Is the Quality of Numerical Subroutine Code Improving?

T. R. Hopkins[1]

ABSTRACT We begin by using a software metric tool to generate a number of software complexity measures and we investigate how these values may be used to determine subroutines which are likely to be of substandard quality.
Following this we look at how these metric values have changed over the years. First we consider a number of freely available Fortran libraries (Eispack, Linpack and Lapack) which have been constructed by teams. In order to ensure a fair comparison we use a restructuring tool to transform original Fortran 66 code into Fortran 77.
We then consider the Fortran codes from the Collected Algorithms from the ACM (CALGO) to see whether we can detect the same trends in software written by the general numerical community.
Our measurements show that although the standard of code in the freely available libraries does appear to have improved over time these libraries still contain routines which are effectively unmaintainable and untestable. Applied to the CALGO codes the metrics indicate a very conservative approach to software engineering and there is no evidence of improvement, during the last twenty years, in the qualities under discussion.

1 Introduction

The last two decades have witnessed dramatic advances in the way we view the creation of software. Since *software engineering* was born in 1969 [BR70] an ever growing number of techniques have been put forward in an attempt to promote software production from an art to a science. For almost as long attempts have been made to quantify the quality of software. This has led to a bewildering number of software metrics being proposed; an excellent review of many of these may be found in the book by Zuse [Zus91]. In addition, a number of software tools have appeared which compute a variety

[1] Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E-mail: trh@ukc.ac.uk

This paper appears in *Modern Software Tools for Scientific Computing*, ed. Arge, Bruaset, and Langtangen, Birkhauser Boston, 1997, ISBN 0-8176-3974-8. Reprinted with permission.

of metrics for code written in most of the commonly used programming languages. Such tools include QAFortran [Pro92], QAC and QAC++, the nag_metrics tool from the NagWare f77 tool suite [Num92] and the LDRA Testbed [LDR].

In this paper we use a small number of these metrics to investigate whether we can detect an improvement in the quality of numerical software written in Fortran over the last twenty years. In section 2 we briefly discuss these metrics and, in the following section, we look at how successful they are at identifying software modules that suffer from readability, maintainability and testability problems.

We use the QAFortran tool to generate these metrics for the routines in a number of public domain packages. We also note, in this section, that during the time spanned by these libraries, the Fortran language has evolved from Fortran 66 [ANS66], through Fortran 77 [ANS79] to Fortran 90 [ISO91]. The additional control structures made available with each new code standard means that it is unfair to make direct comparisons of the code metrics between packages without taking into account the version of Fortran being used. In order to 'normalize' the metrics we have only considered code written in Fortran 66 and Fortran 77 and we have used an automatic code restructurer, *spag* [Pol93], to translate from Fortran 66 to Fortran 77. Metrics from the restructured Fortran 66 codes may then be compared.

The packages used in section 3 have all been produced by teams of researchers which have included wide ranges of expertise from software engineers to numerical analysts. To try to gauge how the quality of scientific software has varied among the community in general we consider the Collected Algorithms from the ACM (CALGO). These codes have been published since 1960 and are widely regarded as being state-of-the-art both in algorithmic and coding terms at the time of publication. In section 4 we report on how our chosen metric values have changed with time by considering the Fortran routines that have appeared in CALGO since 1975.

Finally, in section 5, we draw some conclusions from the results we have presented.

2 Software Metrics

In addition to the number of executable *L*ines *O*f *C*ode (LOC) and the number of explicit GOTO statements in a subprogram, we will also consider three further metrics: cyclomatic complexity [McC76], knot count [WHH79] and a variant of the path count metric proposed by Nejmeh [Nej88].

It should be noted that the metrics used in this paper have been chosen to measure qualities of Fortran code although some of them may also be applied successfully to other procedural languages. Object oriented languages require a different approach and there appears to be, as yet, no general consensus on which metrics are most appropriate. A general discussion of object oriented software metrics may be found in Lorenz and Kidd [LK94].

Cyclomatic Complexity

The control graph of a program unit is a directed graph whose nodes are the basic blocks of code and whose edges are directed arcs corresponding to the flow of control between the basic blocks. A basic block is a section of code which contains no transfer of control (for example, a sequence of assignment statements). The cyclomatic complexity, $V(G)$, is defined to be the cyclomatic number of the control graph, i.e.,

$$V(G) = \text{Number of edges} - \text{Number of nodes} + 1$$

This may be shown to be equivalent to one more than the number of predicates (decision statements) used in the code and hence, we believe, it is a good indicator of the complexity of the underlying algorithm. This metric was extended by Myers [Mye77] who suggested the use of a cyclomatic complexity interval to take account of the additional complexity caused by compound predicates. The lower bound of Myer's interval is the cyclomatic complexity and the upper bound is defined as one more that the total number of conditions appearing in the code.

Cyclomatic complexity was originally advocated both as a measure of the testing effort required for a module and as an effective way of dividing software into subroutines. Shepperd [She88] and Shepperd and Ince [SI94] have questioned the use of the metric for measuring testing effort as it appears to be extremely insensitive to the structure of the software. They state that this may be due to the fact that the measure is based on a lexical rather than a structural view of the code.

In the present paper the metric is used as an indicator that a piece of code would probably benefit from being broken down into a number of simpler program units. McCabe [McC76] suggested a maximum value of 10 for an individual program unit while Grady [Gra94], after analyzing the relationship between the cyclomatic complexity and the number of updates required to each module in 830,000 lines of Fortran code, suggests a maximum value of 14.

Knot Count

A knot occurs in a piece of code whenever the paths associated with two transfers of control intersect (see Figure 1 for two examples). Code with large knot counts is generally extremely difficult to read and understand. The number of knots is, therefore, a good indicator of code clarity.

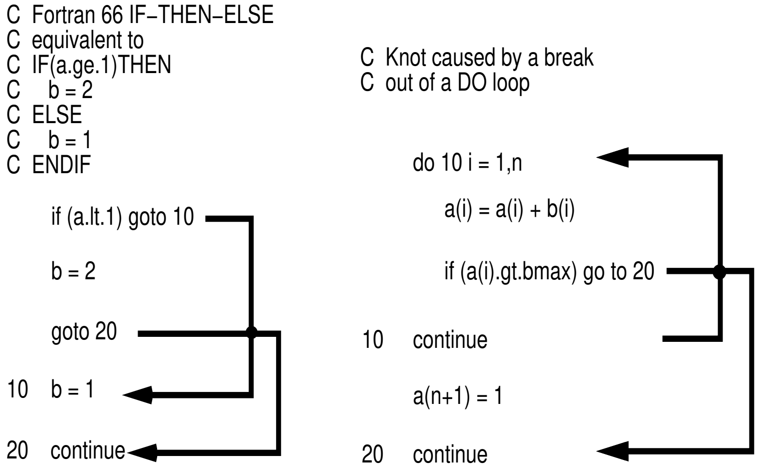

FIGURE 1. Examples of knots in Fortran code

Path Count

The path count metric provides an upper bound on the number of possible static paths through the code and thus gives an estimate of the amount of effort required to thoroughly test a routine. It should be noted that the path count is calculated solely from the syntax of the code; some paths may be counted which are not executable. When computing path counts QAFortran will return a maximum value of 5×10^9.

Using a large library of numerical Fortran subroutines, Hatton and Hopkins [HH88] reported a strong link between very large path counts and the number of defects reported in routines after they had been released. Thus the path count metric may also be regarded as an indicator of the reliability of program units.

3 A Comparison of Freely Available Packages

Table 1.1 shows the values obtained for these metrics, along with the number of executable and explicit GOTO statements, for the single precision level 1 BLAS [LHKK79]. The routines which stand out, SNRM2 and SROTMG, both exhibit a very complicated code structure. Indeed Hanson and Krogh [HK87] remark that they did not translate SROTMG into assembler as part of their INTEL specific version of the level 1 BLAS due to the complexity of its specification.

It has been shown by Hopkins ([Hop96b] and [Hop96a]) that even using Fortran 66 the knot and path counts for these routines may be substantially reduced by more careful structuring of the code. This leads in both cases

Routine	LOC	GOTOs	CC	Knots	Paths
isamax	22	3	8	3	8
sasum	22	3	8	4	8
saxpy	29	2	11	2	16
scopy	31	2	10	2	16
sdot	29	3	10	4	32
snrm2	44	15	13	41	1024
srot	22	1	7	1	8
srotg	22	2	5	2	16
srotm	84	7	13	17	144
srotmg	121	32	18	92	98304
sscal	22	2	8	2	8
sswap	37	2	10	2	16

TABLE 1.1. Metric values for the real level 1 BLAS

to codes that are easy to read and understand. In the case of SROTMG the restructuring exercise also discovered a minor error in the code.

The cyclomatic complexity intervals of these two routines changed from 13:14 to 11:16 for SNRM2 and from 18:18 to 13:17 for SROTMG, showing that, although restructuring led to fewer conditional statements being used, more of the resulting logical expressions were compound. Basically the underlying algorithms for these routines require a number of conditional tests to be made and restructuring has little or no effect on this.

The metric values obtained for the newer Fortran 77 level 2 [DDHH88] and level 3 [DDDH90] BLAS show a distinct improvement. There are no GOTO statements (and therefore no knots) in any of the routines and the highest path counts are all less than 10^4. These large path counts are almost invariably associated with routines which carry out a large number of input parameter checks and do not, therefore, reflect complexity in the numerical part of the code.

Eispack (see [SBD+76] and [GBDM77]) was one of the first attempts to create a library of high quality, freely available, numerical routines. Many of the codes were Fortran 66 versions of the Algol 60 procedures presented in Wilkinson and Reinsch [WR71]. Figure 2 shows a scatter plot of knots against number of executable statements for the 75 double precision routines along with the regression line through this data. We note that there is a very strong correlation between the number of knots and the number of executable statements in a routine and this may be used to identify three routines which exhibit unusually high knot counts for their length; BISECT, TRIDIB and TSTURM.

Generating similar figures for the double precision real and complex versions from two other similar projects, Linpack [DMBS79] (91 routines) and Lapack [ABB+95] (542 routines), allow further subprograms to be identified which have unusually high knot counts for their size. Table 1.3 gives

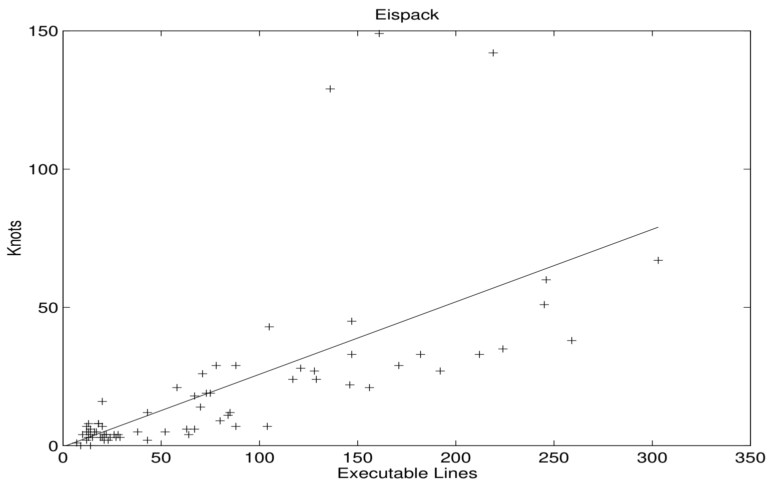

FIGURE 2. Scatter plot and regression line for double precision EISPACK routines

a summary of the routines with large standardized residuals; these are the routines which the Minitab statistical package [RJR92] flags as being outliers and thus worthy of further investigation. In the cases given the standardized residual values are extremely high.

In Table 1.2 we present the slope of the regression line for the Eispack, Linpack and Lapack packages. This indicates a general improvement in the readability of code over time. It should be expected that the Lapack value would be the smallest since this is the only one of the three packages to have been written in Fortran 77. Although both written in Fortran 66, the Linpack routines are generally better structured and therefore more comprehensible than the Eispack routines; a fact reflected in the smaller slope value.

Fortran 77 provided one major new control structure that was missing from Fortran 66; the block IF. Using Fortran 66 to simulate a simple IF-THEN-ELSE construction required the use of a pair of GOTO statements and resulted in a knot appearing in the code (see Figure 1). In order to make a fair comparison between the quality of Fortran 66 and Fortran 77 software it is necessary to 'normalise' the codes in some way to take account

Library	Slope of Regression Line
Eispack	0.262
Linpack	0.145
Lapack	0.048

TABLE 1.2. Slopes of regression lines for knots against executable statements

Routine		LOC	Knots	Residual
Eispack				
	BISECT	136	129	4.43
	TRIDIB	161	149	5.09
	TSTURM	219	142	4.11
Linpack				
	DSVDC	285	54	3.18
	ZSVDC	310	58	3.35
Lapack				
	DLARFX	440	132	11.23
	ZLARFX	440	132	11.23
	ZHGEQZ	340	60	4.46

TABLE 1.3. Routines identified as difficult to read/maintain due to the large knot count relative to the number of executable statements

of the availability of the block IF.

A number of code restructuring tools are available which translate between Fortran 66 and Fortran 77. These tools are based on the work of Baker [Bak77] and will, amongst other changes, create block IFs from equivalent Fortran 66 code. The *spag* restructuring tool [Pol93] was used here for this purpose.

Table 1.4 shows the effects of using *spag* on all the Eispack, Linpack and Lapack codes detailed in section 2. As well as showing the aggregate values for the number of executable lines of code, GOTO statements and knots for all the routines considered from each package, we also provide the maximum value, in any routine, of each of these metrics along with the maximum value of the cyclomatic complexity (CC). Note that each of these maximum values may be associated with a different routine.

For the Eispack routines only the number of executable statements has increased during restructuring due to *spag* duplicating small blocks of code

	Aggregate			Routine Maxima			
	LOC	GOTO	Knots	LOC	GOTO	Knots	CC
Eispack							
f66	5774	863	1482	303	50	149	76
spag	6482	298	807	337	25	116	76
Linpack							
f66	7381	1022	1003	310	50	58	86
spag	7250	139	226	300	11	27	85
Lapack							
f77	52001	993	1972	1159	26	132	199
spag	51101	672	1469	1145	21	61	199

TABLE 1.4. Effects of using the *spag* restructurer on routines from the Eispack, Linpack and Lapack libraries

in order to obtain better code structure. Although the number of GOTO statements has been substantially reduced the number of knots has only decreased to just over a half of the original number, pointing to substantial amounts of unstructured code. Of the remaining 807 knots 268 occur in the three routines, BISECT, TRIDIB and TSTURM, identified above as possible problem routines.

Automatic translation into Fortran 77 was far more successful on the Linpack routines eliminating 86% of the GOTO statements and 77% of the knots. This indicates that the original Fortran 66 code was well structured and the majority of the GOTO's and knots were only present due to the lack of adequate control structures within the implementation language. As with Eispack the two routines identified in Table 1.2 account for a good proportion (just over a third) of the remaining knots.

Perhaps the biggest surprise was that an automatic restructing tool should have such a marked effect on the Lapack software. While GOTO statements are still necessary in Fortran 77 (for example, to construct while loops) and may generate knots (for example, when breaking out of a loop) such constructions would not be removed by *spag* when generating Fortran 77. The routines DLARFX and ZLARFX have all 132 knots removed, the restructured code containing no GOTO statements at all and this accounts for almost half the total number of knots removed. The other routine, ZHGEQZ, however is largely unaffected with a reduction from 60 to 52.

Restructuring has no effect on the cyclomatic complexity, the values being largely unchanged. Perhaps the only worrying trend is that the maximum value has increased quite dramatically with time, the routine ZHBEST has a cyclomatic complexity of 199 although it contains just 653 executable statements; given the suggested maximum values for this metric quoted in Section 2 this routine looks ripe for modularization.

		LOC per		% of routines with Path Counts
		GOTO	Knots	of Max Path (5×10^9
Eispack	f66	7	4	6.6%
	Spag	22	8	1.3%
Linpack	f66	7	7	8.7%
	Spag	52	32	2.2%
Lapack	f77	52	26	3.6%
	Spag	76	35	3.3%

TABLE 1.5. Density of knots and GOTO statements and the percentage of routines exhibiting high Path Counts.

Table 1.5 gives the density of both GOTO statements and knots in terms of the number of executable statements. The original Eispack code has an extremely high knot density (one for every four executable statements) which, even with restructuring, is only marginally better than the original Linpack code. The values obtained for the restructured Linpack code are extremely close to those of the original Lapack although the Lapack knot density requires the use of *spag* to improve on Linpack. The improvement due to restructuring the Lapack code is startling for a translation of Fortran 77 into itself.

Considering only restructured codes the percentage of routines with path counts exceeding the maximum value returnable by QAFortran has increased. This, along with the occurrence of large values for the cyclomatic complexity, implies that neither time nor software engineering principles have done anything to reduce the fraction of long and, logically, extremely complex routines appearing in state-of-the-art numerical Fortran libraries.

4 CALGO Fortran Codes

Since 1975 all the Collected Algorithms from the ACM have appeared as fully refereed papers in ACM Transactions on Mathematical Software and the accompanying code has been thoroughly tested prior to publication. It was felt that these sources should provide evidence of how both research workers and referees, who are mostly scientists whose primary expertise is not in computer programming, have altered the way they write Fortran code. It might be expected that there would be a lag between new software engineering principles being expounded and their practical application by the scientific community.

There are few restrictions imposed on the languages that may be used to implement algorithms, nevertheless, as Figure 3 shows, the majority of the published code appearing over the last twenty years has been in Fortran. It is interesting to note how long it took both for Fortran 66 to die out and for Fortran 77 to become the dominant dialect (almost 10 years after the appearance of the standard). Up until the end of 1995 only a single algorithm has been published in Fortran 90.

Many of the published codes use previously published or freely available software (for example, BLAS, Linpack, etc); in generating the data presented in this section all such routines were removed along with any drivers and other testing code.

In all the following figures the asterisks signify the average of the plotted quantity over all subroutines appearing in all the algorithms implemented in Fortran that were published in that year, whilst the piecewise line represents the maximum values. Both also have plotted their respective regression lines; as a dashed line for average values and a dot-dash line for

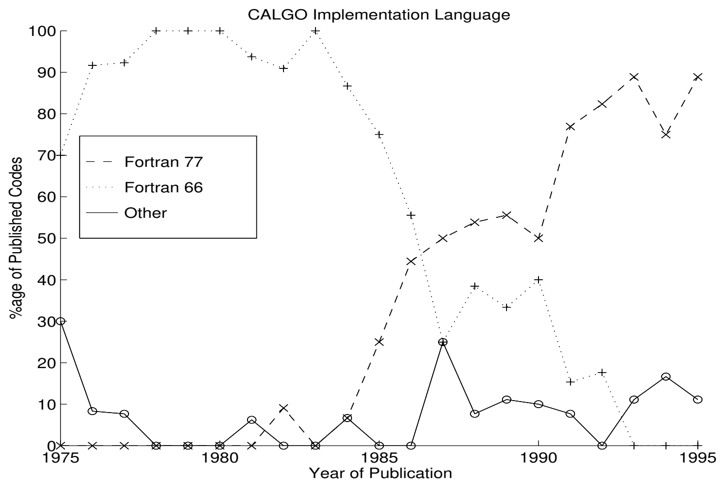

FIGURE 3. (1975–1995) CALGO algorithms by implementation language

the maximum values. Note that none of the Fortran 66 codes have been restructured.

It may be observed from Figure 4 that on average the number of executable lines per subroutine has gradually increased whilst the maximum number shows a more rapid upward trend. Even with the introduction of Fortran 77, Figure 4 shows that the average number of explicit GOTO statements has remained almost level. This is surprising as we would have expected to see a sharp decrease in these values as authors switched to Fortran 77. It is certainly worrying that in 1995 we are still seeing published routines containing around 200 explicit GOTO statements even if they may have been machine generated. The effect of the growth in GOTOs, is clearly reflected in the knot counts (Figure 5) which exhibits a very similar trend.

Figure 5 also shows the maximum values for the cyclomatic complexity of each routine. Again the average values show a gradual increase with time showing a general unwillingness by authors to increase the modularity of their code. We also note that the average value of around 50 is far in excess of the value of 14 suggested by the study reported in Section 2.

5 Conclusion

The overall picture obtained from section 4 is that the general practitioners of scientific software production are an extremely conservative group of people. The past twenty years have witnessed a general reticence to embrace new versions of Fortran and on average the values of the metrics used in

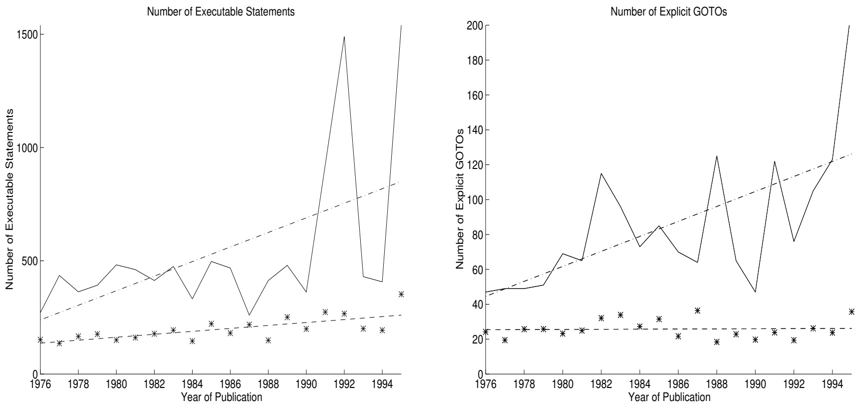

FIGURE 4. CALGO Algorithms: Executable Lines (left) and GOTO statements (right)

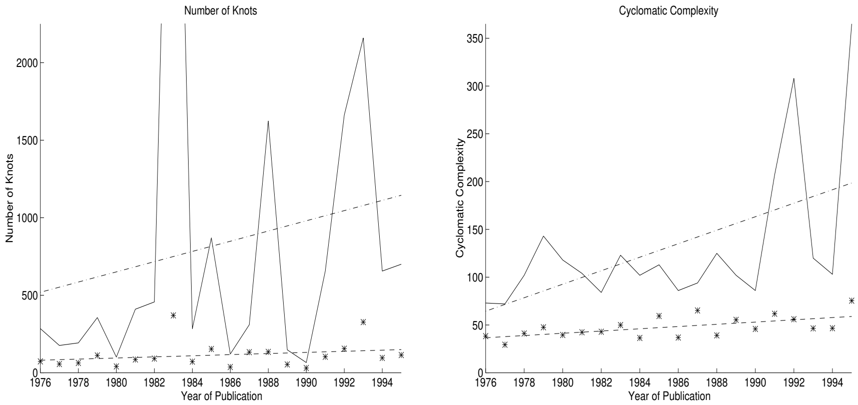

FIGURE 5. CALGO Algorithms: Knots (left) and Cyclomatic Complexity (right)

this paper have at best remained static but are generally showing a gradual increase for the worse.

There is no doubt that there has been a dramatic increase in the underlying algorithmic complexity of computational numerical methods over the period studied and that this has resulted in an overall increase in the complexity of their software implementations.

On the positive side many authors do reuse existing software both for portability purposes and as building blocks for their own code. However, while use of, for example, the BLAS may help to reduce the complexity of individual routines there is ample evidence that the basic software engineering principle of implementing complex algorithms using a number of small, well understood and easily testable modules is being largely ignored. Thus, as the overall complexity has increased, we would have expected to see the number of components increase whilst the complexity of the individual subprograms decreased. However routines have grown in size and logical complexity; many of the larger program units are likely to prove impenetrable even to a trained reader and would be extremely difficult to change or maintain.

On the other hand the software libraries considered in section 3 show some signs of improving quality, certainly the frequency of GOTO statements and knots has gradually decreased although the use of large, logically complex and virtually untestable routines has increased.

Tools to generate metric values could usefully be employed at all levels of numerical and scientific Fortran software production and reasonable restrictions on, for example, knots and cyclomatic complexity, would certainly improve the overall readability of the resultant code. Restructuring tools like *spag* have also proved to be useful not only in improving the structure (and thus decreasing the knot and path counts and the number of explicit GOTO statements) in dusty deck code but also in code written from scratch in Fortran 77.

Finally we note again the strong link, reported by Hatton and Hopkins [HH88], between very large path counts and the number of defects reported in released program units. On the above evidence this surely cannot bode well for the future.

Acknowledgments: My thanks to the two anonymous referees whose comments helped to improve this paper considerably.

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.

[ANS66] ANSI. *Programming Language Fortran X3.9-1966*. American National Standards Institute, New York, 1966.

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

[Bak77] B. S. Baker. An algorithm for structuring flowgraphs. *JACM*, 24(1):98–120, January 1977.

[BR70] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques; Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969.*, Brussels, 1970. NATO, NATO Scientific Affairs Division.

[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.

[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.

[Gra94] R. B. Grady. Successfully applying software metrics. *Computer*, 27(9):18–25, September 1994.

[HH88] T. R. Hopkins and L. Hatton. Experiences with FLINT, a software metrication tool for fortran 77. In *The Symposium on Software Tools*, Edinburgh, 1988. Napier Polytechnic.

[HK87] R. J. Hanson and F. T. Krogh. Translation of Algorithm 539: PC-BLAS basic linear algebra subprograms for Fortran usage with the INTEL 8087 80287 numeric data processor. *ACM Transactions on Mathematical Software*, 13(3):311–317, September 1987.

[Hop96a] T. R. Hopkins. Restructuring software: A case study. *Software — Practice and Experience*, 26(8):967–982, July 1996.

[Hop96b] T. R. Hopkins. Restructuring the BLAS level 1 fast Givens routines. Technical Report 13-96, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK, July 1996.

[ISO91] ISO/IEC. *Information Technology – Programming Languages – Fortran (ISO/IEC 1539:1991(E)).* ISO/IEC Copyright Office, Geneva, 1991.

[LDR] LDRA Group, Liverpool, UK. *LDRA Testbed.*

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

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

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[Mye77] G. J. Myers. An extension to the cyclomatic measure of program complexity. *Sigplan Notices*, 12(10):61–64, 1977.

[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.

[Pol93] Polyhedron Software, Oxford, UK. *plusFORT*, Revision B edition, 1993.

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

[RJR92] B. F. Ryan, B. L. Joiner, and T. A. Ryan Jr. *MINITAB Handbook.* Duxbury Press, Belmont, California, second edition, 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.

[She88] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, pages 30–36, March 1988.

[SI94] M. Shepperd and D. C. Ince. A critique of three metrics. *J. Systems Software*, pages 197–210, 1994.

[WHH79] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure to control flow complexity in program text. *IEEE Transactions on Software Engineering*, SE-5(1):45–50, 1979.

[WR71] J. H. Wilkinson and C. Reinsch. *Linear algebra*, volume 2 of *Handbook for automatic computation*. Springer-Verlag, New York, 1971.

[Zus91] H. Zuse. *Software complexity: measures and methods*. W. de Gruyter, Berlin, 1991.