comunicaciones técnicas

Numal in FORTRAN

0. Introduction and Summary.

P. Wynn*

serie azul: monografías

INSTITUTO DE INVESTIGACIONES
EN MATEMATICAS APLICADAS
Y EN SISTEMAS

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

APARTADO POSTAL 20-726
MEXICO 20, D. F.
548-54-65

IIMAS
UNAM

comunicaciones técnicas

1981 Serie Azul: Monografías No. 48.0

Numal in FORTRAN

0. Introduction and Summary.

P. Wynn*

* Profesor Visitante

Recibida: 11 de febrero de 1981.

INSTITUTO DE INVESTIGACIONES
EN MATEMATICAS APLICADAS
Y EN SISTEMAS
UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

APARTADO POSTAL 20-726
MEXICO 20, D. F.
548-54-65

0. PREFACE

The library of numerical algorithms, Numal, written in Algol 60, contains many of the refinements of existing numerical methods and the newly developed computational techniques introduced by nembers of the Mathematical Centre at Amsterdam. For his own use during a sojourn in North America, the writer supervised a translation of Numal into a FORTRAN version suitable for use on a mini-computer, and prepared documentations supporting this version; the translation it self was carried out almost single-handedly by H.T. Lau. The director of IIMAS has graciously consented to the inclusion of the FORTRAN version of Numal in the research and development programme of his institute.

This introductory volume is followed by further volumes as follows:

1. Output subroutines; multi-length integers; double precision and complex arithmetic; merging and sorting operations.

2. Elementary operations of linear algebra; solution of linear equations; matrix inversion; evaluation of determinants.

3. Real matrix eigenvalue-vector determinations; singular value decompositions.

4. Linearly stored upper triangular parts of symmetric matrices; matrices with complex elements.

5. Auxiliary transformations of linear algebra.

6. Manipulations of polynomials and their evaluation; transformations of series.

7.- Numerical solution of initial value problems by explicit and implicit methods; exponentially fitted methods.

8.- Systems of second order differential equations; boundary value problems.

9.- Numerical integration; Fourier series.

10.- Approximation of Jacobian matrices; zero finding algorithms; minimisation; parameter estimation.

11.- Special functions.

The writer assumes responsibility for imperfections in both programmes and documentations; neither IIMAS nor he accepts responsibility for the consequences of such imperfections.

1. Introduction

General acceptance of a new programming language is opposed by economic and educational inertia: the translation of an accumulated store of computer programmes from a currently used language is, over the short term, costly; persons who belong to established groups and communicate in one language are reluctant to express themselves in another. However powerful and efficient the new language may be, these two forces of inertia are effective. The adoption of a new language may be facilitated in three ways: firstly, by amply demonstrating, with reference to practical examples, that this language is indeed more powerful than others currently used; secondly, by making available an extensive library of programmes written in the new language; thirdly, by producing a version of this library written in a currently used language in a manner which simulates, in a necessarily clumsy manner, some of the constructions of the new language, and thus offers the enterprising individual a practical introduction to its resources and power.

Although advance to the use of Algol 60 has been generally attained elsewhere, for reasons explained above FORTRAN is still widely used in North America. This language is itself in a process of evolution, and in the latest version (see, for example, Brainerd W., Fortran 77, Comm. A.C.M., 21 (1978) 806-820) a number of elementary and easily implemented features of Algol 60 have been incorporated; for example, the limits and interval in a DO statement are no longer required to be positive integers; the IF... THEN... ELSE construction is made available; remote blocks, corresponding to internal procedures in Algol 60, may now be executed. Tentative versions of FORTRAN implementing recursion are also being essayed (see, for example, Arisawa M. and Iuchi M., Debugging methods in recursive structured FORTRAN, Software practice and experience, 10 (1980) 29-43). Automatic allocation of auxiliary storage has yet to be absorbed into FORTRAN. It is, in short, conceivable that by the year 2050 FORTRAN will have evolved to a form closely resembling Algol 60.

This process of evolution may be assisted in the three ways described above. Firstly, it is the purpose of the following passages to demonstrate the relative superiority, with regard to requirements imposed by practical use, of Algol 60. Secondly, an extensive collection of procedures is already available in the form of NUMAL, the numerical algorithm library constructed at the Mathematical Centre at Amsterdam, which makes fuller and more enlightened use of Algol 60's resources than that evidenced by other such collections, and incorporates methods, particularly those concerning the numerical solution of ordinary and partial differential equations, in advance of those available elsewhere (for a systematic treatment of these methods, see van der Houwen P.J., Construction of integration formulas for initial value problems, North Holland (1977)). Thirdly, the volumes which follow this introduction contain a FORTRAN version of NUMAL which the writer, unable to lay his hands upon an efficient Algol 60 compiler, has prepared for his own use during a sojourn in North America, and which, in addition to the necessary documentation, contains subroutines directly implementing many of Algol 60's constructions.

2. Algol 60 and FORTRAN

Practical use imposes certain requirements upon a programming language; the following headings are ventured as a basis for later discussion:

a) perspicuity: the language should afford a perspicuous presentation of the numerical process being implemented.

b) succintness: the constructions of which a language is capable should permit the description of numerical processes in concise form; in particular, each construction should be capable of use in many contexts.

c) facility: it should be possible to implement the strategies directing the course of an extensive computation with ease; in particular, it should be easy to manipulate data sets in a manner incurring minimum waste of computer storage.

d) coherence: it should be possible to combine operations between blocks of statements (procedures or subroutines) without difficulty

e) efficiency: it should be possible to implement computationally efficient processes in a correspondingly efficient manner

f) power: the power of the language should conduce to the rapid development of powerful computational technology

g) efficacy: it should be possible to write a programme in such a way that its competently compiled object programme functions almost as effectively as the corresponding programme written directly in machine code.

It is clear that the above classifications overlap; nevertheless they may serve as a basis for assessing the merit of an algorithmic language. Algol 60 and FORTRAN will be contrasted, with reference to practical examples, in the light of these classifications.

2.1. Perspicuity

New computational methods are being and will be developed continually. The first requirement that any user who wishes to understand what he is doing will impose upon a programme is that he should easily see how it works, and how well it works. Algol 60 satisfies this requirement in large measure: there is a strong correspondence between mathematical notation and the statements of the language; the block structure of the language naturally expresses successive stages of a mathematical argument; the language is relatively free from inelegant constraints which both impede understanding and obstruct the implementation of a mathematical process upon a computer. The absence of apparent structure of a FORTRAN programme renders the latter difficult to follow in all but the simplest cases (this remark holds true for numerous other languages; for example APL); the versions of FORTRAN in current use are replete with petty restrictions (for example, the limits and interval in a DO statement must be positive integers; the lower limit of array reference integers is always 1, and so on; again other languages are deficient in this respect; for example, the version of PASCAL now in current use requires that the interval in a *for* statement must be ±1, a one-dimensional array whose name features in the parameter list of a procedure must have declared length agreeing with the actual array whose name replaces that of the declared array upon call, and so on).

2.2. Succintness

One important consequence of constructional power in any language is that meanings may be expressed in concise form. Individual programmes written in a constructionally powerful programming language may be condensed and stored compactly; groups of such programmes may, by exploiting the power of the language, be made to combine operations in a way which reduces to a minimum wastage of computer storage.

In order to compare Algol 60 and FORTRAN with respect to economy in the use of computer storage, first consider the following Algol 60 procedure:

```
real procedure sum (index, low, interval, up, term);
integer index, low, interval, up; real term;
begin real locsum; locsum:=0;
   for index:=low step interval until up do locsum:=locsum+term;sum:=locsum
end;
```

When used, *interval*, *up* and *term* in the above procedure may be real expressions depending upon *index*. Subject to suitable prior declarations, *sum* may be used in the following ways:

```
a)                        simple sum:=sum(k,0,1,10,exp(-k))
b)                  for i:=1 step 1 until n do
                    row sum[i]:=sum(j,1,1,n,a[i,j])
c)                  for j:=1 step 1 until n do
                    col sum[j]:=sum(i,1,1,n,a[i,j])
d)                  trace:=sum(k,1,1,n,a[k,k])
e)                  power sum:=sum(k,1,k,16,k)
```

f) *another sum*:=*sum*(k,1,1,10-k,k*k)

g) *double sum*:=*sum*(i,1,1,n,*sum*(j,1,1,i,a[i,j]))

Call a) computes $\sum_{k=0}^{10} e^{-k}$; calls b-d) compute the row sums, column sums, and sum of diagonal elements of a matrix; call e) computes $\sum_{\ell=0}^{4} 2^{\ell}$; call f) computes $\sum_{k=1}^{5} k^2$; call g) computes the double sum $\sum_{i=1}^{n} \sum_{j=1}^{i} a_{i,j}$.

Naturally, individual FORTRAN programmes can be written to carry out both the computations of the above examples and any other function which the above procedure may be made to fulfil. However, in the above example, many individual FORTRAN programmes must be constructed to implement all possible functions of one short Algol 60 procedure. Numerous examples, more meaningful but less readily understood than the above, might be given. As these examples suggest, an extensive library of Algol 60 procedures can be made to depend upon a small number of procedures each capable of use in many different ways; the programmes of a corresponding FORTRAN library require the use of significantly larger machine storage.

2.3. Facility

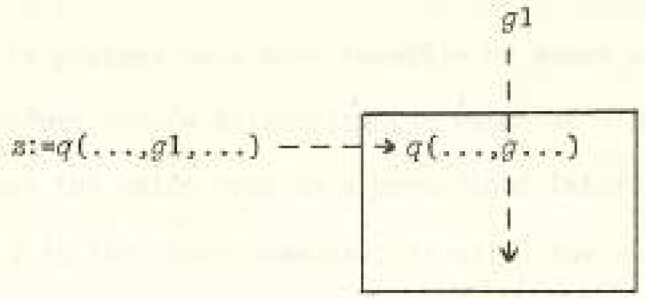
Any syntactically restricted language imposes burdens upon the person wishing to use it as a medium of expression. This is true of programming languages, and is true of FORTRAN with respect to the allocation of auxiliary storage.

During a stage in an extensive computation it may be necessary to store certain numbers (the amount of required storage possibly depending upon the results of previous calculations) which are no longer required when the stage has been completed. When using Algol 60 it is permissible to enclose the instructions implementing the computations of the stage in a block and, in the block heading, to declare the extent of the storage required during the stage. At the conclusion of the stage the block is left, the storage declared as above now becomes free for alternative use (if so desired, during the execution of further stages). The above stage may contain an intermediate stage, itself requiring auxiliary storage; in such a case the above block contains an interior block, equipped with the necessary storage declarations which reserve storage additional to that reserved at the heading to the outer block. This nesting process may be continued. The above blocks and subblocks may be declared procedures. The way in which dynamic storage allocation takes place naturally depends upon the order in which the procedures are called. The storage allocation strategy varies from programme to programme, but in every case it is implemented automatically by the Algol compiler. In this way, machine storage is used in the most economical way possible; it is reserved when needed, and made free when not required.

It is to a certain extent possible to simulate the dynamic storage allocation described above when using FORTRAN. The blocks and sub-blocks are written as subroutines which contain in their parameter lists one-dimensional real array names such as (for example) WORK1, WORK2,... . The user calculates in advance the amount of auxiliary storage needed by each subroutine when used in the solution of his problem, declares in the heading to the main programme a global one-dimensional real array, WORK say, and replaces WORK1, WORK2,... upon call by suitably displaced references to WORK. In other words, each time the user writes a programme, he effectively implements the automatic storage allocation facilities of an Algol 60 compiler by himself (and must do so without error). If he is not prepared to do this, he must reserve auxiliary storage statically (e.g. declare WORK1, WORK2,... to be independent arrays of maximal size) and leave considerable portions of machine storage idle during most of the programme execution time.

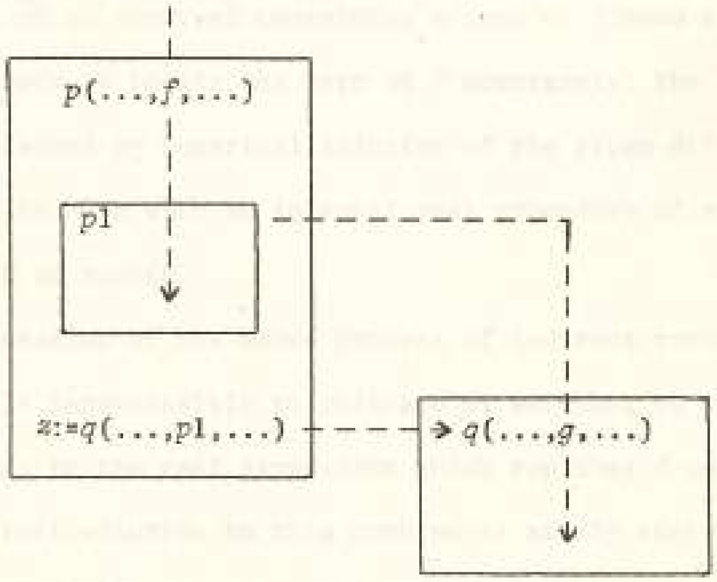
2.4. Coherence

That Algol 60 procedures may be assembled in a compact library may be demonstrated by considering the following example: let a real procedure *q* have in its parameter list a real expression *g* which the user must supply to suit his particular needs when using *q*. The user may make use of the allocation *s*:=*q*(...,*g*1,...), for example, *g*1 being the real expression in which he is interested (see Diag. I).



Diag. I

Let a further procedure *p* also have in its parameter list a real expression *f* which the user must supply; further, let *p* make use of an internal real procedure *p*1 which itself makes use of *f*; lastly, let *p* call *q*, with *p*1 replacing the *g* of *q* upon call (i.e. suppose that the body of *p* contains a statement of the form *z*:=*q*(...,*p*1,...); see Diag. II).



Diag. II

Two features may be observed in this case: that of limited indirect recursion (*p* calls *q* which calls part of *p* which does not call *q*), with the further complication that *q* (which is external to *p*) makes use of a real expression supplied upon call in the parameter list of *p*.

The above discussion is perhaps made more tangible by means of an example. NUMAL contains a real procedure *zeroin* delivering the value of x for which a real valued function $g(x)$ assumes the value zero in a prescribed interval (*zeroin* corresponds to q and $g(x)$ to g in the above example); it will, for example, locate a zero of cos(x) in the range [0,2](cos(x) corresponds to $g1$ in the above). NUMAL also contains a procedure *rk5a* for the numerical solution of a differential equation $\frac{dy}{dx}=a(x,y)$ ($y(x_0)=y_0$ prescribed) until a zero of a real expression f (depending upon x and y) is encountered; thus, if the user wishes to determine the point at which the graph of $y(x)$ intersects that of $1+x$, f may be taken to be $y-1-x$ (*rk5a* corresponds to p above, and f to f). *rk5a* integrates the given differential equation, inspecting f at every step, until a change in sign of f is experienced over a step. The endpoints of an interval containing a zero of f have now been located. *rk5a* now calls upon *zeroin* to locate the zero of f accurately; the value of y, upon which f depends, is obtained by numerical solution of the given differential equation; *zeroin* is called by *rk5a* with an internal real procedure of *rk5a* in place of g in the parameter list of *zeroin*.

A FORTRAN implementation of the above process of indirect recursion is difficult to contrive. It is impermissible to relocate p1 external to p since, to begin with, p1 requires access to the real expression which replaces f upon call of p by the user. The most direct solution to this problem is simply that of stripping the parameter list from q, and placing an entire copy of the text of q within p. A FORTRAN programme library implementing numerous advanced mathematical techniques in this way is necessarily vitiated by massive duplication of programme text.

The coherence of Algol 60 has been fully exploited in NUMAL. The major procedures of NUMAL are assembled in compact modular form with minimal duplication of text between auxiliary procedures. Since the standard operation sequences implemented by the auxiliary procedures are of frequent occurrence, the development of new computational techniques is greatly facilitated.

2.5. Efficiency

Power in a language conduces to efficiency. That this is so in the case of Algol 60 may be seen by considering the procedure *quadrat* of NUMAL, which may be used to evaluate $I(f;a,b)=\int_a^b f(x)\mathrm{d}x$ $(-\infty<a,b<\infty)$ where f is real valued. *Quadrat* makes use of an internal real procedure *quadsum*, which functions recursively. After some initial administration, two quadrature sums involving values of f over $[a,b]$ are evaluated and compared; if they agree to within a specified tolerance, one of them is accepted as a suitable approximation to $I(f;a,b)$. If they do not agree, the interval $[a,b]$ is split into two parts $[a,\frac{1}{2}(a+b)]$ and $[\frac{1}{2}(a+b),b]$; the integrals over the two subintervals are treated in the same way. After an allocation of the form *mean*:= $0.5*(a+b)$, *quadsum* contains a statement of the form

sum:=*quadsum*(*a*,*mean*,...)+*quadsum*(*mean*,*b*,...).

The function values used in the evaluation of the two quadrature sums over $[a,b]$ are used, with others, in the evaluation of the sums over the two subranges (they replace the dots in the above statement).

When applied, with $a=0$, $b=1$, to a function which is smooth over $[0,\frac{1}{4}]$ and $[\frac{1}{2},1]$, but irregular over $[\frac{1}{4},\frac{1}{2}]$, *quadrat* might function as follows: the first two quadrature sums over $[0,1]$ do not agree, since f misbehaves over $[\frac{1}{4},\frac{1}{2}]$; after the first split, the two sums evaluated over $[\frac{1}{2},1]$ do agree, since f is smooth over this interval, and integration over this interval is terminated; they do not agree over $[0,\frac{1}{2}]$, since this interval contains $[\frac{1}{4},\frac{1}{2}]$; after the second split, the integral over $[0,\frac{1}{4}]$ is disposed of; further subdivisions over $[\frac{1}{4},\frac{1}{2}]$ take place, until all subintervals have been dealt with. For other functions behaving in another way, the splitting patterns are different: *quadrat* uses a fine interval where required, and not otherwise. This efficient method of adaptive quadrature is easy to programme in a recursive language such as Algol 60, but difficult to implement when using a non-recursive language such as FORTRAN.

2.6. Power

It is to be expected that power in a programming language facilitates the generation of powerful computational technology. That this is so may be illustrated by reference to the Algol 60 procedure *quadrat* above. This is called in the form

$$integral := quadrat(a,b,x,f,\ldots)$$

where, in particular, f is a real expression depending upon x (just as *term* depends upon *index* in the example of §2.2). During operation, *quadrat* allocates to x the successive argument values in $[a,b]$ for which a function value $f(x)$ is required. The double integral

$$\int_a^b \int_{l(x)}^{u(x)} g(x,y)\,dy\,dx$$

where l, u and g are real valued functions, may immediately be evaluated by a call of the form

$$double\ int := quadrat(a,b,x,quadrat(l(x),u(x),y,g(x,y),\ldots)\ldots).$$

Integrals of higher order may be evaluated in the same way; *quadrat* is then carrying out adaptive integration in many dimensions. This efficient and powerful usage is accomplished very simply by means of a single assignment statement. It is considerably more difficult to achieve the same result employing FORTRAN. Again, many examples of a similar nature could be given.

2.7. Efficacy

Algol 60 is an economically designed language; it is thus possible to construct Algol 60 compilers for use with small computers (it is remarked that PL1 is not economically designed; as far as the writer is aware, no mini-computer full PL1 compiler exists). FORTRAN is a programming language with limited capabilities, and for this reason FORTRAN compilers have also been constructed for use with minicomputers.

The major procedures of an Algol 60 library may be assembled in compact modular form from auxiliary procedures; furthermore, optimal use of available machine storage is made during execution. Thus an extensive Algol 60 library may be committed to a minicomputer backing store; all but excessively large numerical problems can be solved by use of a minicomputer in this way. A corresponding FORTRAN library requires far more extensive backing store (indeed, so much as to render the use of the library in this way infeasible) and its subroutines occupy far more high speed storage during execution; the limitations of FORTRAN virtually necessitate waste of data storage space; in short, the range of problems which can be handled by a small computer when FORTRAN is used is restricted.

Execution times for programmes assembled from an Algol 60 library may, by exploiting the resources of the language, be made far shorter than those of their FORTRAN equivalents. That this is so can perhaps be most easily explained with reference to NUMAL.

Many processes in numerical analysis involve strategic decisions that are taken relatively infrequently, and subsequent executions of elementary operations such as the assignment of initial values to vector and matrix slices, duplication and interchange of such slices, rotations and reflections, etc., the latter operations consuming most of programme execution time. NUMAL contains a set of two dozen or so procedures implementing the above elementary operations (the procedures are taken from the books by T.J. Dekker and W. Hoffmann: Algol 60 procedures in numerical algebra, vols. 1 and 2, Mathematical Centre (1968)). The major procedures of NUMAL are thus easy to read: a call such as *inivec*(v,2,10,x) is easily understood to mean that the elements with suffices 2,...,10 of the vector v are being given the initial value x, for example; it is slightly harder to deduce this fact from an equivalent <u>for</u> statement. The major procedures of NUMAL are more concise than those of a library not containing such a set of elementary procedures: numerous for statements carrying out initialisation of vectors are replaced by subroutine calls exploiting the single for statement belonging to *inivec*; similarly for the further elementary operations. Use of the above set has a third, and more important consequence. The elementary procedures in question are all quite short; prepared versions may be written, making optimal use of machine capabilities, in the internal language of the machine being used. The user of NUMAL is thus offered the possibility of both working with a language which has the power and elegance of Algol 60, and using programmes which function for most of the time (the proportion depending upon the nature of the programme in question) directly and with optimal performance in internal machine language.

The above artifice is not available to the user of FORTRAN. Due to the lack of coherence between blocks of statements, the above elementary processes must be described in extenso within major subroutine bodies; execution time of the compiled programme is hence greater than that of its Algol 60 counterpart.

In the above remarks concerning the merits of Algol 60, competent compilation is assumed. Certainly many competently constructed Algol 60 compilers exist outside North America.

It has also been assumed that the Algol 60 library is assembled with maximum utilisation of language resources; this condition is not, at present, always satisfied. Although games such as chess are easy to devise, the full implications of the rules of chess itself are still the subject of research; it is evident that some chess players have less clear understanding of these implications than others. Similarly, languages such as Algol 60 are easily invented, but it is clear that the resources of Algol 60 itself have not yet been fully understood; some collections of Algol 60 procedures have been assembled with less clear understanding of these resources than that sustaining the construction of others. For example, in the collection of linear algebra subroutines (Linear Algebra, Springer (1971)) edited by Wilkinson and Reinsch, the elementary operations referred to above are written out, repeatedly and extensively, in the texts of the subroutines of this collection (the sole use of machine dependent standard operations concerns the accumulation of double precision inner products) and the resulting translated machine programmes are in consequence less efficient than those produced by the NUMAL system.

3. The FORTRAN version of NUMAL

3.1. Modifications to NUMAL

The documentations and subroutines which follow this introduction depart in many respects from being a direct translation of the NUMAL procedures.

A number of straightforward semantic errors (see e.g. *lngmatvec* and *lngtamvec* of §1.5.2 (1977)) have been corrected. Certain of the numerical results given in the documentation to NUMAL purporting to be produced by test examples are not and should not be produced by use of the procedure in question (see e.g. *bessiaplusn* of §6.10.2 (1979)); the documentation has been improved at such points. It has also been improved where misleading. For example the subroutine *start*, which determines the starting point of a backward recursion for the stable evaluation of a sequence of Bessel functions $J_{a+i}(x)$ or $I_{a+i}(x)$ with consecutive values of i, is not required (as is suggested in on page 10 of §6.10.1 (1979)) for the evaluation of values of $K_{a+i}(x)$. In places where lines of programme can be omitted without loss (see *tfmsymtri1* of §3.2.1.2.1.1 (1974)) this has been done. Again, where variables are declared but never used (see e.g. *tricub* §4.2.2 (1975) where some real variables remain as detritus from a previous version of the procedure) they are omitted from the translated versions.

The Algol 60 compiler for the CDC Cyber 70 upon which the NUMAL procedures have been tested, very kindly allocates the value zero to all real and integer variables immediately following declaration. Thus (allowing for format modification) the programme

begin real *x*; print (*x*) end

prints the value 0.0. Occasional unwitting use of this facility has been made in NUMAL (see e.g. *ark* §5.2.1.1.1.1.H (1979)); the translated versions may be implemented by unkind compilers. (Thus if comparison between two versions of a procedure reveals the presence of a number of apparently pointless allocations to zero at the beginning of the FORTRAN version, the above provides an explanation.)

For the CDC Cyber 70, integers lie roughly in the range $\pm 10^{18}$ and real numbers, with precision 10^{-14}, lie roughly in the range $10^{\pm 250}$. Ranges for other computers are not so generous; for example, the PDP 11/45 minicomputer integer range is roughly ± 32000, and REAL*8 real numbers, with precision 10^{-17} lie roughly in the range $10^{\pm 37}$. Certain NUMAL Algol 60 procedures appear to benefit explicitly from the generous ranges of CDC Cyber 70 numbers. For example, a value which can be as small as the real number precision must be allocated to location 0 of a real array *em* occurring in the parameter list of the procedure *equilbroom* (§3.2.1.1.2 (1974)) prior to call. *equilbroom* contains the statement $eps := em[0]\uparrow 4$; it is important in the subsequent course of the computation that the value of *eps* should be nonzero. For the CDC Cyber 70 the fourth power of the real number precision lies within the range of real numbers having a nonzero representation; for other computers this is not so (e.g. $(10^{-17})^4 < 10^{-37}$ with regard to the PDP 11/45) and in such a case *eps* above acquires the value zero and *equilbroom* fails. The procedure *equilbroom* and others have accordingly been modified in translation, simply to ensure that they function when implemented by computers with restricted range.

Similar modifications have been carried out to increase the range of numbers that can be dealt with. For example, the quotient of two real numbers, both near the upper limit of the real number representation, can be formed (the result being approximately unity). The quotient $x+iy=(a+ib)/(c+id)$ of two complex numbers can be formed by use of the following allocations:

$$f=1.0/(c*c+d*d);x=f*(a*c+b*d);y:=f*(b*c-a*d).$$

These allocations cannot be used when $a,\ldots,d$ are all near the limit of the real number representation: the formation of the denominator of f results in overflow. However, such overflow can be avoided by use of the numbers c/r and d/r, where $r=\max(|c|,|d|)$. Such use is made in the procedure *comdiv* (§1.3.2 (1975)). However, similar considerations apply in to the use of elementary reflectors $\underset{\sim}{I}-2\underset{\sim}{u}^T\underset{\sim}{u}/(\underset{\sim}{u}\underset{\sim}{u}^T)$ (for a short account and references to the early history of such matrices, see Ch. 8, §2 of: Turnbull H.W. and Aitken A.C., An introduction to the theory of canonical matrices, Blackie (1932); Dover reprint of the 3rd edition (1961)) and the NUMAL procedures implementing use of these elementary reflectors have not been modified as above. In the translated version, they have.

A number of NUMAL procedures make use of certain special procedures whose functions are explicitly based upon the characteristics of CDC Cyber 70 arithmetic, which differ from those of other computers. For example, it is possible that for certain numerical values allocated to the real variable x during the course of a computation, the value of the boolean expression (x=0.0) is false, while that of (1.0*x=0.0) is true. Precautions against this sort of numerical behaviour were explicitly incorporated in an early version of *zeroin* (Bus J.C.P. and Dekker T.J., Two efficient algorithms for finding a zero of a function, Report NW 13/75, Mathematical Centre (1975)). Floating point numbers have a full representation of the form $\pm 2^e m$ where e is an integer exponent and $\frac{1}{2}\leq m<1$; for the smallest positive real number having such a representation, e has its maximum negative value and $m=\frac{1}{2}$; the value of this number is delivered by the NUMAL real procedure *dwarf*. Computations can produce nonzero numbers in a twilight region for which e has its maximum negative value but $m<\frac{1}{2}$. Evidently special steps may have to be taken when such numbers are produced; NUMAL contains a number of subroutines in which, for example, the value of an inner product of two vectors is compared with that of *dwarf*, special steps being taken if the product is small. Again, NUMAL contains a boolean procedure *overflow* which is accorded the value true when, upon call, its real parameter is given an overflow value. Since many computers and many compilers do not function in the above way, machine arithmetic oriented procedures have been expunged from the translated versions; the affected procedures have been modified.

Certain procedures have been slightly modified in translation (for example, the illustration of the use of *eft* (§5.2.1.1.1.3.B (1974)) in the NUMAL version appears to make great use of the known analytic solution of the differential equations solved numerically by that procedure; modifications to the latter assisting the user in cases in which the analytic solution is unknown have been introduced.

Some procedures which will doubtless be added to NUMAL (for example, RSTABP and ISTABP, delivering coefficients of stability polynomials, and EFFORK, implementing an exponentially fitted fourth order Runge-Kutta method) have been added to the translated versions. An addendum contains a number of format subroutines designed to combat the FORTRAN format conventions.

Although many of the translated procedures (those dealing with linear algebra, for example) may to a great extent be used without knowledge of the underlying numerical analysis, the writer is of the opinion that some, in particular those implementing advanced techniques for the numerical solution of ordinary and partial differential equations, cannot be so used. Accordingly, very brief accounts of the underlying methods, touching upon the significance of the required input parameter values, have been added. The documentations provided with the FORTRAN versions have been adapted from sundry internal reports and colloquium publications of the Mathematical Centre. In common with all great and enduring literature, the latter contain passages which mean precisely that which the reader would wish them to mean; it may well be that the writer's reformulation does not conform to the original intention. The documentations are in all cases provided with a subroutine tree, indicating the way in which other required subroutines are called; they are also provided with a storage map (produced automatically by the PDP 11/45 minicomputer upon which the FORTRAN versions have been tested) of the associated variables. Numerous elementary test programmes, serving no other purpose than to indicate to the user how the subroutines are to be used, are juxtaposed to individual subroutine documentation (they are prefixed by T in the documentation headings; thus TLINT, etc.).

The arrangement of the NUMAL library reflects the interest of its authors in numerical analysis; thus necessary auxiliary procedures are placed adjacent to principal procedures carrying out a major purpose (for example, a procedure for solving a set of linear equations, in which a general user might well be interested, is immediately followed by one for a required matrix decomposition, in which he might not). The translated versions have been reordered (the subjects in the new order being integers, real numbers, complex numbers, algebra,...) and auxiliary subroutines have been assembled in sections which the user not primarily interested in numerical analysis may choose to overlook.

The set of programme documentations is prefaced by a comprehensive list of contents giving one or two-line summaries of the subroutine functions; the complete scope of NUMAL may be roughly assessed by a glance through this list.

3.2. The translation of the NUMAL Algol 60 procedures into FORTRAN

The petty restrictions (see §2.1) with which FORTRAN is afflicted cannot directly be overcome; for this reason the FORTRAN versions are far longer than their Algol 60 originals.

An attempt has been made to preserve similarity of appearance between the original and translated versions of the NUMAL procedures. Thus a construction such as

```
if ϕ then
   begin
    ...
   end else
   begin
    ...
   end;
   etc
```

where ϕ is a boolean expression, appears as

```
   IF (.NOT.ϕ) GOTO n1
      ...
      GOTO n2
n1    ...
n2 ETC
```

Again, the translated versions of internal procedures set at the beginning of an Algol 60 procedure also occur at the commencement of the equivalent subroutine, use being made of appropriate GOTO statements; thus, a construction such as

```
procedure proc(...)
...
begin...
   procedure intproc1(...)
   begin
   ...
   end;
   ...
   intproc1(...)
   ...
end
```

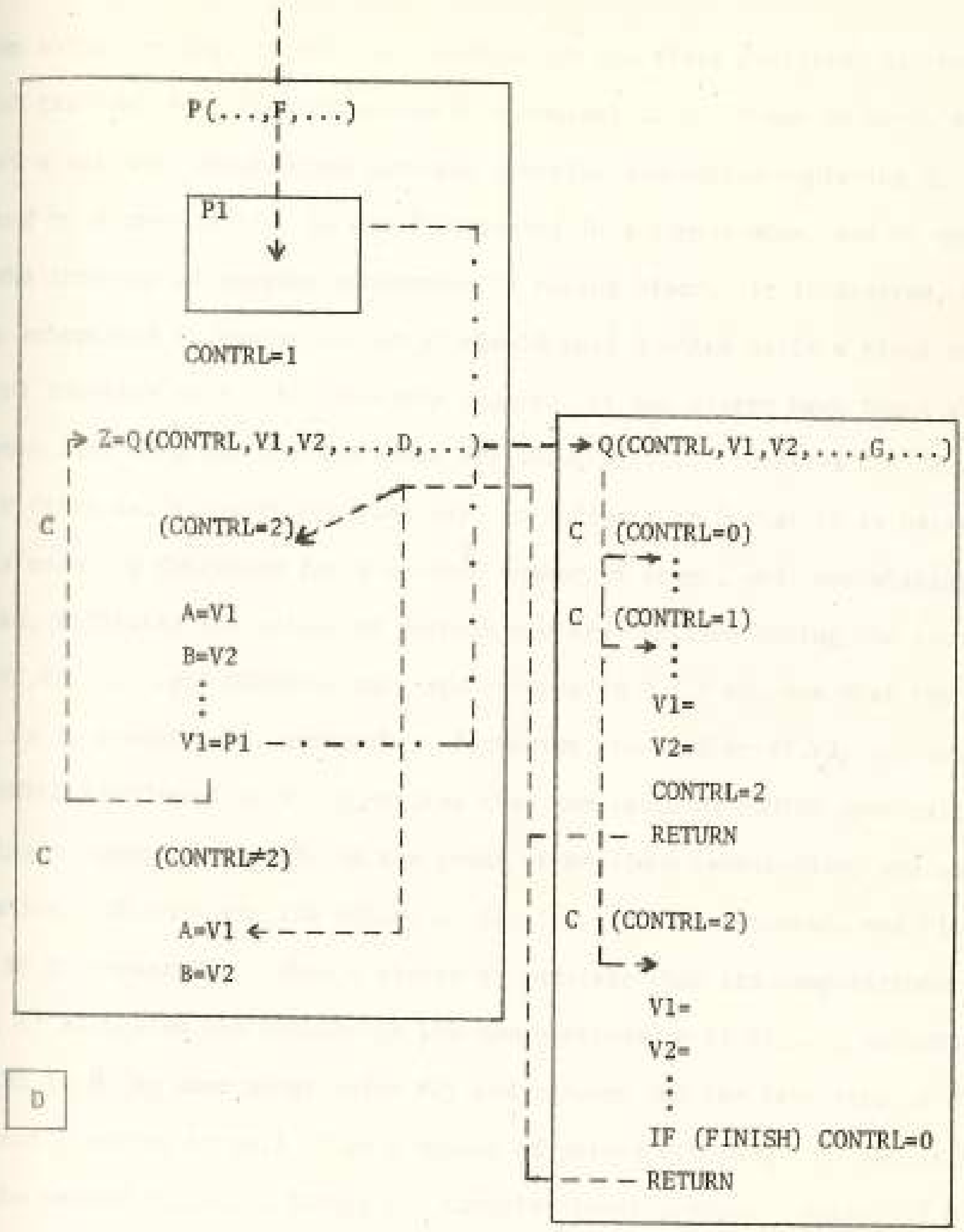
appears as

```
      SUBROUTINE PROC(...)
C     DATA DECLARATIONS
      GOTO n1
C     INTPROC1
m1    ...
      GOTO (n2,...) JUMP1
n1    ...
C     PREPARE DATA FOR INTPROC1
      JUMP1=1
      GOTO m1
n2    ...
```

Sequences of internal procedures declared at the beginning of a procedure are dealt with in the same way.

Allocation of auxiliary storage is left to the user of the translated programmes. The dimensions of auxiliary storage arrays are fixed at some convenient magnitude by a declaration at the beginning of a subroutine (AUX(10,15), for example); if an auxiliary storage array features in a subroutine call within a subroutine body, so that the row dimension of the array must also occur in the parameter list of the called subroutine at call, a dimension allocation statement, corresponding to the conveniently chosen magnitudes, occurs at the beginning of the subroutine (thus: AUXDIM=10, for example; the internal call has the form CALL SUB(...,AUX,..., AUXDIM)). If the user requires more extensive auxiliary storage, he must change both the declaration and the allocation statement accordingly (AUX(10,15) and AUXDIM=10 become AUX(20,25) and AUXDIM=20, for example).

It proved necessary to overcome the lack of facilities for communication between blocks (see §3.3) by application of an artifice (due to N. Rafla). The use of this artifice in connection with the example of §3.3 is illustrated in Diag. III.


P(...,F,...)
P1
CONTRL=1
Z=Q(CONTRL,V1,V2,...,D,...)
C (CONTRL=2)
A=V1
B=V2
V1=P1
C (CONTRL≠2)
A=V1
B=V2
D
Q(CONTRL,V1,V2,...,G,...)
C (CONTRL=0)
C (CONTRL=1)
V1=
V2=
CONTRL=2
RETURN
C (CONTRL=2)
V1=
V2=
IF (FINISH) CONTRL=0
RETURN

Diag. III

An extra integer, CONTRL, is inserted (in the first position) of the parameter list of the real function subroutine Q equivalent to *q*. Q may be used, with the name of a suitably constructed external function subroutine replacing G, and CONTRL replaced by 0 upon call; Q is now functioning in a simple mode, and no communication with the interior of another subroutine is taking place. It is desired, however, that a subroutine P (equivalent to *p*) should call Q which calls a block of instructions P1 interior to P. In the cases treated, it has always been found that the parameter list of Q has certain other variables V1,V2,... besides G. In a non-simple mode, P first calls Q with CONTRL=1 say, to indicate to Q that it is being called in this mode. Q functions for a certain number of steps, and, now wishing P1 to be executed, allocates the values of certain numbers required during the execution of P1 to V1,V2,..., sets CONTRL=2 say, and returns to P. P notices that the value of CONTRL is 2, executes P1, using the information provided by V1,V2,..., and calls Q with CONTRL left equal to 2. Q notices that the value of CONTRL upon call is 2, and accordingly jumps immediately to the point of previous termination, and continues computation. In this way the values of V1,V2,... may be adjusted, and P1 may be called by Q, repeatedly. When Q wishes to indicate that its computations are completed, it allocates the results of its computations to V1,V2,..., adjusts the value of CONTRL to 0 (or some other value $\neq 2$) and returns for the last time to P. It may occur that Q wishes to call P1 at a number of points within Q. CONTRL is accordingly given the values 2,3,4,... during the conversational process. Again, if Q wishes to call internal blocks P2,P3,... of P, CONTRL may be used for this purpose. In the non-simple mode of operation, G is unused by Q; P must insert the name of a dummy external function subroutine (D, in Diag. III) in place of G upon call of Q. CONTRL, in the above, functions as a multiple input and exit device, and as a device for routing computational paths in both merged subroutines.

Extensive use of the above device is made in the translation of the NUMAL collection. The FORTRAN user who uses the above subroutine Q in the simple mode (i.e. what is for him a normal way) must pay the price of calling Q in the form Q(0,...); he may, at his own speed, experiment with recursive use of subroutines.

The limitations upon communication in FORTRAN have also had a further effect upon the structure of the translated procedures. It may happen, at a certain stage in the implementation of a numerical process, that numbers threaten to go out of range or that some other computational misfortune occurs. Suppose that the user constructed procedure implementing this stage is *p*1, and that it has been called by the library procedure *p*2,..., which has been called by *pn*, which has been called by the main programme. In the use of Algol 60 it is permissible for *p*1 to break off operations (as would be desired in the above case) and jump directly to an emergency exit in the main programme; control now passes to this programme and appropriate remedial action can be taken. Such a direct jump is impermissible in the use of FORTRAN. For this reason, many subroutines are declared as logical function subroutines. In the above example P1 (equivalent to *p*1) may set a failure indicator (by use of a common declaration) in the main programme, set its own value to *false* and return control to P2, which notices that P1 has the value *false* and sets its own value to *false*, and so on; the main programme is reached and the failure indicator can be inspected. It may occur that failure can occur for many reasons; to assist in such cases, the subroutine concerned is declared to be an integer subroutine (given the value 0 upon return from a successful call, 1,2,... otherwise).

Fully recursive processes, such as that implemented by *quadrat* as described above, have been re-programmed by the writer ab initio; the Algol 60 and FORTRAN versions bear little resemblance.

3.3. Machine dependent features

Some of the modifications to NUMAL described in the preceding sections result indirectly from the effect of machine and compiler characteristics. Further features of the translated versions result directly from considerations of machine hardware and compiler construction.

In the FORTRAN subroutines themselves, all real variables are declared to be of REAL*8 length; since many FORTRAN compilers are not provided with complex declarations yielding real and imaginary parts of REAL*8 length, the real and imaginary parts of complex numbers are declared separately (to be of REAL*8 length) and arithmetic operations upon complex numbers are programmed. The programmes are transportable, in the sense that those subroutines which require knowledge of machine precision, maximum single length integer representation, etc., have these values in their parameter lists. Certain subroutines of linear algebra (for example, those concerned with the iterative refinement of a numerical solution to a set of linear equations, and the iterative refinement of eigenvector determinations) require operations upon double precision numbers; such numbers have representations of the form $a=2^e h+2^{e-d}t$ where e is an integer exponent, d is the number of binary digits used to represent the mantissa of a single precision (e.g. REAL*8) mantissa, and h and t are single precision mantissae. The subroutines in question require, for example, that a double precision sum be formed from two single precision constituents (e.g. $a=b+c,a$ in double precision as above, b and c in single precision). Correct execution of such an assignment is machine dependent, and the elementary subroutines (DPADD,...) below should be programmed independently in the internal language of the machine in use. At the moment, to test operation of further subroutines, the artifice of treating a as above in single precision has been adopted (so that, in the above, $a=2^e h$ and $t=0$). The dependent subroutines for the time being produce no useful result.

Since they are machine dependent, it has also been left to the user to construct transfer programmes which, upon receipt of information that a major subroutine is to be used, automatically transfer both it and required auxiliary subroutines from backing storage into high speed storage (with cancellation of duplicated transfer if two or more major subroutines use the same auxiliary subroutines).

3.4. FORTRAN compiler dependent features

The translated versions of the NUMAL procedures have been written in such a way that they may be implemented by a number of FORTRAN compilers. No advantage has been taken of the characteristics of particular compilers, which appear to differ considerably. For example, the way in which parameters are passed seems to be the subject of debate among constructors of FORTRAN compilers. When run upon the PDP 11/45 mini-computer, the following programme prints 2,2 and 2,2.

```
      SUBROUTINE SUB (I,J)
      INTEGER I,J(2)
         J(1)=J(1)+I
         J(2)=J(2)+I
         WRITE (6,1)J(1),J(2)
1           FORMAT (I1)
         RETURN
      END
      INTEGER K(2)
         K(1)=1
         K(2)=1
         CALL SUB (K(2),K)
         WRITE (6,2)K(1),K(2)
2           FORMAT (I1)
      END
```

Both inside the subroutine body, and outside, 1 and 1 make 2, as perhaps they should. When run upon the WATFIV compiler, the programme prints 2,2 and 2,1: 1 and 1 no longer always make 2 outside the subroutine body. Many examples of a similar nature could be given. Precautions against the production of anomalous results, as in the above example, have been taken; this means that when run on a particular compiler, unnecessary precautions have been taken.

Contents

2.4 Auxiliary transformations

2.4.1 General real matrices

2.4.2 Linearly stored upper triangular parts of symmetric matrices

2.4.3 Complex matrices

3 Algebraic computations

3.1 Manipulations upon polynomials

3.2 Evaluations of polynomials

3.3 Transformation of series

3.4 Continued fractions

4 Numerical solution of differential equations

4.1 Initial value problems

4.1.1 Single first order equations

4.1.2 Single second order equations

4.1.3 Systems of first order equations

4.1.3.1 Direct methods

4.1.3.2 Implicit methods requiring the use of the Jacobian matrix associated with the system of differential equations (e.g. for the equation $\frac{d\underset{\sim}{y}(x)}{dx} = f(x,\underset{\sim}{y})$ ($\underset{\sim}{y},\underset{\sim}{f} \in R^n$) of the matrix with elements $\partial f_i/\partial y_j$ $(i,j=1,\ldots,n)$)

4.1.3.3 Methods making use of a stability polynomial (requiring a bound upon the spectral radius with respect to the eigenvalues in the closed left half-plane of the Jacobian matrix associated with the system of differential equations, assuming these eigenvalues to be (approximately) real alone (systems of this form arise from certain discretized parabolic partial differential equations) or (approximately) pure imaginary alone (systems of this form arise from certain discretized hyperbolic partial differential equations))