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 test 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 program. After the validator is satisfied, the contributed material is passed to the NAG Central Office for assembly into the master library file system (NLFS). 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 contribute formally or informally, and the M/S 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 alleviated 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:

```
<p>| Non-Linear Optimisation Problem |</p>
<table>
<thead>
<tr>
<th>Unconstrained</th>
<th>Constrained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univariate</td>
<td>Direct</td>
</tr>
<tr>
<td>Direct</td>
<td>Calculated</td>
</tr>
<tr>
<td>Search</td>
<td>Gradient</td>
</tr>
<tr>
<td>Gradient</td>
<td>Vector and</td>
</tr>
</tbody>
</table>
```

The class of problems which require direct search techniques, probably contains distinct subsets which are not yet clearly identified, and a similar situation undoubtedly 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 disastrous consequences. Bari et al. [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 \( y \) 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 speed 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 [1] is to record details of an algorithm's performance over a carefully selected set of test problems. This approach is subject to criticism because the results obtained may 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 differences 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 Algorithms

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 idiosyncrasies 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

```
.1000 0!
```

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

```
* RECORD NO. 1 0.1E1
```

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 ICL ignores it. Such inconstancy 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 idiosyncrasies, 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 them 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 Hall 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:

```

```
to obtain:

```

```

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 \( \text{MACHEXP2} \), for instance, defined to be the smallest real number such that \( 1.0 \cdot \text{MACHEXP2} \cdot 1.0 \), 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 the constants chapter values for \( x \) and \( y \), 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 compile 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 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 Algon 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 intend 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 simple 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 coefficients. 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

\[ x^2 + \sqrt{x} + 1 = 0, \]

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 \( \sqrt{x} \) on the computer. Analysis of the program below shows that the principal error is \( 2\sqrt{x} \), where the calculated square root is \( (1+\epsilon)^{1/2} \).

\[
\begin{array}{ll}
\text{error in } 2\sqrt{x}, \text{ where the calculated square root is } (1+\epsilon)^{1/2} \\
\end{array}
\]
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 user of the library has asked us 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 BMDNL [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 for this.

\[
\begin{align*}
&1 \text{ REAL} \ 4.36 \\
&10 \text{ FORMAT}(2,99999,4) \\
&20 \text{ CALL} \ 400 \\
&21 \text{ IF} \ (A \lt B) \text{ GO TO} \ 1 \\
&22 \text{ STOP} \\
&23 \text{ END} \\
&100 \text{ FORMAT}(2,99999,4) \\
&200 \text{ IF} \ (C \lt D) \text{ GO TO} \ 3 \\
&210 \text{ WRITE}(2,200) \\
&220 \text{ END} \\
&150 \text{ IF} \ (E \lt F) \text{ GO TO} \ 4 \\
&160 \text{ WRITE}(2,200) \\
&200 \text{ IF} \ (G \lt H) \text{ GO TO} \ 5 \\
&210 \text{ WRITE}(2,200) \\
&220 \text{ END} \\
&400 \text{ IF} \ (I \lt J) \text{ GO TO} \ 6 \\
&410 \text{ WRITE}(2,200) \\
&420 \text{ END} \\
&500 \text{ IF} \ (K \lt L) \text{ GO TO} \ 7 \\
&510 \text{ WRITE}(2,200) \\
&520 \text{ END} \\
\end{align*}
\]

Where the numbers in column 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 MONTE, with values for two integer parameters, have been inserted at the head of these blocks. The parameters can be used with a user-supplied MONTE routine, such as that below, to analyze the run-time path of the program.
When assembled, the modified program and subroutine produced the following results on the Oxford 1906A:

```
NO NEW IN DETERMINED VARIABLE
COMPLEX HOPES -8.23680958E05 32 0.00000000E00 32
REAL HOPES -8.23680958E05 32 1.00000000E20 32
REAL HOPES -8.23680958E05 32 1.00000000E20 32
0 1 2 1 2 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

```
-30 23 23 16 16
0.25
14
-60 30 25
30 23 23 16 16 17
11 9
30
30 26
```

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 the 'base' machine
version, i.e. the version initially incorporated and from which other
machine ranges 'diverge' if necessary. Since most contributors have de-
vveloped 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 imple-
mentations of this language and different representations are employed. A
more awkward problem is that of input and output procedures which are ex-
tensively used in the testing software. To combat the lack of standardis-
ation here, implementors have directly adapted the provided input/output
statements to suit their own machine variants or provided 'jacket' proced-
ures with the same specification. Wherever possible systematic textual
changes are performed mechanically and not by hand-editing.

At the end of this initial conversion phase which would include the adjust-
ment 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 dis-
tributed 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 avail-
able 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 con-
venience or consistency. The scale of the operation must also be con-
sidered. At Mark 4, for instance, a group starting its first implemen-
tation of the Algol 60 and FORTRAN Libraries has about 500 routines and
800 example or stringent test programs to process. Additional implemen-
tation 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.

9. 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 1966.

References