DEVELOPING A SINGLE NUMERICAL ALGORITHMS LIBRARY FOR DIFFERENT MACHINE RANGES

Dr B. Ford and Dr D.K. Sayers, NAG Central Office, Oxford University Computing Laboratory, 13 Banbury Road, OXFORD OX2 6NN, England.

1. Introduction - The NAG Library

The Numerical Algorithms Group (NAG) arose from a meeting in Nottingham in May 1970 when representatives from the universities of Birmingham, Leeds, Manchester, Nottingham and Oxford and the Science Research Council's Atlas Laboratory at Chilton agreed to collaborate in the development of a numerical algorithms library for the ICL 1906A. Originally this activity was co-ordinated from Nottingham and because of this was known as the 'Nottingham Algorithms Group' during the first three years of its existence.

The group had three objectives:

(i) To develop for use on the ICL 1906A, a balanced, general purpose numerical library in Algol 60 and ANSI FORTRAN, that would satisfy the requirements of the majority of users of that machine. This was to be achieved without the library growing unwieldy, for a small library facilitates access, support and maintenance. Consequently is was agreed that algorithms, selected to fulfil particular requirements, were to be removed from the library whenever they were superceded.

(ii) To support the library with documentation for each routine and with documentation containing general guidance for the user on the selection of routines for his particular problem. The existing presentation is described in [1] but is currently under review, because the group attaches great importance to this aspect of the library and is constantly seeking ways of improving the clarity of its documentation.

(iii) To ensure confidence in the contents of the library by the maintenance of a test program library.

In 1972 the 1900 Universities User Group and representatives from British universities with IBM machines expressed interest in implementing the library for use on their machines. Subsequently representatives from universities with other machines wanted to implement the library for their local use. As a consequence of their interest the NAG library, originally intended as a library for one machine range, is now implemented on the ICL 1906A, ICL 1900, ICL 4100, ICL KDF9, IBM 360/370, ICL System 4, Burroughs 55/6700, CDC 6000/7000 and DEC PDP 10 machines. Consequently the original objectives for the library on one machine have been extended:

(iv) To implement the library on each of the machine ranges providing a computing service in British universities.

A further extension of NAG's activity has recently been initiated. This is the translation of the library into Algol 68. The main purpose of this paper, however, is to describe the problems that we have encountered in extending the library to several machine ranges.

SEAS Anniversary Meeting 1974

2. Development Stages

In this section we summarise the tasks and manpower involved in the development of a single library for many machines. Some of the problems which arise are discussed in greater detail in later sections of this paper.

The principal development stages are:

Contribution → Validation → Assembly → Implementation

Contribution: In specific subject areas a contributor has the responsibility for:

(a) choice of algorithm
(b) coding of the chosen algorithms
(c) provision of example and stringent test programs
(d) writing of documentation.

Each of these activities is a non-trivial exercise. To assist the contributor in task (a) the NAG Library Contents Committee (and in some areas, subject working parties) can advise on algorithm selection. This problem is discussed in section 3. For (b), (c), (d) carefully constructed standards are collated in an internal reference manual. Further discussion of coding practice is given in section 4 of this paper.

Validation: This is a shared responsibility. The principal participant is the validator. He, like the contributor, is chosen because of his expertise in particular subject areas. His task is to take the routines, test software and documentation provided by the contributor, and examine each component critically. The additional minimum level of validation expected is the application of at least one further test problem. After the validator is satisfied, the contributed material is passed to the NAG Central Office for assembly into the master library file system (mlfs). Before incorporation, the software is vetted by Central Office staff. This stage of validation concentrates more on formatting, adherence to language standards and coding practices than on algorithmic assessment. Some of the difficulties in the validation exercise are considered in section 5.

Assembly: Validated software is incorporated into the master library file system. The significance and function of this system are discussed briefly in section 5.

Implementation: Software from the master library file is sent by the Central Office to an implementation group associated with a particular machine-range. Such a group is responsible for the implementation of the routines, test software and documentation on its computer range. Its objective, to be achieved with minimal text alteration, is the production of an 'equivalent' library. Problems of implementation are considered in section 6.

The stages described above give an over-simplified view of the development process, which is presented as a one-way exercise from contributor to implementor. There are arrangements for the rejection or recycling of material, and facilities for the transfer of information in either

direction. In particular, implemented software is returned to the Central Office for incorporation into the master library file system. The experiences of the implementor can be relayed to the contributor formally or informally, and the mlfs provides a machine-based record of the software changes found necessary.

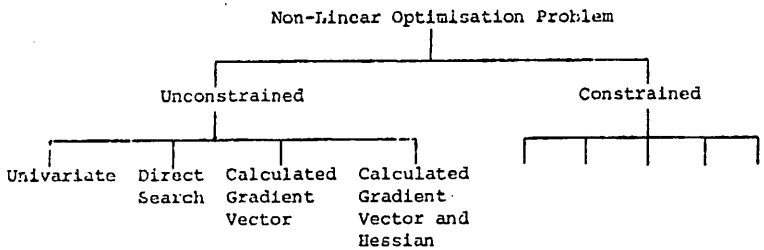
In this way, as successive marks of the contributed library are generated, the implementation process influences the contribution process. The development framework is fixed but the standards and constraints may not be. As new machine range implementations are attempted or new subject areas added, new problems may arise so the operational and technical standards might require further refinement.

Most of the work of contribution, validation and implementation is provided voluntarily by staff from universities and research establishments (Harwell and the National Physical Laboratory have been particularly active in this respect). The co-ordination of the activities of more than 100 individuals imposes an administrative burden. This has been allieviated by financial assistance from the Computer Board for Universities and Research Councils, which has enabled the NAG Central Office to expand. Their help has also allowed NAG to employ full-time machine-range implementation co-ordinators supported by the Computer Board.

3. Choice of Algorithm

The NAG library has a chapter structure based upon the Modified Share Classification, which is the best known international standard. Experience has shown, however, that this classification is inadequate in some areas, particularly linear algebra, operational research, statistics and non-linear optimisation.

The subdivision of a chapter can be important when algorithms are being selected. Linear algebra problems can be subdivided almost completely down to the level of individual algorithms. Problems in non-linear optimisation on the other hand can only be partially classified:



The class of problems which require direct search techniques, probably contains distinct subsets which are not yet clearly identified, and a similar situation undoubtably exists in many other problem areas. Ultimately we would like to define the subsets and provide algorithms to solve problems for each of them.

At the moment much of the onus of deciding upon the algorithms to be implemented is placed upon the contributor of the chapter. This is justified by the contributor being an expert in that particular field and consequently having knowledge of the latest developments in that area.

There are a number of characteristics we would hope to find in a good algorithm: (1) Stability (2) Robustness (3) Accuracy (4) Trustworthiness (5) Speed.

Numerical stability ensures that any errors introduced during the calculation do not grow unduly. Examples of unstable algorithms abound in ordinary and partial differential equations, where an unthinking quest for high order formulae can lead to disasterous consequences. Henrici [2] page 219 illustrates this by use of the formula

$$Y_n = 4y_{n-1} - (3+2h)y_{n-2}$$

to solve $y'=y$, $y(0)=1$, he uses the starting conditions $Y_0=1$, $Y_1=1.10517$ where Y_1 agrees with the true value of y to the number of figures given and where the step size h is taken to be 0.1. He states that at 1.0 the calculated value of y is in error by over 9.2, which means that the calculated solution has no correct figures.

Robustness is the ability of the algorithm to cope adequately with a wide range of situations, which may not be evident before steps of the algorithm have been taken. We might say that the domain of problems which the routine is able to accept is 'sufficiently large'. An example of a difficult situation might be a discontinuity in the derivatives supplied to a routine for solving an initial value problem in ordinary differential equations. Some algorithms might be led away from the correct solution by this and we would hope to avoid this eventuality by a better choice of method, if possible.

Naturally we would like to include an algorithm that is capable of achieving high accuracy, if requested, subject only to the limitations of particular computer upon which the algorithm is implemented. This is not particularly easy in the approximation of special functions where a different number of terms in the approximation may be required to accommodate the different word-lengths of the various machines.

Trustworthiness enables the user to have confidence in the results obtained using the algorithm. It implies that the requested accuracy is attained nearly all the time. The emphasis placed upon trustworthiness varies between individuals. It is our belief that a numerical library should consider it of prime importance, for an unsophisticated user frequently puts an alarming amount of trust in results obtained by the computer. In our experience the typical user hardly ever verifies his results by using consistency checks, even though these may be particularly easy to incorporate.

Finally and of lowest priority is the requirement of speed. Clearly if two algorithms solve the same class of problems and satisfy the previous conditions then that which requires fewer operations is judged to be better. Sometimes however the requirements of high accuracy and speedy calculation clash and it may be desirable to have fast methods for low

accuracy calculation. Examples of this are the lower order Runge-Kutta methods in ordinary differential equations. Such methods are excused the requirements of high accuracy, but they would need to have a more accurate counterpart to fill this gap in the library, wherever this was feasible. (High accuracy multi-dimensional quadrature is probably a case where this may not be possible.)

In order to choose between algorithms it is useful to have some measure of the characteristics above. An approach pioneered by Professor T.E. Hull et alia at Toronto [3] is to record details of an algorithms performance over a carefully selected set of test problems. This approach is subject to criticism because the results obtained may well reflect the standard of coding of the algorithm and not the algorithm itself. Professor Hull and his colleagues were aware of these difficulties and took pains to use the authors' coding whenever possible and to state along with their findings, the source of the implemented algorithm. This is probably the only realistic way of tackling the problem and we would hope that there can be universal agreement on the characteristics to be measured in this way and on a suitable set of test problems in each subject area.

An additional difficulty in the selection of algorithms for a multi-machine library, occurs when an algorithm performs well on one machine and does not do so on another. For instance consider a high fixed order method for solving ordinary differential equations, started independently by some means, which uses the Nordseick formulation to hold past values. If the error estimate is used to control step size the method would behave erratically on a machine with a small word-length because of the high order differencing but might be acceptable on machines with a larger word-length. It would be undesirable to have different libraries for each machine range because a user may wish to transfer his program to another machine for a number of reasons. He may wish to use the more powerful regional computer, or send his program to a colleague at another university or he may himself move. Consequently the library contents must be essentially the same to facilitate a ready transfer.

4. Coding the Algorithm

Each machine range has, of course, different compilers associated with it. We are concerned with Algol 60, Fortran and more recently with Algol 68. Regrettably the compilers do not always accept the full language and particularly in the case of Fortran there are often local dialects. This means that we have to be very careful when coding an algorithm to ensure that the source code can be correctly compiled on all the different machine ranges. A failure to do this would mean that we had to hold several versions of a coded algorithm and would increase the problems of maintenance.

Just as we could distinguish five desirable features for a numerical algorithm so is it possible to recognise five properties that we would hope to find in the coded routine:

(1) It should compile
(2) It should faithfully reflect the algorithm
(3) It should be convenient to use

(4) The code should be easy to understand
(5) The routine should be reliable.

To obtain the first property on all the machine ranges covered by NAG it has been agreed to write the library in subsets of Algol 60, ANSI Fortran and Algol 68. This has the added advantage of conforming to many international standards, but rules out some of the more powerful language constructs, such as those of Extended Fortran. Experience suggests that this is not a great disadvantage at run-time because of the little extra cost of the object code ultimately generated. Writing to conform to a subset of a language also prohibits the generation of optimised code which relies upon the idiosyncracies of a particular compiler. These can be particularly dangerous. If the program

```
      REAL A
      READ(1,100) A
100   FORMAT(E9.4)
      WRITE(2,200) A
200   FORMAT(1H ,E9.4)
      STOP
      END
```

is run on an ICL 1906A computer, taking data
0.1E1
we obtain the results

```
.1000E 01
```

The CDC 7600, using the FTN compiler and using the same data produced

```
* RECORD NO. 1    0.1E1

*ERROR DATA INPUT*   FORMAT NO. 100  *DATA OVERFLOW*>*
```

The difference between the results is explained by the action of the compilers on encountering a space to the right in an exponent field. The CDC interprets this as zero whilst the 1906 ignores it. Such inconsistency between manufacturers is deplorable and may result in many undetected errors appearing in a user's program, should he be used to the alternative convention.

The solution adopted by NAG in such a situation is to insist upon the data exactly satisfying the format used. Whilst such restrictions may appear irksome, a reliance of the library on compiler idiosyncracies, besides giving the library a very limited application, would lead to it having a short and precarious existence because of the high rate of replacement and updating of language compilers.

The second desirable feature of the coded routine, that it should faithfully reflect the algorithm, presents problems of testing, which we discuss in the next section.

The convenience to users of the routine is a particularly important

important property of a library routine. In an effort not to deter would-be users of the library NAG tries to keep the calling sequence of its routines as simple as possible. It is important to encourage users to use good numerical methods and, as they have varying degrees of numerical experience and mathematical knowledge, we try to avoid having too many constants and control parameters in the routine heading. Often the user has little idea how to set these and little interest in investigating them. He requires the routine, if possible, to do this work for him.

The typical user of the NAG library never sees the source text of the routines, although by special arrangement he may, if he has sufficient reason. Nevertheless validators and other machine implementors have to understand the contributor's source code in order to carry out their own tasks. It is therefore highly desirable that the contributor provide adequate comments within his software. Understanding is also improved by writing the routine according to some structure. The report by Hull and Enright [4] illustrates this point. In the future all NAG software will be tidied by automatic processing and it is to be hoped that this will improve the understanding of the routines. The software, SOAP [5] used for the Algol 60 routines, and test programs, is particularly impressive as may be judged from the results obtained by processing the code:

```
'PROCEDURE' CHOLDET 1 (X,A,P,D1,D2,FAIL)!
 'VALUE'
X! 'INTEGER' D2, N! 'REAL' D1! 'ARRAY' A,P! 'LABEL' FAIL!'BEGIN' 'INTEGER'
 I,J,K!
'REAL' X,Y!! D1:=1! D2:=0! 'FOR' I:=1 'STEP' 1 'UNTIL' N 'DO' 'FOR'
 J:=I 'STEP' 1
'UNTIL' N 'DO' 'BEGIN' INNERPROD(1,A[I,J]-A[-K,I],0,A[K,I],A[J,K]
,X,N,X)!
Y:=-X! 'IF' J=I 'THEN' 'BEGIN' D1:=D1*X! 'IF'X<0'THEN' 'GOTO' FAIL!
 L1: 'IF' ABS(D1)
>1 'THEN' 'BEGIN' D1:=D1*0.0625! D2:=D2+4! 'GOTO' L1 'END'! L2:
'IF' ABS(D1)=
0.0625 'THEN' 'BEGIN' D1:=D1*16! D2:=D2-4! 'GOTO' L2 'END'! P[I]
:=1/SQRT(X)
'END' 'ELSE' A[J,I]:=Y*P[I] 'END' I! 'END' CHOLDET 1!
```

to obtain:-

```
'PROCEDURE' CHOLDET 1(X, A, P, D1, D2, FAIL)!
   'VALUE' N!
   'INTEGER' D2, N!
   'REAL' D1!
   'ARRAY' A, P!
   'LABEL' FAIL!
   'BEGIN'
   'INTEGER' I, J, K!
   'REAL' X, Y!!
   D1 := 1!
   D2 := 0!
   'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
      'FOR' J := I 'STEP' 1 'UNTIL' N 'DO'
         'BEGIN'
         INNERPRODD(1, 1, I = 1, - A[I, J], 0, A[I, K], A[J,
            K], X, N, X)!!
         X := - X!
         'IF' J = I 'THEN'
            'BEGIN'
            D1 := D1 * X!
            'IF' X < 0 'THEN'
               'GO TO' FAIL!
         L1 :
            'IF' ABS(D1) = 1 'THEN'
               'BEGIN'
               D1 := D1 * 0.0625!
               D2 := D2 + 4!
               'GO TO' L1
               'END'!
```



```
         L2 :
            'IF' ABS(D1) < 0.0625 'THEN'
               'BEGIN'
               D1 := D1 * 16!
               D2 := D2 - 4!
               'GO TO' L2
               'END'!
            P[I] := 1 / SQRT(X)
            'END'
         'ELSE'
            A[J, I] := X * P[I]
      'END' J!
'END' CHOLDET 1!
```

It cannot of course insert comments or alter the underlying structure of the program. Consequently it is the responsibility of our contributors to ensure that the code reaches an acceptable standard in these respects.

A well-structured and tidy program or routine goes a long way towards ensuring that it is reliable. By this we mean that the code will not break down causing a machine fault, such as overflow, during a calculation. Consequently all possibilities must be anticipated by the programmer and dealt with adequately. Additionally it implies that obviously poor coding practices, such as the careless testing for equality between real quantities, are to be avoided.

The programmer is encouraged to cross-reference other NAG routines in order to achieve higher reliability, for not only are these routines thoroughly tested, but they are written by experts in that particular field. This policy has the additional advantage of keeping to a minimum the length of new coding required for a new algorithm.

It is at this stage that machine dependencies can be isolated. Tests which are machine dependent are coded so that automatic substitution of machine numbers is possible when different versions of the library are issued. Other machine dependencies are treated slightly differently. The quantity 'MACHEPS', for instance, defined to be the smallest real number such that 1.0+MACHEPS>1.0, occurs frequently in the linear algebra routines. It is evaluated, where necessary, once within each library routine by a call to a small sub-library, local to the particular machine range.

The sub-library is termed 'the constants and utilities library' and includes in the constants chapter values for π and γ, the Euler constant, evaluated to the correct machine precision. Other constants convey information about the particular machine and include the largest integer and the largest positive real floating-point number exactly representable on the machine. A separate chapter of this library is devoted to a number of inner-product routines, which are called whenever they are required by a NAG routine.

By placing the inner-products in the constants and utilities library and accepting the cost of a routine call whenever they are required we obtain one major benefit. It becomes possible to regulate for each routine of an implementation whether inner-product accumulation is performed in single or double length arithmetic. The choice between the two is often influenced by whether the machine has hardware or software extended precision, for in the latter case the overheads are more expensive. Some numerical

techniques however demand extended precision arithmetic; for instance the method of residual correction for solving a set of n×n non-singular linear algebraic equations, requires the calculation of inner-products to greater than single-precision. It is nevertheless a very useful method and it may or may not be worth the cost of software extended precision to include it within the library.

5. Testing the Coded Algorithm

In many ways the testing of a coded algorithm presents the greatest problem of all. We cannot expect a program to give identical results on different machines, because of the differences in word-length, rounding and the available elementary functions.

The coded algorithm must satisfy two conditions. It must be acceptable to the compilers on all the various machines and it must efficiently solve the type of problem it is designed to solve, given correct data.

The first of these conditions is relatively easy to check, because we have access to compilers which require strict language standards. These indicate any discrepancies between the code and many of the standards we have imposed. In Fortran this is particularly useful, because mixed type arithmetic and non-ANSI array subscripts can very easily slip in without checking. The PDP 10 and Burrough's Algol 60 compilers require declarations in a specified order, a further restriction on the language. Although strictly speaking this must be regarded as a compiler error we intent to comply with this order, because it imposes little constraint on the library.

The second problem is much more difficult and is largely unsolved; although we hope that as the machine dependencies are isolated the problem will be eased.

Some branches of numerical analysis present little difficulty. Many sections of linear algebra, for example, cause little trouble as the routines are moved from machine to machine. This is undoubtedly because the command path in these routines is relatively simple, and because the only errors that arise are due to the finite word-length of the machine.

Slightly more troublesome are those problems which need some type of approximation and which consequently give rise to a single local truncation error. The evaluation of special functions lies in this category and the approximation used has to vary slightly from machine to machine.

Certain problems are more sensitive to machine differences than others. Such problems are termed ill-conditioned and may easily occur in practice. Wilkinson [6] gives an example showing the extreme sensitivity of one root of a certain polynomial to a slight perturbation in one of its coeffi-cients. In this case the roots are well-separated, lying at -1,-2,..,-20 and the polynomial is of high order, but if we consider the simple quadratic

$$\tfrac{1}{2}x^2 + \sqrt{2}x + 1 = 0 \ ,$$

which has a double root at $x = -\sqrt{2}$ we can see that the results produced by a root-finding algorithm may be crucially affected by the evaluation of $\sqrt{2}$ on the computer. Analysis of the program below shows that the principal

error is $2\sqrt{\varepsilon}$, where the calculated square root is $(1+\varepsilon)\sqrt{2}$

```
      DOUBLE PRECISION B,51
      REAL B,X1,X2
      B=SQRT(2.0)
      B=DBLE(B)+DBLE(B)-2.
      IF (B) 1,2,3
1     X1=-B
      B=DSQRT(-B)
      X2=SNGL(B)
      WRITE(2,100) X1,X2
100   FORMAT(14H COMPLEX ROOTS,E20.10,18HPLUS OR MINUS I,E20.10)
      STOP
2     X1=B
      WRITE(2,200) X1
200   FORMAT(12H EQUAL ROOTS,E20.10)
      STOP
3     B=DSQRT(B)
      B=DSIGN(B,DBLE(-B))+DBLE(B)
      X1=2.0/B
      X1=SNGL(B)
      X2=SNGL(B1)
      WRITE(2,300) X1,X2
300   FORMAT(11H REAL ROOTS,2E20.10)
      STOP
      END
```

If ε is negative we generally obtain complex roots, whilst positive values of ε produce errors magnified by the square root. This is verified by the following results:

ICL 1906A		$-1.4142 \pm 0.000003i$
CDC 7600	(unrounded)	$-1.4142 \pm 0.0000009i$
CDC 7600	(rounded)	$-1.4142 \pm 0.0000009i$
PDP 10		-1.41434 and -1.41404
IBM 370/155		$-1.4142 \pm 0.00104i$

To add to the difficulties of testing a routine some numerical areas require complicated algorithms whose paths are determined by previously computed quantities. Examples of such areas are minimisation and ordinary differential equations. In the latter case, for instance, the algorithm may have to decide how far it can reliably advance the solution at the next step, on the basis of previously computed information. If at some stage the calculated quantities cause the algorithm to follow one path on one machine and a different path on another then the subsequent behaviour may differ markedly. This has significance when we consider the problem of testing the coded algorithm for although we may be able to test each major path through an algorithm on one machine by supplying one set of test problems, these problems may not suffice for a different machine.

This is typified in the extreme case by a PDP 10 implementation of Gear's method for the solution of a stiff system of ordinary differential equations. Without recourse to double-length the PDP 10 has only 7 decimal places and yet high order differences are used in the algorithm to determine the command path. We cannot expect the algorithm to follow the same command path on this machine as it would on one with a longer word-length, such as the CDC 7600.

Examples such as this demonstrate the impracticability of generating a

collection of test problems to examine all the command paths on all the different machines. Failing this we must force the routine to satisfy as many hurdles as we can contrive.

The routine is required to solve successfull a single well-conditioned problem. The contributor is asked to supply this and he is asked to construct it in such a way that the results produced will not vary between machine ranges. The problem is termed the example program and is included, along with the results obtained, in the documentation for the routine to illustrate its use.

We may then present a set of harder problems to the routine in the hope of exercising it thoroughly. These are called the stringent test cases and are provided by the author of the routine and sometimes additionally by the validator, who also pursues his own methods to satisfy himself that the routine is sound.

Ideally we would like the stringent test cases to encompass all the command paths of the routine, at least on one machine. For Fortran we have available on the 1906 a version of BRNANL [7] which enables us to identify parts of the code which are not exercised by the stringent test cases. The program below has been specially constructed to illustrate this.

```
      REAL A,B,C
1     READ(1,100) A,B,C
100   FORMAT(3F10.4)
      CALL QUAD(A,B,C)
      IF (A.GT.-5).) GO TO 1
      STOP
      END
      SUBROUTINE QUAD(A,B,C)
      REAL A,B,C,X1,X
      IF (A.NE.0.) GO TO 2
      IF (B.NE.0.) GO TO 1
      WRITE(2,100)
100   FORMAT(32H NO TERM IN INDEPENDENT VARIABLE)
      RETURN
1     X=-C/B
      WRITE(2,200)
200   FORMAT(18H SINGLE ROOT AT X=,E20.10)
      RETURN
2     X=B*B-4.0*A*C
      IF (X) 3,4,5
3     X=SQRT(-X)/(2.0*A)
      X1=-B/(2.0*A)
      WRITE(2,300) X1,X
300   FORMAT(14H COMPLEX ROOTS,E20.10,4H+OR-,2H I,E20.10)
      RETURN
4     X=-B/(2.0*A)
      WRITE(2,400) X
400   FORMAT(12H EQUAL ROOTS,E20.10)
      RETURN
5     X=SQRT(X)
      X1=SIGN(X,-B)-B
      X=C/(A*X1)
      WRITE(2,500) X1,X
500   FORMAT(11H REAL ROOTS,2E20.10)
      RETURN
      END
```

```
  0.      0.      0.
  1.      2.      1.
  1.      1.      1.
  1.      3.      1.
-50.     60.      1.
```

. . . changed by BRNANL into the logically similar code:

```
      LOGICAL BOOLEA                                                    1
      REAL A,B,C                                                        1
      CALL MONITE(0000,1)                                               1
1     CALL MONITE(0002,2)                                               2
      READ(1,100) A,B,C                                                 2
100   FORMAT(3F10.4)                                                    2
      CALL QUAD(A,B,C)                                                  2
      BOOLEA=A.GT.-50.                                                  2
      IF(BOOLEA)CALL MONITE(0003,2)                                     3*
      IF(BOOLEA)    GO TO 1                                             3*
      CALL MONITE(0004,2)                                               4
      CALL MONITE(0000,3)                                               4
      STOP                                                              4
      END                                                               4
      SUBROUTINE QUAD(A,B,C)                                            4
      LOGICAL BOOLEA                                                    4
      REAL A,B,C,X1,X                                                   4
      CALL MONITE(0005,2)                                               5
      BOOLEA=A.NE.0.                                                    6
      IF(BOOLEA)CALL MONITE(0006,2)                                     6*
      IF(BOOLEA)    GO TO 2                                             6*
      CALL MONITE(0007,2)                                               7
      BOOLEA=B.NE.0.                                                    7
      IF(BOOLEA)CALL MONITE(0008,2)                                     8*
      IF(BOOLEA)    GO TO 1                                             8*
      CALL MONITE(0009,2)                                               9
      WRITE(2,100)                                                      9
100   FORMAT(32H NO TERM IN INDEPENDENT VARIABLE)                       9
      RETURN                                                            9
1     CALL MONITE(0010,2)                                               10
      X=-C/B                                                            10
      WRITE(2,200)                                                      10
200   FORMAT(18H SINGLE ROOT AT X=,E20.10)                              10
      RETURN                                                            10
2     CALL MONITE(0011,2)                                               11
      X=B*B-4.0*A*C                                                     11
      IF (X) 3,4,5                                                      11
3     CALL MONITE(0012,2)                                               12
      X=SQRT(-X)/(2.0*A)                                                12
      X1=-B/(2.0*A)                                                     12
      WRITE(2,300) X1,X                                                 12
300   FORMAT(14H COMPLEX ROOTS,E20.10,4H+OR-,2H I,E20.10)               12
      RETURN                                                            12
4     CALL MONITE(0013,2)                                               13
      X=-B/(2.0*A)                                                      13
      WRITE(2,400) X                                                    13
400   FORMAT(12H EQUAL ROOTS,E20.10)                                    13
      RETURN                                                            13
5     CALL MONITE(0014,2)                                               14
      X=SQRT(X)                                                         14
      X1=SIGN(X,-B)-B                                                   14
      X=C/(A*X1)                                                        14
      WRITE(2,500) X1,X                                                 14
500   FORMAT(11H REAL ROOTS,2E20.10)                                    14
      RETURN                                                            14
      END                                                               14
```

Where the numbers in columns 72-80 indicate different blocks of code which must be executed if the first statement of the block is executed and where calls to a sub-routine MONITE, with values for two integer parameters, have been inserted at the head of these blocks. The parameters can be used with a user-supplied MONITE routine, such as that below, to analyse the run-time path of the program.

```
      SUBROUTINE MONITR(N,M)
      COMMON/PATH/MAXBB,PATHS
      DIMENSION PATHS(100)
      INTEGER MAXBB,PATHS
      IF (N.GT.100) RETURN
      IF (N.EQ.0) GOTO 3
      IF(M.GT.0) GO TO 2
      MAXBB=N
      PATHS(1)=1
      DO 1 I=2,100
      PATHS(I)=0
    1 CONTINUE
      RETURN
    2 MAXBB=MAX(MAXBB,N)
      PATHS(N)=PATHS(N)+1
      IF (N.EQ.2) RETURN
    3 WRITE(6,100) (PATHS(I),I=1,MAXBB)
  100 FORMAT(1H ,8I10)
      RETURN
      END
```

When assembled, the modified program and subroutine produced the following results on the Oxford 1906A:-

```
 NO TERM IN INDEPENDENT VARIABLE
 EQUAL ROOTS   -0.10000000000E 01
 COMPLEX ROOTS   -0.50000000000E 00+0I-  I    0.86602540378E 00
 REAL ROOTS   -0.52360679775E 01   -0.19098300563E 00
 REAL ROOTS   -0.12190977356E 03    0.13064016252E-03
         1         5         4         1         5         4         1         0
         1         0         4         1         1         2
```

The zeros in PATH(8) and PATH(10) indicate that the blocks of code headed by MONITE(8,2) and MONITE(10,2) have not been exercised. This is because the case of the quadratic degenerating into a single linear equation has not been tested.

Finally we may examine the code for slight errors. SOAP, the tidying program, is useful in this respect for it can give, in addition to a well-formatted routine, a list of constants and identifiers used, along with the line numbers in which they occur. For the example given earlier, SOAP produces the following additional information about the routine.

AN INDEX OF ALL CONSTANTS USED IN THE PROGRAM

```
0.
      USED ON LINES,    30      25      20      14      10
0525.
      USED ON LINES,    30      25
1.
      USED ON LINES,    36      23      14      14      14      12
 11       11        9
16.
      USED ON LINES,    32
4.
      USED ON LINES,    33      26
```


AN INDEX OF ALL IDENTIFIERS IN THE PROGRAM

```
A. 'ARRAY' ON LINE     4
      USED ON LINES,    39      14      14      14

ABS. NOT DECLARED
      USED ON LINES,    30      23

CHKDIM1, 'PRECISION' ON LINE     0
      NOT USED

EM. 'REAL' ON LINE     3
      USED ON LINES,    32      32      30      25      25      23
 17       10        9

EQ. 'INTEGER' ON LINE     2
      USED ON LINES,    33      33      24      26      13

FAIL. 'LABEL' ON LINE     5
      USED ON LINES,    21

I. 'INTEGER' ON LINE     7
      USED ON LINES,    32      30      36      17      14      14
 14       12       11

IHERRMSG, NOT DECLARED
      USED ON LINES,    14

J. 'INTEGER' ON LINE     7
      USED ON LINES,    39      17      14      14      12

K. 'INTEGER' ON LINE     7
      USED ON LINES,    15      15      14

L1. 'LABEL' ON LINE    27
      USED ON LINES,    27

L2. 'LABEL' ON LINE    29
      USED ON LINES,    34
M. 'INTEGER' 'VALUE' ON LINE     1
      USED ON LINES,    12      11

P. 'ARRAY' ON LINE     4
      USED ON LINES,    39      36

SQRT, NOT DECLARED
      USED ON LINES,    36

X. 'REAL' ON LINE     8
      USED ON LINES,    39      38      20      19      16      16
 15

XX. 'REAL' ON LINE     9
      USED ON LINES,    15
```

Having been tested and processed in the manner described above, the contributed software is assembled for incorporation into the master library files. Despite the precautions taken, errors are inevitably missed at the testing stage. Exposure to a large body of users (the NAG library is available in all British universities' computer centres whose total number of users is about 25,000) is the final stage in testing. There is an error handling scheme in operation. As the validation procedure is strengthened the number of serious errors reported remains reassuringly low. Moreover these errors and any others are all investigated and appropriate corrective action is taken for each implementation.

6. Master Library File System

The source text of the library will vary from one machine range to another. It may also change in time as corrections or improvements are made or new items added. A secure environment for the source text for the various implementations of the routines and test programs is essential if the

integrity of the code is to be maintained. Such a system has recently been developed [8].

The master library file system operated by the Central Office, stores all existing marks of each implementation in a compact form, so that common lines of code are only stored once. It is possible to compare different marks and implementations of a routine and to see the changes which were found necessary to obtain one from the other. The instructions needed to do this can be machine generated.

The system is sufficiently flexible to lead us to expect that from its contents it will be possible to generate a good attempt at a numerical library for a new machine range, given details of the radix and arithmetic.

An interesting problem associated with the system is in the 'base' machine version, i.e. the version initially incorporated and from which other machine ranges 'diverge' if necessary. Since most contributors have developed their software on ICL 1900's the early base version is biased towards that machine range. In the new series of master library files now established that bias will largely disappear. The base version will be inserted in a generalised form (known as the G-version). In this way the number of records deemed in common between implementations can be increased.

7. Implementing the Library

Implementors receive from the Central Office software extracted from the master library file system. They may also receive comparative information to assist in the implementation. This information, provided by the system, may indicate for example, what changes were made since the previous mark, or the alterations found necessary by another implementation group.

The first problem for implementors to overcome is the conversion of the received software into a convenient local form. Magnetic tapes are usually used for inter-machine transfer so each implementation group must have tape-handling software for the necessary structural and character transformations (the Central Office must also maintain utility software for the reverse transformation when the implemented software is returned).

The preliminary conversion process can be non-trivial particularly for Algol 60. Different compound basic symbols are used in the various implementations of this language and different representations are employed. A more awkward problem is that of input and output procedures which are extensively used in the testing software. To combat the lack of standardisation here, implementors have directly adapted the provided input/output statements to suit their own machine variants or provided 'jacket' procedures with the same specification. Wherever possible systemmatic textual changes are performed mechanically and not by hand-editing.

At the end of this initial conversion phase which would include the adjustment of machine constants the implementation proper can commence. The implementor hopes to have compilable routine and test program software.

Even when the library and test programs compile they may not necessarily run satisfactorily. The results obtained on the 1906A (which are distributed on the library tape) are compared with the local results, for any significant discrepancies. If there are, modifications may be required. For instance a PDP 10 implementation of a variable order Adams method for solving an ordinary differential equation cannot be allowed to use as high an order formula as a CDC 7600 implementation. These finer tunings have to be made by hand, but some implementors, the PDP 10 group in Oxford and the Burroughs group in Edinburgh, have managed to automate the previous processes with some success.

The aim of the implementor is to produce a library similar to that available on other machine ranges. To do this the contributed library and its associated test software are adapted with minimal alteration so that the stringent test programs produce results which are in as near agreement with the distributed results as can be reasonably expected. This is not ideal but there is probably no perfect solution to the problem. Automatic results comparison has a useful part to play in the implementation process. For instance, in the final check of a library ready for release, the large scale running of the associated example programs is most suited to results checking by software. The implementation process as a whole, however, should never be an entirely machine-based activity.

The emphasis on mechanical aids for implementation (e.g. machine-based results checking, text editing by software) is not simply a matter of convenience or consistency. The scale of the operation must also be considered. At Mark 4, for instance, a group starting its first implementation of the Algol 60 and FORTRAN libraries has about 500 routines and 800 example or stringent test programs to process. Additional implementation information for users must also be provided for the library documentation.

When the implementation of a mark is considered complete, the software is returned to the Central Office for inclusion in the master library file system. The system is then regarded as holding the definitive version of that implementation. This return of implemented software is central to the notion of a single library for several machines.

8. Conclusion

Some progress has been made towards the creation of a reliable numerical algorithms library to run on a number of machine ranges. The selection of algorithms is still a matter of personal judgement, but in this we may be increasingly guided by groups undertaking systematic testing of algorithms. The testing of the coded routine (or alternatively obtaining a proof of correctness), is probably the most difficult problem to be solved, but we hope that in this area too progress can be made.

Acknowledgements

The authors are grateful to Mr S.J. Hague for his guidance in the preparation of this paper. We are indebted to Dr D. Taylor and Mrs J. Clarke who enabled us to obtain the quoted results on the IBM and PDP 10 machines and to Hawker Siddeley Aviation Limited who provided us with a version of SOAP for the 1906A.

References

[1] Lill, S. User documentation for a general numerical library: The NAG approach, Software for Numerical Mathematics, Academic Press, London 1974.

[2] Henrici, P. Discrete Variable Methods in Ordinary Differential Equations. Wiley, New York 1968.

[3] Hull, T.E., Enright, W.H., Feller, B.M., and Sedgewick, A.E. Comparing numerical methods for ordinary differential equations. SIAM J. Num. Anal. 9, 603-637.

[4] Hull, T.E., and Enright W.H., A structure for programs that solve ordinary differential equations, University of Toronto Technical Report No. 66, May 1974.

[5] Scowen, R.S., Allin, D., Hillman, A.L., and Shirnell, M. SOAP - A program which documents and edits ALGOL 60 programs, Comp. J. pp 133-135, Vol. 14, No. 2, 1971.

[6] Wilkinson, J. Rounding Errors in Algebraic Processes, H.M.S.O., London, 1963.

[7] Fosdick, L.D. BRNANL, A fortran program to identify basic blocks in fortran programs. University of Colorado Report # CM-CS-040-74, March 1974.

[8] Hague, S.J., and Richardson, M. The NAG master library management system. To appear.