The Evolution and Testing of a Medium Sized Numerical Package

David Barnes and Tim Hopkins
Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, U.K.

February 18, 2000

ABSTRACT We investigate the evolution of a medium sized software package, LAPACK, through its public releases over the last six years and establish a correlation, at a subprogram level, between a simply computable software metric value and the number of coding errors detected in the released routines. We also quantify the code changes made between issues of the package and attempt to categorize the reasons for these changes.
We then consider the testing strategy used with LAPACK. Currently this consists of a large number of mainly self-checking driver programs along with sets of configuration files. These suites of test codes run a very large number of test cases and consume significant amounts of cpu time. We attempt to quantify how successful this testing strategy is from the viewpoint of the coverage of the executable statements within the routines being tested.

1 Introduction

Much time, effort and money is now spent in the maintenance and upgrading of software; this includes making changes to existing code in order to correct errors as well as adding new code to extend functionality. Some sources suggest that as much as 80% of all money spent on software goes on post-release maintenance [Hat98]. When any type of change is made, programmers need to have enough understanding of the code to be confident that any changes they make do not have a detrimental effect on the overall performance of the software. All too often a change that fixes one particular problem causes the code to act incorrectly in other circumstances.

We are interested in investigating whether it is possible, via the use of some quantitative measurements, to determine which parts of a software package are the most difficult to understand and thus, probably, the most likely to contain undetected errors and the most likely to cause further problems if subjected to code changes and updates.

Prior work in this area ([Hop96] and [Hop97b]) has shown that, for a number of rather simple codes, software metrics can be used successfully to identify problem routines in a library of Fortran subroutines. This paper extends this work by applying it to a much larger body of code which has gone through a number of revisions both to extend functionality and to correct errors. This allows us to identify where it has been necessary to make changes to the code and why, and to correlate the occurrence of errors in the software with the values of a particular software metric.

Our hope is that it will be possible to identify, prior to its release, whether a package contains subprograms which are likely to provide future maintenance problems. This would allow authors to rethink (and possibly reimplement) parts of a package in order to simplify logic and structure and, hence, improve maintainability and understandability. Our ultimate goal is to be able to improve the quality and reliability of released software by automatically identifying future problem subprograms.

LAPACK [ABB$^+$95] was an NSF supported, collaborative project to provide a modern, comprehensive library of numerical linear algebra software. It contains software for the solution of linear systems; standard, non-symmetric and generalized eigenproblems; linear least squares problems and singular value decomposition. The package serves as an efficient update to the Eispack ([SBD$^+$76] and [GBDM77]) and Linpack [DMBS79] libraries that were developed in the 1970's. The complete package is available from netlib (see http://www.netlib.org/bib/mirror.html for information as to where your nearest server is). Mostly written in standard Fortran 77 [ANS79], LAPACK uses the BLAS Level 2 [DDHH88] and Level 3 [DDDH90] routines as building blocks. LAPACK both extends the functionality and improves the accuracy of its two predecessors. The use of block algorithms helps to provide good performance from LAPACK routines on modern workstations as well as supercomputers and the project has spawned a number of additional projects which have produced, for example, a distributed memory [BCC+97] and a Fortran 90 [WD98] version of at least subsets of the complete library.

An integral part of the complete software package is the very extensive test suite which includes a number of test problems scaled at the extremes of the arithmetic range of the target platform. Such a test suite proved invaluable in the porting exercise which involved the package being implemented on a large number of compiler/hardware combinations prior to its release.

In section 2, following a short introduction to LAPACK, we provide a detailed analysis of the size of the package and the extent of the source changes made between successive versions. We also categorize all the changes made to the executable statements and obtain a count of the number of routines that have had failures fixed. We then report on a strong connection between the size of a relatively simple software code metric and a substantial fraction of the routines in which failures have been corrected.

In section 4 we look quantitatively at how well the testing material supplied with the package exercises the LAPACK code and suggest how the use of a software tool may improve this testing process. Finally we present our conclusions.

2 The LAPACK Library Source Code

The LAPACK routines consist of both user callable and support procedures; in what follows we do not differentiate between these. The source directory of the original release, 1.0, consisted of 930 files (only two files, dlamch and slamch contain more than one subprogram; the six routines in each of these files being used to compute machine parameters for the available double and single precision arithmetics). Table 1 shows how the number of files has increased with successive releases of the package along with the release dates of each version.

A number of straightforward metrics exist for sizing software, for example, the number of lines in the source files. This is somewhat crude and we present in Table 2 a more detailed view of the size of the package. The column headed 'executable' shows the number of executable statements in the entire package whilst that headed 'non-exec' gives the number of declarative and other non-executable

Version	No. of files	Release Date
1.0	930	29 February 1992
1.0a	932	30 June 1992
1.0b	932	31 October 1992
1.1	1002	31 March 1993
2.0	1080	30 September 1994

TABLE 1. Number of library source files for each released version

statements. The third column gives the total number of code statements being the sum of the previous two columns. The final two columns provide the total number of comment lines in the code and the total number of blank lines and blank comment lines.

The large number of non-executable statements is partially due to the use of the NAGWare 77 declarization standardizer [Num92] which generates separate declaration blocks for subroutine arguments, local variables, parameter values, functions, etc. This is actually no bad thing as it aids the maintenance of the code by allowing a reader to immediately identify the type and scope of each identifier. A ratio of executable to non-executable statements of 1.8 is, however, on the low side, as this implies that there are relatively small amounts of code packaged amidst large quantities of declarations which generally makes code difficult to read and assimilate.

Comments form an important part of the documentation of any software and this is especially the case for LAPACK where the description of the arguments to all procedures (both user callable and internal) is detailed enough to allow the use of the routine without the need for a separate printed manual. This accounts for the high level of commenting, around 1.5 non-blank comments per executable line. There is also heavy use of blank comment lines (or totally blank

Version	Executable	Non-exec	Lines	Comments	Blank
1.0	59684	35010	94694	143249	52581
1.0a	59954	35104	95058	143439	52679
1.0b	59897	34876	94773	142185	52243
1.1	67473	38312	105785	156077	57516
2.0	76134	41986	118120	169813	62618

TABLE 2. Statement counts by type

Version	Operators	Operands	Total	% Increase
1.0	370784	325907	696691	
1.0a	371605	326524	698129	0.02
1.0b	370928	326089	697017	−0.02
1.1	415626	364816	780442	11.9
2.0	468487	411122	879609	12.7

TABLE 3. Total number of operators and operands

lines); such lines act as an aid to readability within both the textual information and the source code.

A more detailed view of code size may be obtained by considering the operators and operands that make up the source. These are defined abstractly in Halstead [Hal77] and there appears to be no general agreement as to which language tokens are considered operators and which operands for any particular programming language. The values given in Table 3 were generated using the nag_metrics tool [Num92] which defines an operator to be

- a primitive operator, for example, $*$, $+$, $.EQ.$, $.AND.$ etc,
- a statement which counts as an operator, for example, `ASSIGN`, `IF`, `ELSE IF`, `GOTO`, `PRINT`, `READ` and `WRITE`,
- a pair of parentheses, an end-of-statement, or a comma

and operands as

- constants,
- name of variables and constants,
- strings (all strings are considered distinct from each other).

Thus the declarative part of any program is ignored by this metric as it is not considered to add to the complexity of the code.

At each new version of the package complete routines were added and deleted and changes were made to routines that were common to both the new and previous versions. Table 4 show the distribution of affected routines. Changes to program units common to successive versions have been categorized depending on whether only comments, only non-comments, or both comments and non-comments were changed. This shows that although there were textual changes

Version	Added	Deleted	Total	Changed c/only	Changed s/only	Both
1.0 →1.0a	2	0	147	75	34	38
1.0a→1.0b	2	2	339	236	33	70
1.0b→1.1	72	2	570	554	0	16
1.1 →2.0	84	6	634	279	128	227

TABLE 4. Routine changes at each version

to 1690 routines over the four revisions 1144 of these involved changes to comment lines only (this accounts for 64% of all the changed routines).

We analyzed the routines that had been changed between releases by running the two versions through the Unix file comparison tool *diff* and processing the output to count the number of changed comment and non-comment statements.

Diff classifies changes in three ways, lines in the new version that did not appear in the old (App), lines in the old version that did not appear in the new (Del) and blocks of lines that have changed between the two (Changes). Table 5 gives the totals for comment and non-comment statements according to the categories for all the changed routines at each version. It should be noted here that some changes to statements are due to changes in statement labels which occur when a label is either inserted or deleted and the code is passed through the NAGWare Fortran 77 source code formatter, nag_polish [Num92]. In a few cases diff exaggerates the number of changes due to synchronization problems which may occur if a line happens to be repeated or sections of code are moved. No attempt was made to compensate for this; indeed it may be argued that a move of a section of code should be treated as both a delete and an add. Resynchronization problems appeared to be relatively few and far between and, it was felt, they were unlikely to perturb the final results by more than a few percent. In all cases comment changes reflecting the new version number and release date have been ignored.

Finally Table 6 gives a breakdown of the non-comment and comment lines added and deleted via complete routines between releases.

Although the release notes made available with each new revision gave some details of which routines had had bug fixes applied to them this information was far from complete. It is not safe to assume that

Version	Comments			Non-comments		
			Changes			Changes
	Del	App	(old/new)	Del	App	(old/new)
1.0 →1.0a	38	134	550/484	69	312	775/808
1.0a→1.0b	258	605	2183/1573	384	593	1325/1722
1.0b→1.1	227	70	9428/8354	34	36	724/752
1.1 →2.0	541	527	4393/4688	449	730	2363/2274

TABLE 5. Interversion statement changes from diff

all non-comment code changes are necessarily bug fixes. In order to determine the nature of the changes to routines at each release, a visual inspection of each altered single precision complex and double precision real routine was conducted using a graphical file difference tool [BRW88]. As a result we have categorized the code changes between all at each revision as being one of

i: enhanced `INFO` checks or diagnostics and `INFO` bug fix (usually an extended check on the problem size parameter `N`),

pr: further uses of machine parameters (for example, the use of `DLAMCH('Precision')` in `DLATBS` at version 1.0a to derive platform dependent values),

c: cosmetic changes (for example, the use of `DLASET` in place of `DLAZRO` in the routine `DPTEQR` at version 2.0),

en: enhancement (for example, the addition of equilibration functionality to the routine `CGBSVX` at version 1.0b),

ef: efficiency change (for example, the quick return from the routine `CGEQPF` in the case when `M` or `N` is zero at version 1.1),

Version	Added		Deleted	
	Com	Non-com	Com	Non-com
1.0 →1.0a	160	86	0	0
1.0a→1.0b	99	65	160	86
1.0b→1.1	14351	10196	160	86
1.1 →2.0	13770	12248	311	129

TABLE 6. Added and deleted routines by comment and non-comment lines

re: removal of redundant code (for example, a CABS2 calculation in CGEEQU was not required and was removed at version 1.0a),

mb: minor bug (typically a few lines of changed code; for example, the changes made to DLAGS2 at version 2.0 to add an additional variable and to modify a conditional expression to use it),

Mb: major bug (a relatively large code change; for example, the changes made to the routine DSTEQR at version 2.0).

Such a classification provides a much firmer base from which to investigate a possible correlation between complexity and coding errors.

Routines in the LAPACK library are in one of four precisions; single or double precision, real or complex. While the single and double precision versions of a routine can generally be automatically generated from one another using a tool like nag_apt [Num92], the real and complex parts of the package are often algorithmically quite different. For this reason we only consider the single precision complex and the double precision real routines in the remaining sections of this paper. These routines may be identified by their name starting with either a C (single precision complex) or a D (double precision real).

3 The Path Count Metric

Table 7 provides a summary of the number of single precision complex and double precision real routines that fall into each category for successive releases of the package. We were interested in discovering whether there was any relationship between those routines needing bug fixes and any software complexity metric values. One metric, a version of Nejmeh's path count metric [Nej88], is calculated by the QA Fortran tool [Pro92]. This metric is an estimate of the upper bound on the number of possible static paths through a program unit (based solely on syntax). Note that some paths so defined may not be executable but such impossible routes cannot usually be determined by simple visual inspection. The path count complexity of a function is defined as the product of the path complexities of the individual constructions. Thus, for example, the path complexity of a simple if-then-else statement is 2 while three consecutive if-then-else statements would have an associated value of $2^3 = 8$. Three nested

	1.0a C, D	1.0b C, D	1.1 C, D	2.0 C, D
mb	9, 9	12, 9	1, 3	16,14
Mb	3, 3	1, 1	0, 0	1, 2
re	2, 0	1, 1	0, 0	0, 0
i	3, 5	8,10	1, 1	9, 9
pr	3, 4	1, 0	0, 0	3, 4
en	0, 0	5, 5	0, 0	1, 2
ef	0, 0	3, 1	1, 1	1, 1
c	1, 0	0, 1	0, 1	105,15

TABLE 7. Number of routines affected by each category of source code change

if statements would have a path count of 4. This metric provides a useful measure of the effort required to test the code stringently as well as giving an indication of code comprehensibility and maintainability. A reduction in the path count caused by restructuring code would imply the elimination of paths through the code which were originally either impossible to execute or irrelevant to the computation. This would be likely to reduce the time spent in the testing phase of the software development. An example of the use of this metric to identify problem code within a small library of relatively small Fortran routines may be found in [Hop96] and [Hop97a].

The metric value returned by the package has a maximum value of 5×10^9 although, realistically, a program unit can be classified as too complex to test fully when this value exceeds 10^5. Table 8 gives the number of routines in which faults were correlated against the value of the path metric. (We differentiate here between routines that were introduced before version 2 and those that were added at version 2 and cannot, therefore, have had source changes applied to them.) This data clearly shows a high correlation between routines identified as complex by the path count metric and those having had bug fixes applied to them. 41% of all the bugs occurred in routines with a path count metric in excess of 10^5 and these routines constitute just 16% of the total number of subprograms making up the library. The chance of a bug occurring in these routines would appear to be around 6 times more likely than in routines with a path count of less than 10^5. It should also be noted that a large number of the routines added at version 2.0 have extremely large path counts and, from

	$\lfloor \log_{10}(\text{Path Counts}) \rfloor$					%
	> 8	7	6	5	≤ 5	> 5
v2.0	5	1	0	4	26	28
<2.0	10	6	10	8	285	16
Faults	8	3	7	3	30	41
%routines	80	50	70	37	11	–

TABLE 8. Occurence of Faults against Path Count Metric

our current analysis, we would expect a high percentage of these to require patches to be applied in forthcoming releases.

4 Testing

The test suite forms an integral part of the LAPACK software package. The code was designed to be *transportable*; all new LAPACK code was to be portable while efficiency of the package as a whole was to be platform dependent. This was achieved by coding in standard Fortran 77 [ANS79] for portability of the higher level routines and using platform specific versions of the BLAS to obtain high efficiency. Thus, by using the BLAS Levels 2 and 3 definition codes ([DDHH88], [DDDH90]) the entire package may be made portable at the price of suboptimal execution speed.

Two test suites are provided with the released package; one for checking the installation and the other for producing platform dependent timings. All the tests are self checking in that the only output that a user has to check is in the form of summaries of the number of tests applied to each user routine along with the number of successes and counts and details of any failures.

Matrix data is either generated randomly or specially constructed (see [DM89] for more details). In both cases the testing software uses a metric to decide whether each computed solution is 'correct'. Test cases are also generated to exercise the routines on data at the extremes of the floating point arithmetic ranges. The number, and to a minor extent, the range of tests applied, may be controlled by the user at a data file level. However this generally precludes the user from forcing execution through specific sections of the code which usually requires specially constructed data. To all intents and purposes the test strategy must be categorized as white box (or glass box) testing, where test cases are selected by considering their effect

	Routines	Basic Blocks	% Executed
C install	258	12019	89.56
C timing	124	5324	72.84
D install	263	12852	89.20
D timing	143	6615	72.41

TABLE 9. Basic block execution for installation and timing test suites

on the code rather than just testing their adherence to the specification as is the case with black box testing.

We were interested in determining quantitatively how effective this strategy was in exercising the code. The minimal requirement of a test suite should be to ensure that all the executable statements in the package are executed at least once. Note that this is very different from path coverage as defined earlier.

We used the NAGWare 77 source code instrumenter and execution analysis tools nag_profile and nag_history [Num92] to determine cumulatively the number of times each basic block was executed. (A basic block is a straight line section of code, i.e., it does not contain any transfer of control statements.) This allowed us to determine how many basic blocks were not exercised by the test suite. We ran the instrumented code on both the installation and timing test suites.

Table 9 compares the number and percentage of basic blocks executed using both test suites.

Further analysis showed that of the 1388 unexecuted blocks, 214 (15.4%) were concerned with the checking of input arguments. The worst case was the routine, `DGEGS`, used for computing the general Schur factorization which has 48% of its basic blocks untested.

We looked in detail at the routine `DGBBRD`. This routine consisted of 111 basic blocks containing 124 executable statements. The installation test omitted to cover 13 basic blocks of which 11 were concerned with checking the consistency of the input arguments. The final two were in the main body of the code.

The code operates on a banded, rectangular ($M \times N$), matrix where the user provides the number of sub- and super-diagonals, KL and KU respectively. The configuration file used to provide data for this routine specified M, N and K, the total bandwidth. Code within the test routine then splits K into KL and KU. Although a total of 1500 calls are made to `DGBBRD` the test data failed to generate the special case $M = N = 2$, $KU = 0$ and $KL > 0$. The problem was that with M and N both greater than zero the code to set KL and KU could not generate $KU = 0$ and $KL > 0$. Providing data for a separate test is straightforward and the two previously untested blocks executed successfully.

As well as using the profiling information from nag_profile to identify the basic blocks of code that were not being executed by the test data, we can also use it to check that individual tests (or batches of tests) actually contribute to the overall statement coverage. Ideally we would like to minimize the number of test cases being run to obtain maximum coverage. We note here that it may not be possible to exercise 100% of the basic blocks, for example, there may be defensive code that exits if convergence is not attained although no numerical examples are known which trip this condition.

Using `DGBBRD` again as our example, we obtained individual basic block coverage profiles for each of the 20 M and N pairs (each pair generates a number of calls to the routine for different KL and KU values). It was found that two of these 20 tests covered 83 of the 98 executed blocks and four could be chosen to cover all 98. The effect of using just this minimal number of tests was to reduce the execution time for testing this routine by a factor of four without any loss of code coverage. A further reduction in the execution time could be made by reducing the number of (KL, KU) pairs chosen for a given (M, N) pair.

The above analysis could be applied to all the test drivers in order to reduce the total number of tests being executed whilst maximizing the code coverage.

5 Conclusion

We have analyzed the source code changes made between successive versions of the LAPACK software library and we have presented strong evidence that the path count software metric is a good indicator of routines that are likely to require post release maintenance. We note that many of the newly introduced routines have metric values indicative of problem code.

By using a profiling tool we have been able to measure how well the installation testing software, provided with the package, executes the LAPACK sources. The coverage is extremely good although we believe that it could be improved further. In addition we have shown that, from the point of view of code coverage, many of the tests do not contribute as they fail to exercise any blocks of code that are not already executed by other tests. We are certain that by analyzing the output from this tool it would be possible both to reduce the number of tests necessary to obtain the attained code coverage and to improve the coverage by pinpointing untested sections of code.

One of the most worrying trends is that there appears to be a definite trend towards very complex routines being added to the library. Of the new routines introduced at version 2.0 almost 30% had path counts in excess of 10^5; this is almost double the percentage of routines introduced prior to that version. Whilst it may be argued that new routines are solving more complex problems it is possible to structure these codes so that the components are far simpler from a software complexity viewpoint and are thus much easier to test thoroughly. It is highly likely that over half of the routines with a path count of 10^5 or more will require bug fixes in the near future.

It is difficult to compare the quality of LAPACK with other libraries of numerical software since, as far as we know, no other public domain numerical package has either a complete source code change history or a complete set of the sources of all the relevant releases available. That apart we believe that LAPACK is a high quality software package with a bug fix being applied on average for approximately every 600 executable statements.

A similar complexity analysis to the one performed in this paper could be applied to any Fortran 77 software; extracting the required metric values is very simple via QA Fortran, and the NagWare 77 profiling tool allows a relatively painless analysis of the statement coverage. Far more difficult is the automatic generation of test data; there appear to be no tools available, at least not in the public domain, that would help in this area. Generating test drivers and data to ensure a high percentage of statement coverage is thus both difficult and time consuming.

The path count metric detailed in section 3 could be used in a similar way for both C and Fortran 90 and indeed for almost all imperative languages. For object oriented languages a different approach would be necessary and there is not as yet any general consensus on which metrics are most appropriate. A general discussion of object oriented software metrics may be found in Lorenz and Kidd [LK94].

It is our intention to extend the work presented in this paper to investigate the detection of errors in released software. We propose using the path count metric to identify possible problem routines and to subject these routines to extensive white box testing in an attempt to exercise as many paths through the code as possible. Current work ([TCM98b], [TCM98a] and [TCMM98]) provides some hope of generating data to exercise specified paths through code and this would certainly open up new possibilities for fault detection.

6 References

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK: users' guide.* SIAM, Philadelphia, second edition, 1995.

[ANS79] ANSI. *Programming Language Fortran X3.9-1978.* American National Standards Institute, New York, 1979.

[BCC+97] 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, Phiadelphia, 1997.

[BRW88] David Barnes, Mark Russell, and Mark Wheadon. Developing and adapting UNIX tools for workstations. In *Autumn 1988 Conference Proceedings*, pages 321–333. European UNIX systems User Group, October 1988.

[DDDH90] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990.

[DDHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1):18–32, March 1988.

[DM89] J. Demmel and A. McKenney. A test matrix generation suite. Technical Report MCS-P69-0389, Argonne National Laboratories, Illinois, March 1989.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK: Users' Guide*. SIAM, Philadelphia, 1979.

[GBDM77] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51 of *Lecture notes in computer science*. Springer-Verlag, New York, 1977.

[Hal77] M.H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier, New York, 1977.

[Hat98] Les Hatton. Does OO sync with how we think? *IEEE Software*, pages 46–54, May/June 1998.

[Hop96] T.R. Hopkins. Restructuring software: A case study. *Software — Practice and Experience*, 26(8):967–982, July 1996.

[Hop97a] T.R. Hopkins. Is the quality of numerical subroutine code improving? In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 311–324. Birkhäuser Verlag, Basel, 1997.

[Hop97b] T.R. Hopkins. Restructuring the BLAS Level-1 routine for computing the modified Givens transformation. *ACM SIGNUM*, 32(4):2–14, October 1997.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Object-Oriented Series. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Nej88] B. A. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188–200, 1988.

[Num92] Numerical Algorithms Group Ltd., Oxford, UK. *NAGWare f77 Tools*, second edition, September 1992.

[Pro92] Programming Research Ltd, Hersham, Surrey. *QA Fortran 6.0*, 1992.

[SBD+76] 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*, volume 6 of *Lecture notes in computer science*. Springer-Verlag, New York, second edition, 1976.

[TCM98a] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis*, volume 23, pages 73–81. ACM/SIGSOFT, March 1998.

[TCM98b] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.

[TCMM98] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, October 1998.

[WD98] J. Wasniewski and J.J. Dongarra. High performance linear algebra package – LAPACK90. Technical Report CS-98-384, University of Tennessee, Knoxville, April 1998.

A Availability of Tools

The main software tools mentioned in the paper are available as follows:

QA Fortran: Programming Research Limited, 1/11 Molesey Road, Hersham, Surrey, KT12 4RH, UK.
(http://www.prqa.co.uk/qafort.htm)

NagWare 77: NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK.
(http://www.nag.co.uk/nagware/NANB.html)

Perl: http://www.perl.com/pace/pub.

Vdiff: A graphical file comparator, Kent Software Tools.
(http://www.cs.ukc.ac.uk/development/kst/).

Other analysis of the code reported in the paper was performed using bespoke perl scripts.