A TESTING INFRASTRUCTURE FOR LAPACK'S SYMMETRIC EIGENSOLVERS*

JAMES W. DEMMEL‡ OSNI A. MARQUES† BERESFORD N. PARLETT‡ AND CHRISTOF VÖMEL†

Abstract. LAPACK is often mentioned as a positive example of a software library that encapsulates complex, robust, and widely used numerical algorithms for a wide range of applications. At installation time, the user has the option of running a (limited) number of test cases to verify the integrity of the installation process. On the algorithm developer's side, however, more exhaustive tests are usually performed to study algorithm behavior on a variety of problem settings and also computer architectures. In this process, difficult test cases need to be found that reflect particular challenges of an application or push algorithms to extreme behavior. These tests are then assembled into a comprehensive collection, therefore making it possible for any new or competing algorithm to be stressed in a similar way. This note describes such an infrastructure for exhaustively testing the symmetric tridiagonal eigensolvers implemented in LAPACK. It consists of two parts: a selection of carefully chosen test matrices with particular idiosyncrasies and a portable testing framework that allows easy testing and data processing. The tester facilitates experiments with algorithmic choices, parameter and threshold studies, and performance comparisons on different architectures.

AMS subject classifications. 15A18, 15A23.

1. Introduction. The LAPACK software library [2] provides a variety of drivers and computational routines for the sequential solution of Numerical Linear Algebra Problems. In preparation for a new release of LAPACK in 2007, we found it necessary to create a comprehensive testing environment to evaluate and compare the latest versions of LAPACK's symmetric eigensolvers. This paper describes our infrastructure, consisting of a test program and a set of test matrices.

Our approach was guided by the following goals:

1. to facilitate a critical examination of algorithm parameters,
2. to allow easy post-processing of test results by tools such as Matlab or Excel,
3. to create an easily portable infrastructure for various architectures, and
4. to provide interesting and challenging test cases.

LAPACK provides testers for a basic verification of its drivers and computational routines at installation time. This is useful for detecting problems caused by too aggressive compiler optimization options or IEEE arithmetic features.

Our new testing infrastructure, `stetester`, is more comprehensive and useful for deeper studies of algorithms than the LAPACK tester. Following the approach adopted in the development of LAPACK, the tester is written in FORTRAN. It has three main applications:

1. to experiment with and select algorithmic variants as in [38],
2. to tune parameters and thresholds such as those to be set by LAPACK's ILAENV,
3. to carry out large scale performance comparisons such as [38, 10], and
4. to validate and benchmark new algorithms such as [39].

*This work was partly supported by a grant from the National Science Foundation (Cooperative Agreement no. ACI-9619020), and by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract No. DE-AC03-76SF00098.

†Lawrence Berkeley National Laboratory, 1 Cyclotron Road, MS 50F-1650, Berkeley, CA 94720, USA, OAMarques,CVoemel@lbl.gov.

‡Mathematics Department and Computer Science Division, University of California, Berkeley, CA 94720, USA. demmel@cs.berkeley.edu,parlett@math.berkeley.edu

A brief summary of LAPACK's symmetric eigensolvers for which this tester has been developed is given in Table 1.1. [1]

Algorithm name	LAPACK subroutine	Subset Feature	Workspace Real/Integer	References
QR algorithm	STEQR	N	$2n-2/0$	[9, 31, 40]
Divide & Conquer	STEDC	N	$1+4n+n^2/3+5n$	[5, 7, 33, 34]
Bisection/Inverse Iter.	STEBZ/STEIN	Y	$8n/5n$	[13, 36]
MRRR algorithm	STEGR	Y	$18n/10n$	[12, 41, 42, 15, 16, 17]

TABLE 1.1

LAPACK codes for computing eigenpairs of a symmetric tridiagonal matrix of dimension n, see also [2, 4]. Note that in addition to computing all eigenpairs, inverse iteration and the MRRR algorithm (MRRR = Multiple Relatively Robust Representations) also allow the computation of eigenpair subsets at reduced cost.

While it is possible to perform some of the tasks of this testing environment in Matlab, our tester does have some benefits:

- Being written in Fortran, it is portable across architectures at no cost.
- Low-level access to compiler options, IEEE directives, algorithm parameters, and detailed performance measures is possible.
- The overhead with respect to memory is negligible so that large matrices can be tested.

One particularly useful feature is our collection of *interesting* test matrices. We are interested in two types, ones that

- exhibit the strengths, weaknesses, or idiosyncrasies of a particular algorithm, and/or
- stem from an important application and thus are relevant to a group of users.

Matrix *classes* or *families* can be of additional use as they allow scalability studies. As an example, matrices with clustered eigenvalues are considered to be difficult. Inverse iteration requires Gram-Schmidt orthogonalization to guarantee orthogonal eigenvectors. Likewise, the MRRR algorithm struggles with very tight clusters of eigenvalues, see the examples in [18]. As a second example, in order to understand the complexity of the Divide-and-Conquer algorithm, it is crucial to study the matrix-dependent deflation process (see for example [9]).

Test matrices were originally designed to validate new algorithms. As the authors of [32] state: 'what is needed is a collection of numerical examples with which to test each algorithm as soon as it is proposed.' Today, aspects other than pure validation have become equally important, in particular performance-related ones. There exist a number of sparse matrix collections that are most commonly used to evaluate the performance of sparse direct solvers [20, 21, 22, 8]. Furthermore, Matlab provides a test matrix collection based on [35]. Matrix market contains non-Hermitian eigenvalue problems from [3]. Finally, a test matrix generator of large sparse matrices with given spectrum for the evaluation of iterative methods was recently presented in [37].

This paper makes two contributions:

1. The testing infrastructure with the features mentioned above.
2. A documentation of the reasons for including certain test matrices and how they are interesting.

[1] The workspace that is reported for Divide & Conquer corresponds to the case COMPZ = 'I'. The workspace that is reported for Bisection/Inverse Iteration is for SYEVX, the driver that combines STEBZ and STEIN.

TABLE 2.1
Lapack-style test matrices with a given eigenvalue distribution. For distributions 1-5, the parameter k can be chosen as ulp^{-1} like in the LAPACK tester but other choices are also possible, see the options for parameter `ECOND` *in Table A.1. For distribution 6, see parameter* `EDIST` *in Table A.1.*

type	description
1	$\lambda_1 = 1,\ \lambda_i = \frac{1}{k},\ i = 2, 3, \ldots n$
2	$\lambda_i = 1,\ i = 2, 3, \ldots n-1,\ \lambda_n = \frac{1}{k}$
3	$\lambda_i = k^{-(\frac{i-1}{n-1})},\ i = 1, 2, \ldots n$
4	$\lambda_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k}),\ i = 1, 2, \ldots n$
5	n random numbers in the range $(\frac{1}{k}, 1)$, their logarithms are uniformly distributed
6	n random numbers from a specified distribution
7	$\lambda_i = ulp \times i,\ i = 1, 2, \ldots n-1,\ \lambda_n = 1$
8	$\lambda_1 = ulp,\ \lambda_i = 1 + \sqrt{ulp} \times i,\ i = 2, 3, \ldots n-1, \lambda_n = 2$
9	$\lambda_1 = 1,\ \lambda_i = \lambda_{i-1} + 100 \times ulp, i = 2, \ldots n$

The rest of this paper is organized as follows. In Section 2, we describe the matrix types that are currently available in the tester and justify their relevance as important test cases. Section 3 describes the design of the testing infrastructure. Concluding remarks are given in Section 4. The Appendix A contains additional information on the tester including input and output sample files.

2. Test matrices.

2.1. Random matrices with given eigenvalue Distributions. Table 2.1 lists test matrix classes that are of 'LAPACK-style', that is distributions that are already used in LAPACK's tester [11], fabricated distributions, and matrices that are used in [12]. Given the eigenvalues $D = \mathrm{diag}(\lambda)$, the matrix $A = Q^T D Q$ is formed using an orthogonal matrix Q generated from random entries. Subsequently, A is reduced to tridiagonal form using LAPACK's `sytrd`.

These test matrices are useful for the following reasons:

- The user can freely specify an eigenvalue distribution.
- The matrices do not need to be stored and can be generated on the fly.
- Matrices of arbitrary sizes can be generated.
- Clustered eigenvalues can be created easily.

In contrast to the tester in the LAPACK distribution, test parameters are directly accessible and can be changed easily without recompilation.

In addition to the distributions listed in Table 2.1, the user can also read a list of eigenvalues from a file and then generate a tridiagonal as above using LAPACK's `latms`. See Section 3 for a description of this feature.

2.2. Matrices with interesting properties. Table 2.2 lists a number of matrix types included in our collection. Their performance-relevant properties are explained in the following Sections 2.2.1 and 2.2.2.

2.2.1. Classical test matrices. The (1-2-1) matrix is defined as

$$T = \mathrm{tridiag}\begin{pmatrix} & 1 & & 1 & & 1 & \\ 2 & & 2 & & \cdots & & 2 \\ & 1 & & 1 & & 1 & \end{pmatrix},$$

TABLE 2.2
Built-in matrices with distinguishing performance-relevant features.

type	description
0	zero matrix
1	identity matrix
2	(1,2,1) tridiagonal matrix
3	Wilkinson-type tridiagonal matrix
4	Clement-type tridiagonal matrix
5	Legendre-type tridiagonal matrix
6	Laguerre-type tridiagonal matrix
7	Hermite-type tridiagonal matrix

the Wilkinson matrix is

$$W_{2m+1}^{+} = \text{tridiag}\left(\begin{array}{ccccccccccc} & 1 & & 1 & & & & & & 1 & \\ m & & m-1 & \cdots & 1 & 0 & 1 & \cdots & m-1 & & m \\ & 1 & & 1 & & & & 1 & & 1 & \end{array} \right),$$

and the (symmetrized) Clement matrix is given by

$$T = \text{tridiag}\left(\begin{array}{ccccccccc} & \sqrt{n} & & \sqrt{2(n-1)} & & \sqrt{(n-1)2} & & \sqrt{n} & \\ 0 & & 0 & & \cdots & & 0 & & 0 \\ & \sqrt{n} & & \sqrt{2(n-1)} & & \sqrt{(n-1)2} & & \sqrt{n} & \end{array} \right),$$

that is the off-diagonal entries are $\sqrt{i(n+1-i)}, i = 1, \ldots, n$.

These matrices have a number of interesting features that may favor one algorithm over another. We give a short summary here, a more detailed description and analysis can be found in [10].

- The (1-2-1) matrix [32, 35] is the archetype of the symmetric tridiagonal Toeplitz matrix. Its eigenvalues and -vectors are known analytically, see [29] for a derivation. The eigenvalue clustering is not very strong (although clustering becomes tighter with growing dimension). Thus the matrix is relatively easy for MRRR to tackle. On the other hand, the top and bottom eigenvector entries that govern deflation in the Divide and Conquer algorithm do not decay quickly making this matrix class a difficult one for this algorithm.
- Wilkinson matrices [43, 32, 35] are the opposite of (1-2-1) matrices in that they strongly favor Divide and Conquer over the MRRR algorithm. The top and bottom eigenvector entries decay very quickly below the deflation threshold. It can be verified that Divide and Conquer deflates all but a small number of eigenvalues (the number depends on the precision and the deflation threshold). On the other hand, since almost all eigenvalues of W_{2m+1}^{+} come in pairs which are well separated but increasingly close, the representation tree generated by the MRRR algorithm is very broad and the overhead for the tree generation is considerable. (For a definition of the representation tree and a discussion of its importance for the MRRR algorithm, see for example [17].)
- The symmetrized version of the Clement (also called Kac) matrix [6, 32, 35] represent one interesting example of Golub-Kahan type. Its eigenvalues are the integers $\pm(n), \pm(n-2), \ldots$. From the computational point of view, the matrix defines all its eigenvalues to high relative accuracy. As all eigenvalues

are well spaced, matrices of this type are easy for inverse iteration and MRRR. On the other hand, deflation in Divide and Conquer is limited (especially when the matrix dimension is odd) because the top and bottom entries of the eigenvectors do not decay quickly.

In addition to the matrices listed in Table 2.2, the user can also read a tridiagonal matrix from a file. See Section 3 for a description of this feature.

2.2.2. Tridiagonals from classical orthogonal polynomials. There is a direct correspondence between tridiagonal matrices and certain families of orthogonal polynomials, see for example [30, 25, 26, 27, 28, 40]. Our presentation uses the notation from Section 22.7 in [1]. [2] For $i \geq 1$, the three-term recurrence

$$\frac{a_{4i}}{a_{3i}} f_{i-1} + \left(\frac{-a_{2i}}{a_{3i}} - \lambda \right) f_i + \frac{a_{1i}}{a_{3i}} f_{i+1} = 0 \tag{2.1}$$

defines a (non-symmetric) tridiagonal matrix which is similar to a symmetric one provided that the coefficients satisfy $a_{4i+1}a_{1i} \geq 0$. Along these lines, the following symmetric tridiagonal matrices can be derived from Section 22.7 in [1].

The Legendre recurrence, (22.7.10) in [1], yields

$$T = \text{tridiag} \begin{pmatrix} & 2/\sqrt{3\cdot 5} & & 3/\sqrt{5\cdot 7} & & n/\sqrt{(2n-1)(2n+1)} & \\ 0 & & 0 & & \cdots & & 0 \\ & 2/\sqrt{3\cdot 5} & & 3/\sqrt{5\cdot 7} & & n/\sqrt{(2n-1)(2n+1)} & \end{pmatrix},$$

that is the off-diagonal entries are $i/\sqrt{(2i-1)(2i+1)}, i = 2, \ldots, n$, see also [29].

Using the Laguerre recurrence, (22.7.12) in [1], with $\alpha = 0$ and off-diagonal entries chosen to be positive, one obtains

$$T = \text{tridiag} \begin{pmatrix} & 2 & & 3 & & n-1 & & n & \\ 3 & & 5 & & \cdots & & 2n-1 & & 2n+1 \\ & 2 & & 3 & & n-1 & & n & \end{pmatrix}.$$

Lastly, the Hermite recurrence, (22.7.14) in [1], gives

$$T = \text{tridiag} \begin{pmatrix} & \sqrt{1} & & \sqrt{2} & & \sqrt{n-2} & & \sqrt{n-1} & \\ 0 & & 0 & & \cdots & & 0 & & 0 \\ & \sqrt{1} & & \sqrt{2} & & \sqrt{n-2} & & \sqrt{n-1} & \end{pmatrix}.$$

(Note that the Chebyshev polynomials result in symmetric tridiagonal Toeplitz matrices that are affine translates of the (1-2-1) matrix from Section 2.2.)

All of these matrices have a fairly spread-out spectrum: compared to the spectral diameter, their eigenvalues are not very strongly clustered. Consequently, the MRRR algorithm can cope with them very efficiently. On the other hand, these matrix classes are challenging for Divide & Conquer which does no or little deflation on them compared to other matrix types.

[2] In [1], the coefficients of classical three-term recurrences are reported in the form $a_{1i}f_{i+1}(\lambda) = (a_{2i}+a_{3i}\lambda)f_i(\lambda) - a_{4i}f_{i-1}(\lambda)$, where $a_{3i} \neq 0$. We use the equivalent (2.1) from which the tridiagonal can be directly derived.

2.3. Glued Matrices. Glued matrices, in particular glued Wilkinson matrices, emerged as important test matrices in particular for the MRRR algorithm, see [18].

For symmetric tridiagonal matrices $T_1, \ldots, T_p$, define

$$T = \begin{bmatrix} T_1 & & \\ & \ddots & \\ & & T_p \end{bmatrix} + \sum_{i=1}^{p-1} \gamma_i \left(x_i y_i^T + y_i x_i^T \right) , \tag{2.2}$$

where $x_i, y_i, i = 1, \ldots, p-1$ are columns of the identity corresponding to the interfaces between the diagonal blocks. The quantities γ_i are the glue factors.

The tester allows the user to generate general glued matrices by taking any combination of glue factors and matrices generated from the types listed in Tables 2.1 and 2.2; see Sections 2.4 and 3 for details.

2.4. Tridiagonal matrices from applications. In the following, we describe those tridiagonal matrices in our tester that stem from applications.

- Examples from applications in computational quantum chemistry and electronic structure calculations. The tridiagonal matrices stem from solving a Schroedinger equation [12, 14] and were provided by George Fann using the NWChem computational chemistry package [24, 23]. Their dimensions range from 120 to 2053. These matrices have clustered eigenvalues that require a large number of reorthogonalizations when inverse iteration is used. This motivated the development of the MRRR algorithm which can cope well with this type of matrix.
- Examples from sparse matrix collections. These include matrices from the BCSSTRUC1 set in the Harwell-Boeing Collection [20, 21, 22] and matrices from the Alemdar, NASA, and Cannizzo sets in the University of Florida Sparse Matrix Collection [8]. The matrices are related to the modeling of power system networks, a finite-difference model for the shallow wave equations for the Atlantic and Indian Oceans, and finite-element problems. The dimension of the corresponding tridiagonal matrices range from 48 to 8012. These matrices were chosen for their spectrum which typically consists of a part with eigenvalues varying almost 'continuously' and another one with several isolated large clusters of eigenvalues of varying tightness. A large number of reorthogonalization steps is required within these clusters when inverse iteration is used.

For small matrices, the tridiagonal form of the sparse matrices was obtained with LAPACK's tridiagonal reduction routine `sytrd`. For the larger matrices we generated tridiagonals by means of a simple Lanczos algorithm without reorthogonalization and a starting vector filled with ones, as a way to provoke the appearance of very close eigenvalues in the resulting tridiagonal. In this case, we ran Lanczos for $k \times n$ steps, where $k = 1, \ldots, 4$ and n is the dimension of the original sparse matrix. A Matlab description of the Lanczos algorithm is given in Appendix A.4.

2.5. What can be learned. Algorithm development and systematic testing using multiple platforms and various matrix classes is a challenging task. A systematic comparison of LAPACK's eigensolvers with respect to floating point operations, time, and accuracy can be found in [10]. To give the reader a glimpse of the findings, we describe a few interesting issues.

Why can the Divide & Conquer algorithm on a (1-2-1) matrix of dimension 2048 be almost twice as fast as on one of dimensions 2047 or 2049? Why, in contrast, does

the MRRR algorithm run equaly fast for all three of them? (Answer: substantially more deflation occurs in Divide & Conquer for n=2048. On the other hand, the eigenvalue distributions of the three matrices are so close that there is no runtime difference in MRRR.)

What impact has the BLAS library on tridiagonal eigensolvers? (Answer: except for Divide & Conquer, LAPACK's tridiagonal eigensolvers do not make use of higher level BLAS. However, between the self-compiled reference BLAS from Netlib and ATLAS BLAS, we noted timing differences of up to a factor of 10 for the Divide & Conquer algorithm.)

If the MRRR algorithm uses significantly fewer floating point operations than Divide & Conquer, why can it still be slower on some matrices? (Answer: while Divide & Conquer spends a majority of its time on matrix-matrix multiplication, MRRR relies heavily on Sturm counts which have an expensive division operation in each step.)

3. Design of the testing infrastructure. This section describes in detail the testing infrastructure `stetester`.

Section 3.1 describes performance measures and the criteria for evaluating numerical results.

Input data for the tester is specified by means of key words or macros. These macros are groups of characters that uniquely define a specific subset of the input data, such as matrix types and dimensions, matrices to be read from files, etc. The tester includes a parser that interprets the data and the appropriate matrix types, parameters, and eigensolvers to be called. This is described in Section 3.2. After completion, the tester can return a large number of statistics regarding the performed tests, see Section 3.3.

3.1. Test criteria.

3.1.1. Performance measures: time, flops, flips. The Fortran 95 function `cpu_time` is used by default for cross-architecture portability. To obtain more detailed performance-related information on floating point operations or instructions, PAPI [19] can be used.

In order to achieve accurate timing results for matrices of smaller dimension, the same test case can be run multiple times and the average runtime is reported. The number of tests can be set such that the sum of all run times is significantly larger than the timer resolution.

3.1.2. Numerical accuracy. For a tridiagonal matrix T and computed eigenvectors $Z = [z_1\, z_2 \cdots z_m]$ and eigenvalues $W = diag(w_1\, w_2 \cdots w_m)$, $m \leq n$, `stetester` performs the following tests using the LAPACK routine `lansy`:

$$\frac{||I - ZZ^T||}{n \times ulp},\ if\ m = n \tag{3.1}$$

$$\frac{||I - Z^T Z||}{n \times ulp},\ if\ m < n \tag{3.2}$$

which measures the orthogonality of the computed eigenvectors, and

$$\frac{||T - ZWZ^T||}{||T|| \times n \times ulp},\ if\ m = n \tag{3.3}$$

$$\frac{||Z^T TZ - W||}{||T|| \times n \times ulp},\ if\ m < n \tag{3.4}$$

which measures the accuracy of the computed eigenpairs.

For large matrices these tests can take a significant amount of time, so the user is given the option of choosing how many diagonals of the involved arrays are actually computed. Because eigenvalues are sorted from smallest to larges, faulty results in the orthogonality tests are likely to be caused by eigenvectors related to eigenvalues that are nor far apart from each other. By taking say 10% or less of the diagonals we can save a significant amount of CPU time in large cases.

For testing the subset functionality in `stevx` and `stegr`, we pursue two approaches. For computing eigenvalues from a given smallest to a given largest index (parameter RANGE='I'), the code can pick random pairs of integers between 1 and n. For computing eigenvalues within an interval (parameter RANGE='V'), the code can select random interval bounds between ($\lambda_{\min}$ and $\lambda_{\max}$) (which are obtained with bisection, subroutine `stebz`).

3.2. Input Data and Key Words. The interface to `stetester` is a text parser. On input, key words can be specified in any order in the input file, either in lower or upper case, with the corresponding subset of data that they define. Data can be separated by blanks or commas, and the character % is interpreted as the beginning of a comment. Therefore, a line in the input file that begins with a % is simply ignored. The macros currently supported by `stetester` are listed in Tables A.1-A.3 in Appendix A.1. As an example, a simple input file is given in Table A.4 in Appendix A.2.

3.3. Output Files. One of the important features of the testing infrastructure is its ability to print data for post-processing. The following output files can be generated optionally:

- File `stetester.out.T`. Tridiagonal matrix written as triplets j, d_j, e_j for each case (including the current seed of the random number generator if applicable).
- File `stetester.out.W`. Eigenvalues obtained by each tridiagonal eigensolver called.
- File `stetester.out.Z`. Eigenvectors obtained by each tridiagonal eigensolver called.
- File `stetester.out.log`. Timing and results of the tests for orthogonality (3.1) and residual norm (3.3) for each tridiagonal eigensolver called.
- File `stetester.out.m`. Tridiagonal matrix, eigenvalues and eigenvectors written in Matlab notation. The tridiagonal is represented by arrays `D_i` (diagonal entries) and `E_i` (off-diagonal entries), where `i` is the case number. The eigenvalues and eigenvectors are given in arrays `W_i_s` and `Z_i_s`, respectively, where `i` is the case number and `s` identifies the tridiagonal eigensolver according to the algorithm list as used for parameter `CALLST` in Table A.1. For example, 1 stands for `steqr` with COMPZ='V', 2 for `stevx` with RANGE='A', and so forth. This is to distinguish between results computed with different algorithms.

As an illustration, a simple Matlab-style output file is given in Table A.5 in Appendix A.3.

4. Conclusions and availability. The main strength of the testing infrastructure described here is its ability to easily generate, read, and process a variety of test cases for symmetric tridiagonal eigensolvers. Together with our collection of test matrices, it allows users to perform exhaustive tests on a large range of computer

platforms, with various compilers, revealing not only interesting numerical behavior but also important performance issues. It is available upon request.

In our experience, the presented infrastructure is an invaluable tool. As a collection of interesting matrices, it constitutes a benchmark for algorithm evaluation. As portable software infrastructure, it can help identify failures in algorithms, devise mechanisms to make algorithms more robust and efficient, and compare algorithms across computer architectures. We therefore anticipate it to be helpful for users who are interested in similar studies.

REFERENCES

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, NY, 9 edition, 1965.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3. edition, 1999.

[3] Z. Bai, D. Day, J. W. Demmel, and J. J. Dongarra. A Test Matrix Collection for Non-Hermitian Eigenvalue Problems. Technical Report UT-CS-97-355, University of Tennessee, Knoxville, TN, USA, 1997. Also as LAPACK Working Note 123.

[4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems - A practical guide*. SIAM, Philadelphia, 2000.

[5] J. Bunch, P. Nielsen, and D. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.

[6] P. A. Clement. A class of triple-diagonal matrices for test purposes. *SIAM Review*, 1:50–52, 1959.

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

[8] T. A. Davis. University of florida sparse matrix collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[9] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[10] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel. Lapack Working Note xxx: Accuracy and Performance of LAPACK's Symmetric Tridiagonal Eigensolvers. University of California, Berkeley, 2006. In preparation.

[11] J. W. Demmel and A. McKenney. A test matrix generation suite. Computer science dept. technical report, Courant Institute, New York, NY, 1989. (also LAPACK Working Note #9).

[12] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, University of California, Berkeley, California, 1997.

[13] I. S. Dhillon. Current inverse iteration software can fail. *BIT*, 38:4:685–704, 1998.

[14] I. S. Dhillon, G. Fann, and B. N. Parlett. Application of a new algorithm for the symmetric eigenproblem to computational quantum chemistry. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[15] I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl.*, 387:1–28, 2004.

[16] I. S. Dhillon and B. N. Parlett. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2004.

[17] I. S. Dhillon, B. N. Parlett, and C. Vömel. LAPACK working note 162: The design and implementation of the MRRR algorithm. Technical Report UCBCSD-04-1346, University of California, Berkeley, 2004. Revised version to appear in ACM Trans. Math. Soft.

[18] I. S. Dhillon, B. N. Parlett, and C. Vömel. Glued matrices and the MRRR algorithm. *SIAM J. Sci. Comput.*, 27(2):496–510, 2005. Revised version of LAPACK Working Note 163.

[19] J. J. Dongarra, S. Moore, P. Mucci, K. Seymour, D. Terpstra, and H. You. Performance Application Programming Interface (PAPI).

[20] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[21] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report RAL-TR-92-086, Atlas Centre, Rutherford Appleton Laboratory, 1992.

[22] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Atlas Centre, Rutherford Appleton Laboratory, 1997.

Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.
[23] E. Apra et al. NWChem, a computational chemistry package for parallel computers, version 4.7. Technical report, Pacific Northwest National Laboratory, Richland, WA. USA, 2005.
[24] R. A. Kendall et al. High Performance Computational Chemistry: An overview of NWChem a distributed parallel application. *Computer Phys. Comm.*, 128:260–283, 2000.
[25] W. Gautschi. Algorithm 726: ORTHPOL–a package of routines for generating orthogonal polynomials and Gauss-type quadrature rules. *ACM Trans. Math. Software*, 20(1):21–62, 1994.
[26] W. Gautschi. The interplay between classical analysis and (numerical) linear algebra — a tribute to Gene H. Golub. *Electronic Transactions on Numerical Analysis*, 13:119–147, 2002.
[27] W. Gautschi. *Orthogonal polynomials: computation and approximation*. Oxford University Press, Oxford, 2004.
[28] W. Gautschi. Orthogonal polynomials (in Matlab). *J. Comp. Appl. Math.*, 178:215–234, 2005.
[29] S. K. Godunov, A. G. Antonov, O. P. Kiriljuk, and V. I. Kostin. *Guaranteed Accuracy in Numerical linear Algebra*. Kluwer Academic, Dordrecht, The Netherlands, 1993.
[30] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15:318–334, 1973.
[31] G. H. Golub and C. van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, Maryland, 3. edition, 1996.
[32] R. Gregory and D. Karney. *A Collection of Matrices for Testing Computational Algorithms*. Wiley, New York, 1969.
[33] M. Gu and S. C. Eisenstat. A stable and efficient algorithm for the rank-1 modification of the symmetric eigenproblem. *SIAM J. Matrix Anal. Appl.*, 15(4):1266–1276, 1994.
[34] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16(1):172–191, 1995.
[35] N. J. Higham. Algorithm 694: A Collection of Test Matrices in MATLAB. *ACM Trans. Math. Software*, 17(3):289–305, 1991.
[36] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2):254–291, 1997.
[37] C. R. Lee and G. W. Stewart. Eigentest: A test matrix generator for large-scale eigenproblems. UMIACS TR-2006-07, CMSC TR-4783, University of Maryland, College Park, MD, 2006.
[38] O. A. Marques, E. J. Riedy, and C. Vömel. Lapack working note 172: Benefits of IEEE-754 features in modern symmetric tridiagonal eigensolvers. Technical Report UCBCSD-05-1414, University of California, Berkeley, 2005. To appear in SIAM J. Sci. Comp.
[39] A. M. Matsekh. The Godunov-inverse iteration: a fast and accurate solution to the symmetric tridiagonal eigenvalue problem. *Appl. Numer. Math.*, 54(2):208–221, 2005.
[40] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM Press, Philadelphia, PA, 1998.
[41] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: an application of double factorization. *Linear Algebra and Appl.*, 267:247–279, 1997.
[42] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and Appl.*, 309(1-3):121–151, 2000.
[43] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.

Appendix A. Usage of the testing infrastructure `stetester`.

A.1. Supported macros. This section serves as a reference. Tables A.1, A.2, and A.3 contain all currently supported macros.

A.2. A sample input file. Table A.4 contains a sample input file. After the test matrices have been specified, the algorithms to be tested ('ALL') and the output format are selected.

A.3. A sample output file in Matlab format. Table A.5 contains a sample output file in Matlab format of a tridiagonal matrix with diagonal D and offdiagonal E. Printed are the computed eigenpairs W, Z from two different computations with the same matrix, first using QR and then MRRR.

A.4. Lanczos code. In Table A.6 we present a simple block Lanczos procedure without reorthogonalization in Matlab format. It can be used to generate tridiagonal matrices from sparse matrices or matrix pairs.

TABLE A.1
Key words for `stetester`*, part 1.*

Key word	argument	purpose
`CALLST`	*list*	Defines the subroutines to be tested. Possible entries in *list* are: `STEQRV` (calls STEQR with COMPZ='V') `STEVXA` (calls STEVX with RANGE='A') `STEVXI` (calls STEVX with RANGE='I') `STEVXV` (calls STEVX with RANGE='V') `STEDCI` (calls STEDC with COMPZ='I') `STEGRA` (calls STEGR with RANGE='A') `STEGRI` (calls STEGR with RANGE='I') `STEGRV` (calls STEGR with RANGE='V') `ALL` (performs all tests above)
`DUMP`	*list*	Defines data to be written into files. Possible entries in *list* are: `T` (writes the tridiagonal matrix as triplets $i, t_{i,i}, t_{i,i+1}$ in file *stetester.out.T*) `W` (writes the eigenvalues in file *stetester.out.W*) `Z` (writes the eigenvectors in file *stetester.out.Z*) `LOG` (writes timings, residuals and orthogonality level in file *stetester.out.log*) `T.M` (writes the tridiagonal matrix in Matlab format in file *stetester.out.m*) `W.M` (writes the eigenvalues in Matlab format in file *stetester.out.m*) `Z.M` (writes the eigenvectors in Matlab format in file *stetester.out.m*)
`ECOND`	*int*	Sets the condition number for types 1 to 4 in Table 2.1. Possible values of *int* are: 1, then $k = \frac{1}{\sqrt{ulp}}$, default 2, then $k = \frac{1}{n \times \sqrt{ulp}}$ 3, then $k = \frac{1}{10 \times n \times \sqrt{ulp}}$ 4, then $k = \frac{1}{ulp}$ 5, then $k = \frac{1}{n \times ulp}$ 6, then $k = \frac{1}{10 \times n \times ulp}$
`EDIST`	*int*	Sets the random distribution to be used in type 6 in Table 2.1. Possible values of *int* are: 1, for uniform distribution (-1,1), default 2, for uniform distribution (0,1) 3, for normal distribution (0,1)
`ESIGN`	*int*	Assigns (random) signs to the eigenvalues defined in Table 2.1. Possible values of *int* are: 0, then the eigenvalues will not be negative, default 1, then the eigenvalues can be positive, negative or zero

TABLE A.2
Key words for `stetester`*, part 2.*

Key word	argument	purpose
`EIGVAL`		Defines the built-in eigenvalue distributions to be used in the generation of test matrices. The next two lines must set integers $etype_1 \quad etype_2 \quad etype_3 \quad \ldots$ $esize_1 \quad esize_2 \quad esize_3 \quad \ldots$ where *etype* is a list of types (see Table 2.1) and *esize* is a list of dimensions. A negative *etype* reverses the eigenvalue distribution. For example, *etype*$= -1$ results in $\lambda_i = \frac{1}{k},\ i = 1, 2, \ldots n-1,\ \lambda_n = 1.$
`EIGVALF`	*string*	Defines a file containing an eigenvalue distribution to be used in the generation of a tridiagonal matrix, as discussed in Section 2.1. The file defined by *string* should contain only one entry per line as follows n λ_1 $\vdots$ λ_n
`EIGVI`		Defines indices of the smallest and largest eigenvalues to be computed. The next two lines must define pairs of integers $\mathtt{IL}_1 \ \mathtt{IL}_2 \ \ldots$ $\mathtt{IU}_1 \ \mathtt{IU}_2 \ \ldots$ with $1 \leq \mathtt{IL}_i \leq \mathtt{IU}_i$. These indices are used only in the tests where RANGE=‘I’.
`EIGVV`		Defines lower and upper bounds of intervals to be searched for eigenvalues. The next two lines must define pairs of values $\mathtt{VL}_1 \ \mathtt{VL}_2 \ \ldots$ $\mathtt{VU}_1 \ \mathtt{VU}_2 \ \ldots$ with $\mathrm{VL}_i \leq \mathrm{VU}_i$. These indices are used only in the tests where RANGE=‘V’.
`GLUED`		Defines glued matrices as in (2.2). The next fours lines must set $gform_1 \quad gform_2 \quad \ldots \quad gform_{k-1} \quad gform_k$ $gtype_1 \quad gtype_2 \quad \ldots \quad gtype_{k-1} \quad gtype_k$ $gsize_1 \quad gsize_2 \quad \ldots \quad gsize_{k-1} \quad gsize_k$ $\gamma_1 \quad \gamma_2 \quad \cdots \quad \gamma_{k-1}$ where the integers *gform*, *gtype* and *gsize* define, respectively, how the matrix is generated (1 for built-in eigenvalue distribution, 2 for built-in tridiagonal matrix), its type (accordingly to Tables 2.1 and 2.2) and its dimension. The real value γ (real) is the glue factor.

TABLE A.3
Key words for `stetester`*, part 3.*

Key word	argument	purpose
`HBANDA`	k	Sets the halfbandwidth of the symmetric matrix A to be generated and then tridiagonalized; k must be an integer between 1 and 100, which corresponds to $\max(1,(kn)/100))$ subdiagonals, where n is the dimension of the matrix. By default $k = 100$.
`HBANDR`	k	Sets the halfbandwidth of the matrices used in the tests 3.1 and 3.3; k must be an integer between 0 and 100, then $\max(1,(kn)/100))$ subdiagonals of those matrices are computed. If $k = 0$ the tests are not performed and the corresponding results are simply set to 0. By default, $k = 100$.
`ISEED`	k_1 k_2 k_3 k_4	Sets the (initial) seed of the random number generator. Each (integer) k should lie between 0 and 4095 inclusive and k_4 should be odd. The default is $k_i = 5 - i$.
`MATRIX`		Defines built-in tridiagonal matrices to be used in the tests. The next two lines must set integers $mtype_1$ $mtype_2$ $mtype_3$ ... $msize_1$ $msize_2$ $msize_3$ ... where *mtype* (integer) is a list of built-in tridiagonal matrices (see Table 2.2), and *msize* (integer) is a list of dimensions.
`MATRIXF`	*string*	Defines a file containing a tridiagonal matrix, where *string* is a file name. This file should contain n 1 d_1 e_1 2 d_2 e_2 ⋮ n d_n 0.0 which will then be used to generate a tridiagonal matrix with diagonal entries set to d_i and offdiagonals set to e_i.
`NRILIU`	k	Defines the number of k random indices of the smallest and largest eigenvalues to be computed. These indices are used only in the tests where RANGE='I'.
`NRVLVU`	k	Defines the number of k random lower and upper bounds of intervals to be searched for eigenvalues. These i intervals are used only in the tests where RANGE='V'.
`END`		End of data (subsequent lines are ignored).

TABLE A.4
A sample input file for stetester.

```
%------------------------------------------------------------------
% This is a simple input file for STETESTER.
%------------------------------------------------------------------
%
EIGVAL               % Sets built-in eigenvalue distributions
       3             % Distribution 3, EIG(i)=COND**(-(i-1)/(N-1))
      10  15         % Dimensions of the matrices to be generated
%
MATRIX               % Sets built-in matrices
       2   3         % Matrix type 2 and 3
      20             % Dimension of the matrices to be generated
%
GLUED                % Sets glued matrices
     1     2     1   % If 1, set eigenvalues; if 2, set matrix
     1     2     3   % Eigenvalue distribution or matrix type
    10    11    12   % Dimensions
 0.001 0.002         % Glue factors
%
EIGVALF DATA/T_0010.eig % Eigenvalues read from file 'T10.eig'
%
MATRIXF DATA/T_0010.dat % Matrix read from file 'T10.dat'
%
% Tests to be performed. Note that 'ALL' is equivalent to
%
%   "STEQRV"  (calls STEQR with COMPZ='V'),
%   "STEVXA"  (calls STEVX with RANGE='A'),
%   "STEVXI"  (calls STEVX with RANGE='I'),
%   "STEVXV"  (calls STEVX with RANGE='V'),
%   "STEDCI"  (calls STEDC with COMPZ='I'),
%   "STEGRA"  (calls STEGR with RANGE='A'),
%   "STEGRI"  (calls STEGR with RANGE='I'),
%   "STEGRV"  (calls STEGR with RANGE='V'),
%
% Also note that no interval was specified (by means of EIGVI,
% EIGVV, NRILIU or NRVLVU) so in spite of 'ALL' some tests
% will be skipped.
%
CALLST  ALL
%
% Halfbandwidth of the symmetric matrix to be generated and then
% tridiagonalized. This can save time for big matrices.
%
HBANDA  100
%
% Dump results in different formats (including Matlab)
%
DUMP    LOG  T  W  Z  T.M  W.M  Z.M
%
END
```

TABLE A.5

A sample output file generated by `stetester`. *The data is printed in Matlab format and stored with a name whose trailing part identifies the test that has been executed*

```
% Case:    1 ############################################################
N =    5;
N_001 = N;
D = zeros(N,1); E = zeros(N,1);
D(    1)= 6.364984420732012E-002; E(    1)=-2.438638589637976E-001;
D(    2)= 9.364644735979822E-001; E(    2)=-4.811682261688812E-003;
D(    3)= 1.093556433149412E-002; E(    3)= 3.729709837873370E-005;
D(    4)= 1.218230276935389E-004; E(    4)= 5.319506124657539E-006;
D(    5)= 2.722043635334067E-007; E(    5)= 0.000000000000000E+000;
D_001 = D; E_001 = E; clear D E;
% QR algorithm STEQR(COMPZ=I) ================================================
M =    5;
W = zeros(M,1);
W(    1)= 1.490116120469489E-008; W(    2)= 1.348699152530776E-006;
W(    3)= 1.220703124999231E-004; W(    4)= 1.104854345603982E-002;
W(    5)= 1.000000000000000E+000;
W_001_1 = W; clear W;
Z = zeros(N,M);
Z(    1,    1)= 1.307984066973555E-001; Z(    2,    1)= 3.413911473056844E-002;
Z(    3,    1)= 1.518458390297948E-002; Z(    4,    1)=-4.786418109812126E-002;
Z(    5,    1)= 9.895477483327149E-001; Z(    1,    2)= 9.521524057428332E-001;
Z(    2,    2)= 2.485118884672883E-001; Z(    3,    2)= 1.094543724694096E-001;
Z(    4,    2)=-2.781623695716281E-002; Z(    5,    2)=-1.374541190267801E-001;
Z(    1,    3)= 3.316346817614305E-002; Z(    2,    3)= 8.639251903968444E-003;
Z(    3,    3)= 4.003930825830011E-004; Z(    4,    3)= 9.984606995074429E-001;
Z(    5,    3)= 4.360755587691128E-002; Z(    1,    4)= 1.080661771201757E-001;
Z(    2,    4)= 2.330981518906979E-002; Z(    3,    4)=-9.938646010435163E-001;
Z(    4,    4)=-3.392442840664115E-003; Z(    5,    4)=-1.633388614144078E-006;
Z(    1,    5)=-2.520306703255168E-001; Z(    2,    5)= 9.677077957618335E-001;
Z(    3,    5)=-4.707784724629880E-003; Z(    4,    5)=-1.756081031362012E-007;
Z(    5,    5)=-9.341486344518498E-013;
Z_001_1 = Z; clear Z;
M_001_1 = M; clear M;
% MRRR algorithm STEGR(RANGE=A) ==============================================
M =    5;
W = zeros(M,1);
W(    1)= 1.490116120299405E-008; W(    2)= 1.348699152440647E-006;
W(    3)= 1.220703124999230E-004; W(    4)= 1.104854345603979E-002;
W(    5)= 9.999999999999978E-001;
W_001_6 = W; clear W;
Z = zeros(N,M);
Z(    1,    1)= 1.307984067061942E-001; Z(    2,    1)= 3.413911473287538E-002;
Z(    3,    1)= 1.518458390399541E-002; Z(    4,    1)=-4.786418109837592E-002;
Z(    5,    1)= 9.895477483314390E-001; Z(    1,    2)= 9.521524057416197E-001;
Z(    2,    2)= 2.485118884669718E-001; Z(    3,    2)= 1.094543724692678E-001;
Z(    4,    2)=-2.781623695669255E-002; Z(    5,    2)=-1.374541190359646E-001;
Z(    1,    3)= 3.316346817611779E-002; Z(    2,    3)= 8.639251903961875E-003;
Z(    3,    3)= 4.003930825800720E-004; Z(    4,    3)= 9.984606995074433E-001;
Z(    5,    3)= 4.360755587691132E-002; Z(    1,    4)=-1.080661771201750E-001;
Z(    2,    4)=-2.330981518906962E-002; Z(    3,    4)= 9.938646010435160E-001;
Z(    4,    4)= 3.392442840664119E-003; Z(    5,    4)= 1.633388614144083E-006;
Z(    1,    5)=-2.520306703255170E-001; Z(    2,    5)= 9.677077957618334E-001;
Z(    3,    5)=-4.707784724629883E-003; Z(    4,    5)=-1.756081031362015E-007;
Z(    5,    5)=-9.341486344518528E-013;
Z_001_6 = Z; clear Z;
M_001_6 = M; clear M;
clear N;
```

TABLE A.6
A simple block Lanczos procedure without reorthogonalization; the block size is chosen as one.

```
function [Q,T] = lztd(A,B,s)
%*********************************************************************
%*                                                                   *
%* lztd performs s steps of the single-vector Lanczos algorithm      *
%* ==== for the symmetric eigenvalue problem A*x = lambda*B*x        *
%*                                                                   *
%* Usage:                                                            *
%*    [Q,T] = lztd(A,B,s)                                            *
%* Input arguments:                                                  *
%*    A : the matrix A in A*x = lambda*B*x                           *
%*    B : the matrix B in A*x = lambda*B*x                           *
%*    s : number of steps                                            *
%* Output arguments:                                                 *
%*    Q : basis of vectors                                           *
%*    T : tridiagonal matrix                                         *
%*                                                                   *
%*********************************************************************

n = size(A,1);
[L,U,P] = lu(A);
Q = [ ]; q_jm1 = zeros(n,1); b_j = 0;
r_0 = ones(n,1);
q_j = r_0/(sqrt(r_0'*B*r_0));

for j = 1:s
%.. three-term recurrence .....................................
    r_j = U\(L\(P*(B*q_j)));
    r_j = r_j - q_jm1*b_j';
    a_j = q_j'*B*r_j;
    r_j = r_j - q_j*a_j;
%.. check for invariant subspace ..............................
    b_jp1 = sqrt(r_j'*B*r_j);
    if b_jp1 <= eps*100
       q_jp1 = zeros(n,1);
       b_jp1 = 0;
       display(sprintf('Invariant subspace, quitting'))
       return
    end
%.. normalize r_j and get q_jp1 ...............................
    q_jp1 = r_j/b_jp1;
%.. insert a_j and b_jp1 into T, and q_j into Q ...............
    T(j,j) = a_j; if j>1, T(j-1,j) = b_j; T(j,j-1) = b_j; end
    Q = [Q q_j];
    q_jm1 = q_j;
    q_j = q_jp1;
    b_j = b_jp1;
end

return
```