# Renovating the Collected Algorithms from ACM

Tim Hopkins
University of Kent, UK.

Since 1960 the Association for Computing Machinery has published a series of refereed algorithm implementations known as the Collected Algorithms of the ACM (CALGO). Most of those published since 1975 are mathematical algorithms, and many of them remain useful today. In this paper we describe measures that have been taken to bring some 400 of these latter codes to an up-to-date and consistent state.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Certification and testing; Reliability and robustness*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability; Restructuring, reverse engineering, and reengineering*

General Terms: Algorithms

Additional Key Words and Phrases: Standardization, Software Re-engineering, Software Tools, Software Testing

## 1. INTRODUCTION

We report on the steps that were taken to upgrade the codes, which form the electronic version of the *Collected Algorithms from ACM* (CALGO), into a more modern, consistent and usable state. These revised codes form the major content of the first release of the CALGO Collection Special Edition CD.

We begin in Section 2 by providing a brief history of the CALGO and trace how the requirements for publication have changed as our understanding of software engineering has improved. This is followed by a description of the original state of the codes and the initial steps necessary to make the component files easier to identify and handle.

Section 4 looks at the ways in which we were able to improve the readability, reliability and portability of the Fortran codes which form by far the largest fraction of the available algorithms. We then consider the testing of the algorithms and detail some of the common problems that were found and corrected.

Name: Tim Hopkins
Address: University of Kent, Computing Laboratory, Canterbury, Kent, CT2 7NF, UK.
Address: FAX: +44 1227 762811
Address: Tel: +44 1227 823793

In Section 6 we discuss how the updated versions of the algorithms have been packaged and what value-added features have been included on the CD. Finally, we consider how the software may be further improved in later releases.

## 2. A BRIEF HISTORY OF THE CALGO

The first algorithm published in the CALGO series appeared in the February 1960 edition of *Communications of the ACM* (CACM) [Herbold 1960]. The idea was to provide a means for programmers to make available their coded versions of algorithms to a wider audience for both pedagogical and reuse reasons. The first 331 published algorithms were all in Algol 60 and, although the Algorithm Policy was extended in September 1966 to allow Fortran, it was not until June 1968 that the first Fortran code made its appearance [Witte 1968]. The early codes were published as submitted and there was certainly no formal refereeing of the software before publication. Rather, other interested readers performed the quality control by rekeying the published code, running their own tests and submitting a follow-up certificate or remark reporting either successful execution or the changes required to obtain a working code.

The fact that the vast majority of the software being written in the 1960s and 1970s was for scientific applications is reflected in the preponderance of numerical algorithms appearing in the CALGO during that time. During the early 1970s the size of the published algorithms increased and there was a move away from Algol 60 to Fortran as the language of choice.

In 1975 the publication of CALGO moved to ACM's *Transactions on Mathematical Software* (ACM-TOMS) and this proved to be a turning point. Algorithm papers were now placed on the same footing as regular papers with both the accompanying paper and, more importantly, the software being rigorously refereed. The new Algorithms Policy [Fosdick 1975] required submitted algorithms to be either a new and interesting idea or a new and useful implementation. Code could be in Fortran, Algol or PL/1 but it was noted that other languages, while not excluded, needed to be "reasonably well known" and their use in preference to the three above "justified". Evidence of testing was required and it was expected that referees would independently run the submitted code. As an aid to portability there was a further requirement that any machine dependent constants were to be collected together and clearly identified.

This policy was revised in June 1978 [Krogh 1978] and June 1979 [Krogh 1979] to allow algorithm submissions that provided "capabilities not readily available" or that performed "a task better in some way than had been done before". In this context "better" was defined as meaning "anything from improved reliability or efficiency to more attractive packaging". The testing requirements were also extended to include an example driver program along with a single stringent test driver that provided "a sufficient variety of test cases to exercise all the main features of the code".

Submissions were now constrained more by portability; "It must be possible to move the code in machine readable form from one machine to another with only minor, well documented changes". Implementation languages were controlled by the need to provide evidence that the software executed successfully on three computers with different basic instruction sets. An alternative form of proof was to

| | Number of Algorithms | First Appearance Year | Number |
|---|---|---|---|
| Algol 60 | 414 | 1960 | 1 |
| Fortran (66/77) | 360 | 1968 | 332 |
| Fortran (90/95) | 9 | 1994 | 734 |
| C | 8 | 1993 | 722 |
| PL/1 | 4 | 1973 | 444 |
| Matlab | 3 | 1991 | 694 |
| C++ | 3 | 1996 | 764 |
| Ada | 1 | 1999 | 795 |
| Lisp | 1 | 1995 | 744 |
| Nitpack | 1 | 1984 | 620 |
| Pascal | 1 | 1989 | 673 |
| Ratfor | 1 | 1981 | 568 |
| Awk | 1 | 2000 | 803 |

Fig. 1. CALGO Implementation Languages (1960-2000)

run the code through a verifier that checked for portability and/or conformance to a standard. In particular it was a pre-requisite that Fortran code be given a clean bill of health from the PFORT portability verifier [Ryder 1974]. Brief advice was also given to Fortran programmers regarding the use of array space and common blocks.

The increasing availability of Fortran 77 was acknowledged in the March 1982 revision of the Algorithms Policy [Krogh 1982] with a requirement for codes to adhere to the new standard. Submissions in Fortran 66, checked by PFORT, were still acceptable and this remained the case until the September 1990 revision [Krogh 1990]. The use of the PORT library functions [Fox et al. 1978a] for setting common machine dependent constants was also advocated, otherwise standard names and definitions were to be used [Ford 1978].

Table 1 gives a breakdown of the CALGO by implementation language along with the number of algorithms appearing in each language, and the year and algorithm number of the first use of a language.

The majority of the pre-ACM-TOMS codes were never collected in machine readable form although a number of the later codes appearing in CACM were made available by the Algorithms Editor on magnetic tape; the first one appears to be Algorithm 468 in 1973. At this time a user had a choice of a single file of BCD, 80-character, card images at either 556 or 800 bpi.

With the move to ACM-TOMS, ACM started to offer an official algorithms distribution service with individual codes available as program listings, card decks or magnetic tape (now in three formats; EBCDIC, ASCII and BCD!). It appears that each magnetic tape contained all the algorithms from a particular issue of ACM-TOMS. By 1980 it was possible to obtain combinations of algorithm codes on a variety of media. 1985 saw the introduction of the floppy disk as a media option while 9 track magnetic tapes were still on offer in 1993. The postal distribution of algorithms ceased in 1996.

Meanwhile 1985 saw the introduction of the *netlib* service for distributing software via electronic mail [Dongarra and Grosse 1987] with one of the first available packages being the CALGO. This allowed algorithms to be easily obtained, as soon as they were published in ACM-TOMS, to anyone with a network connection. The intervening years have seen the introduction of both FTP and HTTP access to the netlib collection as well as the mirroring of the netlib service throughout the world.

## 3. ORIGINAL STATE

In the remainder of this paper we discuss the improvements we have made to the originally published algorithms to make them easier both to test and to integrate into user code, more portable, more reliable and more maintainable. From now on the collection of algorithms referred to are those that have been published in ACM-TOMS (numbered from 493). Since the vast majority of the CACM published algorithms were implemented in Algol, we see from Table 1 that a large percentage of the current collection is in Fortran.

To make their distribution simpler, each algorithm was originally held in a single file and most of these files contain no separators. This resulted in algorithm sources, driver programs, data and results files, makefiles, support subroutines and documentation being found in largely random order within each file.

One requirement for a published algorithm is that the code appearing in the CALGO should be stand alone, and the use of freely available software has always been advocated in preference to authors re-inventing the wheel. So, for example, routines from the Lapack [Anderson et al. 1999], Linpack [Dongarra et al. 1979] and Eispack ([Smith et al. 1976] and [Garbow et al. 1977]) software libraries are commonly used to solve linear algebra subproblems within an algorithm code.

The sources to all such auxiliary routines were included with every algorithm that used them. Although guaranteeing that each algorithm was self-contained this inclusion policy had two serious drawbacks. First, any corrections within this external software rarely found its way back into the CALGO submissions. Second, many of the libraries are already available at installations, often as specially tuned versions. This means that these auxiliary routines need to be removed from the CALGO code to ensure that any more efficient platform dependent versions are used.

The implementation and testing of a CALGO algorithm available in a single flat file is thus often a lengthy and error prone task consisting of some or all of the following steps

—identify and, possibly, remove auxiliary library routine sources,
—identify and collect together the routines making up the algorithm source. Even these routines might not appear contiguously within the file and a single file may contain implementations in more than one precision often with a duplicated namespace,
—identify the test drivers and supporting routines along with corresponding data and results, if present. Many of the currently available algorithms do not include a set of expected results which makes the testing stage more difficult,
—identify any machine dependent values and/or code and alter, as necessary, for the target platform.

The first task in the renovation process was thus to improve the accessibility of the various components which make up each individual algorithm submission. This was achieved using a directory structure with a consistent naming convention which

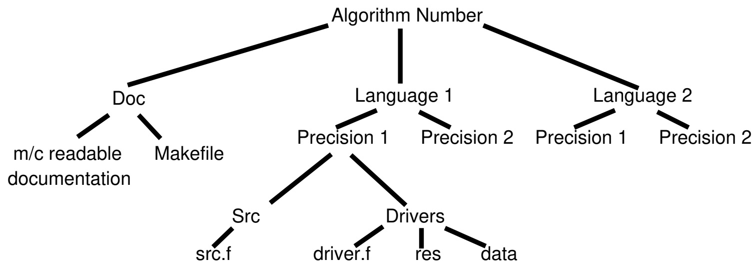



Fig. 2. Overview of directory structure used for individual algorithm codes

allows the contents of most files to be identified from their path name. A very small number of submissions did not fit easily into this convention and these have been left largely "as submitted".

Figure 2 shows the directory layout used for most of the algorithms. An example file path might be

```
493/Fortran77/Sp/Src/src.f
```

which would lead to the single precision, Fortran 77, source code for algorithm 493. None of the files within these algorithm directories contains any sources of external library routines, rather the complete libraries are provided within a separate `Packages` directory. Currently this directory contains

—*Blas.* The Level 1, 2 and 3 routines are provided. The Level 1 routines consist of a subset of the routines which appeared as Algorithm 539 [Lawson et al. 1979]. This subset is the same as that offered by the netlib service and consists of the most heavily used of the original routines. The Level 2 and 3 routines are the complete set as published as Algorithms 656 [Dongarra et al. 1988] and 679 [Dongarra et al. 1990].

It should be noted that one or more of these libraries are often available as a manufacturer-tuned library which will generally provide much better performance than that produced by compiling the standard Fortran versions.

—*Lapack.* The Lapack library of linear algebra routines was designed to both replace and extend the functionality of Linpack and Eispack. As with its two predecessors it is designed to be both efficient and portable over a wide range of architectures. A full description of the contents of the library, the methods used and details of the individual routines may be found in the extensive user manual [Anderson et al. 1999].

—*Linpack.* The Linpack library of linear system solver routines was first released in the late 1970s. It has been replaced by Lapack.

—*Eispack.* Eispack represents one of the first attempts to create a software library of high quality numerical routines. These eigensolvers have been replaced by routines in the Lapack library.

—*Fftpack.* This is a package of subprograms for performing Fast Fourier Transforms on periodic and other symmetric sequences. Routines are available for handling both real and complex data. See [Swarztrauber 1982] for more details.

—*Port.* The Port mathematical subroutine was a library of portable Fortran subroutines for numerical computation developed at Bell Laboratories in the 1970s [Fox et al. 1978b]. A number of these routines, dealing with the setting of machine dependent constants, storage management and error handling, were published as Algorithm 528 [Fox et al. 1978a]. The three routines that are used extensively in the CALGO codes are I1MACH, R1MACH and D1MACH which allow a programmer to gain access to a number of machine dependent constants.

The original routines were implemented in Fortran 66 and required a user either to uncomment data statements defining the constants for a particular machine or to set the values required. With the arrival of Fortran 90 all the values describing the underlying number systems on the machine may be obtained using intrinsic functions.

A number of different implementations of these routines are available including the original versions, a Fortran 90 implementation and a Fortran 77 implementation which automatically sets the required constants [Gay and Grosse 1999].

—*Error Handling.* The CALGO codes offer little consistency in the way in which internally discovered errors and failures are handled. Some routines just print a message and stop, others return an error flag and may, or may not, generate an explanatory message.

Algorithm 528 [Fox et al. 1978a] offered some support for error handling but, like the routines for dynamic storage allocation, this facility has been largely ignored by the scientific software writing community. Although more complex, versions of the Slatec error handling package [Jones and Kahaner 1983] have appeared in a number of published CALGO codes, there would appear to be a number of different versions of these routines. The one provided with this library is a Fortran 77 version using `CHARACTER` variables rather than integer arrays to store and manipulate string data.

Since error messages are always passed as character constants to the required error handling routines the above change has no other effects on the underlying software. Thus all calls of the form

```
CALL XERROR(23HSMOOTH -- NUM WAS ZERO.,23,1,2)
```

may be replaced by

```
CALL XERROR('SMOOTH -- NUM WAS ZERO.',23,1,2)
```

without problems. It should be noted that some of the arguments to the routines in this package are superfluous in Fortran 77, for example, the length of the message (argument 2 in the call to `XERROR` above) may be deduced from argument 1 using the `LEN` intrinsic function.

—*Others.* This section contains routines for generating random numbers and obtaining timing information along with the subroutine `machar` [Cody 1988] which may be used as a starting point for obtaining floating point characteristics.

The random number generator provided is by Schrage [Schrage 1979] and has replaced the large variety of different generators that were originally embedded within the algorithm collection. The implementation is self contained and consists of two routines, SEED, which takes an integer argument and is used to initialize the generator and a real function, RAND, which returns a single random real in the range $(0, 1)$. The implementation is in standard Fortran 77 and may easily be made to use the new Fortran 90 routines `random_seed` and `random_number`. The test drivers associated with all algorithms that require random numbers set the initial seed to a particular value to ensure that the sequence generated is repeatable across platforms and compilers.

The timing routine is a real function, SECOND, that returns the time in seconds since the start of program execution. Fortran 90 users may like to implement this using the standard Fortran 90 timing function `system_clock` or the Fortran 95 function `cpu_time`.

## 4. CODE IMPROVEMENTS

Early codes, written in Fortran 66, are often difficult to understand due, for example, to the lack of adequate control structures and the restricted use of integer expressions in array subscripts, do-loop ranges, etc. No attempt was made to update the codes by removing such restrictions although a number of restructuring tools are available for Fortran (for example, spag [Polyhedron Software 1997] and nag_struct [Numerical Algorithms Group Ltd. 1999]) which may be able to perform some of these improvements. However, the resultant code often requires manual tuning and this was considered to be too time consuming and error prone for a first upgrade.

Other software tools were used extensively on the algorithm codes. Once split into components, all algorithm files were stored using the source code control system CVS [Fogel 1999]. This allowed all code changes to be documented and previous versions of all files (including the original submission) to be easily regenerated.

The Edinburgh Portable Compilers (EPC) Fortran 90 compiler was chosen to be the project compiler as it allowed a comprehensive set of both compile and run time checks to be performed. Among the problems reported at compile time were

—the use of undeclared variables,

—the use of Hollerith constants in both assignment and format statements,

—the use of some inconsistent argument types in subprogram calls.

At run time the system checked, amongst other things, that

—all array indexes were within their declared bounds.

—all variables had been assigned a value before use, this included all array elements,

—actual arguments supplied in subprogram calls agreed in number and type with the subprogram definition. Note that these checks were only performed when actual calls were made.

A small number of algorithms could not be compiled with the EPC compiler without extensive source changes. These codes typically involved the illegal use of either common blocks or array arguments with the incorrect type. In such cases the more forgiving Sun f77 compiler was used. All the development work was performed on a variety of Sun workstations running the Solaris version 8 operating system.

In addition all code was checked by the NagWare Fortran 77 standard conformance checking tools, nag_pfort [Numerical Algorithms Group Ltd. 1999]. By concatenating the driver and source files it was possible to check the complete submission for adherence and consistency. As well as performing a number of global checks, the verifier reported a number of useful 'clutter' problems. These included such things as unreferenced labels (also reported by the EPC compiler), variables that were set but never used, and variables that were declared but never used. Among the global checks performed were the consistent use of common blocks and associated save statements, recursive calling sequences and possible problems with variables being passed to subprograms via common blocks and as an argument.

Several of the more minor problems reported by the compiler and the verifier could have been resolved using existing tools; for example, declaring all variables and removing those that have been declared but never used may be achieved by the NagWare tool, nag_decs. However, an early decision on the project was to keep the source codes as close to the original submissions as possible and, unfortunately, one side effect of using such tools is that they completely reformat the software. A small number of simple perl scripts [Wall et al. 2000] were thus constructed to

—insert declaration statements for all undeclared variables,

—replace Hollerith constants that appeared inside format statements by character constants,

—replace all array arguments declared to be of length one by assumed size arrays.

while making as few changes as possible to the original sources. The removal of unused variables and unreferenced labels was performed manually.

### Additional Portability Changes

Having cleaned up the submitted sources to bring them in line with the Fortran 77 standard, the next stage was to improve the portability of the code by ensuring that all system dependent parameters were, as far as possible, set using calls to the Port library functions R1MACH, D1MACH and I1MACH.

The R1MACH and D1MACH functions provide a reasonably effective way of obtaining the largest and smallest available floating point numbers and the machine epsilon for single and double precision respectively. These values need to be provided for a user's system either as hardwired constants or, if using Fortran 90, via the new numeric enquiry functions HUGE, TINY and EPSILON.

The original CALGO submissions included a variety of different methods of obtaining these constants, for example, the machine epsilon was set by

—hardwiring constants into the sources (using either DATA or assignment statements),

—using the Port library routines,

—using the routines DPMPAR and SPMPAR from the Minpack library [Moré et al. 1980],

—using the `epslon` function from Eispack [Hopkins and Slater 1993],

—computing the value by various means, for example,

```
      eps = one
   10 if (one .ne. eps+one) then
        eps = eps/two
        goto 10
      endif
```

```
eps = two*eps
```

To ensure that these values were set in a consistent manner it was, therefore, necessary to consider each algorithm source to ascertain whether it used any machine dependent constants and, if so, to set them via calls to the Port functions.

A second inconsistent use of machine dependent parameters occurred in the definition of the unit numbers for the standard input and output channels. The vast majority of the submitted codes required at most one read and one write channel. Although it could be argued that failure messages should be output on a separate error channel, the standard error and output streams were not distinguished. In the updated sources the standard input and output unit numbers are both set by calls to the Port library function, I1MACH, thus allowing a user the freedom to change them easily.

Open statements connecting named files to the standard input and output channels were removed. Any additional channels required are set via an open statement and these may require user changes to provide system or application dependent file names. Additionally, all output statements are now of the form

```
write(int_variable, format_specifier)output_list
```

which replaces an assortment of other statements including the use of print statements and list directed I/O.

Finally, a number of machine dependent constants which were required by algorithms were not explicitly covered by the Port functions and could not be computed from the available values. Such constants are currently set explicitly to the IEEE arithmetic values in a parameter statement; in a later release of the algorithms it is expected that an extended version of the Port routines will be used.

## 5. TESTING

No supporting test material was included with the base versions of algorithms 493 to 521 and in a number of the later algorithms data files were missing. The overall standard and scope of the provided test material differed widely, from simple example uses of the algorithm to extensive, self-checking, stringent test rigs. Relatively few of the submissions included sample output which made the task of checking the implementation time consuming.

If no main programs were available for an algorithm, simple example drivers were constructed. Where possible these implemented one or more of the sample problems defined in the accompanying published paper. In a number of cases it was necessary to consult further reference material to locate suitable problem definitions and/or data.

In order to help prevent the introduction of new errors during the upgrading process it was important to generate results using source as close to their submitted state as possible. Some changes were inevitable, for example, setting machine dependent constants, and, to obtain this initial execution, it was generally necessary to turn off the aggressive checking of the EPC compiler or, in several cases, use an alternative compiler that accepted non-standard conforming code. The results obtained in this way were used as a base version for comparing with those obtained from further code revisions.

The most common faults that arose at run time were

—*Array bound check failures.* These were caused in a number of different ways. First, there was the widespread malpractice of defining the last dimension of array arguments to be one. This was effectively a way of obtaining assumed size arrays in Fortran 66 which had very restrictive rules regarding the dimensioning of array arguments. Where the lengths of array arguments could easily be deduced, say from the comments in the source, this length was used in the upgraded code, otherwise assumed size arrays were imposed using a perl script.

Similar bad practice occurred when using common blocks to make workspace available throughout a package. In the algorithm code the labelled common block would contain an array dimensioned to be of length one while the driver program would contain a correctly dimensioned array. This contravened the standard which stated that all labelled common blocks should be of the same length. Here the problem is that, in order to produce standard conforming code, constants need to be defined in all routines that access the common blocks; these constants are either hardwired or are provided using parameter statements. Should the value need changing, either to increase the maximum size of problem solvable or to make more efficient use of memory, many source lines need to be edited and the code recompiled. Without automation this process is obviously prone to error. This is one reason why the use of common blocks to provide workspace should have been more actively discouraged.

Finally, there were a number of 'off-by-one' problems with array accesses where loops which accessed every element of an array using an index, say `I`, also attempted to access indexes `I-1` or `I+1`. These problems were commonly solved by guarding the offending statement or by redefining the loop limits and providing special case code.

It is the norm that arrays, especially those used for workspace, are declared to be much larger than required in order to make the driver program more flexible. This practice may well be masking subtle errors which are very difficult to detect. If accesses are out of range as far as the problem is concerned, but still technically in range because the array is larger than necessary, then, even if the element has not been assigned to, a value of zero is often returned due to the run-time system zeroing all data. If the access is just out of range then other program data may be accessed or overwritten; which particular data is affected will be system dependent. In both cases it is possible to obtain plausible, but incorrect, results. Finally, it should be noted that many compilers appear not to perform range checks on assumed size arrays even when bound checking is in force.

—*Unassigned variable accesses.* These are variables whose values are used before they have been explicitly assigned. On many systems all variables are set, by default, to zero; whilst this is often what is required, relying on such system dependent initializations is both dangerous and non-standard. Generally, unassigned variable errors occurred when an initialization has been completely overlooked, when an unusual flow of control through a routine causes an initialization to be bypassed, or when a variable is tested unnecessarily. In a very small number of cases it was due to the misspelling of a variable name. As mentioned above, it may also have been masking a problem with array bounds.

—*Mismatched arguments.* The Fortran 77 standard required that the actual arguments used in a subprogram call should match the dummy arguments in both number and type. It was rare that the number of arguments was found to be incorrect and this situation probably arose when unused arguments were removed from a routine definition but not from all of the calls.

The two most commonly occurring type-mismatching problems were

—The use of one type of workspace array, usually real, for both integer and real actual arguments to internal routine calls. Such misuse usually requires changes to the top level calling sequence to allow the provision of workspace of both types as well as changes to argument lists further down the call tree to ensure that the different types are available where they are needed.

—The use of simple variables for arrays when either the array argument was not expected to be accessed by the call or the actual array should have been of length one. This latter problem occurred frequently when the BLAS level 1 routine SCOPY was used to zero an array using code of the form

```
real zero, array(20)
parameter (zero = 0.0e0)
call scopy(20,zero,0,array,1)
```

where the second argument was expected to be a real array.

Infrequently there were single/double precision type mismatches but these tended to be either overlooked type declarations or misspelled variable names.

Both the EPC Fortran 90 compiler and the nag_pfort tool were used to check all argument lists provided the complete source code was available. At run-time the EPC compiler would treat a call with mismatched arguments as a fatal error.

For the vast majority of algorithms, the source changes required to correct the types of compiler detected errors described above led to codes that produced results equivalent to those obtained using the original software. In rare cases the submitted source codes generated incorrect results using one or more of the test compilers; where such circumstances arose more extensive changes were occasionally necessary to fix the problems; see, for example, Hopkins [Hopkins 2002].

In these latter cases, where source code changes were necessary to correct errors in the original submissions, articles describing the changes will be published as "Remarks" in ACM-TOMS, both to provide an audit trail of the changes and to notify current users of the routines of potential problems.

## 6. PACKAGING

The renovated codes are available via a Web site on a CD ROM. This includes all sources to the algorithms along with package sources and index information designed to allow users to locate suitable software for a given class of problem.

The sources are available as both *tar* and *zip* files to make access to the material straightforward for both Unix and Windows users. Each file contains the complete directory structure as described in Figure 2. The zip files were generated from the Unix files by using the *-l* flag to zip to translate the Unix end of line character into the PC convention of CR LF.

All index information was generated from the data available as part of Algorithm 620 [Hopkins and Morse 1990] which provides, in a compact format,

—full bibliographic details of the original paper along with any associated Remarks and Certificates,

—implementation language,

—the ACM modified Share classification, used to produce the Cumulative Index of the CALGO [Hopkins 2000], and the Gams [Boisvert et al. 1985] classifications. Gams (The Guide to Available Mathematical Software) allows a framework for end users and software librarians to deal with the ever increasing amounts of available mathematical and statistical software.
Both classification schemes provide a structured taxonomy of mathematical problems; the difference is that, while the Share scheme is restricted to just two levels, Gams is a variable level scheme which allows a much finer granularity to problem specification. For example, Share has just D1 to cover all integration problems; Gams has around 40 categories in up to four levels of classification.

Three indexes are currently provided

—a list of all the algorithms in publication order, most recent first,

—the ACM modified Share classifications,

—the first two levels of their Gams classifications.

The last two indexes consist of a page containing descriptions of each problem area with links to any relevant algorithms. Each algorithm entry in these indexes consists of title, keywords, links to the source files and a link to more detailed information on each algorithm.

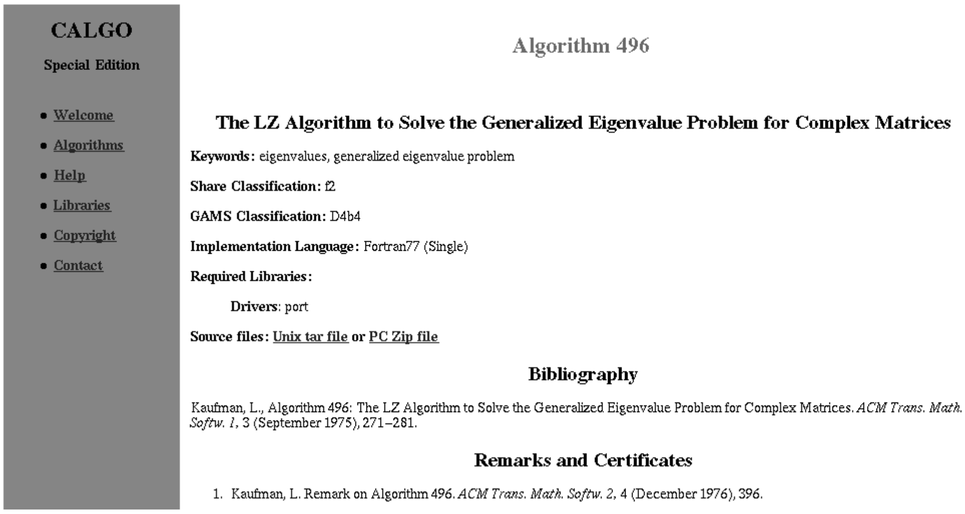
The *Full Information* page associated with each algorithm contains the title, full bibliographic details of the original publication along with any associated remarks and certificates, keywords, Share and Gams classifications, implementation language and available precisions, any required external libraries and links to the source files. An example is given in Figure 3.

The ACM holds the copyright to all the code originally published in the CALGO and continues to do so on the modified sources appearing on the CD. As with the original software, the new versions of the algorithms may be freely used provided that such use is not for profit. All commercial applications require a licence to be obtained from the ACM (see `http://www.acm.org/calgo/` for more details).

## 7. FUTURE IMPROVEMENTS

We aim to improve the algorithm codes still further in future releases. In some cases this means retrofitting modern software engineering practices and in others upgrading sources to take on board features available in newer versions of the implementation language. In particular, Fortran 90 offers a number of facilities which could be used to simplify the argument lists. Some of these improvements may be achieved by providing Fortran 90 wrapper routines to the existing Fortran 77 code while others require extensive changes to the sources.

Wrapper routines would relieve the user of having to provide workspace arrays (which could be replaced by automatic arrays), the leading dimension of multi-dimensional array (which could be replaced by assumed shape arrays) and other information which may be deduced from the problem-defining data. Far more effort would be required to restrict the scope of workspace arrays to just the portions of



CALGO

Special Edition

- Welcome
- Algorithms
- Help
- Libraries
- Copyright
- Contact

Algorithm 496

**The LZ Algorithm to Solve the Generalized Eigenvalue Problem for Complex Matrices**

**Keywords:** eigenvalues, generalized eigenvalue problem

**Share Classification:** f2

**GAMS Classification:** D4b4

**Implementation Language:** Fortran77 (Single)

**Required Libraries:**

**Drivers**: port

**Source files:** Unix tar file or PC Zip file

**Bibliography**

Kaufman, L., Algorithm 496: The LZ Algorithm to Solve the Generalized Eigenvalue Problem for Complex Matrices. *ACM Trans. Math. Softw. 1*, 3 (September 1975), 271–281.

**Remarks and Certificates**

1. Kaufman, L. Remark on Algorithm 496. *ACM Trans. Math. Softw. 2*, 4 (December 1976), 396.

Fig. 3. Example Full Information page

the call tree that require the data. Such modifications of the source code would appear to be well beyond the scope of current software tools.

However, a number of tools are available for performing some very useful source upgrades. Spag [Polyhedron Software 1997] and nag_polish95 [Numerical Algorithms Group Ltd. 1999] both offer the ability to translate Fortran 77 fixed format code into the new free format; in addition spag will replace old style Fortran do-loops with the new `do-end do` version with exit statements as appropriate. The nag_decs95 tool [Numerical Algorithms Group Ltd. 1999] can generate generic precision code by using the KIND specifier thus enabling a user to generate a version of the required precision by changing a single line and recompiling. In addition, nag_cbm95 [Numerical Algorithms Group Ltd. 1999] could be used to replace common blocks with modules.

Finally, whenever performance or reliability is an issue, Linpack, Eispack and user supplied linear algebra routines should be replaced by their counterparts in Lapack.

Before embarking on extensive changes to the sources, it would be necessary, for a large number of the algorithms, to improve test coverage of the codes so as to increase confidence that no new errors were being introduced.

With the web-based approach taken it would be relatively straightforward to improve the documentation accompanying the algorithms by providing links to the abstracts and accompanying papers within ACM's digital library.

## 8. CONCLUSIONS

The work so far invested in the CALGO algorithms which have appeared in ACM-TOMS has generated a far more easy-to-use and consistent library of software than was previously available. A large percentage of the algorithms have been tested using a strict-checking compiler and a standard-conformance tool which resulted in many hundreds of source changes being made. These changes have improved the portability of the software and removed minor errors and code clutter. The portability of the Fortran codes, in particular, has been improved further by ensuring that the majority of machine dependent constants are set using the Port library routines I1MACH, R1MACH and D1MACH. A number of software tools were used to generate a consistent layout for the sources.

Apart from a relatively few platform dependent codes or those implemented in languages for which no compiler was available, all algorithms were compiled and run. Each of these codes now has at least one accompanying driver program and makefile, data and results files are also provided.

Source material is available as both a tar file, for UNIX based systems and zip files for PC users.

## REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide* (Third ed.), Volume 9 of *Software, Environments, and Tools.* Society for Industrial and Applied Mathematics, Philadelphia.

Boisvert, R. F., Howe, S. E., and Kahaner, D. K. 1985. GAMS: A framework for the management of scientific software. *ACM Trans. Math. Softw. 11*, 4 (Dec.), 313–355.

Cody, W. J. 1988. MACHAR: A subroutine to dynamically determine machine parameters. *ACM Trans. Math. Softw. 14*, 4 (Dec.), 303–311.

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

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

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

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

Fogel, K. F. 1999. *Open Source Development with CVS.* The Coriolis Group, Scottsdale, AZ.

Ford, B. 1978. Parameterization of the environment for transportable numerical software. *ACM Trans. Math. Softw. 4*, 2 (June), 100–103.

Fosdick, L. D. 1975. Algorithms policy. *ACM Trans. Math. Softw. 1*, 1 (March), 5–6.

Fox, P. A., Hall, A. D., and Schryer, N. L. 1978a. Algorithm 528: Framework for a portable library. *ACM Trans. Math. Softw. 4*, 2 (June), 177–188.

Fox, P. A., Hall, A. D., and Schryer, N. L. 1978b. The PORT mathematical subroutine library. *ACM Trans. Math. Softw. 4*, 2 (June), 104–126.

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

Gay, D. M. and Grosse, E. 1999. Self-adapting Fortran 77 machine constants: Comment on algorithm 528. *ACM Trans. Math. Softw. 25*, 1 (Jan.), 123–126.

Herbold, R. J. 1960. Quad I. *Commun. ACM 3*, 2 (Feb.), 74.

Hopkins, T. 2000. Cumulative index for the Collected Algorithms from the ACM. CALGO Supplement.

Hopkins, T. 2002. Diary of an installation and a remark on Algorithm 639: To integrate some infinite oscillating tails. Technical Report 4-02 (March), University of Kent, Computing Laboratory, Canterbury, Kent, CT2 7NF, UK.

HOPKINS, T. AND SLATER, J. 1993. A comment on the Eispack machine epsilon routine. *ACM SIGNUM 28*, 4 (Oct.), 2–6.

HOPKINS, T. R. AND MORSE, D. R. 1990. Remark on algorithm 620. *ACM Trans. Math. Softw. 16*, 4 (December), 401–403.

JONES, R. E. AND KAHANER, D. K. 1983. XERROR, the SLATEC error-handling package. *Softw. Pract. Exper. 13*, 3 (March), 251–257.

KROGH, F. T. 1978. Algorithms policy. *ACM Trans. Math. Softw. 4*, 2 (June), 97–99.

KROGH, F. T. 1979. Algorithms policy (revised by W. Miller). *ACM Trans. Math. Softw. 5*, 2 (June), 129–131.

KROGH, F. T. 1982. Algorithms policy (revised by R.J. Hanson). *ACM Trans. Math. Softw. 8*, 1 (March), 1–4.

KROGH, F. T. 1990. Algorithms policy (revised by R. Renka). *ACM Trans. Math. Softw. 16*, 3 (Sept.), 293–296.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebraic subprograms for FORTRAN usage. *ACM Trans. Math. Softw. 5*, 3 (Sept.), 324–325.

MORÉ, J. J., GARBOW, B. S., AND HILLSTROM, K. E. 1980. User guide for MINPACK-1. Technical Report ANL-80-74 (Aug.), Argonne National Laboratories, Argonne, Ill.

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

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

RYDER, B. G. 1974. The PFORT verifier. *Softw. Pract. Exper. 4*, 4 (April), 359–377.

SCHRAGE, L. 1979. A more portable Fortran random number generator. *ACM Trans. Math. Softw. 5*, 2 (June), 132–138.

SMITH, B. T., BOYLE, J. M., DONGARRA, J. J., GARBOW, B. S., IKEBE, Y., KLEMA, V. C., AND MOLER, C. B. 1976. *Matrix Eigensystem Routines – EISPACK Guide* (second ed.), Volume 6 of *Lecture notes in computer science*. Springer-Verlag, New York.

SWARZTRAUBER, P. N. 1982. Vectorizing the FFTs. In G. RODRIGUE Ed., *Parallel Computations*, Volume 1 of *Computational Techniques*, pp. 51–83. New York, N. Y.: Academic Press.

WALL, L., CHRISTIANSEN, T., AND ORWANT, J. 2000. *Programming Perl* (Third ed.). Nutshell Handbook. O'Reilly, Sebastopol, California.

WITTE, B. F. W. 1968. Jacobi polynomials. *Commun. ACM 11*, 6 (June), 436–437.