Applying Software Testing Metrics to Lapack

David J. Barnes[1] and Tim R. Hopkins[1]

Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF, UK
{d.j.barnes, t.r.hopkins}@kent.ac.uk

Abstract. We look at how the application of software testing metrics affects the way in which we view the testing of the Lapack suite of software. We discuss how we may generate a test suite that is easily extensible and provides a high degree of confidence that the package has been well tested.

1 Introduction

Good software engineering practice decrees that testing be an integral activity in the design, implementation and maintenance phases of the software life cycle. Software that executes successfully on an extensive, well constructed suite of test cases provides increased confidence in the code and allows changes to and maintenance of the code to be closely monitored for unexpected side effects. An important requirement is that the test suite evolves with the software; data sets that highlight the need for changes to be made to the source code should automatically be added to the suite and new tests generated to cover newly added features.

To gauge the quality of a test suite we require quantitative measures of how well it performs. Such metrics are useful to developers in a number of ways; first, they determine when we have achieved a well-defined, minimal level of testing; second, they can reduce the amount (and, thus, the cost) of testing by allowing tests that fail to improve the metric to be discarded and, third, they can provide a starting point for a search for new test cases.

In this paper we discuss the strategy that has been used for testing the Lapack suite of linear algebra routines and we quantify the level of testing achieved using three test metrics. Lapack is unusual amongst numerical software packages in that it has been updated and extended over a substantial period of time and several public releases have been made available in that time. It is thus possible to obtain a change history of the code and thus identify maintenance fixes that have been applied to the software either to remove errors or to improve the numerical qualities of the implementation.

In section 2 we discuss the testing metrics we are using and why we would consider using more than one of these measurements. In section 3 we describe the testing strategies used by the Lapack package and discuss how the use of

the proposed testing metrics alters the way in which we test some parts of the software.

Section 4 presents our results and our conclusions are given in section 5.

2 Software Testing Metrics

Although some work has been done on the application of formal methods to provide the basis for correctness proofs of linear algebra software and formal derivation of source code (see, for example, Gunnels [5]) the vast majority of numerical software in day-to-day use has been produced by more traditional styles of software development techniques that require extensive testing to increase confidence that it is functioning correctly.

One of the most potent and often underused tools available to assist with this process is the compiler. Most modern compilers will perform a variety of static checks on the source code for adherence to the current language standard (for example, ANSI Fortran [9] and ANSI C [8]). Some (for example, Nag [14] and Lahey [11]) go further and provide feedback on such things as undeclared variables, variables declared but never used, variables set but never used, etc. Although these often only require cosmetic changes to the software in some cases these messages are flagging genuine errors where, for example, variable names have been misspelt.

Furthermore, with judicious use of compilation options, compilers will now produce executable code that performs extensive run-time checks that would be exceedingly difficult to perform in any other way. Almost all will check that array indexes are within their declared bounds but many will go further and check, for example, that variables and array elements have been assigned values before they are used. Such checks are useful even where correctness proofs are available, since simple typographical errors are common wherever human editing is involved in the implementation or maintenance processes.

While it is true that such checking imposes a run-time overhead that would make the executable code produced unusable for many applications where high-performance is essential, it should be mandatory that these capabilities are used during the development, testing and maintenance stages of any software project.

Thus our first requirement is that all the software compiles without any warning messages and that no run-time errors are reported when running the test suite and all the possible execution-time checks have been enabled for the compilation system being used. For such a requirement to have teeth, we must have confidence that the test suite thoroughly exercises the source code.

In order to determine the effectiveness of the testing phase, quantitative measurements are required. Most of the basic testing metrics are white-box and aim to measure how well the test data exercises the code. The simplest is to calculate the percentage of basic blocks (linear sections of code) that are executed. While 100% basic-block coverage is considered by many experts [10] to be the weakest of all coverage criteria its achievement should always be regarded as an initial

goal by testers. After all how can we have confidence that software will perform as required if blocks of statements are never executed during the testing process?

Confidence in the software can be further enhanced by extending the testing process to provide more stringent forms of code coverage, for example, branch coverage and linear code sequences and jumps (LCSAJ).

For branch coverage we attempt to generate data sets that will ensure that all branches within the code are executed. For example, in a simple **if-then-endif** block we seek data that will cause the condition to be both true and false. We note here that 100% branch coverage implies 100% basic-block coverage. Thus in the example above basic-block coverage would only require the test to be true in order to execute the statements within the **if**; branch coverage testing would require data to be found that causes the test to be false as well.

The LCSAJ metric measures coverage of combinations of linear sequences of basic blocks, on the grounds that computational effects in a single basic block are likely to have impacts on the execution behavior of immediately following basic blocks. Errors not evident from the analysis of a single basic block may be revealed when executed in combination with other basic blocks, in the same way that module integration testing often reveals further errors beyond those revealed by individual module testing. However, since some calculated LCSAJ combinations may be infeasible, for instance if they involve contradictory condition values, a requirement of 100% LCSAJ coverage is unreasonable and a more modest figure such as 60% would be more approachable.

For the testing metrics described above the amount and quality of test data required to obtain a determined percentage of coverage will generally grow as we progress from basic-block coverage, through branch coverage to LCSAJ testing. Even more thorough testing procedures are available, for example, mutation testing [15] and MC/DC [6]. We do not consider these in this paper.

To be able to obtain metrics data requires tool support. Some compilers provide the ability to profile code execution (for example Sun Fortran 95 Version 7.1 [16] and the NagWare tool *nag_profile* [13]) although, typically, this only provides basic-block coverage.

Tools which perform more sophisticated profiling tend to be commercial; we are using the LDRA Fortran Testbed [12] to generate basic-block and branch coverage and LCSAJ metrics.

3 Testing Lapack

The Lapack suite [1] consists of well over a thousand routines which solve a wide range of numerical linear algebra problems. In the work presented here we have restricted our attention to the single precision real routines thus reducing the total amount of code to be inspected to a little over 25% of the total.

The first release of Lapack was in 1992. Part of the intention of the package's designers was to provide a high quality and highly-portable library of numerical routines. In this, they have been very successful. Since its early releases, which were written in Fortran 77, the software has been modernized with the release

of Lapack 95 [2]. Lapack 95 took advantage of features of Fortran 95, such as optional parameters and generic interfaces, to provide wrapper routines to the original Fortran 77 code. This exploited the fact that parameter lists of some existing routines were effectively configured by the types of their actual parameters, and that calls to similar routines with different precisions could be configured using this type information. These wrappers provided the end user with improved and more robust calling sequences whilst preserving the investment in the underlying, older code. Finally, while the injudicious use of wrapper routines may cause performance overheads, this is unlikely to be a problem for the high level user callable routines in Lapack.

Comprehensive test suites are available for both the Fortran 77 and Fortran 95 versions of the package and these have provided an excellent starting point for the focus of our work on the testing of the package. Indeed a core component of the Lapack package is its testing material. This has been an important resource in assisting with the implementation of the library on a wide variety of platform/compiler combinations. However, little, if any, measurement appears to have been made of how well the testing material actually performs from a software engineering sense.

Finally, a package of this size and longevity is potentially a valuable source of data for the evaluation of software quality metrics. Over the past 12 years there have been eight releases of Lapack and this history has allowed us to track the changes made to the software since its initial release. This has provided us with, among other things, data on the number of faults corrected and this has enabled us to investigate whether software quality metrics can be used to predict which routines are liable to require future maintenance (see [3] for further details). It has also been possible to determine whether the associated test suites have been influenced by the fault history.

The main purpose of the supplied testing material is to support the porting process. An expectation of this process is that all of the supplied tests should pass without failures. Following on from the work by Hopkins [7], we were keen to explore how safe this expectation was, by using state-of-the-art compile-time and run-time checks. We began by executing all the supplied test software using a variety of compilers (including NagWare Fortran 95, Lahey Fortran 95 and Sun Fortran 95) that allowed a large number of internal consistency checks to be performed. This process detected a number of faults in both the testing code and the numerical routines of the current release, including accessing array elements outside of their declared bounds, type mismatches between actual arguments and their definitions and the use of variables before they have been assigned values. Such faults are all non-standard Fortran and could affect the final computed results. In all, some 50 of the single precision real and complex routines were affected.

The test-rigs provided are monolithic in that each generates a large number of datasets and routine calls. The number and size of the datasets used are controlled by a user supplied configuration file with the majority of the data being generated randomly and error exits tested separately (but still inside the

monolithic drivers and outside of user control). We used the default configuration file in our analysis. Results obtained from each call are checked automatically via an oracle rather than against predefined expected results. This approach has a number of disadvantages

1. there is no quantitative feedback as to how thorough the actual testing process is,
2. running multiple datasets in a single run masks problems that could be detected easily (for example, via run-time checking) if they were run one at a time,
3. the use of a large number of randomly generated datasets is very inefficient in that, from the tester's point of view, the vast majority are not contributing to improving any of the test metrics.

Having removed all the run-time problems mentioned above our next goal was to reduce the number of tests being run (i.e., the number of calls being made to Lapack routines) by ensuring that each test made a positive contribution to the metrics being used. In order to preserve the effort already invested in testing the Lapack routines it was decided to extract as much useful data as possible from the distributed package. To do this we modified *nag_profile* [13] so that it generated basic-block coverage data (profiles) for each individual call to an Lapack routine. We discovered that, for most of the routines, we could obtain the same basic-block coverage using only a small fraction of the generated test sets; see Section 4 for details. Scripts were constructed to extract relevant datasets and to generate driver programs for testing each routine. In order to keep these driver programs simple we only extracted legal data, preferring to run the error exit tests separately; this ensured that such tests were generated for all the Lapack routines.

The routines within the Lapack package split into two distinct subsets; those that are expected to be called directly by the user of the package and those that are only envisioned as being called internally and not described in the user manual. The testing strategy currently employed by the Lapack 77 test suite actually goes further and creates first and second class user callable routines. For example, the testing of the routine SGBBRD is piggy-backed onto the data used in the testing of a more general routine. This leads to 13 out of the 111 basic blocks in SGBBRD being untested; of these 11 were concerned with argument checking for which no out-of-bounds data were provided as it was not treated as a first class citizen — such data had already been filtered out by its caller. The other two were in the main body of code. The data used by the test program was generated from a standard data file that defines the total bandwidth. This routine also requires the number of sub- and super- diagonals to be specified and this pair of values is generated from the total bandwidth value. Unfortunately the method fails to produce one of the special cases.

However there is a deeper testing problem revealed by this example. When testing we usually generate data which constitutes a well-defined and well documented, high level problem (for example, a system of linear equations), we call

the relevant solver and then check the computed results. We assume that it is relatively straightforward to perform checking at this level of problem definition. But, having run enough data sets to obtain 100% basic-block coverage of the user callable routine we often find that one of the internally called routines has a much lower coverage. The problem now for the tester is *can they actually find high level user data that will provide the required coverage in the internal routines?* In many cases the answer will be *no*. For example, when using many of the level 1 Blas routines the only possible stride value may be one; no data to the higher level routines will then test the non-unit stride sections of the Blas code.

Where it is impossible for user callable routines to fully exercise internal routines, there are two possibilities. Either require full coverage to be demonstrated through unit tests that are independent of higher level calls, or supplement the higher level calls with additional unit test data for the lower level calls. The former approach is less fragile than the latter, in that it retains complete coverage even in the face of changes to the data sets providing coverage of the higher level routines. On the other hand, it conflicts with the goal of trying to minimise duplicate and redundant coverage.

Over the course of its history, Lapack has undergone a number of revisions, which have included the usual modifications typical of a package of this size: addition of new routines, bug fixes, enhancements of routines, minor renaming, commentary changes, and so on. Somewhat surprising is that the test coverage does not always appear to have been extended to reflect these changes. For instance between version 2.0 and 3.0, several changes were made to SBDSQR, such as the section concerned with QR iteration with zero shift, but many of these changes remain untested by the suite, as they were before the modification.

4 Results

Out of the 316 single precision routines (s*.f) executed when running the Lapack 77 testing software 14,139 basic blocks out of a total of 16,279 were executed. This represents a basic-block coverage metric of 86.9% which is extremely high for numerical software; many packages tested achieve values closer to 50%. Of the approximately 2,000 unexecuted blocks many are due to the lack of coverage of argument checking code. However, there are still a substantial number of statements concerned with core functionality that require data sets to be provided at a higher problem definition level.

Ideally we would like each test to count; each test should cause basic blocks to be executed that no other test exercises. To obtain some idea of the possible degree of redundancy of the supplied tests we looked at individual execution profiles. We define a profile as a bit string that shows the basic blocks executed by each call to a routine; i.e., the string contains a one if the block has been executed and zero if it has not. Note that we cannot extract any path information from this.

The Lapack 77 test cases cause almost 24 million subroutine calls to be made (note that this does not include low level routines like LSAME which was called over 150 million times.) The most called routine was SLARFG which accounted for over 1.7 million calls. This routine illustrates the difficulty in obtaining complete coverage as a side effect of calling higher level routines in that out of the 12 basic blocks that make up this routine one still remained unexecuted.

More telling was that out of more than 20 million profiles obtained less than ten thousand were distinct. Note that it is not the case that we can reduce the number of subroutine calls down to 10,000 since repeated profiles will occur in higher level routines to generate different profiles within lower level routines and vice versa. However it is clear that we should be able to reduce the total number of tests being performed substantially without reducing basic-block coverage. In SLARFG only 3 distinct profiles were generated.

SSBGVD provides a good illustration of the value of the combined compile-time and run-time approach we have been discussing. It has only 2 distinct profiles from the Lapack 77 test cases, and analysis of the basic-block coverage reveals that neither covers cases where N has a value greater than one. However, the Lapack 95 tests do exercise those cases and reveal a work-array dimension error because of the incorrect calculation of LWMIN.

The original 1.3 million data sets generated were reduced to only 6.5K, producing an equivalent basic-block coverage of the top level routines. The data extraction process was not perfect in that we only extracted data which generated unique execution profiles for the user callable routines. We would thus expect a reduction in the overall coverage due to the fact that two identical profiles to a high level routine generated distinct profiles at a lower level.

Having instrumented the Lapack routines using LDRA Testbed the driver programs were run using all the extracted datasets. Our experience of using this package was that the initial effort of instrumenting and running the bulk of the reduced data to analyse the execution histories took several overnight runs on a 1GHz Pentium 3 with 256K RAM. Adding new datasets is then relatively cheap taking just a few minutes per dataset. We obtained the following coverage metric values with the reduced data sets

1. basic-block coverage: 80%
2. branch coverage: 72%
3. LCSAJ: 32%

These results show that we have lost about 7% coverage of the basic blocks in the secondary level routines. At this level it is worthwhile upgrading our extraction process in order to associate the secondary level profiles with the top level data sets that generated them. This will allow the checking of results to take place at a higher level. It is still most likely that we will not be able to achieve 100% basic-block coverage without making independent calls to second level routines, although, in this case, there is no way of knowing whether such calls are possible using legal data at the user callable routine level. Additionally it is possible that some paths and blocks of code can never be executed; in some instances it may be possible to identify dead code and remove it. However there may be

circumstances where unexecuted code is the result of defensive programming and in these cases the statements should certainly not be removed. It would be useful (and possibly interesting) if the user could be prompted to submit data causing execution of these sections of code for inclusion in the test data.

The achieved branch coverage is lagging a further 8% behind the basic-block coverage and the LCSAJ metric is well below the recommended level of 60%. Further work is needed with individual routines where LCSAJ coverage is particularly low from the reduced datasets in order to determine whether it is significantly better with the full complement of randomly generated datasets.

While most of the work we have reported here has focussed on the Lapack 77 testing strategy, we are also actively looking at the Lapack 95 test routines in order to compare the coverage they provide. It is interesting to note that the addition of compile-time and run-time checks to the test routines and example programs supplied with that version of the package reveal many problems simlar to those thrown up with the 77 version: unchecked use of optional arguments (SGELSD_F90); deallocation of unallocated space (SSPGVD_F90); use of an incorrect-precision routine (SCHKGE); incorrect array sizes (LA_TEST_SGBSV, SERRGE and many others); use of literals as INOUT parameters (SERRVX); use of unset OUT parameters (LA_TEST_SGECON); incorrect intent (LA_TEST_SGELSS); incorrect slicing (LA_TEST_SGESDD); floating-point overflow (LA_TEST_SPPSV).

5 Conclusions

Particular testing strategies should be chosen to fit specific goals and it is important to appreciate that the existing test strategy of the Lapack suite is strongly influenced by the desire to check portability rather than code coverage. A feature of the existing testing strategy is to batch up tests in two ways

- using multiple data sets on a single routine
- using similar data on multiple routines.

It was discovered that the very act of batching up tests allowed some errors to be masked — typically through uninitialised variables used in later tests having been set by earlier tests. Thus, while it requires far more organization, there are very definite advantages to be gained from running each test separately. An approach using separate tests also supports the incremental approach to improving testing metrics along with the introduction of additional tests whenever a routine is enhanced, or a new error is discovered and then corrected. Such additional tests serve as regression tests for future releases.

The need for dealing with a large number of test cases has led us to develop a more flexible testing environment which allows for the easy documentation and addition of test cases while keeping track of the code coverage obtained. Ideally, we would aim to extend the existing test suite in order to achieve 100% basic-block coverage as well as increasing the metric values obtained for branch and conditional coverage. Our major problem at present is finding data that will exercise the remaining unexecuted statements. The calling structure of the

numerical routines is hierarchical and, in many cases it is clear that unexecuted code in higher level routines would only be executed after the failure of some lower level routine. It is not clear whether such situations are actually known to occur or if the higher level code is the result of defensive programming. In the former case it might be useful to issue a message encouraging the user to submit such data to the project team. While defensive programming practices are obviously not to be discouraged, they can potentially confuse the picture when considering how thorough a test suite is in practice and defensive code should be well documented within the source code.

We have shown how the use of software testing metrics may be used to provide a quantitative measure of how good the testing process actually is as a confidence boosting measure of program correctness.

We have set up a framework for testing the Lapack suite of routines in terms of a highly targetted reduced collection of datasets. Even so we are still some way from achieving 100% basic-block coverage which is considered to be the weakest coverage metric. Indeed Beizer [4] has argued that even if complete branch coverage is achieved then probably less than 50% of the faults left in the released software will have been found. We will be endeavouring to increase the basic-block coverage as well as improving the condition and branch coverage and LCSAJ metrics.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK: users' guide*. SIAM, Philadelphia, third edition, 1999.
2. V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Waśniewski, and P. Yalamov. *LAPACK95: Users' Guide*. SIAM, Philadelphia, 2001.
3. D.J. Barnes and T.R. Hopkins. The evolution and testing of a medium sized numerical package. In H.P. Langtangen, A.M. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, pages 225–238. Springer-Verlag, Berlin, 2000.
4. B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, New York, US, 1984.
5. John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.
6. Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, Langley Research Center, Hampton, Virginia 23681-2199, May 2001.
7. Tim Hopkins. A comment on the presentation and testing of CALGO codes and a remark on Algorithm 639: To integrate some infinite oscillating tails. *ACM Transactions on Mathematical Software*, 28(3):285–300, September 2002.
8. ISO, Geneva, Switzerland. *ISO/IEC 9899:1990 Information technology - Programming Language C*, 1990.

9. ISO/IEC. *Information Technology – Programming Languages – Fortran - Part 1: Base Language (ISO/IEC 1539-1:1997)*. ISO/IEC Copyright Office, Geneva, 1997.
10. C. Kaner, J. Falk, and H.Q. Nguyen. *Testing Computer Software*. Wiley, Chichester, UK, 1999.
11. Lahey Computer Systems, Inc., Incline Village, NV, USA. *Lahey/Fujitsu Fortran 95 User's Guide*, Revision C edition, 2000.
12. LDRA Ltd, Liverpool, UK. *LDRA Testbed: Technical Description v7.0*, 2000.
13. Numerical Algorithms Group Ltd., Oxford, UK. *NAGWare Fortran Tools (Release 4.0)*, September 1999.
14. Numerical Algorithms Group Ltd., Oxford, UK. *NAGWare f95 Compiler (Release 5.0)*, November 2003.
15. A.J. Offut, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.
16. Sun Microsystems , Inc., Santa Clara, CA. *Fortran User's Guide (Forte Developer 7)*, Revision A edition, May 2002.