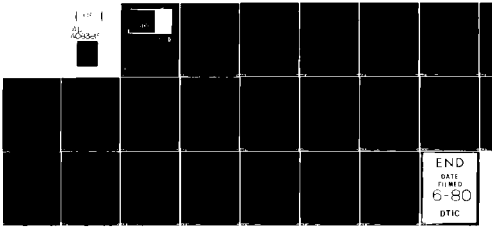
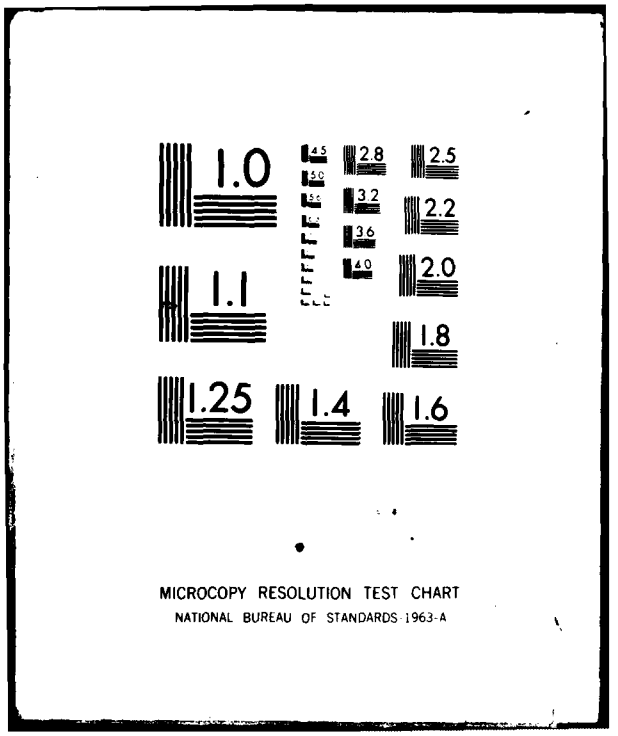
AD-A083 815 WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
PROGRAMMING LANGUAGE FACILITIES FOR NUMERICAL COMPUTATION.(U)
JAN 80 J R RICE DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-2033 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

MRC Technical Summary Report #2033

PROGRAMMING LANGUAGE FACILITIES FOR NUMERICAL COMPUTATION

John R. Rice

LEVEL II

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

January 1980

(Received November 30, 1979)

Approved for public release
Distribution unlimited

Sponsored by
U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

(See 1473)

UNIVERSITY OF WISCONSIN-MADISON
MATHEMATICS RESEARCH CENTER

PROGRAMMING LANGUAGE FACILITIES FOR NUMERICAL COMPUTATION

John R. Rice

Technical Summary Report #2033
January 1980

ABSTRACT

This is a working paper for IFIP Working Group 2.5 (Numerical Softwar to stimulate discussion of a possible group project. The project is to describe the facilities in a programming language that make it useful for numerical computation. The facilities are classified and their importance measured. This paper presents a framework for this project and is an initial draft of the facility descriptions. The opinions and ratings contained here are those of the author and do not reflect a consensus of IFIP WG 2.5.

AMS (MOS) Subject Classifications: 65-02, 00-A25, 68-A05

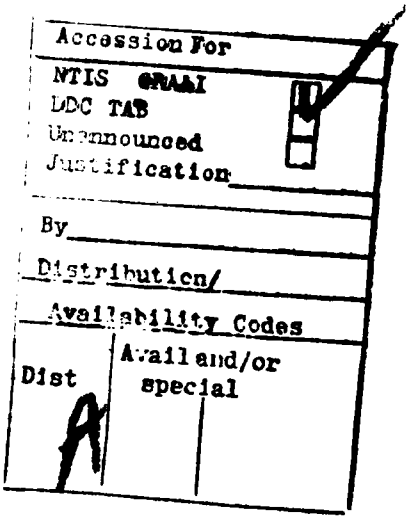
Key Words: Numerical computation, Programming language, Design criteria, Working paper

Work Unit Number 3 (Numerical Analysis and Computer Science)

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

SIGNIFICANCE AND EXPLANATION

Many programming languages being introduced are used primarily in numerical applications, but there is no source to guide the language designer in selecting the facilities (capabilities) to be included. This working paper provides a framework for such a source and presents an initial draft of material for ten topics. Facilities are described in general terms; as "what it should be possible to" rather than "this is how it should be included". It is recognized that there are many ways to provide a facility within the design of a programming language. This working paper is part of the activities of IFIP Working Group 2.5 (Numerical Software).



CONTENTS

PROGRAMMING LANGUAGE FACILITIES FOR NUMERICAL COMPUTATION

John R. Rice

INTRODUCTION

New programming languages are still appearing and old ones are being modernized. Even though the bulk of the use of these languages is numerical, there is no source to guide the language designer in the selection of facilities to be included in a language. The aim here is to produce such a source. Facilities are described in functional terms and there is no intention to describe syntax or even the details of semantics. There is no discussion of "general purpose" language facilities unless there is some special aspect of this facility relevant to numerical computation. It is recognized that there are normally many ways to provide a given capability within a particular programming language design. This working paper has been motivated and stimulated by the report of Dekker [1979] where some of the ideas expressed here are already found.

A framework is presented here for WG2.5 to consider for developing a source useful to language designers. It is proposed to have a list of perhaps 20 topics and to discuss each in the following format: (a) brief general definition, significance and background for the topic. (b) Brief descriptions of specific facilities that have been or would be useful for numerical computation. (c) Examples of how such facilities have been or would be included. The facilities are rated according to their estimated ease of inclusion in a language; facilities already widely available are rated near 100; those that are premature within the present programming technology and/or computer resource constraints are rated near 0. The facilities are also rated

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

by importance; a rating near 100 means a language lacking this facility is unlikely to be used for significant numerical computation.

2. CLASSIFICATION OF FACILITIES

The numerical values to be assigned to properties of the facilities discussed will have considerable subjectivity. Furthermore, some facilities are speculative in nature and there could well be unforeseen difficulties or benefits from their inclusion in a language. Nevertheless, judgements should still be made. A guide is given to assigning numerical values for ease of inclusion and importance.

For ease of inclusion we consider two aspects: the actual effort to include a facility and the frequency with which it is currently included. A facility that requires considerable effort to include and yet which is widely included would receive a high rating for ease of inclusion on the basis that a language designer is likely to put it in anyway.

Ease of Inclusion

Rating	Description of Facility Inclusion
80-100	Widely available or something that should be relatively straightforward to do without a significant perturbation of the language design or implementation (e.g. a new intrinsic function).
50-85	Something currently missing from some important languages, but which can be added with moderate effort and without a major change in the language (e.g. inclusion of expressions in output statements or loop control, inclusion of bit data type).
30-60	Something which requires a major change to add to a language; something which may substantially increase the complexity of the translator (e.g. completely dynamic storage allocation with garbage collection, symbolic differentiation in a Fortran or Algol-60 like language).

1-2 Something which changes the character of the language so as require resources in translation or execution that are presently not generally available (e.g. inclusion of a range of general problem solving statements, symbolic and numerical manipulation of functions and arrays of functions).

For importance we simply judge the overall usefulness of a facility. Keep in mind that given any facility there are some programs that do not use it and others which are helped greatly by it. The aspects of computing considered in the evaluations are: (a) clarity and effort of programming, (b) use of machine resources, (c) methodology for large scale projects (portability, modularity, etc.).

Importance

Rating	Description
90-100	Lack of this facility greatly complicates a wide variety of programs and probably eliminates the language from consideration in numerical computations.
60-80	A smaller percentage of the numerical software will be affected but the facility is still important enough to merit a redesign of the language in order to have it included.
30-50	A facility with relatively infrequent use (only 3-8% of the programs) or for which there is a straightforward (but perhaps obscure, cumbersome or inefficient) means to accomplish the same result. (Example: partitioning workspace through a dummy intermediate subprogram.) Such facilities should be included, but might be sacrificed to other considerations of the language design.
5-20	Nice, convenient at times, but not essential.

3. LIST OF TOPICS

A suggested list of topics is given. There is considerable overlap at some points and discussions of one topic may refer to those of others. In any case, this overlap is alright as long as no contradictions are created. The given order is random.

1. Library Facilities
2. Array Definitions and Manipulation
3. Vector-Matrix Algebra and Operations
4. Input/Output Facilities
5. Calculus Operators
6. Mathematical Functions Definition and Manipulation
7. Inter-program Communication
8. Standard Mathematical Functions
9. Exception and Error Handling
10. Variable Types
11. Environment Information
12. Precision Control
13. Declarations and Names
14. Problem Solving Statements
15. Storage Management
16. Data Structures

4. INITIAL DRAFT OF FACILITY DESCRIPTIONS

TOPIC 1: Library Facilities

A. General Discussion

Numerical computation is normally supported by an extensive library of programs. The contents vary from general purpose (e.g. sin(x), solve a linear system of equations) to very specialized (e.g. calculate the thermal conductivity of laminate of several materials). A typical user has access to 500-1000 library programs.

B. Facilities

1. Access to machine parameters. Many standard programs depend on parameters such as machine word length, characters per word or machine precision. A comprehensive set of these should be available within the computing environment.

Rating: Ease = 95, Importance = 5

The paper of Ford [1978] gives a standard set of machine parameters. These can be made available either as reserved names of the language or by calling an intrinsic library function. Some of these parameters are available as by-products of functions to manipulate integer and floating point variables, see Reid [1978], Brown [1979]. Mechanisms to provide this information are used rather widely by those preparing software targeted to many machines, see Aird [1977] for an example.

2. Separate Compilation. The concept of a library itself depends on the ability to independently develop subprograms for other, unknown programs to use.

Rating: Ease = 80, Importance = 90

Separate compilation is available in a number of languages, but missing in some important ones. Those languages where it is missing are either

designed for small scale computing or allow "extra-lingual" access to independently compiled programs, including programs from other languages. This latter alternative is better than nothing, but clearly inadequate as it is compiler and/or system dependent.

3. Dynamic Workspace Allocation. It is common for an algorithm to need temporary variables whose number is not known until execution. It should be possible to obtain memory space for these variables.

Rating: Ease = 50, Importance = 60

The objective is to create an array ARRAY(I), I = 1 to N within a subprogram when N is specified at execution time. Many languages allow this, but if it is missing then a workspace array must be passed to the subprogram. The majority of programmers do not understand how parameters are passed between programs; they are particularly ignorant about array parameters and thus passing workspace is particularly error prone. Those languages whose typing of arrays is so strong that it prevents passing them to separately compiled subprograms are, of course, particularly inadequate for libraries.

4. Hiding Names. Library programs are sometimes themselves large programs with many subprograms. The choice of names for or within these programs should not affect a user's program.

Rating: Ease = 70, Importance = 30

Block structures may be used to localize the existence of names and is quite common. Group structures, as presently proposed for inclusion in the next Fortran, are equally effective and are less constraining on the other facilities within a language.

TOPIC 2: Array Definitions and Facilities

A. General Discussion

Arrays are the most fundamental data structure of numerical computation. They permeate the mathematical and physical theories as well as arise naturally in a wide variety of algorithms. It is therefore important that special attention be paid to provide versatile facilities for their use.

B. Facilities

1. Complete Internal Specification. The internal representation of an array should contain complete information about its nature as well as its value. Thus all declared information (e.g. Type = Real, 3-Dimensions, Symmetric Storage, Dimension Ranges = (10,10,5), Range Indexes = (IA,JA,KA)) should be a part of the representation or directly available from the context without any special effort on the part of the programmer. Any passing of the array between programs would automatically carry along this information.

Rating: Ease = 70, Importance = 75

This facility is aimed at making true data types of arrays. At execution time one knows all the information about numeric variables and this treatment should extend to arrays. Block structure languages can accomplish this by never allowing the use of a variable to be "hidden" from its declarations.

2. Array Building and Subarray Extraction. It is common to build arrays from pieces and to analyze or use them in parts. The most common situations are to build N-dimensional arrays from (N - 1)-dimensional ones, to "border" a given array to enlarge it, to extract (N - 1)-dimensional arrays from N-dimensional ones and to extract similar but smaller arrays from a given one. Facilities are needed to do this naturally and directly. These

facilities probably depend on the presence of the array operations mentioned below.

Rating: Ease = 55, Importance = 35

These facilities have been included in several languages and considerable experience gained with their use, see NAPSS [1966], Bayer [1971], Paul [1979]. Such a set of facilities has been proposed for the next revision of Fortran.

3. Array Operations. The basic arithmetic operations should be extended to arrays on an element by element basis. Thus C = A + B or A = A - B is allowed provided the arrays conform. Similarly functions are extended to arrays as log(A) = array(log (a_{ij})) if A = a_{ij}. Finally, constants should be "broadcast" within array expression so that, for example, 3 can be used for the array with all elements = 3.

Rating: Ease = 65, Importance = 55

4. Range Indices. A range index of an array is a variable whose value is the current size of the array. Thus one might declare (in a clumsy form for clarity)

Real Array A/Dimension = 1/Storage = 100/Range Index = NA/

Thus changing NA changes the working size of A as it would be used in a calculation. If the language has full dynamic storage allocation, then the storage information is not needed; it is particularly useful to explicitly distinguish between storage information and the working size of an array.

Rating: Ease = 60, Importance = 65

The VECTRAN language contains range indices in an environment without dynamic storage allocation. A similar proposal has been made for the next Fortran revision. People experienced in consulting with users of libraries report that a significant percentage (30-50) of all the "non-trivial"

basis. Then A * B and A ↑ 2 do not mean "A times B" and "A squared". The identity matrix could be represented by a generic (with respect to size as well as type) reserved word, a function such as IDENT(N) or IDENT(A) (here A determines type and size) or A^0. The transpose can also be represented by a function or special operator. It is a delicate problem to choose natural syntax in a language that has both array operations and vector-matrix algebra.

2. Matrix-Vector Algebra. A vector is considered to be an $N \times 1$ matrix (column vector) and all matrix operations on vectors are interpreted this way.

Rating: Ease = 60, Importance = 75

It is natural to include vectors in the matrix algebra scheme and then one must resolve the ambiguity of "x times y"; it is $\Sigma\, x_i y_i$ (dot product) or the array $A = \{x_i y_j = a_{ij}\}$?

3. Submatrix Selection. A mechanism to select submatrices (including row and column vectors).

Rating: Ease = 55, Importance = 40

See the remarks in Example 2, item 2.

4. Inverse Matrices and the Solution of Linear Equations. The matrix inverse A^{-1} is to be included and one can express the solution of the equations $Ax = b$ as $x = A^{-1}b$. A separate linear equation solution facility might be provided for additional flexibility.

Rating: Ease = 50, Importance = 60

The matrix inverse and solution of equations is a facility at a higher level than commonly included in current "high" level languages. The code to implement it is longer (50 plus statements in Fortran) and there is no truly reliable test for the failure of the operation (when the matrix is

singular). A singular matrix failure is of a different nature than other exceptions and must be indicated appropriately.

5. Matrix-Vector Constants. See Facility 5 of Example 2.

Rating: Ease = 75, Importance = 30

See the remark for Example 2, item 5.

6. Specialized Matrix Structures. In addition to the general full matrices, there is a variety of well identified and common structures: band, symmetric, symmetric band. Special representations of these matrices are essential in many applications in order to efficiently use computer time and memory. Such representations can be included in the language if appropriate declarations and operator varieties are included.

Rating: Ease = 40, Importance = 40

The specialized matrix structures are important enough to have relevant facilities included in major libraries. Thus a translation writer will have little difficulty in implementing the matrix operations, but the complexity of declared types increases somewhat (e.g. REAL POSITIVE DEFINITE BAND MATRIX A(10,10)). There is also the added complexity of checking compatibility in matrix/vector operations.

TOPIC 4: Input/Output

A. General Discussion

Numerical computation naturally involves vectors, matrices and functions, thus a language should facilitate input and output involving them.

B. Facilities

1. Tabular Output. One wants to say "TABLE X, DATA 1,DATA 2" or "TABLE MATRIX" and automatically receive a reasonable table on the standard output device (printer). The dimensions of the table are determined by the working

sizes of the vectors or matrices involved; headings are given along with an indexing.

Rating: Ease = 90, Importance

The same facility for functions is needed. There is a slight complication in choosing the range and increment; defaults and/or specifications (e.g. "TABULATE FUNCTION [0,6]") should be implemented.

Rating: Ease = 85, Importance = 55

This facility is primarily to reduce to detailed and error-prone specifications commonly required to produce even a simple table. The implementation of tabulation is not very difficult if fairly inflexible formats are used. The tabulation of functions requires more information (the range and, for multivariate functions, the independent variable plus, possibly, values for other variables) and hence more complex syntax. Such facilities have been included in a variety of systems.

2. Printer Plotting. The printer plotting facility is similar to the tabulation facility above; a similar syntax could be used to produce "working quality" plots of groups of vector (e.g. PLOT K VERSUS DATA 1, DATA 2) or functions.

Rating: Ease = 80, Importance = 70

Printer plotting is currently a common library routine which should be part of a numerical computation language. Even crude printer plots require considerable thought (and code) to implement well, see [xxxx] for example algorithms.

3. Graphical System I/O. In addition to the common printer plotting capability, one wants access to a system which gives much higher resolutions and more capabilities. A detailed description of such capabilities is not

given; hopefully they will be rather common and somewhat standardized in the near future.

Rating: Ease = 50, Importance = 75

A connection to a graphical system essentially requires the very widespread availability of facilities accessed from the language. The trend in graphical systems is in this direction (see [Synder, 1978], [SIGGRAPH Notices] for an example and for further discussion). The range of capabilities possible within a graphics system is very broad and one of the critical design points is to identify a useful subset that can be accessed naturally and concisely. Universality of the language can be achieved by having the graphical access default to printer plotting when no graphical system is available.

4. Data Storage Access. Data in the form of vectors, matrices or just groups of variable values should be available from the computer system data storage facility. This capability is useful both for input and output.

Rating: Ease = 40, Importance = 35

The basic aim is to allow one to create or enter data into a computer system, manipulate it and pass it from program to program with minimal attention (on the programmer's part) to storage formats and related matters. The basic facility needed here is much less than a general data base system in that simple data structures are involved (aggregates of variables, vectors or matrices) which are handled as integral units. Such facilities are commonly available in file handling systems. As with access to a graphics system, the crux of the design here is to identify a useful subset that can be accessed naturally and concisely.

TOPIC 5: Calculus Operators

A. General Discussion

The training of scientists and engineers includes an almost universal introduction to a number of basic operators which we loosely call the calculus operators. Some of these (sum = Σ, product = Π, max, min) are simple finite operators which can be incorporated easily into a language. Others - differentiation, integration, infinite summations and function extrema - are algorithmically much more complex.

The operators of facilities 2 through 5 cannot be implemented by an algorithm, their values are not computable functions (technically speaking) of their arguments. Their inclusion in a language thus represents a significant new feature. Implicit in the ease ratings is the belief that the state of the art allows one to implement them reliably. That is, in practice good estimates can be obtained in most cases; computational failure can be detected in most of the remaining cases and truly erroneous results are produced in a very small fraction of the computations. The level of reliability of these implementations will be comparable to that of general numerical computation and probably better than currently exists in most programs where such operators are supposedly used.

Another new aspect of these operators is the complexity of the algorithms used to implement them; they will run to hundreds (and possibly thousands) of statements in a language at the Fortran-Algol level.

B. Facilities

1. The Finite Operators. The three most important are MAX, MIN and Σ which operate on vectors or matrices. A range or domain must be used; vectors and matrices are likely to have a range variable which can be used as

default. The product operator Π is much less commonly used, but it might be included for the sake of completeness. Extensions to other operators of a similar nature (e.g. average, median) could also be incuded.

Rating: Ease = 90, Importance = 80

The finite operators are straightforward to include in a language and their widespread occurrence makes it unreasonable to exclude them.

2. Differentiation. The derivative of a function of one variable may be evaluated symbolically or estimated numerically. The difference between these two choices is very large; both in the nature of the results obtained and the technique of implementation. The partial derivative of a multivariate function can be obtained by indicating which variable is to be considered as the independent variable.

Rating (symbolic): Ease = 40, Importance = 75

Rating (numerical): Ease = 60, Importance = 50

Differentiation is unique in that a numerical implementation is tricky while a symbolic implementation is not conceptually difficult. Technically speaking, differentiation is an unbounded operator and this manifests itself in practice by making the estimation difficult. On the other hand, it is feasible to symbolically differentiate more or less arbitrary programs (see [Crary, 1979], [Kedem, 1979]). Symbolic differentiation does require substantial manipulation of the source text. This can be done at translation time, but this approach is not very compatible with separate compilation. Both numerical and symbolic differentiation were implemented in the NAPSS system [Roman, 1968], [Symes, 1967], [Oldehoeft, 1972] and it was found that the symbolic implementation was significantly more efficient and significantly more reliable.

3. Integration. The integral of a function of one variable may be estimated numerically or, in some cases, evaluated symbolically. The difference between these two choices is as large as for differentiation, but the balance is considerably different. A syntax close to the mathematical one

$$\int_a^b f(x)\,dx$$

should be adopted. The function $f(x)$ may depend on variables in addition to x.

Rating (symbolic): Ease = 30, Importance = 40

Rating (numerical): Ease = 70, Importance = 50

Integration is the classical example of a non-computable operator; one can easily construct a function for any given algorithm which leads to zero for the estimated value and for which the true value is 1. On the other hand, very reliable algorithms exist for numerical integration, see [Lyness, 1979] and cited works for more details. Great practical and theoretical advances have been made in symbolic integration since the 1960s [Moses, 1978], but there are still many functions which cannot be integrated so numerical integration must be used often. In an ideal language, symbolic integration would be used when it produces results quickly and numerical integration otherwise.

4. Infinite Series Summation. An infinite series is of the basic form

$$\sum_{N=0}^{\infty} F(N)$$

where F is a function defined on the integers. F might depend on variables

or parameters other than N. Symbolic methods of summation exist, but are probably not feasible to use.

Rating: Ease = 65, Importance = 40

Computational experience with infinite series is not very large, but it seems plausible that the difficulty of implementing this operator reliably lies somewhere between that of differentiation and integration.

TOPIC 6: Mathematical Function Definition and Manipulation

A. General Discussion

Functions are basic items in mathematical models and occur pervasively in numerical computations. It is essential that special attention be paid to providing adequate facilities for them.

B. Facilities

1. Natural Definitions. A function should be defined in a natural, concise form. The definition should be an independent program unit with all the programming language facilities available for the definition. It is natural to have functions depend on both "independent variables" and "parameters"; it is convenient, but not essential, that this distinction be possible in the function definition.

Rating: Ease = 90, Importance = 90

Most current languages provide good facilities for defining functions. The distinction between independent variables and parameters is not normally made and this is reflected in the common difficulty in using something like a library integration routine. The routine expects as argument f(x) and the user has a function M(x,a,b,q) where a, b and q are parameters. Most current languages force one to create an "intermediate" function from M which can be passed to the integration routine.

2. Functions as Program Variables. The common notation of science allows one to manipulate functions as separate entities [e.g. $h(x) = f(x) + A*(x^2 + 3 \sin(x))$] and a programming language can allow this also. Different choices are possible for the treatment of parameters, that is, A in the above example could be fixed at the time h(x) is assigned or it could continue to be an ordinary program variable (the latter is the more logical). Functions should be passed between programs as single entities in a natural way.

Rating: Ease = 30, Importance = 65

The natural manipulation of functions appears to require some kind of symbolic facility at execution time. This is seen as a large burden on languages that are compiled; some slightly limited manipulation facility is probably possible through manipulating object code instead of source code. This facility was included in NAPSS [1966].

3. Arrays of Functions. Function arrays $f_i(x)$ and $f_{ij}(x)$ occur frequently in applications (gradients, Jacobians, etc.). In principle, $f_i(x)$ is the same as f(x,i) but, in practice, one wants to allow the vector/matrix operations of the language to apply naturally to these special kinds of functions. The rating assumes that functions are already included in the language.

Rating: Ease = 80, Importance = 55

Arrays of functions were included in NAPSS [1966].

4. Mathematical Typing. Mathematics has a large number of schemes to classify (or type) functions (e.g. analytic, polynomial, differentiable, trigonometric). These types might or might not reflect anything about the

representations of the functions; the information is of use in processing the functions within mathematical procedures.

Rating: Ease = 55, Importance = 40

It is anticipated that more complete languages for numerical computation will include some procedures for processing functions (e.g. integration, differentiation). The algorithms to accomplish this must either "be told" or "discover" various properties of the functions; the additional information available from even a modest typing facility can increase the efficiency of these algorithms dramatically. If the symbolic text is available at execution time then this kind of information could be obtained by a symbolic scan.

TOPIC 7: Inter-program Communication

A. General Discussion

Inter-program communication is a "general" programming language feature, but certain facilities are widely needed in numerical computations which are frequently lacking. The facilities are usually needed by large programs or libraries and the need is probably not restricted to numerical computation.

B. Facilities

1. Global Declaration. A declaration like GLOBAL A,B,XYZ,... should make the variables named available to all programs present at translation time.

Rating: Ease = 80, Importance = 50

Many large numerical software projects involve a model of something which is described by program variables. These variables are meaningful to everyone working on the project and it is to be possible to agree on names for them and then allow any program (of the project) to use any one of them as needed.

2. Partitioned Global Declarations. Global declarations are allowed to have different scopes. Thus GLOBAL A,B; GLOBAL C,D could be used for 10

programs use COMMON A,B; COMMON X,Y used for a second 10 programs. The variable A of the first 10 programs would not conflict with the variable A of the second 10.

Rating: Ease = 70, Importance = 30

This facility is essentially the same as "Hiding Names" for libraries. It is useful for "top down" organizations of programs; the variables of programs to handle the tail assembly of an airplane need not be aware of and should not conflict with variables of the fuel supply systems.

3. Subprogram Argument Lists. The argument list of a subprogram can include any variable of a program in a simple, direct way.

Rating: Ease = 55, Importance = 35

The essential information about any particular program variable should be attached to the variable and passed to subprograms (or through other communication mechanisms) along with the variable. Thus a matrix is passed along with its dimension and range information.

4. Internal Procedures. A simple procedure (function) to be defined inside a larger program. All the variables, definitions, etc. of the larger program extend to the internal procedure.

Rating: Ease = 65, Importance = 50

This is a simple case of the block structure of Algol; separate compilation of internal procedures would not occur. It is very natural to introduce such procedures and sometimes cumbersome to simulate in a language like Fortran.

5. Variable Argument Lists, Defaults. The number of arguments to a subprogram should allow for default values for missing values. A mechanism to identify arguments explicitly (instead of by position in the list) is needed.

Rating: Ease = 40, Importance = 50

A common problem is the necessity of lengthy argument lists where most

invocations use only a part of the variables and yet valid values must be provided for the unused arguments. A scheme like FUNCTION F(x,y,ABAR = 0, BBAR = 1.0, LENGTH = ZAP) would allow F(x,y) to be used. This approach leads to complexity if only LENGTH is to be provided, e.g. F(x,y,,,17). An alternative is to identify arguments (using the same syntax as above) so that F(x,y) is equivalent to F(x,y,0,1.0,ZAP) and F(x,y,LENGTH = 17.) is equivalent to F(x,y,0,1.0,17.). Note that this particular facility could be used to provide a distinction between independent variables and parameters for functions.

TOPIC 8: Standard Mathematical Functions

A. General Discussion

A large number of functions have been identified that are useful in various branches of mathematics, statistics and science. They have standardized definitions and are part of the "tools" of science in their particular areas.

B. Facilities

1. Elementary Functions. These functions include the trigonometric (sine, cosine, secant, arctan, etc.), exponential and logarithm, exponentiation and roots, simply evaluated (absolute value, modulus, etc.) and the hyperbolic.

Rating: Ease = 95, Importance = 95

These functions are present in most current languages and there is a large body of knowledge about their efficient evaluations [Hart et al., 1968], [Fike, 1968]. These functions permeate scientific computation.

2. Higher Mathematical Functions. There is a large number of more specialized functions, often called higher transcendental functions, which are

important to specific fields of science. The more common include [illegible] (factorial), Bessel functions, Elliptic Integrals and Mathieu functions.

Rating: Ease = 90, Importance [illegible]

The inclusion of higher mathematical functions does not pose any language design problems; there is an increased number of special names and the support library becomes larger as more of these are included. There are several general references for these functions [Abramowitz and Stegun, 1964], [Erdelyi, 195x-5y]. Careful analysis of evaluation techniques have been made for many of these functions [Cody, 197x]; the Collected Algorithms of the ACM containing many instances. Some of the less common functions have never been analyzed for computer evaluation and are evaluated by formulas from classical mathematical analysis.

TOPIC 9: Exception and Error Handling

A. General Discussion

A common situation in numerical computation is the use of multi-layered software where errors may occur in a program completely unknown to the programmer, the operating system or the language translator. Thus the language itself should have facilities to adequately detect errors and exceptions and to transmit relevant information to the programmer. The four basic types of errors are: (a) Arithmetic limits exceeded (overflow, underflow, NAN (Not-A-Number) arithmetic), (b) Mathematical Errors (1.1^{10}, $\sqrt{-1.}$, log(-2.3), $J_{1/3}(-4.5)$), (c) System Errors (inadequate storage, undefined I/O media, time exceeded) and (d) Numerical Failures (Ax = b problem with A singular, $\sin(50^{50})$, solve $x^2 + 1 = 0$).

B. Facilities

1. Identification of Error Type. A complete classification of errors is defined and the type is identified in the error report. Relevant parameters

of the error occurrence are provided (e.g. arguments to [illegible] errors, data for system routine failures (bad formats)).

Rating: Ease = 75, Importance = 75

2. Identification of Error Location. The location of errors should be reported in the programmer's terms (i.e. error at line 16 of program SUB1CX, called from line 14 of BEST4).

Rating: Ease = 65, Importance = 80

The language implementation is to maintain a record at execution time of program locations or to be able to reconstruct this information in almost all circumstances (including things like "time exceeded", "operator abort" or "illegal input format"). Absolute address tracing through system loader mapping tables is to be done by the computer, not people.

3. Message Transmission. Messages should be in the programmer's terms as much as possible. A software project should have the ability to modify the messages to put them in the context of the project.

Rating: Ease = 70, Importance = 75

4. Error Recovery Control. The error type should be available for test before an irreversible action is initiated. The language should allow tests on the type and permit the program to remedy or modify the situation without aborting the entire computation.

Rating: Ease = 75, Importance = 60

Even such situations as "time exceeded" could transfer control to a user program for a final, short computation. This option is nice for small or individual projects, it is almost essential for robust, user oriented application systems.

TOPIC D: Variable Types

A. General Discussion

Mathematics has developed several number systems that occur regularly in numerical computation and there are number systems (e.g. different precisions) that are relevant only to computation. In addition, there are non-numerical types (e.g. strings) that many programs need and there are mathematical data structures (e.g. vectors, matrices, function arrays) that are variable types various computationally important mathematical systems.

B. Facilities

1. Real Numbers. Numerical computation cannot occur without a reasonable real number facility.

Rating: Ease = 100, Importance = 100

Most meaningful numerical applications require a precision of 5 or so decimal digits and a numerical range of $10^{\pm 50}$ or so. Eight to ten digits of precision appears to be an economical choice for the basic real number arithmetic. See Reinsch [1979] for a detailed analysis of the properties required of real number arithmetic.

2. Integer Numbers.

Rating: Ease = 100, Importance = 15

Integer arithmetic is traditional for programming languages, but there is little evidence that it is necessary or even desirable in a language for numerical computation. This does not mean that certain integer oriented operations (e.g. fractional part, modulus) can be omitted.

3. Higher Precision Real Arithmetic. Whatever the precision of the basic real arithmetic, there needs to be an additional level of precision.

Rating: Ease = 95, Importance = 90

The basic need for higher precision is to provide a mechanism for evaluating

the accuracy of numerical computations. This need is recognized in the design of many computers where hardware is present for double-precision. People who require truly extended precision (50, 500 or 5,000 decimal digits) should have access to a portable, unlimited precision package, e.g. Brent [1977].

4. Complex Variables. A significant, but not large, proportion of numerical applications require the complex number system.

Rating: Ease = 70, Importance = 55

Note that if complex arithmetic is included then two levels of precision are needed to provide for accuracy testing.

5. Character Strings. The basic operations (matching, concatenation, etc.) for character strings are needed in providing user oriented output and in manipulating data of various types.

Rating: Ease = 60, Importance = 50

6. Combinations. Variable types tend to refer to different attributes and thus be natural candidates for combining. One can visualize "Triple Precision Complex Positive Definite Band" matrices or "Integer *20 Complex Polynomial" functions. The number of combinations (and effort of implementation) grows rapidly with the number of basic types to be combined. A language designer can attempt some general "type combination" mechanism or make a decision on which combinations are worth including.

7. Mathematical Entities. Variables like vectors, functions and matrices are considered under other topic headings of this report.

5. INCOMPLETE DESCRIPTIONS

TOPIC 11: Environment Information

1. Numeric Representations, Word Structure
2. Accuracy/Precision Information
3. System Information

TOPIC 12: Precision Control

1. General Context, Levels
2. Variable Precision

TOPIC 13: Declarations, Words and Names

1. Declaration Methods
2. Reserved Words
3. Name Structure

TOPIC 14: Problem Solving Statements

1. Statement Appearance
2. Flexibility

TOPIC 15: Storage Management

1. Block Storage Allocation/Deallocation
2. Dynamic Storage Allocation

TOPIC 16: Data Structures

1. Basic Types (Lists, Stacks, Queues,...)
2. Matrices
3. Trees
4. Graphs

REFERENCES (Rough and Incomplete)

Dekker [1979] = T. J. Dekker, Design of Languages for Numerical Algorithms, Dept. Math. Report 79-10, University of Amsterdam.

Ford [1978] = Brian Ford, Parameterization of the Environment for Transportable Numerical Software, Trans. Math. Software, 4, pp. 100-103.

Reid [1978] = John K. Reid, Intrinsic Functions for Numerical Computation, IFIP WG2.5 (Toronto-21) 421, to appear.

Brown [1979] = W. S. Brown and S. I. Feldman, Environmental Parameters and Basic Functions for Floating-Point Computation, IFIP WG2.5 (Novosibirsk-AA) 6AA, to appear.

Aird [1977] = T. J. Aird, E. L. Battiste and W. C. Gregory, Portability of Mathematical Software Coded in Fortran, ACM Trans. Math. Software, 3, pp. 113-127.

NAPSS [1966] = General reference to NAPSS system (Rice and Rosen paper).

Bayer [1971] = Paper in Mathematical Software (J. Rice, Ed.), Academic Press.

Paul [1979] = Manuals on the potential language VECTRAN, G. Paul and W. Wilson, IBM Corp, Yorktown Hgts., N.Y.

Novosibirsk [1979] = Recommendations from IFIP WG2.5 subcommittee on X3J3 proposals for linear algebra.

Synder [1978] = W. V. Snyder, Algorithm 531. Contour Plotting, ACM Trans. Math. Software, 4, pp. 290-294.

SIGGRAPH NOTICES = Selected references on the progress towards standards in graphical systems.

Crary [1979] = F. D. Crary, A Versatile Precompiler for Nonstandard Arithmetics, ACM Trans. Math. Software, 5, pp. 204-217.

Kedem [1979] = A paper to appear in TOMS using the Crary [1979] system to differentiate Fortran programs.

Symes [1967] = Ph.D. thesis contains info on NAPSS systems.

Roman [1968] = ditto.

Oldehoeft [1972] = ditto.

Lyness [1979] = References on the effectiveness of numerical integration algorithms:

F. T. Krogh and W. V. Snyder, Comparison of Software for Numerical Quadrature, SIAM Review 20 (1978), p. 634; also Report CL77-576 Jet Propulsion Lab.

J. N. Lyness and J. J. Kaganove, A Technique for Comparing Automatic Quadrature Routines, Comp. J. 20 (1977), pp. 170-177.

Moses [1978] = One of Joel Moses' surveys of the state of the art in symbolic integration.

[xxxx] = Reference to algorithms for graphs on printer output.

Abramowitz and Stegun [1964] = Abramowitz, M. and Stegun, I., Handbook of Mathematical Functions, AMS 55, National Bureau of Standards.

Hart, et al. [1968] = Hart, J. F., Cheney, E. W., Lawson, C. L., Maehly, H. J., Mesztenyi, C. K., Rice, J. R., Thacher, H. C. and Witzgall, C., Computer Approximations, John Wiley, New York, 1968.

Erdelyi [195x-5y] = Series of books on Higher Transcendental Functions and related topics.

Fike [1968] = Fike, C. T., Computer Evaluation of Mathematical Functions, Prentice Hall, Englewood Cliffs, NJ, 1968.

Cody [197x] = FUNPACK reference.

Reinsch [1979] = Reinsch, C., Principles and preferences for computer arithmetic, SIGNUM Newsletter, xx (1979), pp.

Brent [1977] = Brent, R., Algorithm xxx. Multiple-Precision Arithmetic Package, TOMS.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) **REPORT DOCUMENTATION PAGE**

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MRC-TSR-2033	2. GOVT ACCESSION NO.: AD-A083815
3. RECIPIENT'S CATALOG NUMBER: (9) Technical	
4. TITLE (and Subtitle): (6) PROGRAMMING LANGUAGE FACILITIES FOR NUMERICAL COMPUTATION	5. TYPE OF REPORT & PERIOD COVERED: Summary Report, no specific reporting period
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) John R. Rice	8. CONTRACT OR GRANT NUMBER(s): (15) DAAG29-75-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Work Unit #3 - Numerical Analysis and Computer Science
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office P. O. Box 12211 Research Triangle Park, North Carolina 27709	12. REPORT DATE: (11) January 1980
	13. NUMBER OF PAGES: (12) 34 ~~22~~
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Numerical computation
Programming language
Design criteria
Working paper

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a working paper for IFIP Working Group 2.5 (Numerical Software) to stimulate discussion of a possible group project. The project is to describe the facilities in a programming language that make it useful for numerical computation. The facilities are classified and their importance measured. This paper presents a framework for this project and is an initial draft of the facility descriptions. The opinions and ratings contained here are those of the author and do not reflect a consensus of IFIP WG 2.5.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

221200

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)