LAPACK Working Note 81
Quick Installation Guide for LAPACK on Unix Systems[1]

Authors and Jack Dongarra
Department of Computer Science
University of Tennessee
Knoxville, Tennessee 37996-1301

REVISED: VERSION 3.1.1, February 2007

Abstract

This working note describes how to install, and test version 3.1.1 of LAPACK, a linear algebra package for high-performance computers, on a Unix System. The timing routines are not actually included in release 3.1.1, and that part of the LAWN refers to release 3.0. Non-Unix installation instructions and further details of the testing and timing suites are only contained in LAPACK Working Note 41, and not in this abbreviated version.

[1]This work was supported by NSF Grant No. ASC-8715728 and NSF Grant No. 0444486

Contents

1 Introduction

LAPACK is a linear algebra library for high-performance computers. The library includes Fortran 77 subroutines for the analysis and solution of systems of simultaneous linear algebraic equations, linear least-squares problems, and matrix eigenvalue problems. Our approach to achieving high efficiency is based on the use of a standard set of Basic Linear Algebra Subprograms (the BLAS), which can be optimized for each computing environment. By confining most of the computational work to the BLAS, the subroutines should be transportable and efficient across a wide range of computers.

This working note describes how to install, test, and time this release of LAPACK on a Unix System.

The instructions for installing, testing, and timing [2] are designed for a person whose responsibility is the maintenance of a mathematical software library. We assume the installer has experience in compiling and running Fortran programs and in creating object libraries. The installation process involves untarring the file, creating a set of libraries, and compiling and running the test and timing programs [2].

Section 3 describes how the files are organized in the file, and Section 4 gives a general overview of the parts of the test package. Step-by-step instructions appear in Section 5.

For users desiring additional information, please refer to LAPACK Working Note 41. Appendix A, entitled "Caveats", is a compendium of the known problems from our own experiences, with suggestions on how to overcome them.

It is strongly advised that the user read Appendix A before proceeding with the installation process.

2 Revisions Since the First Public Release

Since its first public release in February, 1992, LAPACK has had several updates, which have encompassed the introduction of new routines as well as extending the functionality of existing routines. The first update, June 30, 1992, was version 1.0a; the second update, October 31, 1992, was version 1.0b; the third update, March 31, 1993, was version 1.1; version 2.0 on September 30, 1994, coincided with the release of the Second Edition of the LAPACK Users' Guide; version 3.0 on June 30, 1999 coincided with the release of the Third Edition of the LAPACK Users' Guide; version 3.1 was released on November, 2006. and finally version 3.1.1 was released on November, 2007.

All LAPACK routines reflect the current version number with the date on the routine indicating when it was last modified. For more information on revisions in the latest release, please refer to the `revisions.info` file in the lapack directory on netlib.

`http://www.netlib.org/lapack/revisions.info`

3 File Format

The software for LAPACK is distributed in the form of a gzipped tar file (via anonymous ftp or the World Wide Web), which contains the Fortran source for LAPACK, the Basic

[2] timing are only provided in LAPACK 3.0 and before

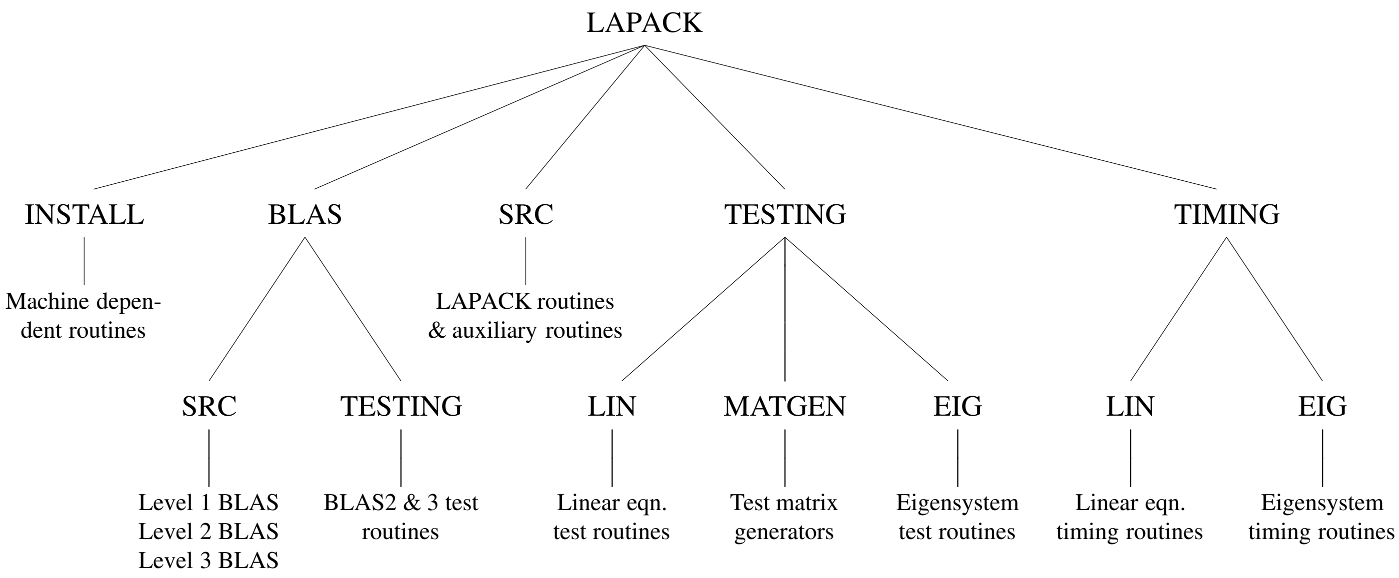

Figure 1: Unix organization of LAPACK 3.0

Linear Algebra Subprograms (the Level 1, 2, and 3 BLAS) needed by LAPACK, the testing programs, and the timing programs[2]. Users who wish to have a non-Unix installation should refer to LAPACK Working Note 41, although the overview in section 4 applies to both the Unix and non-Unix versions.

The package may be accessed via the World Wide Web through the URL address:

```
http://www.netlib.org/lapack/lapack.tgz
```

Or, you can retrieve the file via anonymous ftp at netlib:

```
ftp ftp.netlib.org
login:  anonymous
password:  <your email address>
cd lapack
binary
get lapack.tgz
quit
```

The software in the `tar` file is organized in a number of essential directories as shown in Figure 1. Please note that this figure does not reflect everything that is contained in the `LAPACK` directory. Input and instructional files are also located at various levels. Libraries are created in the LAPACK directory and executable files are created in one of the directories BLAS, TESTING, or TIMING[2]. Input files for the test and timing[2] programs are also found in these three directories so that testing may be carried out in the directories LAPACK/BLAS, LAPACK/TESTING, and LAPACK/TIMING [2]. A top-level makefile in the LAPACK directory is provided to perform the entire installation procedure.

4 Overview of Tape Contents

Most routines in LAPACK occur in four versions: REAL, DOUBLE PRECISION, COMPLEX, and COMPLEX*16. The first three versions (REAL, DOUBLE PRECISION, and COMPLEX) are written in standard Fortran 77 and are completely portable; the COMPLEX*16 version is provided for those compilers which allow this data type. For convenience, we often refer to routines by their single precision names; the leading 'S' can be replaced by a 'D' for double precision, a 'C' for complex, or a 'Z' for complex*16. For LAPACK use and testing you must decide which version(s) of the package you intend to install at your site (for example, REAL and COMPLEX on a Cray computer or DOUBLE PRECISION and COMPLEX*16 on an IBM computer).

4.1 LAPACK Routines

There are three classes of LAPACK routines:

- **driver** routines solve a complete problem, such as solving a system of linear equations or computing the eigenvalues of a real symmetric matrix. Users are encouraged to use a driver routine if there is one that meets their requirements. The driver routines are listed in LAPACK Working Note 41 [3] and the LAPACK Users' Guide [1].

- **computational** routines, also called simply LAPACK routines, perform a distinct computational task, such as computing the LU decomposition of an m-by-n matrix or finding the eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the QR algorithm. The LAPACK routines are listed in LAPACK Working Note 41 [3] and the LAPACK Users' Guide [1].

- **auxiliary** routines are all the other subroutines called by the driver routines and computational routines. The auxiliary routines are listed in LAPACK Working Note 41 [3] and the LAPACK Users' Guide [1].

4.2 Level 1, 2, and 3 BLAS

The BLAS are a set of Basic Linear Algebra Subprograms that perform vector-vector, matrix-vector, and matrix-matrix operations. LAPACK is designed around the Level 1, 2, and 3 BLAS, and nearly all of the parallelism in the LAPACK routines is contained in the BLAS. Therefore, the key to getting good performance from LAPACK lies in having an efficient version of the BLAS optimized for your particular machine. Optimized BLAS libraries are available on a variety of architectures, refer to the BLAS FAQ on netlib for further information.

`http://www.netlib.org/blas/faq.html`

There are also freely available BLAS generators that automatically tune a subset of the BLAS for a given architecture. E.g.,

`http://www.netlib.org/atlas/`

And, if all else fails, there is the Fortran 77 reference implementation of the Level 1, 2, and 3 BLAS available on netlib (also included in the LAPACK distribution tar file).

`http://www.netlib.org/blas/blas.tgz`

No matter which BLAS library is used, the BLAS test programs should always be run.

Users should not expect too much from the Fortran 77 reference implementation BLAS; these versions were written to define the basic operations and do not employ the standard tricks for optimizing Fortran code.

The formal definitions of the Level 1, 2, and 3 BLAS are in [10], [8], and [6]. The BLAS Quick Reference card is available on netlib.

4.3 LAPACK Test Routines

This release contains two distinct test programs for LAPACK routines in each data type. One test program tests the routines for solving linear equations and linear least squares problems, and the other tests routines for the matrix eigenvalue problem. The routines for generating test matrices are used by both test programs and are compiled into a library for use by both test programs.

4.4 LAPACK Timing Routines (for LAPACK 3.0 and before)

This release also contains two distinct timing programs for the LAPACK routines in each data type. The linear equation timing program gathers performance data in megaflops on the factor, solve, and inverse routines for solving linear systems, the routines to generate or apply an orthogonal matrix given as a sequence of elementary transformations, and the reductions to bidiagonal, tridiagonal, or Hessenberg form for eigenvalue computations. The operation counts used in computing the megaflop rates are computed from a formula; see LAPACK Working Note 41 [3]. The eigenvalue timing program is used with the eigensystem routines and returns the execution time, number of floating point operations, and megaflop rate for each of the requested subroutines. In this program, the number of operations is computed while the code is executing using special instrumented versions of the LAPACK subroutines.

5 Installing LAPACK on a Unix System

Installing, testing, and timing[2] the Unix version of LAPACK involves the following steps:

1. Gunzip and tar the file.
2. Copy and edit the file `LAPACK/make.inc.example to LAPACK/make.inc`.
3. Edit the file `LAPACK/Makefile` and type `make`.

5.1 Untar the File

If you received a tar file of LAPACK via the World Wide Web or anonymous ftp, enter the following command:

```
gunzip -c lapack.tgz | tar xvf -
```

This will create a top-level directory called `LAPACK`, which requires approximately 34 Mbytes of disk space. The total space requirements including the object files and executables is approximately 100 Mbytes for all four data types.

5.2 Copy and edit the file `LAPACK/make.inc.example to LAPACK/make.inc`

Before the libraries can be built, or the testing and timing[2] programs run, you must define all machine-specific parameters for the architecture to which you are installing LAPACK. All machine-specific parameters are contained in the file `LAPACK/make.inc`. An example of `LAPACK/make.inc` for a LINUX machine with GNU compilers is given in `LAPACK/make.inc.example`, copy that file to LAPACK/make.inc by entering the following command:

```
cp LAPACK/make.inc.example LAPACK/make.inc
```

Now modify your `LAPACK/make.inc` by applying the following recommendations. The first line of this `make.inc` file is:

SHELL = /bin/sh

and it will need to be modified to `SHELL = /sbin/sh` if you are installing LAPACK on an SGI architecture. Second, you will need to modify the `PLAT` definition, which is appended to all library names, to specify the architecture to which you are installing LAPACK. This features avoids confusion in library names when you are installing LAPACK on more than one architecture. Next, you will need to modify `FORTRAN`, `OPTS`, `DRVOPTS`, `NOOPT`, `LOADER`, and `LOADOPTS` to specify the compiler, compiler options, compiler options for the testing and timing[2] main programs, loader, loader options. Next you will have to choose which function you will use to time in the `SECOND` and `DSECND` routines.

```
#The Default : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME
TIMER    = EXT_ETIME
# For RS6K : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME_
# TIMER    = EXT_ETIME_
# For gfortran compiler: SECOND and DSECND will use the INTERNAL FUNCTION ETIME
# TIMER    = INT_ETIME
# If your Fortran compiler does not provide etime (like Nag Fortran Compiler, etc...)
# SECOND and DSECND will use a call to the INTERNAL FUNCTION CPU_TIME
# TIMER    = INT_CPU_TIME
# If neither of this works...you can use the NONE value...
# In that case, SECOND and DSECND will always return 0
# TIMER     = NONE
```

Refer to the section 6.1.3 to get more information.

Next, you will need to modify `ARCH`, `ARCHFLAGS`, and `RANLIB` to specify archiver, archiver options, and ranlib for your machine. If your architecture does not require `ranlib` to be run after each archive command (as is the case with CRAY computers running UNICOS, Hewlett Packard computers running HP-UX, or SUN SPARCstations running Solaris), set `ranlib=echo`. And finally, you must modify the `BLASLIB` definition to specify the BLAS library to which you will be linking. If an optimized version of the BLAS is available on your machine, you are highly recommended to link to that library. Otherwise, by default, `BLASLIB` is set to the Fortran 77 version.

NOTE: Example `make.inc` include files are contained in the `LAPACK/INSTALL` directory. Please refer to Appendix A for machine-specific installation hints, and/or the `release_notes` file on `netlib`.

`http://www.netlib.org/lapack/release_notes`

5.3 Edit the file LAPACK/Makefile

This `Makefile` can be modified to perform as much of the installation process as the user desires. Ideally, this is the ONLY makefile the user must modify. However, modification of lower-level makefiles may be necessary if a specific routine needs to be compiled with a different level of optimization.

First, edit the definitions of `blaslib`, `lapacklib`, `tmglib`, `lapack_testing`, and `timing`[2] in the file `LAPACK/Makefile` to specify the data types desired. For example, if you only wish to compile the single precision real version of the LAPACK library, you would modify the `lapacklib` definition to be:

```
lapacklib:
        ( cd SRC; $(MAKE) single )
```

Likewise, you could specify `double, complex, or complex16` to build the double precision real, single precision complex, or double precision complex libraries, respectively. By default, the presence of no arguments following the `make` command will result in the building of all four data types. The make command can be run more than once to add another data type to the library if necessary.

Next, if you will be using a locally available BLAS library, you will need to remove `blaslib` from the `lib` definition. And finally, if you do not wish to build all of the libraries individually and likewise run all of the testing and timing separately, you can modify the `all` definition to specify the amount of the installation process that you want performed. By default, the `all` definition is set to

```
all: lapack_install lib lapack_testing blas_testing
```

which will perform all phases of the installation process – testing of machine-dependent routines, building the libraries, BLAS testing and LAPACK testing.

The entire installation process will then be performed by typing `make`.

Questions and/or comments can be directed to the authors as described in Section 6.8. If test failures occur, please refer to the appropriate subsection in Section 6.

If disk space is limited, we suggest building each data type separately and/or deleting all object files after building the libraries. Likewise, all testing and timing executables can be deleted after the testing and timing process is completed. The removal of all object files and executables can be accomplished by the following:

```
cd LAPACK

make clean
```

6 Further Details of the Installation Process

Alternatively, you can choose to run each of the phases of the installation process separately. The following sections give details on how this may be achieved.

6.1 Test and Install the Machine-Dependent Routines.

There are six machine-dependent functions in the test and timing package, at least three of which must be installed. They are

LSAME	LOGICAL	Test if two characters are the same regardless of case
SLAMCH	REAL	Determine machine-dependent parameters
DLAMCH	DOUBLE PRECISION	Determine machine-dependent parameters
SECOND	REAL	Return time in seconds from a fixed starting time
DSECND	DOUBLE PRECISION	Return time in seconds from a fixed starting time
ILAENV	INTEGER	Checks that NaN and infinity arithmetic are IEEE-754 compliant

If you are working only in single precision, you do not need to install DLAMCH and DSECND, and if you are working only in double precision, you do not need to install SLAMCH and SECOND.

These six subroutines are provided in `LAPACK/INSTALL`, along with six test programs. To compile the six test programs and run the tests, go to `LAPACK` and type `make lapack_install`. The test programs are called `testlsame, testslamch, testdlamch, testsecond, testdsecnd` and `testieee`. If you do not wish to run all tests, you will need to modify the `lapack_install` definition in the `LAPACK/Makefile` to only include the tests you wish to run. Otherwise, all tests will be performed. The expected results of each test program are described below.

6.1.1 Installing LSAME

LSAME is a logical function with two character parameters, A and B. It returns .TRUE. if A and B are the same regardless of case, or .FALSE. if they are different. For example, the expression

```
LSAME( UPLO, 'U' )
```

is equivalent to

```
( UPLO.EQ.'U' ).OR.( UPLO.EQ.'u' )
```

The test program in `lsametst.f` tests all combinations of the same character in upper and lower case for A and B, and two cases where A and B are different characters.

Run the test program by typing `testlsame`. If LSAME works correctly, the only message you should see after the execution of `testlsame` is

```
ASCII character set
Tests completed
```

The file `lsame.f` is automatically copied to `LAPACK/BLAS/SRC/` and `LAPACK/SRC/`. The function LSAME is needed by both the BLAS and LAPACK, so it is safer to have it in both libraries as long as this does not cause trouble in the link phase when both libraries are used.

6.1.2 Installing SLAMCH and DLAMCH

SLAMCH and DLAMCH are real functions with a single character parameter that indicates the machine parameter to be returned. The test program in `slamchtst.f` simply prints out the different values computed by SLAMCH, so you need to know something about what the values should be. For example, the output of the test program executable `testslamch` for SLAMCH on a Sun SPARCstation is

```
Epsilon                      =     5.96046E-08
Safe minimum                 =     1.17549E-38
Base                         =     2.00000
Precision                    =     1.19209E-07
Number of digits in mantissa =     24.0000
Rounding mode                =     1.00000
Minimum exponent             =    -125.000
Underflow threshold          =     1.17549E-38
Largest exponent             =     128.000
Overflow threshold           =     3.40282E+38
Reciprocal of safe minimum   =     8.50706E+37
```

On a Cray machine, the safe minimum underflows its output representation and the overflow threshold overflows its output representation, so the safe minimum is printed as 0.00000 and overflow is printed as R. This is normal. If you would prefer to print a representable number, you can modify the test program to print SFMIN*100. and RMAX/100. for the safe minimum and overflow thresholds.

Likewise, the test executable `testdlamch` is run for DLAMCH.

If both tests were successful, go to Section 6.1.3.

If SLAMCH (or DLAMCH) returns an invalid value, you will have to create your own version of this function. The following options are used in LAPACK and must be set:

'B': Base of the machine

'E': Epsilon (relative machine precision)

'O': Overflow threshold

'P': Precision = Epsilon*Base

'S': Safe minimum (often same as underflow threshold)

'U': Underflow threshold

Some people may be familiar with R1MACH (D1MACH), a primitive routine for setting machine parameters in which the user must comment out the appropriate assignment statements for the target machine. If a version of R1MACH is on hand, the assignments in SLAMCH can be made to refer to R1MACH using the correspondence

SLAMCH('U') = R1MACH(1)

SLAMCH('O') = R1MACH(2)

SLAMCH('E') = R1MACH(3)

SLAMCH('B') = R1MACH(5)

The safe minimum returned by SLAMCH('S') is initially set to the underflow value, but if 1/(overflow) $\geq$ (underflow) it is recomputed as $(1/(\text{overflow})) * (1 + \varepsilon)$, where ε is the machine precision.

BE AWARE that the initial call to SLAMCH or DLAMCH is expensive. We suggest that installers run it once, save the results, and hard-code the constants in the version they put in their library.

6.1.3 Installing SECOND and DSECND

Both the timing routines[2] and the test routines call SECOND (DSECND), a real function with no arguments that returns the time in seconds from some fixed starting time. Our version of this routine returns only "user time", and not "user time + system time". The following version of SECOND in `second_EXT_ETIME.f, second_INT_ETIME.f` calls ETIME, a Fortran library routine available on some computer systems. If ETIME is not available or a better local timing function exists, you will have to provide the correct interface to SECOND and DSECND on your machine.

Since LAPACK 3.1.1 we provide 5 different flavours of the SECOND and DSECND routines. The version that will be used depends on the value of the TIMER variable in the make.inc

- If ETIME is available as an external function, set the value of the TIMER variable in your make.inc to `EXT_ETIME`:`second_EXT_ETIME.f` and `dsecnd_EXT_ETIME.f` will be used. Usually on HPPA architectures, the compiler and loader flag `+U77` should be included to access the function `ETIME`.

- If ETIME_ is available as an external function, set the value of the TIMER variable in your make.inc to `EXT_ETIME_`:`second_EXT_ETIME_.f` and `dsecnd_EXT_ETIME_.f` will be used. It is the case on some IBM architectures such as IBM RS/6000s.

- If ETIME is available as an internal function, set the value of the TIMER variable in your make.inc to `INT_ETIME:second_INT_ETIME.f` and `dsecnd_INT_ETIME.f` will be used. This is the case with gfortan.

- If CPU_TIME is available as an internal function, set the value of the TIMER variable in your make.inc to `INT_CPU_TIME:second_INT_CPU_TIME.f` and `dsecnd_INT_CPU_TIME.f` will be used.

- If none of these function is available, set the value of the TIMER variable in your make.inc to `NONE:second_NONE.f` and `dsecnd_NONE.f` will be used. These routines will always return zero.

The test program in `secondtst.f` performs a million operations using 5000 iterations of the SAXPY operation $y := y + \alpha x$ on a vector of length 100. The total time and megaflops for this test is reported, then the operation is repeated including a call to SECOND on each of the 5000 iterations to determine the overhead due to calling SECOND. The test program executable is called `testsecond` (or `testdsecnd`). There is no single right answer, but the times in seconds should be positive and the megaflop ratios should be appropriate for your machine.

6.1.4 Testing IEEE arithmetic and ILAENV

As some new routines in LAPACK rely on IEEE-754 compliance, two settings (`ISPEC=10` and `ISPEC=11`) have been added to ILAENV (`LAPACK/SRC/ilaenv.f`) to denote IEEE-754 compliance for NaN and infinity arithmetic, respectively. By default, ILAENV assumes an IEEE machine, and does a test for IEEE-754 compliance. **NOTE: If you are installing LAPACK on a non-IEEE machine, you MUST modify ILAENV, as this test inside ILAENV will crash!**

If `ILAENV( 10, ... )` or `ILAENV( 11, ... )` is issued, then `ILAENV=1` is returned to signal IEEE-754 compliance, and `ILAENV=0` if the architecture is non-IEEE-754 compliant.

Thus, for non-IEEE machines, the user must hard-code the setting of (`ILAENV=0`) for (`ISPEC=10` and `ISPEC=11`) in the version of `LAPACK/SRC/ilaenv.f` to be put in his library. There are also specialized testing and timing[2] versions of ILAENV that will also need to be modified.

- Testing/timing version of `LAPACK/TESTING/LIN/ilaenv.f`

- Testing/timing version of `LAPACK/TESTING/EIG/ilaenv.f`

- Testing/timing version of `LAPACK/TIMING/LIN/ilaenv.f`

- Testing/timing version of `LAPACK/TIMING/EIG/ilaenv.f`

The test program in `LAPACK/INSTALL/tstiee.f` checks an installation architecture to see if infinity arithmetic and NaN arithmetic are IEEE-754 compliant. A warning message to the user is printed if non-compliance is detected. This same test is performed inside the function ILAENV. If `ILAENV( 10, ... )` or `ILAENV( 11, ... )` is issued, then `ILAENV=1` is returned to signal IEEE-754 compliance, and `ILAENV=0` if the architecture is non-IEEE-754 compliant.

To avoid this IEEE test being run every time you call `ILAENV( 10, ...)` or `ILAENV( 11, ... )`, we suggest that the user hard-code the setting of `ILAENV=1` or `ILAENV=0` in the version of `LAPACK/SRC/ilaenv.f` to be put in his library. As aforementioned, there are also specialized testing and timing[2] versions of ILAENV that will also need to be modified.

6.2 Create the BLAS Library

Ideally, a highly optimized version of the BLAS library already exists on your machine. In this case you can go directly to Section 6.3 to make the BLAS test programs.

a) Go to `LAPACK` and edit the definition of `blaslib` in the file `Makefile` to specify the data types desired, as in the example in Section 5.3.

   If you already have some of the BLAS, you will need to edit the file `LAPACK/BLAS/SRC/Makefile` to comment out the lines defining the BLAS you have.

b) Type `make blaslib`. The make command can be run more than once to add another data type to the library if necessary.

The BLAS library is created in `LAPACK/blas_PLAT.a`, where `PLAT` is the user-defined architecture suffix specified in the file `LAPACK/make.inc`.

6.3 Run the BLAS Test Programs

Test programs for the Level 1, 2, and 3 BLAS are in the directory `LAPACK/BLAS/TESTING`.

To compile and run the Level 1, 2, and 3 BLAS test programs, go to `LAPACK` and type `make blas_testing`. The executable files are called `xblat_s`, `xblat_d`, `xblat_c`, and `xblat_z`, where the `_` (underscore) is replaced by 1, 2, or 3, depending upon the level of BLAS that it is testing. All executable and output files are created in `LAPACK/BLAS/`. For the Level 1 BLAS tests, the output file names are `sblat1.out`, `dblat1.out`, `cblat1.out`, and `zblat1.out`. For the Level 2 and 3 BLAS, the name of the output file is indicated on the first line of the input file and is currently defined to be `sblat2.out` for the Level 2 REAL version, and `sblat3.out` for the Level 3 REAL version, with similar names for the other data types.

If the tests using the supplied data files were completed successfully, consider whether the tests were sufficiently thorough. For example, on a machine with vector registers, at least one value of N greater than the length of the vector registers should be used; otherwise, important parts of the compiled code may not be exercised by the tests. If the tests were not successful, either because the program did not finish or the test ratios did not pass the threshold, you will probably have to find and correct the problem before continuing. If you have been testing a system-specific BLAS library, try using the Fortran BLAS for the routines that did not pass the tests. For more details on the BLAS test programs, see [9] and [7].

6.4 Create the LAPACK Library

a) Go to the directory `LAPACK` and edit the definition of `lapacklib` in the file `Makefile` to specify the data types desired, as in the example in Section 5.3.

b) Type `make lapacklib`. The make command can be run more than once to add another data type to the library if necessary.

The LAPACK library is created in `LAPACK/lapack_PLAT.a`, where `PLAT` is the user-defined architecture suffix specified in the file `LAPACK/make.inc`.

6.5 Create the Test Matrix Generator Library

a) Go to the directory `LAPACK` and edit the definition of `tmglib` in the file `Makefile` to specify the data types desired, as in the example in Section 5.3.

b) Type `make tmglib`. The make command can be run more than once to add another data type to the library if necessary.

The test matrix generator library is created in `LAPACK/tmglib_PLAT.a`, where `PLAT` is the user-defined architecture suffix specified in the file `LAPACK/make.inc`.

6.6 Run the LAPACK Test Programs

There are two distinct test programs for LAPACK routines in each data type, one for the linear equation routines and one for the eigensystem routines. In each data type, there is one input file for testing the linear equation routines and eighteen input files for testing the eigenvalue routines. The input files reside in `LAPACK/TESTING`. For more information on the test programs and how to modify the input files, please refer to LAPACK Working Note 41 [3].

If you do not wish to run each of the tests individually, you can go to `LAPACK`, edit the definition `lapack_testing` in the file `Makefile` to specify the data types desired, and type `make lapack_testing`. This will compile and run the tests as described in sections 6.6.1 and 6.6.2.

6.6.1 Testing the Linear Equations Routines

a) Go to `LAPACK/TESTING/LIN` and type `make` followed by the data types desired. The executable files are called `xlintsts, xlintstc, xlintstd`, or `xlintstz` and are created in `LAPACK/TESTING`.

b) Go to `LAPACK/TESTING` and run the tests for each data type. For the REAL version, the command is

```
xlintsts < stest.in > stest.out
```

The tests using `xlintstd`, `xlintstc`, and `xlintstz` are similar with the leading 's' in the input and output file names replaced by 'd', 'c', or 'z'.

If you encountered failures in this phase of the testing process, please refer to Section 6.8.

6.6.2 Testing the Eigensystem Routines

a) Go to `LAPACK/TESTING/EIG` and type `make` followed by the data types desired. The executable files are called **`xeigtsts, xeigtstc, xeigtstd`**, and **`xeigtstz`** and are created in `LAPACK/TESTING`.

b) Go to `LAPACK/TESTING` and run the tests for each data type. The tests for the eigensystem routines use eighteen separate input files for testing the nonsymmetric eigenvalue problem, the symmetric eigenvalue problem, the banded symmetric eigenvalue problem, the generalized symmetric eigenvalue problem, the generalized nonsymmetric eigenvalue problem, the singular value decomposition, the banded singular value decomposition, the generalized singular value decomposition, the generalized QR and RQ factorizations, the generalized linear regression model, and the constrained linear least squares problem. The tests for the REAL version are as follows:

```
xeigtsts < nep.in > snep.out
xeigtsts < sep.in > ssep.out
xeigtsts < svd.in > ssvd.out
xeigtsts < sec.in > sec.out
xeigtsts < sed.in > sed.out
xeigtsts < sgg.in > sgg.out
xeigtsts < sgd.in > sgd.out
xeigtsts < ssg.in > ssg.out
xeigtsts < ssb.in > ssb.out
xeigtsts < sbb.in > sbb.out
xeigtsts < sbal.in > sbal.out
xeigtsts < sbak.in > sbak.out
xeigtsts < sgbal.in > sgbal.out
xeigtsts < sgbak.in > sgbak.out
xeigtsts < glm.in > sglm.out
xeigtsts < gqr.in > sgqr.out
xeigtsts < gsv.in > sgsv.out
xeigtsts < lse.in > slse.out
```

The tests using `xeigtstc`, `xeigtstd`, and `xeigtstz` also use the input files `nep.in`, `sep.in`, `svd.in`, `glm.in`, `gqr.in`, `gsv.in`, and `lse.in`, but the leading 's' in the other input file names must be changed to 'c', 'd', or 'z'.

If you encountered failures in this phase of the testing process, please refer to Section 6.8.

6.7 Run the LAPACK Timing Programs (For LAPACK 3.0 and before)

There are two distinct timing programs for LAPACK routines in each data type, one for the linear equation routines and one for the eigensystem routines. The timing program for the linear equation routines is also used to time the BLAS. We encourage you to conduct these timing experiments in REAL and COMPLEX or in DOUBLE PRECISION and COMPLEX*16; it is not necessary to send timing results in all four data types.

Two sets of input files are provided, a small set and a large set. The small data sets are appropriate for a standard workstation or other non-vector machine. The large data sets are appropriate for supercomputers, vector computers, and high-performance workstations. We are mainly interested in results from the large data sets, and it is not necessary to run both the large and small sets. The values of N in the large data sets are about five times larger than those in the small data set, and the large data sets use additional values for parameters such as the block size NB and the leading array dimension LDA. Small data sets finished with the _small in their name , such as `stime_small.in`, and large data sets finished with _large in their name, such as `stime_large.in`. Except as noted, the leading 's' in the input file name must be replaced by 'd', 'c', or 'z' for the other data types.

We encourage you to obtain timing results with the large data sets, as this allows us to compare different machines. If this would take too much time, suggestions for paring back the large data sets are given in the instructions below. We also encourage you to experiment with these timing programs and send us any interesting results, such as results for larger problems or for a wider range of block sizes. The main programs are dimensioned for the large data sets, so the parameters in the main program may have to be reduced in order to run the small data sets on a small machine, or increased to run experiments with larger problems.

The minimum time each subroutine will be timed is set to 0.0 in the large data files and to 0.05 in the small data files, and on many machines this value should be increased. If the timing interval is not long enough, the time for the subroutine after subtracting the overhead may be very small or zero, resulting in megaflop rates that are very large or zero. (To avoid division by zero, the megaflop rate is set to zero if the time is less than or equal to zero.) The minimum time that should be used depends on the machine and the resolution of the clock.

For more information on the timing programs and how to modify the input files, please refer to LAPACK Working Note 41 [3].

If you do not wish to run each of the timings individually, you can go to `LAPACK`, edit the definition `lapack_timing` in the file `Makefile` to specify the data types desired, and type `make lapack_timing`. This will compile and run the timings for the linear equation routines and the eigensystem routines (see Sections 6.7.1 and 6.7.3).

If you encounter failures in any phase of the timing process, please feel free to contact the authors as directed in Section 6.8. Tell us the type of machine on which the tests were run, the version of the operating system, the compiler and compiler options that were used, and details of the BLAS library or libraries that you used. You should also include a copy of the output file in which the failure occurs.

Please note that the BLAS timing runs will still need to be run as instructed in 6.7.2.

6.7.1 Timing the Linear Equations Routines

The linear equation timing program is found in `LAPACK/TIMING/LIN` and the input files are in `LAPACK/TIMING`. Three input files are provided in each data type for timing the linear equation routines, one for square matrices, one for band matrices, and one for rectangular matrices. The small data sets for the REAL version are `stime_small.in`, `sband_small.in`, and `stime2_small.in`, respectively, and the large data sets are `stime_large.in`, `sband_large.in`, and `stime2_large.in`.

The timing program for the least squares routines uses special instrumented versions of the LAPACK routines to time individual sections of the code. The first step in compiling the timing program is therefore to make a library of the instrumented routines.

a) To make a library of the instrumented LAPACK routines, first go to `LAPACK/TIMING/LIN/LINSRC` and type `make` followed by the data types desired, as in the examples of Section 5.3. The library of instrumented code is created in `LAPACK/TIMING/LIN/linsrc_PLAT.a`, where `PLAT` is the user-defined architecture suffix specified in the file `LAPACK/make.inc`.

b) To make the linear equation timing programs, go to `LAPACK/TIMING/LIN` and type `make` followed by the data types desired, as in the examples in Section 5.3. The executable files are called `xlintims`, `xlintimc`, `xlintimd`, and `xlintimz` and are created in `LAPACK/TIMING`.

c) Go to `LAPACK/TIMING` and make any necessary modifications to the input files. You may need to set the minimum time a subroutine will be timed to a positive value, or to restrict the size of the tests if you are using a computer with performance in between that of a workstation and that of a supercomputer. The computational requirements can be cut in half by using only one value of LDA. If it is necessary to also reduce the matrix sizes or the values of the blocksize, corresponding changes should be made to the BLAS input files (see Section 6.7.2).

d) Run the programs for each data type you are using. For the REAL version, the commands for the small data sets are

```
xlintims < stime_small.in > stime_small.out
xlintims < sband_small.in > sband_small.out
xlintims < stime2_small.in > stime2_small.out
```

or the commands for the large data sets are

```
xlintims < stime_large.in > stime_large.out
xlintims < sband_large.in > sband_large.out
xlintims < stime2_large.in > stime2_large.out
```

Similar commands should be used for the other data types.

6.7.2 Timing the BLAS

The linear equation timing program is also used to time the BLAS. Three input files are provided in each data type for timing the Level 2 and 3 BLAS. These input files time the BLAS using the matrix shapes encountered in the LAPACK routines, and we will use the results to analyze the performance of the LAPACK routines. For the REAL version, the small data files are `sblasa_small.in`, `sblasb_small.in`, and `sblasc_small.in` and the large data files are `sblasa_large.in`, `sblasb_large.in`, and `sblasc_large.in`. There are three sets of inputs because there are three parameters in the Level 3 BLAS, M, N, and K, and in most applications one of these parameters is small (on the order of the blocksize) while the other two are large (on the order of the matrix size). In `sblasa_small.in`, M and N are large but K is small, while in `sblasb_small.in` the small parameter is M, and in `sblasc_small.in` the small parameter is N. The Level 2 BLAS are timed only in the first data set, where K is also used as the bandwidth for the banded routines.

a) Go to `LAPACK/TIMING` and make any necessary modifications to the input files. You may need to set the minimum time a subroutine will be timed to a positive value. If you modified the values of N or NB in Section 6.7.1, set M, N, and K accordingly. The large parameters among M, N, and K should be the same as the matrix sizes used in timing the linear equation routines, and the small parameter should be the same as the blocksizes used in timing the linear equation routines. If necessary, the large data set can be simplified by using only one value of LDA.

b) Run the programs for each data type you are using. For the REAL version, the commands for the small data sets are

```
xlintims < sblasa_small.in > sblasa_small.out
xlintims < sblasb_small.in > sblasb_small.out
xlintims < sblasc_small.in > sblasc_small.out
```

or the commands for the large data sets are

```
xlintims < sblasa_large.in > sblasa_large.out
xlintims < sblasb_large.in > sblasb_large.out
xlintims < sblasc_large.in > sblasc_large.out
```

Similar commands should be used for the other data types.

6.7.3 Timing the Eigensystem Routines

The eigensystem timing program is found in `LAPACK/TIMING/EIG` and the input files are in `LAPACK/TIMING`. Four input files are provided in each data type for timing the eigensystem routines, one for the generalized nonsymmetric eigenvalue problem, one for the nonsymmetric eigenvalue problem, one for the symmetric and generalized symmetric eigenvalue problem, and one for the singular value decomposition. For the REAL version, the small data sets are called `sgeptim_small.in`, `sneptim_small.in`, `sseptim_small.in`, and ssvdtim_small.in, respectively. and the large data sets are called sgeptim_large.in, sneptim_large.in, sseptim_large.in, and ssvdtim_large.in. Each of the four input files reads a different set of parameters, and the format of the input is indicated by a 3-character code on the first line.

The timing program for eigenvalue/singular value routines accumulates the operation count as the routines are executing using special instrumented versions of the LAPACK routines. The first step in compiling the timing program is therefore to make a library of the instrumented routines.

a) To make a library of the instrumented LAPACK routines, first go to LAPACK/TIMING/EIG/EIGSRC and type make followed by the data types desired, as in the examples of Section 5.3. The library of instrumented code is created in LAPACK/TIMING/EIG/eigsrc_PLAT.a, where PLAT is the user-defined architecture suffix specified in the file LAPACK/make.inc.

b) To make the eigensystem timing programs, go to LAPACK/TIMING/EIG and type make followed by the data types desired, as in the examples of Section 5.3. The executable files are called xeigtims, xeigtimc, xeigtimd, and xeigtimz and are created in LAPACK/TIMING.

c) Go to LAPACK/TIMING and make any necessary modifications to the input files. You may need to set the minimum time a subroutine will be timed to a positive value, or to restrict the number of tests if you are using a computer with performance in between that of a workstation and that of a supercomputer. Instead of decreasing the matrix dimensions to reduce the time, it would be better to reduce the number of matrix types to be timed, since the performance varies more with the matrix size than with the type. For example, for the nonsymmetric eigenvalue routines, you could use only one matrix of type 4 instead of four matrices of types 1, 3, 4, and 6. Refer to LAPACK Working Note 41 [3] for further details.

d) Run the programs for each data type you are using. For the REAL version, the commands for the small data sets are

```
xeigtims < sgeptim_small.in > sgeptim_small.out
xeigtims < sneptim_small.in > sneptim_small.out
xeigtims < sseptim_small.in > sseptim_small.out
xeigtims < ssvdtim_small.in > ssvdtim_small.out
```

or the commands for the large data sets are

```
xeigtims < sgeptim_large.in > sgeptim_large.out
xeigtims < sneptim_large.in > sneptim_large.out
xeigtims < sseptim_large.in > sseptim_large.out
xeigtims < ssvdtim_large.in > ssvdtim_large.out
```

Similar commands should be used for the other data types.

6.8 Send the Results to Tennessee

Congratulations! You have now finished installing, testing, and timing LAPACK. If you encountered failures in any phase of the testing or timing process, please consult our `release_notes` file on netlib.

`http://www.netlib.org/lapack/release_notes`

This file contains machine-dependent installation clues which hopefully will alleviate your difficulties or at least let you know that other users have had similar difficulties on that machine. If there is not an entry for your machine or the suggestions do not fix your problem, please feel free to contact the authors at

`lapack@cs.utk.edu`.

Tell us the type of machine on which the tests were run, the version of the operating system, the compiler and compiler options that were used, and details of the BLAS library or libraries that you used. You should also include a copy of the output file in which the failure occurs.

We would like to keep our `release_notes` file as up-to-date as possible. Therefore, if you do not see an entry for your machine, please contact us with your testing results.

Comments and suggestions are also welcome.

We encourage you to make the LAPACK library available to your users and provide us with feedback from their experiences.

6.9 Get support

First, take a look at the complete installation manual in the LAPACK Working Note 41 [3]. if you still cannot solve your problem, you have 2 ways to go:

- either send a post in the LAPACK forum

  `http://icl.cs.utk.edu/lapack-forum`

- or send an email to the LAPACK mailing list:

  `lapack@cs.utk.edu`.

Acknowledgments

Ed Anderson and Susan Blackford contributed to previous versions of this report.

Appendix A

Caveats

In this appendix we list a few of the machine-specific difficulties we have encountered in our own experience with LAPACK. A more detailed list of machine-dependent problems, bugs, and compiler errors encountered in the LAPACK installation process is maintained on *netlib*.

`http://www.netlib.org/lapack/release_notes`

We assume the user has installed the machine-specific routines correctly and that the Level 1, 2 and 3 BLAS test programs have run successfully, so we do not list any warnings associated with those routines.

1 `LAPACK/make.inc`

All machine-specific parameters are specified in the file `LAPACK/make.inc`.
The first line of this `make.inc` file is:

SHELL = /bin/sh

and will need to be modified to `SHELL = /sbin/sh` if you are installing LAPACK on an SGI architecture.

2 ETIME

On HPPA architectures, the compiler and loader flag `+U77` should be included to access the function `ETIME`.

3 ILAENV and IEEE-754 compliance

As some new routines in LAPACK rely on IEEE-754 compliance, two settings (`ISPEC=10` and `ISPEC=11`) have been added to ILAENV (`LAPACK/SRC/ilaenv.f`) to denote IEEE-754 compliance for NaN and infinity arithmetic, respectively. By default, ILAENV assumes an IEEE machine, and does a test for IEEE-754 compliance. **NOTE: If you are installing**

LAPACK on a non-IEEE machine, you MUST modify ILAENV, as this test inside ILAENV will crash!

Thus, for non-IEEE machines, the user must hard-code the setting of (`ILAENV=0`) for (`ISPEC=10` and `ISPEC=11`) in the version of `LAPACK/SRC/ilaenv.f` to be put in his library. For further details, refer to section 6.1.4.

Be aware that some IEEE compilers by default do not enforce IEEE-754 compliance, and a compiler flag must be explicitly set by the user.

On SGIs for example, you must set the `-OPT:IEEE_NaN_inf=ON` compiler flag to enable IEEE-754 compliance.

And lastly, the test inside ILAENV to detect IEEE-754 compliance, will result in IEEE exceptions for "Divide by Zero" and "Invalid Operation". Thus, if the user is installing on a machine that issues IEEE exception warning messages (like a Sun SPARCstation), the user can disregard these messages. To avoid these messages, the user can hard-code the values inside ILAENV as explained in section 6.1.4.

4 Lack of `/tmp` space

If `/tmp` space is small (i.e., less than approximately 16 MB) on your architecture, you may run out of space when compiling. There are a few possible solutions to this problem.

1. You can ask your system administrator to increase the size of the `/tmp` partition.

2. You can change the environment variable `TMPDIR` to point to your home directory for temporary space. E.g.,

   ```
   setenv TMPDIR /home/userid/
   ```

   where `/home/userid/` is the user's home directory.

3. If your archive command has an `l` option, you can change the archive command to `ar crl` so that the archive command will only place temporary files in the current working directory rather than in the default temporary directory /tmp.

5 BLAS

If you suspect a BLAS-related problem and you are linking with an optimized version of the BLAS, we would strongly suggest as a first step that you link to the Fortran 77 version of the suspected BLAS routine and see if the error has disappeared.

We have included test programs for the Level 1 BLAS. Users should therefore beware of a common problem in machine-specific implementations of xNRM2, the function to compute the 2-norm of a vector. The Fortran version of xNRM2 avoids underflow or overflow by scaling intermediate results, but some library versions of xNRM2 are not so careful about scaling. If xNRM2 is implemented without scaling intermediate results, some of the LAPACK test ratios may be unusually high, or a floating point exception may occur in the problems scaled near underflow or overflow. The solution to these problems is to link the Fortran version of xNRM2 with the test program. *On some CRAY architectures, the Fortran77 version of xNRM2 should be used.*

6 Optimization

If a large numbers of test failures occur for a specific matrix type or operation, it could be that there is an optimization problem with your compiler. Thus, the user could try reducing the level of optimization or eliminating optimization entirely for those routines to see if the failures disappear when you rerun the tests.

7 Compiling testing/timing drivers

The testing and timing main programs (xCHKAA, xCHKEE, xTIMAA, and xTIMEE) allocate large amounts of local variables. Therefore, it is vitally important that the user know if his compiler by default allocates local variables statically or on the stack. It is not uncommon for those compilers which place local variables on the stack to cause a stack overflow at runtime in the testing or timing process. The user then has two options: increase your stack size, or force all local variables to be allocated statically.

On HPPA architectures, the compiler and loader flag `-K` should be used when compiling these testing and timing main programs to avoid such a stack overflow. I.e., set `DRVOPTS = -K` in the `LAPACK/make.inc` file.

For similar reasons, on SGI architectures, the compiler and loader flag `-static` should be used. I.e., set `DRVOPTS = -static` in the `LAPACK/make.inc` file.

8 IEEE arithmetic

Some of our test matrices are scaled near overflow or underflow, but on the Crays, problems with the arithmetic near overflow and underflow forced us to scale by only the square root of overflow and underflow. The LAPACK auxiliary routine SLABAD (or DLABAD) is called to take the square root of underflow and overflow in cases where it could cause difficulties. We assume we are on a Cray if $\log_{10}$(overflow) is greater than 2000 and take the square root of underflow and overflow in this case. The test in SLABAD is as follows:

```
IF( LOG10( LARGE ).GT.2000. ) THEN
   SMALL = SQRT( SMALL )
   LARGE = SQRT( LARGE )
END IF
```

Users of other machines with similar restrictions on the effective range of usable numbers may have to modify this test so that the square roots are done on their machine as well. *Usually on HPPA architectures, a similar restriction in SLABAD should be enforced for all testing involving complex arithmetic.* SLABAD is located in `LAPACK/SRC`.

For machines which have a narrow exponent range or lack gradual underflow (DEC VAXes for example), it is not uncommon to experience failures in sec.out and/or dec.out with SLAQTR/DLAQTR or DTRSYL. The failures in SLAQTR/DLAQTR and DTRSYL occur with test problems which are very badly scaled when the norm of the solution is very close to the underflow threshold (or even underflows to zero). We believe that these failures could probably be avoided by an even greater degree of care in scaling, but we did not want to delay the release of LAPACK any further. These tests pass successfully on most other machines. An example failure in dec.out on a MicroVAX II looks like the following:

```
Tests of the Nonsymmetric eigenproblem condition estimation routines
DLALN2, DLASY2, DLANV2, DLAEXC, DTRSYL, DTREXC, DTRSNA, DTRSEN, DLAQTR

Relative machine precision (EPS) =     0.277556D-16
Safe minimum (SFMIN)             =     0.587747D-38

Routines pass computational tests if test ratio is less than   20.00

DEC routines passed the tests of the error exits ( 35 tests done)
Error in DTRSYL: RMAX =   0.155D+07
LMAX =     5323 NINFO=    1600 KNT=   27648
Error in DLAQTR: RMAX =   0.344D+04
LMAX =    15792 NINFO=   26720 KNT=   45000
```

9 Timing programs

In the eigensystem timing program, calls are made to the LINPACK and EISPACK equivalents of the LAPACK routines to allow a direct comparison of performance measures. In some cases we have increased the minimum number of iterations in the LINPACK and EISPACK routines to allow them to converge for our test problems, but even this may not be enough. One goal of the LAPACK project is to improve the convergence properties of these routines, so error messages in the output file indicating that a LINPACK or EISPACK routine did not converge should not be regarded with alarm.

In the eigensystem timing program, we have equivalenced some work arrays and then passed them to a subroutine, where both arrays are modified. This is a violation of the Fortran 77 standard, which says "if a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of the subprogram." [1] If this causes any difficulties, the equivalence can be commented out as explained in the comments for the main eigensystem timing programs.

[1] ANSI X3.9-1978, sec. 15.9.3.6

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Second Edition, SIAM, Philadelphia, PA, 1995.

[2] E. Anderson and J. Dongarra, *LAPACK Working Note 16: Results from the Initial Release of LAPACK*, University of Tennessee, CS-89-89, November 1989.

[3] E. Anderson, J. Dongarra, and S. Ostrouchov, *LAPACK Working Note 41: Installation Guide for LAPACK*, University of Tennessee, CS-92-151, February 1992 (revised June 1999).

[4] C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, *LAPACK Working Note #5: Provisional Contents*, Argonne National Laboratory, ANL-88-38, September 1988.

[5] Z. Bai, J. Demmel, and A. McKenney, *LAPACK Working Note #13: On the Conditioning of the Nonsymmetric Eigenvalue Problem: Theory and Software*, University of Tennessee, CS-89-86, October 1989.

[6] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, 16, 1:1-17, March 1990

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft.*, 16, 1:18-28, March 1990

[8] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, 14, 1:1-17, March 1988.

[9] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft.*, 14, 1:18-32, March 1988.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Soft.*, 5, 3:308-323, September 1979.