

ANL/BK--75732

DE92 010936

LAPACK Users' Guide

Release 1.0

APR 03 1992

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz,
A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen

31 January 1992

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

42

Abstract

LAPACK is a transportable library of Fortran 77 subroutines for solving the most common problems in numerical linear algebra: systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems.

LAPACK is designed to supersede LINPACK and EISPACK, principally by restructuring the software to achieve much greater efficiency on vector processors, high-performance "superscalar" workstations, and shared-memory multi-processors. LAPACK also adds extra functionality, uses some new or improved algorithms, and integrates the two sets of algorithms into a unified package.

The LAPACK Users' Guide gives an informal introduction to the design of the algorithms and software, summarizes the contents of the package, describes conventions used in the software and documentation, and includes complete specifications for calling the routines.

This edition of the Users' Guide describes Release 1.0 of LAPACK.

Dedication

This work is dedicated to Jim Wilkinson whose ideas and spirit have given us inspiration and influenced the project at every turn.

Contents

Preface	8
1 Essentials	11
1.1 LAPACK	11
1.2 Problems LAPACK Can Solve	11
1.3 Computers for which LAPACK is suitable	12
1.4 LAPACK compared with LINPACK and EISPACK	12
1.5 LAPACK and the BLAS	12
1.6 Documentation for LAPACK	12
1.7 Availability of LAPACK	13
1.8 Installation of LAPACK	13
1.9 Support for LAPACK	13
1.10 Known Problems in LAPACK	14
2 Contents of LAPACK	15
2.1 Structure of LAPACK	15
2.1.1 Levels of routines	15
2.1.2 Data types and precision	16
2.1.3 Naming Scheme	16
2.2 Driver Routines	18
2.2.1 Linear Equations	18
2.2.2 Linear Least Squares Problems (LLS)	20
2.2.3 Standard Eigenvalue And Singular Value Problems	20

2.2.4	Generalized Eigenvalue Problems	23
2.3	Computational routines	24
2.3.1	Linear Equations	24
2.3.2	Orthogonal Factorizations	29
2.3.3	Symmetric Eigenproblem	31
2.3.4	Nonsymmetric Eigenproblem	33
2.3.5	Singular Value Decomposition	34
2.3.6	Generalized Symmetric-Definite Eigenproblems	36
3	Performance of LAPACK	38
3.1	Factors That Affect Performance	38
3.1.1	Vectorization	39
3.1.2	Data movement	39
3.1.3	Parallelism	39
3.2	The BLAS as the Key To Portability	39
3.3	Block Algorithms And Their Derivation	41
3.4	Examples of block algorithms in LAPACK	43
3.4.1	Factorizations for solving linear equations	44
3.4.2	QR factorization	45
3.4.3	Eigenvalue problems	45
4	Accuracy and Stability	48
4.1	Roundoff Errors in Floating Point Arithmetic	48
4.2	Vector and Matrix Norms	49
4.3	Standard Error Analysis	50
4.4	Improved Error Bounds	51
4.5	How to Read Error Bounds	52
4.6	Error Bounds for Linear Equation Solving	53
4.7	Error Bounds for Linear Least Squares Problems	54
4.8	Error Bounds for the Singular Value Decomposition	55

4.9	Error Bounds for the Symmetric Eigenproblem	57
4.10	Error Bounds for the Nonsymmetric Eigenproblem	58
4.10.1	Summary	58
4.10.2	Balancing and Conditioning	60
4.10.3	Computing s and sep	60
4.11	Error bounds for the generalized symmetric-definite eigenproblem	62
4.12	Error bounds for Fast Level 3 BLAS	64
5	Documentation and Software Conventions	65
5.1	Design and Documentation of Argument Lists	65
5.1.1	Structure of the Documentation	65
5.1.2	Order of Arguments	65
5.1.3	Argument Descriptions	66
5.1.4	Option Arguments	66
5.1.5	Problem Dimensions	67
5.1.6	Array Arguments	67
5.1.7	Work Arrays	67
5.1.8	Error handling and the diagnostic argument INFO	68
5.2	Determining the block size for block algorithms	68
5.3	Matrix storage schemes	69
5.3.1	Conventional Storage	70
5.3.2	Packed Storage	71
5.3.3	Band Storage	71
5.3.4	Tridiagonal and Bidiagonal Matrices	73
5.3.5	Unit Triangular Matrices	73
5.3.6	Real Diagonal Elements of Complex Matrices	73
5.4	Representation of orthogonal or unitary matrices	73
6	Installing LAPACK routines	75
6.1	Points to note	75

6.2	Installing ILAENV	76
7	Troubleshooting	79
7.1	Failures or wrong results	79
7.2	Poor performance	80
	Bibliography	82
A	Index of Driver and Computational Routines	85
B	Index of Auxiliary Routines	95
C	Quick Reference Guide to the BLAS	102
D	Converting from LINPACK or EISPACK	107
E	LAPACK Working Notes	115
F	Specifications of Routines	118

List of Tables

2.1	Matrix types in the LAPACK naming scheme	17
2.2	Driver routines for linear equations	19
2.3	Driver routines for linear least squares problems	20
2.4	Driver routines for standard eigenvalue and singular value problems	22
2.5	Driver routines for generalized eigenvalue problems	23
2.6	Computational routines for linear equations	27
2.7	Computational routines for linear equations (continued)	28
2.8	Computational routines for orthogonal factorizations	30
2.9	Computational routines for the symmetric eigenproblem	32
2.10	Computational routines for the nonsymmetric eigenproblem	35
2.11	Computational routines for the singular value decomposition	36
2.12	Computational routines for the generalized symmetric-definite eigenproblem	37
3.1	Speed in megaflops of Level 2 and Level 3 BLAS operations on a CRAY Y-MP	40
3.2	Speed in megaflops of Cholesky factorization $A = U^T U$ for $n = 500$	43
3.3	Speed in megaflops of SGETRF/DGETRF for square matrices of order n	44
3.4	Speed in megaflops of SPOTRF/DPOTRF for matrices of order n with UPLO = 'U'	44
3.5	Speed in megaflops of SSYTRF for matrices of order n with UPLO = 'U' on a CRAY-2	45
3.6	Speed in megaflops of SGEQRF/DGEQRF for square matrices of order n	46
3.7	Speed in megaflops of reductions to condensed forms on an IBM 3090E VF	47
4.1	Vector and Matrix Norms	49
4.2	Asymptotic error bounds for the Nonsymmetric Eigenproblem	59

4.3	Global error bounds for the Nonsymmetric Eigenproblem	59
• 6.1	Use of the block parameters NB, NBMIN, and NX in LAPACK	77

Preface

The development of LAPACK was a natural step after specifications of the Level 2 and 3 BLAS were drawn up in 1984-86 and 1987-88. Research on block algorithms had been ongoing for several years, but agreement on the BLAS made it possible to construct a new software package to take the place of LINPACK and EISPACK. This also seemed to be a good time to implement a number of algorithmic advances that had been made since LINPACK and EISPACK were written in the 1970's. The proposal for LAPACK was submitted while the Level 3 BLAS were still being developed and funding was obtained from the National Science Foundation beginning in 1987.

LAPACK is more than just an update of its popular predecessors. It extends the functionality of LINPACK and EISPACK by including driver routines for linear systems, iterative refinement and error bounds, eigencondition estimation, and the capability for finding selected eigenvalues and eigenvectors. LAPACK improves on the accuracy of the standard algorithms in EISPACK by including high accuracy algorithms for finding eigenvalues of the bidiagonal and tridiagonal matrices that arise in SVD and symmetric eigenvalue problems. It is also a research project on achieving good performance in a portable way by calling the BLAS. We have tried to be consistent with our documentation and coding style throughout LAPACK in the hope that LAPACK will serve as a model for other software development efforts. In particular, we hope that LAPACK and this guide will be of value in the classroom. Finally, we hope that LAPACK will be used, both as a library of subroutines and as a source of building blocks for larger applications.

We have encountered some obstacles to our goal of a portable library, most of which should not be apparent to a casual user. We have assumed the BLAS are implemented efficiently on the target machine, but the optimal performance of the LAPACK routines depends to some extent on a small set of parameters, such as the block size, which must be computed for each machine (reasonable default values are provided). Most of the LAPACK code is written in standard Fortran 77, but the double precision complex data type is not part of the standard, so we have had to make some assumptions about the names of intrinsic functions that do not hold on all machines (see section 6.1). Finally, our rigorous testing suite included test problems scaled at the extremes of the arithmetic range, which can vary greatly from machine to machine. On some machines, we have had to restrict the range more than on others.

Since most of the performance improvements in LAPACK come from restructuring the algorithms to use the Level 2 and 3 BLAS, we benefited greatly by having access from the early stages of the project to a complete set of BLAS developed for the CRAY machines by Cray Research. Later, the BLAS library developed by IBM for the IBM RISC/6000 was very helpful in proving the worth of

block algorithms and LAPACK on super-scalar workstations. Many of our test sites, both computer vendors and research institutions, also worked on optimizing the BLAS and thus helped to get good performance from LAPACK. We are very pleased at the extent to which the user community has embraced the BLAS, not only for performance reasons, but also because we feel developing software around a core set of common routines like the BLAS is good software engineering practice.

A number of technical reports were written during the development of LAPACK and published as LAPACK Working Notes, initially by Argonne National Laboratory and later by the University of Tennessee. Many of these reports later appeared as journal articles. Appendix E lists the LAPACK Working Notes, and the bibliography gives the most recent published reference.

A follow-on project, LAPACK 2, has been funded in the US by the NSF and DARPA. One of its aims will be to add a modest amount of additional functionality to the current LAPACK package — for example, routines for the generalized SVD and additional routines for generalized eigenproblems. These routines will be included in a future release of LAPACK when they are available. LAPACK 2 will also produce routines which implement LAPACK-type algorithms for distributed-memory machines, routines which take special advantage of IEEE arithmetic, and versions of parts of LAPACK in C and Fortran 90. The precise form of these other software packages which will result from LAPACK 2 has not yet been decided.

As the successor to LINPACK and EISPACK, LAPACK has drawn heavily on both the software and documentation from those collections. The test and timing software for the Level 2 and 3 BLAS was used as a model for the LAPACK test and timing software, and in fact the LAPACK timing software includes the BLAS timing software as a subset. Formatting of the software and conversion from single to double precision was done using Toolpack/1 [9], which was indispensable to the project. We owe a great debt to our colleagues who have helped create the infrastructure of scientific computing on which LAPACK has been built.

The development of LAPACK was primarily supported by NSF grant ASC-8715728. Zhaojun Bai had partial support from DARPA grant F49620-87-C0065, Christian Bischof was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38, James Demmel had partial support from NSF grant DCR-8552474, and Jack Dongarra had partial support from the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400.

We acknowledge with gratitude the support which we have received from the following organizations, and the help of individual members of their staff: Cornell Theory Center; Cray Research Inc; IBM ECSEC Rome; IBM Scientific Center, Bergen; NAG Ltd.

We also thank many, many people who have contributed code, criticism, ideas and encouragement. We wish especially to acknowledge the contributions of: Mario Arioli, Mir Assadullah, Jesse Barlow, Mel Ciment, Percy Deift, Augustin Dubrulle, Iain Duff, Alan Edelman, Sam Figueroa, Pat Gaffney, Nick Higham, Liz Jessup, Bo Kågström, Velvel Kahan, Linda Kaufman, L.-C. Li, Bob Manchek, Peter Mayes, Cleve Moler, Beresford Parlett, Mick Pont, Giuseppe Radicati, Tom Rowan, Pete Stewart, Peter Tang, Carlos Tomei, Charlie Van Loan, Krešimir Veselić, Phuong Vu, and Reed Wade.

Finally we thank all the test sites who received three preliminary distributions of LAPACK software and who ran an extensive series of test programs and timing programs for us; their efforts have influenced the final version of the package in numerous ways.

Chapter 1

Essentials

RTFM - *Anonymous*

1.1 LAPACK

LAPACK 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 Linear Algebra PACKage.

1.2 Problems LAPACK Can Solve

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 band matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices. See Chapter 2 for a complete summary of the contents.

1.3 Computers for which LAPACK is suitable

LAPACK is designed to give high efficiency on vector processors, high-performance “superscalar” workstations, and shared-memory multi-processors. LAPACK in its present form is less likely to give good performance on other types of parallel architectures (for example, massively parallel SIMD machines, or distributed-memory machines), but work has begun to try to adapt LAPACK to these new architectures. LAPACK can also be used satisfactorily on all types of scalar machines (PC’s, workstations, mainframes). See Chapter 3 for some examples of the performance achieved by LAPACK routines.

1.4 LAPACK compared with LINPACK and EISPACK

LAPACK has been designed to supersede LINPACK [15] and EISPACK [39, 27], 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.

Appendix D lists the LAPACK counterparts of LINPACK and EISPACK routines. Not all the facilities of LINPACK and EISPACK are covered by Release 1.0 of LAPACK.

1.5 LAPACK and the BLAS

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS) [36, 19, 17]. 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 methodology for constructing LAPACK routines in terms of calls to the BLAS is described in Chapter 3.

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* (see below). This code constitutes the “model implementation” [18, 16].

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.

1.6 Documentation for LAPACK

This **Users’ Guide** gives an informal introduction to the design of the package, and a detailed description of its contents. Chapter 5 explains the conventions used in the software and documentation. Appendix F contains complete specifications of all the driver routines and computational

routines. These specifications have been derived from the leading comments in the source text.

1.7 Availability of LAPACK

Individual routines from LAPACK are most easily obtained by electronic mail through *netlib* [21]. At the time of this writing, the e-mail addresses for *netlib* are

`netlib@cornl.gov`
`netlib@research.att.com`

General information about LAPACK can be obtained by sending mail to one of the above addresses with the message

`send index from lapack`

The complete package, including test code and timing programs in four different Fortran data types, constitutes some 600,000 lines of Fortran source and comments. A magnetic tape of the complete LAPACK package can be obtained from NAG for a nominal handling charge.

For further details contact NAG at one of the following addresses:

NAG Inc
1400 Opus Place, Suite 200
Downers Grove, IL 60515-5702
USA
Tel: +1 708 971 2337
Fax: +1 708 971 2706

NAG Ltd
Wilkinson House
Jordan Hill Road
Oxford OX2 8DR
England
Tel: +44 865 511245
Fax: +44 865 310139

NAG GmbH
Schleissheimerstrasse 5
W-8046 Garching bei München
Germany
Tel: +49 89 3207395
Fax: +49 89 3207396

1.8 Installation of LAPACK

A comprehensive Implementors' Guide [2] is distributed with the complete package. This includes descriptions of the test programs and timing programs, and detailed instructions on running them. See also Chapter 6.

1.9 Support for LAPACK

LAPACK has been thoroughly tested before release, on many different types of computers. The LAPACK project supports the package in the sense that reports of errors or poor performance will gain immediate attention from the developers. Such reports — and also descriptions of interesting applications and other comments — should be sent to:

LAPACK Project
c/o J.J. Dongarra
Computer Science Department
University of Tennessee
Knoxville, Tennessee 37996-1301
USA
Email: lapack@cs.utk.edu

1.10 Known Problems in LAPACK

A list of known problems, bugs, and compiler errors for LAPACK is maintained on *netlib*. For a copy of this report, send email to netlib of the form:

`send bugreport from lapack`

Chapter 2

Contents of LAPACK

2.1 Structure of LAPACK

2.1.1 Levels of routines

The subroutines in LAPACK are classified as follows:

- **driver** routines, each of which solves a complete problem, for example solving a system of linear equations, or computing the eigenvalues of a real symmetric matrix. Users are recommended to use a driver routine if there is one that meets their requirements. They are listed in Section 2.2.
- **computational** routines, each of which performs a distinct computational task, for example an *LU* factorization, or the reduction of a real symmetric matrix to tridiagonal form. Each driver routine calls a sequence of computational routines. Users (especially software developers) may need to call computational routines directly to perform tasks, or sequences of tasks, that cannot conveniently be performed by the driver routines. They are listed in Section 2.3.
- **auxiliary** routines, which in turn can be classified as follows:
 - routines that perform subtasks of block algorithms — in particular, routines that implement unblocked versions of the algorithms;
 - routines that perform some commonly required low-level computations, for example scaling a matrix, computing a matrix-norm, or generating an elementary Householder matrix; some of these may be of interest to numerical analysts or software developers and could be considered for future additions to the BLAS;
 - a few extensions to the BLAS, such as routines for applying complex plane rotations, or matrix-vector operations involving complex symmetric matrices (the BLAS themselves are not strictly speaking part of LAPACK).

Both driver routines and computational routines are fully described in this Users' Guide, but not the auxiliary routines. A list of the auxiliary routines, with one-line descriptions of their functions, is given in Appendix B.

2.1.2 Data types and precision

LAPACK provides the same range of functionality for **real** and **complex** data.

For most computations, there are matching routines, one for real and one for complex data, but there are a few exceptions. For example, corresponding to the routines for real symmetric indefinite systems of linear equations, there are routines for complex Hermitian and complex symmetric systems, because both types of complex systems occur in practical applications. However, there is no complex analogue of the routine for finding selected eigenvalues of a real symmetric tridiagonal matrix, because a complex Hermitian matrix can always be reduced to a real symmetric tridiagonal matrix.

Matching routines for real and complex data have been coded to maintain a close correspondence between the two, wherever possible. However, in some areas (especially the nonsymmetric eigenproblem) the correspondence is necessarily weaker.

All routines in LAPACK are provided in both **single** and **double** precision versions. The double precision versions have been generated automatically, using Toolpack/1 [9].

Double precision routines for complex matrices require the non-standard Fortran data type COMPLEX*16, which is available on most machines where double precision computation is usual.

2.1.3 Naming Scheme

The name of each LAPACK routine is a coded specification of its function (within the very tight limits of standard Fortran 77 6-character names).

All driver and computational routines have names of the form **XYZZZ**, where for some driver routines the 6th character is blank.

The first letter, **X**, indicates the data type as follows:

S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16 or DOUBLE COMPLEX

When we wish to refer to an LAPACK routine generically, regardless of data type, we replace the first letter by "x". Thus xGESV refers to any or all of the routines SGESV, CGESV, DGESV and ZGESV.

The next two letters, **YY**, indicate the type of matrix (or of the most significant matrix). Most of these two-letter codes apply to both real and complex matrices; a few apply specifically to one or

Table 2.1: Matrix types in the LAPACK naming scheme

BD	bidiagonal
GB	general band
GE	general (i.e. unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e. a pair of general matrices)
GT	general tridiagonal
HE	(complex) Hermitian
HG	upper Hessenberg matrix, generalized problem (i.e. a Hessenberg and a triangular matrix)
HP	(complex) Hermitian, packed storage
HS	upper Hessenberg
OR	(real) orthogonal
OP	(real) orthogonal, packed storage
PB	symmetric or Hermitian positive definite band
PO	symmetric or Hermitian positive definite
PP	symmetric or Hermitian positive definite, packed storage
PT	symmetric or Hermitian positive definite tridiagonal
SB	(real) symmetric band
SP	symmetric, packed storage
ST	(real) symmetric tridiagonal
SY	symmetric
TB	triangular band
TG	triangular matrices, generalized problem (i.e. a pair of triangular matrices)
TP	triangular, packed storage
TR	triangular (or in some cases quasi-triangular)
TZ	trapezoidal
UN	(complex) unitary
UP	(complex) unitary, packed storage

the other, as indicated in Table 2.1.

When we wish to refer to a class of routines that performs the same function on different types of matrices, we replace the first three letters by “xyy”. Thus xyySVX refers to all the expert driver routines for systems of linear equations that are listed in Table 2.2.

The last three letters **ZZZ** indicate the computation performed. Their meanings will be explained in Section 2.3. For example, SGEBRD is a single precision routine that performs a bidiagonal reduction (BRD) of a real general matrix.

The names of auxiliary routines follow a similar scheme except that the 2nd and 3rd characters YY are usually LA (for example, SLASCL or CLARFG). There are two kinds of exception. Auxiliary routines that implement an unblocked version of a block algorithm have similar names to the routines that perform the block algorithm, with the 6th character being ‘2’ (for example, SGETF2 is the unblocked version of SGETRF). A few routines that may be regarded as extensions to the

BLAS are named according to the BLAS naming schemes (for example, CROT, CSYR).

2.2 Driver Routines

This section describes the driver routines in LAPACK. Further details on the terminology and the numerical operations they perform are given in Section 2.3, which describes the computational routines.

2.2.1 Linear Equations

Two types of driver routines are provided for solving systems of linear equations:

- a **simple** driver (name ending -SV), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X ;
- an **expert** driver (name ending -SVX), which can also perform the following functions:
 - estimate the condition number of A and check for near-singularity;
 - refine the solution and compute forward and backward error bounds;
 - (optionally) equilibrate the system if A is poorly scaled.

The expert driver requires roughly twice as much storage as the simple driver in order to perform these extra functions.

Both types of driver routines can handle multiple right hand sides (the columns of B).

Different driver routines are provided to take advantage of special properties or storage schemes of the matrix A , as shown in Table 2.2.

All of the computational routines for solving linear systems are used in the context of the driver routines except the matrix inversion routines (xyyTRI). In most cases, a factorization plus solve is faster and more accurate than inverting the coefficient matrix explicitly.

Table 2.2: Driver routines for linear equations

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
general	simple driver	SGESV	CGESV	DGESV	ZGESV
	expert driver	SGESVX	CGESVX	DGESVX	ZGESVX
general band	simple driver	SGBSV	CGBSV	DGBSV	ZGBSV
	expert driver	SGBSVX	CGBSVX	DGBSVX	ZGBSVX
general tridiagonal	simple driver	SGTSV	CGTSV	DGTSV	ZGTSV
	expert driver	SGTSVX	CGTSVX	DGTSVX	ZGTSVX
symmetric/Hermitian positive-definite	simple driver	SPOSV	CPOSV	DPOSV	ZPOSV
	expert driver	SPOSVX	CPOSVX	DPOSVX	ZPOSVX
symmetric/Hermitian positive-definite (packed storage)	simple driver	SPPSV	CPPSV	DPPSV	ZPPSV
	expert driver	SPPSVX	CPPSVX	DPPSVX	ZPPSVX
symmetric/Hermitian positive-definite band	simple driver	SPBSV	CPBSV	DPBSV	ZPBSV
	expert driver	SPBSVX	CPBSVX	DPBSVX	ZPBSVX
symmetric/Hermitian positive-definite tridiagonal	simple driver	SPTSV	CPTSV	DPTSV	ZPTSV
	expert driver	SPTSVX	CPTSVX	DPTSVX	ZPTSVX
symmetric/Hermitian indefinite	simple driver	SSYSV	CHESV	DSYSV	ZHESV
	expert driver	SSYSVX	CHESVX	DSYSVX	ZHESVX
complex symmetric	simple driver		CSYSV		ZSYSV
	expert driver		CSYSVX		ZSYSVX
symmetric/Hermitian indefinite (packed storage)	simple driver	SSPSV	CHPSV	DSPSV	ZHPSV
	expert driver	SSPSVX	CHPSVX	DSPSVX	ZHPSVX
complex symmetric (packed storage)	simple driver		CSPSV		ZSPSV
	expert driver		CSPSVX		ZSPSVX

Table 2.3: Driver routines for linear least squares problems

Operation	Single precision		Double precision	
	real	complex	real	complex
solve LLS or using <i>QR</i> or <i>LQ</i> factorization	SGELS	CGELS	DGELS	ZGELS
solve LLS using complete orthogonal factorization	SGELSX	CGELSX	DGELSX	ZGELSX
solve LLS using SVD	SGELSS	CGELSS	DGELSS	ZGELSS

2.2.2 Linear Least Squares Problems (LLS)

The linear least squares problem is:

$$\underset{x}{\text{minimize}} \quad \|b - Ax\|_2 \quad (2.1)$$

where A is an m -by- n matrix, b is a given m element vector and x is the n element solution vector. When $m > n$ the problem is also referred to as finding a **least-squares solution** to an **over-determined** system of linear equations, and when $m < n$ the problem is also referred to as finding a **least-squares solution** to an **under-determined** system of linear equations.

In the most usual case $m \geq n$ and $\text{rank}(A) = n$, and in this case the solution to problem (2.1) is unique. When $m < n$, or $m \geq n$ and $\text{rank}(A) < n$, then the solution is not unique. The particular solution for which $\|x\|_2$ is minimized is called the **minimum norm solution**.

The driver routine xGELS solves the problem (2.1) on the assumption that A has full rank, using a *QR* or *LQ* factorization of A .

The driver routines xGELSX and xGELSS solve problem (2.1), allowing for the possibility that A is rank-deficient; xGELSX uses a **complete orthogonal factorization** of A , while xGELSS uses the **singular value decomposition** of A .

The routine xGELS (but not xGELSX or xGELSS) allows A to be replaced by A^T in the statement of the problem. The linear least squares driver routines are listed in Table 2.3.

2.2.3 Standard Eigenvalue And Singular Value Problems

Symmetric eigenproblems (SEP)

The **symmetric eigenvalue problem** is to find the **eigenvalues**, λ , and corresponding **eigenvectors**, $z \neq 0$, such that

$$Az = \lambda z, \quad A = A^T, \text{ where } A \text{ is real.}$$

For the **Hermitian eigenvalue problem** we have

$$Az = \lambda z, \quad A = A^H.$$

For both problems the eigenvalues λ are real.

When all eigenvalues and eigenvectors have been computed, we write:

$$AZ = Z\Lambda$$

where Λ is a diagonal matrix whose diagonal elements are the eigenvalues, and Z is an orthogonal (or unitary) matrix whose columns are the eigenvectors.

Two types of driver routines are provided for symmetric or Hermitian eigenproblems:

- a **simple** driver (name ending -EV), which computes all the eigenvalues and (optionally) the eigenvectors of a symmetric or Hermitian matrix A ;
- an **expert** driver (name ending -EVX), which can compute either all or a selected subset of the eigenvalues, and (optionally) the corresponding eigenvectors;

Different driver routines are provided to take advantage of special structure or storage of the matrix A , as shown in Table 2.4.

Nonsymmetric eigenproblems (NEP)

The **nonsymmetric eigenvalue problem** is to find the **eigenvalues**, λ , and corresponding **eigenvectors**, $v \neq 0$, such that

$$Av = \lambda v.$$

This problem can be solved via the **Schur factorization** of A , defined in the real case as

$$A = ZTZ^T,$$

where Z is an orthogonal matrix and T is an upper quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks, the 2-by-2 blocks corresponding to complex conjugate pairs of eigenvalues of A . In the complex case the Schur factorization is

$$A = ZTZ^H,$$

where Z is unitary and T is a complex upper triangular matrix.

The columns of Z are called the **Schur vectors**. For each k ($1 \leq k \leq n$), the first k columns of Z form an orthonormal basis for the invariant subspace corresponding to the first k eigenvalues on the diagonal of T . Because this basis is orthonormal, it is preferable in many applications to compute Schur vectors rather than eigenvectors. It is possible to order the Schur factorization so that any desired set of k eigenvalues occupy the k leading positions on the diagonal of T .

Two pairs of drivers are provided, one pair focusing on the Schur factorization, and the other pair on the eigenvalues and eigenvectors as shown in Table 2.4:

- **xGEES**: a simple driver that computes all or part of the Schur factorization of A , with optional ordering of the eigenvalues;

Table 2.4: Driver routines for standard eigenvalue and singular value problems

Type of problem	Function and storage scheme	Single precision		Double precision	
		real	complex	real	complex
SEP	simple driver	SSYEV	CHEEV	DSYEV	ZHEEV
	expert driver	SSYEVX	CHEEVX	DSYEVX	ZHEEVX
	simple driver (packed storage)	SSPEV	CHPEV	DSPEV	ZHPEV
	expert driver (packed storage)	SSPEVX	CHPEVX	DSPEVX	ZHPEVX
	simple driver (band matrix)	SSBEV	CHBEV	DSBEV	ZHBEV
	expert driver (band matrix)	SSBEVX	CHBEVX	DSBEVX	ZHBEVX
	simple driver (tridiagonal matrix)	SSTEVE		DSTEVE	
	expert driver (tridiagonal matrix)	SSTEVEVX		DSTEVEVX	
NEP	simple driver for Schur factorization	SGEES	CGEES	DGEES	ZGEES
	expert driver for Schur factorization	SGEESX	CGEESX	DGEESX	ZGEESX
	simple driver for eigenvalues/vectors	SGEEV	CGEEV	DGEEV	ZGEEV
	expert driver for eigenvalues/vectors	SGEEVX	CGEEVX	DGEEVX	ZGEEVX
SVD	singular values/vectors	SGESVD	CGESVD	DGESVD	ZGESVD

- xGEESX: an expert driver that additionally can compute a condition number for the average of a selected subset of the eigenvalues, and for the corresponding right invariant subspace;
- xGEEV: a simple driver that computes all the eigenvalues of A , and (optionally) the right or left eigenvectors (or both);
- xGEEVX: an expert driver that additionally can balance the matrix to improve the conditioning of the eigenvalues and eigenvectors, and can compute condition numbers for the eigenvalues or right eigenvectors (or both).

Singular value decomposition (SVD)

The **singular value decomposition** of an m -by- n matrix A is given by

$$A = U \Sigma V^T \quad A = U \Sigma V^H \text{ (in the complex case),}$$

where U and V are orthogonal (unitary) and Σ is an m -by- n matrix with real diagonal elements, σ_i , such that

$$\sigma_1 \geq \sigma_2 \geq \dots \sigma_{\min(m,n)} \geq 0.$$

The σ_i are the **singular values** of A and the first $\min(m, n)$ columns of U and V are the **left** and **right singular vectors** of A . A single driver routine xGESVD computes all or part of the singular value decomposition of a general nonsymmetric matrix (see Table 2.4).

Table 2.5: Driver routines for generalized eigenvalue problems

Type of problem	Function and storage scheme	Single precision		Double precision	
		real	complex	real	complex
GSEP	simple driver	SSYGV	CHEGV	DSYGV	ZHEGV
	simple driver (packed storage)	SSPGV	CHPGV	DSPGV	ZHPGV

2.2.4 Generalized Eigenvalue Problems

Generalized symmetric-definite eigenproblems (GSEP)

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

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where A and B are symmetric or Hermitian and B is positive-definite, as shown in Table 2.5.

Generalized nonsymmetric eigenproblems (GNEP)

Routines for generalized nonsymmetric eigenproblems will be provided in a future release of LAPACK.

2.3 Computational routines

2.3.1 Linear Equations

We use the standard notation for a system of simultaneous linear equations:

$$Ax = b \quad (2.2)$$

where A is the **coefficient matrix**, b is the **right hand side**, and x is the **solution**. A is assumed to be a square matrix of order n , but the LU factorization is provided for a general m -by- n matrix. If there are several right hand sides, we write

$$AX = B \quad (2.3)$$

where the columns of B are the individual right hand sides, and the columns of X are the corresponding solutions. The basic task is to compute x , given A and b .

If A is upper or lower triangular, (2.2) can be solved by a straightforward process of backward or forward substitution. Otherwise, the solution is obtained after first factorizing A as a product of triangular matrices (and possibly also a diagonal matrix or permutation matrix).

The form of the factorization depends on the properties of the matrix A . LAPACK provides routines for the following types of matrices, based on the stated factorizations:

- **general matrices** (LU factorization with partial pivoting):

$$A = PLU$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$).

- **general band matrices** (LU factorization with partial pivoting): If A is m -by- n with kl subdiagonals and ku superdiagonals, the factorization is

$$A = LU$$

where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl + ku$ superdiagonals.

- **symmetric positive-definite matrices** (Cholesky factorization):

$$A = U^T U \text{ or } A = LL^T$$

where U is an upper triangular matrix and L is lower triangular.

- **symmetric positive-definite tridiagonal matrices** (LDL^T factorization):

$$A = UDU^T \text{ or } A = LDL^T$$

where U is a unit upper bidiagonal matrix, L is unit lower bidiagonal, and D is diagonal.

- **symmetric indefinite matrices** (symmetric indefinite factorization):

$$A = UDU^T \text{ or } A = LDL^T$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with diagonal blocks of order 1 or 2.

The factorization for a general tridiagonal matrix is like that for a general band matrix with $kl = 1$ and $ku = 1$. The factorization for a symmetric positive-definite band matrix with k superdiagonals (or subdiagonals) has the same form as for a symmetric positive-definite matrix, but the factor U (or L) is a band matrix with k superdiagonals (subdiagonals). Band matrices use a compact band storage scheme described in section 5.3.3. LAPACK routines are also provided for symmetric matrices (whether positive-definite or indefinite) using **packed** storage, as described in section 5.3.2.

While the primary use of a matrix factorization is to solve a system of equations, other related tasks are provided as well. Wherever possible, LAPACK provides routines to perform each of these tasks for each type of matrix and storage scheme (see Table 2.6). The following list relates the tasks to the last 3 characters of the name of the corresponding computational routine:

- xyyTRF**: factorize (obviously not needed for triangular matrices);
- xyyTRS**: use the factorization (or the matrix A itself if it is triangular) to solve (2.3) by forward or backward substitution;
- xyyCON**: estimate the reciprocal of the condition number $\kappa(A) = \|A\| \|A^{-1}\|$; Higham's modification [33] of Hager's method [30] is used to estimate $\|A^{-1}\|$, except for symmetric positive-definite tridiagonal matrices for which it is computed directly with comparable efficiency [31];
- xyyRFS**: compute bounds on the error in the computed solution (returned by the **xyyTRS** routine), and refine the solution to reduce the backward error (see below);
- xyyTRI**: use the factorization (or the matrix A itself if it is triangular) to compute A^{-1} (not provided for band matrices, because the inverse does not in general preserve bandedness).
- xyyEQU**: compute scaling factors to equilibrate A (not provided for tridiagonal, symmetric indefinite, or triangular matrices);

Note that some of the above routines depend on the output of others:

- xyyTRF**: may work on an equilibrated matrix from **xyyEQU** + **xLAQyy**, if yy is one of {GE, GB, PO, PP, PB} (see driver routine **xyySVX** for sample usage);
- xyyTRS**: requires the factorization returned by **xyyTRF**;
- xyyCON**: requires the norm of the original matrix A , and the factorization returned by **xyyTRF**;
- xyyRFS**: requires the original matrices A and B , the factorization returned by **xyyTRF**, and the solution X returned by **xyyTRS**;

xyyTRI: requires the factorization returned by xyyTRF.

The RFS ("refine solution") routines perform iterative refinement and compute backward and forward error bounds for the solution. Iterative refinement is done in the same precision as the input data. In particular, the residual is *not* computed with extra precision, as has been traditionally done. The benefit of this procedure is discussed in Chapter 4.

Table 2.6: Computational routines for linear equations

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
general	factorize	SGETRF	CGETRF	DGETRF	ZGETRF
	solve using factorization	SGETRS	CGETRS	DGETRS	ZGETRS
	estimate condition number	SGECON	CGECON	DGECON	ZGECON
	error bounds for solution	SGERFS	CGERFS	DGERFS	ZGERFS
	invert using factorization	SGETRI	CGETRI	DGETRI	ZGETRI
	equilibrate	SGEQU	CGEQU	DGEQU	ZGEQU
general band	factorize	SGBTRF	CGBTRF	DGBTRF	ZGBTRF
	solve using factorization	SGBTRS	CGBTRS	DGBTRS	ZGBTRS
	estimate condition number	SGBCON	CGBCON	DGBCON	ZGBCON
	error bounds for solution	SGBRFS	CGBRFS	DGBRFS	ZGBRFS
	equilibrate	SGBEQU	CGBEQU	DGBEQU	ZGBEQU
general tridiagonal	factorize	SGTTRF	CGTTRF	DGTTRF	ZGTTRF
	solve using factorization	SGTTRS	CGTTRS	DGTTRS	ZGTTRS
	estimate condition number	SGTCON	CGTCON	DGTCON	ZGTCON
	error bounds for solution	SGTRFS	CGTRFS	DGTRFS	ZGTRFS
symmetric/Hermitian positive-definite	factorize	SPOTRF	CPOTRF	DPOTRF	ZPOTRF
	solve using factorization	SPOTRS	CPOTRS	DPOTRS	ZPOTRS
	estimate condition number	SPOCON	CPOCON	DPOCON	ZPOCON
	error bounds for solution	SPORFS	CPORFS	DPORFS	ZPORFS
	invert using factorization	SPOTRI	CPOTRI	DPOTRI	ZPOTRI
	equilibrate	SPOEQU	CPOEQU	DPOEQU	ZPOEQU
symmetric/Hermitian positive-definite (packed storage)	factorize	SPPTRF	CPPTRF	DPPTRF	ZPPTRF
	solve using factorization	SPPTRS	CPPTRS	DPPTRS	ZPPTRS
	estimate condition number	SPPCON	CPPCON	DPPCON	ZPPCON
	error bounds for solution	SPPRFS	CPPRFS	DPPRFS	ZPPRFS
	invert using factorization	SPPTRI	CPPTRI	DPPTRI	ZPPTRI
	equilibrate	SPPEQU	CPPEQU	DPPEQU	ZPPEQU
symmetric/Hermitian positive-definite band	factorize	SPBTRF	CPBTRF	DPBTRF	ZPBTRF
	solve using factorization	SPBTRS	CPBTRS	DPBTRS	ZPBTRS
	estimate condition number	SPBCON	CPBCON	DPBCON	ZPBCON
	error bounds for solution	SPBRFS	CPBRFS	DPBRFS	ZPBRFS
	equilibrate	SPBEQU	CPBEQU	DPBEQU	ZPBEQU
symmetric/Hermitian positive-definite tridiagonal	factorize	SPTTRF	CPTTRF	DPTTRF	ZPTTRF
	solve using factorization	SPTTRS	CPTTRS	DPTTRS	ZPTTRS
	estimate condition number	SPTCON	CPTCON	DPTCON	ZPTCON
	error bounds for solution	SPTRFS	CPTRFS	DPTRFS	ZPTRFS

Table 2.7: Computational routines for linear equations (continued)

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
symmetric/Hermitian indefinite	factorize	SSYTRF	CHETRF	DSYTRF	ZHETRF
	solve using factorization	SSYTRS	CHETRS	DSYTRS	ZHETRS
	estimate condition number	SSYCON	CHECON	DSYCON	ZHECON
	error bounds for solution	SSYRFS	CHERFS	DSYRFS	ZHERFS
	invert using factorization	SSYTRI	CHETRI	DSYTRI	ZHETRI
complex symmetric	factorize		CSYTRF		ZSYTRF
	solve using factorization		CSYTRS		ZSYTRS
	estimate condition number		CSYCON		ZSYCON
	error bounds for solution		CSYRFS		ZSYRFS
	invert using factorization		CSYTRI		ZSYTRI
symmetric/Hermitian indefinite (packed storage)	factorize	SSPTRF	CHPTRF	DSPTRF	ZHPTRF
	solve using factorization	SSPTRS	CHPTRS	DSPTRS	ZHPTRS
	estimate condition number	SSPCON	CHPCON	DSPCON	ZHPCON
	error bounds for solution	SSPRFS	CHPRFS	DSPRFS	ZHPRFS
	invert using factorization	SSPTRI	CHPTRI	DSPTRI	ZHPTRI
complex symmetric (packed storage)	factorize		CSPTRF		ZSPTRF
	solve using factorization		CSPTRS		ZSPTRS
	estimate condition number		CSPCON		ZSPCON
	error bounds for solution		CSPRFS		ZSPRFS
	invert using factorization		CSPTRI		ZSPTRI
triangular	solve	STRTRS	CTRTRS	DTRTRS	ZTRTRS
	estimate condition number	STRCON	CTRCON	DTRCON	ZTRCON
	error bounds for solution	STRRFS	CTRRFS	DTRRFS	ZTRRFS
	invert	STRTRI	CTRTRI	DTRTRI	ZTRTRI
triangular (packed storage)	solve	STPTRS	CTPTRS	DTPTRS	ZTPTRS
	estimate condition number	STPCON	CTPCON	DTPCON	ZTPCON
	error bounds for solution	STPRFS	CTPRFS	DTPRFS	ZTPRFS
	invert	STPTRI	CTPTRI	DTPTRI	ZTPTRI
triangular band	solve	STBTRS	CTBTRS	DTBTRS	ZTBTRS
	estimate condition number	STBCON	CTBCON	DTBCON	ZTBCON
	error bounds for solution	STBRFS	CTBRFS	DTBRFS	ZTBRFS

2.3.2 Orthogonal Factorizations

LAPACK provides a number of routines for performing orthogonal factorizations (unitary in the complex case) of an m -by- n matrix A , for use in applications such as the solution of linear least squares problems. They may also be used as steps in the solution of eigenvalue or singular value problems.

The most common, and best known, of these factorizations is the **QR factorization** given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \quad m \geq n,$$

where R is an n -by- n upper triangular matrix and Q is an m -by- m orthogonal (or unitary) matrix. If A is of full rank n , then R is non-singular.

The routine xGEQRF performs the QR factorization. The matrix Q is not formed explicitly, but is represented as a product of elementary reflectors, as described in section 5.4. Users need not be aware of the details of this representation, because associated routines are provided to work with Q : xORGQR (or xUNGQR in the complex case) can generate all or part of Q , while xORMQR (or xUNMQR) can multiply a given matrix by Q or its transpose (conjugate transpose if complex).

The QR factorization can be used to solve the linear least squares problem of equation (2.1) when A is of full rank, since

$$\|b - Ax\|_2 = \left\| \begin{pmatrix} \tilde{c} - Rx \\ \hat{c} \end{pmatrix} \right\|_2, \quad \text{where } c \equiv \begin{pmatrix} \tilde{c} \\ \hat{c} \end{pmatrix} = Q^T b \quad (Q^H b \text{ in the complex case});$$

c can be computed by xORMQR (or xUNMQR) and then x is the solution of the upper triangular system

$$Rx = \tilde{c}$$

and the residual sum of squares is given by

$$\|b - Ax\|_2 = \|\hat{c}\|_2.$$

If A is not of full rank, or the rank of A is in doubt, then we can perform either a QR factorization with column pivoting or a singular value decomposition (see section 2.3.5). The **QR factorization with column pivoting** is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} P^T, \quad m \geq n,$$

where Q and R are as before and P is a permutation matrix, chosen so that

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix},$$

where R_{11} is non-singular. The so-called basic solution to the linear least squares problem can be obtained from this factorization.

Table 2.8: Computational routines for orthogonal factorizations

Type of factorization and matrix	Operation	Single precision		Double precision	
		real	complex	real	complex
QR , general	factorize with pivoting	SGEQPF	CGEQPF	DGEQPF	ZGEQPF
	factorize, no pivoting	SGEQRF	CGEQRF	DGEQRF	ZGEQRF
	generate Q	SORGQR	CUNGQR	DORGQR	ZUNGQR
	multiply matrix by Q	SORMQR	CUNMQR	DORMQR	ZUNMQR
LQ , general	factorize, no pivoting	SGELQF	CGELQF	DGELQF	ZGELQF
	generate Q	SORGLQ	CUNGLQ	DORGLQ	ZUNGLQ
	multiply matrix by Q	SORMLQ	CUNMLQ	DORMLQ	ZUNMLQ
QL , general	factorize, no pivoting	SGEQLF	CGEQLF	DGEQLF	ZGEQLF
	generate Q	SORGQL	CUNGQL	DORGQL	ZUNGQL
	multiply matrix by Q	SORMQL	CUNMQL	DORMQL	ZUNMQL
RQ , general	factorize, no pivoting	SGERQF	CGERQF	DGERQF	ZGERQF
	generate Q	SORGRQ	CUNGRQ	DORGRQ	ZUNGRQ
	multiply matrix by Q	SORMRQ	CUNMRQ	DORMRQ	ZUNMRQ
RQ , trapezoidal	factorize, no pivoting	STZRQF	CTZRQF	DTZRQF	ZTZRQF

By applying further orthogonal (or unitary) transformations from the right to the upper trapezoidal matrix $(R_{11}R_{12})$, using $xTZRQF$, R_{12} can be eliminated:

$$\begin{pmatrix} R_{11} & R_{12} \end{pmatrix} Z = \begin{pmatrix} \tilde{R}_{11} & 0 \end{pmatrix}$$

This gives the **complete orthogonal factorization**

$$A = Q \begin{pmatrix} \tilde{R}_{11} & 0 \\ 0 & 0 \end{pmatrix} Z^T$$

from which the minimum norm solution can be obtained. See Golub and Van Loan [28] for further details.

Apart from the QR factorization, other flavors of orthogonal factorization are provided, namely the LQ , QL and RQ factorizations. These may be useful when $m < n$ or when a lower triangular matrix L is required rather than an upper triangular R . In fact, all four basic factorization routines allow arbitrary m and n , so that in some cases the matrices R or L are trapezoidal rather than triangular. A routine that performs pivoting is provided only for the QR factorization.

As for the QR factorization, associated routines are provided for the LQ , QL , and RQ factorizations either to generate Q (or part of it) explicitly, or to compute matrix products of the form QC , Q^TC (or Q^HC), CQ or CQ^T (or CQ^H) without explicitly forming Q . See Table 2.8.

2.3.3 Symmetric Eigenproblem

Let A be a real symmetric or complex Hermitian n -by- n matrix. A scalar λ is called an **eigenvalue** and a nonzero column vector z the corresponding **eigenvector** if $Az = \lambda z$. λ is always real when A is real symmetric or complex Hermitian.

The basic task of the symmetric eigenproblem routines is to compute values of λ and "optionally" corresponding vectors z for a given matrix A .

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. The real symmetric tridiagonal matrix T is factorized as $T = P\Lambda P^T$, where P is orthogonal and Λ is diagonal. The diagonal entries of Λ are the eigenvalues of T and the columns of P the eigenvectors of T . The eigenvectors of A are in turn the columns of QP .

In the real case, the decomposition $A = QTQ^T$ is computed by one of the routines xSYTRD, xSPTRD, or xSBTRD, depending on whether the symmetric matrix is stored in a two-dimensional matrix, as a packed matrix, or as a band matrix. The complex analogues of these routines are called xHETRD, xHPTRD, and xHBTRD. The matrix Q is stored as a dense, packed, or banded matrix, depending on the storage mode of A . A different routine is used for each storage mode (xSYTRD, xSPTRD and xSBTRD for real A , and xHETRD, xHPTRD and xHBTRD for complex A , respectively). The matrix Q is stored in factored form by these routines. If A is real, the matrix Q may be computed explicitly with the subroutine xORGTR, or it may be multiplied by another matrix without forming Q explicitly using the subroutine xORMTR. If A is complex, one instead uses the subroutines xUNGTR and xUNMTR, respectively.

There are several routines for the computation $T = P\Lambda P^T$ 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.

xSTEQR This routine uses the implicitly shifted QR algorithm of Wilkinson. It switches between the QR and QL variants in order to handle graded matrices more effectively than the simple QL variant that is provided by the EISPACK routines IMTQL1 and IMTQL2. See [29] for details.

xSTERF This routine uses a square-root free version of QR, and can only compute all the eigenvalues. See [29] for details.

xPTEQR This routine applies to symmetric positive-definite tridiagonal matrices only. It uses a combination of Cholesky factorization and bidiagonal QR iteration (see xBDSQR) and may be significantly more accurate than the other routines. See [8, 13, 10] for details.

Table 2.9: Computational routines for the symmetric eigenproblem

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
dense symmetric (or Hermitian)	tridiagonal reduction	SSYTRD	CHETRD	DSYTRD	ZHETRD
packed symmetric (or Hermitian)	tridiagonal reduction	SSPTRD	CHPTRD	DSPTRD	ZHPTRD
band symmetric (or Hermitian)	tridiagonal reduction	SSBTRD	CHBTRD	DSBTRD	ZHBTRD
orthogonal/unitary	generate matrix after reduction by xSYTRD	SORGTR	CUNGTR	DORGTR	ZUNGTR
	multiply matrix after reduction by xSYTRD	SORMTR	CUNMTR	DORMTR	ZUNMTR
orthogonal/unitary (packed storage)	generate matrix after reduction by xSPTRD	SOPGTR	CUPGTR	DOPGTR	ZUPGTR
	multiply matrix after reduction by xSPTRD	SOPMTR	CUPMTR	DOPMTR	ZUPMTR
symmetric tridiagonal	eigenvalues/ eigenvectors	SSTEQR	CSTEQR	DSTEQR	ZSTEQR
	eigenvalues only via root-free QR	SSTERF		DSTERF	
	eigenvalues only via bisection	SSTEBZ		DSTEBZ	
	eigenvectors by inverse iteration	SSTEIN	CSTEIN	DSTEIN	ZSTEIN
symmetric tridiagonal positive-definite	eigenvalues/ eigenvectors	SPTEQR	CPTEQR	DPTEQR	ZPTEQR

xSTEBZ 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 i^{th} to the j^{th} largest. It can be highly accurate, but may be adjusted to run faster if lower accuracy is acceptable.

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

See Table 2.9.

2.3.4 Nonsymmetric Eigenproblem

Let A be a square n -by- n matrix. A scalar λ is called an **eigenvalue** and a non-zero column vector x the corresponding **right eigenvector** if $Ax = \lambda x$. A nonzero column vector y satisfying $y^H A = \lambda y^H$ is called the **left eigenvector** (the superscript H denotes conjugate-transpose). The first basic task of these routines is to compute all n values of λ and, if desired, its associated eigenvectors x and/or y for a given matrix A .

A second basic task is to compute the **Schur decomposition** of a matrix. If A is complex, then its Schur decomposition is $A = QTQ^H$, where Q is unitary and T is upper triangular. If A is real, its Schur decomposition is $A = QTQ^T$, where Q is orthogonal (the superscript T denotes transpose) and T is upper quasi-triangular; thus, T may have 2-by-2 as well as 1-by-1 blocks on its diagonal. The columns of Q are called the **Schur vectors** of A . The eigenvalues of A appear on the diagonal of T ; complex conjugate eigenvalues of a real A correspond to 2-by-2 blocks on the diagonal of T . The Schur form depends on the order of the eigenvalues on the diagonal of T and this may optionally be chosen by the user. Suppose the user chooses that $\lambda_1, \dots, \lambda_j$, $0 < j < n$, appear in the upper left corner of T . Then the first j columns of Q span the **right invariant subspace** of A corresponding to $\lambda_1, \dots, \lambda_j$.

The user may want to compute condition numbers as well as eigenvalues, eigenvectors, and the Schur form, for these quantities. Routines for this purpose are provided as well.

These computations proceed in the following stages:

1. A general matrix A is reduced to **upper Hessenberg form**. If A is real this decomposition is $A = QHQT^T$ with Q orthogonal and H zero below the first subdiagonal. If A is complex, this decomposition is $A = QHQT^H$ with Q unitary and H as before.
2. The upper Hessenberg matrix H is reduced to Schur form $H = PTPT^T$ (for H real) or $H = PTP^H$ (for H complex). The matrix P may optionally be computed as well. The eigenvalues are obtained from the diagonal of T . This is done by subroutine xHSEQR.
3. Given the eigenvalues, the eigenvectors may be computed in two different ways. xHSEIN performs inverse iteration on H to compute H 's eigenvectors, and xTREVC computes the eigenvectors of T . One may optionally transform the right eigenvectors of H (or of T) to the right eigenvectors of the original matrix A by multiplying them by Q (or by QP); the left eigenvectors may be similarly transformed.

The reduction to Hessenberg form is performed by subroutine xGEHRD, which represents Q in a factored form. If A is real, the matrix Q may be computed explicitly using subroutine xORGHR, or multiplied by another matrix without forming it using subroutine xORMHR. If A is complex, one instead uses subroutines xUNGHR and xUNMHR, respectively.

In addition, the routine xGEBAL may be used to **balance** the matrix A prior to reduction to Hessenberg form. Balancing involves applying a similarity transformation with permutation matrices to try to make A as nearly triangular as possible, and a diagonal similarity transformation to make the rows and columns of A as close in norm in possible. These transformations can improve speed

and accuracy of later processing in some cases. xGEBAL performs the balancing, and xGEBAK backtransforms the eigenvectors of the balanced matrix.

In addition to these basic routines, four other routines xTREXC, xTRSYL, xTRSNA and xTRSEN are available for further processing.

1. xTREXC will move an eigenvalue (or 2-by-2 block) on the diagonal of the Schur form from its original position to any other position. It may be used to choose the order in which eigenvalues appear in the Schur form.
2. xTRSYL solves the Sylvester matrix equation $BX + XC = D$ for X given matrices B , C and D , with B and C (quasi) triangular. It is used in the routines xTRSNA and xTRSEN, but it is also of independent interest.
3. xTRSNA computes the condition numbers of the eigenvalues and/or right eigenvectors of a matrix T in Schur form. These are the same as the condition numbers of the eigenvalues and right eigenvectors of the original matrix A from which T is derived. The user may compute these condition numbers for all eigenvalue/eigenvector pairs, or for any selected subset. For more details, see [7].
4. xTRSEN moves a selected subset of the eigenvalues of a matrix T in Schur form to the upper left corner of T , and optionally computes the condition numbers of their average value and of their right invariant subspace. These are the same as the condition numbers of the average eigenvalue and right invariant subspace of the original matrix A from which T is derived. For more details, see [7] (see Table 2.10).

2.3.5 Singular Value Decomposition

Let A be a general real m -by- n matrix. The **singular value decomposition (SVD)** of A is the factorization $A = U\Sigma V^T$, where U and V are orthogonal, the superscript T denotes transpose, and $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_r)$, $r = \min(m, n)$, and $\sigma_1 \geq \dots \geq \sigma_r \geq 0$. If A is complex, then its SVD is $A = U\Sigma V^H$ where U and V are unitary, the superscript H denotes conjugate transpose, and Σ is as before with real diagonal elements. The σ_i are called the **singular values**, the first r columns of V the **right singular vectors** and the first r columns of U the **left singular vectors**.

The routines described in this section, and listed in Table 2.11, are used to compute this decomposition. This computation proceeds in the following stages:

1. The matrix A is reduced to bidiagonal form: $A = U_1 B V_1^T$ if A is real ($A = U_1 B V_1^H$ if A is complex). Here U_1 and V_1 are orthogonal (unitary if A is complex), and B is real and bidiagonal. This means that B is nonzero only on the main diagonal and either on the first superdiagonal (if $m \geq n$) or the first subdiagonal (if $m < n$).
2. The SVD of the bidiagonal matrix B is computed: $B = U_2 \Sigma V_2^T$. Here U_2 and V_2 are orthogonal and Σ is diagonal as described above. The singular vectors of A are then $U = U_1 U_2$ and $V = V_1 V_2$.

Table 2.10: Computational routines for the nonsymmetric eigenproblem

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
general	Hessenberg reduction	SGEHRD	CGEHRD	DGEHRD	ZGEHRD
	balancing	SGEBAL	CGEBAL	DGEBAL	ZGEBAL
	backtransforming	SGEBAK	CGEBAK	DGEBAK	ZGEBAK
orthogonal/unitary	generate matrix after Hessenberg reduction	SORGHR	CUNGHR	DORGHR	ZUNGHR
	multiply matrix after Hessenberg reduction	SORMHR	CUNMHR	DORMHR	ZUNMHR
Hessenberg	Schur factorization	SHSEQR	CHSEQR	DHSEQR	ZHSEQR
	eigenvectors by inverse iteration	SHSEIN	CHSEIN	DHSEIN	ZHSEIN
(quasi)triangular	eigenvectors	STREVC	CTREVC	DTREVC	ZTREVC
	reordering eigenvalues	STREXC	CTREXC	DTREXC	ZTREXC
	Sylvester equation	STRSYL	CTRSYL	DTRSYL	ZTRSYL
	condition numbers of eigenvalues/vectors	STRSNA	CTRSNA	DTRSNA	ZTRSNA
	condition numbers of eigenvalue cluster/ invariant subspace	STRSEN	CTRSEN	DTRSEN	ZTRSEN

Table 2.11: Computational routines for the singular value decomposition

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
general	bidagonal reduction	SGBERD	CGBERD	DGBERD	ZGBERD
orthogonal/unitary	generate matrix after bidagonal reduction	SORGBR	CUNGBR	DORGBR	ZUNGBR
	multiply matrix after bidagonal reduction	SORMBR	CUNMBR	DORMBR	ZUNMBR
bidagonal	singular values/ singular vectors	SBDSQR	CBDSQR	DBDSQR	ZBDSQR

This reduction to bidagonal form is performed by the subroutine xGBERD, which represents U_1 and V_1 in factored form. If A is real, the matrices U_1 and V_1 may be computed explicitly using routine xORGBR, or multiplied by other matrices without forming them using routine xORMBR. If A is complex, one instead uses xUNGBR and xUNMBR, respectively. The SVD of the bidagonal matrix is computed by the subroutine xBDSQR. xBDSQR also has the option to multiply a separate input matrix by the transpose of the right singular vectors; this feature is used to solve least squares problems.

xBDSQR is more accurate than its counterparts in LINPACK and EISPACK: barring underflow and overflow, it computes all the singular values of B to nearly full relative precision, independent of their magnitudes. It also computes the singular vectors much more accurately. See [13, 10] for details.

2.3.6 Generalized Symmetric-Definite Eigenproblems

This section is concerned with the solution of the generalized eigenvalue problems $Ax = \lambda Bx$, $ABx = \lambda x$, and $BAx = \lambda x$, where A and B are symmetric and B is positive definite. Each of these problems can be reduced to the standard symmetric eigenvalue problem by factorizing B as either LL^T or U^TU through a Cholesky factorization and applying the factors to the matrix A .

For the matrix B , storing the lower triangle, we have $B = LL^T$,

$$Ax = \lambda Bx \Rightarrow (L^{-1}AL^{-T})(L^Tx) = \lambda(L^Tx).$$

Hence the eigenvalues of $Ax = \lambda Bx$ are those of $Cy = \lambda y$, where C is the symmetric matrix $C = L^{-1}AL^{-T}$ and $y = L^Tx$.

Similarly we have,

$$ABx = \lambda x \Rightarrow (L^TAL)(L^Tx) = \lambda(L^Tx)$$

and,

Table 2.12: Computational routines for the generalized symmetric-definite eigenproblem

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		real	complex	real	complex
symmetric/Hermitian	reduction	SSYGST	CHEGST	DSYGST	ZHEGST
symmetric/Hermitian (packed storage)	reduction	SSPGST	CHPGST	DSPGST	ZHPGST

$$BAx = \lambda x \Rightarrow (L^T AL)(L^{-1}x) = \lambda(L^{-1}x).$$

When the matrix B is stored in the upper triangle, we have $B = U^T U$,

$$Ax = \lambda Bx \Rightarrow (U^{-T} A U^{-1})(Ux) = \lambda(Ux),$$

$$ABx = \lambda x \Rightarrow (U A U^T)(Ux) = \lambda(Ux)$$

and,

$$BAx = \lambda x \Rightarrow (U A U^T)(U^{-T}x) = \lambda(U^{-T}x).$$

Given A and a Cholesky factorization of B , the routines xyyGST overwrite A with the matrix C of the corresponding standard problem $Cy = \lambda y$ (see Table 2.12). No special routines are needed to recover the eigenvectors x of the generalized problem from the eigenvectors y of the standard problem, because these computations are simple applications of Level 2 or Level 3 BLAS.

Chapter 3

Performance of LAPACK

Note: this chapter presents some performance figures for LAPACK routines. The figures are provided for illustration only, and should not be regarded as a definitive up-to-date statement of performance. They have been selected from performance figures obtained in 1990–91 during the development of LAPACK. Performance is affected by many factors that may change from time to time, such as details of hardware (cycle time, cache size), compiler, and BLAS. To obtain up-to-date performance figures, use the timing programs provided with LAPACK.

3.1 Factors That Affect Performance

Can we provide **portable** software for computations in dense linear algebra that is **efficient** on a wide range of modern high-performance computers? If so, how? Answering these questions — and providing the desired software — has been the goal of the LAPACK project.

LINPACK [15] and EISPACK [39, 27] have for many years provided high-quality portable software for linear algebra; but on modern high-performance computers they often achieve only a small fraction of the peak performance of the machines. Therefore, LAPACK has been designed to supersede LINPACK and EISPACK, principally by achieving much greater efficiency — but at the same time also adding extra functionality, using some new or improved algorithms, and integrating the two sets of algorithms into a single package.

LAPACK was originally targeted to achieve good performance on single-processor vector machines and on shared-memory multi-processor machines with a modest number of powerful processors. Since the start of the project, another class of machines has emerged for which LAPACK software is equally well-suited—the high-performance “super-scalar” workstations. (LAPACK is intended to be used across the whole spectrum of modern computers, but when considering performance, the emphasis is on machines at the more powerful end of the spectrum.)

Here we discuss the main factors that affect the performance of linear algebra software on these classes of machines.

3.1.1 Vectorization

Designing vectorizable algorithms in linear algebra is usually straightforward. Indeed, for many computations there are several variants, all vectorizable, but with different characteristics in performance (see, for example, [22]). Linear algebra algorithms can come close to the peak performance of many machines — principally because peak performance depends on some form of chaining of vector addition and multiplication operations, and this is just what the algorithms require.

However, when the algorithms are realized in straightforward Fortran 77 code, the performance may fall well short of the expected level, usually because vectorizing Fortran compilers fail to minimize the number of memory references — that is, the number of vector load and store operations. This brings us to the next factor.

3.1.2 Data movement

What often limits the actual performance of a vector—or scalar— floating-point unit is the rate of transfer of data between different levels of memory in the machine. Examples include: the transfer of vector operands in and out of vector registers, the transfer of scalar operands in and out of a high-speed scalar processor, the movement of data between main memory and a high-speed cache or local memory, and paging between actual memory and disk storage in a virtual memory system.

It is desirable to maximize the ratio of floating-point operations to memory references, and to re-use data as much as possible while it is stored in the higher levels of the memory hierarchy (for example, vector registers or high-speed cache).

A Fortran programmer has no explicit control over these types of data movement, although one can often influence them by imposing a suitable structure on an algorithm.

3.1.3 Parallelism

The nested loop structure of most linear algebra algorithms offers considerable scope for loop-based parallelism on shared-memory machines. This is the principal type of parallelism that LAPACK at present aims to exploit. It can sometimes be generated automatically by a compiler, but often requires the insertion of compiler directives.

3.2 The BLAS as the Key To Portability

How then can we hope to be able to achieve sufficient control over vectorization, data movement, and parallelism in portable Fortran code, to obtain the levels of performance that machines can offer?

The LAPACK strategy for combining efficiency with portability is to construct the software as much as possible out of calls to the BLAS (Basic Linear Algebra Subprograms); the BLAS are used as building blocks.

Table 3.1: Speed in megaflops of Level 2 and Level 3 BLAS operations on a CRAY Y-MP

(all matrices are of order 500; U is upper triangular)

Number of processors:	1	2	4	8
Level 2: $y \leftarrow \alpha Ax + \beta y$	311	611	1197	2285
Level 3: $C \leftarrow \alpha AB + \beta C$	312	623	1247	2425
Level 2: $x \leftarrow Ux$	293	544	898	1613
Level 3: $B \leftarrow UB$	310	620	1240	2425
Level 2: $x \leftarrow U^{-1}x$	272	374	479	584
Level 3: $B \leftarrow U^{-1}B$	309	618	1235	2398

The efficiency of LAPACK software depends on efficient implementations of the BLAS being provided by computer vendors (or others) for their machines. Thus the BLAS form a low-level interface between LAPACK software and different machine architectures. Above this level, almost all of the LAPACK software is truly portable.

There are now three levels of BLAS:

Level 1 BLAS [36]: for vector operations, such as $y \leftarrow \alpha x + y$

Level 2 BLAS [19]: for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$

Level 3 BLAS [17]: for matrix-matrix operations, such as $C \leftarrow \alpha AB + \beta C$

Here, A , B and C are matrices, x and y are vectors, and α and β are scalars.

The Level 1 BLAS are used in LAPACK, but for convenience rather than for performance: they perform an insignificant fraction of the computation, and they cannot achieve high efficiency on most modern supercomputers.

The Level 2 BLAS can achieve near-peak performance on many vector-processors, such as a single processor of a CRAY X-MP or Y-MP, or Convex C-2 machine. However on other vector processors, such as a CRAY-2 or an IBM 3090 VF, their performance is limited by the rate of data movement between different levels of memory.

This limitation is overcome by the Level 3 BLAS, which perform $O(n^3)$ floating-point operations on $O(n^2)$ data, whereas the Level 2 BLAS perform only $O(n^2)$ operations on $O(n^2)$ data.

The BLAS also allow us to exploit parallelism in a way that is transparent to the software that calls them. Even the Level 2 BLAS offer some scope for exploiting parallelism, but greater scope is provided by the Level 3 BLAS, as Table 3.1 illustrates.

3.3 Block Algorithms And Their Derivation

It is comparatively straightforward to recode many of the algorithms in LINPACK and EISPACK so that they call Level 2 BLAS. Indeed, in the simplest cases the same floating-point operations are performed, possibly even in the same order: it is just a matter of reorganizing the software. To illustrate this point we derive the Cholesky factorization algorithm that is used in the LINPACK routine SPOFA, which factorizes a symmetric positive-definite matrix as $A = U^T U$. Writing these equations as:

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ . & a_{jj} & \alpha_j^T \\ . & . & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}$$

and equating coefficients of the j^{th} column, we obtain:

$$\begin{aligned} a_j &= U_{11}^T u_j \\ a_{jj} &= u_j^T u_j + u_{jj}^2 \end{aligned}$$

Hence, if U_{11} has already been computed, we can compute u_j and u_{jj} from the equations:

$$\begin{aligned} U_{11}^T u_j &= a_j \\ u_{jj}^2 &= a_{jj} - u_j^T u_j \end{aligned}$$

Here is the body of the code of the LINPACK routine SPOFA, which implements the above method:

```

DO 30 J = 1, N
  INFO = J
  S = 0.0E0
  JM1 = J - 1
  IF (JM1 .LT. 1) GO TO 20
  DO 10 K = 1, JM1
    T = A(K,J) - SDOT(K-1,A(1,K),1,A(1,J),1)
    T = T/A(K,K)
    A(K,J) = T
    S = S + T*T
  10  CONTINUE
  20  CONTINUE
  S = A(J,J) - S
C      .....EXIT
      IF (S .LE. 0.0E0) GO TO 40
      A(J,J) = SQRT(S)
  30  CONTINUE

```

And here is the same computation recoded in "LAPACK-style" to use the Level 2 BLAS routine STRSV (which solves a triangular system of equations). The call to STRSV has replaced the loop over K which made several calls to the Level 1 BLAS routine SDOT. (For reasons given below, this is not the actual code used in LAPACK — hence the term "LAPACK-style".)

```

DO 10 J = 1, N
  CALL STRSV( 'Upper', 'Transpose', 'Non-unit', J-1, A, LDA,
$           A(1,J), 1 )
  S = A(J,J) - SDOT( J-1, A(1,J), 1, A(1,J), 1 )
  IF( S.LE.ZERO ) GO TO 20
  A(J,J) = SQRT( S )
10 CONTINUE

```

This change by itself is sufficient to make big gains in performance on a number of machines— for example, from 72 to 251 megaflops for a matrix of order 500 on one processor of a CRAY Y-MP. Since this is 81% of the peak speed of matrix-matrix multiplication on this processor, we cannot hope to do very much better by using Level 3 BLAS.

On an IBM 3090E VF (using double precision) there is virtually no difference in performance between the LINPACK-style and the LAPACK-style code. Both run at about 23 megaflops. This is unsatisfactory on a machine on which matrix-matrix multiplication can run at 75 megaflops. To exploit the faster speed of Level 3 BLAS, the algorithms must undergo a deeper level of restructuring, and be re-cast as a **block algorithm** — that is, an algorithm that operates on **blocks** or submatrices of the original matrix.

To derive a block form of Cholesky factorization, we write the defining equation in partitioned form thus:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ . & A_{22} & A_{23} \\ . & . & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

Equating submatrices in the second block of columns, we obtain:

$$\begin{aligned} A_{12} &= U_{11}^T U_{12} \\ A_{22} &= U_{12}^T U_{12} + U_{22}^T U_{22} \end{aligned}$$

Hence, if U_{11} has already been computed, we can compute U_{12} as the solution to the equation

$$U_{11}^T U_{12} = A_{12}$$

by a call to the Level 3 BLAS routine STRSM; and then we can compute U_{22} from

$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

This involves first updating the symmetric submatrix A_{22} by a call to the Level 3 BLAS routine SSYRK, and then computing its Cholesky factorization. Since Fortran does not allow recursion, a separate routine must be called (using Level 2 BLAS rather than Level 3), named SPOTF2 in the code below. In this way successive blocks of columns of U are computed. Here is LAPACK-style code for the block algorithm. In this code-fragment NB denotes the width of the blocks.

Table 3.2: Speed in megaflops of Cholesky factorization $A = U^T U$ for $n = 500$

Machine:	IBM 3090 VF	CRAY Y-MP	CRAY Y-MP
Number of processors:	1	1	8
<i>j</i> -variant: LINPACK	23	72	72
<i>j</i> -variant: using Level 2 BLAS	24	251	378
<i>j</i> -variant: using Level 3 BLAS	49	287	1225
<i>i</i> -variant: using Level 3 BLAS	50	290	1414

```

DO 10 J = 1, N, NB
  JB = MIN( NB, N-J+1 )
  CALL STRSM( 'Left', 'Upper', 'Transpose', 'Non-unit', J-1, JB,
$           ONE, A, LDA, A(1,J), LDA )
  CALL SSYRK( 'Upper', 'Transpose', JB, J-1, -ONE, A(1,J), LDA,
$           ONE, A(J,J), LDA )
  CALL SPOTF2( JB, A(J,J), LDA, INFO )
  IF( INFO.NE.0 ) GO TO 20
10 CONTINUE

```

This code runs at 49 megaflops on a 3090, more than double the speed of the LINPACK code. On a CRAY Y-MP, the use of Level 3 BLAS squeezes a little more performance out of one processor, but makes a large improvement when using all 8 processors.

But that is not the end of the story, and the code given above is not the code that is actually used in the LAPACK routine SPOTRF. We mentioned in subsection 3.1.1 that for many linear algebra computations there are several vectorizable variants, often referred to as *i*-, *j*- and *k*-variants, according to a convention introduced in [22] and used in [28]. The same is true of the corresponding block algorithms.

It turns out that the *j*-variant that was chosen for LINPACK, and used in the above examples, is not the fastest on many machines, because it is based on solving triangular systems of equations, which can be significantly slower than matrix-matrix multiplication. The variant actually used in LAPACK is the *i*-variant, which does rely on matrix-matrix multiplication.

Table 3.2 summarizes the results.

3.4 Examples of block algorithms in LAPACK

Having discussed in detail the derivation of one particular block algorithm, we now describe examples of the performance that has been achieved with a variety of block algorithms.

See Gallivan *et al.* [26] and Dongarra *et al.* [20] for an alternative survey of algorithms for dense linear algebra on high-performance computers.

Table 3.3: Speed in megaflops of SGETRF/DGETRF for square matrices of order n

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000-530	1	32	19	25	29	31	33
Alliant FX/8	8	16	9	26	32	46	57
IBM 3090J VF	1	64	23	41	52	58	63
Convex C-240	4	64	31	60	82	100	112
CRAY Y-MP	1	1	132	219	254	272	283
CRAY-2	1	64	110	211	292	318	358
Siemens/Fujitsu VP 400-EX	1	64	46	132	222	309	397
NEC SX2	1	1	118	274	412	504	577
CRAY Y-MP	8	64	195	556	920	1188	1408

Table 3.4: Speed in megaflops of SPOTRF/DPOTRF for matrices of order n with UPLO = 'U'

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000-530	1	32	21	29	34	36	38
Alliant FX/8	8	16	10	27	40	49	52
IBM 3090J VF	1	48	26	43	56	62	67
Convex C-240	4	64	32	63	82	96	103
CRAY Y-MP	1	1	126	219	257	275	285
CRAY-2	1	64	109	213	294	318	362
Siemens/Fujitsu VP 400-EX	1	1	53	145	237	312	369
NEC SX2	1	1	155	387	589	719	819
CRAY Y-MP	8	32	146	479	845	1164	1393

3.4.1 Factorizations for solving linear equations

The well-known LU and Cholesky factorizations are the simplest block algorithms to derive. No extra floating-point operations nor extra working storage are required.

Table 3.3 illustrates the speed of the LAPACK routine for LU factorization of a real matrix, SGETRF in single precision on CRAY machines, and DGETRF in double precision on all other machines. Double precision corresponds to 64-bit floating point arithmetic on all machines tested. A block size of 1 means that the unblocked algorithm is used, since it is faster than — or at least as fast as — a blocked algorithm.

Table 3.4 gives similar results for Cholesky factorization, extending the results given in Table 3.2.

LAPACK, like LINPACK, provides a factorization for symmetric indefinite matrices, so that A is factorized as $PUDU^TP^T$, where P is a permutation matrix, and D is block diagonal with blocks

Table 3.5: Speed in megaflops of SSYTRF for matrices of order n with UPLO = 'U' on a CRAY-2

Block size	Values of n				
	100	200	300	400	500
1	75	128	154	164	176
64	78	160	213	249	281

of order 1 or 2. A block form of this algorithm has been derived, and is implemented in the LAPACK routine SSYTRF/DSYTRF. It has to duplicate a little of the computation in order to "look ahead" to determine the necessary row and column interchanges, but the extra work is more than compensated for by the greater speed of updating the matrix by blocks, as is illustrated in Table 3.5.

LAPACK, like LINPACK, provides LU and Cholesky factorizations of band matrices. The LINPACK algorithms can easily be restructured to use Level 2 BLAS, though that has little effect on performance for matrices of very narrow bandwidth. It is also possible to use Level 3 BLAS, at the price of doing some extra work with zero elements outside the band [25]. This becomes worthwhile for matrices of large order and semi-bandwidth greater than 100 or so.

3.4.2 QR factorization

The traditional algorithm for QR factorization is based on the use of elementary Householder matrices of the general form

$$H = I - \tau vv^T$$

where v is a column vector and τ is a scalar. This leads to an algorithm with very good vector performance, especially if coded to use Level 2 BLAS.

The key to developing a block form of this algorithm is to represent a product of b elementary Householder matrices of order n as a block form of a Householder matrix. This can be done in various ways. LAPACK uses the following form [38]:

$$H_1 H_2 \dots H_b = I - VTV^T$$

where V is an n -by- b matrix whose columns are the individual vectors v_1, v_2, \dots, v_b associated with the Householder matrices H_1, H_2, \dots, H_b , and T is an upper triangular matrix of order b . Extra work is required to compute the elements of T , but once again this is compensated by the greater speed of applying the block form. Table 3.6 summarizes results obtained with the LAPACK routine SGEQRF/DGEQRF.

3.4.3 Eigenvalue problems

Eigenvalue problems have so far provided a less fertile ground for the development of block algorithms than the factorizations so far described. Nevertheless, useful improvements in performance have been obtained.

Table 3.6: Speed in megaflops of SGEQRF/DGEQRF for square matrices of order n

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000-530	1	32	18	26	30	32	34
Alliant FX/8	8	16	11	28	39	47	50
IBM 3090J VF	1	32	28	54	68	75	80
Convex C-240	4	16	35	65	82	97	106
CRAY Y-MP	1	1	177	253	276	286	292
CRAY-2	1	32	105	208	269	303	326
Siemens/Fujitsu VP 400-EX	1	1	101	237	329	388	426
NEC SX2	1	1	217	498	617	690	768

The first step in solving many types of eigenvalue problems is to reduce the original matrix to a “condensed form” by orthogonal transformations.

In QR factorizations, the unblocked algorithms all use elementary Householder matrices and have good vector performance. Block forms of these algorithms have been developed [23], but all require additional operations, and a significant proportion of the work must still be performed by Level 2 BLAS, so there is less possibility of compensating for the extra operations.

The algorithms concerned are:

- reduction of a symmetric matrix to tridiagonal form to solve a symmetric eigenvalue problem: LAPACK routine SSYTRD applies a symmetric block update of the form

$$A \leftarrow A - UX^T - XU^T$$

using the Level 3 BLAS routine SSYR2K; Level 3 BLAS account for at most half the work.

- reduction of a rectangular matrix to bidiagonal form to compute a singular value decomposition: LAPACK routine SGEHRD applies a block update of the form

$$A \leftarrow A - UX^T - YV^T$$

using two calls to the Level 3 BLAS routine SGEMM; Level 3 BLAS account for at most half the work.

- reduction of a nonsymmetric matrix to Hessenberg form to solve a nonsymmetric eigenvalue problem: LAPACK routine SGEHRD applies a block update of the form

$$A \leftarrow (I - VT^TV^T)(A - XV^T)$$

Level 3 BLAS account for at most three-quarters of the work.

Table 3.7: Speed in megaflops of reductions to condensed forms on an IBM 3090E VF

(all matrices are square of order n)

	Block size	Values of n			
		128	256	384	512
SSYTRD	1	15	22	26	27
	16	15	26	32	34
SGEBRD	1	23	26	28	29
	12	23	33	38	41
SGEHRD	1	27	29	30	30
	24	36	51	57	58

Note that only in the reduction to Hessenberg form is it possible to use the block Householder representation described in subsection 3.4.2. Extra work must be performed to compute the n -by- b matrices X and Y that are required for the block updates (b is the block-size) — and extra workspace is needed to store them.

Nevertheless, the performance gains can be worthwhile on some machines, for example, on an IBM 3090, as shown in Table 3.7.

Following the reduction to condensed form, there is no scope for using Level 2 or Level 3 BLAS in computing the eigenvalues and eigenvectors of a symmetric tridiagonal matrix, or in computing the singular values and vectors of a bidiagonal matrix.

However, for computing the eigenvalues and eigenvectors of a Hessenberg matrix—or rather for computing its Schur factorization—yet another flavour of block algorithm has been developed: a **multishift** QR iteration [6]. Whereas the traditional EISPACK routine HQR uses a double shift (and the corresponding complex routine COMQR uses a single shift), the multishift algorithm uses block shifts of higher order. It has been found that the total number of operations *decreases* as the order of shift is increased until a minimum is reached typically between 4 and 8; for higher orders the number of operations increases quite rapidly. Because the speed of applying the shift increases steadily with the order, the optimum order of shift is typically in the range 8-16.

Chapter 4

Accuracy and Stability

In addition to providing faster routines than previously available, LAPACK provides more comprehensive and better error bounds.

Our ultimate goal is to provide error bounds for all quantities computed by LAPACK, although this work is not yet complete. It is beyond the scope of this manual to prove all these error bounds are valid. Instead, we explain the overall approach, provide enough information to use the software, and give references for further explanation. The leading comments of the individual routines should be consulted for details. Much standard material on error analysis can be found in [28].

Traditional error bounds are based on the fact that the algorithms in LAPACK, like their predecessors in LINPACK and EISPACK, are **normwise backward stable**; the tighter error bounds provided by some LAPACK routines depend on algorithms which satisfy a stronger criterion called **componentwise relative backward stability**.

In section 4.1 we discuss roundoff error. Section 4.2 discusses the vector and matrix norms we need to measure errors, as well as other notation. Standard *normwise* error bounds satisfied by LAPACK (as well as LINPACK and EISPACK) routines are reviewed in section 4.3. Section 4.4 discusses the new *componentwise* approach to error analysis used in some LAPACK routines. Section 4.5 discusses how to read and understand the error bounds stated in the following sections, 4.6 through 4.11, which present bounds for linear equation solving, least squares problems, the singular value decomposition, the symmetric eigenproblem, the nonsymmetric eigenproblem, and the generalized symmetric-definite eigenproblem, respectively. Section 4.12 discusses the impact of fast Level 3 BLAS on the accuracy of LAPACK routines.

4.1 Roundoff Errors in Floating Point Arithmetic

We will let ϵ denote the *machine precision*, which is loosely described as the largest relative error in any floating point operation which neither overflows nor underflows. In other words, it is the smallest number satisfying

$$|fl(a \oplus b) - (a \oplus b)| \leq \epsilon \cdot |a \oplus b|$$

where a and b are floating point numbers, \oplus is one of the four operations $+$, $-$, \times and \div , and $fl(a \oplus b)$ is the floating point result of $a \oplus b$. A precise characterization of ϵ depends on the details of the machine arithmetic and even of the compiler. For example, if addition and subtraction are implemented without a guard digit¹ we must redefine ϵ to be the smallest number such that

$$|fl(a \pm b) - (a \pm b)| \leq \epsilon \cdot (|a| + |b|)$$

There are many other parameters required to specify computer arithmetic, such as the overflow threshold, underflow threshold, and so on. In order that LAPACK be portable, they are computed at runtime by the auxiliary routine xLAMCH².

Throughout our discussion, we will ignore overflow and significant underflow in discussing error bounds.

LAPACK routines are generally insensitive to the details of rounding, just as their counterparts in LINPACK and EISPACK. One newer algorithm (xLASV2) can return significantly more accurate results if addition and subtraction have a guard digit (see the end of section 4.8). Future releases of LAPACK will contain more routines whose performance depends strongly on having accurate and robust arithmetic, such as IEEE Standard Floating Point Arithmetic [3].

4.2 Vector and Matrix Norms

Loosely speaking, a norm of a vector or matrix measures the size of its largest entry. This is true for the norms we shall use, which are defined in Table 4.1.

Table 4.1: Vector and Matrix Norms

	Vector	Matrix
infinity-norm	$\ x\ _\infty = \max_i x_i $	$\ A\ _\infty = \max_i \sum_j a_{ij} $
one-norm	$\ x\ _1 = \sum_i x_i $	$\ A\ _1 = \max_j \sum_i a_{ij} $
two-norm	$\ x\ _2 = (\sum_i x_i ^2)^{1/2}$	$\ A\ _2 = \max_{x \neq 0} \ Ax\ _2 / \ x\ _2$
Frobenius norm	$\ x\ _F = \ x\ _2$	$\ A\ _F = (\sum_{ij} a_{ij} ^2)^{1/2}$

The two-norm of A , $\|A\|_2$, is the **largest singular value** $\sigma_{\max}(A)$ of A . The **smallest singular value**, $\min_{x \neq 0} \|Ax\|_2 / \|x\|_2$, is denoted $\sigma_{\min}(A)$. These last two definitions make sense for rectangular A as well (if A has more columns than rows, transpose A in the definition of σ_{\min}). The two norm, Frobenius norm, and singular values of a matrix do not change if it is multiplied by a real orthogonal (or complex unitary) matrix.

$\kappa_p(A)$ will denote $\|A\|_p \cdot \|A^{-1}\|_p$ for $p = 1, 2$ and ∞ , and A square and invertible.

We will denote the vector of absolute values of x by $|x|$ ($|x|_i = |x_i|$), and similarly for $|A|$ ($|A|_{ij} = |a_{ij}|$). The dimensions of A will be n by n if not otherwise specified.

¹This is the case on Cybers and current Crays.

²See subsection 2.1.3 for explanation of the naming convention used for LAPACK routines.

4.3 Standard Error Analysis

We illustrate standard error analysis with the simple example of evaluating the scalar function $y = f(z)$. Let the output of the subroutine which implements $f(z)$ be denoted $\text{alg}(z)$; this includes the effects of roundoff. If $\text{alg}(z) = f(z + \delta)$ where δ is small, then we say alg is a **backward stable** algorithm for f , or that the **backward error** δ is small. In other words, $\text{alg}(z)$ is the exact value of f at a slightly perturbed input $z + \delta$.³

Suppose now that f is a smooth function, so that we may approximate it near z by a straight line: $f(z + \delta) \approx f(z) + f'(z) \cdot \delta$. Then we have the simple error estimate

$$\text{alg}(z) - f(z) = f(z + \delta) - f(z) \approx f'(z) \cdot \delta$$

Thus, if δ is small, and the derivative $f'(z)$ is moderate, the error $\text{alg}(z) - f(z)$ will be small⁴. This is often written in the similar form

$$\left| \frac{\text{alg}(z) - f(z)}{f(z)} \right| \lesssim \left| \frac{f'(z) \cdot z}{f(z)} \right| \cdot \left| \frac{\delta}{z} \right| \equiv \kappa(f, z) \cdot \left| \frac{\delta}{z} \right|$$

This approximately bounds the **relative error** $\frac{\text{alg}(z) - f(z)}{f(z)}$ by the product of the **condition number of f at z** , $\kappa(f, z)$, and the **relative backward error** $|\frac{\delta}{z}|$. Thus we get an error bound by multiplying a condition number and a backward error (or bounds for these quantities). We call a problem **ill-conditioned** if its condition number is large, and **ill-posed** if its condition number is infinite (or does not exist)⁵.

If f and z are vector quantities, then $f'(z)$ is a matrix (the Jacobian). So instead of using absolute values as before, we now measure δ by a vector norm $\|\delta\|$ and $f'(z)$ by a matrix norm $\|f'(z)\|$. The conventional (and coarsest) error analysis uses the infinity norm (or similar norm). We therefore call this **normwise backward stability**. For example, a normwise stable method for solving a system of linear equations $Ax = b$ will produce a solution \hat{x} satisfying $(A + E)\hat{x} = b + f$ where $\|E\|_\infty/\|A\|_\infty$ and $\|f\|_\infty/\|b\|_\infty$ are both small (close to ϵ). In this case the condition number is $\kappa_\infty(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty$ (see section 4.6 below).

Almost all the algorithms in LAPACK (as well as LINPACK and EISPACK) are stable in the sense just described⁶: when applied to a matrix A they produce the exact result for a slightly different matrix $A + E$, where $\|E\|_\infty/\|A\|_\infty$ is near ϵ .

³Sometimes our algorithms satisfy only $\text{alg}(z) = f(z + \delta) + \eta$ where both δ and η are small. This does not significantly change the following analysis.

⁴More generally, we only need Lipschitz continuity of f , and may use the Lipschitz constant in place of f' in deriving error bounds.

⁵This is a different use of the term ill-posed than used in other contexts. For example, to be well-posed (not ill-posed) in the sense of Hadamard, it is sufficient for f to be continuous, whereas we require Lipschitz continuity.

⁶There are some caveats to this statement. When computing the inverse of a matrix, the backward error E is small taking the columns of the computed inverse one at a time, with a different E for each column [24]. The same is true when computing the eigenvectors of a nonsymmetric matrix. When computing the eigenvalues and eigenvectors of $A - \lambda B$, $AB - \lambda I$ or $BA - \lambda I$, with A symmetric and B symmetric and positive definite (using SSYGV or CHEGV) then the method may not be backward normwise stable if B has a large condition number $\kappa_\infty(B)$, although it has useful error bounds in this case too (see section 4.11). Solving the Sylvester equation $AX + XB = C$ for the matrix X may not be backward stable, although there are again useful error bounds for X .

Condition numbers may be expensive to compute exactly. For example, it costs $O(n^3)$ operations to solve $Ax = b$ for a general matrix A , and computing $\kappa_\infty(A)$ exactly is at least three times as expensive. But $\kappa_\infty(A)$ can be estimated in only $O(n^2)$ operations beyond those necessary for solution. Therefore, most of LAPACK's condition numbers and error bounds are based on estimated condition numbers, using the method of [30, 32, 33]. The price one pays for using an estimator is occasional (but very rare) underestimates; years of experience attest to the reliability of our estimators, although examples where they badly underestimate can be constructed [34]. In particular, once an estimate is large enough (usually $O(1/\epsilon)$) it means that the computed answer may be completely incorrect, but the condition estimate itself may be a serious underestimate.

4.4 Improved Error Bounds

The standard error analysis just outlined has a drawback: by using the infinity norm $\|\delta\|_\infty$ to measure the backward error, entries of equal magnitude in δ contribute equally to the final error bound $\kappa(f, z)(\|\delta\|/\|z\|)$. This means that if z is sparse or has some very tiny entries, a normwise backward stable algorithm may make very large changes in these entries compared to their original values. If these tiny values are known accurately by the user, these errors may be unacceptable, or the error bounds may be unacceptably large.

For example, consider solving a diagonal system of linear equations $Ax = b$. Each component of the solution is computed accurately by Gaussian elimination: $x_i = b_i/a_{ii}$. The usual error bound is approximately $\epsilon \cdot \kappa_\infty(A) = \epsilon \cdot \max_i |a_{ii}| / \min_i |a_{ii}|$, which can arbitrarily overestimate the true error.

LAPACK addresses this inadequacy by providing some algorithms whose backward error δ is a tiny relative change in each component of z : $|\delta_i| = O(\epsilon)|z_i|$. This backward error retains both the sparsity structure of z as well as the information in tiny entries. These algorithms are therefore called **componentwise relative backward stable**. Furthermore, computed error bounds reflect this tinier backward error⁷.

If the input data has independent uncertainty in each component, each component must have at least a small *relative* uncertainty, since each is a floating point number. In this case, the extra uncertainty contributed by the algorithm is not much worse than the uncertainty in the input data, so one could say the answer provided by a componentwise relative backward stable algorithm is as accurate as the data deserves [1].

When solving $Ax = b$ using expert driver xyySVX or computational routine xyyRFS, for example, this means that we (almost always) compute \hat{x} satisfying $(A + E)\hat{x} = b + f$, where e_{ij} is a small relative change in a_{ij} and f_k is a small relative change in b_k . In particular, if A is diagonal, the corresponding error bound is always tiny, as one would expect (see the next section).

LAPACK can achieve this accuracy for linear equation solving, the bidiagonal singular value decomposition, the symmetric tridiagonal eigenproblem, and provides facilities for achieving this accuracy

⁷For other algorithms, the answers (and computed error bounds) are as accurate as though the algorithms were componentwise relative backward stable, even though they are not. These algorithms are called *forward componentwise relative stable*.

for least squares problems. Future versions of LAPACK will also achieve this accuracy for other linear algebra problems, as discussed below.

4.5 How to Read Error Bounds

Here we discuss some notation used in all the error bounds of later subsections.

All our bounds will contain the factor $p(n)$ (or $p(m, n)$), which grows as a function of matrix dimension n (or matrix dimensions m and n). It measures how errors can grow as a function of matrix dimension, and represents a potentially different function for each problem. In practice, it usually grows just linearly; $p(n) \leq 10n$ is often true. But we can generally only prove much weaker bounds of the form $p(n) = O(n^3)$, since we can not rule out the extremely unlikely possibility of rounding errors all adding together instead of canceling on average. Using $p(n) = O(n^3)$ would give very pessimistic and unrealistic bounds, especially for large n , so we content ourselves with describing $p(n)$ as a “modestly growing” function of n . For detailed derivations of various $p(n)$, see [28, 43].

There is also one situation where $p(n)$ can grow as large as 2^{n-1} : Gaussian elimination. This only occurs on specially constructed matrices presented in numerical analysis courses [43, p. 212]. Thus we can assume $p(n) \leq 10n$ in practice for Gaussian elimination too.

For linear equation and least squares solvers for $Ax = b$, we will bound the relative error $\|x - \hat{x}\|/\|x\|$ in the computed solution \hat{x} where x is the true solution (the choice of norm $\|\cdot\|$ will differ). For eigenvalue problems we bound the error $|\lambda_i - \hat{\lambda}_i|$ in the i -th computed eigenvalue $\hat{\lambda}_i$, where λ_i is the true i -th eigenvalue. For singular value problems we similarly bound $|\sigma_i - \hat{\sigma}_i|$.

Bounding the error in computed eigenvectors and singular vectors \hat{v}_i is more subtle because these vectors are not unique: even though we restrict $\|\hat{v}_i\|_2 = 1$ and $\|v_i\|_2 = 1$, we may still multiply them by arbitrary constants of absolute value 1. So to avoid ambiguity we bound the *angular difference* between \hat{v}_i and the true vector v_i :

$$\begin{aligned}\theta(v_i, \hat{v}_i) &= \text{acute angle between } v_i \text{ and } \hat{v}_i \\ &= \arccos |v_i^H \hat{v}_i|\end{aligned}\tag{4.1}$$

When $\theta(v_i, \hat{v}_i)$ is small, one can choose a constant α with absolute value 1 so that $\|\alpha v_i - \hat{v}_i\|_2 \approx \theta(v_i, \hat{v}_i)$.

In addition to bounds for individual eigenvectors, we supply bounds for the spaces spanned by collections of eigenvectors, because these may be much more accurately determined than the individual eigenvectors which span them. These spaces are called *invariant subspace* in the case of eigenvectors, because if v is any vector in the space, Av is also in the space, where A is the matrix. Again, we will use angle to measure the difference between a computed space \hat{S} and the true space S :

$$\begin{aligned}\theta(S, \hat{S}) &= \text{acute angle between } S \text{ and } \hat{S} \\ &= \max_{\substack{s \in S \\ s \neq 0}} \min_{\substack{\hat{s} \in \hat{S} \\ \hat{s} \neq 0}} \theta(s, \hat{s}) \text{ or } \max_{\substack{\hat{s} \in \hat{S} \\ \hat{s} \neq 0}} \min_{\substack{s \in S \\ s \neq 0}} \theta(s, \hat{s})\end{aligned}\tag{4.2}$$

We may compute $\theta(S, \hat{S})$ as follows. Let S be a matrix whose columns are orthonormal and span S . Similarly let \hat{S} be a orthonormal matrix with columns spanning \hat{S} . Then

$$\theta(S, \hat{S}) = \arccos \sigma_{\min}(S^H \hat{S})$$

Finally, we remark on the accuracy of our bounds when they are large. Relative errors like $\|\hat{x} - x\|/\|x\|$ and angular errors like $\theta(\hat{v}_i, v_i)$ are only of interest when they are much less than 1. We have correspondingly stated some bounds so that they are not strictly true when they are close to 1, since rigorous bounds would have been more complicated and supplied little extra information in the interesting case of small errors. We have indicated these bounds by using the symbol \lesssim , or “approximately less than”, instead of the usual \leq . Thus, when these bounds are close to 1 or greater, they indicate that the computed answer may have no significant digits at all, but do not otherwise bound the error.

4.6 Error Bounds for Linear Equation Solving

The conventional error analysis of linear equation solving goes as follows. Let $Ax = b$ be the system to be solved. Let \hat{x} be the solution computed by LAPACK (or LINPACK) using any of their linear equation solvers. Let r be the residual $r = b - A\hat{x}$. In the absence of rounding error r would be zero and \hat{x} would equal x ; with rounding error one can only say the following:

The normwise backward error ω_∞ , measured using the infinity norm, is the smallest value of

$$\max \left(\frac{\|E\|_\infty}{\|A\|_\infty}, \frac{\|f\|_\infty}{\|b\|_\infty} \right)$$

such that the computed solution \hat{x} exactly satisfies $(A + E)\hat{x} = b + f$. The normwise backward error is given by

$$\omega_\infty = \frac{\|r\|_\infty}{\|A\|_\infty \cdot \|\hat{x}\|_\infty + \|b\|_\infty} \leq p(n) \cdot \epsilon$$

where $p(n)$ is a modestly growing function of n . The corresponding condition number is $\kappa_\infty(A) \equiv \|A\|_\infty \cdot \|A^{-1}\|_\infty$. The error $x - \hat{x}$ is bounded by

$$\frac{\|x - \hat{x}\|_\infty}{\|x\|_\infty} \lesssim 2 \cdot \omega_\infty \cdot \kappa_\infty(A)$$

Approximations of $\kappa_\infty(A)$ are computed by computational routines `xyeCON` (subsection 2.3.1) or LAPACK driver routines `xyySVX` (subsection 2.2.1).

Driver `xyySVX` returns an estimate of $1/\kappa_\infty(A)$ (called `RCOND`).

As stated in the last section, this approach does not respect the presence of zero or tiny entries in A . In contrast, the LAPACK computational routines `xyyRFS` (subsection 2.3.1) or driver routines `xyySVX` (subsection 2.2.1) will (except in rare cases) compute a solution \hat{x} with the following properties:

The componentwise backward error ω_c is the smallest value of

$$\max_{i,j,k} \left(\frac{|e_{ij}|}{|a_{ij}|}, \frac{|f_k|}{|b_k|} \right)$$

(where we interpret $0/0$ as 0) such that the computed solution \hat{x} exactly satisfies $(A + E)\hat{x} = b + f$. The componentwise backward error is given by

$$\omega_c = \max_i \frac{|r_i|}{(|A| \cdot |\hat{x}| + |b|)_i} \leq p(n) \cdot \epsilon$$

where $p(n)$ is a modestly growing function of n . In other words, \hat{x} is the exact solution of the perturbed problem $(A + E)\hat{x} = b + f$ where E and f are small relative perturbations in each entry of A and b , respectively. The corresponding condition number is $\kappa_c(A, b, \hat{x}) \equiv \|A^{-1}(|A| \cdot |\hat{x}| + |b|)\|_\infty / \|\hat{x}\|_\infty$. The error $x - \hat{x}$ is bounded by

$$\frac{\|x - \hat{x}\|_\infty}{\|\hat{x}\|_\infty} \leq \omega_c \cdot \kappa_c(A, b, \hat{x}).$$

The routines xyyRFS and xyySVX return bounds on the componentwise relative backward error ω_c (called BERR) and the actual error $\|x - \hat{x}\|_\infty / \|\hat{x}\|_\infty$ (called FERR). xyySVX also returns an upper bound RCOND on the reciprocal of $\kappa_\infty(A)$.

Even in the rare cases where xyyRFS fails to make ω_c close to its minimum ϵ , the error bound computed by the routine may remain small. See [4] for details.

4.7 Error Bounds for Linear Least Squares Problems

The conventional error analysis of linear least squares problems goes as follows. The problem is to find the x minimizing $\|Ax - b\|_2$. Let \hat{x} be the solution computed by LAPACK using one of the least squares drivers xGELS, xGELSS or xGELSX (see subsection 2.2.2). We discuss the most common case, where A is overdetermined (i.e., has more rows than columns) and has full rank [28]:

The computed solution \hat{x} has a small normwise backward error. In other words \hat{x} minimizes $\|(A + E)\hat{x} - (b + f)\|_2$, where

$$\max \left(\frac{\|E\|_2}{\|A\|_2}, \frac{\|f\|_2}{\|b\|_2} \right) \leq p(n)\epsilon$$

where $p(n)$ is a modestly growing function of n . Let $\kappa_2(A) = \sigma_{\max}(A)/\sigma_{\min}(A)$, $\rho = \|Ax - b\|_2$, and $\sin(\theta) = \rho/\|b\|_2$. Then if $p(n)\epsilon$ is small enough, the error $\hat{x} - x$ is bounded by

$$\frac{\|x - \hat{x}\|_2}{\|x\|_2} \lesssim p(n)\epsilon \left\{ \frac{2\kappa_2(A)}{\cos(\theta)} + \tan(\theta)\kappa_2^2(A) \right\}$$

$\kappa_2(A) = \sigma_{\max}(A)/\sigma_{\min}(A)$ may be computed from the singular values of A returned by xGELSS or xGESVD (in array S, sorted from largest to smallest). $\|b\|_2$ and $\rho = \|A\hat{x} - b\|_2$ (and then $\sin(\theta) = \rho/\|b\|_2$, $\cos(\theta)$ and $\tan(\theta)$) may be easily computed from the arguments of xGELSS.

If A is rank deficient, xGELSS and xGELSX can be used to **regularize** the problem by treating all singular values less than a user-specified threshold ($\text{RCOND} \cdot \sigma_{\max}(A)$) as exactly zero. The number of singular values treated as nonzero is returned in RANK. See [28] for error bounds in this case, as well as [11, 28] for the underdetermined case.

The solution of the overdetermined, full-rank problem may also be characterized as the solution of the linear system of equations

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

By solving this linear system using xyyRFS or xyySVX (see section 4.6) componentwise error bounds can also be obtained [5].

4.8 Error Bounds for the Singular Value Decomposition

The singular value decomposition (SVD) of a real m by n matrix is the factorization $A = U\Sigma V^T$ ($A = U\Sigma V^H$ in the complex case), where U and V are orthogonal (unitary) matrices and $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_{\min(m,n)})$ is diagonal, with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$. The σ_i are the **singular values** of A and the leading $\min(m, n)$ columns u_i of U and v_i of V the **left and right singular vectors**, respectively.

The usual error analysis of the SVD algorithm xGESVD in LAPACK (see subsection 2.2.3) or the routines in LINPACK and EISPACK is as follows [28, 37]:

The computed SVD $\hat{U}\hat{\Sigma}\hat{V}^T$ is nearly the exact SVD of $A + E$, i.e. $A + E = (\hat{U} + \delta\hat{U})\hat{\Sigma}(\hat{V} + \delta\hat{V})$ is the true SVD, where $\|E\|_2/\|A\|_2 \leq p(m, n)\epsilon$, $\|\delta\hat{U}\| \leq p(m, n)\epsilon$, and $\|\delta\hat{V}\| \leq p(m, n)\epsilon$. Here $p(m, n)$ is a modestly growing function of m and n . Each computed singular value $\hat{\sigma}_i$ differs from the true σ_i by at most

$$|\hat{\sigma}_i - \sigma_i| \leq p(m, n) \cdot \epsilon \cdot \sigma_1$$

Thus large singular values (those near σ_1) are computed to high relative accuracy and small ones may not be. The singular values are returned in array S.

The angular difference between the computed singular vector \hat{u}_i and the true u_i by at most about

$$\theta(\hat{u}_i, u_i) \lesssim \frac{p(m, n)\epsilon}{\text{gap}_i},$$

where $\text{gap}_i = \min_{j \neq i} |\sigma_i - \sigma_j|$ is the **absolute gap** between σ_i and the nearest other singular value. Thus, if σ_i is close to other singular values, its corresponding singular

vector u_i may be inaccurate. The same bound applies to \hat{v}_i and v_i . The gaps may be easily computed from the computed singular values in array S.

Let \hat{S} be the space spanned by a collection of computed singular vectors $\{\hat{u}_i, i \in \mathcal{I}\}$, where \mathcal{I} is a subset of the integers from 1 to n . Let S be the corresponding true space. Then

$$\theta(\hat{S}, S) \lesssim \frac{p(m, n)\epsilon}{\text{gap}_{\mathcal{I}}}$$

where

$$\text{gap}_{\mathcal{I}} = \min_{\substack{i \in \mathcal{I} \\ j \notin \mathcal{I}}} |\sigma_i - \sigma_j|$$

is the absolute gap between the singular values in \mathcal{I} and the nearest other singular value. Thus, a cluster of close singular values which is far away from any other singular value may have a well determined space \hat{S} even if its individual singular vectors are ill-conditioned. The same bound applies to $\{\hat{v}_i, i \in \mathcal{I}\}$.

In the special case of bidiagonal matrices, the singular values and singular vectors may be computed much more accurately. A bidiagonal matrix B has nonzero entries only on the main diagonal and the diagonal immediately above it (or immediately below it). xGESVD computes the SVD of a general matrix by first reducing it to bidiagonal form B , and then calling xBDSQR (subsection 2.3.5) to compute the SVD of B . Reduction of a dense matrix to bidiagonal form B can introduce additional errors, so the following bounds for the bidiagonal case do not apply to the dense case⁸.

Each computed singular value of a bidiagonal matrix is accurate to nearly full relative accuracy, no matter how tiny it is:

$$|\hat{\sigma}_i - \sigma_i| \leq p(m, n) \cdot \epsilon \cdot \sigma_i$$

The computed singular vector \hat{u}_i has an angular error at most about

$$\theta(\hat{u}_i, u_i) \lesssim \frac{p(m, n)\epsilon}{\text{relgap}_i}$$

where $\text{relgap}_i = \min_{j \neq i} |\sigma_i - \sigma_j| / (\sigma_i + \sigma_j)$ is the **relative gap** between σ_i and the nearest other singular value. The same bound applies to \hat{v}_i and v_i . Since the relative gap may be much larger than the absolute gap, this error bound may be much smaller than the previous one. The relative gaps may be easily computed from the singular values in array S.

In the very special case of 2 by 2 bidiagonal matrices, xBDSQR calls auxiliary routine xLASV2 to compute the SVD. xLASV2 will actually compute nearly correctly rounded singular vectors independent of the relative gap, but this requires accurate computer arithmetic: if leading digits cancel during floating point subtraction, the resulting difference must be exact. On machines without guard digits one has the slightly weaker result that the algorithm is componentwise relative backward stable.

⁸Recent work has extended some of these results to dense matrices [14]. This work will appear in a later version of LAPACK.

4.9 Error Bounds for the Symmetric Eigenproblem

The eigendecomposition of an n by n real symmetric matrix is the factorization $A = Z\Lambda Z^T$ ($A = Z\Lambda Z^H$ in the complex Hermitian case), where Z is orthogonal (unitary) and $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ is real and diagonal. The λ_i are the **eigenvalues** of A and the columns z_i of Z are the **eigenvectors**. This is also often written $Az_i = \lambda_i z_i$.

The usual error analysis of the symmetric eigenproblem (using any LAPACK routine in subsection 2.2.3 such as drivers xSYEV and xSYEVX, or any EISPACK routine) is as follows [37]:

The computed eigendecomposition $\hat{Z}\hat{\Lambda}\hat{Z}^T$ is nearly the exact eigendecomposition of $A + E$, i.e., $A + E = (\hat{Z} + \delta\hat{Z})\hat{\Lambda}(\hat{Z} + \delta\hat{Z})^T$ is the true eigendecomposition, where $\|E\|_2/\|A\|_2 \leq p(n)\epsilon$ and $\|\delta\hat{Z}\|_2 \leq p(n)\epsilon$. Here $p(n)$ is a modestly growing function of n . Each computed eigenvalue $\hat{\lambda}_i$ differs from the true λ_i by at most

$$|\hat{\lambda}_i - \lambda_i| \leq p(n) \cdot \epsilon \cdot \|A\|_2$$

Thus large eigenvalues (those near $\max_i |\lambda_i| = \|A\|_2$) are computed to high relative accuracy and small ones may not be. The eigenvalues are returned in array W .

The angular difference between the computed unit singular vector \hat{z}_i and the true z_i by at most about

$$\theta(\hat{z}_i, z_i) \lesssim \frac{p(n)\epsilon}{\text{gap}_i}$$

if $p(n)\epsilon$ is small enough, where $\text{gap}_i = \min_{j \neq i} |\lambda_i - \lambda_j|$ is the **absolute gap** between λ_i and the nearest other eigenvalue. Thus, if λ_i is close to other eigenvalues, its corresponding eigenvector z_i may be inaccurate. The gaps may be easily computed from the computed eigenvalues in array W .

Let \hat{S} be the space spanned by a collection of eigenvectors $\{\hat{z}_i, i \in \mathcal{I}\}$, where \mathcal{I} is a subset of the integers from 1 to n . Let S be the corresponding true space. Then

$$\theta(\hat{S}, S) \lesssim \frac{p(n)\epsilon}{\text{gap}_{\mathcal{I}}}$$

where

$$\text{gap}_{\mathcal{I}} = \min_{\substack{i \in \mathcal{I} \\ j \notin \mathcal{I}}} |\lambda_i - \lambda_j|$$

is the absolute gap between the eigenvalues in \mathcal{I} and the nearest other eigenvalue. Thus, a cluster of close eigenvalues which is far away from any other eigenvalue may have a well determined space \hat{S} even if its individual eigenvectors are ill-conditioned.

In the special case of a real symmetric tridiagonal matrix T , the eigenvalues and eigenvectors can be computed much more accurately. xSYEV (and the other symmetric eigenproblem drivers) computes the eigenvalues and eigenvectors of a dense symmetric matrix by first reducing it to tridiagonal form T , and then finding the eigenvalues and eigenvectors of T . Reduction of a dense

matrix to tridiagonal form T can introduce additional errors, so the following bounds for the tridiagonal case do not apply to the dense case⁹.

The eigenvalues of T may be computed with small componentwise relative backward error ($O(\epsilon)$) by using subroutine xSTEBZ (subsection 2.3.3) or driver xSTEVX (subsection 2.2.3). If T is also positive definite, they may also be computed at least as accurately by xPTEQR (subsection 2.3.3). To compute error bounds for the computed eigenvalues $\hat{\lambda}_i$ we must make some assumptions about T . The bounds discussed here are from [8]. Suppose T is positive definite, and write $T = DAD$ where $D = \text{diag}(t_{11}^{1/2}, \dots, t_{nn}^{1/2})$ and $a_{ii} = 1$. Then the computed eigenvalues $\hat{\lambda}_i$ can differ from the true eigenvalues λ_i by

$$|\hat{\lambda}_i - \lambda_i| \leq p(n) \cdot \epsilon \cdot \kappa_2(A) \cdot \lambda_i$$

where $p(n)$ is a modestly growing function of n . Thus if $\kappa_2(A)$ is moderate, each eigenvalue will be computed to high relative accuracy, no matter how tiny it is. The eigenvectors \hat{z}_i computed by xPTEQR can differ from the true eigenvectors z_i by at most about

$$\theta(\hat{z}_i, z_i) \lesssim \frac{p(n) \cdot \epsilon \cdot \kappa_2(A)}{\text{relgap}_i}$$

if $p(n)\epsilon$ is small enough, where $\text{relgap}_i = \min_{j \neq i} |\lambda_i - \lambda_j| / (\lambda_i + \lambda_j)$ is the **relative gap** between λ_i and the nearest other eigenvalue. Since the relative gap may be much larger than the absolute gap, this error bound may be much smaller than the previous one.

$\kappa_2(A)$ could be computed by applying xPTSVX (subsection 2.2.1) or xPTCON (subsection 2.3.1) to A . The relative gaps are easily computed from the eigenvalues.

For further results, including error bounds appropriate to indefinite matrices, see [8].

4.10 Error Bounds for the Nonsymmetric Eigenproblem

4.10.1 Summary

The nonsymmetric eigenvalue problem is more complicated than the symmetric eigenvalue problem. In this subsection, as in previous sections, we will just summarize the bounds; in later subsections we provide some further details.

Bounds for individual eigenvalues and eigenvectors are provided by driver xGEEVX (subsection 2.2.3) or computational routine xTRSNA (subsection 2.3.4). Bounds for clusters of eigenvalues and their associated invariant subspace are provided by driver xGEESX (subsection 2.2.3) or computational routine xTRSEN (subsection 2.3.4). Further details can be found in [7].

We let $\hat{\lambda}_i$ be the i -th computed eigenvalue and λ_i the i -th true eigenvalue. Let \hat{v}_i be the corresponding computed right eigenvector, and v_i the true right eigenvector (so $Av_i = \lambda_i v_i$). If \mathcal{I} is a subset of the integers from 1 to n , we let $\lambda_{\mathcal{I}}$ denote the average of the selected eigenvalues:

⁹Recent work has extended some of these results to dense symmetric positive definite matrices [14]. This work will appear in a later version of LAPACK.

$\lambda_I = (\sum_{i \in I} \lambda_i) / (\sum_{i \in I} 1)$, and similarly for $\hat{\lambda}_I$. We also let \mathcal{S}_I denote the subspace spanned by $\{v_i, i \in I\}$; it is called a right invariant subspace because if v is any vector in \mathcal{S}_I then Av is also in \mathcal{S}_I . $\hat{\mathcal{S}}_I$ is the corresponding computed subspace.

The algorithms for the nonsymmetric eigenproblem are backward stable: they compute the exact eigenvalues, eigenvectors and invariant subspaces of slightly perturbed matrices $A + E$, where $\|E\| \leq p(n)\epsilon$. Some of the bounds are stated in terms of $\|E\|_2$ and others in terms of $\|E\|_F$; one may use $p(n)\epsilon$ for either quantity.

xGEEVX (or xTRSNA) returns two quantities for each $\hat{\lambda}_i, \hat{v}_i$ pair: s_i and sep_i . xGEESX (or xTRSEN) returns two quantities for a selected subset I of eigenvalues: s_I and sep_I . The error bounds in the Table 4.2 are true for sufficiently small $\|E\|$, which is why they are called asymptotic:

Table 4.2: Asymptotic error bounds for the Nonsymmetric Eigenproblem

Simple eigenvalue	$ \hat{\lambda}_i - \lambda_i \lesssim \ E\ _2 / s_i$
Eigenvalue cluster	$ \lambda_I - \hat{\lambda}_I \lesssim \ E\ _2 / s_I$
Eigenvector	$\theta(\hat{v}_i, v_i) \lesssim \ E\ _F / \text{sep}_i$
Invariant subspace	$\theta(\hat{\mathcal{S}}_I, \mathcal{S}_I) \lesssim \ E\ _F / \text{sep}_I$

If the problem is ill-conditioned, the asymptotic bounds may only hold for extremely small $\|E\|$. Therefore, we also provide global bounds which are guaranteed to hold for all $\|E\|_F < s \cdot \text{sep}/4$:

Table 4.3: Global error bounds for the Nonsymmetric Eigenproblem

Simple eigenvalue	$ \hat{\lambda}_i - \lambda_i \leq n\ E\ _2 / s_i$	Holds for all E
Eigenvalue cluster	$ \lambda_I - \hat{\lambda}_I \leq 2\ E\ _2 / s_I$	Requires $\ E\ _F < s_I \cdot \text{sep}_I / 4$
Eigenvector	$\theta(\hat{v}_i, v_i) \leq \arctan(2\ E\ _F / (\text{sep}_i - 4\ E\ _F / s_i))$	Requires $\ E\ _F < s_i \cdot \text{sep}_i / 4$
Invariant subspace	$\theta(\hat{\mathcal{S}}_I, \mathcal{S}_I) \leq \arctan(2\ E\ _F / (\text{sep}_I - 4\ E\ _F / s_I))$	Requires $\ E\ _F < s_I \cdot \text{sep}_I / 4$

Finally, the quantities s and sep tell us how we can best (block) diagonalize a matrix A by a similarity, $V^{-1}AV = \text{diag}(A_{11}, \dots, A_{bb})$, where each diagonal block A_{ii} has a selected subset of the eigenvalues of A . The goal is to choose a V with a nearly minimum condition number $\kappa_2(V)$ which performs this decomposition. This may be done as follows. Let A_{ii} be n_i by n_i . Then columns $1 + \sum_{j=1}^{i-1} n_j$ through $\sum_{j=1}^i n_j$ of V span the invariant subspace of A corresponding to the eigenvalues of A_{ii} ; these columns should be chosen to be any orthonormal basis of this space (as computed by xGEESX, for example). Let s_i be the value corresponding to the cluster of eigenvalues of A_{ii} , as computed by xGEESX or xTRSEN. Then $\kappa_2(V) \leq b / \min_i s_i$, and no other choice of V can make its condition number smaller than $1 / \min_i s_i$. Thus choosing orthonormal subblocks of V gets $\kappa_2(V)$ to within a factor b of its minimum value.

4.10.2 Balancing and Conditioning

There are two preprocessing steps one may perform on a matrix A in order to make its eigenproblem easier. The first is **permutation**, or reordering the rows and columns to make A more nearly upper triangular (closer to Schur form): $A' = PAP^H$, where P is a permutation matrix. If A' is permutable to upper triangular form (or close to it), then no floating point operations (or very few) are needed to reduce it to Schur form. The second is **scaling** by a diagonal matrix D to make the rows and columns of A' more nearly equal in norm: $A'' = DA'D^{-1}$. Scaling can make the eigenvalues larger with respect to the matrix norm, and so possibly reduce the inaccuracy contributed by roundoff [44, Chap 2/11]. We refer to these two operations as **balancing**.

Balancing is performed by driver xGEEVX, which calls computational routine xGEBAL. The user may tell xGEEVX to optionally permute, scale, do both, or do neither; this is specified by input parameter BALANC. Permuting has no effect on the condition numbers or their interpretation as described in previous subsections. Scaling, however, does change their interpretation, as we now describe.

The output parameters of xGEEVX – SCALE (real array of length N), ILO (integer), IHI (integer) and ABNRM (real) – describe the result of balancing a matrix A into A'' , where N is the dimension of A . The matrix A'' is block upper triangular, with at most three blocks: from 1 to ILO–1, from ILO to IHI, and from IHI+1 to N . The first and last blocks are upper triangular, and so already in Schur form. These are not scaled; only the block from ILO to IHI is scaled. Details of the scaling and permutation are described in SCALE (see the specification of xGEEVX or xGEBAL for details). The one norm of A'' is returned in ABNRM.

The condition numbers described in earlier subsections are computed for the balanced matrix A'' , and so some interpretation is needed to apply them to the eigenvalues and eigenvectors of the original matrix A . To use the bounds for eigenvalues in Tables 4.2 and 4.3, we must replace ϵ_2 and ϵ_F by $O(\epsilon)\|A''\| = O(\epsilon) \cdot \text{ABNRM}$. To use the bounds for eigenvectors, we also need to take into account that bounds on rotations of eigenvectors are for the eigenvectors x'' of A'' , which are related to the eigenvectors x of A by $DPx = x''$, or $x'' = P^T D^{-1}x$. One coarse but simple way to do this is as follows: let θ'' be the bound on rotations of x'' from the Perturbation Table, and let θ be the desired bound on rotation of x . Let

$$\kappa(D) = \frac{\max_{\text{ILO} \leq i \leq \text{IHI}} \text{SCALE}(i)}{\min_{\text{ILO} \leq i \leq \text{IHI}} \text{SCALE}(i)}$$

be the condition number of D . Then

$$\theta \leq \arccos \left(\frac{\cos \theta''}{\kappa^2(D)} \right)$$

4.10.3 Computing s and sep

To explain s and sep , we need to introduce the **spectral projector** P [40, 35], and the **separation of two matrices** A and B , $\text{sep}(A, B)$ [40, 42].

We may assume the matrix A is in Schur form, because reducing it to this form does not change the values of s and sep . Consider a cluster of $m \geq 1$ eigenvalues, counting multiplicities. Further assume the n by n matrix A is

$$A = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \quad (4.3)$$

where the eigenvalues of the m by m matrix A_{11} are exactly those in which we are interested. In practice, if the eigenvalues on the diagonal of A are in the wrong order, routine xTREXC can be used to put the desired ones in the upper left corner as shown.

We define the **spectral projector**, or simply projector P belonging to the eigenvalues of A_{11} as

$$P = \begin{pmatrix} I_m & R \\ 0 & 0 \end{pmatrix} \quad (4.4)$$

where R satisfies the system of linear equations

$$A_{11}R - RA_{22} = A_{12} \quad (4.5)$$

Equation (4.5) is called a Sylvester equation. Given the Schur form (4.3), we solve equation (4.5) for R using the subroutine xTRSYL.

We can now define s for the eigenvalues of A_{11} :

$$s = \frac{1}{\|P\|_2} = \frac{1}{\sqrt{1 + \|R\|_2^2}} \quad (4.6)$$

In practice we do not use this expression since $\|R\|_2$ is hard to compute. Instead we use the more easily computed underestimate

$$\frac{1}{\sqrt{1 + \|R\|_F^2}} \quad (4.7)$$

which can underestimate the true value of s by no more than a factor $\sqrt{\min(m, n-m)}$. This underestimation makes our error bounds more conservative.

The **separation** $\text{sep}(A_{11}, A_{22})$ of the matrices A_{11} and A_{22} is defined as the smallest singular value of the linear map in (4.5) which takes X to $A_{11}X - XA_{22}$, i.e.

$$\text{sep}(A_{11}, A_{22}) = \min_{X \neq 0} \frac{\|A_{11}X - XA_{22}\|_F}{\|X\|_F} \quad (4.8)$$

This formulation lets us estimate $\text{sep}(A_{11}, A_{22})$ using the condition estimator xLACON [30, 32, 33], which estimates the norm of a linear operator $\|T\|_1 = \max_j \sum_i |t_{ij}|$ given the ability to compute $T^T x$ and $T^T x$ quickly for arbitrary x . In our case, multiplying an arbitrary vector by T means solving the Sylvester equation (4.5) with an arbitrary right hand side using xTRSYL, and multiplying by T^T means solving the same equation with A_{11} replaced by A_{11}^T and A_{22} replaced by A_{22}^T . Solving either equation costs at most $O(n^3)$ operations, or as few as $O(n^2)$ if $m \ll n$. Since the true value of sep is $\|T\|_2$ but we use $\|T\|_1$, our estimate of sep may differ from the true value by as much as $\sqrt{m(n-m)}$.

Another formulation which in principle permits an exact evaluation of $\text{sep}(A_{11}, A_{22})$ is

$$\text{sep}(A_{11}, A_{22}) = \sigma_{\min}(I_{n-m} \otimes A_{11} - A_{22}^T \otimes I_m) \quad (4.9)$$

where $X \otimes Y \equiv [x_{ij}Y]$ is the Kronecker product of X and Y . This method is generally impractical, however, because the matrix whose smallest singular value we need is $m(n-m)$ dimensional, which can be as large as $n^2/4$. Thus we would require as much as $O(n^4)$ extra workspace and $O(n^6)$ operations, much more than the estimation method of the last paragraph.

The expression $\text{sep}(A_{11}, A_{22})$ measures the “separation” of the spectra of A_{11} and A_{22} in the following sense. It is zero if and only if A_{11} and A_{22} have a common eigenvalue, and small if there is a small perturbation of either one that makes them have a common eigenvalue. If A_{11} and A_{22} are both symmetric matrices, then $\text{sep}(A_{11}, A_{22})$ is just the gap, or minimum distance between an eigenvalue of A_{11} and an eigenvalue of A_{22} . On the other hand, if A_{11} and A_{22} are nonsymmetric, $\text{sep}(A_{11}, A_{22})$ may be much smaller than this gap.

In the case of a symmetric matrix, $s = 1$ and sep is the absolute gap, as defined in subsection 4.9.

4.11 Error bounds for the generalized symmetric-definite eigenproblem

There are three types of problems to consider. In all cases A and B are real symmetric (or complex Hermitian) and B is positive definite.

1. $A - \lambda B$. The eigendecomposition may be written $\Lambda = Z^T A Z$ and $I = Z^T B Z$ (or $\Lambda = Z^H A Z$ and $I = Z^H B Z$ if A and B are complex). Here Λ is real and diagonal, and the columns z_i of Z are independent vectors. The diagonal entries $\lambda_i = \Lambda_{ii}$ are called **eigenvalues** and the z_i are **eigenvectors**. This may also be written $A z_i = \lambda_i B z_i$.
2. $AB - \lambda I$. The eigendecomposition may be written $AB = Z \Lambda Z^{-1}$. Here Λ is real diagonal with diagonal entries λ_i , and the columns z_i of Z are independent vectors. The λ_i are called **eigenvalues** and the z_i are **eigenvectors**. This may also be written $AB z_i = \lambda_i z_i$.
3. $BA - \lambda I$. The eigendecomposition may be written $BA = Z \Lambda Z^{-1}$. Here Λ is real diagonal with diagonal entries λ_i , and the columns z_i of Z are independent vectors. The λ_i are called **eigenvalues** and the z_i are **eigenvectors**. This may also be written $BA z_i = \lambda_i z_i$.

The error analysis of the driver routine xSYGV, or xHEGV in the complex case (see subsection 2.2.4) goes as follows. In all cases $\text{gap}_i = \min_{j \neq i} |\lambda_i - \lambda_j|$ is the **absolute gap** between λ_i and the nearest other eigenvalue.

1. $A - \lambda B$. The computed eigenvalues $\hat{\lambda}_i$ can differ from the true eigenvalues λ_i by at most about

$$|\hat{\lambda}_i - \lambda_i| \lesssim p(n) \cdot \epsilon \cdot \|B^{-1}\|_2 \cdot \|A\|_2$$

The angular difference between the computed eigenvector \hat{z}_i and the true eigenvector z_i is

$$\theta(\hat{z}_i, z_i) \lesssim \frac{p(n) \cdot \epsilon \cdot \|B^{-1}\|_2 \cdot \|A\|_2 \cdot (\kappa_2(B))^{1/2}}{\text{gap}_i}$$

2. $AB - \lambda I$ or $BA - \lambda I$. The computed eigenvalues $\hat{\lambda}_i$ can differ from the true eigenvalues λ_i by at most about

$$|\hat{\lambda}_i - \lambda_i| \lesssim p(n) \cdot \epsilon \cdot \|B\|_2 \cdot \|A\|_2$$

The angular difference between the computed eigenvector \hat{z}_i and the true eigenvector z_i is

$$\|\hat{z}_i - z_i\|_2 \lesssim \frac{q(n) \cdot \epsilon \cdot \|B\|_2 \cdot \|A\|_2 \cdot (\kappa_2(B))^{1/2}}{\text{gap}_i}$$

These error bounds are large when B is ill-conditioned ($\kappa_2(B)$ is large). It is often the case that the eigenvalues and eigenvectors are much better conditioned than indicated here. We mention two ways to get tighter bounds. The first way is effective when the diagonal entries of B differ widely in magnitude¹⁰:

1. $A - \lambda B$. Let $D = \text{diag}(B_{11}^{-1/2}, \dots, B_{nn}^{-1/2})$ be a diagonal matrix. Then replace B by DBD and A by DAD in the above bounds.
2. $AB - \lambda I$ or $BA - \lambda I$. Let $D = \text{diag}(B_{11}^{-1/2}, \dots, B_{nn}^{-1/2})$ be a diagonal matrix. Then replace B by DBD and A by $D^{-1}AD^{-1}$ in the above bounds.

The second way to get tighter bounds does not actually supply guaranteed bounds, but its estimates are often better in practice. It is not guaranteed because it assumes the algorithm is backward stable, which is not necessarily true when B is ill-conditioned. It estimates the **chordal distance** between a true eigenvalue λ_i and a computed eigenvalue $\hat{\lambda}_i$:

$$\chi(\hat{\lambda}_i, \lambda_i) = \frac{|\hat{\lambda}_i - \lambda_i|}{\sqrt{1 + \hat{\lambda}_i^2} \cdot \sqrt{1 + \lambda_i^2}}$$

To interpret this measure we write $\lambda_i = \tan \theta$ and $\hat{\lambda}_i = \tan \hat{\theta}$. Then $\chi(\hat{\lambda}_i, \lambda_i) = |\sin(\hat{\theta} - \theta)|$. Thus χ is bounded by one, and is small when both arguments are large¹¹. It applies only to the first problem, $A - \lambda B$.

Suppose a computed eigenvalue $\hat{\lambda}_i$ of $A - \lambda B$ is the exact eigenvalue of a perturbed problem $(A + E) - \lambda(B + F)$. Let x_i be the unit eigenvector ($\|x_i\|_2 = 1$) for the exact

¹⁰This is true only if the Level 3 BLAS are implemented in a conventional way, not in a fast way as described in section 4.12.

¹¹Another interpretation of chordal distance is as half the usual Euclidean distance between the projections of $\hat{\lambda}_i$ and λ_i on the Riemann sphere, i.e. half the length of the chord connecting the projections.

eigenvalue λ_i . Then if $\|E\|$ is small compared to $\|A\|$, and if $\|F\|$ is small compared to $\|B\|$, we have

$$\chi(\hat{\lambda}_i, \lambda_i) \lesssim \frac{\|E\| + \|F\|}{\sqrt{(x_i^H A x_i)^2 + (x_i^H B x_i)^2}}.$$

Thus $1/\sqrt{(x_i^H A x_i)^2 + (x_i^H B x_i)^2}$ is a condition number for eigenvalue λ_i .

Other yet more refined algorithms and error bounds are discussed in [8, 41, 43], and will be available in future releases.

4.12 Error bounds for Fast Level 3 BLAS

The Level 3 BLAS specifications [17] specify the input, output and call sequence for each routine, but allow freedom of implementation, subject to the requirement that the routines be numerically stable. Level 3 BLAS implementations can therefore be built using matrix multiplication algorithms that achieve a more favorable operation count (for suitable dimensions) than the standard multiplication technique, provided that these “fast” algorithms are numerically stable. The most well-known fast matrix multiplication technique is Strassen’s method, which can multiply two $n \times n$ matrices in fewer than $4.7n^{\log_2 7}$ operations, where $\log_2 7 \approx 2.807$.

The effect on the results in this chapter of using a fast Level 3 BLAS implementation can be explained as follows. In general, reasonably implemented fast Level 3 BLAS preserve all the bounds presented here (except those at the end of subsection 4.11), but the constant $p(n)$ may increase somewhat. Also, the iterative refinement routine xyyRFS may take more steps to converge.

This is what we mean by reasonably implemented fast Level 3 BLAS. Here, c_i denotes a constant depending on the specified matrix dimensions.

(1) If A is $m \times n$, B is $n \times p$ and \hat{C} is the computed approximation to $C = AB$, then

$$\|\hat{C} - AB\|_\infty \leq c_1(m, n, p)\epsilon\|A\|_\infty\|B\|_\infty + O(\epsilon^2).$$

(2) The computed solution \hat{X} to the triangular systems $TX = B$, where T is $m \times m$ and B is $m \times p$, satisfies

$$\|T\hat{X} - B\|_\infty \leq c_2(m, p)\epsilon\|T\|_\infty\|\hat{X}\|_\infty + O(\epsilon^2).$$

For conventional Level 3 BLAS implementations these conditions hold with $c_1(m, n, p) = n^2$ and $c_2(m, p) = m(m + 1)$. Strassen’s method satisfies these bounds for slightly larger c_1 and c_2 .

For further details, and references to fast multiplication techniques, see [12].

Chapter 5

Documentation and Software Conventions

5.1 Design and Documentation of Argument Lists

The argument lists of all LAPACK routines conform to a single set of conventions for their design and documentation.

Specifications of all LAPACK driver and computational routines are given in Appendix F. These are derived from the specifications given in the leading comments in the code, but in Appendix F the specifications for real and complex versions of each routine have been merged, in order to save space.

5.1.1 Structure of the Documentation

The documentation of each LAPACK routine includes:

- the SUBROUTINE or FUNCTION statement, followed by statements declaring the type and dimensions of the arguments
- a summary of the **Purpose** of the routine
- descriptions of each of the **Arguments** in the order of the argument list
- (optionally) **Further Details** (only in the code, not in Appendix F)
- (optionally) **Internal Parameters** (only in the code, not in Appendix F)

5.1.2 Order of Arguments

Arguments of an LAPACK routine appear in the following order:

- arguments specifying options
- problem dimensions
- array or scalar arguments defining the input data; some of them may be overwritten by results
- other array or scalar arguments returning results
- work arrays (and associated array dimensions)
- diagnostic argument INFO

5.1.3 Argument Descriptions

The style of the argument descriptions is illustrated by the following example:

N	(input) INTEGER The number of columns of the matrix A. $N \geq 0$.
A	(input/output) REAL array, dimension (LDA,N) On entry, the m-by-n matrix to be factored. On exit, the factors L and U from the factorization $A = PLU$; the unit diagonal elements of L are not stored.

The description of each argument gives:

- a classification of the argument as input, output, input/output or workspace;
- the type of the argument;
- (for an array) its dimension(s);
- a specification of the value(s) that must be supplied for the argument (if it's an input argument), or of the value(s) returned by the routine (if it's an output argument), or both (if it's an input/output argument). In the last case, the two parts of the description are introduced by the phrases "On entry" and "On exit".
- (for a scalar input argument) any constraints that the supplied values must satisfy (such as " $N \geq 0$ " in the example above).

5.1.4 Option Arguments

Arguments specifying options are usually of type CHARACTER*1. The meaning of each valid value is given, as in this example:

UPLO	(input) CHARACTER*1 = 'U': Upper triangle of A is stored; = 'L': Lower triangle of A is stored.
------	---

The corresponding lower-case characters may be supplied (with the same meaning), but any other value is illegal (see subsection 5.1.8).

A longer character string can be passed as the actual argument, making the calling program more readable, but only the first character is significant. For example:

```
CALL SPOTRS ('upper', . . . )
```

5.1.5 Problem Dimensions

It is permissible for the problem dimensions to be passed as zero, in which case the computation (or part of it) is skipped. Negative dimensions are regarded as erroneous.

5.1.6 Array Arguments

Each 2-dimensional array argument is immediately followed in the argument list by its leading dimension, whose name has the form LD<array-name>. For example:

```
A      (input/output) REAL/COMPLEX array, dimension (LDA,N)
...
LDA     (input) INTEGER
        The leading dimension of the array A.  $LDA \geq \max(1,M)$ .
```

It should be assumed, unless stated otherwise, that vectors and matrices are stored in 1- and 2-dimensional arrays in the conventional manner. That is, if an array X of dimension (N) holds a vector x , then $X(i)$ holds x_i for $i = 1, \dots, n$. If a 2-dimensional array A of dimension (LDA,N) holds an m -by- n matrix A, then $A(i,j)$ holds a_{ij} for $i = 1, \dots, m$ and $j = 1, \dots, n$ (LDA must be at least m). See Section 5.3 for more about storage of matrices.

Note that array arguments are usually declared in the software as assumed-size arrays (last dimension *), for example:

```
REAL A( LDA, * )
```

although the documentation gives the dimensions as (LDA,N). The latter form is more informative since it specifies the required minimum value of the last dimension. However an assumed-size array declaration has been used in the software, in order to overcome some limitations in the Fortran 77 standard. In particular it allows the routine to be called when the relevant dimension (N, in this case) is zero. However actual array dimensions in the calling program must be at least 1 (LDA in this example).

5.1.7 Work Arrays

Many LAPACK routines require one or more work arrays to be passed as arguments. The name of a work array is usually WORK — sometimes IWORK or RWORK to distinguish work arrays of

integer or real type.

A number of routines implementing block algorithms require workspace sufficient to hold one block of rows or columns of the matrix, for example, workspace of size n -by- nb , where nb is the block size. In such cases, the actual declared length of the work array must be passed as a separate argument LWORK, which immediately follows WORK in the argument-list.

See Section 5.2 for further explanation.

5.1.8 Error handling and the diagnostic argument INFO

All documented routines have a diagnostic argument INFO that indicates the success or failure of the computation, as follows:

- INFO = 0: successful termination
- INFO < 0: illegal value of one or more arguments - no computation performed
- INFO > 0: failure in the course of computation

All routines described in this document check that input arguments such as N or LDA or option arguments of type character have permitted values. If an illegal value of the i^{th} argument is detected, the routine sets INFO = $-i$, and then calls an error-handling routine XERBLA.

The standard version of XERBLA issues an error message and halts execution, so that no LAPACK routine would ever return to the calling program with INFO < 0. However this might occur if a non-standard version of XERBLA is used.

5.2 Determining the block size for block algorithms

LAPACK routines that implement block algorithms need to determine what block size to use. The intention behind the design of LAPACK is that the choice of block size should be hidden from users as much as possible, but at the same time easily accessible to installers of the package when tuning LAPACK for a particular machine.

LAPACK routines call an auxiliary enquiry function ILAENV, which returns the optimal block size to be used, as well as other parameters. The version of ILAENV supplied with the package contains default values that led to good behavior over a reasonable number of our test machines, but to achieve optimal performance, it may be beneficial to tune ILAENV for your particular machine environment. Ideally a distinct implementation of ILAENV is needed for each machine environment (see also Chapter 6). The optimal block size may also depend on the routine, the combination of option arguments (if any), and the problem dimensions.

If ILAENV returns a block size of 1, then the routine performs the unblocked algorithm, calling Level 2 BLAS, and makes no calls to Level 3 BLAS.

Some LAPACK routines require a work array whose size is proportional to the block size (see subsection 5.1.7). The actual length of the work array is supplied as an argument LWORK. The description of the arguments WORK and LWORK typically goes as follows:

WORK (workspace) REAL array, dimension (LWORK)
If INFO = 0, then WORK(1) returns the optimal LWORK.
LWORK (input) INTEGER
The dimension of the array WORK. $LWORK \geq \max(1, N)$. For optimal performance $LWORK \geq N \cdot NB$, where NB is the optimal blocksize returned by ILAENV.

The routine determines the block size to be used by the following steps:

1. the optimal block size is determined by calling ILAENV;
2. if the value of LWORK indicates that enough workspace has been supplied, the routine uses the optimal block size;
3. otherwise, the routine determines the largest block size that can be used with the supplied amount of workspace;
4. if this new block size does not fall below a threshold value (also returned by ILAENV), the routine uses the new value;
5. otherwise, the routine uses the unblocked algorithm.

The minimum value of LWORK that would be needed to use the optimal block size, is returned in WORK(1).

Thus, the routine uses the largest block size allowed by the amount of workspace supplied, as long as this is likely to give better performance than the unblocked algorithm. WORK(1) is not always a simple formula in terms of N and NB. The comments will specify a lower bound on LWORK for correct functioning.

If LWORK indicates that there is insufficient workspace to perform the unblocked algorithm, the value of LWORK is regarded as an illegal value, and is treated like any other illegal argument value (see subsection 5.1.8).

If in doubt about how much workspace to supply, users should supply a generous amount (assume a block size of 64, say), and then examine the value of WORK(1) on exit.

5.3 Matrix storage schemes

LAPACK allows the following different storage schemes for matrices:

- conventional storage in a 2-dimensional array;

- packed storage for symmetric, Hermitian or triangular matrices;
- band storage for band matrices;
- the use of two or three 1-dimensional arrays to store tridiagonal or bidiagonal matrices.

These storage schemes are compatible with those used in LINPACK and the BLAS, but EISPACK uses incompatible schemes for band and tridiagonal matrices.

In the examples below, * indicates an array element that need not be set and is not referenced by LAPACK routines. Elements that “need not be set” are never read, written to, or otherwise accessed by the LAPACK routines. The examples illustrate only the relevant part of the arrays; array arguments may of course have additional rows or columns, according to the usual rules for passing array arguments in Fortran 77.

5.3.1 Conventional Storage

The default scheme for storing matrices is the obvious one described in subsection 5.1.6: a matrix A is stored in a 2-dimensional array A , with matrix element a_{ij} stored in array element $A(i, j)$.

If a matrix is **triangular** (upper or lower, as specified by the argument UPLO), only the elements of the relevant triangle are accessed. The remaining elements of the array need not be set. Such elements are indicated by * in the examples below. For example, when $n = 4$:

UPLO	Triangular matrix A	Storage in array A
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
'L'	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

Similarly, if the matrix is upper Hessenberg, elements below the first subdiagonal need not be set.

Routines that handle **symmetric** or **Hermitian** matrices allow for either the upper or lower triangle of the matrix (as specified by UPLO) to be stored in the corresponding elements of the array; the remaining elements of the array need not be set. For example, when $n = 4$:

UPLO	Hermitian matrix A	Storage in array A
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} \\ \bar{a}_{14} & \bar{a}_{24} & \bar{a}_{34} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
'L'	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & \bar{a}_{41} \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

5.3.2 Packed Storage

Symmetric, Hermitian or triangular matrices may be stored more compactly, if the relevant triangle (again as specified by UPLO) is packed **by columns** in a 1-dimensional array. In LAPACK, arrays that hold matrices in packed storage, have names ending in 'P'. So:

- if UPLO = 'U', a_{ij} is stored in $AP(i + j(j - 1)/2)$ for $i \leq j$;
- if UPLO = 'L', a_{ij} is stored in $AP(i + (2n - j)(j - 1)/2)$ for $j \leq i$.

For example:

UPLO	Triangular matrix A	Packed storage in array AP
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11} \quad \underbrace{a_{12} \ a_{22}} \quad \underbrace{a_{13} \ a_{23} \ a_{33}} \quad \underbrace{a_{14} \ a_{24} \ a_{34} \ a_{44}}$
'L'	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11} \ a_{21} \ a_{31} \ a_{41}} \quad \underbrace{a_{22} \ a_{32} \ a_{42}} \quad \underbrace{a_{33} \ a_{43}} \quad a_{44}$

Note that for real or complex symmetric matrices, packing the upper triangle by columns is equivalent to packing the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the upper triangle by rows. For complex Hermitian matrices, packing the upper triangle by columns is equivalent to packing the conjugate of the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the conjugate of the upper triangle by rows.

5.3.3 Band Storage

A band matrix with kl subdiagonals and ku superdiagonals may be stored compactly in a 2-dimensional array with $kl + ku + 1$ rows and n columns. Columns of the matrix are stored in corresponding columns of the array, and diagonals of the matrix are stored in rows of the array.

This storage scheme should be used in practice only if $kl, ku \ll n$, although LAPACK routines work correctly for all values of kl and ku . In LAPACK, arrays that hold matrices in band storage have names ending in 'B'.

To be precise, a_{ij} is stored in $AB(ku + 1 + i - j, j)$ for $\max(1, j - ku) \leq i \leq \min(n, j + kl)$. For example, when $n = 5$, $kl = 2$ and $ku = 1$:

Band matrix A	Band storage in array AB
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

The elements marked * in the upper left and lower right corners of the array AB need not be set, and are not referenced by LAPACK routines.

Note: when a band matrix is supplied for LU factorization, space must be allowed to store an additional kl superdiagonals, generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $kl + ku$ superdiagonals.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular.

For symmetric or Hermitian band matrices with kd subdiagonals or superdiagonals, only the upper or lower triangle (as specified by UPLO) need be stored:

- if UPLO = 'U', a_{ij} is stored in $AB(kd + 1 + i - j, j)$ for $\max(1, j - kd) \leq i \leq j$;
- if UPLO = 'L', a_{ij} is stored in $AB(1 + i - j, j)$ for $j \leq i \leq \min(n, j + kd)$.

For example, when $n = 5$ and $kd = 2$:

UPLO	Hermitian band matrix A	Band storage in array AB
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \bar{a}_{24} & \bar{a}_{34} & a_{44} & a_{45} \\ & & \bar{a}_{35} & \bar{a}_{45} & a_{55} \end{pmatrix}$	$\begin{matrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{matrix}$
'L'	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & & \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} & a_{53} \\ & a_{42} & a_{43} & a_{44} & a_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

EISPACK routines use a different storage scheme for band matrices, in which rows of the matrix are stored in corresponding rows of the array, and diagonals of the matrix are stored in columns of the array.

5.3.4 Tridiagonal and Bidiagonal Matrices

An unsymmetric tridiagonal matrix of order n is stored in three 1-dimensional arrays, one of length n containing the diagonal elements, and two of length $n - 1$ containing the subdiagonal and superdiagonal elements in elements $1:n - 1$.

A symmetric tridiagonal or bidiagonal matrix is stored in two 1-dimensional arrays, one of length n containing the diagonal elements, and one of length $n - 1$ containing the off-diagonal elements. (EISPACK routines store the off-diagonal elements in elements $2:n$ of a vector of length n .)

5.3.5 Unit Triangular Matrices

Some LAPACK routines have an option to handle unit triangular matrices (that is, triangular matrices with diagonal elements = 1). This option is specified by an argument **DIAG**. If **DIAG** = 'U' (Unit triangular), the diagonal elements of the matrix need not be stored, and the corresponding array elements are not referenced by the LAPACK routines. The storage scheme for the rest of the matrix (whether conventional, packed or band) remains unchanged, as described in subsections 5.3.1, 5.3.2 and 5.3.3.

5.3.6 Real Diagonal Elements of Complex Matrices

Complex Hermitian matrices have diagonal matrices that are by definition purely real. In addition, some complex triangular matrices computed by LAPACK routines are defined by the algorithm to have real diagonal elements — in Cholesky or QR factorization, for example.

If such matrices are supplied as input to LAPACK routines, the imaginary parts of the diagonal elements are not referenced, but are assumed to be zero. If such matrices are returned as output by LAPACK routines, the computed imaginary parts are explicitly set to zero.

5.4 Representation of orthogonal or unitary matrices

A real orthogonal or complex unitary matrix (usually denoted Q) is often represented in LAPACK as a product of **elementary reflectors** — also referred to as **elementary Householder matrices** (usually denoted H_i). For example,

$$Q = H_1 H_2 \dots H_k.$$

Most users need not be aware of the details, because LAPACK routines are provided to work with this representation:

- routines whose names begin **SORG-** (real) or **CUNG-** (complex) can generate all or part of Q explicitly;
- routines whose name begin **SORM-** (real) or **CUNM-** (complex) can multiply a given matrix by Q or Q^H without forming Q explicitly.

The following further details may occasionally be useful.

An elementary reflector (or elementary Householder matrix) H of order n is a unitary matrix of the form

$$H = I - \tau v v^H \quad (5.1)$$

where τ is a scalar, and v is an n -vector, with $|\tau|^2 \|v\|_2^2 = 2\text{Re}(\tau)$; v is often referred to as the **Householder vector**. Often v has several leading or trailing zero elements, but for the purpose of this discussion assume that H has no such special structure.

There is some redundancy in the representation (5.1), which can be removed in various ways. The representation used in LAPACK (which differs from those used in LINPACK or EISPACK) sets $v_1 = 1$; hence v_1 need not be stored. In real arithmetic, $1 \leq \tau \leq 2$, except that $\tau = 0$ implies $H = I$.

In complex arithmetic, τ may be complex, and satisfies $1 \leq \text{Re}(\tau) \leq 2$ and $|\tau - 1| \leq 1$. Thus a complex H is not Hermitian (as it is in other representations), but it is unitary, which is the important property. The advantage of allowing τ to be complex is that, given an arbitrary complex vector x , Hx can be computed so that

$$Hx = \beta(1, 0, \dots, 0)^*$$

with **real** β . This is useful, for example, when reducing a complex Hermitian matrix to real symmetric tridiagonal form, or a complex rectangular matrix to real bidiagonal form.

Chapter 6

Installing LAPACK routines

6.1 Points to note

For anyone who obtains the complete LAPACK package from NAG (see Chapter 1), a comprehensive installation guide will be provided. We recommend installation of the complete package as the most convenient and reliable way to make LAPACK available.

People who obtain copies of a few LAPACK routines from *netlib*, need to be aware of the following points:

1. Double precision complex routines (names beginning Z-) use a COMPLEX*16 data type. This is an extension to the Fortran 77 standard, but is provided by many Fortran compilers on machines where double precision computation is usual. The following related extensions are also used:
 - the intrinsic function DCONJG, with argument and result of type COMPLEX*16;
 - the intrinsic functions DBLE and DIMAG, with COMPLEX*16 argument and DOUBLE PRECISION result, returning the real and imaginary parts respectively;
 - the intrinsic function DCMPLX, with DOUBLE PRECISION argument(s) and COMPLEX*16 result;
 - COMPLEX*16 constants, formed from a pair of double precision constants in parentheses.

Some compilers provide DOUBLE COMPLEX as an alternative to COMPLEX*16, and an intrinsic function DREAL (rather than DBLE) may be used to return the real part of a COMPLEX*16 argument.

2. Machine-dependent parameters such as the block size, minimum block size, crossover point when an blocked routine should be used, etc. are set by calls to an inquiry function ILAENV which may be set with different values on each machine. See section 6.2 for more about ILAENV.

3. SLAMCH/DLAMCH determines the properties of the floating point arithmetic at runtime. It tries to determine the roundoff level, underflow threshold, overflow threshold, radix and related parameters. It works satisfactorily on all commercially important machines of which we are aware, but will necessarily be updated from time to time as new machines and compilers are produced.

6.2 Installing ILAENV

Machine-dependent parameters such as the block size are set by calls to an inquiry function which may be set with different values on each machine. The declaration of the environment inquiry function is

```
INTEGER FUNCTION ILAENV( ISPEC, NAME, OPTS, N1, N2, N3, N4 )
```

where ISPEC, N1, N2, N3, and N4 are integer variables and NAME and OPTS are CHARACTER(*). NAME specifies the subroutine name, OPTS is a character string of options to the subroutine, and N1-N4 are the problem dimensions. ISPEC specifies the parameter to be returned; the following values are currently used in LAPACK:

- ISPEC = 1: NB, optimal blocksize
- = 2: NBMIN, minimum block size for the block routine to be used
- = 3: NX, crossover point (in a block routine, for $N < NX$, an unblocked routine should be used)
- = 4: NS, number of shifts
- = 6: NXSVD, crossover point for the SVD
- = 8: MAXB, crossover point for block multishift QR

The three block size parameters, NB, NBMIN, and NX, are used in many different subroutines (see Table 6.1). NS and MAXB are used in the block multishift QR algorithm, xHSEQR. NXSVD is just a constant multiple of N : $1.6N$; it is used in the driver routines xGELSS and xGESVD.

The LAPACK timing programs were designed to collect data for all the routines in Table 6.1. The range of problem sizes needed to determine the optimal block size or crossover point is machine-dependent, but the input files provided with the LAPACK test and timing package can be used as a starting point. For subroutines that require a crossover point, it is best to start by finding the best blocksize with the crossover point set to 0, and then to locate the point at which the performance of the unblocked algorithm is beaten by the block algorithm. The best crossover point will be somewhat smaller than the point where the curves for the unblocked and blocked methods cross.

For example, for SGEQRF on a single processor of a CRAY-2, $NB = 32$ was observed to be a good block size, and the performance of the block algorithm with this block size surpasses the unblocked algorithm for square matrices between $N = 176$ and $N = 192$. Experiments with crossover points from 64 to 192 found that $NX = 128$ was a good choice, although the results for NX from $3 \cdot NB$

REAL	COMPLEX	NB	NBMIN	NX
SGETRF	CGETRF	•		
SGBTRF	CGBTRF	•		
SPOTRF	CPOTRF	•		
SPBTRF	CPBTRF	•		
SSYTRF	CHETRF	•	•	
	CSYTRF	•	•	
SGETRI	CGETRI	•	•	
SPOTRI	CPOTRI	•		
STRTRI	CTRTRI	•		
SGEQRF†	CGEQRF†	•	•	•
SORGQR†	CUNGQR†	•	•	•
SORMQR†	CUNMQR†	•	•	
SGEHRD	CGEHRD	•	•	•
SSYTRD	CHETRD	•	•	•
SGBRD	CGBRD	•	•	•
SSTEBZ		•		

†– also RQ, QL, and LQ

Table 6.1: Use of the block parameters NB, NBMIN, and NX in LAPACK

to $5 \cdot \text{NB}$ are broadly similar. This means that matrices with $N \leq 128$ should use the unblocked algorithm, and for $N > 128$ block updates should be used until the remaining submatrix has order less than 128. The performance of the unblocked ($\text{NB} = 1$) and blocked ($\text{NB} = 32$) algorithms for SGEQRF and for the blocked algorithm with a crossover point of 128 are compared in Figure 6.1.

By experimenting with small values of the block size, it should be straightforward to choose NBMIN, the smallest block size that gives a performance improvement over the unblocked algorithm. Note that on some machines, the optimal block size may be 1 (the unblocked algorithm gives the best performance); in this case, the choice of NBMIN is arbitrary.

Complicating the determination of optimal parameters is the fact that the orthogonal factorization routines and SGBRD accept non-square matrices as input. The LAPACK timing program allows M and N to be varied independently. We have found the optimal blocksize to be generally insensitive to the shape of the matrix, but the crossover point is more dependent on the matrix shape. For example, if $M \gg N$ in the QR factorization, block updates may always be faster than unblocked updates on the remaining submatrix. For example, one might set $\text{NX} = \text{NB}$ if $M \geq 2N$.

Parameter values for the number of shifts, etc. used to tune the block multishift QR algorithm can be varied from the input files to the eigenvalue timing program. Interested users should consult [2] for a description of the timing program input files.

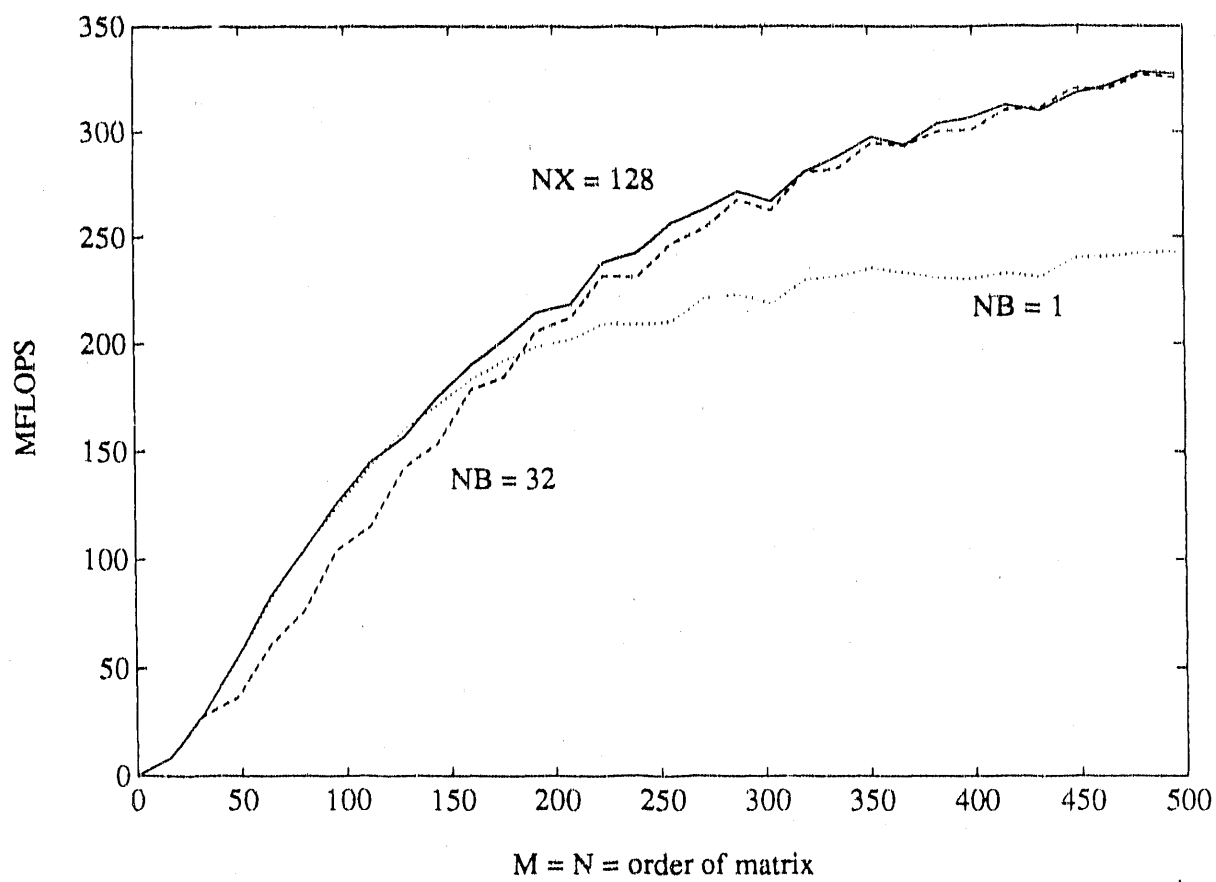


Figure 6.1: QR factorization on CRAY-2 (1 processor)

Chapter 7

Troubleshooting

7.1 Failures or wrong results

Failures and wrong results can often be due to incorrect argument types or count in a subroutine call, particularly when users are not familiar with Fortran. The following points give some common mistakes, which are worth considering before assuming that the LAPACK routine is failing.

Array dimensions Check that array arguments are correctly dimensioned in the (sub)program from which LAPACK is called. In particular, all two-dimensional array arguments in LAPACK have an associated leading dimension argument, which must be set to the value of the first dimension of the array in the calling (sub)program. For example, SPOTRF has the calling sequence:

```
SUBROUTINE SPOTRF( UPLO, N, A, LDA, INFO )  
CHARACTER          UPLO  
INTEGER            INFO, LDA, N  
REAL               A( LDA, * )
```

and so a calling program might have the structure:

```
PROGRAM             MAIN  
PARAMETER           ( NMAX = 100, LDA = NMAX )  
REAL                A( LDA, NMAX )  
:  
N = 50  
:  
CALL SPOTRF( 'Upper', N, A, LDA, INFO )
```

Precision and type Check that arguments have the correct type declarations for the LAPACK routine being called. In particular, the precision of real and complex arguments should

match the precision being used: REAL and COMPLEX for Sxxxxx and Cxxxxx routines, and DOUBLE PRECISION and COMPLEX*16 for Dxxxxx and Zxxxxx routines.

Argument matching The order and the number of arguments should match the calling sequence. Unfortunately most compilers accept, without complaint, an incorrect calling sequence.

Workspace A number of LAPACK routines require one or more workspace arguments. Check that sufficient workspace is being supplied to the LAPACK routine. Some LAPACK routines that require workspace have an associated length argument associated with the workspace argument, (... , WORK, LWORK, ...) for example, and this should match the declared length of the workspace.

INFO Check the parameter INFO on exit from an LAPACK routine. If an LAPACK routine detects an error or failure, then a non-zero value of INFO is returned. For example, if A is not positive-definite, then the above routine SPOTRF cannot compute the Cholesky factorization and returns a positive value of INFO.

Failures during installation In the course of running our LAPACK testcode on various machines and compilers, a number of compiler and mathematical library bugs were discovered and reported to the developers of these products. While these bugs are a rare cause of failure, they do represent a possible reason for our testcode to indicate the presence of inaccuracies during testing.

In addition to the above points, the LAPACK routine to determine machine parameters, SLAMCH in single precision and DLAMCH in double precision, may have been incorrectly installed on your machine. A simple test routine is supplied with LAPACK, so if there is any doubt this test should be run. See Chapter 6 for further information.

7.2 Poor performance

To avoid poor performance of an LAPACK routine, please note the following recommendations:

BLAS Whenever possible, one should link to efficient versions of the BLAS for the machine being used. A number of manufacturers supply highly efficient versions, and to gain the best possible performance from LAPACK those versions should be used. A portable set of Fortran 77 BLAS are supplied with LAPACK, so that it is always possible to run LAPACK, but no attempt has been made to tune these for specific machines.

ILAENV The LAPACK routine ILAENV returns machine dependent parameters, such as the block size, that are important for the efficiency of many LAPACK routines. Correct installation of this routine is essential. See Chapter 6 for further information on installing ILAENV.

Workspace A number of the LAPACK routines require additional workspace, which is dependent upon the block size, to work efficiently. The routines will work correctly with less than the optimum workspace, but the efficiency may be compromised. For example, an unblocked

algorithm may be used in place of the blocked algorithm. Routines that require this additional workspace return the value of the optimum workspace in the first element of the workspace array and hence, if necessary, the workspace can be increased so that subsequent runs can be performed with the optimum workspace.

xLAMCH The *first* call to xLAMCH in a program may be quite expensive, as it attempts to determine dynamically the parameters of the machine arithmetic. These values are saved within the routine so that the cost of subsequent calls is trivial. A good practice is to include a call to xLAMCH in the timing program, before any calls to LAPACK routines being timed, for example in single precision:

```
XXXXXX = SLAMCH( 'P' )
```

or in double precision:

```
XXXXXX = DLAMCH( 'P' )
```

Installers may wish to save the values computed by SLAMCH/DLAMCH for a specific machine and hard code them in DATA statements, provided that no accuracy is lost in the translation.

Bibliography

- [1] E. ANDERSON, Z. BAI, C. BISCHOF, J. W. DEMMEL, J. J. DONGARRA, J. DUCROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND S. D., *LAPACK: A portable linear algebra library for high-performance computers*, Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #20).
- [2] E. ANDERSON, J. J. DONGARRA, AND S. OSTROUCHOV, *Implementation guide for LAPACK*, Computer Science Dept. Technical Report CS-91-138, University of Tennessee, Knoxville, 1991. (LAPACK Working Note #35).
- [3] ANSI/IEEE, *IEEE Standard for Binary Floating Point Arithmetic*, New York, Std 754-1985 ed., 1985.
- [4] M. ARIOLI, J. W. DEMMEL, AND I. S. DUFF, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 165-190.
- [5] M. ARIOLI, I. S. DUFF, AND P. P. M. DE RIJK, *On the augmented system approach to sparse least-squares problems*, Num. Math., 55 (1989), pp. 667-684.
- [6] Z. BAI AND J. W. DEMMEL, *On a block implementation of Hessenberg multishift QR iteration*, International Journal of High Speed Computing, 1 (1989), pp. 97-112. (also LAPACK Working Note #8; submitted to ACM Trans. Math. Soft.).
- [7] Z. BAI, J. W. DEMMEL, AND A. MCKENNEY, *On the conditioning of the nonsymmetric eigenproblem: Theory and software*, Computer Science Dept. Technical Report 469, Courant Institute, New York, NY, October 1989. (LAPACK Working Note #13).
- [8] J. BARLOW AND J. DEMMEL, *Computing accurate eigensystems of scaled diagonally dominant matrices*, SIAM J. Num. Anal., 27 (1990), pp. 762-791.
- [9] W. R. COWELL, S. J. HAGUE, AND R. M. J. ILES, *Toolpack/1 Introductory Guide*, Numerical Algorithms Group Ltd, 1985. publication reference NP1007.
- [10] P. DEIFT, J. W. DEMMEL, L.-C. LI, AND C. TOMEI, *The bidiagonal singular values decomposition and Hamiltonian mechanics*, SIAM J. Num. Anal., 28 (1991), pp. 1463-1516. (LAPACK Working Note #11).
- [11] J. W. DEMMEL AND N. J. HIGHAM, *Improved error bounds for underdetermined systems solvers*, Computer Science Dept. Technical Report CS-90-113, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #23; to appear in SIAM J. Mat. Anal. Appl.).

- [12] —, *Stability of block algorithms with fast level 3 BLAS*, Computer Science Dept. Technical Report CS-90-110, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #22; to appear in ACM Trans. Math. Soft.).
- [13] J. W. DEMMEL AND W. KAHAN, *Accurate singular values of bidiagonal matrices*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 873–912.
- [14] J. W. DEMMEL AND K. VESELIĆ, *Jacobi's method is more accurate than QR*, Computer Science Dept. Technical Report 468, Courant Institute, New York, NY, October 1989. (also LAPACK Working Note #15), to appear in SIAM J. Mat. Anal. Appl.
- [15] J. J. DONGARRA, J. R. BUNCH, C. B. MOLER, AND G. W. STEWART, *LINPACK User's Guide*, SIAM, Philadelphia, PA, 1979.
- [16] J. J. DONGARRA, J. DU CROZ, I. S. DUFF, AND S. HAMMARLING, *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 18–28.
- [17] —, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.
- [18] J. J. DONGARRA, J. DU CROZ, S. HAMMARLING, AND R. J. HANSON, *Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subroutines*, ACM Trans. Math. Soft., 14 (1988), pp. 18–32.
- [19] —, *An extended set of fortran basic linear algebra subroutines*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.
- [20] J. J. DONGARRA, I. S. DUFF, D. C. SORESENSEN, AND H. A. VAN DER VORST, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, 1991.
- [21] J. J. DONGARRA AND E. GROSSE, *Distribution of mathematical software via electronic mail*, Communications of the ACM, 30 (1987), pp. 403–407.
- [22] J. J. DONGARRA, F. G. GUSTAFSON, AND A. KARP, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Review, 26 (1984), pp. 91–112.
- [23] J. J. DONGARRA, S. HAMMARLING, AND D. C. SORESENSEN, *Block reduction of matrices to condensed forms for eigenvalue computations*, JCAM, 27 (1989), pp. 215–227. (LAPACK Working Note #2).
- [24] J. DU CROZ AND N. J. HIGHAM, *Stability of methods for matrix inversion*, IMA J. Num. Anal., (1992). (LAPACK Working Note #27).
- [25] J. DU CROZ, P. J. D. MAYES, AND G. RADICATI DI BROZOLO, *Factorizations of band matrices using Level 3 BLAS*, Computer Science Dept. Technical Report CS-90-109, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #21).
- [26] K. A. GALLIVAN, R. J. PLEMMONS, AND A. H. SAMEH, *Parallel algorithms for dense linear algebra computations*, SIAM Review, 32 (1990), pp. 54–135.

- [27] B. S. GARBOW, J. M. BOYLE, J. J. DONGARRA, AND C. B. MOLER, *Matrix Eigensystem Routines – EISPACK Guide Extension*, vol. 51 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1977.
- [28] G. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 2nd ed., 1989.
- [29] A. GREENBAUM AND J. J. DONGARRA, *Experiments with QL/QR methods for the symmetric tridiagonal eigenproblem*, Computer Science Dept. Technical Report CS-89-92, University of Tennessee, Knoxville, 1989. (LAPACK Working Note #17).
- [30] W. W. HAGER, *Condition estimators*, SIAM J. Sci. Stat. Comput., 5 (1984), pp. 311–316.
- [31] N. J. HIGHAM, *Efficient algorithms for computing the condition number of a tridiagonal matrix*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 150–165.
- [32] ———, *A survey of condition number estimation for triangular matrices*, SIAM Review, 29 (1987), pp. 575–596.
- [33] ———, *FORTTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation*, ACM Trans. Math. Soft., 14 (1988), pp. 381–396.
- [34] ———, *Experience with a matrix norm estimator*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 804–809.
- [35] T. KATO, *Perturbation Theory for Linear Operators*, Springer Verlag, Berlin, 2 ed., 1980.
- [36] C. L. LAWSON, R. J. HANSON, D. KINCAID, AND F. T. KROGH, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.
- [37] B. PARLETT, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs, NJ, 1980.
- [38] R. SCHREIBER AND C. F. VAN LOAN, *A storage efficient WY representation for products of Householder transformations*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 53–57.
- [39] 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*, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1976.
- [40] G. W. STEWART, *Error and perturbation bounds for subspaces associated with certain eigenvalue problems*, SIAM Review, 15 (1973), pp. 727–764.
- [41] G. W. STEWART AND J.-G. SUN, *Matrix Perturbation Theory*, Academic Press, New York, 1990.
- [42] J. VARAH, *On the separation of two matrices*, SIAM J. Num. Anal., 16 (1979), pp. 216–222.
- [43] J. H. WILKINSON, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.
- [44] J. H. WILKINSON AND C. REINSCH, eds., *Handbook for Automatic Computation, vol 2.: Linear Algebra*, Springer-Verlag, Heidelberg, 1971.

Appendix A

Index of Driver and Computational Routines

Notes

1. This index lists related pairs of real and complex routines together, for example, SBDSQR and CBDSQR.
2. Driver routines are listed in bold type, for example **SGBSV** and **CGBSV**.
3. Routines are listed in alphanumeric order of the real (single precision) routine name (which always begins with S-). (See subsection 2.1.3 for details of the LAPACK naming scheme.)
4. Double precision routines are not listed here; they have names beginning with D- instead of S-, or Z- instead of C-.
5. This index gives only a brief description of the purpose of each routine. For a precise description, consult the specifications in Appendix F, where the routines appear in the same order as here.
6. The text of the descriptions applies to both real and complex routines, except where alternative words or phrases are indicated, for example “symmetric/Hermitian”, “orthogonal/unitary” or “quasi-triangular/triangular”. For the real routines A^H is equivalent to A^T . (The same convention is used in Appendix F.)
7. In a few cases, three routines are listed together, one for real symmetric, one for complex symmetric, and one for complex Hermitian matrices (for example SSPCON, CSPCON and CHPCON).
8. A few routines for real matrices have no complex equivalent (for example SSTEbZ).

Routine		Description
real	complex	
SBDSQR	CBDSQR	Computes the singular value decomposition (SVD) of a real bidiagonal matrix, using the bidiagonal QR algorithm.
SGBCON	CGBCON	Estimates the reciprocal of the condition number of a general band matrix, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGBTRF/CGBTRF.
SGBEQU	CGBEQU	Computes row and column scalings to equilibrate a general band matrix and reduce its condition number.
SGBRFS	CGBRFS	Improves the computed solution to a general banded system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides forward and backward error bounds for the solution.
SGBSV	CGBSV	Solves a general banded system of linear equations $AX = B$.
SGBSVX	CGBSVX	Solves a general banded system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides an estimate of the condition number and error bounds on the solution.
SGBTRF	CGBTRF	Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges.
SGBTRS	CGBTRS	Solves a general banded system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, using the LU factorization computed by SGBTRF/CGBTRF.
SGEBAK	CGEBAK	Transforms eigenvectors of a balanced matrix to those of the original matrix supplied to SGEBAL/CGEBAL.
SGEBAL	CGEBAL	Balances a general matrix in order to improve the accuracy of computed eigenvalues.
SGEBRD	CGEBRD	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation.
SGECON	CGECON	Estimates the reciprocal of the condition number of a general matrix, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGETRF/CGETRF.
SGEQU	CGEQU	Computes row and column scalings to equilibrate a general rectangular matrix and reduce its condition number.
SGEES	CGEES	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
SGEESX	CGEESX	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization so that selected eigenvalues are at the top left of the Schur form, and computes reciprocal condition numbers for the average of the selected eigenvalues, and for the associated right invariant subspace.
SGEEV	CGEEV	Computes the eigenvalues and left and right eigenvectors of a general matrix.
SGEEVX	CGEEVX	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary balancing of the matrix, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Routine		Description
real	complex	
SGEHRD	CGEHRD	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation.
SGELQF	CGELQF	Computes an LQ factorization of a general rectangular matrix.
SGELS	CGELS	Computes the least-squares solution to an over-determined system of linear equations, $AX = B$ or $A^H X = B$, or the minimum-norm solution of an under-determined system, where A is a general rectangular matrix of full rank, using a QR or LQ factorization of A .
SGELSS	CGELSS	Computes the minimum-norm least-squares solution to an over- or under-determined system of linear equations $AX = B$, using the singular value decomposition of A .
SGELSX	CGELSX	Computes the minimum-norm least-squares solution to an over- or under-determined system of linear equations $AX = B$, using a complete orthogonal factorization of A .
SGEQLF	CGEQLF	Computes a QL factorization of a general rectangular matrix.
SGEQPF	CGEQPF	Computes a QR factorization with column pivoting of a general rectangular matrix.
SGEQRF	CGEQRF	Computes a QR factorization of a general rectangular matrix.
SGERFS	CGERFS	Improves the computed solution to a general system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides forward and backward error bounds for the solution.
SGERQF	CGERQF	Computes an RQ factorization of a general rectangular matrix.
SGESV	CGESV	Solves a general system of linear equations $AX = B$.
SGESVD	CGESVD	Computes the singular value decomposition (SVD) of a general rectangular matrix.
SGESVX	CGESVX	Solves a general system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides an estimate of the condition number and error bounds on the solution.
SGETRF	CGETRF	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges.
SGETRI	CGETRI	Computes the inverse of a general matrix, using the LU factorization computed by SGETRF/CGETRF.
SGETRS	CGETRS	Solves a general system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, using the LU factorization computed by SGETRF/CGETRF.
SGTCON	CGTCON	Estimates the reciprocal of the condition number of a general tridiagonal matrix, in either the 1-norm or the infinity-norm, using the LU factorization computed by SGTTRF/CGTTRF.
SGTRFS	CGTRFS	Improves the computed solution to a general tridiagonal system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides forward and backward error bounds for the solution.
SGTSV	CGTSV	Solves a general tridiagonal system of linear equations $AX = B$.

Routine		Description
real	complex	
SGTSVX	CGTSVX	Solves a general tridiagonal system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, and provides an estimate of the condition number and error bounds on the solution.
SGTTRF	CGTTRF	Computes an LU factorization of a general tridiagonal matrix, using partial pivoting with row interchanges.
SGTTRS	CGTTRS	Solves a general tridiagonal system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, using the LU factorization computed by SGTTRF/CGTTRF.
SHSEIN	CHSEIN	Computes specified right and/or left eigenvectors of an upper Hessenberg matrix by inverse iteration.
SHSEQR	CHSEQR	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the multishift QR algorithm.
SOPGTR	CUPGTR	Generates the orthogonal/unitary transformation matrix from a reduction to tridiagonal form determined by SSPTRD/CHPTRD.
SOPMTR	CUPMTR	Multiplies a general matrix by the orthogonal/unitary transformation matrix from a reduction to tridiagonal form determined by SSPTRD/CHPTRD.
SORGBR	CUNGBR	Generates the orthogonal/unitary transformation matrices from a reduction to bidiagonal form determined by SGEBRD/CGEBRD.
SORGHR	CUNGHR	Generates the orthogonal/unitary transformation matrix from a reduction to Hessenberg form determined by SGEHRD/CGEHRD.
SORGLQ	CUNGLQ	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by SGELQF/CGELQF.
SORGQL	CUNGQL	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by SGEQLF/CGEQLF.
SORGQR	CUNGQR	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by SGEQRF/CGEQRF.
SORGRQ	CUNGRQ	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by SGERQF/CGERQF.
SORGTR	CUNGTR	Generates the orthogonal/unitary transformation matrix from a reduction to tridiagonal form determined by SSYTRD/CHETRD.
SORMBR	CUNMBR	Multiplies a general matrix by one of the orthogonal/unitary transformation matrices from a reduction to bidiagonal form determined by SGEBRD/CGEBRD.
SORMHR	CUNMHR	Multiplies a general matrix by the orthogonal/unitary transformation matrix from a reduction to Hessenberg form determined by SGEHRD/CGEHRD.
SORMLQ	CUNMLQ	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by SGELQF/CGELQF.
SORMQL	CUNMQL	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by SGEQLF/CGEQLF.
SORMQR	CUNMQR	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by SGEQRF/CGEQRF.

Routine		Description
real	complex	
SORMRQ	CUNMRQ	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by SGERQF/CGERQF.
SORMTR	CUNMTR	Multiplies a general matrix by the orthogonal/unitary transformation matrix from a reduction to tridiagonal form determined by SSYTRD/CHETRD.
SPBCON	CPBCON	Estimates the reciprocal of the condition number of a symmetric/Hermitian positive definite band matrix, using the Cholesky factorization computed by SPBTRF/CPBTRF.
SPBEQU	CPBEQU	Computes row and column scalings to equilibrate a symmetric/Hermitian positive definite band matrix and reduce its condition number.
SPBRFS	CPBRFS	Improves the computed solution to a symmetric/Hermitian positive definite banded system of linear equations $AX = B$, and provides forward and backward error bounds for the solution.
SPBSV	CPBSV	Solves a symmetric/Hermitian positive definite banded system of linear equations $AX = B$.
SPBSVX	CPBSVX	Solves a symmetric/Hermitian positive definite banded system of linear equations $AX = B$, and provides an estimate of the condition number and error bounds on the solution.
SPBTRF	CPBTRF	Computes the Cholesky factorization of a symmetric/Hermitian positive definite band matrix.
SPBTRS	CPBTRS	Solves a symmetric/Hermitian positive definite banded system of linear equations $AX = B$, using the Cholesky factorization computed by SPBTRF/CPBTRF.
SPOCON	CPOCON	Estimates the reciprocal of the condition number of a symmetric/Hermitian positive definite matrix, using the Cholesky factorization computed by SPOTRF/CPOTRF.
SPOEQU	CPOEQU	Computes row and column scalings to equilibrate a symmetric/Hermitian positive definite matrix and reduce its condition number.
SPORFS	CPORFS	Improves the computed solution to a symmetric/Hermitian positive definite system of linear equations $AX = B$, and provides forward and backward error bounds for the solution.
SPOSV	CPOSV	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$.
SPOSVX	CPOSVX	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$, and provides an estimate of the condition number and error bounds on the solution.
SPOTRF	CPOTRF	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix.
SPOTRI	CPOTRI	Computes the inverse of a symmetric/Hermitian positive definite matrix, using the Cholesky factorization computed by SPOTRF/CPOTRF.

Routine		Description
real	complex	
SPOTRS	CPOTRS	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$, using the Cholesky factorization computed by SPOTRF/CPOTRF.
SPPCON	CPPCON	Estimates the reciprocal of the condition number of a symmetric/Hermitian positive definite matrix in packed storage, using the Cholesky factorization computed by SPPTRF/CPPTRF.
SPPEQU	CPPEQU	Computes row and column scalings to equilibrate a symmetric/Hermitian positive definite matrix in packed storage and reduce its condition number.
SPPRFS	CPPRFS	Improves the computed solution to a symmetric/Hermitian positive definite system of linear equations $AX = B$, where A is held in packed storage, and provides forward and backward error bounds for the solution.
SPPSV	CPPSV	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$, where A is held in packed storage.
SPPSVX	CPPSVX	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$, where A is held in packed storage, and provides an estimate of the condition number and error bounds on the solution.
SPPTRF	CPPTRF	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix in packed storage.
SPPTRI	CPPTRI	Computes the inverse of a symmetric/Hermitian positive definite matrix in packed storage, using the Cholesky factorization computed by SPPTRF/CPPTRF.
SPPTRS	CPPTRS	Solves a symmetric/Hermitian positive definite system of linear equations $AX = B$, where A is held in packed storage, using the Cholesky factorization computed by SPPTRF/CPPTRF.
SPTCON	CPTCON	Computes the reciprocal of the condition number of a symmetric/Hermitian positive definite tridiagonal matrix, using the LDL^H factorization computed by SPTTRF/CPTTRF.
SPTEQR	CPTEQR	Computes all eigenvalues and eigenvectors of a real symmetric positive definite tridiagonal matrix, by computing the SVD of its bidiagonal Cholesky factor.
SPTRFS	CPTRFS	Improves the computed solution to a symmetric/Hermitian positive definite tridiagonal system of linear equations $AX = B$, and provides forward and backward error bounds for the solution.
SPTSV	CPTSV	Solves a symmetric/Hermitian positive definite tridiagonal system of linear equations $AX = B$.
SPTSVX	CPTSVX	Solves a symmetric/Hermitian positive definite tridiagonal system of linear equations $AX = B$, and provides an estimate of the condition number and error bounds on the solution.
SPTTRF	CPTTRF	Computes the LDL^H factorization of a symmetric/Hermitian positive definite tridiagonal matrix.

Routine		Description
real	complex	
SPTTRS	CPTTRS	Solves a symmetric/Hermitian positive definite tridiagonal system of linear equations, using the LDL^H factorization computed by SPTTRF/CPTTRF.
SSBEV	CHBEV	Computes all eigenvalues and eigenvectors of a symmetric/Hermitian band matrix.
SSBEVX	CHBEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian band matrix.
SSBTRD	CHBTRD	Reduces a symmetric/Hermitian band matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation.
SSPCON	CSPCON CHPCON	Estimates the reciprocal of the condition number of a real symmetric/complex symmetric/complex Hermitian indefinite matrix in packed storage, using the factorization computed by SSPTRF/CSPTRF/CHPTRF.
SSPEV	CHPEV	Computes all eigenvalues and eigenvectors of a symmetric/Hermitian matrix in packed storage.
SSPEVX	CHPEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian matrix in packed storage.
SSPGST	CHPGST	Reduces a symmetric/Hermitian-definite generalized eigenproblem $Ax = \lambda Bx$, $ABx = \lambda x$, or $BAX = \lambda x$, to standard form, where A and B are held in packed storage, and B has been factorized by SPTTRF/CPPTTRF.
SSPGV	CHPGV	Computes all eigenvalues and eigenvectors of a generalized symmetric/Hermitian-definite generalized eigenproblem, $Ax = \lambda Bx$, $ABx = \lambda x$, or $BAX = \lambda x$, where A and B are in packed storage.
SSPRFS	CSPRFS CHPRFS	Improves the computed solution to a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, where A is held in packed storage, and provides forward and backward error bounds for the solution.
SSPSV	CSPSV CHPSV	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, where A is held in packed storage.
SSPSVX	CSPSVX CHPSVX	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, where A is held in packed storage, and provides an estimate of the condition number and error bounds on the solution.
SSPTRD	CHPTRD	Reduces a symmetric/Hermitian matrix in packed storage to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation.
SSPTRF	CSPTRF CHPTRF	Computes the factorization of a real symmetric/complex symmetric/complex Hermitian indefinite matrix in packed storage, using the diagonal pivoting method.

Routine		Description
real	complex	
SSPTRI	CSPTRI CHPTRI	Computes the inverse of a real symmetric/complex symmetric/complex Hermitian indefinite matrix in packed storage, using the factorization computed by SSPTRF/CSPTRF/CHPTRF.
SSPTRS	CSPTRS CHPTRS	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, where A is held in packed storage, using the factorization computed by SSPTRF/CSPTRF/CHPTRF.
SSTEBZ		Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.
SSTEIN	CSTEIN	Computes selected eigenvectors of a real symmetric tridiagonal matrix by inverse iteration.
SSTEQR	CSTEQR	Computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix, using the implicit QL or QR algorithm.
SSTERF		Computes all eigenvalues of a real symmetric tridiagonal matrix, using a root-free variant of the QL or QR algorithm.
SSTEV		Computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
SSTEVM		Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
SSYCON	CSYCON CHECON	Estimates the reciprocal of the condition number of a real symmetric/complex symmetric/complex Hermitian indefinite matrix, using the factorization computed by SSYTRF/CSYTRF/CHETRF.
SSYEVM	CHEEV	Computes all eigenvalues and eigenvectors of a symmetric/Hermitian matrix.
SSYEVX	CHEEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian matrix.
SSYGST	CHEGST	Reduces a symmetric/Hermitian-definite generalized eigenproblem $Ax = \lambda Bx$, $ABx = \lambda x$, or $BAX = \lambda x$, to standard form, where B has been factorized by SPOTRF/CPOTRF.
SSYGV	CHEGV	Computes all eigenvalues and the eigenvectors of a generalized symmetric/Hermitian-definite generalized eigenproblem, $Ax = \lambda Bx$, $ABx = \lambda x$, or $BAX = \lambda x$.
SSYRFS	CSYRFS CHERFS	Improves the computed solution to a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, and provides forward and backward error bounds for the solution.
SSYSV	CSYSV CHESV	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$.
SSYSVX	CSYSVX CHESVX	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, and provides an estimate of the condition number and error bounds on the solution.
SSYTRD	CHETRD	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation.

Routine		Description
real	complex	
SSYTRF	CSYTRF CHETRF	Computes the factorization of a real symmetric/complex symmetric/complex Hermitian indefinite matrix, using the diagonal pivoting method.
SSYTRI	CSYTRI CHETRI	Computes the inverse of a real symmetric/complex symmetric/complex Hermitian indefinite matrix, using the factorization computed by SSYTRF/CSYTRF/CHETRF.
SSYTRS	CSYTRS CHETRS	Solves a real symmetric/complex symmetric/complex Hermitian indefinite system of linear equations $AX = B$, using the factorization computed by SSPTRF/CSPTRF/CHPTRF.
STBCON	CTBCON	Estimates the reciprocal of the condition number of a triangular band matrix, in either the 1-norm or the infinity-norm.
STBRFS	CTBRFS	Provides forward and backward error bounds for the solution of a triangular banded system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$.
STBTRS	CTBTRS	Solves a triangular banded system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$.
STPCON	CTPCON	Estimates the reciprocal of the condition number of a triangular matrix in packed storage, in either the 1-norm or the infinity-norm.
STPRFS	CTPRFS	Provides forward and backward error bounds for the solution of a triangular system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, where A is held in packed storage.
STPTRI	CTPTRI	Computes the inverse of a triangular matrix in packed storage.
STPTRS	CTPTRS	Solves a triangular system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$, where A is held in packed storage.
STRCON	CTRCON	Estimates the reciprocal of the condition number of a triangular matrix, in either the 1-norm or the infinity-norm.
STREVC	CTREVC	Computes left and right eigenvectors of an upper quasi-triangular/triangular matrix.
STREXC	CTREXC	Reorders the Schur factorization of a matrix by a unitary similarity transformation.
STRRFS	CTRRFS	Provides forward and backward error bounds for the solution of a triangular system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$.
STRSEN	CTRSN	Reorders the Schur factorization of a matrix in order to find an orthonormal basis of a right invariant subspace corresponding to selected eigenvalues, and returns reciprocal condition numbers (sensitivities) of the average of the cluster of eigenvalues and of the invariant subspace.
STRSNA	CTRSNA	Estimates the reciprocal condition numbers (sensitivities) of selected eigenvalues and eigenvectors of an upper quasi-triangular/triangular matrix.
STRSYL	CTRSYL	Solves the Sylvester matrix equation $AX \pm XB = C$ where A and B are upper quasi-triangular/triangular, and may be transposed.

Routine		Description
real	complex	
STRTRI	CTRTRI	Computes the inverse of a triangular matrix.
STRTRS	CTRTRS	Solves a triangular system of linear equations $AX = B$, $A^T X = B$ or $A^H X = B$.
STZRQF	CTZRQF	Computes an RQ factorization of an upper trapezoidal matrix.

Appendix B

Index of Auxiliary Routines

Notes

1. This index lists related pairs of real and complex routines together, in the same style as in Appendix A.
2. Routines are listed in alphanumeric order of the real (single precision) routine name (which always begins with S-). (See subsection 2.1.3 for details of the LAPACK naming scheme.)
3. A few complex routines have no real equivalents, and they are listed first; routines listed in italics (for example, *CROT*), have real equivalents in the Level 1 or Level 2 BLAS.
4. Double precision routines are not listed here; they have names beginning with D- instead of S-, or Z- instead of C-. The only exceptions to this simple rule are that the double precision versions of ICMAX1, SCSUM1 and CSRSL are named IZMAX1, DZSUM1 and ZDRSL.
5. A few routines in the list have names that are independent of data type: ILAENV, LSAME, LSAMEN and XERBLA.
6. This index gives only a brief description of the purpose of each routine. For a precise description consult the leading comments in the code, which have been written in the same style as for the driver and computational routines.

Routine		Description
real	complex	
	CLACGV	Conjugates a complex vector.
	CLACRT	Applies a plane rotation with complex cosine and sine to a pair of complex vectors.
	CLAESY	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.
	CROT	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
	CSPMV	Computes the matrix-vector product $y = \alpha Ax + \beta y$, where α and β are complex scalars, x and y are complex vectors and A is a complex symmetric matrix in packed storage.
	CSPR	Performs the symmetric rank-1 update $A = \alpha xx^T + A$, where α is a complex scalar, x is a complex vector and A is a complex symmetric matrix in packed storage.
	CSROT	Applies a plane rotation with real cosine and sine to a pair of complex vectors.
	CSYMV	Computes the matrix-vector product $y = \alpha Ax + \beta y$, where α and β are complex scalars, x and y are complex vectors and A is a complex symmetric matrix.
	CSYR	Performs the symmetric rank-1 update $A = \alpha xx^T + A$, where α is a complex scalar, x is a complex vector and A is a complex symmetric matrix.
	ICMAX1	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 BLAS ICAMAX, but using the absolute value of the real part).
ILAENV		Environmental enquiry function which returns values for tuning algorithmic performance.
LSAME		Tests two characters for equality regardless of case.
LSAMEN		Tests two character strings for equality regardless of case.
	SCSUM1	Forms the 1-norm of a complex vector (similar to the Level 1 BLAS SCASUM, but using the true absolute value).
SGBTF2	CGBTF2	Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges (unblocked algorithm).
SGBD2	CGEBD2	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
SGEHD2	CGEHD2	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
SGELQ2	CGELQ2	Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).
SGEQL2	CGEQL2	Computes a QL factorization of a general rectangular matrix (unblocked algorithm).
SGEQR2	CGEQR2	Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Routine		Description
real	complex	
SGERQ2	CGERQ2	Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).
SGETF2	CGETF2	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (unblocked algorithm).
SLABAD		Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
SLABRD	CLABRD	Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
SLACON	CLACON	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
SLACPY	CLACPY	Copies all or part of one two-dimensional array to another.
SLADIV	CLADIV	Performs complex division in real arithmetic, avoiding unnecessary overflow.
SLAE2		Computes the eigenvalues of a 2-by-2 symmetric matrix.
SLAEBZ		Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine SSTEBC.
SLAEIN	CLAEIN	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
SLAEV2	CLAEV2	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
SLAEXC		Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
SLAGTF		Computes an LU factorization of a matrix $(T - \lambda I)$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
SLAGTM	CLAGTM	Performs a matrix-matrix product of the form $C = \alpha AB + \beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
SLAGTS		Solves the system of equations $(T - \lambda I)x = y$ or $(T - \lambda I)^T x = y$, where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by SLAGTF.
SLAHQR	CLAHQR	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
SLAHRD	CLAHRD	Reduces the first nb columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
SLAIC1	CLAIC1	Applies one step of incremental condition estimation.

Routine		Description
real	complex	
SLALN2		Solves a 1-by-1 or 2-by-2 system of equations of the form $(\gamma A - \lambda D)x = \sigma b$ or $(\gamma A^T - \lambda D)x = \sigma b$, where D is a diagonal matrix, λ , b and x may be complex, and σ is a scale factor set to avoid overflow.
SLAMCH		Determines machine parameters for floating-point arithmetic.
SLANGB	CLANGB	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general band matrix.
SLANGE	CLANGE	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
SLANGT	CLANGT	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general tridiagonal matrix.
SLANHS	CLANHS	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
SLANSB	CLANSB CLANHB	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric/complex symmetric/complex Hermitian band matrix.
SLANSP	CLANSP CLANHP	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric/complex symmetric/complex Hermitian matrix in packed storage.
SLANST	CLANST	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a symmetric/Hermitian tridiagonal matrix.
SLANSY	CLANSY CLANHE	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric/complex symmetric/complex Hermitian matrix.
SLANTB	CLANTB	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular band matrix.
SLANTP	CLANTP	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix in packed storage.
SLANTR	CLANTR	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
SLANV2		Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in Schur canonical form.
SLAPY2		Returns $\sqrt{x^2 + y^2}$, avoiding unnecessary overflow or harmful underflow.
SLAPY3		Returns $\sqrt{x^2 + y^2 + z^2}$, avoiding unnecessary overflow or harmful underflow.
SLAQGB	CLAQGB	Scales a general band matrix, using row and column scaling factors computed by SGBEQU/CGBEQU.

Routine		Description
real	complex	
SLAQGE	CLAQGE	Scales a general rectangular matrix, using row and column scaling factors computed by SGEEQU/CGEEQU.
SLAQSB	CLAQSB	Scales a symmetric/Hermitian band matrix, using scaling factors computed by SPBEQU/CPBEQU.
SLAQSP	CLAQSP	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by SPPEQU/CPPEQU.
SLAQSY	CLAQSY	Scales a symmetric/Hermitian matrix, using scaling factors computed by SPOEQU/CPOEQU.
SLAQTR		Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
SLAR2V	CLAR2V	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
SLARF	CLARF	Applies an elementary reflector to a general rectangular matrix.
SLARFB	CLARFB	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
SLARFG	CLARFG	Generates an elementary reflector (Householder matrix).
SLARFT	CLARFT	Forms the triangular factor T of a block reflector $H = I - VTV^H$.
SLARFX	CLARFX	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10 .
SLARGV	CLARGV	Generates a vector of plane rotations with real cosines and real/complex sines.
SLARNV	CLARNV	Returns a vector of random numbers from a uniform or normal distribution.
SLARTG	CLARTG	Generates a plane rotation with real cosine and real/complex sine.
SLARTV	CLARTV	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
SLARUV		Returns a vector of n random real numbers from a uniform (0,1) distribution ($n \leq 128$).
SLAS2		Computes the singular values of a 2-by-2 triangular matrix.
SLASCL	CLASCL	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .
SLASET	CLASET	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
SLASR	CLASR	Applies a sequence of plane rotations to a general rectangular matrix.
SLASSQ	CLASSQ	Updates a sum of squares represented in scaled form.
SLASV2		Computes the singular value decomposition of a 2-by-2 triangular matrix.
SLASWP	CLASWP	Performs a sequence of row interchanges on a general rectangular matrix.
SLASY2		Solves the Sylvester matrix equation $AX \pm XB = \sigma C$ where A and B are of order 1 or 2, and may be transposed, and σ is a scale factor.

Routine		Description
real	complex	
SLASYF	CLASYF CLAHEF	Computes a partial factorization of a real symmetric/complex symmetric/complex Hermitian indefinite matrix, using the diagonal pivoting method.
SLATBS	CLATBS	Solves a triangular banded system of equations $Ax = \sigma b$, $A^T x = \sigma b$, or $A^H x = \sigma b$, where σ is a scale factor set to prevent overflow.
SLATPS	CLATPS	Solves a triangular system of equations $Ax = \sigma b$, $A^T x = \sigma b$, or $A^H x = \sigma b$, where A is held in packed storage, and σ is a scale factor set to prevent overflow.
SLATRD	CLATRD	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
SLATRS	CLATRS	Solves a triangular system of equations $Ax = \sigma b$, $A^T x = \sigma b$, or $A^H x = \sigma b$, where σ is a scale factor set to prevent overflow.
SLATZM	CLATZM	Applies an elementary reflector generated by STZRQF/CTZRQF to a general rectangular matrix.
SLAUU2	CLAUU2	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices (unblocked algorithm).
SLAUUM	CLAUUM	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices.
SLAZRO	CLAZRO	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
SORG2L	CUNG2L	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by SGEQLF/CGEQLF (unblocked algorithm).
SORG2R	CUNG2R	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by SGEQRF/CGEQRF (unblocked algorithm).
SORGL2	CUNGL2	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by SGELQF/CGELQF (unblocked algorithm).
SORGR2	CUNGR2	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by SGERQF/CGERQF (unblocked algorithm).
SORM2L	CUNM2L	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by SGEQLF/CGEQLF (unblocked algorithm).
SORM2R	CUNM2R	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by SGEQRF/CGEQRF (unblocked algorithm).
SORML2	CUNML2	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by SGELQF/CGELQF (unblocked algorithm).

Routine		Description
real	complex	
SORMR2	CUNMR2	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by SGERQF/CGERQF (unblocked algorithm).
SPBTF2	CPBTF2	Computes the Cholesky factorization of a symmetric/Hermitian positive definite band matrix (unblocked algorithm).
SPOTF2	CPOTF2	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
SRSCL	CSRSCL	Multiplies a vector by the reciprocal of a real scalar.
SSYGS2	CHEGS2	Reduces a symmetric/Hermitian-definite generalized eigenproblem $Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$, to standard form, where B has been factorized by SPOTRF/CPOTRF (unblocked algorithm).
SSYTD2	CHETD2	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
SSYTF2	CSYTF2 CHETF2	Computes the factorization of a real symmetric/complex symmetric/complex Hermitian indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
STR12	CTR12	Computes the inverse of a triangular matrix (unblocked algorithm).
XERBLA		Error handling routine called by LAPACK routines if an input parameter has an invalid value.

Appendix C

Quick Reference Guide to the BLAS

Level 1 BLAS

dim scalar vector vector scalars

5-element prefixes
array

```

SUBROUTINE _ROTG (                                A, B, C, S )      S, D
SUBROUTINE _ROTMG(                                D1, D2, A, B,    PARAM ) S, D
SUBROUTINE _ROT ( N,                                X, INCX, Y, INCY,    C, S )      S, D
SUBROUTINE _ROTM ( N,                                X, INCX, Y, INCY,    PARAM ) S, D
SUBROUTINE _SWAP ( N,                                X, INCX, Y, INCY )      S, D, C, Z
SUBROUTINE _SCAL ( N, ALPHA, X, INCX )              S, D, C, Z, CS, ZD
SUBROUTINE _COPY ( N,                                X, INCX, Y, INCY )      S, D, C, Z
SUBROUTINE _AXPY ( N, ALPHA, X, INCX, Y, INCY )      S, D, C, Z
FUNCTION _DOT ( N,                                X, INCX, Y, INCY )      S, D, DS
FUNCTION _DOTU ( N,                                X, INCX, Y, INCY )      C, Z
FUNCTION _DOTC ( N,                                X, INCX, Y, INCY )      C, Z
FUNCTION _DDOT ( N, ALPHA, X, INCX, Y, INCY )      SDS
FUNCTION _NRM2 ( N,                                X, INCX )              S, D, SC, DZ
FUNCTION _ASUM ( N,                                X, INCX )              S, D, SC, DZ
FUNCTION _LAMAX ( N,                                X, INCX )              S, D, C, Z

```

Name	Operation	Prefixes
_ROTG	Generate plane rotation	S, D
_ROTMG	Generate modified plane rotation	S, D
_ROT	Apply plane rotation	S, D
_ROTM	Apply modified plane rotation	S, D
_SWAP	$x \leftrightarrow y$	S, D, C, Z
_SCAL	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
_COPY	$y \leftarrow x$	S, D, C, Z
_AXPY	$y \leftarrow \alpha x + y$	S, D, C, Z
_DOT	$dot \leftarrow x^T y$	S, D, DS
_DOTU	$dot \leftarrow x^T y$	C, Z
_DOTC	$dot \leftarrow x^H y$	C, Z
_DDOT	$dot \leftarrow \alpha + x^T y$	SDS
_NRM2	$nrm2 \leftarrow \ x\ _2$	S, D, SC, DZ
_ASUM	$asum \leftarrow \ re(x)\ _1 + \ im(x)\ _1$	S, D, SC, DZ
_LAMAX	$amax \leftarrow 1^{st} k \ni re(x_k) + im(x_k) $ $= \max(re(x_i) + im(x_i))$	S, D, C, Z

Level 2 BLAS

options	dim	b-width	scalar	matrix	vector	scalar	vector	prefixes
_GEMV (M, N,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	S, D, C, Z	
_GEMV (M, N, KL, KU,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	S, D, C, Z	
_HEMV (UPLO,	N,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	C, Z	
_HEMV (UPLO,	N, K,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	C, Z	
_HPMV (UPLO,	N,		ALPHA, AP,	X,	INCX, BETA, Y,	INCY)	C, Z	
_SYMV (UPLO,	N,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	S, D	
_SBMV (UPLO,	N, K,		ALPHA, A,	LDA, X,	INCX, BETA, Y,	INCY)	S, D	
_SPMV (UPLO,	N,		ALPHA, AP,	X,	INCX, BETA, Y,	INCY)	S, D	
_TRMV (UPLO, TRANS, DIAG,	N,		A,	LDA, X,	INCX)		S, D, C, Z	
_TBMV (UPLO, TRANS, DIAG,	N, K,		A,	LDA, X,	INCX)		S, D, C, Z	
_TPMV (UPLO, TRANS, DIAG,	N,		AP,	X,	INCX)		S, D, C, Z	
_TRSV (UPLO, TRANS, DIAG,	N,		A,	LDA, X,	INCX)		S, D, C, Z	
_TBSV (UPLO, TRANS, DIAG,	N, K,		A,	LDA, X,	INCX)		S, D, C, Z	
_TPSV (UPLO, TRANS, DIAG,	N,		AP,	X,	INCX)		S, D, C, Z	

options	dim	scalar	vector	vector	matrix	prefixes
_GER (M, N,	ALPHA, X,	INCX, Y,	INCY, A,	LDA)	S, D
_GERU (M, N,	ALPHA, X,	INCX, Y,	INCY, A,	LDA)	C, Z
_GERC (M, N,	ALPHA, X,	INCX, Y,	INCY, A,	LDA)	C, Z
_HER (UPLO,	N,	ALPHA, X,	INCX,	A,	LDA)	C, Z
_HPR (UPLO,	N,	ALPHA, X,	INCX,	AP)		C, Z
_HER2 (UPLO,	N,	ALPHA, X,	INCX, Y,	INCY, A,	LDA)	C, Z
_HPR2 (UPLO,	N,	ALPHA, X,	INCX, Y,	INCY, AP)		C, Z
_SYR (UPLO,	N,	ALPHA, X,	INCX,	A,	LDA)	S, D
_SPR (UPLO,	N,	ALPHA, X,	INCX,	AP)		S, D
_SYR2 (UPLO,	N,	ALPHA, X,	INCX, Y,	INCY, A,	LDA)	S, D
_SPR2 (UPLO,	N,	ALPHA, X,	INCX, Y,	INCY, AP)		S, D

Level 3 BLAS

options	dim	scalar	matrix	matrix	scalar	matrix	prefixes
_GEMM (M, N, K,	ALPHA, A,	LDA, B,	LDB, BETA, C,	LDC)	S, D, C, Z	
_SYMM (SIDE, UPLO,	M, N,	ALPHA, A,	LDA, B,	LDB, BETA, C,	LDC)	S, D, C, Z	
_HEMM (SIDE, UPLO,	M, N,	ALPHA, A,	LDA, B,	LDB, BETA, C,	LDC)	C, Z	
_SYRK (N, K,	ALPHA, A,	LDA,	BETA, C,	LDC)	S, D, C, Z	
_HERK (N, K,	ALPHA, A,	LDA,	BETA, C,	LDC)	C, Z	
_SYR2K (N, K,	ALPHA, A,	LDA, B,	LDB, BETA, C,	LDC)	S, D, C, Z	
_HER2K (N, K,	ALPHA, A,	LDA, B,	LDB, BETA, C,	LDC)	C, Z	
_TRMM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A,	LDA, B,	LDB)		S, D, C, Z	
_TRSM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A,	LDA, B,	LDB)		S, D, C, Z	

Name	Operation	Prefixes
_GEMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
_GBMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
_HEMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HBMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HPMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_SYMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SBMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SPMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_TRMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TBMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TPMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TRSV	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
_TBSV	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
_TPSV	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
_GER	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
_GERU	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
_GERC	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
_HER	$A \leftarrow \alpha xx^H + A$	C, Z
_HPR	$A \leftarrow \alpha xx^H + A$	C, Z
_HER2	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
_HPR2	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
_SYR	$A \leftarrow \alpha xx^T + A$	S, D
_SPR	$A \leftarrow \alpha xx^T + A$	S, D
_SYR2	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
_SPR2	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Name	Operation	Prefixes
_GEMM	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
_SYMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
_HEMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
_SYRK	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
_HERK	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
_SYR2K	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C, C \leftarrow \alpha A^T B + \alpha B^T A + \beta C, C - n \times n$	S, D, C, Z
_HER2K	$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C, C \leftarrow \alpha A^H B + \alpha B^H A + \beta C, C - n \times n$	C, Z
_TRMM	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
_TRSM	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

Notes

Meaning of prefixes

S - REAL C - COMPLEX
D - DOUBLE PRECISION Z - COMPLEX*16 (this may not be supported
by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types

GE - GEneral	GB - GEneral Band	
SY - SYmmetric	SB - Symmetric Band	SP - Symmetric Packed
HE - HERmitian	HB - Hermitian Band	HP - Hermitian Packed
TR - TRIangular	TB - Triangular Band	TP - Triangular Packed

Options

Arguments describing options are declared as CHARACTER*1 and may be passed as character strings.

TRANS	= 'No transpose', 'Transpose', 'Conjugate transpose' (X , X^T , X^C)
UPLO	= 'Upper triangular', 'Lower triangular'
DIAG	= 'Non-unit triangular', 'Unit triangular'
SIDE	= 'Left', 'Right' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANS = 'T' and TRANS = 'C' have the same meaning.

For Hermitian matrices, TRANS = 'T' is not allowed.

For complex symmetric matrices, TRANS = 'H' is not allowed.

Appendix D

Converting from LINPACK or EISPACK

This appendix is designed to assist people to convert programs that currently call LINPACK or EISPACK routines, to call LAPACK routines instead.

Notes

1. The appendix consists mainly of indexes giving the nearest LAPACK equivalents of LINPACK and EISPACK routines. These indexes should not be followed blindly or rigidly, especially when two or more LINPACK or EISPACK routines are being used together: in many such cases one of the LAPACK driver routines may be a suitable replacement.
2. When two or more LAPACK routines are given in a single entry, these routines must be combined to achieve the equivalent function.
3. For LINPACK, an index is given for equivalents of the real LINPACK routines; these equivalences apply also to the corresponding complex routines. For EISPACK, an index is given for all real and complex routines, since there is no direct 1-to-1 correspondence between real and complex routines in EISPACK.
4. A few of the less commonly used routines in LINPACK and EISPACK have no equivalents in Release 1.0 of LAPACK; equivalents for some of these (but not all) are planned for a future release.
5. For some EISPACK routines, there are LAPACK routines providing similar functionality, but using a significantly different method; such routines are marked by a reference to this note. For example, the EISPACK routine ELMHES uses non-orthogonal transformations, whereas the nearest equivalent LAPACK routine, SGEHRD, uses orthogonal transformations.
6. In some cases the LAPACK equivalents require matrices to be stored in a different storage scheme. For example:

- EISPACK routines BANDR, BANDV, BQR and the driver routine RSB require the lower triangle of a symmetric band matrix to be stored in a different storage scheme to that used in LAPACK, which is illustrated in subsection 5.3.3. The corresponding storage scheme used by the EISPACK routines is:

symmetric band matrix A	EISPACK band storage
$\begin{pmatrix} a_{11} & a_{21} & a_{31} & & \\ a_{21} & a_{22} & a_{32} & a_{42} & \\ a_{31} & a_{32} & a_{33} & a_{43} & a_{53} \\ & a_{42} & a_{43} & a_{44} & a_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & * & a_{11} \\ * & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} \\ a_{42} & a_{43} & a_{44} \\ a_{53} & a_{54} & a_{55} \end{matrix}$

- EISPACK routines TRED1, TRED2, TRED3, HTRED3, HTRED1, TQL1, TQL2, IMTQL1, IMTQL2, RATQR, TQLRAT and the driver routine RST store the off-diagonal elements of a symmetric tridiagonal matrix in elements $2 : n$ of the array E, whereas LAPACK routines use elements $1 : n - 1$.
7. The EISPACK and LINPACK routines for the singular value decomposition return the matrix of right singular vectors, V , whereas the corresponding LAPACK routines return the transposed matrix V^T .
 8. In general, the argument lists of the LAPACK routines are different from those of the corresponding EISPACK and LINPACK routines, and the workspace requirements are often different.

LAPACK equivalents of LINPACK routines for real matrices		
LINPACK	LAPACK	Function of LINPACK routine
SCHDC		Cholesky factorization with diagonal pivoting option
SCHDD		rank-1 downdate of a Cholesky factorization or the triangular factor of a QR factorization
SCHEX		rank-1 update of a Cholesky factorization or the triangular factor of a QR factorization
SCHUD		modifies a Cholesky factorization under permutations of the original matrix
SGBCO	SLANGB SGBTRF SGBCON	LU factorization and condition estimation of a general band matrix
SGBDI		determinant of a general band matrix, after factorization by SGBCO or SGBFA
SGBFA	SGBTRF	LU factorization of a general band matrix
SGPSL	SGBTRS	solves a general band system of linear equations, after factorization by SGBCO or SGBFA
SGECO	SLANGE SGETRF SGECON	LU factorization and condition estimation of a general matrix
SGEDI	SGETRI	determinant and inverse of a general matrix, after factorization by SGECO or SGEFA
SGEFA	SGETRF	LU factorization of a general matrix
SGESL	SGETRS	solves a general system of linear equations, after factorization by SGECO or SGEFA
SGTSL	SGTSV	solves a general tridiagonal system of linear equations
SPBCO	SLANSB SPBTRF SPBCON	Cholesky factorization and condition estimation of a symmetric positive-definite band matrix
SPBDI		determinant of a symmetric positive-definite band matrix, after factorization by SPBCO or SPBFA
SPBFA	SPBTRF	Cholesky factorization of a symmetric positive-definite band matrix
SPBSL	SPBTRS	solves a symmetric positive-definite band system of linear equations, after factorization by SPBCO or SPBFA
SPOCO	SLANSY SPOTRF SPOCON	Cholesky factorization and condition estimation of a symmetric positive-definite matrix
SPODI	SPOTRI	determinant and inverse of a symmetric positive-definite matrix, after factorization by SPOCO or SPOFA
SPOFA	SPOTRF	Cholesky factorization of a symmetric positive-definite matrix
SPOSL	SPOTRS	solves a symmetric positive-definite system of linear equations, after factorization by SPOCO or SPOFA
SPPCO	SLANSY SPPTRF SPPCON	Cholesky factorization and condition estimation of a symmetric positive-definite matrix (packed storage)

LAPACK equivalents of LINPACK routines for real matrices (continued)		
LINPACK	LAPACK	Function of LINPACK routine
SPPDI	SPPTRI	determinant and inverse of a symmetric positive-definite matrix, after factorization by SPPCO or SPPFA (packed storage)
SPPFA	SPPTRF	Cholesky factorization of a symmetric positive-definite matrix (packed storage)
SPPSL	SPPTRS	solves a symmetric positive-definite system of linear equations, after factorization by SPPCO or SPPFA (packed storage)
SPTSL	SPTSV	solves a symmetric positive-definite tridiagonal system of linear equations
SQRDC	SGEQPF or SGEQRF	QR factorization with optional column pivoting
SQRSL	SORMQR STRSV	solves linear least squares problems after factorization by SQRDC
SSICO	SLANSY SSYTRF SSYCON	symmetric indefinite factorization and condition estimation of a symmetric indefinite matrix
SSIDI	SSYTRI	determinant, inertia and inverse of a symmetric indefinite matrix, after factorization by SSICO or SSIFA
SSIFA	SSYTRF	symmetric indefinite factorization of a symmetric indefinite matrix
SSISL	SSYTRS	solves a symmetric indefinite system of linear equations, after factorization by SSICO or SSIFA
SSPCO	SLANSP SSPTRF SSPCON	symmetric indefinite factorization and condition estimation of a symmetric indefinite matrix (packed storage)
SSPDI	SSPTRI	determinant, inertia and inverse of a symmetric indefinite matrix, after factorization by SSPCO or SSPFA (packed storage)
SSPFA	SSPTRF	symmetric indefinite factorization of a symmetric indefinite matrix (packed storage)
SSPSL	SSPTRS	solves a symmetric indefinite system of linear equations, after factorization by SSPCO or SSPFA (packed storage)
SSVDC	SGESVD	all or part of the singular value decomposition of a general matrix
STRCO	STRCON	condition estimation of a triangular matrix
STRDI	STRTRI	determinant and inverse of a triangular matrix
STRSL	STRTRS	solves a triangular system of linear equations

LAPACK equivalents of EISPACK routines		
EISPACK	LAPACK	Function of EISPACK routine
BAKVEC		Backtransform eigenvectors after transformation by FIGI
BALANC	SGEBAL	Balance a real matrix
BALBAK	SGEBAK	Backtransform eigenvectors of a real matrix after balancing by BALANC
BANDR	SSBTRD	Reduce a real symmetric band matrix to tridiagonal form
BANDV		Selected eigenvectors of a real band matrix by inverse iteration
BISECT	SSTEBZ	Eigenvalues in a specified interval of a real symmetric tridiagonal matrix
BQR	SSBEVX (note 5)	Some eigenvalues of a real symmetric band matrix
CBABK2	CGEBAK	Backtransform eigenvectors of a complex matrix after balancing by CBAL
CBAL	CGEBAL	Balance a complex matrix
CG	CGEEV	All eigenvalues and optionally eigenvectors of a complex general matrix (driver routine)
CH	CHEEV	All eigenvalues and optionally eigenvectors of a complex Hermitian matrix (driver routine)
CHVIT	CHSEIN	Selected eigenvectors of a complex upper Hessenberg matrix by inverse iteration
COMBAK	CUNMHR (note 5)	Backtransform eigenvectors of a complex matrix after reduction by COMHES
COMHES	CGEHRD (note 5)	Reduce a complex matrix to upper Hessenberg form by a non-unitary transformation
COMLR	CHSEQR (note 5)	All eigenvalues of a complex upper Hessenberg matrix, by the LR algorithm
COMLR2	CUNGHR CHSEQR CTREVC (note 5)	All eigenvalues/vectors of a complex matrix by the LR algorithm, after reduction by COMHES
COMQR	CHSEQR	All eigenvalues of a complex upper Hessenberg matrix by the QR algorithm
COMQR2	CUNGHR CHSEQR CTREVC	All eigenvalues/vectors of a complex matrix by the QR algorithm, after reduction by CORTH
CORTB	CUNMHR	Backtransform eigenvectors of a complex matrix, after reduction by CORTH
CORTH	CGEHRD	Reduce a complex matrix to upper Hessenberg form by a unitary transformation
ELMBAK	SORMHR (note 5)	Backtransform eigenvectors of a real matrix after reduction by ELMHES
ELMHES	SGEHRD (note 5)	Reduce a real matrix to upper Hessenberg form by a non-orthogonal transformation
ELTRAN	SORGHR (note 5)	Generate transformation matrix used by ELMHES

LAPACK equivalents of EISPACK routines (continued)		
EISPACK	LAPACK	Function of EISPACK routine
FIGI		Transform a nonsymmetric tridiagonal matrix of special form to a symmetric matrix
FIG2		As FIGI, with generation of the transformation matrix
HQR	SHSEQR	All eigenvalues of a complex upper Hessenberg matrix by the QR algorithm
HQR2	SHSEQR STREVC	All eigenvalues/vectors of a real upper Hessenberg matrix by the QR algorithm
HTRIB3	CUPMTR	Backtransform eigenvectors of a complex Hermitian matrix after reduction by HTRID3
HTRIBK	CUNMTR	Backtransform eigenvectors of a complex Hermitian matrix after reduction by HTRIDI
HTRID3	CHPTRD	Reduce a complex Hermitian matrix to tridiagonal form (packed storage)
HTRIDI	CHETRD	Reduce a complex Hermitian matrix to tridiagonal form
IMTQL1	SSTEQR	All eigenvalues of a symmetric tridiagonal matrix, by the implicit QL algorithm
IMTQL2	SSTEQR	All eigenvalues/vectors of a symmetric tridiagonal matrix, by the implicit QL algorithm
IMTQLV	SSTEQR	As IMTQL1, preserving the input matrix
INVIT	SHSEIN	Selected eigenvectors of a real upper Hessenberg matrix, by inverse iteration
MINFIT	SGELSS	Minimum-norm solution of a linear least-squares problem, using the singular value decomposition
ORTBAK	SORMHR	Backtransform eigenvectors of a real matrix after reduction to upper Hessenberg form by ORTHES
ORTHES	SGEHRD	Reduce a real matrix to upper Hessenberg form by an orthogonal transformation
ORTRAN	SORGHR	Generate orthogonal transformation matrix used by ORTHES
QZHES		Reduce a real generalized eigenproblem $Ax = \lambda Bx$ to a form in which A is upper Hessenberg and B is upper triangular
QZIT		generalized Schur factorization of a real generalized eigenproblem, after reduction by QZHES
QZVAL		
QZVEC		all eigenvectors of a real generalized eigenproblem from generalized Schur factorization
RATQR	SSTEBZ (note 5)	Extreme eigenvalues of a symmetric tridiagonal matrix using the rational QR algorithm with Newton corrections
REBAK	STRSM	Backtransform eigenvectors of a symmetric-definite generalized eigenproblem $Ax = \lambda Bx$ or $ABx = \lambda x$ after reduction by REDUC or REDUC2
REBAKB	STRMM	Backtransform eigenvectors of a symmetric-definite generalized eigenproblem $BAx = \lambda x$ after reduction by REDUC2
REDUC	SSYGST	Reduce the symmetric-definite generalized eigenproblem $Ax = \lambda Bx$ to a standard symmetric eigenproblem

LAPACK equivalents of EISPACK routines (continued)		
EISPACK	LAPACK	Function of EISPACK routine
REDUC2	SSYGST	Reduce the symmetric-definite generalized eigenproblem $ABx = \lambda x$ or $BAx = \lambda x$ to a standard symmetric eigenproblem
RG	SGEEV	All eigenvalues and optionally eigenvectors of a real general matrix (driver routine)
RGG		All eigenvalues and optionally eigenvectors of a real generalized eigenproblem (driver routine)
RS	SSYEV	All eigenvalues and optionally eigenvectors of a real symmetric matrix (driver routine)
RSB	SSBEV	All eigenvalues and optionally eigenvectors of a real symmetric band matrix (driver routine)
RSG	SSYGV	All eigenvalues and optionally eigenvectors of a real symmetric-definite generalized eigenproblem $Ax = \lambda Bx$ (driver routine)
RSGAB	SSYGV	All eigenvalues and optionally eigenvectors of a real symmetric-definite generalized eigenproblem $ABx = \lambda x$ (driver routine)
RSGBA	SSYGV	All eigenvalues and optionally eigenvectors of a real symmetric-definite generalized eigenproblem $BAx = \lambda x$ (driver routine)
RSM	SSYEVS	Selected eigenvalues and optionally eigenvectors of a real symmetric matrix (driver routine)
RSP	SSPEV	All eigenvalues and optionally eigenvectors of a real symmetric matrix (packed storage) (driver routine)
RST	SSTEVS	All eigenvalues and optionally eigenvectors of a real symmetric tridiagonal matrix (driver routine)
RT		All eigenvalues and optionally eigenvectors of a real tridiagonal matrix of special form (driver routine)
SVD	SGESVD	Singular value decomposition of a real matrix
TINVIT	SSTEIN	Selected eigenvectors of a symmetric tridiagonal matrix by inverse iteration
TQL1	SSTEQR (note 5)	All eigenvalues of a symmetric tridiagonal matrix by the explicit QL algorithm
TQL2	SSTEQR (note 5)	All eigenvalues/vectors of a symmetric tridiagonal matrix by the explicit QL algorithm
TQLRAT	SSTERF	All eigenvalues of a symmetric tridiagonal matrix by a rational variant of the QL algorithm
TRBAK1	SORMTR	Backtransform eigenvectors of a real symmetric matrix after reduction by TRED1
TRBAK3	SCPMTR	Backtransform eigenvectors of a real symmetric matrix after reduction by TRED3 (packed storage)
TRED1	SSYTRD	Reduce a real symmetric matrix to tridiagonal form
TRED2	SSYTRD SORGTR	As TRED1, but also generating the orthogonal transformation matrix
TRED3	SSPTRD	Reduce a real symmetric matrix to tridiagonal form (packed storage)
TRIDIB	SSTEBZ	Eigenvalues between specified indices of a symmetric tridiagonal matrix

LAPACK equivalents of EISPACK routines (continued)		
EISPACK	LAPACK	Function of EISPACK routine
TSTORM	SSTEBZ SSTEEN	Eigenvalues in a specified interval of a symmetric tridiagonal matrix, and corresponding eigenvectors by inverse iteration

Appendix E

LAPACK Working Notes

Most of these working notes are available from *netlib*, where they can only be obtained in postscript form. To receive a list of available postscript reports, send email to `netlib@cornl.gov` of the form:

```
send index from lapack
```

1. J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, *Prospectus for the Development of a Linear Algebra Library for High-Performance Computers*, ANL, MCS-TM-97, September 1987.
2. J. Dongarra, S. Hammarling, and D. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, ANL, MCS-TM-99, September 1987.
3. J. Demmel and W. Kahan, *Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy*, ANL, MCS-TM-110, February 1988.
4. J. Demmel, J. Du Croz, S. Hammarling, and D. Sorensen, *Guidelines for the Design of Symmetric Eigenroutines, SVD, and Iterative Refinement and Condition Estimation for Linear Systems*, ANL, MCS-TM-111, March 1988.
5. C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, *Provisional Contents*, ANL, MCS-TM-38, September 1988.
6. O. Brewer, J. Dongarra, and D. Sorensen, *Tools to Aid in the Analysis of Memory Access Patterns for FORTRAN Programs*, ANL, MCS-TM-120, June 1988.
7. J. Barlow and J. Demmel, *Computing Accurate Eigensystems of Scaled Diagonally Dominant Matrices*, ANL, MCS-TM-126, December 1988.
8. Z. Bai and J. Demmel, *On a Block Implementation of Hessenberg Multishift QR Iteration*, ANL, MCS-TM-127, January 1989.
9. J. Demmel and A. McKenney, *A Test Matrix Generation Suite*, ANL, MCS-P69-0389, March 1989.

10. E. Anderson and J. Dongarra, *Installing and Testing the Initial Release of LAPACK - Unix and Non-Unix Versions*, ANL, MCS-TM-130, May 1989.
11. P. Deift, J. Demmel, L.-C. Li, and C. Tomei, *The Bidiagonal Singular Value Decomposition and Hamiltonian Mechanics*, ANL, MCS-TM-133, August 1989.
12. P. Mayes and G. Radicati, *Banded Cholesky Factorization Using Level 3 BLAS*, ANL, MCS-TM-134, August 1989.
13. Z. Bai, J. Demmel, and A. McKenney, *On the Conditioning of the Nonsymmetric Eigenproblem: Theory and Software*, UT, CS-89-86, October 1989.
14. J. Demmel, *On Floating Point Errors in Cholesky*, UT, CS-89-87, October 1989.
15. J. Demmel and K. Veselić, *Jacobi's Method is More Accurate than QR*, UT, CS-89-88, October 1989.
16. E. Anderson and J. Dongarra, *Results from the Initial Release of LAPACK*, UT, CS-89-89, November 1989.
17. A. Greenbaum and J. Dongarra, *Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem*, UT, CS-89-92, November 1989.
18. E. Anderson and J. Dongarra, *Implementation Guide for LAPACK*, UT, CS-90-101, April 1990.
19. E. Anderson and J. Dongarra, *Evaluating Block Algorithm Variants in LAPACK*, UT, CS-90-103, April 1990.
20. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK: A Portable Linear Algebra Library for High-Performance Computers*, UT, CS-90-105, May 1990.
21. J. Du Croz, P. Mayes, and G. Radicati, *Factorizations of Band Matrices Using Level 3 BLAS*, UT, CS-90-109, July 1990.
22. J. Demmel and N. Higham, *Stability of Block Algorithms with Fast Level 3 BLAS*, UT, CS-90-110, July 1990.
23. J. Demmel and N. Higham, *Improved Error Bounds for Underdetermined System Solvers*, UT, CS-90-113, August 1990.
24. J. Dongarra and S. Ostrouchov, *LAPACK Block Factorization Algorithms on the Intel iPSC/860*, UT, CS-90-115, October, 1990.
25. J. Dongarra, S. Hammarling, and J. Wilkinson, *Numerical Considerations in Computing Invariant Subspaces*, UT, CS-90-117, October, 1990.
26. E. Anderson, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, S. Hammarling, and W. Kahan, *Prospectus for an Extension to LAPACK: A Portable Linear Algebra Library for High-Performance Computers*, UT, CS-90-118, November 1990.

27. J. DuCroz, and N. Higham, *Stability of Methods for Matrix Inversion*, UT, CS-90-119, October, 1990.
28. J. Dongarra, P. Mayes, and G. Radicati, *The IBM RISC System/6000 and Linear Algebra Operations*, UT, CS-90-122, December 1990.
29. R. van de Geijn, *On Global Combine Operations*, UT, CS-91-129, April 1991.
30. J. Dongarra, R. van de Geijn, *Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures*, UT, CS-91-130, April 1991.
31. E. Anderson, Z. Bai, and J. Dongarra, *Generalized QR Factorization and its Applications*, UT, CS-91-131, April 1991.
32. C. Bischof, and P. Tang, *Generalized Incremental Condition Estimation*, UT, CS-91-132, May 1991.
33. C. Bischof, and P. Tang, *Robust Incremental Condition Estimation*, UT, CS-91-133, May 1991.
34. J. Dongarra, *Workshop on the BLACS*, UT, CS-91-134, May 1991.
35. E. Anderson, J. Dongarra, and S. Ostrouchov, *Implementation guide for LAPACK*, UT, CS-91-138, August 1991.
36. E. Anderson, *Robust Triangular Solves for Use in Condition Estimation*, UT, CS-91-142, August 1991.
37. J. Dongarra and R. van de Geijn, *Two Dimensional Basic Linear Algebra Communication Subprograms*, UT, CS-91-138, October 1991.
38. Z. Bai and J. Demmel, *On a direct algorithm for computing invariant subspaces with specified eigenvalues*, UT, CS-91-139, November 1991.
39. J. Demmel, J. Dongarra, and W. Kahan, *On Designing Portable High Performance Numerical Libraries* UT, CS-91-141, July 1991.

Appendix F

Specifications of Routines

Notes

1. The specifications which follow, give the calling sequence, purpose, and descriptions of the arguments, of each LAPACK driver and computational routine (but not of auxiliary routines).
2. Specifications of pairs of real and complex routines have been merged (for example SBD-SQR/CBDSQR). In a few cases, specifications of three routines have been merged, one for real symmetric, one for complex symmetric, and one for complex Hermitian matrices (for example SSYTRF/CSYTRF/CHETRF). A few routines for real matrices have no complex equivalent (for example SSTEGBZ).
3. Specifications are given only for *single precision* routines. To adapt them for the double precision version of the software, simply interpret REAL as DOUBLE PRECISION, COMPLEX as COMPLEX*16 (or DOUBLE COMPLEX), and the initial letters S- and C- of LAPACK routine names as D- and Z-.
4. Specifications are arranged in alphabetical order of the real routine name.
5. The text of the specifications has been derived from the leading comments in the source-text of the routines. It makes only a limited use of mathematical typesetting facilities. To eliminate redundancy, A^H has been used throughout the specifications. Thus, the reader should note that A^H is equivalent to A^T in the real case.

END

**DATE
FILMED**

5 / 15 / 92

