value decomposition of a matrix. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66, Philadelphia, PA, 1989. SIAM.

[12] W. Kahan. personal communication, 1993.

[13] K. Li and T.-Y. Li. An algorithm for symmetric tridiagonal eigenproblems — divide and conquer with homotopy continuation. *SIAM J. Sci. Comp.*, 14(3), May 1993.

[14] R.-C. Li. Solving the secular equation stably and efficiently. UC Berkeley Math Dept. Report, in preparation, 1992.

[15] T.-Y. Li and N. H. Rhee. Homotopy algorithm for symmetric eigenvalue problems. *Num. Math.*, 55:265–280, 1989.

[16] T.-Y. Li and Z. Zeng. Laguerre's iteration in solving the symmetric tridiagonal eigenproblem - a revisit. Michigan State University preprint, 1992.

[17] T.-Y. Li, H. Zhang, and X.-H. Sun. Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 12(3):469–487, 1991.

[18] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[19] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Num. Anal.*, 28(6):1752–1775, 1991.

[20] H. Trotter. Eigenvalue distributions of large hermitian matrices: Wigner's semicircle law and a theorem of Kac, Murdock, Szegö. *Advances in Mathematics*, 54:67–82, 1984.

Appendix A

Calling Sequence for sstedc

```
      SUBROUTINE SSTEDC( COMPQ, N, D, E, Q, LDQ, IWORK, WORK, INFO )
*
*     Purpose
*     =======
*
*     SSTEDC computes all eigenvalues and, optionally, eigenvectors of a
*     symmetric tridiagonal matrix using the divide and conquer method.
*     The eigenvectors of a full or band symmetric matrix can also be found
*     if SSYTRD or SSPTRD or SSBTRD has been used to reduce this matrix to
*     tridiagonal form.
*
*     This code makes very mild assumptions about floating point
*     arithmetic.  It will work on machines with a guard digit in
*     add/subtract, or on those binary machines without guard digits
*     which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.
*     It could conceivably fail on hexadecimal or decimal machines
*     without guard digits, but we know of none.  See SLAED3 for details.
*
*     The code currently calls SSTERF if eigenvalues only are desired, since
*     this is faster, but it can be easily modified to use divide and conquer
*     (see the comments below).
*
```

```
*     Arguments
*     =========
*
*     COMPQ (input) CHARACTER*1
*           Specifies whether eigenvectors are to be computed
*           as follows
*
*           COMPQ = 'N' or 'n' Compute eigenvalues only.
*
*           COMPQ = 'I' or 'i' Compute eigenvectors of
*                              tridiagonal matrix also.
*
*           COMPQ = 'V' or 'v' Compute eigenvectors of original
*                              dense symmetric matrix also.
*                              On input, Q contains the orthogonal
*                              matrix used to reduce the original
*                              matrix to tridiagonal form.
*
*     N     (input) INTEGER
*           The dimension of the symmetric tridiagonal matrix.  N >= 0.
*
*     D     (input/output) REAL array, dimension(N)
*           On entry D contains the main diagonal of the tridiagonal
*                    matrix.
*           On exit D, if INFO = 0, contains its eigenvalues.
*
*     E     (input) REAL array, dimension(N-1)
*           Contains the subdiagonal entries of the tridiagonal matrix.
*           On exit, E has been destroyed.
*
*     Q     (input/output) REAL array, dimension(LDQ,N)
*           If COMPQ = 'V' or 'v', then:
*              On entry, Q contains the orthogonal matrix used in the
*              reduction to tridiagonal form.
```

```
*          If COMPQ = 'V' or 'v' or 'I' or 'i', then:
*             On exit, if INFO = 0, Q contains the orthonormal
*             eigenvectors of the symmetric tridiagonal (or full) matrix.
*          If COMPQ = 'N' or 'n', then Q is not referenced.
*
*    LDQ   (input) INTEGER
*          The leading dimension of the array Q. If eigenvectors are
*          desired, then LDQ >= max( 1, N ). In any case, LDQ >= 1.
*
*    IWORK (workspace) INTEGER array
*          If COMPQ = 'N' then no integer workspace is required.
*          If COMPQ = 'V' then the dimension of IWORK must be at least
*                         ( 6 + 6*N + 5 * N * lg N )
*                         ( lg( N ) = ceiling( log-base-2 ( N ) ) )
*          If COMPQ = 'I' then the dimension of IWORK must be at least
*                         ( 2 + 5 * N ).
*
*    WORK  (workspace) REAL array,
*          If COMPQ = 'N' then no workspace is required.
*          If COMPQ = 'V' then the dimension of WORK must be at least
*                         ( 1 + 3 * N + 2 * N * lg N + 3 * N**2 ).
*          If COMPQ = 'I' then the dimension of WORK must be at least
*                         ( 1 + 3 * N + 2 * N * lg N + 2 * N**2 ).
*
*    INFO  (output) INTEGER
*          = 0:  successful exit.
*          < 0:  if INFO = -i, the i-th argument had an illegal value.
*          > 0:  The algorithm failed to compute an eigenvalue while
*                working on the submatrix lying in rows and columns
*                INFO/(N+1) through MOD(INFO,N+1).
*
* =====================================================================
*
*    .. Parameters ..
```

```
INTEGER      SMLSIZ
PARAMETER    ( SMLSIZ = 25 )
```

Appendix B

History of the Code

The code began as a set of four routines from Tang and Sorenson (`dlasud`, `dlaevd`, `dlaevu`, and `dlaacc`) taken from a larger package meant to implement divide and conquer in a way suitable for multi-processor machines. These routines were cleaned up, debugged, and made into a part of a fully working code.

This is a basic description of the changes which have taken place since the incorporation of the Tang & Sorenson code into this working code.

- **Version 2** incorporated changes in the routine which solves for the roots of the secular equation [14]. Instead of always using the left-hand assymptote as the origin in the frame of reference used to solve for a particular root, the right-hand assymptote is used when appropriate. It is appropriate to use the right-hand assymptote as the origin if the value of the secular function at the mid-point of the interval formed by the two assymptotes is negative. Since $\rho > 0$, then this selection of origins assures that the covengence of the iterative solution will be monotonic. Li also removed several instructions which, after analysis, had been determined to serve no useful purpose. The initial guess was also modified to be in the proper sub-interval of $(d(i), (d(i) + d(i+1))/2)$ and $((d(i) + d(i+1))/2, d(i+1))$.

- **Version 3** Convergence is slow when a root is near the left pole. A slight change to the secular equation root-finder was made to speed convergence in this case.

- **Version 4** This is the first version to make use of the trick developed in [10] which enables us to solve this problem without the use of extended precision.

- **Version 5** A minor modification is made to the previous routines to optimize for storage space required. More meaningfully, the matrix multiply at the end of each step was optimized to take advantage of the structure of the matrices involved:

  $$\begin{bmatrix} 1 & & 3 & 4 \\ & 2 & 3 & 4 \end{bmatrix} * [Q]$$

  This means that a permutation is applied to the columns of the previous eigenvector matrix (as modified by deflation) and the same permutation to the rows of the new eigenvector matrix.

- **Version 6** Purposeful conversion to single precision. Other changes are largely cosmetic: loops for copying vectors or matrices were replaced by calls to existing routines, loops for assigning values to matrices were replaced by calls to existing routines, the implementation from [10] was altered slightly to make it more easily vectorized.

- **Version 7** The method of division into subproblems was altered. Previously a rather ad-hoc method was used which divided the problem into as many equal-sized pieces as possible, the remainder separated as "extra". The recombinations required were less than optimal. The new method requires storage of an additional $\lceil lg(n) \rceil$ integer values to save the sizes of the submatrices on the lowest level.

  The new sorting method divides all of the existing subproblems into problems of size $\lfloor \frac{m}{2} \rfloor$ and $\lceil \frac{m}{2} \rceil$ until the largest existing subproblem is of size `SMLSIZ` or less. `SMLSIZ` is a parameter within the program routines `sstedc` and `slaed0`.

- **Version 8** Since the integer workspace is not fully used until the top level of recombinations is reached, part of it can be used to store the $\lceil lg(n) \rceil$ submatrix sizes. In this fashion we need only two integer storage spaces in addition to the integer storage required normally.

  Also in this version, the sorting of values from the diagonals of two previous submatrices is done by a simple merge rather than the bubble sort previously implemented. Since the diagonal elements are sorted at every step, this is a very rapid way to sort.

- **Version 9** This version has the argument orders in several levels cleaned up considerably. It also sorts the eigenvalues after the final step so that they come out in sorted order at the very end – provided that there was no splitting within `sstedc`.

- **Version 10** This version incorporates some revisions to the secular equation solver (first introduction of the routine `slaed6`).

- **Version 11** This version incorporates the eigenvalues only routine. It stores all of the smaller sub-matrices in a large workspace and recalls them for a series of BLAS2 operations to calculate the z-vector needed at the current level.

  The storage scheme uses $n\lceil lg(\frac{n}{SMLSIZ})\rceil$ additional integer storage for the permutations associated with each subproblem in the divide and conquer tree. An array of n integers is used to keep track of where the permutations assciated with a subproblem are located within this storage space.

  Similarly, a maximum of $2n\lceil lg(\frac{n}{SMLSIZ})\rceil$ integer and real storage is required to keep track of the Givens rotations used to deflate a subproblem. Note that this deflation will reduce the size of the subproblem's eigenvector matrix; if done cleverly, the storage saved from storing the eigenvectors could be used to store the real values involved in the Givens rotations. An array of n integers is used to keep track of where the rotations assciated with a subproblem are located within this storage space.

  Finally, n^2 real storage space is required for the eigenvector submatrices found for each subproblem. Since no more z-vectors will be computed, the final $n \times n$ eigenvector matrix is written over the previously computed eigenvector matrices. As above, n integers are used to keep track of where the eigenvectors assciated with a subproblem are located within this storage space.

  In the case of computing the eigensystem of a dense symmetric matrix, the rotations, permutations, and multiplication by a subproblem's eigenvectors are all incorporated directly into the orthogonal matrix used to reduce the dense matrix into a tridiagonal one.