Sca/LAPACK Program Style

v0.3, August 2006

August 27, 2006

Sca/LAPACK Program Style
by v0.3, August 2006

Contents

1 Introduction

The purpose of this document is to facilitate contributions to LAPACK and ScaLAPACK by documenting their design and implementation guidelines. The long-term goal is to provide guidelines for both LAPACK and ScaLAPACK. However, the parallel ScaLAPACK code has more open issues, so this document primarily concerns LAPACK. Details on threading and other issues likewise have been deferred and will be shaped by future contributions discussions.These are style and design *guidelines*, exceptions *may* be allowed if accompanied by sound reasons and good documentation. To enable development and use of automated tools, rules regarding issues like formatting are far less flexible. Copyright and licensing rules are completely inflexible; see the section on "Licensing" below.Many parts of the current code may not live up to these guidelines. Fixes are greatly appreciated. See the section on "Maintenance Projects" and the "TODO" notes in each section.Not all of the guidelines have been completely decided. Questions to contributors are listed after relevant sections.Please coordinate possible contributions with lapackers@cs.utk.edu and lapackers@cs.berkeley.edu. More development resources are available at http://www.netlib.org/lapack-dev/ .

1.1 High-Level Design Choices

These guidelines follow from some considerations of implementation languages, memory managements, and performance goals.We do not have the resources or desire to throw away the existing code and recode the algorithms from scratch in a new language. Our plans are to keep the code primarily in Fortran. Existing code is primarily in the dialect of Fortran 77 available in the 1990s. New code *may* adopt newer language features so long as they

- are widely available in compilers,
- improve ease of use,
- improve ease of maintenance, and
- do not impose a significant cost in performance or memory usage.

Current prohibitions and warnings on new language features are discussed in "Fortran Language Features" .Some components and routines may be in written in C. Current C components include

- the reference Extended Precision BLAS implementation, and
- ScaLAPACK's communication substrate, the PBLAS and BLACS, as well as such support routines as REDIST.

Future C components will be considered on a case-by-case basis. Interoperability with C is important but difficult to achieve both simply and portably. Some considerations are listed in "Interoperability with C".LAPACK strives to accomodate both users who want control over memory allocation as well as users who want automatic workspace allocation. We have decided not to use dynamic allocation within LAPACK computational routines. Instead, we will provide a programmatic mechanism to query routines and determine necessary workspace, see "Workspace Issues". Concerns about memory usage and performance also drive decisions on language features in "Fortran Language Features" .Our design should enable writing user-friendlier wrappers in other high-level languages (including full-featured F95), but the design of these wrappers is not the subject of this document.

1.2 Reproducibility

Reproducing exact results between runs on the same data for debugging or certification is challenging. We must balance common needs for reproducibility against performance gains from randomized algorithms and parallel processing. The trade-offs and design considerations are not completely clear at this point, but we must support reproducible results whenever possible.To allow users to compute reproducible results whenever a platform allows, the following rules apply to uses of randomization in LAPACK:

- Use the LAPACK PRNGs (Pseudo-Random Number Generators) like xLARUV and xLARNV. These are portable routines. When given the same seed, they generate the same numbers on all platforms.
- Testing and timing code must set the seed to a constant in the source.
- LAPACK library routines also must use a constant seed whenever possible. Constant seeds are not possible if the code's randomized performance or correctness relies on not having static bad cases. If a routine cannot use a static seed, it must allow users to input a seed value in order to support debugging.
- The impact of tuning parameters on the numerical results must be documented.

We cannot control the reproducibility of tuned platform BLAS. Users should be able to use the reference BLAS when necessary.There are many unresolved issues regarding reproducing numerical results on parallel platforms. At the very least, ScaLAPACK must provide options to control communication and ensure reproducibility for a particular process layout. Discussions of and contributions towards distributed and shared-memory parallel reproducibility are welcome.

TODO Notes

- Need to test the PRNGs. See "Maintenance Projects".

2 Copyrights and Licensing

All code must have the same license. We have chosen the Modified BSD license, available both in the COPYING file in the *next* distribution as well as at http://www.opensource.org/licenses/bsd-license.php.Before a contribution can be accepted into LAPACK or ScaLAPACK, the contributors must submit documentation that

- states that the contributor grants all necessary copyright and patent licenses of the contribution so we can include it in LAPACK and ScaLAPACK under the Modified BSD license, and
- declares the contributor to have necessary legal authority to grant the above.

The document makes claims only regarding the contribution to LAPACK, not on any other uses of the same material. The necessary form is available at http://www.netlib.org/lapack-dev/LAPACK_contributors_agreement.tx

2.1 Citing the authors of the software

The original authors of the software will be listed in the code first, followed by authors of important changes. Contributors will be listed in both the source distribution and the Users' Guide.

3 Documentation

Each type of documentation has a specific purpose. This section provides an overview of what topics the documentation should address.

Source Code Inform users on how to call a routine.

Users' Guide Tie routines together and provide examples.

Working Notes Document the mathematical details and design considerations.

3.1 Source Code

Each source file for an exported routine (e.g. `SRC/*.f`) must contain documentation on how to use the routine. The documentation should

- briefly describe the routine's purpose,
- describe each argument,
- optionally provide a reference for further details.

The documentation's format is explained in "Source Formatting" along with other aspects of source formatting.

3.2 LAPACK Users' Guide

The LAPACK Users' Guide (LUG) provides higher-level information about the routines and their interfaces without the full technical and mathematical details that drive the design. The current LUG is available through SIAM and electronically at http://www.netlib.org/lapack/lug/index.html.Each published interface must be incorporated into the LUG. The precise form will depend on the routines. The goals are to provide user with

- the relationships between routines,
- mathematical and design details needed for users to chose an appropriate routine,
- brief examples of common uses, and
- high-level advice to users.

The interface documentation from the source code will be reproduced at the end of the LUG. Therefore there is no need to duplicate this detailed interface documentation in the earlier LUG sections.

Questions to contributors

- Would contributors like template LaTeX sections? Or should we make the entire document available and ask contributors to send modifications?

TODO Notes

- The LUG needs updating for MR^3, the extended refinement routines, etc.

3.3 LAPACK Working Notes (LAWNs)

Details about mathematical derivations and design decisions belong in the LAPACK Working Notes (LAWNs). These notes will be published through the LAPACK repository at Netlib; the current notes are available at http://www.netlib.org/lapack/lawns/index.html. We make no restrictions on other publications of the same material.Every major routine should have a LAWN documenting the algorithm, error analysis, performance, and testing. LAWNs describing new routines must address the following points:

- Purpose
- Usage
- Error conditions and returns
- Applicability and restrictions (including arithmetic, see below for current arithmetic restrictions in LAPACK)
- Discussion of the method and algorithm
- Algorithm design decisions
- Accuracy
- Tuning parameters
- Test cases
- Timing tests
- References

Currently the only arithmetic restrictions are in the eigenvalue routines (IEEE exception handling in EGR and EBZ algorithms, monotonicity and rounding accuracy in EBZ, guard digits in EDC). For further details, grep for "IEEE" and "arithmetic" in the SRC directory.Keep in mind that LAWNs are static publications, but LAPACK is evolving code. The LAWNs provide snapshots of design decisions. Engineering choices in the routines' designs may change once the code is reviewed.

4 Workspace and Memory Management

It is natural to ask whether we should dynamically allocate workspace inside LAPACK routines, and so do away with complicated workspace counting code and asking the user to allocate it. Many users would like this. However, there are also important users who want to control their own workspace, both for performance reasons and so to be ensure memory allocation errors cannot occur (e.g. in real-time control loops). To make everyone happy (except perhaps us developers) we currently plan to handle this as follows:

1. Change all routines that need workspace to be queryable for how much workspace is needed. This means that users can quickly ask how much space is necessary and handle it appropriately themselves, and it will let us more easily write wrappers in various languages that provide automatic memory allocation. This option is already available in much of LAPACK by using LWORK=-1; our goal is to extend this feature to all of LAPACK.

2. Routines that call other routines needing workspace will in turn have to query those other routines to determine the total amount of workspace. In other words, no assumptions about workspace needs of other routines should be built in. This will both help eliminate bugs and permit innovations and bug fixes that change workspace to be made to one routine without changing lots of others too.

3. We are not sure how many "levels" of workspace query to provide: a routine could have different values for

   a. the smallest possible workspace for correct functioning

   b. the right amount for reasonable speed given various block size values (from querying ILAENV)

   c. maximum space for very fast functioning. For example, DC needs $3n^2$ + O(n) total space, MRRR needs $2n^2$ + O(n) and QR needs only n^2 + O(n) for the dense symmetric eigenproblem. So a smart driver could call any of these, depending on workspace.

Options (a), (b) and (c) could be implemented using other negative values of LWORK. This is also discussed below in the section on "Determining the Block Size for Block Algorithms".The routines xGEESX and xGGESX are the only exceptions using LWORK for a workspace query. For these routines, the value of LWORK is dependent upon the dimension of the invariant subspace (SDIM), and is not known on entry to the routine.We are considering new drivers that split the functionality of xGEESX and xGGESX into two subroutine calls, so that at the end of the first call all the eigenvalues are known, and the second routine is given the specific desired subset of these eigenvalues, whose cardinality is then known. This means both routines could determine their workspace needs in advance.Testing and timing code can dynamically allocate memory. Arrays larger than some systems' caches may *require* dynamic allocation for some compilers (e.g. Sun's).

Questions to contributors (and users)

- What style of queries are most useful? In new routines, we have the option of using:

  1. An LWORK query for each type of workspace (IWORK, RWORK, CWORK, BWORK).
  2. One single query routine modeled after ILAENV which would take the name of the routine and the type of workspace needed.
  3. A separate query routine for each LAPACK procedure that returns the necessary or requested workspace for each used workspace type. The routines could be named something akin to SGESVX_WORKSPACE.

  Maintaining existing interfaces requires one of the last two choices.

TODO Notes

- Split xGEESX and xGGESX as above.

5 Routine Naming and Design

5.1 Routine Naming

The LAPACK project intends to maintain backwards compatibility. New routines get new names. Fixes to old routines that preserve the interface can keep the names. If the only change to the interface is to *decrease* the amount of workspace needed, the old routine name can be kept.We are no longer limited to 6 characters for routine names. Underscores are ok. At present, this causes problems in a variety of

codes that assume 6 character names when they parse input springs (e.g. ILAENV, XERBLA, xOPLA, etc.). We are working to address these issues (see "Maintenance Projects").

Questions to contributors

- What criteria should we use when deciding if changes to the definitions of numerical output arguments constitutes an interface changes? There is no single answer, and there are obvious limits, but we would like some feedback. For example, if we want to change the definition of pivot growth in linear systems to be measured per column rather than over the entire matrix, should that be considered a change to the interface?

5.2 Internal Design and Name Usage

Recursion is permitted. But beware of excessive stack usage; it should not be used as a substitute for workspace allocation as discussed in "Workspace Issues". Roughly speaking, as long as the stack only grows to O(log n) then there should be no problem. If the alternative is a non-recursive routine that uses the same space, and if the recursive routine is more readable, prefer the recursive version.We will use the BLAS and extended BLAS names from BLAST Forum in new code. There is a new BLAS standard "[blast]", but a partial reference implementation exists only for the extended (and sparse) BLAS. We would like new LAPACK routines to use the new interface and provide a reference implementation that simply calls the appropriate old BLAS routine. Over time this will encourage developers and vendors to migrate to the new BLAS. We will provide an extended BLAS release as part of LAPACK, since it is needed for the new iterative refinement codes.We prohibit the use of I/O within LAPACK computational routines, with a natural exception for parallel communication. Policies regarding any future out-of-core routines will be considered along with the routines. There should be no terminal or user I/O outside the default XERBLA ("Error Handling").

NoteFor background on low-level library design issues relevant to many modern platforms, refer to "[shared-lib]".

5.3 Error Handling and the Diagnostic Argument INFO

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

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

All driver and auxiliary routines check that input arguments such as N or LDA or option arguments of type character have permitted values. If an illegal value of the ith argument is detected, the routine sets INFO = -i, and then calls an error-handling routine XERBLA.The standard version of XERBLA issues an error message and halts execution, so that no LAPACK routine would ever return to the calling program with INFO < 0. However, this might occur if a non-standard version of XERBLA is used.

5.4 Determining Machine Arithmetic Parameters

The xLAMCH interfaces for returning machine arithmetic parameters will be maintained and will be made thread-safe. New code must use xLAMCH.Many Fortran language queries provide equivalent information. For reference, the table below summarizes the widely available Fortran 95 query functions that are equivalent to xLAMCH calls.

xLAMCH parameter	F95 intrinsic	Description
E	EPSILON	Epsilon
B	RADIX	Base
N	DIGITS	Significand digits
M	MINEXPONENT	Min. exponent

xLAMCH parameter	F95 intrinsic	Description
U	TINY	Underflow threshold
L	MAXEXPONENT	Max. exponent
O	HUGE	Overflow threshold

The two queries "P" and "S" can be derived from F95 intrinsics as follows (example are given for double precision quantities):

- P: Product of epsilon and the base

```
EPSILON(0.0d0) * RADIX(0.0d0)
```

- S: Safe minimum such that 1/xLAMCH(*S*) does not overflow

```
SFMIN = TINY(0.0d0)
SMALL = 1.0d0/HUGE(0.0d0)
if (SMALL >= SFMIN) SFMIN = SMALL * (1+EPSILON(0.0d0))
```

xLAMCH("R") has no corresponding query function in Fortran 95 but grep reveals no use of this query in LAPACK. Likewise, SLAMC1's test for round-to-nearest and SLAMC2's test for compliance to IEEE-754 have no corresponding queries in Fortran 95. Fortran 2003 provides these and additional useful predicates (e.g. IEEE_IS_NAN) when an implementation supports IEEE-754 arithmetic, but these compilers are not yet widely available.C89 provides similar support in float.h. C99 includes greatly improved support for IEEE-754 arithmetic in float.h and math.h. This support is in widely available C compilers, but some system libraries do not fully support fused multiply-add and flag tests.

Note

- Fortran 2003 provides additional queries like NaN detection on implementations of IEEE-754, but Fortran 2003 compilers are not yet widely available.
- Note that, technically, Fortran 95's number model does not **allow** gradual underflow. Many implementations still support it.

5.5 Determining the Block Size for Block Algorithms

LAPACK routines that implement block algorithms need to determine what block size to use. The intention behind the design of LAPACK is that the choice of block size should be hidden from users as much as possible, but at the same time easily accessible to installers of the package when tuning LAPACK for a particular machine.LAPACK routines call an auxiliary enquiry function ILAENV, which returns the optimal block size to be used as well as other parameters. The version of ILAENV supplied with the package contains default values that led to good behavior over a reasonable number of our test machines, but to achieve optimal performance, it may be beneficial to tune ILAENV for your particular machine environment. Ideally a distinct implementation of ILAENV is needed for each machine environment (see also Chapter 6 of the LUG). The optimal block size also may depend on

- the routine,
- the combination of option arguments (if any),
- the problem dimensions, and
- the available workspace.

We ultimately plan to determine these values by an automatic tuning process akin to the one used in ATLAS and related packages.If ILAENV returns a block size of 1, then the routine performs the unblocked algorithm, calling Level 2 BLAS, and makes no calls to Level 3 BLAS.Some LAPACK routines require a work array whose size is proportional to the block size (see subsection 5.1.7 of the LUG). The actual length of the work array is supplied as an argument LWORK. The description of the arguments WORK and LWORK typically goes as follows:

```
WORK
     (workspace) REAL array, dimension (LWORK)
     On exit, if INFO = 0, then WORK(1) returns the optimal
     LWORK.
LWORK
     (input) INTEGER
     The dimension of the array WORK. LWORK >= max(1,N).

     For optimal performance LWORK >= N*NB, where NB is the
     optimal block size returned by ILAENV.  The routine
     determines the block size to be used by the following
     steps: ...
```

For such a routine, if LWORK >= max(1,N),

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

The minimum value of LWORK that would be needed to use the optimal block size is returned in WORK(1) (see "Workspace Issues").Thus, the routine uses the largest block size allowed by the amount of workspace supplied, as long as this is likely to give better performance than the unblocked algorithm. WORK(1) is not always a simple formula in terms of N and NB.The specification of LWORK gives the minimum value for the routine to return correct results. If the supplied value is less than the minimum — indicating that there is insufficient workspace to perform the unblocked algorithm — the value of LWORK is regarded as an illegal value, and is treated like any other illegal argument value (see subsection 5.1.9 of the LUG).If in doubt about how much workspace to supply, users must use the query mechanism described in "Workspace Issues". *That section is still in flux. There is no mechanism that will work for the entire library at this point, but that is being addressed. The query mechanism will be available for* ***all*** *routines that use workspace.*

6 Fortran Language Features

We have considered what features of Fortran to use beyond those in the F77 dialect we have used so far, and what features to deprecate. The following summarizes our current plans, based on availability of Fortran compilers, and sometimes conflicting desires of users for ease of use, high performance, and memory safety.New code should meet the following guidelines. Assistance in converting existing code to the guidelines would be gratefully accepted. These guidelines are far more flexible in testing and timing code.

Use DO WHILE and DO / END DO DO WHILE was adopted by Fortran 90 and is widely available. Using DO with END DO rather than a numbered CONTINUE makes the code more readable both by people and parsers.

No new common blocks Currently there are no common blocks in the SRC code, just TESTING and TIMING. These create thread safety problems, and also create problems in TESTING and TIMING, because they assume 6 character routine names, a restriction we are eliminating (see below).

No equivalence statements New Fortran language features make them unnecessary for many uses.

No save statements SAVE statements often impose thread safety problems. The next release will provide SAVE-free versions of xLAMCH, xLACON, and other SAVE-using source routines. xLAMCH routines will preserve the interface. Removing SAVE statements from xLACON and others would require changing the interface. Instead, we provide and use functions contributed by Sven Hammarling and NAG Ltd. The release notes will provide more information on the new routines.

Use internal procedures rather than statement functions Statement functions are an "obsolescent" Fortran language feature. See "Obsolescent Fortran" for a full list.

Provide INTERFACE blocks in include files INTERFACE blocks provide argument type-checking within the Fortran language without needing external tools like ftnchek or NAG's utilities. See "Source Formatting" for guidelines about include files.

Do not use assumed-shape arrays in interfaces Assumed-shape argument arrays, those defined like

```
REAL A(:,:)
```

add information to the low-level arguments passed into the routine, complicating language interoperability. Additionally, passing arrays as assumed-shape arguments may impose performance and memory costs. The F95 wrappers may use assumed arrays in the style of LAPACK95 "[lapack95]", as opposed to LAPACK3E "[lapack3e]".The preferred mechanism for passing arrays is described in "Source Formatting".

Modules are limited to arbitrary / high-precision routines Cleve Moler tells us that their use would greatly complicate MATLAB™'s build process. However, we likely will need modules to provide arbitrary precision versions of LAPACK routines. We will approach this by parsing the module-free code and automatically generating new code with modules inserted to redefine the arithmetic. Fortran abstractions available only through modules permit a variety of longer fixed or variable precision packages to be used (QUAD, ARPREC, MPFR, etc.). Modules may also be used for the F95 wrappers.

Derived types (structures) cannot appear in interfaces Derived types are Fortran's structured data type, and they suffer from many language restrictions.

- They may appear in INTERFACE blocks only when the derived type is defined in another module.
- Two separate derived type declarations for the same name may be represented differently unless the SEQUENCE keyword appears. SEQUENCE packs the elements in declaration order, requiring careful attention to alignment for high performance.
- Fortran prior to F03 does not provide any guarantees for interoperability between derived types and other languages.

Thus, we cannot use derived types in interfaces or pass them between externally visible routines without modules. Work on the arbitrary precision versions may require modules for this reason. We will address interoperability concerns at some future point, possibly by relying on F03's BIND(C).

Be careful with WHERE masks and FORALL assignments Fortran 95 WHERE masks and FORALL assignments have strict evaluation semantics. Both require that the mask and right-hand side be fully evaluated before the WHERE or FORALL is executed. If the mask or right-hand side is an expression, or if there are data dependencies or possible aliasing issues, the compiler may dynamically allocate space to hold an intermediate evaluation.

Be careful with array slicing Modern Fortran allows MATLAB™-like ranges and array slicing. Used with care, these can make code much more readable. However, slicing arrays declared with LAPACK's explicit leading dimension style (see "Array Arguments") may lead to intermediate copies. Many compilers provide options to warn when the particular compiler decides to use intermediate allocations, but the decision varies according to the compiler, the sequence of optimizations applied, and the language requirements. If in doubt, write a loop.

Use BLAS routines rather than intrinsics (e.g. MATMUL) For similar reasons as the above. We know the memory behavior of the BLAS, but not of the compilers' array intrinsics.

6.1 Interoperability with C

We must mix C and Fortran internally to call the reference extended precision BLAS as well as ScaLAPACK's PBLAS and BLACS substrate. We are actively investigating how best to handle C and Fortran interoperability both internally and for external interfaces. We would prefer a light-weight solution as

opposed to systems like Babel "[babel]".Currently, we assume the following types share the same low-level machine representation. C89 does not provide complex numbers or operators, but C99 guarantees the associations between C89 and C99 below.

Fortran	**C99**	**C89**
INTEGER	int	int
REAL	float	float
DOUBLE PRECISION	double	double
COMPLEX	float _Complex	float[2]
DOUBLE COMPLEX	double _Complex	double[2]

Thus, we assume that an array COMPLEX A(5, 6) on the Fortran side corresponds to an array float _Complex a[30] in C99 and float a[60] in C89. The (2,3) entry of the Fortran A would be located at a[2 + 3*5] in C99. In C89, the real part would be located at a[2*(2 + 3*5)], and the imaginary part at a[2*(2 + 3*5)+1].One prohibition to types in cross-language interfaces must apply:

Do not use C's enum in header files that will be parsed by C++ C specifies that all enums are the same size. C++ explicitly allows different enums to have different sizes. Many compilers happen to use the C definition in C++, but that is **not** guaranteed.

6.2 Obsolescent Fortran

The following is an incomplete list of features declared obsolescent by or deleted from the 1995 and 2003 Fortran standards. If the item is listed as "not used", then they almost certainly do not appear in LAPACK. If the item is "likely not used", then searching has not turned up instances, but we are not completely sure.

Real DO variables Obsolescent in Fortran 1995, deleted in Fortran 2003. Likely not used in LAPACK.

Branching to an END IF from outside the IF block Obsolescent in Fortran 1995, deleted in Fortran 2003. Likely not used in LAPACK. Can be replaced by a CONTINUE after the END IF.

PAUSE Obsolescent in Fortran 1995, deleted in Fortran 2003. Likely not used in LAPACK.

ASSIGN and assigned GO TO statements Obsolescent in Fortran 1995, deleted in Fortran 2003. Likely not used in LAPACK.

The H edit descriptor Obsolescent in Fortran 1995, deleted in Fortran 2003. Not used in LAPACK.

Arithmetic IF statements Obsolescent in Fortran 2003. (We have not searched for these.)

Termination of multiple DO loops on the same statement Obsolescent in Fortran 2003. (We have not searched for these.)

Using one CONTINUE or statement for multiple DO loops Obsolescent in Fortran 2003. (We have not searched for these.)

Alternate returns Obsolescent in Fortran 2003. Likely not used in LAPACK.

Computed GO TO Obsolescent in Fortran 2003. There are a few computed GO TOs in xLACON and xLACN2, at least.

Statement functions Obsolescent in Fortran 2003. Replace with internal procedures (CONTAINS). Used as CABS1 and ZABS1, at least.

DATA statements intermixed with executable statements Obsolescent in Fortran 2003. Likely not used in LAPACK.

Assumed length character functions Obsolescent in Fortran 2003. Not used in LAPACK.

Fixed-format source Obsolescent in Fortran 2003. We do not expect compilers to forget how to parse fixed-format source, so we do not prohibit its use.

CHARACTER*len Obsolescent in Fortran 2003. Replace with CHARACTER(len). The CHARACTER*1 form currently is used for option arguments.

New code must not use these features, and existing code should have them removed.

TODO Notes

- Computed GO TOs in xLACON and xLACN2 need removed. See "Maintenance Projects".
- We need to consider converting the statement functions CABS1 and ZABS1 either to far more verbose internal functions (possibly in an INCLUDEd file) or to open definitions within an internal module. See "Maintenance Projects".
- Replace CHARACTER*1 with CHARACTER(1) or just CHARACTER. See "Maintenance Projects".

7 Source Formatting

In general, follow existing LAPACK style. The LAPACK routines conform to a single set of conventions for their design and documentation. To accomodate ease of parsing just described, we will insist on a uniform format.The structure of a LAPACK routine includes:

a. the SUBROUTINE or FUNCTION statement, followed by statements declaring the type and dimensions of the arguments;

b. an IMPLICIT NONE declaration;

c. a summary of the **Purpose** of the routine;

d. descriptions of each of the **Arguments** in the order of the argument list;

e. (optionally) **Further Details** (only in the code, not in the generated LUG pages in Part 2)

f. **Internal Parameters** if any are used (only in the code, not in the generated LUG pages in Part 2).

We are no longer limited to 6 characters for variable names, and underscores are ok.

7.1 File Names and Organization

Computational routines and drivers are placed in the SRC directory of the LAPACK distribution. Each file holds one routine. Testing and timing codes are in subdirectories under TESTING and TIMING, respectively. More information about testing and timing codes is in "Testing and Timing".Fortran sources in fixed format and using only Fortran 77 features shall have the extension ".f".Fortran files designed to be included in other source files, say for defining common constants, shall have extension ".fh". Such files must be formatted for **both** fixed- and free-format source. There will be no characters in the first six columns, and lines will be at most 72 characters wide. Do not use continuations in include files. These files are to be included using the now-standard Fortran INCLUDE statement and not the C preprocessor. We are not supporting C preprocessing in Fortran at this time.

7.2 Order of Arguments

Arguments of an LAPACK routine appear in the following order:

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

This is not a hard and fast rule. It may make sense to cluster arguments according to other considerations.

7.3 Argument Descriptions

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

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

The description of each argument gives:

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

7.4 Option Arguments

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

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

The corresponding lower-case characters may be supplied (with the same meaning), but any other value is illegal (see subsection 5.1.9 of the LUG).A longer character string can be passed as the actual argument, making the calling program more readable, but only the first character is significant; this is a standard feature of Fortran 77. For example:

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

7.5 Problem Dimensions

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

7.6 Array Arguments

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

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

It should be assumed, unless stated otherwise, that vectors and matrices are stored in one- and two-dimensional arrays in the conventional manner. That is, if an array X of dimension (N) holds a vector x, then X(i) holds x_i for i = 1, …, n. If a two-dimensional array A of dimension (LDA,N) holds an m-by-n matrix A, then A(i,j) m). See Section 5.3 of the LUG for more about storage of matrices.Note that array arguments are usually declared in the software as assumed-size arrays (last dimension *), for example:

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

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

8 Testing and Timing Routines

TODO Note

- Describe the different levels of testing: torture, build, and deployment/installation.
- Document the TESTING/ and TIMING/ directory contents, and the dependencies between routines. It's ornate and nasty.
- Separating and documenting the test matrix library (TESTING/MATGEN) would be very useful. See "Maintenance Projects".

8.1 Testing routines

- Test with various NB
- If the new code replaces a current implementation, ensure it passes the current tests.
- If the code introduces a new interface,
  - test all the functionality for typical inputs,
  - test any extra accuracy claims with a few examples,
  - test all the possible INFO returns, and
  - test with all relevant dimensions equal to zero.
- All testing routines should be deterministic. Seeds for generating pseudorandom numbers must be constant to allow for debugging. See the section on "Reproducibility".

8.2 Timing routines

In the future timing will not be a default part of the installation procedure, but optional. If the routine is providing a new function, or is a better version of an old routine that is remaining in the library, new timing code should be added to the existing code. If the new routine simply replaces an old routine, the existing timing code can be used. For example, the new QR and QZ routines will use the old timing code.

9 Suggested Maintenance Projects

List of projects we would greatly appreciate someone handling, and that make for a good introduction to the code. They are time-consuming jobs but they do not require a deep familiarity with the mathematics or algorithms within LAPACK.

- Ensure IMPLICIT NONE is in all routines, and add necessary declarations.

- Remove COMMON blocks.
- Replace obsolescent language features.
- Replace fixed-length CHARACTER*6s throughout. Such declarations that appear in argument lists can be replaced by variable-length strings (CHARACTER(*)) without harm. Fixing local arrays and common block declarations will take more work. The testing and timing code has the most serious issues.
- Make the testing and timing code useful for other developers. It would be widely useful if outside implementations could run in exactly the same test and timing harness as LAPACK routines. This would provide one standard for comparison.
- Clean up the build system. Distributors may appreciate some form of automatic configuration. The GNU autoconf system does not handle cross-compilation well, and that is required for some resource-constrained platforms like vector machines and embedded systems. Portable linker scripts could be useful for controlling which symbols are exported; see "[shared-lib]" for examples on ELF-based systems.
- Add tests for the random number generators that compare the first, say, 1000 generated numbers to 1000 reference numbers.

10 Reference Notes

On-line copies of the Fortran 66, Fortran 77 and MIL-STD 1753 documents are available through http://www.fortran.c stds_docs.html. The Fortran 95 standard is insanely expensive, but the contents are summarized in "[f95handbook]".The Fortran 2003 standard is available for around $30 electronically from ANSI. The "Final Committee Draft", which is essentially the same as the standard, is available at http://j3-fortran.org/-doc/standing/2003/007.pdf and http://std.dkuug.dk/jtc1/sc22/open/n3661.pdf. The 1999 C standard also is available electronically through ANSI for a reasonable price.

11 Bibliography

[1] [f95handbook] Adams, Bainerd, Martin, Smith, and Wegener. Fortran 95 Handbook; Complete ISO/ANSI Reference. MIT Press. ISBN 0-262-51096-0.

[2] [blast] Blackford, et al. Basic Linear Algebra Subprograms Technical Forum Standard. International Journal of High Performance Computing, 15(3-4), 2001. http://www.netlib.org/blas/blast-forum/

[3] [lapack95] Barker, Blackford, Dongarra, Du Croz, Hammarling, Marinova, Wasniewski, and Yalamov. LAPACK95 Users' Guide. SIAM, 2001. http://www.netlib.org/lapack95/

[4] [lapack3e] Anderson. LAWN 158: LAPACK3E - A Fortran 90-enhanced version of LAPACK. http://www.netlib.org/lapack3e/lawn158.pdf

[5] [shared-lib] Drepper. How To Write Shared Libraries. http://people.redhat.com/drepper/dsohowto.pdf

[6] [babel] Dahlgren, Epperly, Kumfert, and Leek. Babel Users' Guide. 2005. http://www.llnl.gov/casc/components/docs.html