A Comment on the Presentation and Testing of CALGO Codes and a Remark on Algorithm 639: To Integrate Some Infinite Oscillating Tails

Tim Hopkins
University of Kent, UK

We report on a number of coding problems that occur frequently in published CALGO software and are still appearing in new algorithm submissions. Using Algorithm 639 as an extended example, we describe how these types of faults may be almost entirely eliminated using available commercial compilers and software tools. We consider the levels of testing required to instil confidence that code performs reliably. Finally, we look at how the source code may be re-engineered, and thus made more maintainable, by taking account of advances in hardware and language development.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms

Additional Key Words and Phrases: debugging, testing, software tools, Fortran

1. INTRODUCTION

The purpose of this paper is twofold; first, to highlight several very common coding problems and second, to describe a number of "good practices" that we believe will largely eliminate such faults from users' programs. These types of errors have been found in almost all the original submissions of Fortran software to the CALGO over the past seven years and the methods proposed for uncovering them have evolved over the same period. These methods have been successfully applied not only to newly submitted algorithms but also to those already in the CALGO collection [Hopkins 2002].

To illustrate the discovery and correction of these types of faults we have used Algorithm 639 [Lyness and Hines 1986] from the CALGO as an extended example.

Name: Tim Hopkins
Address: Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK; email: trh@ukc.ac.uk

This code was chosen because

—it happens to contain several of the problems in relatively few executable lines,

—it was accompanied by an extremely well chosen set of test examples which made the testing process very straightforward, and

—it provided a good example of why code might be updated due to both advances in hardware and language development.

It should be noted that this software was produced at a time when memory was a more valuable resource than at present and when the international standard for Fortran was ANSI Fortran 77 [ANSI 1979].

For all the algorithm papers published in ACM TOMS the associated software packages have been made freely available both from the ACM CALGO Web site and via *netlib* [Dongarra and Grosse 1987]. For many authors gaining access to such a wide audience of potential users is one of the main reasons for submitting their software for publication in the CALGO. Typically each algorithm has been provided as a single flat file that contains all the material necessary to implement the software. This will always contain

—the source code to the algorithm, and

—the sources of any associated, freely available library routines used (for example, BLAS, LAPACK, etc),

and will generally include one or more of the following

—test software,

—data files,

—expected results files,

—makefiles, and

—user documentation.

In the case of Algorithm 639 the publicly available file was split relatively easily into the source code to the algorithm (502 lines) and the test software (1146 lines). The algorithm consisted of one user callable routine, `OSCINT`, and one subsidiary routine, `QRULE`. The provided test driver program did not require any data and, in this case, there was no sample output file. This was not too great a problem although it would have been reassuring to be able to check any results obtained from a new installation with those supplied by the original author.

In the following section we look in detail at the problems we uncovered as we tested Algorithm 639 and the source changes that were necessary to rectify them. Section 3 considers the role of testing and discusses a criterion for selecting test cases. The re-engineering of the algorithm into Fortran 95 is covered in section 4 and the final section generalizes the lessons learned from the specific example.

2. PROCESSING THE SOURCE CODE

2.1 Machine Dependencies

The test software provided with Algorithm 639 uses a routine, `RJBESL`, to compute Bessel functions of the form $J_{n+\alpha}(x)$; this requires the setting of a number of machine dependent constants. The definition of these constants along with a number of sample sets of values are included in the comments at the head of `RJBESL`. This version of the routine is too old to include the constants associated with IEEE arithmetic [IEEE 1985].

A more portable solution would have used the Port routines (`D1MACH` and `I1MACH`) that appeared as part of Algorithm 528 [Fox et al. 1978; Gay and Grosse 1999]. While not directly available, the constants required by `RJBESL` could have been computed from the values available from these Port functions. The downside of this approach is that there is a computational overhead each time the function is called although this may be minimized by the use of `SAVE` variables and an initialization flag. There is thus a trade-off between computational efficiency and portability to consider; from an installer's point of view, having code that will initially compile and execute without changes to the source is very useful.

A later version of `RJBESL` appears as part of the `SPECFUN` package published as Algorithm 715 [Cody 1993]. This version does contain values for the IEEE constants in the comments at the head of the routine but still does not use the `PORT` routines to set the machine constants. We used this newer, updated code.

It should be noted that all the values returned by the Port routines `D1MACH` and `R1MACH` are available as standard numeric enquiry functions in Fortran 95 [ISO/IEC 1997].

2.2 Compiler Checks

The software was tested using the following compilers

(1) Sun Workshop 6 f95, version 6.1,
(2) NAG f95, release 4.0a (309),
(3) Edinburgh Portable Compilers (EPC) epcf90, version 1.5.2.6. This compiler is, unfortunately, no longer commercially available.
(4) Lahey-Fujitsu lf95, Pro v. 6.0.

The first three ran on a four-processor Sun E450 under SunOS 5.8; the Lahey-Fujitsu compiler was on a Toshiba Portège 7010 running RedHat Linux release 6.2.

Having updated the machine-dependent constants, the source files compiled successfully using the default compiler options and linking against existing pre-compiled versions of the Port library routines. In all cases the resultant executable file generated an output file. The results were well laid out with each run of the algorithm well documented. On checking the four results files we identified substantial differences in the values obtained using the NAG compiler.

To investigate this further, all compile-time and run-time checking was activated for the EPC f90 compiler; as well as ensuring that all array elements were within range and that all variables had been assigned values before use, the compiler issued a variety of warning messages including the flagging of undeclared variables and unreferenced labels.

Specifically, no integer variables were found to be explicitly declared and the label 40 was unreferenced in the routine `OSCINT`. Inspection of the code also revealed the use of `IMPLICIT DOUBLE PRECISION` statements. These were removed and all variables explicitly declared. This could have been done automatically using either the NagWare Fortran 90 declaration standardizer [Numerical Algorithms Group Ltd. 1999] or the *spag* tool from Polyhedron [Polyhedron Software 1997]. Removal of the unused label allowed the code to be compiled with no warning messages using the EPC compiler.

In order to illustrate how run-time diagnostics may be used to correct faults in code, we describe each error discovered, detail how it affected the computation and show how it was corrected. The problems are reported in the order that they were found. Line numbers used in this section refer to the originally published algorithm code (with all comments removed) which is given in the Appendix. Context references are given for the driver code.

Any line numbers used below refer to the original algorithm code (with all comments removed) that is given in the Appendix. Context references are given for the driver program.

(1) *Use of an undefined variable, `NDIM2`, detected in the main program at the statement*

```
DO 80 J = 1,NDIM2
```

Inspection showed that this loop and the following `DO 90` loop were used to initialize three arrays to zero prior to passing them as actual arguments to `OSCINT`; a comment prior to the first loop stated: *This is unnecessary for running, but is useful for output.* Unfortunately this code preceded the first assignment to `NDIM2` for the particular problem being solved and for the first test `NDIM2` was not assigned a value at all. Most compilers would set `NDIM2` to zero at this point leading to the situation where, for the first test the loops are ignored and for all following tests `NDIM2` had the value set by the previous test. Such a situation may lead to misleading results being output.
These array arguments are output-only parameters to `OSCINT`, i.e., the elements are only ever written to within the routine. It should, therefore, be safe to remove the loops completely. See (7) below for further details of output problems.

(2) *Array index out of bounds at line 207: index zero.*
When untrapped the value accessed for `QLIST(0)` is compiler dependent and, therefore, `ISTATE(2)` may be set incorrectly.
The error was corrected by protecting the statement by an `IF` block of the form

```
IF (J.GT.0) THEN
  IF (QRULE*QLIST(J).GT.0) ISTATE(2) = J
ENDIF
```

Note that the statement

```
IF (J.GT.0 .AND. QRULE*QLIST(J).GT.0) ISTATE(2) = J
```

is not valid since, unlike C, the Fortran standard does not define partial evaluation rules for composite relational expressions.

(3) *Array index out of bounds at line 166: index zero.*
The value of `IELM` at this point was one so it was the `IELM-1` access that was causing the problem. The obvious solution was to mimic 2 above and protect

the statement by an outer `IF` block. However, reviewing the code reveals that `GMAX` was only assigned a value *following* this test; the correct course appeared to be

```
IF (IELM.EQ.1) THEN
  GMAX = DABS(SAVPER(1))
ELSE
  original IF block
ENDIF
```

(4) *Array index out of bounds at line 64: `MJK` zero.*
This implied that `JP` and `K` were equal and that the two assignments to `PREV` and `CURR` should both be moved after the following `IF` statement. However, since the two variables were only used once, the best solution was to substitute the right hand sides in place of the variables giving

```
      IF ((ABS(WORK(MJ1,K)).LT.EPS) .AND.
+         (ABS(WORK(MJK,K)).LT.EPS)) THEN
```

The declarations of these variables were also removed.

(5) *`NROUND` is undefined at line 99.*
This problem occurred when `NDIM1` was not equal to 10 at line 81; the intention was that `NROUND` should be zero unless some special circumstances prevailed. This initialization to zero needed to be explicit after line 80.
It was correcting this problem that cured the differences in the results obtained using the NAG compiler. This missed initialization had also affected the results generated using the other compilers. Further study of the original results files revealed that, for the third run of function 4 (`NFUN` set 4 and `NTEST` set 3 in the main program), the other three compilers all report an apparently successful execution but the difference was reported as $O(10^{-3})$ rather than the $O(10^{-13})$ accuracy achieved for this problem on previous runs. Initializing `NROUND` restored the accuracy.
We note here that, even when full checking has been requested, the EPC compiler preserves the values of local variables between calls of subprograms even if these variables do not appear in a SAVE statement. This has two effects; first, the system failed to detect a number of instances where the code did not adhere to the Fortran standard. Second, it allowed values generated by previous calls to subprograms to affect later invocations. This both masked further detectable faults completely and widened the cause/effect chasm [Eisenstadt 1997], as we see in fault (6) below. An example of the former problem is when a variable is initialized by the first test but not by the second. The second test then uses this left-over value. Both situations provide good reason for executing test cases individually rather than batching them up.
In this case the value assigned to `NROUND` in the previous test was being used; this happened to be 4 causing the actual results to be computed incorrectly.

(6) *`SAVPER(65)` is undefined at line 166.*
When executing this statement `IELM` had the value 66 and the test driver had reached the second function (`NUMFUN` was two).

The SAVPER array was used to store blocks of NQUAD-1 values returned from GPER in the elements [2, NQUAD] and [NQUAD + 2, 2 * NQUAD]. When each value was stored a test was made between the current array element and the previous value generated. This test should have failed with an undefined value when I was two and J was either one or two as no value was explicitly written to either SAVPER(1) or SAVPER(NQUAD+1). The EPC run-time checking failed to flag the occurrence when J was one as SAVPER(1) had been assigned to during the first test case and that value had been preserved – once again widening the cause/effect chasm.

It was tempting to repeat the argument used in (3) above since, by setting SAVPER(65) to zero, we would have mimicked what most compilers that do not check for undefined values would have done. Gaining more understanding of what was actually being computed led us to a deeper fault. Studying the code revealed that SAVPER(1) was actually being used to determine whether the user function, GPER, was generating values that were independent of X. It was not correct to set this value to zero. Thus, the fix applied in (3) above was itself incorrect; it made the execution error go away but did not correct the fault in the software.

GMAX should have been set to the largest absolute value of GPER(Y) computed over the range [RFIRST, RFIRST+PERIOD]; it was actually being set to the size of the local maximum/minimum closest to RFIRST+PERIOD.

We replaced the block inserted in (3) above by

```
GMAX = MAX(GMAX, ABS(SAVPER(IELM)))
```

and we ensured GMAX was initialized for Gauss rules by setting it to zero when J was zero following line 143. GMAX, SAVPER(1) and SAVPER(NQUAD+1) were all initialized correctly for the trapezium rule by changing line 160 to

```
IF (J.GE.3 .AND. I.EQ.1 .AND. NQUAD.GT.0) GO TO 20
```

and inserting the block

```
IF (I.EQ.1 .AND. NQUAD.GT.0) THEN
  GMAX = DABS(SAVPER(IELM))
  GO TO 20
ENDIF
```

after statement 164.

(7) *WORK(17,14) has an undefined value in the DO 140 loop of the main program.* This loop was used for printing a part of the forward average table and the reported error was a direct result of removing the initialization code in (1) above. The problem was that not all the elements in the WORK array were necessarily assigned to; it depended on how many integral calculations were necessary. The table was built up one diagonal at a time and, for large values of NDIM1 and NDIM2, it was more than likely that the whole table was *not* set before the routine exited. Values assigned to the ISTATE array provided the necessary data to access the relevant data in the WORK array. A better way of printing the relevant part of the table was to replace the DO 140 loop of the main program by

```
          JP = ISTATE(3)
          JPMK = JP - ISTATE(4)
          DO 140 JPR = 1, NPTOP
            ENDV=JP-JPR
            IF(JPR .GT. JPMK)THEN
              ENDV=ENDV+1
            ENDIF
            ENDV = MIN(ENDV,NDIM2)
            WRITE(FORM,'("(1X,I3,",I3,"D9.2,/)")')ENDV
            WRITE(*,FORM)JPR,(WORK(JPR,KPR),KPR=1,ENDV)
  140     CONTINUE
```

These changes meant that the test on function 3 that set NDIM1 to 5 (NUMFUN equals 3 and NTIME equals 3) needed to be removed. This was because NDIM1 was no longer the correct value of the leading dimension used to declare the WORK array in the main program. Use of this value of NDIM1 meant that data were written into locations from which they were not retrievable using the new table output code above. Indeed, further study of the original output revealed that results were being presented in a far from readable form.

(8) To test the portability of the changes, the sources were moved to the Lahey-Fujitsu compiler; this compiler also traps the use of undefined values with the added check that local variables become undefined between calls to a subprogram unless they appear in a SAVE statement. Using this compiler uncovered two additional variables (INDEX and GMAX) that needed to be added to the SAVE statement in QRULE.

(9) The following set of minor changes were discovered while reading the code
   (a) The routine GAUSS was defined as having nine arguments; the first five were not used by the example routine G5AND9 and appeared to be a leftover from an original call to the NAG library routine D01BCF. Since only the last four arguments were relevant to the routine OSCINT only these should be required in the definition of GAUSS. Access to D01BCF, for example, may be achieved from within the user defined function.
   (b) The comments in Note 3 (in OSCINT) were incorrect in two places. First the periodicity of GPER was only checked for in the third and fourth intervals. Second the values of ISTATE(1) mentioned (-3000 and -5000) needed to be interchanged.
   (c) The test at line 200 was replaced by

   ```
   IF (INDEX .EQ. 2*NPTS) THEN
   ```

   since saving the cost of a few multiplies by unity is unlikely to be offset by the cost of the extra test.
   (d) Various magic numbers appeared throughout the code. These should have been either defined as named constants or, as in the case of the hard limit of 100 in the number of integral values, removed completely.
   (e) All specific names for intrinsic functions were changed to generic names.
   (f) The output from the driver program appeared to indicate that the user defined routine G5AND9 was being used even when the trapezium rule was used for the calculations.

(g) In `OSCINT` the two successive `IF` statements (lines 83 and 84) may be combined into a single block-`IF`.

3. TESTING

Dijkstra [Dijkstra 1979] stated that *Testing can be used to show the presence of bugs, but never to show their absence!* Hence what we are striving to achieve by testing is to build up our confidence that the software we produce will perform as designed and, to try and ensure that we have catered for as many unusual circumstances as possible. To this end a minimal test suite should aim to execute every statement at least once; this is termed basic block testing and is a comparatively weak testing requirement [Zhu et al. 1997]. It is sometimes an open question as to whether or not certain conditions may occur in the course of numerical computations. Thus, for the sake of robustness, codes may include statements that no input data have ever caused to be executed. Developers and users alike need to treat such untested (or, possibly, untestable) code with caution. A case can certainly be made for treating all such situations as error conditions; in this way data exercising these untested statements may be captured and carefully analyzed.

Tools are available to measure the basic block coverage attained during testing; for example,

(1) the Sun f95 compiler provides a *-xprofile=tcov* flag, which when used in conjunction with the *tcov* utility, generates an annotated listing of an execution profile,
(2) the Lahey-Fujitsu compiler (under Windows) has a *-cover* flag which generates information for a separate Windows-based coverage tool [Lahey Computer Systems, Inc. 2000],
(3) the NAGWare suite contains a pair of tools, *nag_profile* and *nag_history*, that instrument Fortran 77 code and produce annotated coverage listings.

Using the Sun f95 tool with the test material provided indicated that 99 of the 102 basic blocks (97%) were covered in the routines `OSCINT` and `QRULE`. This decreased to 116 out of 123 (94%) when logical-`IF` statements were replaced by one line block-`IF`s. This transformation may be performed automatically using `spag`. The reason for the difference in coverage was that the Sun profiler counted a single logical-`IF` as a basic block and did not differentiate between the associated statement being executed or not. A block-`IF` was treated as two basic blocks and we obtained extra profiling detail.

Of the seven unexecuted blocks two were error exits, two resulted from the same test failing in successive logical-`IF` statements, one was used to flag a slight concern over the returned result, one was a special input case and the last was the result of the constant function in the tests always being unity. This level of statement coverage was high and is unusual for general numerical software.

The test driver software as provided could be improved in a number of ways. First the driver routines appeared to have originally offered a more extensive set of test functions and to have been partially dismantled to their current state. As we have already noted, it was sometimes easier to find coding problems if each test was performed, as far as possible, in isolation. We thus re-engineered the main program to use a data file defining a single test of the algorithm. These files contained all the major input parameters required by `OSCINT`. The user-provided functions, `GPER`, `HFUN` and `EXACT`, were tidied to support the full six 'original' test functions.

Using separate tests uncovered a minor error in the driver program where an attempt was made to output `RESULT` from `OSCINT` when the period was negative. `RESULT` had not been assigned a value under these circumstances. This was corrected by initializing `RESULT` to zero as the first executable statement of `OSCINT` and the documentation was changed to reflect the fact that a value of zero would be returned on detecting erroneous input.

Extra tests were added and a number of the original tests, which did not provide any extra basic block coverage, were removed. The resultant suite of 20 data sets achieved 100% basic block coverage, took less than a second to execute on the Sun E450 and provided a good degree of confidence in the performance of the software. Six of the tests exhibit the successful basic computation of each of the defined test integrals. The others exercise the error exits along with a number of special cases.

4. FORTRAN 95 IMPLEMENTATION

Using Fortran 95 [ISO/IEC 1997] to re-engineer the software allowed a number of simplifications to be made both to the original algorithm code and to the user interface.

By dropping the storage economy facility obtained by setting `NDIM1` to 10 in the original implementation we were able to simplify the code of `OSCINT` considerably. For example, the 27 executable statements (containing 4 loops and 3 `GOTO` statements) used to compute the final value of `RESULT` in the original software were replaced by a single statement.

Defining `WORK` to be an assumed-shape array allowed the determination of the problem dependent parameters `NDIM1` and `NDIM2`, which could then be removed from the argument list of `OSCINT`.

The original argument list for `OSCINT` contained four arrays `WEIGHT`, `ABSCIS`, `SAVPER` and `QLIST`. The first two were required to store the weights and abscissa points computed by the user supplied function `GAUSS`. It was unnecessary for a user to have to provide both the arrays and the function used to define them. The other two arrays held temporary data and were of no particular interest to the user. All four arrays were made local to the module containing the `OSCINT` and `QRULE` sub-programs and space allocated at run-time using problem defining parameters.

In addition we used the new relational operators, and `INTENT` attributes. The precision of the code was set using a `KIND` value obtained from a separate module; this meant that a single line change and a recompilation was all that was required to change the precision of the software. An alternative solution would have been to provide a generic interface using single and double precision versions of the routines.

A full description of the arguments to `OSCINT` was included at the head of the module.

We also note that 100% block coverage was not achieved with the example data sets used for the Fortran 77 code. This was because the error exits caused by the ALLOCATE and DEALLOCATE statements failing can not be triggered in a platform independent way.

5. CONCLUSION

We have illustrated how the use of commercial compilers and software tools may be used to detect a set of commonly occurring coding faults. This approach was instrumental in finding problems that led to several hundred sets of code changes to the existing CALGO library codes [Hopkins 2002]. Some examples of the errors that may be uncovered in this way are

—use of variables before they have been assigned values,

—attempted access to elements outside the declared range of an array,

—assumption that local variables preserve their values between calls to subprograms without using a `SAVE` statement,

—incorrect type or rank of actual arguments in subprogram calls (a common example is the use of a simple variable for an array of length one).

Many of these produce different effects under different compilers. Most are difficult to uncover without the use of run-time checking and a comprehensive testing process. All are non-standard conforming. Often such faults mask a much deeper problem as in points (3) and (6) in section 2.2 above. All of these faults have the potential to cause the catastrophic failure of the software.

A second class of problem that may be considered less serious but may, nonetheless, cause inaccurate results as well as potentially increased maintenance costs includes

—implicit declarations using Fortran's default typing rules. This may lead to inaccurate results when, for example, double precision variables that have escaped explicit declaration default to real,

—variables declared and/or assigned to but never used,

—labels that are never used as targets.

The `IMPLICIT NONE` statements in Fortran 95 may be used to avoid the first of these problems. "Code clutter" may be either removed automatically using available software tools or detected by setting compiler flags and removed manually.

Most commercial Fortran compilers provide compile-time options that enable run-time array bound checking and many allow more extensive checking. We would highly recommend using these facilities throughout the coding and testing phases of numerical software production. Even though such checks may slow down the execution speed dramatically, the ability to pinpoint problems easily and quickly repays this overhead handsomely.

Using CALGO Algorithm 639 as an illustrative example, we have shown the problems that an implementor may face in trying to ensure that the final code is standard conforming and reliable.

Re-engineering this software in Fortran 95 showed how we could simplify the user interface and, with large amounts of memory being standard on current machines, how the original storage economy facility could be removed to simplify the code. Unfortunately the transition into Fortran 95 was not an automatic process although tools are available to assist ([Numerical Algorithms Group Ltd. 1999], [Polyhedron Software 1997]). From a maintenance point of view the resultant code is far simpler structurally and thus easier to understand than the original version.

The use of a testing goal is important. By requiring that full basic block coverage be achieved we were able to produce a small, but comprehensive, set of test data sets. Fulfilling the testing goal provided us with a metric that allows us to have some confidence in the final code. A number of tools are available to assist with the testing phase of Fortran software development.

Overall we believe that we have improved the quality of the software and that we have corrected the majority of the errors in the original published version. However, we would be very foolish to claim that our final version is perfect.

ACKNOWLEDGMENTS

The author would like to thank Les Hatton for reading a draft version of this paper and suggesting a number of improvements.

REFERENCES

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

Cody, W. J. 1993. SPECFUN: A portable Fortran package of special function routines and test drivers. *ACM Transactions on Mathematical Software 19*, 1 (March), 22–32.

Dijkstra, E. W. 1979. Structured programming. In E. N. Yourdon Ed., *Classics in Software Engineering*, pp. 43–48. New York: Yourdon Press.

Dongarra, J. J. and Grosse, E. 1987. Distribution of mathematical software via electronic mail. *Commun. ACM 30*, 5 (May), 403–407.

Eisenstadt, M. 1997. My hairiest bug war stories. *Commun. ACM 40*, 4 (April), 30–37.

Fox, P. A., Hall, A. D., and Schryer, N. L. 1978. Framework for a portable library. *ACM Transactions on Mathematical Software 4*, 2 (June), 177–188.

Gay, D. M. and Grosse, E. 1999. Self-adapting fortran 77 machine constants: comment on algorithm 528. *ACM Transactions on Mathematical Software 25*, 1, 123–126.

Hopkins, T. 2002. Renovating the collected algorithms from acm. *ACM Transactions on Mathematical Software 28*, 1, ???–???

IEEE. 1985. *IEEE standard for binary floating-point arithmetic* (ANSI/IEEE Standard 754-1985 ed.). New York: Institute of Electrical and Electronic Engineers.

ISO/IEC. 1997. *Information Technology – Programming Languages – Fortran - Part 1: Base Language (ISO/IEC 1539-1:1997)*. ISO/IEC Copyright Office, Geneva.

Lahey Computer Systems, Inc. 2000. *Lahey/Fujitsu Fortran 95 User's Guide* (Revision C ed.). Incline Village, NV, USA: Lahey Computer Systems, Inc.

Lyness, J. and Hines, G. 1986. To integrate some infinite oscillating tails. *ACM Transactions on Mathematical Software 12*, 1 (March), 24–25.

Numerical Algorithms Group Ltd. 1999. *NAGWare Fortran Tools, Release 4.0*. Oxford, UK: Numerical Algorithms Group Ltd.

Polyhedron Software. 1997. *plusFORT* (Revision D ed.). Oxford, UK: Polyhedron Software.

Zhu, H., Hall, P. A. V., and May, J. H. R. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec.), 366–427.

APPENDIX

Original Source Code

```
      SUBROUTINE OSCINT(AZERO,PERIOD,RFIRST,EPS,NQUAD,NDIM1,NDIM2,GAUSS,
     .                  HFUN,GPER,WORK,SAVPER,WEIGHT,ABSCIS,QLIST,
     .                  RESULT,ISTATE)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION WORK(NDIM1,NDIM2),QLIST(100),ISTATE(6)
      DIMENSION SAVPER(*),WEIGHT(*),ABSCIS(*)
      EXTERNAL GAUSS,HFUN,GPER
```

```
      HASPER = .5D0*PERIOD
      WMIN = 1.0D0
      DO 10 I = 1,6
         ISTATE(I) = 0
   10 CONTINUE
      IF (NQUAD.EQ.0 .OR. NQUAD.EQ.1 .OR. NDIM1.LT.1 .OR. NDIM2.LT.
     .    1) ISTATE(1) = -6000
      IF (PERIOD.LT.10.0D-5) ISTATE(1) = -5000
      IF (ISTATE(1).LT.0) GO TO 130
      IF (NDIM1.EQ.10) THEN
          NMAX = 100
      ELSE
          NMAX = MIN(100,NDIM1)
      END IF
      JP = 0
      S = 0.0D0
   20 CONTINUE
      IF (JP.EQ.NMAX) ISTATE(1) = -100
      IF (ISTATE(1).NE.0) GO TO 50
      K = 0
      JP = JP + 1
      J = MIN(JP,NDIM2)
      IF (NDIM1.EQ.10) J = MIN(J,20)
   30 IF (K.LT.J) THEN
          K = K + 1
          IF (NDIM1.EQ.10) THEN
              MJ = MOD(JP,10)
              MJM2 = MOD(JP-2,10)
              MJ1 = MOD(JP-K+1,10)
              MJ2 = MOD(JP-K+2,10)
              MJK = MOD(JP-K,10)
              IF (MJ.EQ.0) MJ = 10
              IF (MJM2.EQ.0) MJM2 = 10
              IF (MJ1.EQ.0) MJ1 = 10
              IF (MJ2.EQ.0) MJ2 = 10
              IF (MJK.EQ.0) MJK = 10
          ELSE
              MJ = JP
              MJM2 = JP - 2
              MJ1 = JP - K + 1
              MJ2 = JP - K + 2
              MJK = JP - K
          END IF
          IF (K.EQ.1) THEN
              WORK(MJ,K) = QRULE(JP-1,AZERO,HASPER,RFIRST,NQUAD,NDIM1,
     .                     NDIM2,GAUSS,HFUN,GPER,SAVPER,WEIGHT,ABSCIS,
     .                     QLIST,ISTATE)
          ELSE
              WORK(MJ1,K) = (WORK(MJ1,K-1)+WORK(MJ2,K-1))/2.0D0
              IF (DABS(WORK(MJ1,K)).LT.WMIN) THEN
                  WMIN = DABS(WORK(MJ1,K))
                  NOW = K
                  NROW = MJ1
                  NOWJP = JP
              END IF
              CURR = ABS(WORK(MJ1,K))
              PREV = ABS(WORK(MJK,K))
              IF (JP.NE.K) THEN
                  IF ((CURR.LT.EPS) .AND. (PREV.LT.EPS)) THEN
                      NOW = K
                      NROW = MJ1
                      NOWJP = JP
                      GO TO 50
                  END IF
              END IF
          END IF
   40     GO TO 30
      END IF
      GO TO 20
   50 CONTINUE
```

```
      DO 60 I = 1,NOWJP - NOW
         S = S + QLIST(I)
   60 CONTINUE
      IF (NDIM1.NE.10) GO TO 110
      NROUND = NOW - 10
      IF (NROUND.LE.0) NROUND = 0
      IF (NROUND.LE.0) GO TO 110
      DO 70 I = NOWJP - NOW + 1,NOWJP - 10
         S = S + QLIST(I)
   70 CONTINUE
      ISUB = NOWJP - 9
   80 IF (ISUB.GT.10) ISUB = ISUB - 10
      IF (ISUB.GT.10) GO TO 80
      DO 90 J = 1,NROUND
         S = S + WORK(ISUB,J)/2.0D0
   90 CONTINUE
      DO 100 I = NROW,NROW + NROUND - 1
         II = I
         IF (I.GT.10) II = I - 10
         S = S - WORK(II,NROUND+1)
  100 CONTINUE
  110 DO 120 J = NROUND + 1,NOW - 1
         S = S + WORK(NROW,J)/2.0D0
  120 CONTINUE
      S = S + WORK(NROW,NOW)
      RESULT = S
      ISTATE(3) = JP
      ISTATE(4) = NOW
      ISTATE(5) = NROW
      NML = ISTATE(3) - ISTATE(2)
      IF (NML.LT.4 .AND. ISTATE(1).EQ.0) ISTATE(1) = MAX(0,4-NML)
  130 CONTINUE
      RETURN
      END
      FUNCTION QRULE(J,AZERO,HASPER,RFIRST,NQUAD,NDIM1,NDIM2,GAUSS,HFUN,
     .               GPER,SAVPER,WEIGHT,ABSCIS,QLIST,ISTATE)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      SAVEB,TENDPT
      EXTERNAL GAUSS,HFUN,GPER
      DIMENSION QLIST(100),ISTATE(6)
      DIMENSION SAVPER(*),WEIGHT(*),ABSCIS(*)
      NPTS = ABS(NQUAD)
      IF (J.EQ.0) THEN
          INDEX = 0
          DO 10 I = 1,100
             QLIST(I) = 0.0D0
   10     CONTINUE
          IF (NQUAD.LT.0) THEN
              ITYPE = 0
              AA = -1.0D0
              BB = 1.0D0
              CC = 0.0D0
              DD = 0.0D0
              IFAIL = 0
              CALL GAUSS(ITYPE,AA,BB,CC,DD,NPTS,WEIGHT,ABSCIS,IFAIL)
              IF (IFAIL.NE.0) THEN
                  ISTATE(1) = -4000
                  GO TO 40
              END IF
          END IF
      END IF
      QRULE = 0.0D0
      IF (J.NE.0) THEN
          A = B
      ELSE
          A = AZERO
      END IF
      IF (RFIRST.LE.AZERO .OR. J.NE.0) THEN
          B = A + HASPER
      ELSE
```

```
       B = RFIRST
   END IF
   DO 30 I = 1,NPTS
      XI = I
      IF (NQUAD.LT.0) THEN
          Y = (B-A)*ABSCIS(I)/2.0D0 + (B+A)/2.0D0
          WT = WEIGHT(I)
      ELSE
          Y = ((XI-1)*B+A* (NPTS-I))/ (NPTS-1)
          WT = 1.0D0
      END IF
      IELM = NPTS*MOD(J-1,2) + I
      IF (J.GT.0 .AND. I.EQ.1 .AND. NQUAD.GT.0) GO TO 20
      IF (J.EQ.0) THEN
          FUN = HFUN(Y)*GPER(Y)
      ELSE IF (J.LT.3) THEN
          SAVPER(IELM) = GPER(Y)
          IF (SAVPER(IELM).EQ.SAVPER(1)) INDEX = INDEX + 1
          IF (DABS(SAVPER(IELM)).GT.DABS(SAVPER(IELM-1))) THEN
              GMAX = DABS(SAVPER(IELM))
          END IF
          FUN = HFUN(Y)*SAVPER(IELM)
      ELSE IF (J.LT.5) THEN
          DIFF = DABS(SAVPER(IELM)-GPER(Y))
          IF (DIFF.LT.GMAX*1.0D-5) THEN
              FUN = HFUN(Y)*SAVPER(IELM)
          ELSE
              ISTATE(1) = -3000
          END IF
      ELSE
          IF (INDEX.EQ.2*NPTS) THEN
              FUN = HFUN(Y)
          ELSE
              FUN = HFUN(Y)*SAVPER(IELM)
          END IF
      END IF
      ISTATE(6) = ISTATE(6) + 1
20    CONTINUE
      IF (NQUAD.GT.0) THEN
          IF (I.EQ.1) THEN
              IF (J.EQ.0) FUN = FUN/2.0D0
              IF (J.GT.0) FUN = TENDPT
          END IF
          IF (I.EQ.NPTS) THEN
              FUN = FUN/2.0D0
              TENDPT = FUN
          END IF
          QRULE = QRULE + FUN
      ELSE
          QRULE = QRULE + WT*FUN
      END IF
30 CONTINUE
   IF (INDEX.EQ.2*NPTS .AND. SAVPER(1).NE.1) THEN
       QRULE = QRULE*SAVPER(1)
   END IF
   IF (NQUAD.GT.0) WSUM = NPTS - 1
   IF (NQUAD.LT.0) WSUM = 2.0D0
   QRULE = QRULE* (B-A)/WSUM
   QLIST(J+1) = QRULE
   IF (QRULE*QLIST(J).GT.0) ISTATE(2) = J
   IF (ISTATE(2).GT.9) ISTATE(1) = -200
40 CONTINUE
   RETURN
   END
```