A Comparative Analysis of LAPACK Routines for Solving Eigenproblems on Real Symmetric Tridiagonal Matrices

Yao Xiao
yaoxiao@g.harvard.edu

Declaration

Textbook chapter. The chapter of Heath's textbook [6] that this work is closest to is *Chapter 4, Eigenvalue Problems*, and in particular *Chapter 4.5, Computing Eigenvalues and Eigenvectors*. The most relevant sections are §4.5.3, §4.5.6, §4.5.9, §4.5.10, and §4.5.11.

Contributions of the authors. I declare that this work was composed entirely by myself, except where it is acknowledged through reference or citation.

Use of AI tools. GitHub Copilot is used for code completion. ChatGPT is used for learning Fortran syntaxes and for reading certain parts of the LAPACK documentation [1].

Other sources. There are no other sources involved in this work, except where it is acknowledged through reference or citation.

1 Introduction

Eigenproblems on symmetric tridiagonal matrices are a cornerstone of numerical linear algebra, with applications spanning physics, engineering, and data science. LAPACK [1], a widely adopted library for linear algebra computations, offers multiple routines for solving these eigenproblems, each leveraging distinct algorithms with varying computational complexities and performance characteristics. This report provides a comparative analysis of the LAPACK routines `STEV`, `STEVD`, `STEVX`, and `STEVR`. Through synthetic and real-world matrices, we evaluate their performance and accuracy under different scenarios.

Organization. In §2 we will go over the routines and their underlying algorithms, and briefly analyze their theoretical computational complexities. In §3 we will discuss the testing environment, testing matrices, and evaluation metrics used in this report. §4 and §5 will demonstrate the evaluation results on performance and accuracy, respectively, of the routines. The report will be concluded in §6.

2 Algorithms Overview

In this section, we breifly will go over the algorithms used by the four LAPACK routines, `STEV`, `STEVD`, `STEVX`, and `STEVR`, for solving eigenproblems on symmetric tridiagonal matrices T of size $n \times n$.

Routine	Algorithm	Complexity	Heath [6]
`STEV` (§2.1)	QR Iteration	$O(n^3)$	§4.5.6
`STEVD` (§2.2)	Divide & Conquer	$O(n^3)$, better when clustered	§4.5.10
`STEVX` (§2.3)	Bisection	$O(n^3)$, $O(n^2)$ when isolated	§4.5.3, §4.5.9
`STEVR` (§2.4)	MRRR	$O(n^2)$	§4.5.11

Table 1: The four LAPACK routines for solving symmetric tridiagonal eigenproblems and their underlying algorithms. Note that the computational complexity here is theoretical and considering solving the full eigenproblem, i.e., solving for all eigenvalues and eigenvectors.

2.1 `STEV`: QR Iteration

The `STEV` routine utilizes the QR iteration algorithm with shifts. At each step k, the algorithm performs QR decomposition on the shifted matrix, such that

$$T_k - \sigma_k I = Q_k R_k, \tag{1}$$

where Q_k is orthogonal, R_k is upper triangular, the σ_k is the shift (typically) chosen as the bottom right corner of T_k (some eigenvalue of a submatrix). The matrix T_k is then updated as

$$T_{k+1} = R_k Q_k + \sigma_k I. \tag{2}$$

This is essentially a similarity transform as eigenvalues are preserved. The tridiagonal form is also preserved across iterations. The symmetry ensures that off-diagonal elements decrease over iterations until negligible so that T_k converges to a diagonal matrix where the diagonal entries are the eigenvalues of T. The implicit use of shifts accelerates this convergence by ensuring more rapid convergence of the largest eigenvalues.

During the QR iteration, the eigenvectors can be accumulated by successively applying the matrices Q_k at each step k to the matrix of basis vectors (identity matrix of standard basis). This will finally result in an orthogonal matrix V, whose columns are the normalized eigenvectors corresponding to the eigenvalues of T. QR iteration is known to be an $O(n^3)$ algorithm.

2.2 STEVD: Divide and Conquer

The STEVD routine implements the divide-and-conquer method [7]. The eigenproblem on T is recursively split into eigenproblems on smaller submatrices of T, with their eigenvalues and eigenvectors computed independently and merged together (conquer). The split will be until the submatrix size drops under a fixed threshold. In particular, each split looks like

$$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \beta \mathbf{u}\mathbf{u}^\top, \tag{3}$$

where T_1 and T_2 are the tridiagonal submatrices to be processed (if under threshold) or split further (otherwise), and the second term is a rank-1 correlation with $\mathbf{u}$ being all 0 except for the two terms in its middle being 1. Consider that the recursive splits form a binary tree, and we find the eigenvalues and eigenvectors of the leaves with some direct method, then the conquer step will merge the eigenvalues (and eigenvectors) in the leaves bottom-up the tree. Let $T_1 = Q_1\Lambda_1 Q_1^\top$ and $T_2 = Q_2\Lambda_2 Q_2^\top$ with Q_1 and Q_2 orthogonal. Plugging into (3), we have that

$$Q\Lambda Q^\top = D + \beta \mathbf{v}\mathbf{v}^\top, \tag{4}$$

where Q and Λ are Q_1, Q_2 and Λ_1, Λ_2 combined in the way T is obtained from T_1, T_2, and $\mathbf{v}$ consists of the last row of Q_1 and the first row of Q_2. The conquer of eigenvalues would essentially be solving the secular equation

$$1 + \beta \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda} = 0. \tag{5}$$

The merged eigenvector corresponding to the merged eigenvalue λ would then be given by $(D - \lambda I)^{-1}\mathbf{v}$ normalized. In practice, the STEVD routine alternatively performs a scaling on the eigenvectors of T_1 and T_2 first, then multiplies by the eigenvector matrix accumulated bottom-up to obtain the merged eigenvector. The cost for obtaining the eigenvalues and eigenvectors of T is theoretically $O(n^3)$. However, deflations can occur when the eigenvalue and eigenvector computed from a smaller eigenproblem can be directly used for a larger eigenproblem, which saves computation. Deflations can happen frequently for symmetric tridiagonal matrices, depending on many factors such as the distribution of eigenvalues. In the worst case when no delation happens, this is still $O(n^3)$, but is much better than $O(n^3)$ especially when eigenvalues of T are clustered and eigenvectors are sparse.

2.3 STEVX: Bisection and Inverse Iteration

The STEVX routine employs the bisection method that can find all or a subset of eigenvalues and eigenvectors [2]. This method is based on an inexpensive recurrence relation to count the number of eigenvalues below a fixed threshold μ, enabled by the tridiagonal nature of T. In particular, let $p_k(\mu)$ be the characteristic polynomial of submatrix of first k rows and k columns of T, then

$$p_{k+1}(\mu) = (\mu - T_{j+1,j+1})p_k(\mu) - T_{j+1,j}^2 p_{k-1}(\mu). \tag{6}$$

The number of eigenvalues less than μ would then be equal to the number of consecutive terms with the same sign in the sequence $\{p_k(\mu)\}$. With this observation, by computing the initial spectral bounds of T (for instance by the Gershgorin circle theorem), the intervals that contain at least one eigenvalue of T can be successively halved until shrinking below the desired precision. Note that (6) is prone to overflow so the recurrence relation used in practice is on $p_k(\mu)/p_{k-1}(\mu)$ instead of directly on $p_k(\mu)$. With the bisection method, each eigenvalue can be found with $O(n)$ additions and divisions. Hence finding a subset of m eigenvalues would be $O(mn)$ and finding all eigenvalues would be $O(n^2)$ operations.

After eigenvalues are obtained, the corresponding eigenvectors can be computed by inverse iteration. In particular, for each eigenvalue λ_i, per step k it solves

$$(T-\lambda_i I)\mathbf{x}^{(k+1)} = \tau_k \mathbf{v}^{(k)}, \tag{7}$$

starting from some approximation $\mathbf{v}^{(0)}$ of the i-th eigenvector and τ_k's are scalars. The inverse iteration works best when the eigenvalues are sufficiently isolated, which results in $O(mn)$ operations for m eigenvectors. If the eigenvalues are clustered together, then Gram-Schmidt orthogonalization is needed as a remedy to retain the orthogonality on the approximated eigenvectors, which increases the number of operations to $O(m^2n)$ (and $O(n^3)$ in the worst case if all eigenvalues are relatively close to each other). If finding all eigenvalues and eigenvectors, this is $O(n^2)$ for well-separated eigenvalues and $O(n^3)$ otherwise.

2.4 `STEVR`: MRRR Algorithm

The `STEVR` routine employs the Multiple Relatively Robust Representations (MRRR) algorithm [3, 4]. Like `STEVX`, it supports computing only a subset of eigenvalues and eigenvectors, say m. The critical component of this algorithm is the Relatively Robust Representation (RRR), such that

$$L_p D_p L_p^\top - \sigma I = LDL^\top, \tag{8}$$

where L is unit lower bidiagonal, D is diagonal, and σ is a proper shift near some cluster of eigenvalues. The eigenvalues of $L_p D_p L_p^\top$ should be computed to high relative accuracy, using techniques like bisection of DQDS (differential quotient with shifts). These eigenvalues will be grouped into clusters (or isolated if an eigenvalue is not close to any of its neighbors). For each isolated eigenvalue, an approximate eigenvector can be formed based on some formula on top of the DQDS transform which we will not specify here. For each cluster, compute a partial RRR based on the DQDS transform, shifting by some eigenvalue in the cluster. The eigenvalues in this cluster will be updated by the partial RRR, and they will go through the grouping process again. This recursively refines the partial RRR until eigenvalues become sufficiently isolated so that a final RRR can be obtained. Note that for the shifting when dealing with clusters, the algorithm can avoid traversing all eigenvalues in the cluster by early breaking when reaching some tolerance, which in practice can save much complexity.

The MRRR algorithm costs $O(n)$ operations per requested eigenvalue and eigenvector, so that the total cost is $O(mn)$ and $O(n^2)$ when requesting all eigenvalues and eigenvectors. Moreover, unlike `STEVX`, numerically orthogonal eigenvectors can be obtained in $O(n^2)$ even when eigenvalues are clustered, without relying on Gram-Schmidt orthogonalization to remedy the results. Note that this is the only theoretically $O(n^2)$ algorithm among `STEV`, `STEVD`, `STEVR`, and `STEVX` to solve the full eigenproblem on a symmetric tridiagonal matrix.

3 Testing Setup

In this section, we will give a brief overview of the testing setup, including test environment (§3.1), test matrices (§3.2), and evaluation metrics (§3.3).

3.1 Environment

All tests will be performed on a single machine with the Intel Core Ultra 7 155H CPU, which has 22 virtual cores and 24MB L3 cache. The operating system is Windows WSL2.

The LAPACK routines used in the testing are their Fortran versions, wrapped by `scipy` with `f2py` under the `scipy.linalg.lapack` module. These are very thin wrappers that perform only bare metal checks like array alignment, which reflects the actual performance of the LAPACK routines effectively. `scipy.linalg.lapack`, however, do not contain wrappers of all LAPACK routines (in particular, it does not contain `STEVD`, `STEVR`, and `STEVX` wrappers as of version 1.15.0), so we implement our own wrappers within its framework (Appendix A). We use the double precision versions of the routines, i.e., the ones prefixed `D`.

The Fortran compiler is GCC Fortran 13.3.0 with O2 level optimization. The underlying BLAS library is OpenBLAS 0.3.28 on the Prescott architecture, with OpenMP disabled.

3.2 Test Matrices

We include two classes of test matrices. The first class consists of symmetric tridiagonal matrices that are synthetically generated based on different eigenvalue distributions and some special matrices that are notoriously famous for failing certain numerical algorithms. The second consists of symmetric tridiagonal matrices from real-world applications, collected from public datasets.

Synthetic matrices. We include the following synthetic matrices in the test suite [4]. The matrix sizes range from 100 to 2000. The implementations are included in Appendix B.1.

- **uniform_eps**: The first $n-1$ eigenvalues are uniformly distributed from ε to $(n-1)\varepsilon$, and the n-th eigenvalue is 1.
- **uniform_sqeps**: The first eigenvalue is ε, the second to the $(n-1)$-th eigenvalues are uniformly distributed from $1+\sqrt{\varepsilon}$ to $1+(n-2)\sqrt{\varepsilon}$, and the n-th eigenvalue is 2.
- **uniform_equi**: The eigenvalues are uniformly distributed from ε to 1.
- **geometric**: The eigenvalues are geometrically distributed between ε and 1.
- **normal**: The eigenvalues are sampled from a standard normal distribution.
- **clustered_one**: The first eigenvalue is ε, and the rest of the eigenvalues are all around 1 with random perturbations at ε-scale.
- **clustered_pmone**: The first eigenvalue is ε, and the rest of the eigenvalues are all around either ± 1 with random perturbations at ε-scale.
- **clustered_machep**: The first $n-1$ eigenvalues are all around ε with random perturbations at ε-scale, and the n-th eigenvalue is 1.

- **clustered_pmmachep**: The first $n-1$ eigenvalues are all around either $\pm\varepsilon$ with random perturbations at ε-scale, and the n-th eigenvalue is 1.
- **wilkinson**: Wilkinson matrices, such that diagonal entries are 1 to n and off-diagonal entries are 1. Wilkinson matrices are known to have highly clustered by unequal eigenvalues.

Real-world matrices. For real-world matrices, we take the BCSSTK01 to BCSSTK33 matrices in the BCS structural engineering matrices dataset (excluding too large matrices $n > 5000$), from the Harwell-Boeing collection [5]. These are large sparse real symmetric matrices for real-world eigenproblems coming from a variety of applications. We will use LAPACK's `SYTRD` routine to generate their tridiagonal forms for testing.

3.3 Evaluation Metrics

Except for the trivial metric of time taken to execute the routine which measures the performance, we include two metrics, loss of orthogonality and maximum residual norm, for evaluating the accuracy of the results. The implementations are included in Appendix B.2. Let T be the original real symmetric tridiagonal matrix of size n, $\lambda_1, \dots, \lambda_n$ be the computed eigenvalues, and $\mathbf{z}_1, \dots, \mathbf{z}_n$ be the corresponding computed eigenvectors.

Loss of orthogonality. The loss of orthogonality metric measures the deviation of the computed eigenvectors from being perfectly orthogonal, defined as

$$\text{Loss of orthogonality} = \max_{i \neq j} \frac{|\mathbf{z}_i^\top \mathbf{z}_j|}{n\varepsilon}, \tag{9}$$

where ε is the machine epsilon and the term $n\varepsilon$ is for normalization of the metric. This essentially captures the largest off-diagonal entry of the Gram matrix of the eigenvectors. In theoretical settings, the eigenvectors of symmetric matrices are expected to form an orthonormal basis, but due to numerical errors and finite-precision arithmetic, the computed eigenvectors may not perfectly adhere this property. The larger this metric, the worse orthogonality is maintained in the computed eigenvectors.

Maximum residual norm. The maximum residual norm metric evaluates the quality of the computed eigenpairs by measuring how closely they satisfy the eigenvalue equation $T\mathbf{z}_i = \lambda_i \mathbf{z}_i$, such that

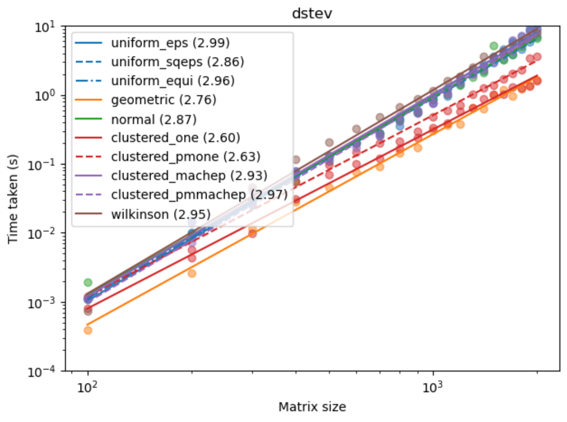
$$\text{Maximum residual norm} = \max_{i} \frac{\|T\mathbf{z}_i - \lambda_i \mathbf{z}_i\|}{n\varepsilon\|T\|}, \tag{10}$$

where the term $n\varepsilon$ is again for normalizing the metric. A large maximum residual norm means that some eigenpair is highly inaccurate, highlighting potential numerical instability in the algorithm.

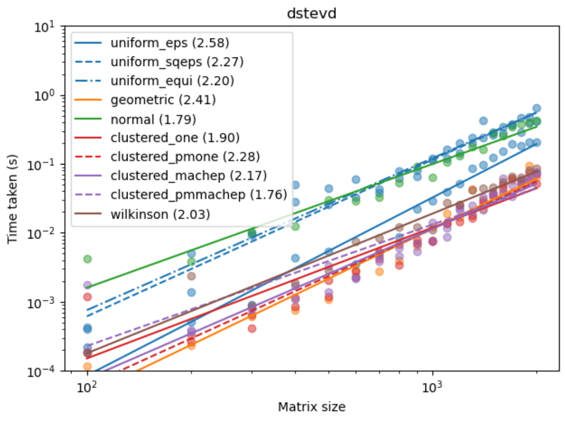
4 Performance

In this section, we will compare the performance of the different LAPACK routines for solving eigenproblems on different real symmetric tridiagonal matrices. We will compare the performances on both synthetic matrices (§4.1) and on real-world matrices (§4.2).

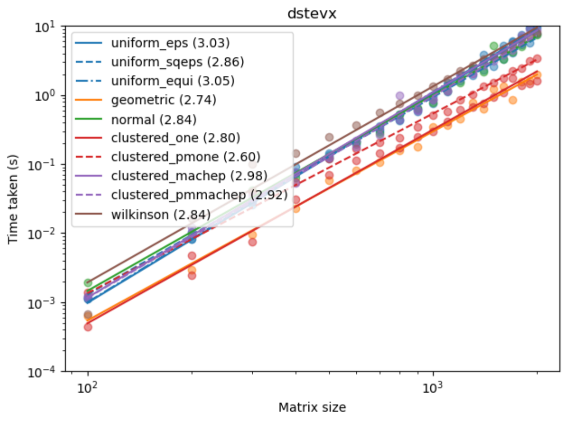
4.1 Synthetic Matrices



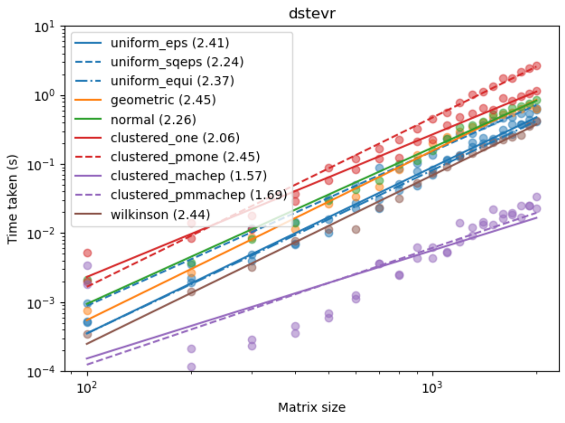
(a) `STEV`: QR iteration



(b) `STEVD`: Divide and conquer



(c) `STEVX`: Bisection and inverse iteration



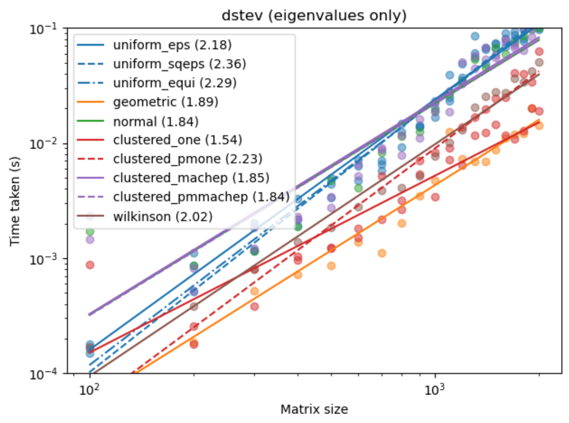
(d) `STEVR`: MRRR algorithm

Figure 1: Performance of the routines on different synthetic matrices, solving for both eigenvalues and eigenvectors. The points are the actual data and the lines are the least-squares fits in log-log scale. The numbers in the legend are the slopes of the fitted lines.

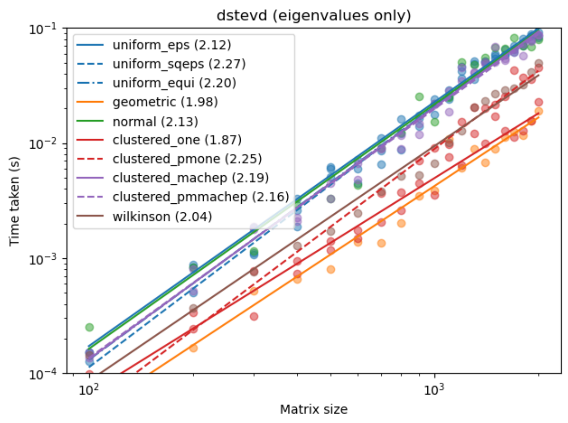
Figure 1 shows the performance of the four LAPACK routines on different synthetic matrices, solving for both eigenvalues and eigenvectors. The y-axis of the four subfigures are aligned, from which we can clearly see that `STEV` and `STEVX` are significantly slower than `STEVD` and `STEVR`. `STEV` (QR iteration) is indeed the theoretical slowest, having a computational complexity of $O(n^3)$. `STEVX` (bisection) needs the inverse iteration process to compute the eigenvectors, which also makes it $O(n^3)$ as long as eigenvalues are not sufficiently isolated. `STEVD` (divide and conquer) is theoretically $O(n^3)$ must benefits from frequent deflations in most scenarios, thus performing much better than the previous two. `STEVR` (MRRR) is a theoretically $O(n^2)$ algorithm so its good performance is also reasonable.

From the slopes, we can see that `STEV` and `STEVX` are less sensitive to different distributions of eigenvalues, almost always near 3 (and indeed they are $O(n^3)$ algorithms). In constrast, `STEVD` and `STEVR` are more sensitive to the distributions of eigenvalues. Indeed, the performance of `STEVD` (divide and conquer) largely depends on how many deflations occur during the process, which is in turn dependent on the distributions of eigenvalues. `STEVR` seems to perform particularly well on clustered_machep and clustered_pmmachep matrices, possibly because it is easy to reach the tolerance when refining the partial RRRs, saving a lot of computations. Both

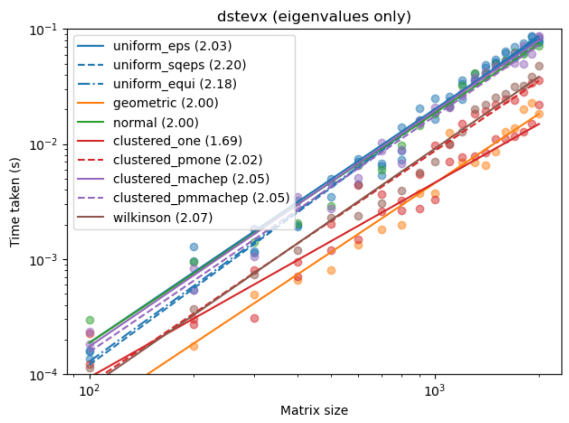
STEVD and STEVR achieves slopes of around 2 (meaning $O(n^2)$ complexity) at least for some types of matrices.



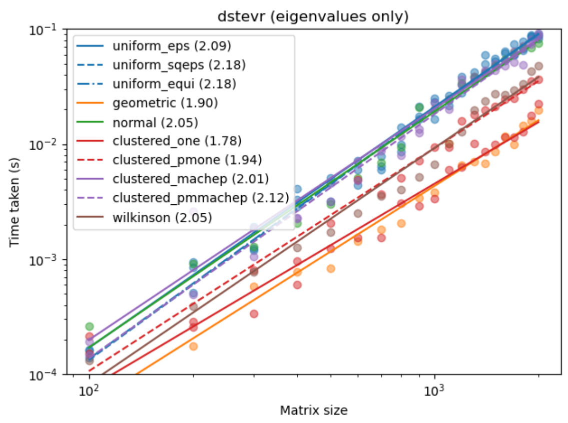
(a) STEV: QR iteration



(b) STEVD: Divide and conquer



(c) STEVX: Bisection and inverse iteration

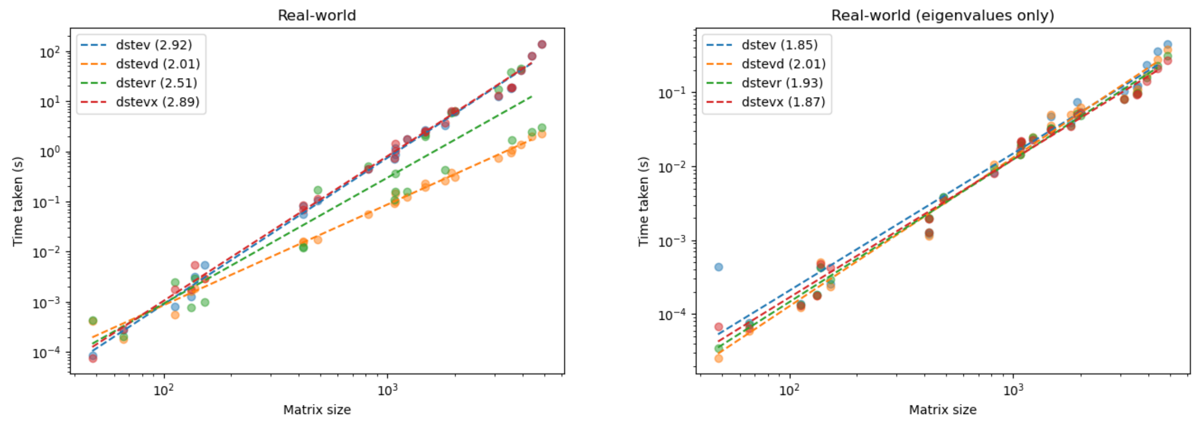


(d) STEVR: MRRR algorithm

Figure 2: Performance of the routines on different synthetic matrices, solving for only eigenvalues (not eigenvectors). The points are the actual data and the lines are the least-squares fits in log-log scale. The numbers in the legend are the slopes of the fitted lines.

Figure 2, different from Figure 1, show the performance comparison when solving for only eigenvalues. The results become very different, particularly for STEV and STEVX. First of all, note that the y-axis has changed from range $[0.0001, 10]$ to $[0.0001, 0.1]$, and we can see that all four routines can finish much faster than when eigenvectors are also requested. Moreover, the performance of STEV and STEVX are now comparable to that of STEVD and STEVR, all having slopes near 2, meaning that they achieve near $O(n^2)$ performance. We particularly focus on the former two. For STEV (QR iteration), we note that square root operations can be tens of times more expensive than additions and multiplications. It has been shown that when computing only for eigenvalues, the $O(n^2)$ square root operations involved in the QR iteration can be mitigated with $O(n^2)$ additions and multiplications with larger constant, but the square root operations are not avoidable when eigenvectors also need to be computed. Thus STEV is around $O(n^3)$ when solving for both eigenvalues and eigenvectors but improves to around $O(n^2)$ when only eigenvalues are needed. For STEVX (bisection), the $O(n^3)$ complexity comes from the inverse iteration process for computing eigenvectors when eigenvalues are not sufficiently isolated, while the computation of all eigenvalues only is indeed $O(n^2)$ in theory.

4.2 Real-World Matrices



(a) Eigenvalues and eigenvectors

(b) Eigenvalues only

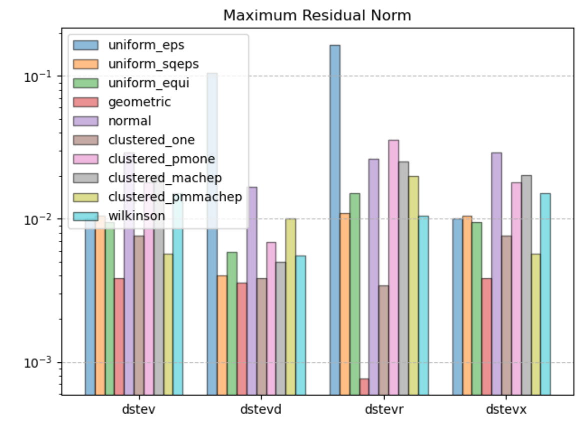
Figure 3: Performance of the routines on the collection of real-world matrices. Each data point is the size of a real-world matrix mapping to computation time. The lines are the least-squares fits in log-log scale. The numbers in the legend are the slopes of the fitted lines. The different characteristics of the matrices are not distinguishable in the plot.

Figure 3 shows the performance evaluation results on real-world matrices, with Figure 3a computing both eigenvalues and eigenvectors and Figure 3b computing only eigenvalues. The results are similar to those on synthetic matrices. From Figure 3a we can see that when computing both eigenvalues and eigenvectors, `STEV` (QR iteration) and `STEVX` (bisection) are approximately $O(n^3)$ as expected. `STEVD` (divide and conquer) performs the best at $O(n^2)$ complexity. `STEVR` (MRRR) is however worse than its theoretical $O(n^2)$ complexity, being approximately $O(n^{2.5})$ on the chosen collection of real-world matrices. This is likely related to the characteristics of the matrices, since we have also seen complexities worse than $O(n^2)$ for `STEVR` in Figure 1d, for instance on Wilkinson matrices. Now from Figure 3b, when computing only eigenvalues, similar to previous results on synthetic matrices, all routines achieve comparable performance at approximately $O(n^2)$. We see that `STEV` and `STEVX`, which perform much worse than `STEVD` and `STEVR` in most test cases, are achieving better performance on real-world matrices.

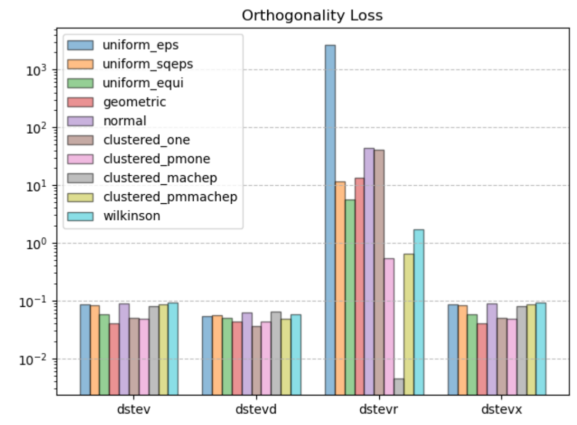
5 Accuracy

In this section, we will report the worst-case maximum residual norm and orthogonality loss metrics of the routines on different synthetic matrices.

From Figure 4a, there is no clear pattern based on the routines and types of matrices, but in general `STEVR` (MRRR) performs slightly worse then the other three routines. This pattern is much clearer in Figure 4b, where `STEVR` has significantly worse orthogonality loss than the other three routines. Though the plot shows only the worst-case scenario, even the best-case scenario follows this pattern. This however makes sense because the MRRR algorithm achieves its $O(n^2)$ computational complexity primarily by mitigating the need of the expensive reorthogonalization via Gram-Schmidt process, and this is at the cost of orthogonality accuracy. The same can be seen in Figure 5 on real-world matrices.



(a) Worst-case maximum residual norm



(b) Worst-case orthogonality loss

Figure 4: Accuracy metrics of different routines on different synthetic matrices.

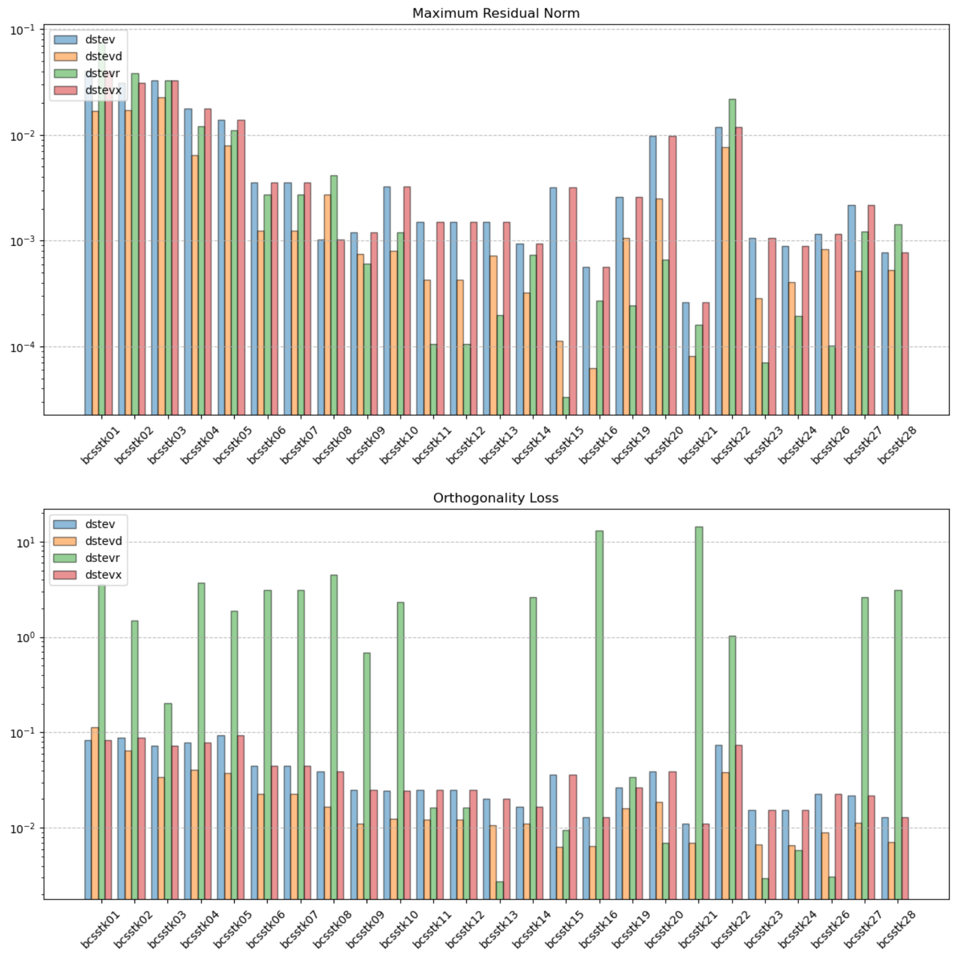


Figure 5: Accuracy metrics of different routines on the collection of real-world matrices.

6 Conclusion

This report has presented a comprehensive evaluation of four LAPACK routines — `STEV`, `STEVD`, `STEVX`, and `STEVR` — for solving eigenproblems on real symmetric tridiagonal matrices. The findings reveal significant differences in performance and accuracy, influenced by algorithmic design and matrix properties. STEVD and STEVR are in general much faster than their counterparts, particularly for large matrices. STEVD is, in many cases, even faster than STEVR despite being a theoretical $O(n^3)$ algorithm, thanks to the large number of deflations in practice on tridiagonal matrices. STEVR can also suffer from inaccuracies, particularly loss of orthgonality in the eigenvectors. Yet though not discussed in detail in this report, STEVR supports finding a subset of eigenvalues and eigenvectors, which STEVD is unable to. As for STEV and STEVX, though being slower when STEVD and STEVR, provide the most accurate results. Moreover, in cases where eigenvectors are not needed, they can also achieve comparable performance with their counterparts.

References

[1] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and others. *LAPACK Users' Guide*. SIAM, 1999.

[2] James W. Demmel, Inderjit Dhillon, and Huan Ren. On the Correctness of Parallel Bisection in Floating Point. Technical Report UCB//CSD-94-805, University of California, Berkeley, March 1994.

[3] Inderjit S. Dhillon, Beresford N. Parlett, and Christof Vomel. LAPACK Working Note 162: The Design and Implementation of the MRRR Algorithm. Technical Report UCB//CSD-04-1346, University of California, Berkeley, December 2004.

[4] Inderjit Singh Dhillon. A New O(n^2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem, 1997. ISBN: 0591525461.

[5] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse Matrix Test Problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

[6] Michael T Heath. *Scientific Computing: An Introductory Survey, Revised Second Edition*. SIAM, 2018.

[7] J. Rutter. A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem. Technical Report UCB/CSD 94/799, University of California, Berkeley, March 1994.

A Fortran LAPACK Routine Wrappers

f2py is a Fortran to Python interface generator to enable calling Fortran routines in Python code. On top of that, scipy wraps the Fortran LAPACK routines with utilities such as array alignment so that LAPACK routines can be called from Python at ease. Unfortunately, scipy only exposes part of the LAPACK routines, not including the ones we are using in this report. In this appendix we show our implementation of the f2py wrappers (signatures) over the routines used in this report. We show only the double precision variants.

A.1 STEV: QR Iteration

The f2py wrapper is implemented as follows.

```
subroutine dstev(d,e,compute_v,n,z,ldz,work,info)
    ! computes all eigenvalues, and, optionally eigvectors of a real,
    ! symmetric tridiagonal matrix, using QR iteration.

    callstatement (*f2py_func)((compute_v?"V":"N"),&n,d,e,z,&ldz,work,
                                 &info)
    callprotoargument char*,F_INT*,double*,double*,double*,F_INT*,
                      double*,F_INT*

    integer optional,intent(in) :: compute_v = 1
    double precision dimension(n),intent(in,out,copy,out=vals) :: d
    integer depend(d),intent(hide),check(n>0) :: n = shape(d,0)
    double precision depend(n),dimension(MAX(n-1,1)),intent(in,copy)
                     :: e
    integer intent(hide),depend(n) :: ldz = (compute_v?n:1)
    double precision dimension(ldz,(compute_v?n:1)),intent(out),
                     depend(n,ldz) :: z
    double precision dimension((compute_v?MAX(1,2*n-2):1)),depend(n),
                     intent(hide) :: work
    integer intent(out) :: info

end subroutine dstev
```

It can be called in Python as follows.

```python
import numpy as np
from scipy.linalg.lapack import get_lapack_funcs

dstev = get_lapack_funcs("stev", dtype=np.float64)

# Inputs:
#   d       The diagonal of the input matrix
#   e       The subdiagonal of the input matrix
# Outputs:
#   vals    First m entries are the obtained eigenvalues
#   z       First m columns are the obtained eigenvectors
vals, z, _ = dstev(d, e)
```

A.2 STEVD: Divide and Conquer

The f2py wrapper is implemented as follows.

```
subroutine dstevd(d,e,compute_v,n,z,ldz,work,lwork,iwork,liwork,info)
    ! computes all eigenvalues, and, optionally eigvectors of a real,
    ! symmetric tridiagonal matrix, using divide and conquer.

    callstatement (*f2py_func)((compute_v?"V":"N"),&n,d,e,z,&ldz,work,
                               &lwork,iwork,&liwork,&info)
    callprotoargument char*,F_INT*,double*,double*,double*,F_INT*,
                      double*,F_INT*,F_INT*,F_INT*,F_INT*

    integer optional,intent(in) :: compute_v = 1
    double precision dimension(n),intent(in,out,copy,out=vals) :: d
    integer depend(d),intent(hide),check(n>0) :: n = shape(d,0)
    double precision depend(n),dimension(n-1),intent(in,copy) :: e
    integer intent(hide),depend(n) :: ldz = (compute_v?n:1)
    double precision dimension(ldz,(compute_v?n:1)),intent(out),
                     depend(n,ldz) :: z
    integer depend(n),optional,intent(in),
            check(lwork>=((compute_v&&n>1)?1+4*n+n*n:1))
            :: lwork = max(((compute_v&&n>1)?1+4*n+n*n:1),1)
    double precision dimension(lwork),depend(lwork),intent(hide)
                     :: work
    integer depend(n),optional,intent(in),
            check(liwork>=((compute_v&&n>1)?3+5*n:1))
            :: liwork = ((compute_v&&n>1)?3+5*n:1)
    integer dimension(MAX(liwork,1)),depend(liwork),intent(hide)
            :: iwork
    integer intent(out) :: info

end subroutine dstevd
```

It can be called in Python as follows.

```
import numpy as np
from scipy.linalg.lapack import get_lapack_funcs

dstevd = get_lapack_funcs("stevd", dtype=np.float64)

# Inputs:
#    d       The diagonal of the input matrix
#    e       The subdiagonal of the input matrix
# Outputs:
#    vals    First m entries are the obtained eigenvalues
#    z       First m columns are the obtained eigenvectors
vals, z, _ = dstevd(d, e)
```

A.3 STEVX: Bisection and Inverse Iteration

The f2py wrapper is implemented as follows.

```
subroutine dstevx(d,e,range,vl,vu,il,iu,compute_v,abstol,n,m,w,z,ldz,
   work,iwork,ifail,info)
    ! computes selected eigenvalues, and, optionally eigvectors of a
    ! real, symmetric tridiagonal matrix, using bisection.

    callstatement (*f2py_func)((compute_v?"V":"N"),
                               (range>0?(range==1?"V":"I"):"A"),&n,d,e,
                               &vl,&vu,&il,&iu,&abstol,&m,w,z,&ldz,
                                    work,iwork,ifail,&info)
    callprotoargument char*,char*,F_INT*,double*,double*,double*,
                      double*,F_INT*,F_INT*,double*,F_INT*,double*,
                      double*,F_INT*,double*,F_INT*,F_INT*,F_INT*

    integer optional,intent(in) :: compute_v = 1
    double precision dimension(n),intent(in,copy) :: d
    integer depend(d),intent(hide),check(n>0) :: n = shape(d,0)
    double precision depend(n),dimension(MAX(n-1,1)),intent(in,copy)
                     :: e
    integer intent(in) :: range
    double precision intent(in) :: vl
    double precision intent(in) :: vu
    integer intent(in) :: il
    integer intent(in) :: iu
    double precision optional,intent(in) :: abstol=0.0
    integer intent(out) :: m
    double precision dimension(n),depend(n),intent(out) :: w
    integer intent(hide),depend(n) :: ldz = (compute_v?n:1)
    double precision dimension(ldz,(compute_v?n:1)),intent(out),
                     depend(n,ldz) :: z
    double precision dimension(5*n),depend(n),intent(hide) :: work
    integer dimension(5*n),depend(n),intent(hide) :: iwork
    integer dimension(n),depend(n),intent(hide) :: ifail
    integer intent(out) :: info

end subroutine dstevx
```

It can be called in Python as follows.

```python
import numpy as np
from scipy.linalg.lapack import get_lapack_funcs

dstevx = get_lapack_funcs("stevx", dtype=np.float64)

# Inputs:
#   d       The diagonal of the input matrix
#   e       The subdiagonal of the input matrix
#   range   0 - all; 1 - value range (vl, vu]; 2 - index range [il, iu]
#   vl      Used when range == 1
#   vu      Used when range == 1
#   il      Used when range == 2
#   iu      Used when range == 2
# Outputs:
#   m       Number of obtained eigenvalues
#   vals    First m entries are the obtained eigenvalues
#   z       First m columns are the obtained eigenvectors
m, vals, z, _ = dstevx(d, e, range, vl, vu, il, iu)
```

A.4 STEVR: MRRR Algorithm

The f2py wrapper is implemented as follows.

```
subroutine dstevr(d,e,range,vl,vu,il,iu,compute_v,abstol,n,m,w,z,ldz,
    isuppz,work,lwork,iwork,liwork,info)
    ! computes selected eigenvalues, and, optionally eigvectors of a
    ! real, symmetric tridiagonal matrix, using MRRR.

    callstatement (*f2py_func)((compute_v?"V":"N"),
                              (range>0?(range==1?"V":"I"):"A"),&n,d,e,
                              &vl,&vu,&il,&iu,&abstol,&m,w,z,&ldz,
                              isuppz,work,&lwork,iwork,&liwork,&info)
    callprotoargument char*,char*,F_INT*,double*,double*,double*,
                      double*,F_INT*,F_INT*,double*,F_INT*,double*,
                      double*,F_INT*,F_INT*,double*,F_INT*,F_INT*,
                      F_INT*,F_INT*

    integer optional,intent(in) :: compute_v = 1
    double precision dimension(n),intent(in,copy) :: d
    integer depend(d),intent(hide),check(n>0) :: n = shape(d,0)
    double precision depend(n),dimension(MAX(n-1,1)),intent(in,copy)
                     :: e
    integer intent(in) :: range
    double precision intent(in) :: vl
    double precision intent(in) :: vu
    integer intent(in) :: il
    integer intent(in) :: iu
    double precision optional,intent(in) :: abstol=0.0
    integer intent(out) :: m
    double precision dimension(n),depend(n),intent(out) :: w
    integer intent(hide),depend(n) :: ldz = (compute_v?n:1)
    double precision dimension(ldz,(compute_v?n:1)),intent(out),
                     depend(n,ldz) :: z
    integer dimension((compute_v?2*n:1)),depend(n),intent(hide)
            :: isuppz
    integer depend(n),optional,intent(in),check(lwork>=MAX(20*n,1))
            :: lwork = max(20*n,1)
    double precision dimension(MAX(1,lwork)),depend(lwork),
                     intent(hide) :: work
    integer depend(n),optional,intent(in),check(liwork>=MAX(10*n,1))
            :: liwork = (10*n)
    integer dimension(MAX(1,liwork)),depend(liwork),intent(hide)
            :: iwork
    integer intent(out) :: info

end subroutine dstevr
```

It can be called in Python as follows.

```python
import numpy as np
from scipy.linalg.lapack import get_lapack_funcs

dstevr = get_lapack_funcs("stevr", dtype=np.float64)

# Inputs:
#   d       The diagonal of the input matrix
#   e       The subdiagonal of the input matrix
#   range   0 - all; 1 - value range (vl, vu]; 2 - index range [il, iu]
#   vl      Used when range == 1
#   vu      Used when range == 1
#   il      Used when range == 2
#   iu      Used when range == 2
# Outputs:
#   m       Number of obtained eigenvalues
#   vals    First m entries are the obtained eigenvalues
#   z       First m columns are the obtained eigenvectors
m, vals, z, _ = dstevr(d, e, range, vl, vu, il, iu)
```

B Testing Suite Implementation

In this appendix we show the implementation of our synthetic matrix generators (§B.1) and the evaluation metrics (§B.2). `machep` is the machine epsilon, `np.finfo(np.float64).eps`.

B.1 Synthetic Matrix Generators

The function names correspond to the types of the generated synthetic matrices, as specified in §3.2. Here `rng` refers to a random number generator instance for reproducibility.

```python
def uniform_eps(n, rng):
    eigenvalues = machep * np.arange(1, n, dtype=np.float64)
    eigenvalues = np.append(eigenvalues, 1.0)

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def uniform_sqeps(n, rng):
    eigenvalues = 1 + np.sqrt(machep) * np.arange(2, n, dtype=np.
                                        float64)
    eigenvalues = np.insert(eigenvalues, 0, machep)
    eigenvalues = np.append(eigenvalues, 2.0)

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def uniform_equi(n, rng):
    tau = (1.0 - machep) / (n - 1)
    eigenvalues = machep + tau * np.arange(n, dtype=np.float64)

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def geometric(n, rng):
    numers = n - np.arange(1, n + 1, dtype=np.float64)
    eigenvalues = np.exp(machep, numers / (n - 1))

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def normal(n, rng):
    eigenvalues = rng.normal(0, 1, n)

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def clustered_one(n, rng):
    eigenvalues = np.ones(n)
    eigenvalues[1:] += machep * rng.random(n - 1)
    eigenvalues[0] = machep

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def clustered_pmone(n, rng):
    eigenvalues = np.ones(n)
    eigenvalues[1:] *= rng.choice([-1, 1], n - 1)
    eigenvalues[1:] += machep * rng.random(n - 1)
    eigenvalues[0] = machep

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def clustered_machep(n, rng):
    eigenvalues = np.full(n, machep)
    eigenvalues[:-1] += machep * rng.random(n - 1)
    eigenvalues[-1] = 1.0

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```python
def clustered_pmmachep(n, rng):
    eigenvalues = np.full(n, machep)
    eigenvalues[:-1] *= rng.choice([-1, 1], n - 1)
    eigenvalues[:-1] += machep * rng.random(n - 1)
    eigenvalues[-1] = 1.0

    Q, _ = np.linalg.qr(rng.random((n, n)))
    Lambda = np.diag(eigenvalues)
    A = Q @ Lambda @ Q.T
    _, d, e, _, info = dsytrd(A, lower=True)
    assert info == 0, f"info={info}"
    return d, e
```

```
def wilkinson(n, __unused):
    d = np.arange(1, n + 1, dtype=np.float64)
    e = np.full(n - 1, 1.0, dtype=np.float64)
    return d, e
```

B.2 Evaluation Metrics

Let n be the matrix size and d and e be the diagonal and subdiagonal elements of the symmetric tridiagonal test matrix. vals and z are the computed eigenvalues and eigenvector matrix, respectively. The loss of orthogonality and largest residual norm metrics, as specified in §3.3, are implemented in orthogonality_loss and max_residual_norm as follows.

```
def orthogonality_loss(z, n):
    inner = np.abs(z.T @ z)
    np.fill_diagonal(inner, 0)
    return np.max(inner) / (n * machep)
```

```
def max_residual_norm(d, e, vals, z, n):
    T = np.diag(d) + np.diag(e, 1) + np.diag(e, -1)
    residuals = [
        np.linalg.norm(T @ z[:, i] - vals[i] * z[:, i])
        for i in range(n)
    ]
    return np.max(residuals) / (np.linalg.norm(T) * n * machep)
```