Chapter 4

LAPACK and the BLAS

4.1 LAPACK

(This section is essentially compiled from the LAPACK User's Guide [1] that is available online from `http://www.netlib.org/lapack/lug/`.)

LAPACK [1] is a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for **L**inear **A**lgebra **PACK**age.

LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

LAPACK contains **driver routines** for solving standard types of problems, **computational routines** to perform a distinct computational task, and **auxiliary routines** to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software developers, so we have documented the Fortran source for these routines with the same level of detail used for the LAPACK routines and driver routines.

Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

LAPACK is designed to give high efficiency on vector processors, high-performance "super-scalar" workstations, and shared memory multiprocessors. It can also be used satisfactorily on all types of scalar machines (PC's, workstations, mainframes). A distributed-memory version of LAPACK, **ScaLAPACK** [2], has been developed for other types of parallel architectures (for example, massively parallel SIMD machines, or distributed memory machines).

LAPACK has been designed to supersede LINPACK [3] and EISPACK [10, 8], principally by restructuring the software to achieve much greater efficiency, where possible, on modern high-performance computers; also by adding extra functionality, by using some new or improved algorithms, and by integrating the two sets of algorithms into a unified package.

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (**BLAS**) [9, 6, 5]. Highly efficient machine-specific implementations of the BLAS are available for many modern

high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code.

The BLAS are not strictly speaking part of LAPACK, but Fortran 77 code for the BLAS is distributed with LAPACK, or can be obtained separately from `netlib` where "model implementations" are found.

The model implementation is not expected to perform as well as a specially tuned implementation on most high-performance computers – on some machines it may give much worse performance – but it allows users to run LAPACK codes on machines that do not offer any other implementation of the BLAS.

The complete LAPACK package or individual routines from LAPACK are freely available from the World Wide Web or by anonymous ftp. The LAPACK homepage can be accessed via the URL `http://www.netlib.org/lapack/`.

4.2 BLAS

By 1976 it was clear that some standardization of basic computer operations on vectors was needed [9]. By then it was already known that coding procedures that worked well on one machine might work very poorly on others. In consequence of these observations, Lawson, Hanson, Kincaid and Krogh proposed a limited set of **B**asic **L**inear **A**lgebra **S**ubprograms (**BLAS**) to be (hopefully) optimized by hardware vendors, implemented in assembly language if necessary, that would form the basis of comprehensive linear algebra packages [9]. These so-called Level 1 BLAS consisted of vector operations and some attendant co-routines. The first major package which used these BLAS kernels was LINPACK [3]. Soon afterward, other major software libraries such as the IMSL library and NAG rewrote portions of their existing codes and structured new routines to use these BLAS. Early in their development, vector computers saw significant optimizations using the BLAS. Soon, however, such machines were clustered together in tight networks and somewhat larger kernels for numerical linear algebra were developed [6, 7] to include matrix-vector operations (Level 2 BLAS). Additionally, FORTRAN compilers were by then optimizing vector operations as efficiently as hand coded Level 1 BLAS. Subsequently, in the late 1980s, distributed memory machines were in production and shared memory machines began to have significant numbers of processors. A further set of matrix-matrix operations was proposed [4] and soon standardized [5] to form a Level 3. The first major package for linear algebra which used the Level 3 BLAS was LAPACK [1] and subsequently a scalable (to large numbers of processors) version was released as ScaLAPACK [2]. Vendors focused on Level 1, Level 2, and Level 3 BLAS which provided an easy route to optimizing LINPACK, then LAPACK. LAPACK not only integrated pre-existing solvers and eigenvalue routines found in EISPACK [10] (which did not use the BLAS) and LINPACK (which used Level 1 BLAS), but incorporated the latest dense and banded linear algebra algorithms available. It also used the Level 3 BLAS which were optimized by much vendor effort. Later, we will illustrate several BLAS routines. Conventions for different BLAS are indicated by

- A **root** operation. For example, `_axpy` for the operation

$$\mathbf{y} := a \cdot \mathbf{x} + \mathbf{y} \tag{4.1}$$

- A prefix (or combination prefix) to indicate the datatype of the operands, for example `saxpy` for **s**ingle precision `_axpy` operation, or `isamax` for the index of the maximum absolute element in an array of type **single**.

- a suffix if there is some qualifier, for example **cdotc** or **cdotu** for conjugated or unconjugated complex dot product, respectively:

$$\texttt{cdotc(n,x,1,y,1)} = \sum_{i=0}^{n-1} x_i \bar{y}_i$$

$$\texttt{cdotu(n,x,1,y,1)} = \sum_{i=0}^{n-1} x_i y_i$$

  where both $\mathbf{x}, \mathbf{y}$ are vectors of complex elements.

Tables 4.1 and 4.2 give the prefix/suffix and root combinations for the BLAS, respectively.

Prefixes:	
S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	DOUBLE COMPLEX
Suffixes:	
U	transpose
C	Hermitian conjugate

Table 4.1: Basic Linear Algebra Subprogram prefix/suffix conventions.

4.2.1 Typical performance numbers for the BLAS

Let us look at typical representations of all three levels of the BLAS, `daxpy`, `ddot`, `dgemv`, and `dgemm`, that perform some basic operations. Additionally, we look at the rank-1 update routine `dger`. An overview on the number of memory accesses and floating point operations is given in Table 4.3. The Level 1 BLAS comprise basic vector operations. A call of one of the Level 1 BLAS thus gives rise to $\mathcal{O}(n)$ floating point operations and $\mathcal{O}(n)$ memory accesses. Here, n is the vector length. The Level 2 BLAS comprise operations that involve matrices *and* vectors. If the involved matrix is n-by-n then both the memory accesses and the floating point operations are of $\mathcal{O}(n^2)$. In contrast, the Level 3 BLAS have a higher order of floating point operations than memory accesses. The most prominent operation of the Level 3 BLAS, matrix-matrix multiplication costs $\mathcal{O}(n^3)$ floating point operations while there are only $\mathcal{O}(n^2)$ reads and writes. The last column in Table 4.3 shows the crucial difference between the Level 3 BLAS and the rest.

Table 4.4 gives some performance numbers for the five BLAS of Table 4.3. Notice that the timer has a resolution of only 1 μsec! Therefore, the numbers in Table 4.4 have been obtained by timing a loop inside of which the respective function is called many times. The Mflop/s rates of the Level 1 BLAS `ddot` and `daxpy` quite precisely reflect the ratios of the memory accesses of the two routines, $2n$ vs. $3n$. The high rates are for vectors that can be held in the on-chip cache of 512 MB. The low 240 and 440 Mflop/s with the very long vectors are related to the memory bandwidth of about 1900 MB/s.

The Level 2 BLAS `dgemv` has about the same performance as `daxpy` if the matrix can be held in cache ($n = 100$). Otherwise it is considerably reduced. `dger` has a high volume of read and write operations, while the number of floating point operations is limited.

Level 1 BLAS	
_rotg, _rot	Generate/apply plane rotation
_rotmg, _rotm	Generate/apply modified plane rotation
_swap	Swap two vectors: $\mathbf{x} \leftrightarrow \mathbf{y}$
_scal	Scale a vector: $\mathbf{x} \leftarrow \alpha\mathbf{x}$
_copy	Copy a vector: $\mathbf{x} \leftarrow \mathbf{y}$
_axpy	_axpy operation: $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$
dot	Dot product: $s \leftarrow \mathbf{x} \cdot \mathbf{y} = \mathbf{x}^*\mathbf{y}$
_nrm2	2-norm: $s \leftarrow \|\mathbf{x}\|_\mathbf{2}$
_asum	1-norm: $s \leftarrow \|\mathbf{x}\|_\mathbf{1}$
i_amax	Index of largest vector element: first i such $\|x_i\| \geq \|x_k\|$ for all k
Level 2 BLAS	
_gemv, _gbmv	General (banded) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
_hemv, _hbmv, _hpmv	Hermitian (banded, packed) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
_semv, _sbmv, _spmv	Symmetric (banded, packed) matrix-vector multiply: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
_trmv, _tbmv, _tpmv	Triangular (banded, packed) matrix-vector multiply: $\mathbf{x} \leftarrow A\mathbf{x}$
_trsv, _tbsv, _tpsv	Triangular (banded, packed) system solves (forward/backward substitution): $\mathbf{x} \leftarrow A^{-1}\mathbf{x}$
_ger, _geru, _gerc	Rank-1 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{y}^* + A$
_her, _hpr, _syr, _spr	Hermitian/symmetric (packed) rank-1 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{x}^* + A$
_her2, _hpr2,_syr2, _spr2	Hermitian/symmetric (packed) rank-2 updates: $A \leftarrow \alpha\mathbf{x}\mathbf{y}^* + \alpha^*\mathbf{y}\mathbf{x}^* + A$
Level 3 BLAS	
_gemm, _symm, _hemm	General/symmetric/Hermitian matrix-matrix multiply: $C \leftarrow \alpha AB + \beta C$
_syrk, _herk	Symmetric/Hermitian rank-k update: $C \leftarrow \alpha AA^* + \beta C$
_syr2k, _her2k	Symmetric/Hermitian rank-k update: $C \leftarrow \alpha AB^* + \alpha^* BA^* + \beta C$
_trmm	Multiple triangular matrix-vector multiplies: $B \leftarrow \alpha AB$
_trsm	Multiple triangular system solves: $B \leftarrow \alpha A^{-1}B$

Table 4.2: Summary of the Basic Linear Algebra Subroutines.

	read	write	flops	flops / mem access
`ddot`	$2n$	1	$2n$	1
`daxpy`	$2n$	n	$2n$	$2/3$
`dgemv`	n^2+n	n	$2n^2$	2
`dger`	n^2+2n	n^2	$2n^2$	1
`dgemm`	$2n^2$	n^2	$2n^3$	$2n/3$

Table 4.3: Number of memory references and floating point operations for vectors of length n.

	$n=100$	500	2'000	10'000'000
`ddot`	1480	1820	1900	440
`daxpy`	1160	1300	1140	240
`dgemv`	1370	740	670	—
`dger`	670	330	320	—
`dgemm`	2680	3470	3720	—

Table 4.4: Some performance numbers for typical BLAS in Mflop/s for a 2.4 GHz Pentium 4.

This leads to a very low performance rate. The Level 3 BLAS `dgemm` performs at a good fraction of the peak performance of the processor (4.8Gflop/s). The performance increases with the problem size. We see from Table 4.3 that the ratio of computation to memory accesses increases with the problem size. This ratio is analogous to a volume to surface area effect.

4.3 Blocking

In the previous section we have seen that it is important to use Level 3 BLAS. However, in the algorithm we have treated so far, there were no blocks. For instance, in the reduction to Hessenberg form we applied Householder (elementary) reflectors from left and right to a matrix to introduce zeros in one of its columns.

The essential point here is to *gather* a number of reflectors to a single block transformation. Let $P_i = I - 2\mathbf{u}_i\mathbf{u}_i^*$, $i = 1,2,3$, be three Householder reflectors. Their product is

$$\begin{aligned} P = P_3P_2P_1 &= (I-2\mathbf{u}_3\mathbf{u}_3^*)(I-2\mathbf{u}_2\mathbf{u}_2^*)(I-2\mathbf{u}_1\mathbf{u}_1^*) \\ &= I - 2\mathbf{u}_3\mathbf{u}_3^* - 2\mathbf{u}_2\mathbf{u}_2^* - 2\mathbf{u}_1\mathbf{u}_1^* + 4\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_2\mathbf{u}_2^* + 4\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_1\mathbf{u}_1^* + 4\mathbf{u}_2\mathbf{u}_2^*\mathbf{u}_1\mathbf{u}_1^* \\ &\quad + 8\mathbf{u}_3\mathbf{u}_3^*\mathbf{u}_2\mathbf{u}_2^*\mathbf{u}_1\mathbf{u}_1^* \\ &= I - [\mathbf{u}_1\mathbf{u}_2\mathbf{u}_3] \begin{bmatrix} 2 & & \\ 4\mathbf{u}_2^*\mathbf{u}_1 & 2 & \\ 4\mathbf{u}_3^*\mathbf{u}_1 + 8(\mathbf{u}_3^*\mathbf{u}_2)(\mathbf{u}_2^*\mathbf{u}_1) & 4\mathbf{u}_3^*\mathbf{u}_2 & 2 \end{bmatrix} [\mathbf{u}_1\mathbf{u}_2\mathbf{u}_3]^*. \end{aligned} \tag{4.2}$$

So, if e.g. three rotations are to be applied on a matrix in blocked fashon, then the three Householder vectors $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ have to be found first. To that end the rotations are first applied only on the first three columns of the matrix, see Fig. 4.1. Then, the blocked rotation is applied to the rest of the matrix.

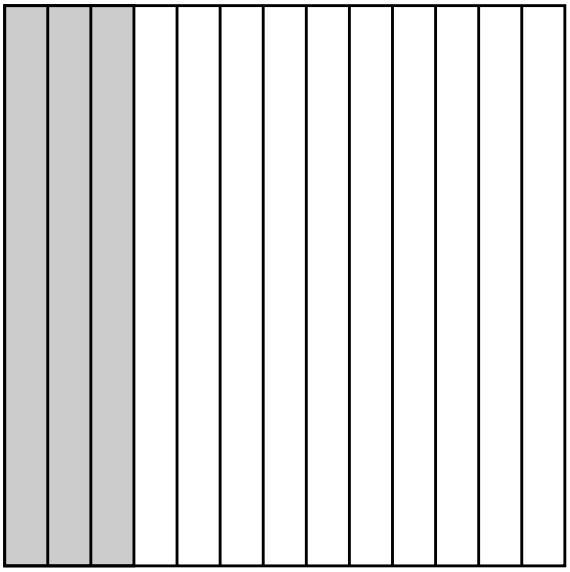

Figure 4.1: Blocking Householder reflections

Remark 4.1. Notice that a similar situation holds for **Gaussian elimination** because

$$\begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & & 1 & & \\ \vdots & & & \ddots & \\ l_{n1} & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & l_{32} & 1 & & \\ & \vdots & & \ddots & \\ & l_{n2} & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & & \ddots & \\ l_{n1} & l_{n2} & & & 1 \end{bmatrix}.$$

However, things are a complicated because of pivoting. □

4.4 LAPACK solvers for the symmetric eigenproblems

To give a feeling how LAPACK is organized we consider solvers for the symmetric eigenproblem (SEP). Except for this problem there are driver routines for linear systems, least squares problems, nonsymmetric eigenvalue problems, the computation of the singular value decomposition (SVD).

The basic task of the symmetric eigenproblem routines is to compute values of λ and, optionally, corresponding vectors $\mathbf{z}$ for a given matrix A.

There are four types of driver routines for symmetric and Hermitian eigenproblems. Originally LAPACK had just the simple and expert drivers described below, and the other two were added after improved algorithms were discovered. Ultimately we expect the algorithm in the most recent driver (called RRR below) to supersede all the others, but in LAPACK 3.0 the other drivers may still be faster on some problems, so we retain them.

- A simple driver computes all the eigenvalues and (optionally) eigenvectors.
- An expert driver computes all or a selected subset of the eigenvalues and (optionally) eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver.

- A divide-and-conquer driver solves the same problem as the simple driver. It is much faster than the simple driver for large matrices, but uses more workspace. The name divide-and-conquer refers to the underlying algorithm.

- A relatively robust representation (RRR) driver computes all or (in a later release) a subset of the eigenvalues, and (optionally) eigenvectors. It is the fastest algorithm of all (except for a few cases), and uses the least workspace. The name RRR refers to the underlying algorithm.

This computation proceeds in the following stages:

1. The real symmetric or complex Hermitian matrix A is reduced to real tridiagonal form T. If A is real symmetric this decomposition is $A = QTQ^T$ with Q orthogonal and T symmetric tridiagonal. If A is complex Hermitian, the decomposition is $A = QTQ^H$ with Q unitary and T, as before, real symmetric tridiagonal.

2. Eigenvalues and eigenvectors of the real symmetric tridiagonal matrix T are computed. If all eigenvalues and eigenvectors are computed, this is equivalent to factorizing T as $T = S\Lambda S^T$, where S is orthogonal and Λ is diagonal. The diagonal entries of Λ are the eigenvalues of T, which are also the eigenvalues of A, and the columns of S are the eigenvectors of T; the eigenvectors of A are the columns of $Z = QS$, so that $A = Z\Lambda Z^T$ ($Z\Lambda Z^H$ when A is complex Hermitian).

In the real case, the decomposition $A = QTQ^T$ is computed by one of the routines `_sytrd`, `_sptrd`, or `_sbtrd`, depending on how the matrix is stored. The complex analogues of these routines are called `_hetrd`, `_hptrd`, and `_hbtrd`. The routine `_sytrd` (or `_hetrd`) represents the matrix Q as a product of elementary reflectors. The routine `_orgtr` (or in the complex case `_unmtr`) is provided to form Q explicitly; this is needed in particular before calling `_steqr` to compute all the eigenvectors of A by the QR algorithm. The routine `_ormtr` (or in the complex case `_unmtr`) is provided to multiply another matrix by Q without forming Q explicitly; this can be used to transform eigenvectors of T computed by `_stein`, back to eigenvectors of A.

For the names of the routines for packed and banded matrices, see [1].

There are several routines for computing eigenvalues and eigenvectors of T, to cover the cases of computing some or all of the eigenvalues, and some or all of the eigenvectors. In addition, some routines run faster in some computing environments or for some matrices than for others. Also, some routines are more accurate than other routines.

`_steqr` This routine uses the implicitly shifted QR algorithm. It switches between the QR and QL variants in order to handle graded matrices. This routine is used to compute all the eigenvalues and eigenvectors.

`_sterf` This routine uses a square-root free version of the QR algorithm, also switching between QR and QL variants, and can only compute all the eigenvalues. This routine is used to compute all the eigenvalues and no eigenvectors.

`_stedc` This routine uses Cuppen's divide and conquer algorithm to find the eigenvalues and the eigenvectors. `_stedc` can be many times faster than `_steqr` for large matrices but needs more work space ($2n^2$ or $3n^2$). This routine is used to compute all the eigenvalues and eigenvectors.

`_stegr` This routine uses the relatively robust representation (RRR) algorithm to find eigenvalues and eigenvectors. This routine uses an LDL^T factorization of a number of

translates $T - \sigma I$ of T, for one shift σ near each cluster of eigenvalues. For each translate the algorithm computes very accurate eigenpairs for the tiny eigenvalues. `_stegr` is faster than all the other routines except in a few cases, and uses the least workspace.

`_stebz` This routine uses bisection to compute some or all of the eigenvalues. Options provide for computing all the eigenvalues in a real interval or all the eigenvalues from the ith to the jth largest. It can be highly accurate, but may be adjusted to run faster if lower accuracy is acceptable.

`_stein` Given accurate eigenvalues, this routine uses inverse iteration to compute some or all of the eigenvectors.

4.5 Generalized Symmetric Definite Eigenproblems (GSEP)

Drivers are provided to compute all the eigenvalues and (optionally) the eigenvectors of the following types of problems:

1. $A\mathbf{z} = \lambda B\mathbf{z}$
2. $AB\mathbf{z} = \lambda\mathbf{z}$
3. $BA\mathbf{z} = \lambda\mathbf{z}$

where A and B are symmetric or Hermitian and B is positive definite. For all these problems the eigenvalues λ are real. The matrices Z of computed eigenvectors satisfy $Z^T AZ = \Lambda$ (problem types 1 and 3) or $Z^{-1}AZ^{-T} = I$ (problem type 2), where Λ is a diagonal matrix with the eigenvalues on the diagonal. Z also satisfies $Z^T BZ = I$ (problem types 1 and 2) or $Z^T B^{-1} Z = I$ (problem type 3).

There are three types of driver routines for generalized symmetric and Hermitian eigenproblems. Originally LAPACK had just the simple and expert drivers described below, and the other one was added after an improved algorithm was discovered.

- a simple driver computes all the eigenvalues and (optionally) eigenvectors.
- an expert driver computes all or a selected subset of the eigenvalues and (optionally) eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver.
- a divide-and-conquer driver solves the same problem as the simple driver. It is much faster than the simple driver for large matrices, but uses more workspace. The name divide-and-conquer refers to the underlying algorithm.

4.6 An example of a LAPACK routines

The double precision subroutine `dsytrd.f` implements the reduction to tridiagonal form. We give it here in full length.

```
      SUBROUTINE DSYTRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO )
*
*  -- LAPACK routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     June 30, 1999
```

```
*
*     .. Scalar Arguments ..
      CHARACTER          UPLO
      INTEGER            INFO, LDA, LWORK, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), D( * ), E( * ), TAU( * ),
     $                   WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  DSYTRD reduces a real symmetric matrix A to real symmetric
*  tridiagonal form T by an orthogonal similarity transformation:
*  Q**T * A * Q = T.
*
*  Arguments
*  =========
*
*  UPLO    (input) CHARACTER*1
*          = 'U':  Upper triangle of A is stored;
*          = 'L':  Lower triangle of A is stored.
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*          N-by-N upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = 'L', the
*          leading N-by-N lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*          On exit, if UPLO = 'U', the diagonal and first superdiagonal
*          of A are overwritten by the corresponding elements of the
*          tridiagonal matrix T, and the elements above the first
*          superdiagonal, with the array TAU, represent the orthogonal
*          matrix Q as a product of elementary reflectors; if UPLO
*          = 'L', the diagonal and first subdiagonal of A are over-
*          written by the corresponding elements of the tridiagonal
*          matrix T, and the elements below the first subdiagonal, with
*          the array TAU, represent the orthogonal matrix Q as a product
*          of elementary reflectors. See Further Details.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  D       (output) DOUBLE PRECISION array, dimension (N)
*          The diagonal elements of the tridiagonal matrix T:
*          D(i) = A(i,i).
*
*  E       (output) DOUBLE PRECISION array, dimension (N-1)
*          The off-diagonal elements of the tridiagonal matrix T:
*          E(i) = A(i,i+1) if UPLO = 'U', E(i) = A(i+1,i) if UPLO = 'L'.
*
*  TAU     (output) DOUBLE PRECISION array, dimension (N-1)
*          The scalar factors of the elementary reflectors (see Further
*          Details).
```

```
*
*  WORK    (workspace/output) DOUBLE PRECISION array, dimension (LWORK)
*          On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
*
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.  LWORK >= 1.
*          For optimum performance LWORK >= N*NB, where NB is the
*          optimal blocksize.
*
*          If LWORK = -1, then a workspace query is assumed; the routine
*          only calculates the optimal size of the WORK array, returns
*          this value as the first entry of the WORK array, and no error
*          message related to LWORK is issued by XERBLA.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*
*  Further Details
*  ===============
*
*  If UPLO = 'U', the matrix Q is represented as a product of elementary
*  reflectors
*
*     Q = H(n-1) . . . H(2) H(1).
*
*  Each H(i) has the form
*
*     H(i) = I - tau * v * v'
*
*  where tau is a real scalar, and v is a real vector with
*  v(i+1:n) = 0 and v(i) = 1; v(1:i-1) is stored on exit in
*  A(1:i-1,i+1), and tau in TAU(i).
*
*  If UPLO = 'L', the matrix Q is represented as a product of elementary
*  reflectors
*
*     Q = H(1) H(2) . . . H(n-1).
*
*  Each H(i) has the form
*
*     H(i) = I - tau * v * v'
*
*  where tau is a real scalar, and v is a real vector with
*  v(1:i) = 0 and v(i+1) = 1; v(i+2:n) is stored on exit in A(i+2:n,i),
*  and tau in TAU(i).
*
*  The contents of A on exit are illustrated by the following examples
*  with n = 5:
*
*  if UPLO = 'U':                       if UPLO = 'L':
*
*    (  d   e   v2  v3  v4 )              (  d                  )
*    (      d   e   v3  v4 )              (  e   d              )
*    (          d   e   v4 )              (  v1  e   d          )
*    (              d   e  )              (  v1  v2  e   d      )
*    (                  d  )              (  v1  v2  v3  e   d  )
*
*  where d and e denote diagonal and off-diagonal elements of T, and vi
*  denotes an element of the vector defining H(i).
```

```
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      LOGICAL            LQUERY, UPPER
      INTEGER            I, IINFO, IWS, J, KK, LDWORK, LWKOPT, NB,
     $                   NBMIN, NX
*     ..
*     .. External Subroutines ..
      EXTERNAL           DLATRD, DSYR2K, DSYTD2, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX
*     ..
*     .. External Functions ..
      LOGICAL            LSAME
      INTEGER            ILAENV
      EXTERNAL           LSAME, ILAENV
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      LQUERY = ( LWORK.EQ.-1 )
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      ELSE IF( LWORK.LT.1 .AND. .NOT.LQUERY ) THEN
         INFO = -9
      END IF
*
      IF( INFO.EQ.0 ) THEN
*
*        Determine the block size.
*
         NB = ILAENV( 1, 'DSYTRD', UPLO, N, -1, -1, -1 )
         LWKOPT = N*NB
         WORK( 1 ) = LWKOPT
      END IF
*
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DSYTRD', -INFO )
         RETURN
      ELSE IF( LQUERY ) THEN
         RETURN
      END IF
*
*     Quick return if possible
*
      IF( N.EQ.0 ) THEN
         WORK( 1 ) = 1
```

```
         RETURN
      END IF
*
      NX = N
      IWS = 1
      IF( NB.GT.1 .AND. NB.LT.N ) THEN
*
*        Determine when to cross over from blocked to unblocked code
*        (last block is always handled by unblocked code).
*
         NX = MAX( NB, ILAENV( 3, 'DSYTRD', UPLO, N, -1, -1, -1 ) )
         IF( NX.LT.N ) THEN
*
*           Determine if workspace is large enough for blocked code.
*
            LDWORK = N
            IWS = LDWORK*NB
            IF( LWORK.LT.IWS ) THEN
*
*              Not enough workspace to use optimal NB:  determine the
*              minimum value of NB, and reduce NB or force use of
*              unblocked code by setting NX = N.
*
               NB = MAX( LWORK / LDWORK, 1 )
               NBMIN = ILAENV( 2, 'DSYTRD', UPLO, N, -1, -1, -1 )
               IF( NB.LT.NBMIN )
     $            NX = N
            END IF
         ELSE
            NX = N
         END IF
      ELSE
         NB = 1
      END IF
*
      IF( UPPER ) THEN
*
*        Reduce the upper triangle of A.
*        Columns 1:kk are handled by the unblocked method.
*
         KK = N - ( ( N-NX+NB-1 ) / NB )*NB
         DO 20 I = N - NB + 1, KK + 1, -NB
*
*           Reduce columns i:i+nb-1 to tridiagonal form and form the
*           matrix W which is needed to update the unreduced part of
*           the matrix
*
            CALL DLATRD( UPLO, I+NB-1, NB, A, LDA, E, TAU, WORK,
     $                   LDWORK )
*
*           Update the unreduced submatrix A(1:i-1,1:i-1), using an
*           update of the form:  A := A - V*W' - W*V'
*
            CALL DSYR2K( UPLO, 'No transpose', I-1, NB, -ONE, A( 1, I ),
     $                   LDA, WORK, LDWORK, ONE, A, LDA )
*
*           Copy superdiagonal elements back into A, and diagonal
*           elements into D
*
            DO 10 J = I, I + NB - 1
```

```
               A( J-1, J ) = E( J-1 )
               D( J ) = A( J, J )
   10       CONTINUE
   20    CONTINUE
*
*        Use unblocked code to reduce the last or only block
*
         CALL DSYTD2( UPLO, KK, A, LDA, D, E, TAU, IINFO )
      ELSE
*
*        Reduce the lower triangle of A
*
         DO 40 I = 1, N - NX, NB
*
*           Reduce columns i:i+nb-1 to tridiagonal form and form the
*           matrix W which is needed to update the unreduced part of
*           the matrix
*
            CALL DLATRD( UPLO, N-I+1, NB, A( I, I ), LDA, E( I ),
     $                   TAU( I ), WORK, LDWORK )
*
*           Update the unreduced submatrix A(i+ib:n,i+ib:n), using
*           an update of the form:  A := A - V*W' - W*V'
*
            CALL DSYR2K( UPLO, 'No transpose', N-I-NB+1, NB, -ONE,
     $                   A( I+NB, I ), LDA, WORK( NB+1 ), LDWORK, ONE,
     $                   A( I+NB, I+NB ), LDA )
*
*           Copy subdiagonal elements back into A, and diagonal
*           elements into D
*
            DO 30 J = I, I + NB - 1
               A( J+1, J ) = E( J )
               D( J ) = A( J, J )
   30       CONTINUE
   40    CONTINUE
*
*        Use unblocked code to reduce the last or only block
*
         CALL DSYTD2( UPLO, N-I+1, A( I, I ), LDA, D( I ), E( I ),
     $                TAU( I ), IINFO )
      END IF
*
      WORK( 1 ) = LWKOPT
      RETURN
*
*     End of DSYTRD
*
      END
```

Notice that most of the lines (indicated by '$*$') contain comments. The initial comment lines also serve as manual pages. Notice that the code only looks at one half (upper or lower triangle) of the symmetric input matrix. The other triangle is used to store the Householder vectors. These are normed such that the first component is one,

$$I - 2\mathbf{u}\mathbf{u}^* = I - 2|u_1|^2(\mathbf{u}/u_1)(\mathbf{u}/u_1)^* = I - \tau\mathbf{v}\mathbf{v}^*.$$

In the main loop of `dsytrd` there is a call to a subroutine `dlatrd` that generates a block reflektor. (The blocksize is NB.) Then the block reflector is applied by the routine

`dsyr2k`.

Directly after the loop there is a call to the 'unblocked `dsytrd`' named `dsytd2` to deal with the first/last few (<NB) rows/columns of the matrix. This excerpt concerns the situation when the upper triangle of the matrix A is stored. In that routine the mentioned loop looks very much the way we derived the formulae.

```
      ELSE
*
*        Reduce the lower triangle of A
*
         DO 20 I = 1, N - 1
*
*           Generate elementary reflector H(i) = I - tau * v * v'
*           to annihilate A(i+2:n,i)
*
            CALL DLARFG( N-I, A( I+1, I ), A( MIN( I+2, N ), I ), 1,
     $                   TAUI )
            E( I ) = A( I+1, I )
*
            IF( TAUI.NE.ZERO ) THEN
*
*              Apply H(i) from both sides to A(i+1:n,i+1:n)
*
               A( I+1, I ) = ONE
*
*              Compute  x := tau * A * v  storing y in TAU(i:n-1)
*
               CALL DSYMV( UPLO, N-I, TAUI, A( I+1, I+1 ), LDA,
     $                     A( I+1, I ), 1, ZERO, TAU( I ), 1 )
*
*              Compute  w := x - 1/2 * tau * (x'*v) * v
*
               ALPHA = -HALF*TAUI*DDOT( N-I, TAU( I ), 1, A( I+1, I ),
     $                  1 )
               CALL DAXPY( N-I, ALPHA, A( I+1, I ), 1, TAU( I ), 1 )
*
*              Apply the transformation as a rank-2 update:
*                 A := A - v * w' - w * v'
*
               CALL DSYR2( UPLO, N-I, -ONE, A( I+1, I ), 1, TAU( I ), 1,
     $                     A( I+1, I+1 ), LDA )
*
               A( I+1, I ) = E( I )
            END IF
            D( I ) = A( I, I )
            TAU( I ) = TAUI
   20    CONTINUE
         D( N ) = A( N, N )
      END IF
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide - Release 2.0*, SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997. (Software and guide are available at URL `http://www.netlib.org/scalapack/`).

[3] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, PA, 1979.

[4] J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, *A proposal for a set of level 3 basic linear algebra subprograms*, ACM SIGNUM Newsletter, 22 (1987).

[5] ——, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of fortran basic linear algebra subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

[7] ——, *An extended set of fortran basic linear algebra subprograms: Model implementation and test programs*, ACM Transactions on Mathematical Software, 14 (1988), pp. 18–32.

[8] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Lecture Notes in Computer Science 51, Springer-Verlag, Berlin, 1977.

[9] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–325.

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

Chapter 5

Cuppen's Divide and Conquer Algorithm

In this chapter we deal with an algorithm that is designed for the efficient solution of the symmetric tridiagonal eigenvalue problem

$$T\mathbf{x} = \lambda\mathbf{x}, \qquad T = \begin{bmatrix} a_1 & b_1 & & \\ b_1 & a_2 & \ddots & \\ & \ddots & \ddots & b_{n-1} \\ & & b_{n-1} & a_n \end{bmatrix}. \tag{5.1}$$

We noticed from Table 3.1 that the reduction of a full symmetric matrix to a similar tridiagonal matrix requires about $\frac{8}{3}n^3$ while the tridiagonal QR algorithm needs an estimated $6n^3$ floating operations (flops) to converge. Because of the importance of this subproblem a considerable effort has been put into finding faster algorithms than the QR algorithms to solve the tridiagonal eigenvalue problem. In the mid-1980's Dongarra and Sorensen [4] promoted an algorithm originally proposed by Cuppen [2]. This algorithm was based on a divide and conquer strategy. However, it took ten more years until a stable variant was found by Gu and Eisenstat [5, 6]. Today, a stable implementation of this latter algorithm is available in LAPACK [1].

5.1 The divide and conquer idea

Divide and conquer is an old strategy in military to defeat an enemy going back at least to Caesar. In computer science, divide and conquer (D&C) is an important algorithm design paradigm. It works by recursively breaking down a problem into two or more subproblems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Translated to our problem the strategy becomes

1. Partition the tridiagonal eigenvalue problem into two (or more) smaller eigenvalue problems.
2. Solve the two smaller problems.
3. Combine the solutions of the smaller problems to get the desired solution of the overall problem.

Evidently, this strategy can be applied recursively.

5.2 Partitioning the tridiagonal matrix

Partitioning the *irreducible* tridiagonal matrix is done in the following way. We write

(5.2)

$$
T = \left[\begin{array}{cccc|cccc}
a_1 & b_1 & & & & & & \\
b_1 & a_2 & \ddots & & & & & \\
 & \ddots & \ddots & b_{m-1} & & & & \\
 & & b_{m-1} & a_m & b_m & & & \\
\hline
 & & & b_m & a_{m+1} & b_{m+1} & & \\
 & & & & b_{m+1} & a_{m+2} & \ddots & \\
 & & & & & \ddots & \ddots & b_{n-1} \\
 & & & & & & b_{n-1} & a_n
\end{array}\right]
$$

$$
= \left[\begin{array}{cccc|cccc}
a_1 & b_1 & & & & & & \\
b_1 & a_2 & \ddots & & & & & \\
 & \ddots & \ddots & b_{m-1} & & & & \\
 & & b_{m-1} & a_m - \mp b_m & & & & \\
\hline
 & & & & a_{m+1} - \mp b_m & b_{m+1} & & \\
 & & & & b_{m+1} & a_{m+2} & \ddots & \\
 & & & & & \ddots & \ddots & b_{n-1} \\
 & & & & & & b_{n-1} & a_n
\end{array}\right]
+ \left[\begin{array}{cc|cc}
 & & & \\
 & \pm b_m & b_m & \\
\hline
 & b_m & \pm b_m & \\
 & & &
\end{array}\right]
$$

$$
= \left[\begin{array}{c|c} T_1 & \\ \hline & T_2 \end{array}\right] + \rho \mathbf{u}\mathbf{u}^T \qquad \text{with } \mathbf{u} = \left[\begin{array}{c} \pm \mathbf{e}_m \\ \hline \mathbf{e}_1 \end{array}\right] \text{ and } \rho = \pm b_m,
$$

where $\mathbf{e}_m$ is a vector of length $m \approx \frac{n}{2}$ and $\mathbf{e}_1$ is a vector of length $n - m$. Notice that the most straightforward way to partition the problem without modifying the diagonal elements leads to a rank-two modification. With the approach of (5.2) we have the original T as a sum of two smaller tridiagonal systems plus a *rank-one modification*.

5.3 Solving the small systems

We solve the half-sized eigenvalue problems,

$$
T_i = Q_i \Lambda_i Q_i^T, \quad Q_i^T Q_i = I, \qquad i = 1, 2. \tag{5.3}
$$

These two spectral decompositions can be computed by any algorithm, in particular also by this divide and conquer algorithm by which the T_i would be further split. It is clear that by this partitioning an large number of small problems can be generated that can be potentially solved in parallel. For a parallel algorithm, however, the further phases of the algorithm must be parallelizable as well.

Plugging (5.3) into (5.2) gives

$$
\left[\begin{array}{c|c} Q_1^T & \\ \hline & Q_2^T \end{array}\right] \left(\left[\begin{array}{c|c} T_1 & \\ \hline & T_2 \end{array}\right] + \rho \mathbf{u}\mathbf{u}^T \right) \left[\begin{array}{c|c} Q_1 & \\ \hline & Q_2 \end{array}\right] = \left[\begin{array}{c|c} \Lambda_1 & \\ \hline & \Lambda_2 \end{array}\right] + \rho \mathbf{v}\mathbf{v}^T \tag{5.4}
$$

with

$$
\mathbf{v} = \left[\begin{array}{c|c} Q_1^T & \\ & Q_2^T \end{array}\right] \mathbf{u} = \left[\begin{array}{c} \pm Q_1^T \mathbf{e}_m \\ Q_2^T \mathbf{e}_1 \end{array}\right] = \left[\begin{array}{c} \pm \text{last row of } Q_1 \\ \text{first row of } Q_2 \end{array}\right]. \tag{5.5}
$$

Now we have arrived at the eigenvalue problem

$$(D + \rho\mathbf{v}\mathbf{v}^T)\mathbf{x} = \lambda\mathbf{x}, \qquad D = \Lambda_1 \oplus \Lambda_2 = \mathrm{diag}(\lambda_1, \ldots, \lambda_n). \tag{5.6}$$

That is, we have to compute the spectral decomposition of a matrix that is a **diagonal plus a rank-one update**. Let

$$D + \rho\mathbf{v}\mathbf{v}^T = Q\Lambda Q^T \tag{5.7}$$

be this spectral decomposition. Then, the spectral decomposition of the tridiagonal T is

$$T = \left[\begin{array}{c|c} Q_1 & \\ & Q_2 \end{array}\right] Q\Lambda Q^T \left[\begin{array}{c|c} Q_1^T & \\ & Q_2^T \end{array}\right]. \tag{5.8}$$

Forming the product $(Q_1 \oplus Q_2)Q$ will turn out to be *the most expensive step of the algorithm*. It costs $n^3 + \mathcal{O}(n^2)$ floating point operations

5.4 Deflation

There are certain solutions of (5.7) that can be given immediately, by just looking carefully at the equation.

If there are zero entries in $\mathbf{v}$ then we have

$$\big(v_i = 0 \Leftrightarrow \mathbf{v}^T\mathbf{e}_i = 0\big) \quad \Longrightarrow \quad (D + \rho\mathbf{v}\mathbf{v}^T)\mathbf{e}_i = d_i\mathbf{e}_i. \tag{5.9}$$

Thus, if an entry of $\mathbf{v}$ vanishes we can read the eigenvalue from the diagonal of D at once and the corresponding eigenvector is a coordinate vector.

If identical entries occur in the diagonal of D, say $d_i = d_j$, with $i < j$, then we can find a plane rotation $G(i, j, \phi)$ (see (3.4)) such that it introduces a zero into the j-th position of $\mathbf{v}$,

$$G^T\mathbf{v} = G(i, j, \varphi)^T\mathbf{v} = \begin{bmatrix} \times \\ \vdots \\ \sqrt{{v_i}^2 + {v_j}^2} \\ \vdots \\ 0 \\ \vdots \\ \times \end{bmatrix} \begin{array}{l} \\ \\ \leftarrow\ i \\ \\ \leftarrow\ j \\ \\ \\ \end{array}$$

Notice, that (for *any* φ),

$$G(i, j, \varphi)^T D G(i, j, \varphi) = D, \qquad d_i = d_j.$$

So, if there are multiple eigenvalues in D we can reduce all but one of them by introducing zeros in $\mathbf{v}$ and then proceed as previously in (5.9).

When working with floating point numbers we deflate if

$$|v_i| < C\varepsilon\|T\| \qquad \text{or} \qquad |d_i - d_j| < C\varepsilon\|T\|, \qquad (\|T\| = \|D + \rho\mathbf{v}\mathbf{v}^T\|) \tag{5.10}$$

where C is a small constant. Deflation changes the eigenvalue problem for $D + \rho\mathbf{v}\mathbf{v}^T$ into the eigenvalue problem for

$$\left[\begin{array}{cc} D_1 + \rho\mathbf{v}_1\mathbf{v}_1^T & O \\ O & D_2 \end{array}\right] = G^T(D + \rho\mathbf{v}\mathbf{v}^T)G + E, \qquad \|E\| < C\varepsilon\sqrt{\|D\|^2 + |\rho|^2\|\mathbf{v}\|^4}, \tag{5.11}$$

where D_1 has no multiple diagonal entries and $\mathbf{v}_1$ has no zero entries. So, we have to compute the spectral decomposition of the matrix in (5.11) which is similar to a slight perturbation of the original matrix. G is the product of Givens rotations.

5.4.1 Numerical examples

Let us first consider

$$T = \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 3 & 1 & & \\ \hline & & 1 & 4 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 0 & & \\ \hline & & 0 & 3 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] + \left[\begin{array}{ccc|ccc} 0 & & & & & \\ & 0 & & & & \\ & & 1 & 1 & & \\ \hline & & 1 & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 0 & & \\ \hline & & 0 & 3 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & 6 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right]^T = T_0 + \mathbf{u}\mathbf{u}^T.$$

Then a little MATLAB experiment shows that

$$Q_0^T T Q_0 = \begin{bmatrix} 0.1981 & & & & & & \\ & 1.5550 & & & & & \\ & & 3.2470 & & & & \\ & & & 2.5395 & & & \\ & & & & 4.7609 & & \\ & & & & & 6.6996 \end{bmatrix} + \begin{bmatrix} 0.3280 \\ 0.7370 \\ 0.5910 \\ 0.9018 \\ -0.4042 \\ 0.1531 \end{bmatrix} \begin{bmatrix} 0.3280 \\ 0.7370 \\ 0.5910 \\ 0.9018 \\ -0.4042 \\ 0.1531 \end{bmatrix}^T$$

with

$$Q_0 = \begin{bmatrix} 0.7370 & -0.5910 & 0.3280 & & & \\ -0.5910 & -0.3280 & 0.7370 & & & \\ 0.3280 & 0.7370 & 0.5910 & & & \\ & & & 0.9018 & -0.4153 & 0.1200 \\ & & & -0.4042 & -0.7118 & 0.5744 \\ & & & 0.1531 & 0.5665 & 0.8097 \end{bmatrix}$$

Here it is not possible to deflate.

Let us now look at an example with more symmetry,

$$T = \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & 1 & & \\ \hline & & 1 & 2 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] = \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 1 & 0 & & \\ \hline & & 0 & 1 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] + \left[\begin{array}{ccc|ccc} 0 & & & & & \\ & 0 & & & & \\ & & 1 & 1 & & \\ \hline & & 1 & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 2 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 1 & 0 & & \\ \hline & & 0 & 1 & 1 & \\ & & & 1 & 2 & 1 \\ & & & & 1 & 2 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right]^T = T_0 + \mathbf{u}\mathbf{u}^T.$$

Now, MATLAB gives

$$Q_0^T T Q_0 = \begin{bmatrix} 0.1981 & & & & & & \\ & 1.5550 & & & & & \\ & & 3.2470 & & & & \\ & & & 0.1981 & & & \\ & & & & 1.5550 & & \\ & & & & & 3.2470 \end{bmatrix} + \begin{bmatrix} 0.7370 \\ -0.5910 \\ 0.3280 \\ 0.7370 \\ -0.5910 \\ 0.3280 \end{bmatrix} \begin{bmatrix} 0.7370 \\ -0.5910 \\ 0.3280 \\ 0.7370 \\ -0.5910 \\ 0.3280 \end{bmatrix}^T$$

with

$$Q_0 = \begin{bmatrix} 0.3280 & 0.7370 & 0.5910 & & & \\ -0.5910 & -0.3280 & 0.7370 & & & \\ 0.7370 & -0.5910 & 0.3280 & & & \\ & & & 0.7370 & -0.5910 & 0.3280 \\ & & & -0.5910 & -0.3280 & 0.7370 \\ & & & 0.3280 & 0.7370 & 0.5910 \end{bmatrix}$$

In this example we have *three* double eigenvalues. Because the corresponding components of $\mathbf{v}$ (v_i and v_{i+1}) are equal we define

$$\begin{aligned} G &= G(1,4,\pi/4)G(2,5,\pi/4)G(3,6,\pi/4) \\ &= \left[\begin{array}{ccc|ccc} 0.7071 & & & 0.7071 & & \\ & 0.7071 & & & 0.7071 & \\ & & 0.7071 & & & 0.7071 \\ \hline -0.7071 & & & 0.7071 & & \\ & -0.7071 & & & 0.7071 & \\ & & -0.7071 & & & 0.7071 \end{array}\right]. \end{aligned}$$

Then,

$$\begin{aligned} G^T Q_0^T T Q_0 G &= G^T Q_0^T T_0 Q_0 G + G^T \mathbf{v}(G^T\mathbf{v})^T = D + G^T\mathbf{v}(G^T\mathbf{v})^T \\ &= \begin{bmatrix} 0.1981 & & & & & & \\ & 1.5550 & & & & & \\ & & 3.2470 & & & & \\ & & & 0.1981 & & & \\ & & & & 1.5550 & & \\ & & & & & 3.2470 \end{bmatrix} + \begin{bmatrix} 1.0422 \\ -0.8358 \\ 0.4638 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix} \begin{bmatrix} 1.0422 \\ -0.8358 \\ 0.4638 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix}^T \end{aligned}$$

Therefore, (in this example) $\mathbf{e}_4$, $\mathbf{e}_5$, and $\mathbf{e}_6$ are eigenvectors of

$$D + G^T\mathbf{v}(G^T\mathbf{v})^T = D + G^T\mathbf{v}\mathbf{v}^T G$$

corresponding to the eigenvalues d_4, d_5, and d_6, respectively. The eigenvectors of T corresponding to these three eigenvalues are the last three columns of

$$Q_0 G = \begin{bmatrix} 0.2319 & -0.4179 & 0.5211 & 0.5211 & -0.4179 & 0.2319 \\ 0.5211 & -0.2319 & -0.4179 & -0.4179 & -0.2319 & 0.5211 \\ 0.4179 & 0.5211 & 0.2319 & 0.2319 & 0.5211 & 0.4179 \\ -0.2319 & 0.4179 & -0.5211 & 0.5211 & -0.4179 & 0.2319 \\ -0.5211 & 0.2319 & 0.4179 & -0.4179 & -0.2319 & 0.5211 \\ -0.4179 & -0.5211 & -0.2319 & 0.2319 & 0.5211 & 0.4179 \end{bmatrix}.$$

5.5 The eigenvalue problem for $D + \rho\mathbf{v}\mathbf{v}^T$

We know that $\rho \neq 0$. Otherwise there is nothing to be done. Furthermore, after deflation, we know that all elements of $\mathbf{v}$ are nonzero and that the diagonal elements of D are all

distinct, in fact,

$$|d_i - d_j| > C\varepsilon\|T\|.$$

We order the diagonal elements of D such that

$$d_1 < d_2 < \cdots < d_n.$$

Notice that this procedure permutes the elements of $\mathbf{v}$ as well. Let $(\lambda, \mathbf{x})$ be an eigenpair of

$$(D + \rho\mathbf{v}\mathbf{v}^T)\mathbf{x} = \lambda\mathbf{x}. \tag{5.12}$$

Then,

$$(D - \lambda I)\mathbf{x} = -\rho\mathbf{v}\mathbf{v}^T\mathbf{x}. \tag{5.13}$$

λ cannot be equal to one of the d_i. If $\lambda = d_k$ then the k-th element on the left of (5.13) vanishes. But then either $v_k = 0$ or $\mathbf{v}^T\mathbf{x} = 0$. The first cannot be true for our assumption about $\mathbf{v}$. If on the other hand $\mathbf{v}^T\mathbf{x} = 0$ then $(D - d_k I)\mathbf{x} = \mathbf{0}$. Thus $\mathbf{x} = \mathbf{e}_k$ and $\mathbf{v}^T\mathbf{e}_k = v_k = 0$, which cannot be true. Therefore $D - \lambda I$ is nonsingular and

$$\mathbf{x} = \rho(\lambda I - D)^{-1}\mathbf{v}(\mathbf{v}^T\mathbf{x}). \tag{5.14}$$

This equation shows that $\mathbf{x}$ is proportional to $(\lambda I - D)^{-1}\mathbf{v}$. If we require $\|\mathbf{x}\| = 1$ then

$$\mathbf{x} = \frac{(\lambda I - D)^{-1}\mathbf{v}}{\|(\lambda I - D)^{-1}\mathbf{v}\|}. \tag{5.15}$$

Multiplying (5.14) by $\mathbf{v}^T$ from the left we get

$$\mathbf{v}^T\mathbf{x} = \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v}(\mathbf{v}^T\mathbf{x}). \tag{5.16}$$

Since $\mathbf{v}^T\mathbf{x} \neq 0$, λ is an eigenvalue of (5.12) if and only if

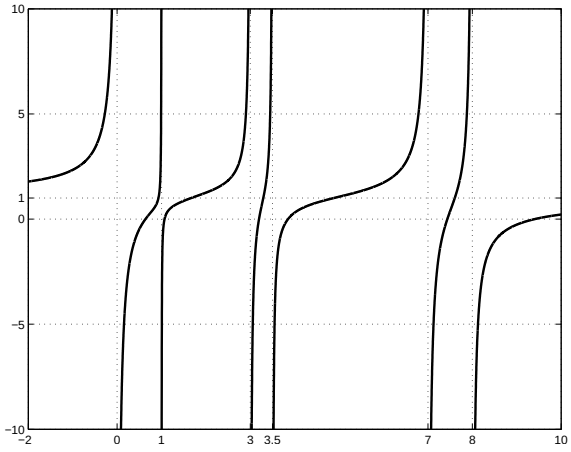

Figure 5.1: Graph of $1 + \frac{1}{0-\lambda} + \frac{0.2^2}{1-\lambda} + \frac{0.6^2}{3-\lambda} + \frac{0.5^2}{3.5-\lambda} + \frac{0.9^2}{7-\lambda} + \frac{0.8^2}{8-\lambda}$

$$\boxed{f(\lambda) := 1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v} = 1 - \rho\sum_{k=1}^{n}\frac{v_k^2}{\lambda - d_k} = 0.} \tag{5.17}$$

This equation is called **secular equation**. The secular equation has **poles** at the eigenvalues of D and **zeros** at the eigenvalues of $D + \rho\mathbf{v}\mathbf{v}^T$. Notice that

$$f'(\lambda) = \rho \sum_{k=1}^{n} \frac{v_k^2}{(\lambda - d_k)^2}.$$

Thus, the derivative of f is positive if $\rho > 0$ wherever it has a finite value. If $\rho < 0$ the derivative of f is negative (almost) everywhere. A typical graph of f with $\rho > 0$ is depicted in Fig. 5.1. (If ρ is negative the image can be flipped left to right.) The secular equation implies the **interlacing property** of the eigenvalues of D and of $D + \rho\mathbf{v}\mathbf{v}^T$,

$$d_1 < \lambda_1 < d_2 < \lambda_2 < \cdots < d_n < \lambda_n, \qquad \rho > 0. \tag{5.18}$$

or

$$\lambda_1 < d_1 < \lambda_2 < d_2 < \cdots < \lambda_n < d_n, \qquad \rho < 0. \tag{5.19}$$

So, we have to compute one eigenvalue in each of the intervals (d_i, d_{i+1}), $1 \le i < n$, and a further eigenvalue in (d_n, ∞) or $(-\infty, d_1)$. The corresponding eigenvector is then given by (5.15). Evidently, these tasks are easy to parallelize.

Equations (5.17) and (5.15) can also been obtained from the relations

$$\begin{aligned}
\begin{bmatrix} \frac{1}{\rho} & \mathbf{v}^T \\ \mathbf{v} & \lambda I - D \end{bmatrix} &= \begin{bmatrix} 1 & \mathbf{0}^T \\ \rho\mathbf{v} & I \end{bmatrix} \begin{bmatrix} \frac{1}{\rho} & \mathbf{0}^T \\ \mathbf{0} & \lambda I - D - \rho\mathbf{v}\mathbf{v}^T \end{bmatrix} \begin{bmatrix} 1 & \rho\mathbf{v}^T \\ \mathbf{0} & I \end{bmatrix} \\
&= \begin{bmatrix} 1 & \mathbf{v}^T(\lambda I - D)^{-1} \\ \mathbf{0} & I \end{bmatrix} \begin{bmatrix} \frac{1}{\rho} - \mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v} & \mathbf{0}^T \\ \mathbf{0} & \lambda I - D \end{bmatrix} \begin{bmatrix} 1 & \mathbf{v}^T \\ (\lambda I - D)^{-1}\mathbf{v} & I \end{bmatrix}.
\end{aligned}$$

These are simply block LDL^T factorizations of the first matrix. The first is the well-known one where the factorization is started with the $(1,1)$ block. The second is a 'backward' factorization that is started with the $(2,2)$ block. Because the determinants of the tridiagonal matrices are all unity, we have

$$\frac{1}{\rho}\det(\lambda I - D - \rho\mathbf{v}\mathbf{v}^T) = \frac{1}{\rho}(1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v})\det(\lambda I - D). \tag{5.20}$$

Denoting the eigenvalues of $D + \rho\mathbf{v}\mathbf{v}^T$ again by $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ this implies

$$\begin{aligned}
\prod_{j=1}^{n}(\lambda - \lambda_j) &= (1 - \rho\mathbf{v}^T(\lambda I - D)^{-1}\mathbf{v}) \prod_{j=1}^{n}(\lambda - d_j) \\
&= \left(1 - \rho \sum_{k=1}^{n} \frac{v_k^2}{\lambda - d_k}\right) \prod_{j=1}^{n}(\lambda - d_j) \\
&= \prod_{j=1}^{n}(\lambda - d_j) - \rho \sum_{k=1}^{n} v_k^2 \prod_{j \ne k}(\lambda - d_j)
\end{aligned} \tag{5.21}$$

Setting $\lambda = d_k$ gives

$$\prod_{j=1}^{n}(d_k - \lambda_j) = -\rho v_k^2 \prod_{\substack{j=1 \\ j \ne i}}^{n}(d_k - d_j) \tag{5.22}$$

or

$$
\begin{aligned}
v_k^2 &= \frac{-1}{\rho}\frac{\prod_{j=1}^{n}(d_k-\lambda_j)}{\prod_{\substack{j=1\\ j\neq i}}^{n}(d_k-d_j)} = \frac{-1}{\rho}\frac{\prod_{j=1}^{k-1}(d_k-\lambda_j)\prod_{j=k}^{n}(\lambda_j-d_k)(-1)^{n-k+1}}{\prod_{j=1}^{k-1}(d_k-d_j)\prod_{j=k+1}^{n}(d_j-d_k)(-1)^{n-k}} \\
&= \frac{1}{\rho}\frac{\prod_{j=1}^{k-1}(d_k-\lambda_j)}{\prod_{j=1}^{k-1}(d_k-d_j)}\frac{\prod_{j=k}^{n}(\lambda_j-d_k)}{\prod_{j=k+1}^{n}(d_j-d_k)} > 0.
\end{aligned} \tag{5.23}
$$

Therefore, the quantity on the right side is positive, so

$$
v_k = \sqrt{\frac{\prod_{j=1}^{k-1}(d_k-\lambda_j)\prod_{j=k}^{n}(\lambda_j-d_k)}{\rho\prod_{j=1}^{k-1}(d_k-d_j)\prod_{j=k+1}^{n}(d_j-d_k)}}. \tag{5.24}
$$

(Similar arguments hold if $\rho < 0$.) Thus, we have the solution of the following **inverse eigenvalue problem**:

> Given $D = \mathrm{diag}(d_1, \ldots, d_n)$ and values $\lambda_1, \ldots, \lambda_n$ that satisfy (5.18). Find a vector $\mathbf{v} = [v_1, \ldots, v_n]^T$ with positive components v_k such that the matrix $D + \mathbf{v}\mathbf{v}^T$ has the *prescribed* eigenvalues $\lambda_1, \ldots, \lambda_n$.

The solution is given by (5.24). The positivity of the v_k makes the solution unique.

5.6 Solving the secular equation

In this section we follow closely the exposition of Demmel [3]. We consider the computation of the zero of $f(\lambda)$ in the interval (d_i, d_{i+1}). We assume that $\rho = 1$.

We may simply apply Newton's iteration to solve $f(\lambda) = 0$. However, if we look carefully at Fig. 5.1 then we notice that the tangent at certain points in (d_i, d_{i+1}) crosses the real axis outside this interval. This happens in particular if the weights v_i or v_{i+1} are small. Therefore that zero finder has to be adapted in such a way that it captures the poles at the interval endpoints. It is relatively straightforward to try the ansatz

$$
h(\lambda) = \frac{c_1}{d_i - \lambda} + \frac{c_2}{d_{i+1} - \lambda} + c_3. \tag{5.25}
$$

Notice that, given the coefficients c_1, c_2, and c_3, the equation $h(\lambda) = 0$ can easily be solved by means of the equivalent quadratic equation

$$
c_1(d_{i+1} - \lambda) + c_2(d_i - \lambda) + c_3(d_i - \lambda)(d_{i+1} - \lambda) = 0. \tag{5.26}
$$

This equation has two zeros. Precisely one of them is inside (d_i, d_{i+1}).

The coefficients c_1, c_2, and c_3 are computed in the following way. Let us assume that we have available an approximation λ_j to the zero in (d_i, d_{i+1}). We request that $h(\lambda_j) = f(\lambda_j)$ and $h'(\lambda_j) = f'(\lambda_j)$. The exact procedure is as follows. We write

$$
f(\lambda) = 1 + \underbrace{\sum_{k=1}^{i}\frac{v_k^2}{d_k - \lambda}}_{\psi_1(\lambda)} + \underbrace{\sum_{k=i+1}^{n}\frac{v_k^2}{d_k - \lambda}}_{\psi_2(\lambda)} = 1 + \psi_1(\lambda) + \psi_2(\lambda). \tag{5.27}
$$

$\psi_1(\lambda)$ is a sum of positive terms and $\psi_2(\lambda)$ is a sum of negative terms. Both $\psi_1(\lambda)$ and $\psi_2(\lambda)$ can be computed accurately, whereas adding them would likely provoke cancellation and loss of relative accuracy. We now choose c_1 and $\hat{c}_1$ such that

$$h_1(\lambda) := \hat{c}_1 + \frac{c_1}{d_i - \lambda} \text{ satisfies } h_1(\lambda_j) = \psi_1(\lambda_j) \text{ and } h_1'(\lambda_j) = \psi_1'(\lambda_j). \tag{5.28}$$

This means that the graphs of h_1 and of ψ_1 are tangent at $\lambda = \lambda_j$. This is similar to Newton's method. However in Newton's method a straight line is fitted to the given function. The coefficients in (5.28) are given by

$$\begin{aligned} c_1 &= \psi_1'(\lambda_j)(d_i - \lambda_j)^2 > 0, \\ \hat{c}_1 &= \psi_1(\lambda_j) - \psi_1'(\lambda_j)(d_i - \lambda_j) = \sum_{k=1}^{i} v_k^2 \frac{d_k - d_i}{(d_k - \lambda_j)^2} \leq 0. \end{aligned}$$

Similarly, the two constants c_2 and $\hat{c}_2$ are determined such that

$$h_2(\lambda) := \hat{c}_2 + \frac{c_2}{d_{i+1} - \lambda} \text{ satisfies } h_2(\lambda_j) = \psi_2(\lambda_j) \text{ and } h_2'(\lambda_j) = \psi_2'(\lambda_j) \tag{5.29}$$

with the coefficients

$$\begin{aligned} c_2 &= \psi_2'(\lambda_j)(d_{i+1} - \lambda_j)^2 > 0, \\ \hat{c}_2 &= \psi_2(\lambda_j) - \psi_2'(\lambda_j)(d_{i+1} - \lambda_j) = \sum_{k=i+1}^{n} v_k^2 \frac{d_k - d_{i+1}}{(d_k - \lambda)^2} \geq 0. \end{aligned}$$

Finally, we set

$$h(\lambda) = 1 + h_1(\lambda) + h_2(\lambda) = \underbrace{(1 + \hat{c}_1 + \hat{c}_2)}_{c_3} + \frac{c_1}{d_i - \lambda} + \frac{c_2}{d_{i+1} - \lambda}. \tag{5.30}$$

This zerofinder is converging quadratically to the desired zero [7]. Usually 2 to 3 steps are sufficient to get the zero to machine precision. Therefore finding a zero only requires $\mathcal{O}(n)$ flops. Thus, finding all zeros costs $\mathcal{O}(n^2)$ floating point operations.

5.7 A first algorithm

We are now ready to give the divide and conquer algorithm, see Algorithm 5.1.
All steps except step 10 require $\mathcal{O}(n^2)$ operations to complete. The step 10 costs n^3 flops. Thus, the full divide and conquer algorithm, requires

$$\begin{aligned} T(n) &= n^3 + 2 \cdot T(n/2) = n^3 + 2\left(\frac{n}{2}\right)^3 + 4T(n/4) \\ &= n^3 + \frac{n^3}{4} + 4\left(\frac{n}{4}\right)^3 + 8T(n/8) = \cdots = \frac{4}{3}n^3. \end{aligned} \tag{5.31}$$

This *serial* complexity of the algorithm very often overestimates the computational costs of the algorithm due to significant deflation that is observed surprisingly often.

Algorithm 5.1 The tridiagonal divide and conquer algorithm

1: Let $T \in \mathbb{C}^{n\times n}$ be a real symmetric tridiagonal matrix. This algorithm computes the spectral decomposition of $T = Q\Lambda Q^T$, where the diagonal Λ is the matrix of eigenvalues and Q is orthogonal.
2: **if** T is 1×1 **then**
3: **return** $(\Lambda = T; Q = 1)$
4: **else**
5: Partition $T = \begin{bmatrix} T_1 & O \\ O & T_2 \end{bmatrix} + \rho\mathbf{u}\mathbf{u}^T$ according to (5.2)
6: Call this algorithm with T_1 as input and Q_1, Λ_1 as output.
7: Call this algorithm with T_2 as input and Q_2, Λ_2 as output.
8: Form $D + \rho\mathbf{v}\mathbf{v}^T$ from $\Lambda_1, \Lambda_2, Q_1, Q_2$ according to (5.4)–(5.6).
9: Find the eigenvalues Λ and the eigenvectors Q' of $D + \rho\mathbf{v}\mathbf{v}^T$.
10: Form $Q = \begin{bmatrix} Q_1 & O \\ O & Q_2 \end{bmatrix} \cdot Q'$ which are the eigenvectors of T.
11: **return** $(\Lambda; Q)$
12: **end if**

5.7.1 A numerical example

Let A be a 4×4 matrix

$$A = D + \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 0 & & & \\ & 2-\beta & & \\ & & 2+\beta & \\ & & & 5 \end{bmatrix} + \begin{bmatrix} 1 \\ \beta \\ \beta \\ 1 \end{bmatrix} \begin{bmatrix} 1 & \beta & \beta & 1 \end{bmatrix}. \tag{5.32}$$

In this example (that is similar to one in [8]) we want to point at a problem that the divide and conquer algorithm possesses as it is given in Algorithm 5.1, namely the loss of orthogonality among eigenvectors.

Before we do some MATLAB tests let us look more closely at D and $\mathbf{v}$ in (5.32). This example becomes difficult to solve if β gets very small. In Figures 5.2 to 5.5 we see graphs of the function $f_\beta(\lambda)$ that appears in the secular equation for $\beta = 1$, $\beta = 0.1$, and $\beta = 0.01$. The critical zeros move towards 2 from both sides. The weights $v_2^2 = v_3^2 = \beta^2$ are however not so small that they should be deflated.

The following MATLAB code shows the problem. We execute the commands for $\beta = 10^{-k}$ for $k = 0, 1, 2, 4, 8$.

```
v = [1 beta beta 1]';          % rank-1 modification
d = [0, 2-beta, 2+beta, 5]';   % diagonal matrix

L = eig(diag(d) + v*v')        % eigenvalues of the modified matrix
e = ones(4,1);
q = (d*e'-e*L').\(v*e');       % unnormalized eigenvectors cf. (5.15)

Q = sqrt(diag(q'*q));
q = q./(e*Q');                 % normalized eigenvectors

norm(q'*q-eye(4))              % check for orthogonality
```

We do not bother how we compute the eigenvalues. We simply use MATLAB's built-in function `eig`. We get the results of Table 5.1.

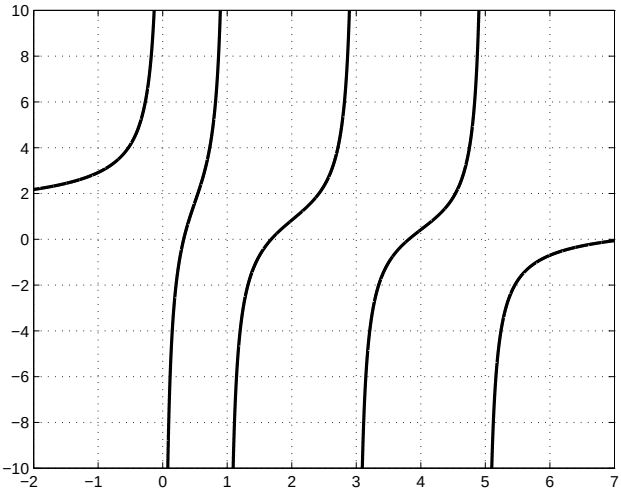

Figure 5.2: Secular equation corresponding to (5.32) for $\beta = 1$

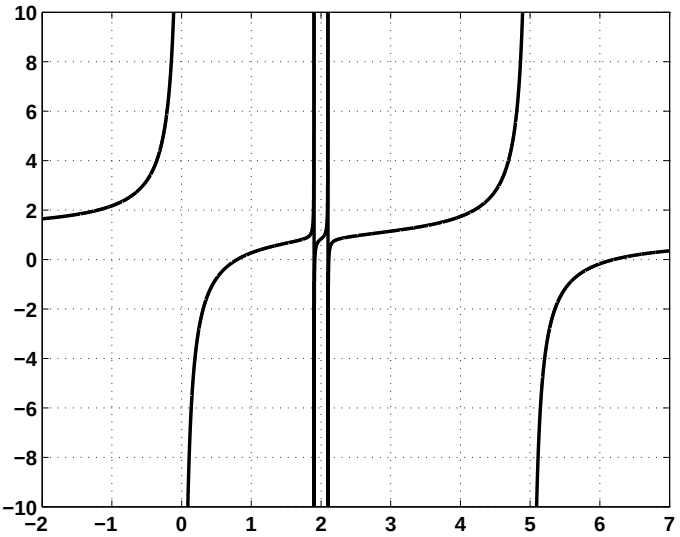

Figure 5.3: Secular equation corresponding to (5.32) for $\beta = 0.1$

We observe loss of orthogonality among the eigenvectors as the eigenvalues get closer and closer. This may not be surprising as we compute the eigenvectors by formula (5.15)

$$\mathbf{x} = \frac{(\lambda I - D)^{-1}\mathbf{v}}{\|(\lambda I - D)^{-1}\mathbf{v}\|}.$$

If $\lambda = \lambda_2$ and $\lambda = \lambda_3$ which are almost equal, $\lambda_2 \approx \lambda_3$ then intuitively one expects almost the same eigenvectors. We have in fact

$$Q^TQ - I_4 = \begin{bmatrix} -2.2204 \cdot 10^{-16} & 4.3553 \cdot 10^{-8} & 1.7955 \cdot 10^{-8} & -1.1102 \cdot 10^{-16} \\ 4.3553 \cdot 10^{-8} & 0 & -5.5511 \cdot 10^{-8} & -1.8298 \cdot 10^{-8} \\ 1.7955 \cdot 10^{-8} & -5.5511 \cdot 10^{-8} & -1.1102 \cdot 10^{-16} & -7.5437 \cdot 10^{-9} \\ -1.1102 \cdot 10^{-16} & -1.8298 \cdot 10^{-8} & -7.5437 \cdot 10^{-9} & 0 \end{bmatrix}.$$

Orthogonality is lost only with respect to the vectors corresponding to the eigenvalues close to 2.

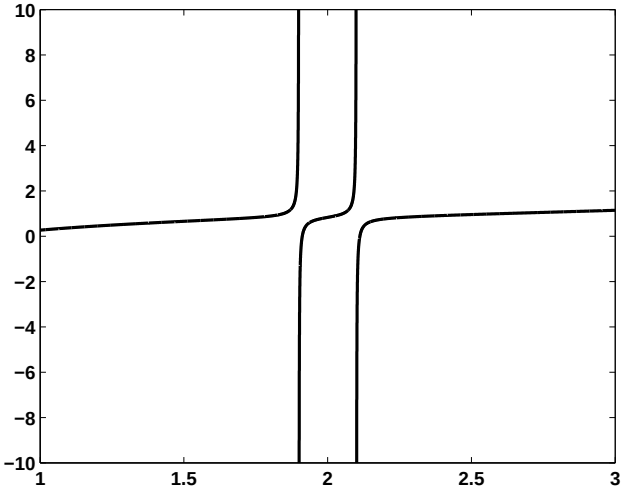

Figure 5.4: Secular equation corresponding to (5.32) for $\beta = 0.1$ for $1 \leq \lambda \leq 3$

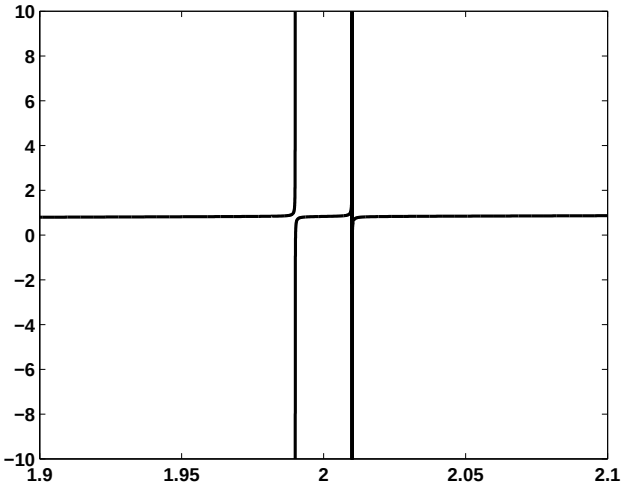

Figure 5.5: Secular equation corresponding to (5.32) for $\beta = 0.01$ for $1.9 \leq \lambda \leq 2.1$

Already Dongarra and Sorensen [4] analyzed this problem. In their formulation they normalize the vector $\mathbf{v}$ of $D + \rho \mathbf{v}\mathbf{v}^T$ to have norm unity, $\|\mathbf{v}\| = 1$.

They formulated

Lemma 5.1 *Let*

$$\mathbf{q}_\lambda^T = \left(\frac{v_1}{d_1 - \lambda}, \frac{v_2}{d_2 - \lambda}, \ldots, \frac{v_n}{d_n - \lambda} \right) \left[\frac{\rho}{f'(\lambda)} \right]^{1/2}. \tag{5.33}$$

Then for any $\lambda, \mu \notin \{d_1, \ldots, d_n\}$ we have

$$|\mathbf{q}_\lambda^T \mathbf{q}_\mu| = \frac{1}{|\lambda - \mu|} \frac{|f(\lambda) - f(\mu)|}{[f'(\lambda) f'(\mu)]^{1/2}}. \tag{5.34}$$

Proof. Observe that

$$\frac{\lambda - \mu}{(d_j - \lambda)(d_j - \mu)} = \frac{1}{d_j - \lambda} - \frac{1}{d_j - \mu}.$$

Then the proof is straightforward. ∎

β	λ_1	λ_2	λ_3	λ_4	$\|Q^TQ - I\|$
1	0.325651	1.682219	3.815197	7.176933	$5.6674 \cdot 10^{-16}$
0.1	0.797024	1.911712	2.112111	6.199153	$3.4286 \cdot 10^{-15}$
0.01	0.807312	1.990120	2.010120	6.192648	$3.9085 \cdot 10^{-14}$
10^{-4}	0.807418	1.999900	2.000100	6.192582	$5.6767 \cdot 10^{-12}$
10^{-8}	0.807418	1.99999999000000	2.00000001000000	6.192582	$8.3188 \cdot 10^{-08}$

Table 5.1: Loss of orthogonality among the eigenvectors computed by (5.15)

Formula (5.34) indicates how problems may arise. In exact arithmetic, if λ and μ are eigenvalues then $f(\lambda) = f(\mu) = 0$. However, in floating point arithmetic this values may be small but nonzero, e.g., $\mathcal{O}(\varepsilon)$. If $|\lambda - \mu|$ is very small as well then we may have trouble! So, a remedy for the problem was for a long time to compute the eigenvalues in doubled precision, so that $f(\lambda) = \mathcal{O}(\varepsilon^2)$. This would counteract a potential $\mathcal{O}(\varepsilon)$ of $|\lambda - \mu|$.

This solution was quite unsatisfactory because doubled precision is in general very slow since it is implemented in software. It took a decade until a proper solution was found.

5.8 The algorithm of Gu and Eisenstat

Computing eigenvector according to the formula

$$\mathbf{x} = \alpha(\lambda I - D)^{-1}\mathbf{v} = \alpha \begin{pmatrix} \frac{v_1}{\lambda - d_1} \\ \vdots \\ \frac{v_n}{\lambda - d_n} \end{pmatrix}, \qquad \alpha = \|(\lambda I - D)^{-1}\mathbf{v}\|, \tag{5.35}$$

is bound to fail if λ is very close to a pole d_k and the difference $\lambda - d_k$ has an error of size $\mathcal{O}(\varepsilon|d_k|)$ instead of only $\mathcal{O}(\varepsilon|d_k - \lambda|)$. To resolve this problem Gu and Eisenstat [5] found a trick that is at the same time ingenious and simple.

They observed that the v_k in (5.24) are very accurately determined by the data d_i and λ_i. Therefore, once the eigenvalues are computed *accurately* a vector $\hat{\mathbf{v}}$ could be computed such that the λ_i are *accurate* eigenvalues of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$. If $\hat{\mathbf{v}}$ approximates well the original $\mathbf{v}$ then the new eigenvectors will be the exact eigenvectors of a slightly modified eigenvalue problem, which is all we can hope for.

The zeros of the secular equation can be computed accurately by the method presented in section 5.6. However, a shift of variables is necessary. In the interval (d_i, d_{i+1}) the origin of the real axis is moved to d_i if λ_i is closer to d_i than to d_{i+1}, i.e., if $f((d_i + d_{i+1})/2) > 0$. Otherwise, the origin is shifted to d_{i+1}. This shift of the origin avoids the computation of the smallest difference $d_i - \lambda$ (or $d_{i+1} - \lambda$) in (5.35), thus avoiding cancellation in this most sensitive quantity. Equation (5.26) can be rewritten as

$$\underbrace{(c_1\Delta_{i+1} + c_2\Delta_i + c_3\Delta_i\Delta_{i+1})}_{b} - \underbrace{(c_1 + c_2 + c_3(\Delta_i + \Delta_{i+1}))}_{-a}\eta + \underbrace{c_3}_{c}\eta^2 = 0, \tag{5.36}$$

where $\Delta_i = d_i - \lambda_j$, $\Delta_{i+1} = d_{i+1} - \lambda_j$, and $\lambda_{j+1} = \lambda_j + \eta$ is the next approximate zero. With equations (5.28)–(5.30) the coefficients in (5.36) get

$$\begin{aligned} a &= c_1 + c_2 + c_3(\Delta_i + \Delta_{i+1}) = (1 + \Psi_1 + \Psi_2)(\Delta_i + \Delta_{i+1}) - (\Psi_1' + \Psi_2')\Delta_i\Delta_{i+1}, \\ b &= c_1\Delta_{i+1} + c_2\Delta_i + c_3\Delta_i\Delta_{i+1} = \Delta_i\Delta_{i+1}(1 + \Psi_1 + \Psi_2), \\ c &= c_3 = 1 + \Psi_1 + \Psi_2 - \Delta_i\Psi_1' - \Delta_{i+1}\Psi_2'. \end{aligned} \tag{5.37}$$

If we are looking for a zero that is closer to d_i than to d_{i+1} then we move the origin to λ_j, i.e., we have e.g. $\Delta_i = -\lambda_j$. The solution of (5.36) that lies inside the interval is [7]

$$\eta = \begin{cases} \dfrac{a - \sqrt{a^2 - 4bc}}{2c}, & \text{if } a \le 0, \\ \dfrac{2b}{a + \sqrt{a^2 - 4bc}}, & \text{if } a > 0. \end{cases} \tag{5.38}$$

The following algorithm shows how step 9 of the tridiagonal divide and conquer algorithm 5.1 must be implemented.

Algorithm 5.2 A stable eigensolver for $D + \mathbf{v}\mathbf{v}^T$

1: This algorithm stably computes the spectral decomposition of $D+\mathbf{v}\mathbf{v}^T = Q\Lambda Q^T$ where $D = \mathrm{diag}(d_1, \ldots d_n)$, $\mathbf{v} = [v_1, \ldots, v_n] \in \mathbb{R}^n$, $\Lambda = \mathrm{diag}(\lambda_1, \ldots \lambda_n)$, and $Q = [\mathbf{q}_1, \ldots, \mathbf{q}_n]$.
2: $d_{i+1} = d_n + \|\mathbf{v}\|^2$.
3: In each interval (d_i, d_{i+1}) compute the zero λ_i of the secular equation $f(\lambda) = 0$.
4: Use the formula (5.24) to compute the vector $\hat{\mathbf{v}}$ such that the λ_i are the 'exact' eigenvalues of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$.
5: In each interval (d_i, d_{i+1}) compute the eigenvectors of $D + \hat{\mathbf{v}}\hat{\mathbf{v}}$ according to (5.15),

$$\mathbf{q}_i = \frac{(\lambda_i I - D)^{-1}\hat{\mathbf{v}}}{\|(\lambda_i I - D)^{-1}\hat{\mathbf{v}}\|}.$$

6: **return** $(\Lambda; Q)$

5.8.1 A numerical example [continued]

We continue the discussion of the example on page 102 where the eigenvalue problem of

$$A = D + \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 0 & & & \\ & 2-\beta & & \\ & & 2+\beta & \\ & & & 5 \end{bmatrix} + \begin{bmatrix} 1 \\ \beta \\ \beta \\ 1 \end{bmatrix} \begin{bmatrix} 1 & \beta & \beta & 1 \end{bmatrix}. \tag{5.39}$$

The MATLAB code that we showed did not give orthogonal eigenvectors. We show in the following script that the formulae (5.24) really solve the problem.

```
dlam = zeros(n,1);
for k=1:n,
  [dlam(k), dvec(:,k)] = zerodandc(d,v,k);
end

V = ones(n,1);
for k=1:n,
  V(k) = prod(abs(dvec(k,:)))/prod(d(k) - d(1:k-1))/prod(d(k+1:n) - d(k));
  V(k) = sqrt(V(k));
end

Q = (dvec).\(V*e');
diagq = sqrt(diag(Q'*Q));
Q = Q./(e*diagq');
```

```
for k=1:n,
  if dlam(k)>0,
    dlam(k) = dlam(k) + d(k);
  else
    dlam(k) = d(k+1) + dlam(k);
  end
end

norm(Q'*Q-eye(n))
norm((diag(d) + v*v')*Q - Q*diag(dlam'))
```

A zero finder returns for each interval the quantity $\lambda_i - d_i$ and the vector $[d_1 - \lambda_i, \ldots, d_n - \lambda_i]^T$ to high precision. These vector elements have been computed as $(d_k - d_i) - (\lambda_i - d_i)$. The zerofinder of Li [7] has been employed here. At the end of this section we list the zerofinder written in MATLAB that was used here. The formulae (5.37) and (5.38) have been used to solve the quadratic equation (5.36). Notice that only one of the `while` loops is traversed, depending on if the zero is closer to the pole on the left or to the right of the interval. The v_k of formula (5.24) are computed next. Q contains the eigenvectors.

β	Algorithm	$\|Q^TQ - I\|$	$\|AQ - Q\Lambda\|$
0.1	I	$3.4286 \cdot 10^{-15}$	$5.9460 \cdot 10^{-15}$
	II	$2.2870 \cdot 10^{-16}$	$9.4180 \cdot 10^{-16}$
0.01	I	$3.9085 \cdot 10^{-14}$	$6.9376 \cdot 10^{-14}$
	II	$5.5529 \cdot 10^{-16}$	$5.1630 \cdot 10^{-16}$
10^{-4}	I	$5.6767 \cdot 10^{-12}$	$6.3818 \cdot 10^{-12}$
	II	$2.2434 \cdot 10^{-16}$	$4.4409 \cdot 10^{-16}$
10^{-8}	I	$8.3188 \cdot 10^{-08}$	$1.0021 \cdot 10^{-07}$
	II	$2.4980 \cdot 10^{-16}$	$9.4133 \cdot 10^{-16}$

Table 5.2: Loss of orthogonality among the eigenvectors computed by the straightforward algorithm (I) and the Gu-Eisenstat approach (II)

Again we ran the code for $\beta = 10^{-k}$ for $k = 0, 1, 2, 4, 8$. The numbers in Table 5.2 confirm that the new formulae are much more accurate than the straight forward ones. The norms of the errors obtained for the Gu-Eisenstat algorithm always are in the order of machine precision, i.e., 10^{-16}.

```
function [lambda,dl] = zerodandc(d,v,i)
% ZERODANDC - Computes eigenvalue lambda in the i-th interval
%             (d(i), d(i+1)) with Li's 'middle way' zero finder
%             dl is the n-vector [d(1..n) - lambda]

n = length(d);
di = d(i);
v = v.^2;
if i < n,
  di1 = d(i+1);  lambda = (di + di1)/2;
else
  di1 = d(n) + norm(v)^2; lambda = di1;
end
eta = 1;
psi1 = sum((v(1:i).^2)./(d(1:i) - lambda));
```

```
psi2 = sum((v(i+1:n).^2)./(d(i+1:n) - lambda));

if  1 + psi1 + psi2 > 0,  %  zero is on the left half of the interval

  d = d - di;  lambda = lambda - di; di1 = di1 - di; di = 0;

  while  abs(eta) > 10*eps
    psi1 = sum(v(1:i)./(d(1:i) - lambda));
    psi1s = sum(v(1:i)./((d(1:i) - lambda)).^2);

    psi2 = sum((v(i+1:n))./(d(i+1:n) - lambda));
    psi2s = sum(v(i+1:n)./((d(i+1:n) - lambda)).^2);

    % Solve for zero
    Di = -lambda; Di1 = di1 - lambda;
    a = (Di + Di1)*(1 + psi1 + psi2) - Di*Di1*(psi1s + psi2s);
    b = Di*Di1*(1 + psi1 + psi2);
    c = (1 + psi1 + psi2) - Di*psi1s - Di1*psi2s;
    if a > 0,
      eta = (2*b)/(a + sqrt(a^2 - 4*b*c));
    else
      eta = (a - sqrt(a^2 - 4*b*c))/(2*c);
    end
    lambda = lambda + eta;
  end

else  %  zero is on the right half of the interval

  d = d - di1; lambda = lambda - di1; di = di - di1; di1 = 0;

  while  abs(eta) > 10*eps
    psi1 = sum(v(1:i)./(d(1:i) - lambda));
    psi1s = sum(v(1:i)./((d(1:i) - lambda)).^2);

    psi2 = sum((v(i+1:n))./(d(i+1:n) - lambda));
    psi2s = sum(v(i+1:n)./((d(i+1:n) - lambda)).^2);

    % Solve for zero
    Di = di - lambda; Di1 = - lambda;
    a = (Di + Di1)*(1 + psi1 + psi2) - Di*Di1*(psi1s + psi2s);
    b = Di*Di1*(1 + psi1 + psi2);
    c = (1 + psi1 + psi2) - Di*psi1s - Di1*psi2s;
    if a > 0,
      eta = (2*b)/(a + sqrt(a^2 - 4*b*c));
    else
      eta = (a - sqrt(a^2 - 4*b*c))/(2*c);
    end
    lambda = lambda + eta;
  end

end

dl = d - lambda;

return
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide - Release 2.0*, SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

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

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

[4] J. J. Dongarra and D. C. Sorensen, *A fully parallel algorithm for the symmetric eigenvalue problem*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. s139–s154.

[5] M. Gu and S. C. Eisenstat, *A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 1266–1276.

[6] ——, *A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 172–191.

[7] R.-C. Li, *Solving secular equations stably and efficiently*, Technical Report UT-CS-94-260, University of Tennessee, Knoxville, TN, Nov. 1994. LAPACK Working Note No. 89.

[8] D. C. Sorensen and P. T. P. Tang, *On the orthogonality of eigenvectors computed by divide-and-conquer techniques*, SIAM J. Numer. Anal., 28 (1991), pp. 1752–1775.