LAPACK3E -- A Fortran 90-enhanced version of LAPACK

Edward Anderson
Science Applications International Corporation
Edward.C.Anderson@saic.com

LAPACK3E is a version of the public domain numerical linear algebra package LAPACK 3 enhanced with selected features of Fortran 90. The use of Fortran 90 enhances LAPACK by allowing a common source for single and double precision, more uniform specification of scaling constants, and encapsulation of some internal subroutine interfaces. Thread-safety is introduced as a new feature for LAPACK by eliminating all the SAVE statements from the Fortran 77 package. Generic interfaces in the style of LAPACK95 are provided for all the subroutines in LAPACK, while maintaining backward compatibility with both the LAPACK 3 and LAPACK95 interfaces. Numerous bug fixes and improvements are also incorporated.

1.0 Introduction -- Evolution of LAPACK interfaces

The interfaces and naming conventions of LAPACK [Anderson *et al.* 1999] follow those of the Level 2 and Level 3 BLAS [Dongarra *et al.* 1988, Dongarra *et al.* 1990]. Like the BLAS, LAPACK was mostly written before Fortran 90 was standardized, so it was constrained to use only Fortran 77 features. Tools developed at NAG from Toolpack were used to enforce strict Fortran 77 compliance, to convert from single precision to double precision, and to format the comments and code in a uniform way. Subsequent updates to the basic LAPACK package, most recently in 1999, have maintained Fortran 77 compatibility despite the general availability of Fortran 90 compilers.

Fortran 90 wrappers for the driver routines in LAPACK were created as part of the LAPACK95 package [Barker *et al.* 2001]. These new interfaces provided

- generic interfaces for the BLAS and some (but not all) LAPACK routines
- dynamic allocation of work space
- optionally, shorter argument lists using optional arguments

LAPACK3E builds on the interface improvements of LAPACK95 by implementing

- generic interface modules for all routines in LAPACK
- at least two interfaces for each BLAS and LAPACK typed routine, the natural interface and a point interface
- common source for single and double precision

LAPACK3E also improves upon LAPACK in the following areas

- centralized specification of machine constants in a module
- elimination of SAVE statements for thread safety
- more uniform use of scaling constants such as `SMLNUM`, `BIGNUM`, etc.
- elimination of `SLABAD`, a crutch for avoiding badly scaled arithmetic
- improved scaling of several routines
- bug fixes from the LAPACK release notes and reports from LAPACK users

In addition, LAPACK3E incorporates many of the algorithmic improvements from the Cray Scientific Library [Anderson and Fahey 1997] and a subsequent LAPACK 3 supplement to libsci [Anderson 1999].

Source code for LAPACK3E and several precompiled libraries are available at

`http://www.netlib.org/lapack3e.`

This report describes the new features of LAPACK3E and the motivation behind them. Section 2.0 describes LAPACK3E features to implement a common source for single and double precision via the specification of Fortran 90 interface modules and compiler preprocessor options. Section 3.0 addresses the issue of thread safety in LAPACK and how it is achieved in LAPACK3E. Section 4.0 details some of the algorithmic improvements in LAPACK3E. Finally, Section 5.0 discusses possible extensions to LAPACK3E to further enhance its usefulness.

2.0 Common source

LAPACK was developed with the aid of a suite of Fortran tools called Toolpack. Three of the features that were used extensively were

1) Automatic conversion from single to double precision (with the aid of a file for mapping single precision names to double precision names)
2) Standard formatting of source code statements, including the placing of descriptive comments around the declarations
3) Checking of arguments for agreement in number, kind, and rank

Toolpack was very strict about Fortran 77 compatibility, so even widely adopted Fortran 77 extensions such as the DO/ENDDO construct could not be used in LAPACK. Fortran 77 compatibility may still be important to some people, but most, if not all, commercial Fortran compilers today fully support Fortran 90/95. LAPACK3E freely uses features of Fortran 90 and no facility is provided for going back to Fortran 77.

Fortunately, many of the functions of Toolpack can now be performed by Fortran 90/95 compilers. Conventions for parameterizing the KIND of floating point declarations make it simple to convert between the supported real and complex types (currently just 32-bit and 64-bit). Generic subroutine interfaces allow users to write more portable code, with the correct specific routine name chosen at compile time. The compile-time resolution of generic subroutine calls also provides some built-in argument checking and the ability to support different calling sequences through overloading. The only Toolpack feature that Fortran 90 compilers don't support is reformatting of the source code.

This section describes how features of the Fortran 90 language and *de facto* standards for preprocessing are used in LAPACK3E to achieve the goal of a common source for single and double precision.

2.1 Parameterizing the KIND

LAPACK95 addressed the problem of how to convert from single to double precision by parameterizing the Fortran 90 KIND using the parameter WP (for "working precision"). The LAPACK95 module LA_PRECISION contained the definition

```
MODULE LA_PRECISION
   INTEGER, PARAMETER :: SP=KIND(1.0), DP=KIND(1.0D0)
END MODULE LA_PRECISION
```

This module could be invoked in the calling routine either as

```
USE LA_PRECISION, ONLY: WP => SP     ! single precision
```

or

```
USE LA_PRECISION, ONLY: WP => DP     ! double precision
```

All subsequent REAL and COMPLEX variables were declared using the KIND value WP, for example,

```
REAL(WP) :: R
COMPLEX(WP) :: C
```

In this way, a subroutine could be changed from single precision to double precision by changing SP to DP on the USE LA_PRECISION line. However, separate subroutines were still provided for each precision.

In LAPACK3E, the KIND is also parameterized using an integer parameter WP. WP is defined along with several other constants in two Fortran 90 modules called LA_CONSTANTS (for 64-bit precision) and LA_CONSTANTS32 (for 32-bit precision), as follows:

```
MODULE LA_CONSTANTS
   INTEGER, PARAMETER     :: WP = 8
   ...
END MODULE LA_CONSTANTS

MODULE LA_CONSTANTS32
   INTEGER, PARAMETER     :: WP = 4
   ...
END MODULE LA_CONSTANTS32
```

Section 2.2 contains details on the other constants defined in these modules. Each subroutine in LAPACK has been modified to USE the LA_CONSTANTS module, and the preprocessor renames it to LA_CONSTANTS32 if necessary.

LAPACK3E makes some assumptions about the functionality of the Fortran preprocessor based on industry standard practices. First, the file extension of an LAPACK3E program unit that requires Fortran preprocessing is changed from .f to .F. The pre-processor is assumed to recognize #include, #define, and #if/#else/#endif constructs, and it is assumed that there are compiler options for macro expansion that will allow a #define statement to be applied to non-comment source code lines. All the information for the preprocessor is contained in a file called lapacknames.inc which is included as the first line of every LAPACK3E .F file. The template for this change is as follows:

```
#include "lapacknames.inc"
   SUBROUTINE SGETRF( ... )
      USE LA_CONSTANTS
      ...
   END
```

The included file contains #define statements for renaming one subroutine or module name to another. The one related to the LA_CONSTANTS module is

```
#ifdef LA_REALSIZE == 4 || LA_REALSIZE == 32
#define LA_CONSTANTS LA_CONSTANTS32
#endif
```

This specification tells the preprocessor to rename LA_CONSTANTS to LA_CONSTANTS32 if the compile time defined constant LA_REALSIZE is set to 4 or 32 (as in 4 bytes, or 32 bits). The compile lines for creating a 32-bit version of SGETRF from the common source file sgetrf.F on several different platforms are as follows:

```
IBM:   xlf -WF,-DLA_REALSIZE=4 -o sgetrf.o -c sgetrf.F
Cray:  f90 -F -DLA_REALSIZE=4 -o hgetrf.o -c sgetrf.F
SGI:   f90 -DLA_REALSIZE=4 -macro_expand -o sgetrf.o
        -c sgetrf.F
Sun:   f90 -DLA_REALSIZE=4 -o sgetrf.o -c sgetrf.F
```

Note that the 32-bit version of SGETRF would be called HGETRF on a Cray platform. Subroutine renaming is discussed further in section 2.3.

Once the KIND of "working precision" has been defined as the integer parameter WP, all floating point REAL and COMPLEX declarations can be written in terms of this KIND, as REAL(WP) or COMPLEX(WP). The KIND is also used in constants and in initialization statements. Most constants in LAPACK were already specified in PARAMETER statements. but those that were not have been made PARAMETERs in LAPACK3E, and their initialization statements have been modified to use WP, for example,

```
REAL(WP)    :: SCL
PARAMETER   :: ( SCL = 0.125_WP )
```

2.2 Parameterizing other constants

The previous section showed how one of the new Fortran 90 modules LA_CONSTANTS and LA_CONSTANTS32 is USE'd in every program unit in LAPACK3E. Besides the KIND parameter WP, many other constants common to several LAPACK routines are defined as PARAMETERs in the LA_CONSTANTS and LA_CONSTANTS32 modules. They include

ZERO	the real constant 0.0
HALF	the real constant 0.5
ONE	the real constant 1.0
TWO	the real constant 2.0
THREE	the real constant 3.0
FOUR	the real constant 4.0
EIGHT	the real constant 8.0
TEN	the real constant 10.0
CZERO	the complex constant (0.0, 0.0)
CHALF	the complex constant (0.5, 0.0)
CONE	the complex constant (1.0, 0.0)
SPREFIX	a character constant, set to ’S’ or ’D’ (’H’ or ’S’ for Cray)
CPREFIX	a character constant, set to ’C’ or ’Z’ (’G’ or ’C’ for Cray)

LAPACK also makes extensive use of a variety of machine parameters that are re-computed whenever they are needed via the auxiliary routines SLAMCH or DLAMCH. Computing these machine parameters added a significant amount of overhead to some subroutines and led to the use of SAVE blocks to retain the values between calls in a few places, a poor programming practice that interferes with thread safety. In LAPACK3E, these quantities are specified as `PARAMETER`s in `LA_CONSTANTS` and `LA_CONSTANTS32`, ensuring that they are defined consistently and obviating the need to SAVE them. They include

`EPS`	the machine epsilon, usually computed as `SLAMCH('E')`
`ULP`	the machine precision, usually computed as `SLAMCH('P')`
`SAFMIN`	the safe minimum, usually computed as `SLAMCH('S')`
`SAFMAX`	the safe maximum, often computed as `1/SAFMIN`
`SMLNUM`	a scaled minimum, often `SLAMCH('S')` / `SLAMCH('P')`
`BIGNUM`	a scaled maximum, often computed as `1/SMLNUM`
`RTMIN`	`sqrt(SMLNUM)`, used in sum-of-squares calculations
`RTMAX`	`sqrt(BIGNUM)`, used in sum-of-squares calculations

The modules `LA_CONSTANTS` and `LA_CONSTANTS32` have been hard-coded with constants appropriate for IEEE arithmetic, Cray IEEE arithmetic (on the CRAY T3D/T3E), and Cray arithmetic (as on Cray PVP systems through the CRAY SV1). Pre-defined constants known to the Cray compilers are used to distinguish between the cases. Versions of `SLAMCH` and `DLAMCH` that use the Fortran 90 intrinsic functions, instead of `LA_CONSTANTS` or `LA_CONSTANTS32`, to determine floating point model parameters are provided with the package and can be used to set the module files appropriately for any other architecture.

2.3 Renaming in the preprocessor

Section 2.1 introduced the include file `lapacknames.inc`, which appears in every `.F` file in LAPACK3E and contains renaming instructions for the preprocessor. Besides instructions for renaming `LA_CONSTANTS` to `LA_CONSTANTS32`, this file also contains instructions for renaming LAPACK routines for different precisions. The default is to use 64-bit precision, but if the defined constant `LA_REALSIZE` is set to 4 or 32, subroutine names appropriate for 32-bit precision are used.

On most machines, the LAPACK routine name for 32-bit precision begins with the letter `S` or `C`, such as `SGETRF` and `CGETRF`, and the routine name for 64-bit precision begins with the letter `D` or `Z`, such as `DGETRF` and `ZGETRF`. This is the case when the default REAL and COMPLEX types are 32 bits in size. On such machines, scientific users typically use 64-bit precision in order to get greater accuracy. However, on Cray platforms (including both the CRAY PVP: YMP/C90/T90/SV1 lines and the CRAY MPP: T3D/T3E lines) the default REAL and COMPLEX kind is 64 bits, and double precision is rarely used or is not available. The Cray Scientific Library provides 64-bit BLAS and LAPACK routines following the naming convention for the default REAL and COMPLEX kind, using routine names beginning with `S` and `C`. The CRAY T3E library also provides some 32-bit BLAS using the non-standard prefixes `H` for real and `G` for complex.

In LAPACK3E, the default precision of the common source modules is 64-bit. The file names and specific routine names on the `SUBROUTINE` or `FUNCTION` line use names starting with `S` for real data and `C` for complex data, following old habits from the LAPACK development days. If the defined constant `LA_REALSIZE` is set to 4 or 32, then the subroutine or function names are left alone on most machines and are renamed to `HYYZZZ` or `GYYZZZ` on Cray machines. If the defined constants `LA_REALSIZE` is not set (or is set to any value other than 4 or 32), then the subroutine or function names are renamed to `DYYZZZ` or `ZYYZZZ` on non-Cray machines, and left alone on Cray machines.

The outline of the `lapacknames.inc` file is as follows:

```
#if LA_REALSIZE == 4 || LA_REALSIZE == 32
#ifdef _CRAY
#define CGBTF2 GGBTF2
```

<other $C \rightarrow G$ and $S \rightarrow H$ rules here>

```
#endif
#define LA_CONSTANTS LA_CONSTANTS32

#else
#ifndef _CRAY
#define CGBTF2 ZGBTF2
```

<other $C \rightarrow Z$ and $S \rightarrow D$ rules here>

```
#endif
#endif
```

Renaming in the preprocessor, combined with the parameterization of constants from Sections 2.1 and 2.2, would have been enough to achieve the goal of a common source for single and double precision without any other changes. But LAPACK95 took the additional step of defining interface modules for the LAPACK routines it called and using generic interfaces for all internal subroutine calls. This forces the compiler to match a specific interface to the generic interface at compile time and provides a compile-time check that the number and type of arguments at the calling site is correct and that a pre-compiled LAPACK library is compatible with a user's code. This is such a powerful feature that it has been adopted in LAPACK3E as well

2.4 Generic interfaces

Fortran 90 has the capability for overloading, that is, defining a generic interface which can be resolved to a type-specific procedure at compile time. For example, one could define a generic interface LA_XFOO to one of four type-specific names SFOO, DFOO, CFOO, and ZFOO as follows:

```
MODULE LA_XFOO
INTERFACE LA_FOO

SUBROUTINE SFOO( X )
   USE LA_CONSTANTS32, ONLY : WP
   REAL(WP), INTENT(INOUT) :: X(*)
END SUBROUTINE SFOO

SUBROUTINE DFOO( X )
   USE LA_CONSTANTS, ONLY : WP
   REAL(WP), INTENT(INOUT) :: X(*)
END SUBROUTINE DFOO

SUBROUTINE CFOO( X )
   USE LA_CONSTANTS32, ONLY : WP
   COMPLEX(WP), INTENT(INOUT) :: X(*)
END SUBROUTINE CFOO

SUBROUTINE ZFOO( X )
   USE LA_CONSTANTS, ONLY : WP
   COMPLEX(WP), INTENT(INOUT) :: X(*)
END SUBROUTINE ZFOO

END INTERFACE ! LA_FOO
END MODULE LA_XFOO
```

Note that each of the specific interfaces has a unique argument list because the parameter WP is defined to be 4 in LA_CONSTANTS32 and 8 in LA_CONSTANTS. A program that USE's the LA_XFOO module can then call one of the xFOO routines using the generic name:

```
USE LA_XFOO        ! must go at the beginning of the program unit
...
CALL LA_FOO( X )
```

Depending on the type (whether real or complex) and kind (32-bit or 64-bit) of the array argument X, the compiler will substitute a call to one of SFOO, DFOO, CFOO, or ZFOO or it will generate an error message if no matching interface was found.

The problem with generic names is that the arguments at the calling site must match one of the specific interface specifications exactly in type, kind, and rank. Type and kind are no problem, but rank is a nuisance. Since Fortran 77 passes all arguments by reference, Fortran programmers are accustomed to treating an array as a block of consecutive storage locations and implicitly changing its shape across subroutine boundaries. A 1-D work array may be allocated in a high-level routine and passed to a subroutine where it is treated as a 2-D matrix, or a higher-dimensional array may call one of the Level-1 BLAS to perform a vector operation on one column of the array. With reference to the above module definition, all of the following calls are invalid:

```
USE LA_CONSTANTS, ONLY: WP
USE LA_XFOO
REAL(WP) :: A(10,10), W(100)

CALL LA_FOO( A )        !!! 2-D array A doesn't match 1-D array X
CALL LA_FOO( A(1,1) )  !!! scalar A(1,1) doesn't match 1-D array X
CALL LA_FOO( W(51) )   !!! scalar W(51) doesn't match 1-D array X
```

All Fortran 90 compilers will treat these calls as errors, complaining that it could not find a specific routine in the generic interface that matches the call. When mismatches occur, your options are

1) Match the interface to the call
2) Match the call to the interface

The first option can quickly lead to exponential explosion. In the BLAS, for example, there are up to three array arguments in the argument list, each of which could be declared as an array of one to seven dimensions in the calling program, or could be an indexed array that appears to be a scalar to the compiler. In LAPACK, there can be 10 or more array arguments, and providing interfaces for all the ways one of those routines could be called would require overloading a generic interface with over 1 billion specific interfaces! Obviously this is a worst case -- most instances of array contraction involve arrays of just one higher dimension, and most instances of array expansion involve arrays of one lower dimension -- but any exponential is a bad exponential when code size is an issue.

The second option has its disadvantages too, because it puts the burden on the user to adjust his call to match the library. Certainly most users would expect that if they declare all the arrays in their program exactly like those in the subroutine they want to call, they should be able to call the generic interface without any special indexing. This is called the "natural interface" in LAPACK3E, and it is provided as one option for every BLAS and LAPACK interface specification. Most of the BLAS and LAPACK interface specifications also provide a second interface, in which all the array arguments are defined as scalars. This interface would match a calling site in which all the arrays were indexed, a common occurrence in LAPACK. It is called the "point interface" in LAPACK3E. The LAPACK3E interface modules specify the point interface as the default, accessible without any additional overhead, and the natural interface as a wrapper to the point interface, with the overhead of an extra subroutine call. The wrapper routines are declared private to the module and are not callable directly.

For example, a generic interface `LA_COPY` implementing both the point and the natural interfaces for a BLAS-like subroutine `SCOPY1` would be defined as follows:

```
MODULE LA_XCOPY

INTERFACE LA_COPY

!  Point interface for xCOPY1

SUBROUTINE SCOPY1( N, X, Y )
   USE LA_CONSTANTS32, ONLY: WP
   INTEGER, INTENT(IN) :: N
   REAL(WP), INTENT(IN) :: X
   REAL(WP), INTENT(OUT) :: Y
END SUBROUTINE SCOPY1

MODULE PROCEDURE SCOPY1_X1Y1

END INTERFACE ! LA_COPY
PRIVATE SCOPY1_X1Y1
CONTAINS

!  Natural interface for xCOPY1

SUBROUTINE SCOPY1_X1Y1( N, X, Y )
   USE LA_CONSTANTS32, ONLY: WP
   INTEGER, INTENT(IN) :: N
   REAL(WP), INTENT(IN) :: X(*)
   REAL(WP), INTENT(OUT) :: Y(*)
   CALL SCOPY1( N, X(1), Y(1) )
END SUBROUTINE SCOPY1_X1Y1

END MODULE LA_XCOPY
```

The actual specific routine `SCOPY1` is generally defined in a separate program module and may reside in another library. In LAPACK3E, most such modules would have specifications for a `DCOPY1` and perhaps `CCOPY1` and `ZCOPY1` as well, but the other types are omitted here. With this definition, `LA_COPY` could be called in a user's program as follows:

```
USE LA_CONSTANTS32
USE LA_XCOPY
REAL(WP) :: W(100), X(100), Y(10,10)
...
CALL LA_COPY( 100, W, X )          ! Matches the natural interface
CALL LA_COPY( 50, W(1), X(51) )    ! Matches the point interface
CALL LA_COPY( 10, Y(1,1), Y(1,2) )! Matches the point interface
```

Any call can be made to match the point interface simply by indexing all the unindexed arrays. So, for example, the second call above, which might have been

```
CALL SCOPY1( 50, W, X(51) )
```

in the original type-specific code, must change `SCOPY1` to `LA_COPY` and `W` to `W(1)` to match one of the generic interfaces.

Alternatively, one could try to match the natural interface by passing array sections in place of indexed arrays. This technique should be used with caution because it can change the dimensions of multi-dimensional arrays and it may force the compiler to make a temporary copy of the array, consuming dynamic memory space and degrading performance. No array sections are used in LAPACK3E.

2.5 LAPACK3E modules

The archive file `liblapack3e.a` created by an LAPACK3E installation includes object code for all the LAPACK routines and auxiliary routines, replacements for the BLAS routines `SNRM2`, `SCNRM2`, `DNRM2`, and `DZNRM2`, any other BLAS needed for that platform, and all the LAPACK3E modules. Users of LAPACK3E can choose to use as many of the Fortran 90 modules as they like. They can use none of the LAPACK3E modules, and continue to call LAPACK routines by their type-specific Fortran 77 names. They can use just the `LA_CONSTANTS` or `LA_CONSTANTS32` modules to parameterize their floating point declarations as `REAL(WP)` or `COMPLEX(WP)`. Or they can use the BLAS and LAPACK modules, and call the BLAS and LAPACK subroutines by their generic names to do compile-time argument checking.

The BLAS modules are `LA_BLAS1`, `LA_BLAS2`, and `LA_BLAS3`. Depending on the ability of one's Fortran 90 compiler, it may be more efficient to specify which interfaces one wants from a particular module, for example,

```
USE LA_BLAS3, ONLY: LA_GEMM
```

Similar modules are defined for the BLAS in LAPACK95; the LAPACK3E modules implement only the Fortran 77 calling sequence, and provide both the natural interface and the point interface as defined in Section 2.4.

The main LAPACK modules are `LA_LAPACK` and `LA_AUXILIARY`. Interfaces for all the LAPACK routines and driver routines are defined in the `LA_LAPACK` module, which is approximately 20,000 lines long. Interfaces for the commonly used LAPACK auxiliary routines are defined in the `LA_AUXILIARY` module. It includes interfaces for routines to compute Givens rotations, Householder reflections, matrix norms, and other operations of possible interest to LAPACK users.

Many other LAPACK auxiliary routines are only used internally and are not of general interest. Their interfaces are defined in separate modules to keep the size of the `LA_AUXILIARY` module reasonable. For example, the module `LA_XGETF2` defines a generic interface to the LAPACK auxiliary routine `xGETF2`, which is only called from `xGETRF`. In version 1.1 of LAPACK3E, there are 63 of these internal modules. A complete list can be found by downloading the LAPACK3E package and entering

```
ls la*.f la*.F
```

in the `LAPACK3E/SRC` directory.

3.0 Thread safety

The issue of thread safety arises when running a parallel program on a symmetric multi-processing (SMP) machines using a shared memory programming model such as OpenMP. With OpenMP, the programmer identifies regions of his or her code that can be run in parallel (often, iterations of a DO loop) and inserts compiler directives to request parallel execution of those regions. In the parallel region, multiple processes or threads may be employed to execute the independent regions in parallel, thereby reducing the overall running time of the code. However, no two threads can modify the same memory location, or results will be unpredictable. COMMON blocks and SAVE blocks are usually allocated from the shared stack or heap, so the presence of these constructs in a subroutine may mean that it can not be called by two independent threads at the same time, in other words, the subroutine is not "thread-safe". There are no COMMON blocks in LAPACK, but there are some SAVE statements, so the most recent LAPACK release (version 3.0, 1999) is not thread-safe.

SAVE statements in LAPACK arise in two different contexts:

1) Reverse communication in the auxiliary routines `xLACON` and `xLASQ3`
2) Saving computed constants so they don't have to be recomputed, for performance reasons

This section discusses the changes that were made to LAPACK to eliminate the SAVE statements and make LAPACK3E thread-safe.

3.1 Reverse communication

Reverse communication is a technique for interfacing with the calling routine to have it provide information you don't need in the argument list. It is used in the LAPACK auxiliary routine `SLACON` to pass back a vector x, have the calling routine compute Ax or $A^T x$ for an array A, and re-enter with a new vector x. The array A may be a general matrix or banded or symmetric or triangular; `SLACON` doesn't need that information. It was a simple matter to take the three integer variables that were needed from one call of `xLACON` to the next and add them to the argument list of a new version, which was renamed `xLACN2`. Then the SAVE statement was not needed and could be deleted. Similar changes were made to `xLASQ3`, which was renamed `xLADQ3`.

3.2 Computed constants in a block

In low-level routines, the overhead of computing certain constants that involve function calls, divides, or square roots can be significant. To reduce this overhead, some LAPACK auxiliary routines enclosed the computation of constants in a block that was computed the first time the routine was called and then saved, using code such as

```
LOGICAL FIRST
DATA FIRST / .TRUE. /
SAVE FIRST, ...
IF( FIRST ) THEN
   ...
   FIRST = .FALSE.
END IF
```

The question of how best to remove SAVE statements inserted for performance reasons was one of the earliest motivations for parameterizing more of the machine parameters in LAPACK3E. As PARAMETERs, these constants are expressed as efficiently as the language will allow, and they never need to be computed, not even the first time through as in LAPACK. SAVE blocks in several auxiliary routines were removed as part of this parameterization, finally making LAPACK3E thread-safe.

4.0 Algorithmic improvements

The modifications to LAPACK necessary to create a common source code for single and double precision, to define Fortran 90 interfaces, and to make the package thread-safe did not require any changes to the algorithms or numerical results. However, that's not all that went into LAPACK3E. LAPACK3E also contains updated versions of some algorithmic improvements from the Cray Scientific Library [Anderson and Fahey 1997], a subsequent LAPACK 3 supplement to libsci [Anderson 1999], the LAPACK release notes (`http://www.netlib.org/lapack/release_notes.html`), and the new BLAS standard [Blackford *et al.* 2002]. In addition, work done to clean up the definitions of scaling constants, and subsequent testing on a Cray T3E-1200, an IBM RS/6000 SP, and a Sun Ultra-4, necessitated some more careful scaling near underflow and overflow, particularly on machines with IEEE arithmetic and gradual underflow.

4.1 SLAMCH

The original LAPACK package included a function SLAMCH to compute and return various parameters from the floating point model, such as the machine epsilon, the maximum exponent, the maximum and minimum magnitude, etc. The function performed a lot of computation and the instructions for installing LAPACK recommended running it once on a particular platform and putting its output in a DATA statement [Anderson, Dongarra, and Ostrouchov 1992]. These parameters are now directly accessible from Fortran 90 intrinsic functions, and a Fortran 90 version of SLAMCH has been available for some time [Anderson 1999]. In LAPACK3E, most uses of SLAMCH have been replaced by direct use of constants defined in the LA_CONSTANTS and LA_CONSTANTS32 module files. The Fortran 90 version is provided anyway in the SRC and INSTALL directories as a guide to setting LA_CONSTANTS and LA_CONSTANTS32 on new architectures.

4.2 SLABAD and the complex divide

Many LAPACK routines in LAPACK 3 call an auxiliary routine SLABAD (or DLABAD) to take the square root of SMLNUM and BIGNUM if the exponent range of the machine is very large. The name referred to "bad" Cray arithmetic, but it turned out that the lack of a guard digit in Cray floating-point was never an issue for scaling. Special scaling in LAPACK is really only needed to avoid overflow or underflow in the complex divide operation x/y. At the time LAPACK was developed, the Cray compiler (and some others) implemented a complex divide as

$$\frac{x}{y} = \frac{x}{y} \cdot \frac{\bar{y}}{\bar{y}} = \frac{x\bar{y}}{Re(y)^2 + Im(y)^2}$$

The denominator of the right-hand expression will overflow if *Re(y)* or *Im(y)* is greater than sqrt(OVERFLOW), and will underflow to zero, causing a divide by zero, if *Re(y)* and *Im(y)* are both less than sqrt(UNDERFLOW). The correct way to perform a complex divide can be found in Knuth [1981] and is implemented in the LAPACK auxiliary routines SLADIV and CLADIV. The algorithm to compute $(p,q) = (a,b)/(c,d)$ is as follows:

```
if( ABS( D ) < ABS( C ) ) then
   E = D / C
   F = C + D*E
   P = ( A+B*E ) / F
   Q = ( B-A*E ) / F
else
   E = C / D
   F = D + C*E
   P = ( B+A*E ) / F
   Q = ( -A+B*E ) / F
end if
```

This algorithm could be made safer still by testing for C or D outside a safe range for reciprocating, and doing additional scaling in those cases. This wasn't needed in xLADIV, but it is done in xRSCL (Section 4.3) and xLARTG (Section 4.5).

SLABAD was removed from all the LAPACK routines and auxiliary routines in LAPACK3E, but it was necessary to adjust the scaling in a few places to avoid some unscaled sums of squares. SLABAD was not removed from the test package, but it was replaced by a "stub" that does no work. This allowed for much more rigorous testing of the LAPACK software than had previously been done, because test cases scaled closer to the underflow and overflow thresholds were generated. In testing on a CRAY T3E, a special version of SLABAD was used that takes the square root of underflow and overflow in the test code for the complex cases only.

4.3 SRSCL/CRSCL

LAPACK extended the BLAS by providing a reciprocal scale routine that, given a scalar SA and a vector X, computes (1/SA)*X. In order to guard against overflow or underflow if SA were outside the range of numbers that could be safely reciprocated, the subroutine set up an iterative scaling loop that scaled by the safe minimum or its reciprocal as needed to get SA into a safe range. A pseudo-code[1] outline of the LAPACK algorithm is as follows (note d = denominator, u = numerator, |d| = ABS(d)):

```
      d = SA; u = 1; t = .TRUE.
      do while (t)
         d1 = d*SMLNUM; u1 = u/BIGNUM
         if( |d1| > |u| .and. u /= 0 ) then
!              Pre-multiply x by SMLNUM if d is large compared to u.
            s = SMLNUM; d = d1
         else if( |u1| > |d| ) then
!              Pre-multiply x by BIGNUM if d is small compared to u.
            s = BIGNUM; u = u1
         else
!              Multiply x by u/d and return.
            s = u / d; t = .FALSE.
         end if
         x = s*x
      end do
```

[1] This is valid Fortran 90 code except for $|x|$ = ABS(x), but the code structure and some of the operators and variable names have been modified from the original to make it easier to read.

The problem with this algorithm is that if SA is Inf or -Inf on a machine with IEEE arithmetic, no amount of scaling will ever get it into a safe range. Consider what happens when SA = Inf. Then d = Inf, u = 1, d1 = Inf, u1 = SMLNUM, and every time through the (now infinite) loop, x is scaled by SMLNUM with no change in d or u.

In the LAPACK3E version, there is no infinite scaling loop. One scaling pass should be enough to get a number outside the range of invertible numbers into that range (if that were not true, then more than half the representable numbers would be "denormalized" numbers. Put another way, the number of exponent bits would be fewer than the log of the number of mantissa bits. This is theoretically possible, but unlikely.) Consequently, the LAPACK3E version of SRSCL does at most one extra scaling. It could perhaps be improved further by checking for exceptional values on entry to avoid even this one extra scaling. The simplified algorithm has the following structure:

```
      d = SA
      if( |d| > BIGNUM ) then
!        Pre-multiply x by SMLNUM if |SA| is very large
         x = SMLNUM*x; d = d*SMLNUM
      else if( |d| < SMLNUM ) then
!        Pre-multiply x by BIGNUM if |SA| is very small
         x = BIGNUM*x; d = d*BIGNUM
      end if
      s = 1/d
      x = s*x
```

4.4 SLASSQ/CLASSQ

The LAPACK auxiliary routine SLASSQ computes a sum of squares for a vector *x*, returning scalar values SCL and SUMSQ such that

$$SCL^2 \cdot SUMSQ = x(1)^2 + x(2)^2 + \ldots + x(n)^2 + s^2 \cdot q$$

where *s* is the initial value of SCL and *q* is the initial value of SUMSQ. The values *s* and *q* allow xLASSQ to be used to compute a single sum of squares for a series of vectors, as is required for the Frobenius norm of a matrix. If properly done, SLASSQ can be used to implement SNRM2, the 2-norm routine from Level 1 BLAS, as follows:

```
SCL = ONE
SUMSQ = ZERO
CALL SLASSQ( N, X, INCX, SCL, SUMSQ )
SNRM2 = SCL*SQRT( SUMSQ )
```

The scaling factor `SCL` is key to the safe implementation of `SLASSQ`; without it, the sum of squares would overflow if the magnitude of any element of *x* were greater than sqrt(`SAFMAX`), or it would underflow to zero if the magnitude of each element of *x* were less than sqrt(`SAFMIN`).

The LAPACK version of `SLASSQ` makes one pass through the vector *x* and continually rescales the sum by the largest *x(i)* in absolute value. This method avoids overflow by keeping the sum of squares near 1 while the scale factor `SCL` is always less than 1. The LAPACK algorithm for a unit-stride vector *x* is as follows:

```
do i = 1, N
   if( x(i) /= ZERO ) then
      d = |x(i)|
      if( SCL < d ) then
         SUMSQ = 1 + SUMSQ*(SCL/d)**2
         SCL = d
      else
         SUMSQ = SUMSQ + (d/SCL)**2
      end if
   end if
end do
```

There are two problems with this algorithm:

1) If `SCL` >= `ONE` and `SUMSQ` = `ZERO` on entry and |`x(i)`| < 1 for each `x(i)`, then none of the `x(i)`'s will be scaled and the sum of squares will underflow to zero if each `x(i)` is less than sqrt(`SAFMIN`) in absolute value. This is bad because a natural way to initiate a sum of squares is with `SCL` = `ONE` and `SUMSQ` = `ZERO`.
2) The algorithm inhibits vectorization and consequently exhibits poor performance.

The LAPACK3E version is based on a version described for the Cray Scientific Library [Anderson and Fahey 1997] with additional logic for denormalized numbers, which are not invertible. In the LAPACK3E algorithm, an initial pass is made through the vector *x* to find the maximum entry in absolute value. If the maximum is in a safe range, a second pass is made to compute the sum of squares without scaling. If the maximum is outside the safe range, the second pass computes the sum of squares with scaling, where the scaling constant is always an invertible number. The two-pass algorithm is efficient on a vector machine because it removes the IF tests from the loops, and it also does well on a cache-based machine because the second pass will typically find the vector in the cache. The new algorithm is up to 40 times faster than LAPACK on a Cray vector machine, and up to four times faster on an IBM SP.

The LAPACK3E algorithm for a unit-stride vector *x* is outlined below.

```
      HITEST = RTMAX / real( N+1, WP )

!     Pass through once to find the maximum value in x.

      p = ZERO
      do i = 1, N
         p = max( p, |x(i)| )
      end do
      q = max( SCL*sqrt( SUMSQ ), p )
      if( SCL == ONE .and. q > RTMIN .and. q < HITEST ) then

!        No scaling should be needed.

         do i = 1, N
            SUMSQ = SUMSQ + x(i)**2
         end do
      else if( p > ZERO ) then

!        Scale by q if SCL = ONE, otherwise scale by max( q, SCL ).

         q = min( max( q, SAFMIN ), SAFMAX )
         if( SCL == ONE .or. SCL < q ) then
            SUMSQ = ( SUMSQ*( SCL / q ) )*( SCL / q )
            SCL = q
         end if

!        Add the sum of squares of values of X scaled by SCL.

         do i = 1, N
            SUMSQ = SUMSQ + ( x(i) / SCL )**2
         end do
       end if
```

An alternative algorithm due to Blue [1978] is found in the original Level 1 BLAS routine `SNRM2` [Lawson *et al.* 1979]. That algorithm makes one pass through the vector *x* as in the LAPACK version, but it only scales when necessary as in the LAPACK3E version. Blue's algorithm is efficient and, in fact, is faster than LAPACK3E, but it is written in a very unstructured programming style and consequently many scientific library writers have apparently tried to rewrite it, with not much success. The instructions for installing LAPACK3E include compiling a new version of `xNRM2` to avoid the widespread scaling inadequacies in the netlib and vendor implementations of this function.

4.5 SLARTG/CLARTG

The LAPACK auxiliary routines for generating and applying Givens rotations are used extensively in the package and have been through several recent revisions. In the real case, a Givens rotation is a rank-2 correction to the identity of the form

$$G(i,j,\theta) = \begin{bmatrix} 1 &&&&&&&&& \\ & \cdots &&&&&&&& \\ && 1 &&&&&&& \\ &&& c &&&& s &&& \\ &&&& 1 &&&&& \\ &&&&& \cdots &&&& \\ &&&&&& 1 &&& \\ &&& -s &&&& c &&& \\ &&&&&&&& 1 && \\ &&&&&&&&& \cdots & \\ &&&&&&&&&& 1 \end{bmatrix}$$

where $c = \cos(\theta)$and $s = \sin(\theta)$for some angle θ. Premultiplication of a vector *x* by *G(i,j,*θ*)* amounts to a clockwise rotation of θ radians in the *(i, j)* plane. If $y = G(i, j, \theta)\, x$, the vector *y* can be described by

$$y_k = \begin{cases} cx_i + sx_j, & k = i \\ -sx_i + cx_j, & k = j \\ x_k, & k \neq i, j \end{cases}$$

We can force y_j to be zero by choosing θ to be the angle described by the vector $\begin{bmatrix} x_i & x_j \end{bmatrix}^T$ in the *(i, j)* plane, which leads to the formulas

$$r = \pm\sqrt{x_i^2 + x_j^2}$$
$$c = x_i / r$$
$$s = x_j / r$$

In practice, *r*, like any square root of a sum of squares, must be computed with scaling to avoid underflow when both x_i and x_j are less than the square root of underflow, or when one of x_i or x_j is greater than the square root of overflow.

Since a Givens rotation only modifies two elements of a vector, its action can be described by the 2-by-2 linear transformation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad c^2 + s^2 = 1$$

The LAPACK auxiliary routine `SLARTG` computes c, s, and r satisfying this equation, given f and g. The choice of sign for c and s gives rise to several algorithmic variants [Anderson 2000]:

1. BLAS SROTG [Lawson *et al.* 1979]

$c = 1, s = 0$:	$g = 0$
$c = 0, s = 1$:	$f = 0$
$c > 0$, $\mathrm{sign}(s) = \mathrm{sign}(g/f)$:	$\lvert f\rvert > \lvert g\rvert$
$s > 0$, $\mathrm{sign}(c) = \mathrm{sign}(f/g)$:	otherwise

2. LAPACK 3 SLARTG

$c = 1, s = 0$:	$g = 0$
$c = 0, s = 1$:	$f = 0$
$\mathrm{sign}(c) = -\mathrm{sign}(f)$, $\mathrm{sign}(s) = -\mathrm{sign}(g)$:	$\lvert f\rvert > \lvert g\rvert$ and $f < 0$
$\mathrm{sign}(c) = \mathrm{sign}(f)$, $\mathrm{sign}(s) = \mathrm{sign}(g)$:	otherwise

3. LAWN150

$c = \mathrm{sign}(f)$, $s = 0$:	$g = 0$
$c = 0$, $\mathrm{s} = \mathrm{sign}(g)$:	$f = 0$
$\mathrm{sign}(c) = \mathrm{sign}(f)$, $\mathrm{sign}(s) = \mathrm{sign}(g)$:	otherwise

4. BLAS Technical Forum version [Bindel *et al.* 2002]

$c = 1, s = 0$:	$g = 0$
$c = 0$, $s = \mathrm{sign}(g)$:	$f = 0$
$c > 0$, $\mathrm{sign}(s) = \mathrm{sign}(f)*\mathrm{sign}(g)$:	otherwise

Algorithm 3 has the smallest set of points of discontinuity in the real case, but Algorithm 4 has the best continuity in the complex case if we add the requirement that the complex code produce identical results to the real code when f and g are real [Bindel *et al.* 2002]. In LAPACK3E, an algorithm equivalent to Algorithm 4 is used. The mathematical specification for this algorithm in the complex case is as follows:

$$\begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \qquad c^2 + s \cdot \bar{s} = 1$$

where c is real and r, f, g, and s are complex. When $f = g = 0$, $r = 0$ and we choose $c = 1$, $s = 0$. Otherwise,

$$\operatorname{sgn}(x) \equiv \begin{cases} x/|x|, & x \neq 0 \\ 1, & x = 0 \end{cases}$$

$$r = \operatorname{sgn}(f)\sqrt{|f|^2 + |g|^2}$$

$$c = \frac{|f|}{\sqrt{|f|^2 + |g|^2}}$$

$$s = \frac{\operatorname{sgn}(f) \cdot \bar{g}}{\sqrt{|f|^2 + |g|^2}}$$

Bindel *et al.* [2002] describe an implementation of `xLARTG` which contains SAVE blocks and tries to scale at least 3 times if given IEEE exceptional values as inputs. The LAPACK3E version fixes these problems in a straightforward implementation. `SLARTG` is shown in abbreviated form below:

```
SUBROUTINE SLARTG( F, G, CS, SN, R )
USE LA_CONSTANTS
REAL(WP)           CS, F, G, R, SN
REAL(WP)           D, F1, FS, G1, GS, T, TT
INTRINSIC          ABS, SIGN, SQRT
F1 = ABS( F )
G1 = ABS( G )
IF( G == ZERO ) THEN
   CS = ONE
   SN = ZERO
   R = F
ELSE IF( F == ZERO ) THEN
   CS = ZERO
   SN = SIGN( ONE, G )
   R = G1
ELSE IF( F1 > G1 ) THEN
   IF( F1 > SAFMIN .AND. F1 < SAFMAX ) THEN
      T = G / F
      TT = SQRT( ONE+T*T )
      CS = ONE / TT
      SN = T*CS
      R = F*TT
   ELSE
      F1 = MIN( SAFMAX, MAX( F1, SAFMIN ) )
      FS = F / F1
      GS = G / F1
      TT = SQRT( FS*FS + GS*GS )
      D = ONE / TT
      CS = ABS( FS )*D
      SN = GS*SIGN( D, F )
      R = F1*SIGN( TT, F )
   END IF
```

```
ELSE
   IF( G1 > SAFMIN .AND. G1 < SAFMAX ) THEN
      T = F / G
      TT = SQRT( ONE+T*T )
      D = ONE / TT
      CS = ABS(T)*D
      SN = SIGN( D, F )*SIGN( ONE, G )
      R = G1*SIGN( TT, F )
   ELSE
      G1 = MIN( SAFMAX, MAX( G1, SAFMIN ) )
      FS = F / G1
      GS = G / G1
      TT = SQRT( FS*FS + GS*GS )
      D = ONE / TT
      CS = ABS( FS )*D
      SN = GS*SIGN( D, F )
      R = G1*SIGN( TT, F )
   END IF
END IF
RETURN
END
```

The complex case is similar but is complicated slightly by concerns over the safety of dividing a complex number by a real. According to the Fortran 90 standard, `Z/R` where `Z` is complex and `R` is real should be computed by first converting `R` to complex and then computing `Z/Y`, where

```
Y = CMPLX( R, ZERO, KIND(R) )
```

However, a few machines are known to compute `Z/Y` as `(Z*CONJG(Y))/(Y*CONJG(Y))` as described in Section 4.2, which is prone to overflow or underflow when computing `Y*CONJG(Y)` = `REAL(Y)**2` + `IMAG(Y)**2`. In LAPACK3E's version of `CLARTG`, `Z/R` computations are always converted to `(1/R)*Z`. For performance reasons, we would like this to be computed as

```
CMPLX( (1/R)*REAL(Z), (1/R)*IMAG(Z) )
```

If an optimizing compiler does not do this for you, it could be done explicitly.

The LAPACK auxiliary routines `SLARGV` and `CLARGV` are related to `SLARTG` and `CLARTG`, computing a vector of Givens rotations instead of just one. In LAPACK3E, these routines are consistent with `SLARTG/CLARTG`, a feature not shared by all previous versions of LAPACK.

4.6 SGEBAL/CGEBAL

Subroutines for balancing (BALANC in EISPACK [Smith *et al.* 1976] and xGEBAL in LAPACK) are applied to general nonsymmetric matrices in order to isolate eigenvalues and make the magnitudes of elements in corresponding rows and columns nearly equal. This can reduce the 1-norm of a matrix and improve the accuracy of its computed eigenvalues and/or eigenvectors. The first step looks for zeros in a dense *n*-by-*n* matrix A and permutes it to the form

$$PAP^T = \begin{bmatrix} R_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & R_{33} \end{bmatrix}$$

where R_{11} and R_{33} are upper triangular. The diagonal elements of R_{11} and R_{33} are the isolated eigenvalues of A. The second step determines a diagonal scaling matrix D_{22} that is applied to A_{22}, replacing it with $D_{22}^{-1}A_{22}D_{22}$. The scaling matrix is determined iteratively by scaling each row and column by powers of the radix until they are nearly equal. After the permutations and balancing, the matrix A is transformed to

$$D^{-1}PAP^TD = \begin{bmatrix} R_{11} & D_{22}A_{12} & A_{13} \\ 0 & D_{22}^{-1}A_{22}D_{22} & D_{22}^{-1}A_{23} \\ 0 & 0 & R_{33} \end{bmatrix}$$

The scaling for a particular row and column of A_{22} proceeds as follows. Let

`C` = 1-norm of the *i*-th column of A_{22} excluding the diagonal

`R` = 1-norm of the *i*-th row of A_{22} excluding the diagonal

Assume `C` <= `R`, `SCL` = 1/8 (LAPACK) or 1/16 (EISPACK), and `SCL2` = `SCL`^{2}. In `BALANC` from EISPACK, the row scaling factor `G` is computed as[2]

```
F = R*SCL
G = ONE
DO WHILE( C < F )
   F = F*SCL2
   G = G*SCL
END DO
```

After the loop, the row is scaled by `G` < 1 and the column by `1/G`. This scaling is safe for A_{22}, but there are no checks for overflow or destructive underflow in the scaling of A_{12} and A_{23}. The LAPACK version adds these checks, but in an unnecessarily cumbersome way.

In `SGEBAL` from LAPACK3E, the checks are implemented as additional tests on the `DO WHILE` condition. Two additional norms are required:

[2] This code fragment is restructured from the original -- there are no DO WHILE statements in EISPACK.

CA = max norm of the *i*-th column of *A*
RA = max norm of the *i*-th row of *A*

Then the balancing step takes the form

```
F = R*SCL
G = ONE
DO WHILE( C < F .AND. CA < G*SFMAX .AND. RA*G > SFMIN )
   F = F*SCL2
   G = G*SCL
END DO
```

The extra test `CA < G*SFMAX` guards against overflow in the diagonal element of A_{22} or in the *i*-th column of A_{12} when scaled by `1/G`. EISPACK probably should have included this test as well. The test `RA*G > SFMIN` guards against destructive underflow when scaling the *i*-th row by `G`. This condition probably wouldn't have occurred in the days of EISPACK because most machines would have underflowed to zero, making F = 0 and forcing C < F to be false. However, in IEEE arithmetic, precision can be lost when scaling the row even before it underflows to zero.

We haven't said what `SFMAX` and `SFMIN` are. To implement EISPACK-style balancing, one would set `SFMAX` to be the safe maximum (`SAFMAX` in LAPACK3E) and `SFMIN` to be the safe minimum (`SAFMIN` in LAPACK3E). But there are test cases for which EISPACK-style balancing makes the matrix norm much worse and can lead to overflow later on. This issue needs further study, but as an interim fix, LAPACK3E's `SGEBAL` limits `SFMAX` to 100 times the norm of *A*. The LAPACK3E test package also replaces the test code for `xGEBAL` and `xGEBAK` in the LAPACK 3 package with a more thorough suite of tests.

4.7 Solve routines xxxTRS

One of the more dramatic improvements to LAPACK for the Cray Scientific Library was a redesign of the routines to solve linear systems with multiple right-hand sides [Anderson and Fahey 1997]. The LAPACK routines, in keeping with the strategy of pushing all parallelism into the BLAS, solve across all the right-hand sides at once using Level 2 and 3 BLAS. This design is inefficient if there is only one or a small number of right-hand sides, the most commonly occurring case, because one of the dimensions in the Level 2 or 3 BLAS call is small. A better approach is to take advantage of the independence of the right-hand sides and solve for them in parallel. In the Cray Scientific Library, parallelism was expressed in terms of Cray autotasking directives; in LAPACK3E, it is expressed using OpenMP. This method is also faster for one right-hand side because it avoids calling Level 2 and 3 BLAS with one dimension equal to 1.

The following discussion of the redesign of the LAPACK solve routines borrows from Anderson and Fahey [1997]. The standard solve routine xxxTRS was renamed xxxTS2, and special case code for one right-hand side was added to xxxTS2. Then a new routine xxxTRS was written to call xxxTS2 in a parallel loop. The stride for the parallel loop, called NB by analogy with the block factorization routines where NB is the block size, is determined by a call to a new auxiliary routine ILATRS, which returns NB = 1 if it is more efficient to solve for each right-hand side independently, and NB > 1 if it is more efficient to solve for NB right-hand sides at a time.

The structure of SSYTRS with the new design is as follows:

```
      IF( NRHS.EQ.1 ) THEN
         NB = 1
      ELSE
         NB = MAX( 1, ILATRS( 1, SPREFIX // 'SYTRS',
     &              UPLO, N, NRHS, -1, -1 ) )
      END IF
      IF( NB.GE.NRHS ) THEN
         CALL LA_SYTS2( IUPLO, N, NRHS, A, LDA, IPIV, B,
     &                   LDB)
      ELSE

!$OMP PARALLEL DO PRIVATE(J,JB)
!$OMP CNCALL

         DO J = 1, NRHS, NB
            JB = MIN( NRHS-J+1, NB )
            CALL LA_SYTS2( IUPLO, N, JB, A(1,1), LDA,
     &                     IPIV(1), B(1,J),LDB)
         END DO

!$OMP END PARALLEL DO

      END IF
```

Note in particular that if NB = 1, then each column of the right hand side matrix B is solved independently in the DO loop.

This leaves the question of how to set the "blocksize" NB in ILATRS. Based on experience with optimizing LAPACK for the Cray library, we assume that it is better to solve for one right-hand side at a time if NB < 8. The OpenMP function OMP_GET_NUM_THREADS is used to determine how many threads are available for parallel execution, and the function OMP_GET_DYNAMIC is used to determine if the number of threads is dynamic (if true) or static (if false). If the number of threads is dynamic, the right-hand sides are parceled out in chunks of about 32. If the number of threads is static, the right-hand sides are divided evenly among the threads. The function ILATRS is implemented as follows in LAPACK3E:

```
#ifdef _OPENMP
   NCPU = OMP_GET_NUM_THREADS()
#else
   NCPU = 1
#endif
   ILEN = 32

   IF( NCPU.EQ.1 ) THEN
      NB = N2
   ELSE
#ifdef _OPENMP
      IF( .NOT.OMP_GET_DYNAMIC() ) THEN
         NUSE = NCPU
      ELSE
         NUSE = ( N2+ILEN-1 )/ILEN
      END IF
#else
      NUSE = NCPU
#endif
      NB = ( N2+NUSE-1 ) / NUSE
   END IF

   IF( NB.LT.8 ) NB = 1
   ILATRS = NB
```

Note that the value of NUSE may be greater than the number of threads if the number of threads is dynamic and the number of right-hand sides N2 is large. This is a heuristic choice intended to help balance the load in the dynamic environment.

5.0 Future extensions to LAPACK3E

LAPACK3E generalizes some of the type-specific features of LAPACK in a way that is portable, maintainable, and extensible. However, these features have only been applied to the LAPACK source routines. Applying these design improvements to the LAPACK test and timing code would improve the package in many of the same ways that LAPACK has been improved, by combining the source code for single and double precision, defining scaling parameters more consistently, and reducing the overhead of small cases. It would also simplify testing new variants of LAPACK, such as a 32-bit version for Cray platforms or a higher precision version on a machine with extended precision arithmetic. Complicating this task is the lack of any tools to apply LAPACK3E-style changes to LAPACK-style code.

Much more could also be done to simplify the argument lists of certain LAPACK routines. Currently, the generic interfaces in LAPACK3E only save the users from having to decide what should be the first letter of the subroutine. LAPACK95 went further, overloading the interfaces of the LAPACK driver routines with other, simpler, interfaces. So, for example, instead of an option argument to indicate whether or not to compute an array of eigenvectors, the array itself would be an optional argument, and the driver routine would compute the eigenvectors if that argument were provided. LAPACK95 also allocates workspace dynamically in the LAPACK driver routines, so that the workspace arguments are not needed. With some similar additions to the interface specifications, LAPACK3E could be made a superset of LAPACK95.

One concern about creating yet another version of LAPACK is that it further complicates the problem of keeping all the LAPACK versions consistent. Currently there are versions of LAPACK in Fortran, Fortran 90/95, C, C++, and Java, as well as related projects such as ScaLAPACK [Blackford *et al.* 1997] that all start from the same base. The creation of a master version for all of these LAPACKs is well beyond the scope of this paper. But LAPACK3E does unify LAPACK 3 with variations in the Cray Scientific Library and in the LAPACK 3 supplement to libsci, and it could subsume LAPACK95 with some additional work overloading the interfaces of the driver routines. In this respect, it is a step in the right direction.

References

E. Anderson, Installing LAPACK 3 on CRAY Machines, online technical report, Dec. 1999. (`http://www.cs.utk.edu/~eanderso/lapack3.html`)

E. Anderson, Discontinuous Plane Rotations and the Symmetric Eigenvalue Problem, LAPACK Working Note 150, University of Tennessee, CS-00-454, December 2000.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition*, SIAM, Philadelphia, 1999.

E. Anderson, J. Dongarra, and S. Ostrouchov, Installation Guide for LAPACK, LAPACK Working Note 41, University of Tennessee, CS-92-151, June 1992.

E. Anderson and M. Fahey, Performance Improvements to LAPACK for the Cray Scientific Library, LAPACK Working Note 126, University of Tennessee, CS-97-359, April 1997.

V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Wasniewski, and P. Yalamov, *LAPACK95 Users' Guide*, SIAM, Philadelphia, 2001.

D. Bindel, J. Demmel, W. Kahan, and O. Marques, On Computing Givens Rotations Reliably and Efficiently, *ACM Trans. Math. Soft.*, Vol. 28, No. 2, June 2002, pp. 206-238.

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, An Updated Set of Basic Linear Algebra Subprograms (BLAS), *ACM Trans. Math. Soft.*, Vol. 28, No. 2, June 2002, pp. 135-151.

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

J. L. Blue, A portable Fortran program to find the Euclidean norm of a vector, *ACM Trans. Math. Soft.*, Vol. 4, No. 1, 1978, pp. 15-23.

S. Browne, J. Dongarra, E. Grosse, and T. Rowan, The Netlib Mathematical Software Repository, *D-Lib Magazine*, September 1995. (`http://www.netlib.org/srwn/srwn21.html`)

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

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 14(1):1-17, March 1988.

D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Second edition, Addison-Wesley, Reading, MA, 1981.

C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Soft.*, Vol. 5, 1979, pp. 308-323.

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, Second Edition*, Springer-Verlag, 1976.