Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Kok, G.T. Symm

A proposal for standard basic functions in Ada

Department of Numerical Mathematics Report NM-R8407 June

A PROPOSAL FOR STANDARD BASIC FUNCTIONS IN ADA

J. KOK
Centre for Mathematics and Computer Science, Amsterdam

G.T. SYMM
Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, U.K.

This paper contains a proposal for a standard basic mathematical functions package for scientific computation in Ada. The package is transportable to machines with different floating-point types and its availability will enhance the portability of numerical software.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 69D49, 65-04.
KEY WORDS & PHRASES: Ada, high level language, basic mathematical functions, scientific libraries, portability.
NOTE: This report will be submitted for publication elsewhere.

Report NM-R8407
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

PREFACE

During 1982 and 1983 the National Physical Laboratory, Teddington, and the Centrum voor Wiskunde en Informatica, Amsterdam, were engaged in an investigation of the possibilities of designing large modular scientific libraries in Ada. The project was funded by the Commission of the European Communities and culminated in the production of a set of Guidelines (Symm et al., 1984) which include recommendations on the ways in which Ada can and might be used in this context.

One of the recommendations made was that a standard specification of the basic mathematical functions should be adopted as soon as possible. In this paper, we present the proposal for such a standard with an outline of the several possibilities considered and with justification for the options chosen.

For further details we refer the reader to the full report on the project (Symm et al., 1984), which also treats the following subjects: types for composite data structures (COMPLEX, VECTOR, MATRIX), information passing, error handling, working-space organisation, computations in a real-time environment (use of tasks).

1. INTRODUCTION

The programming language Ada (ANSI/MIL-STD 1815 A, 1983; hereafter referred to by the abbreviation LRM for Language Reference Manual) was primarily designed for the production of large portions of readable, modular, portable and maintainable software for real-time applications. It is generally expected, however, that it will also be widely used in large-scale scientific computation.

For the production of portable, reliable and efficient software for scientific computation, the acceptance of a standard specification for the basic mathematical functions (like SQRT, LN or LOG, EXP, SIN, COS, ARCTAN) is a prerequisite (see, for example, Rice, 1983).

Aspects to be considered when discussing a proposal for such a standard are:

- in what sense calculations may be considered to be portable,
- the introduction of the available floating-point types and of user-defined types into the collection of functions (in the sequel we assume this collection to be a **package**),
- the choice of the basic mathematical functions,
- the specification of each function, including: name, types or subtypes used for parameters, formal names of parameters and possible defaults, result type,

- hierarchy, if any, of the components of the package of basic mathematical functions,

- use of exceptions and other implementation recommendations.

A full discussion can be found in the final report on the project (Symm et al., 1984), especially in Chapter 3 (on the introduction of floating-point types into a package), Chapter 4 (discussion of the options and choices made) and Appendix C (concerning argument ranges and an exemplary implementation).

The package proposed here offers users a desirable amount of flexibility without any cumbersome preliminaries. The relevant considerations are summarized in the following sections.

2. PORTABILITY OF COMPUTATIONS

In general, programmed floating-point computations are never fully transportable as regards their results, because of the differences, between machines, in the accuracy of the available hardware types and in the performance of the hardware real arithmetic.

In Ada one can use predefined types, such as FLOAT, LONG_FLOAT, etc., which presumably (but not necessarily) exploit the available hardware types as well as possible. However, one can also declare real types which are independent of the predefined types, e.g.

```
type REAL is digits 10 range -1.0E+40 .. +1.0E+40;
```

If all computations (e.g. in the basic mathematical functions) are programmed for this type REAL, then they will be transportable to all machines where the definition of such accuracy and range constraints is allowed (i.e. where the resources requested by the type definition can be met by the hardware types). However, the results will still differ because of the differences in the floating-point arithmetic. Moreover, for the source code to be transportable between even two machines, one may be forced to choose a small number of digits in the accuracy constraint. For many applications this is not a desirable approach.

The portability which numerical analysts would like can be described as follows:

> An algorithm should perform as well as possible on any machine to which it may be moved without the need for large adaptations of its source text. Good performance here includes efficient and accurate computation.

Such portability may be obtained for source code which uses sufficient information about the hardware types, this information being extracted from a standard set of environment parameters. This set should consist of constants and manipulative functions (see, for example, Ford, 1978; Cody, 1982). The language Ada offers environment parameters which describe the floating-point types through the so-called type attributes (e.g. REAL'DIGITS or REAL'MANTISSA) and additional information is given in the standard package SYSTEM (LRM 3.5.8 and 13.7).

Whether computations in Ada use a predefined type like FLOAT or a user-defined floating-point type, the algorithms should not depend on the specific accuracy and range of this real type. Instead, they should contain source code which will perform well on a range of intended target machines. This may be achieved by implementing algorithms which are branched with respect to values of the real type attributes. (We do not discuss the sufficiency of the Ada environment parameters.)

We believe that for some mathematical functions (e.g. SQRT) it is possible to design portable bodies yielding function results with accuracy deviating little from the accuracy of the definition of the type used. Such bodies can be implemented completely in Ada and in such a way that, for two different floating-point types, function evaluations require less computational effort for the type with the smaller accuracy constraint. For other functions, however, this design, which requires the numerical stability of the function as well as of the available methods, cannot be achieved. For these functions, we have to accept larger differences in performance when we move source code.

If installations have hardware functions available for the different hardware floating-point types, they will (usually) provide implementations of the basic mathematical functions by connecting the declarations to these hardware functions. The effect to the user should be the same as if the bodies were (portable) Ada source text, ignoring the expected differences in computational effort which are insignificant for the basic mathematical functions. However, this is only true if the hardware functions are sufficiently accurate and if out-of-bounds values are or can be trapped (equivalent to raising an exception in Ada). These requirements should always be checked.

3. PARAMETRIZING WITH DIFFERENT FLOATING-POINT TYPES

From the previous section, it is clear that the use of FLOAT or LONG_FLOAT in calculations does not automatically yield portable mathematical functions. Attempts to achieve a certain accuracy with FLOAT might raise an exception on one machine, where the required accuracy may be available through LONG_FLOAT, whereas on a different machine FLOAT may be sufficient (or even more than adequate). Use of the predefined types gives no flexibility at all.

If a user desires to calculate with LONG_FLOAT, he cannot apply a mathematical function using the type FLOAT (see, e.g., Whitaker and Eicholtz, 1982). He can of course write explicit type conversions everywhere, but these do not yield more accuracy in the function values.

Even if the implementation of a mathematical function takes account of the actual precision of FLOAT, by exploiting the available floating-point attributes (see Wallis, 1983), and uses different approximations for different values of the MANTISSA attribute for example, application to values of the type LONG_FLOAT is invalid.

A first attempt to obtain some flexibility might lead to packages which use floating-point types appropriate to the machine, these types being declared in a library package such as:

```
package REAL_TYPES is
   type REAL is new FLOAT;
   type DOUBLE is new LONG_FLOAT;
   -- or other suitable implementations for REAL and DOUBLE
end REAL_TYPES;
```

Then a user's program, or a library package, e.g. for providing basic mathematical functions, can use these types through a context clause, thus:

```
with REAL_TYPES; use REAL_TYPES;
package BASIC_MATHEMATICAL_FUNCTIONS is
   ...
end BASIC_MATHEMATICAL_FUNCTIONS;
```

However, a disadvantage of the latter package is that it cannot be used for user-defined floating-point types (except for derived types).

A better solution, which is more in keeping with Ada Style (Nissen and Wallis, 1984), is to write library packages which are generic with respect to the real type(s) to be used within them. For the basic mathematical functions, we then have:

```
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is
   ...
end GENERIC_MATH_FUNCTIONS;
```

This library unit can be used with any user-defined floating-point type but we recommend that an installation should also provide a standard instance as a library unit for those users who do not want to give further thought to the possibilities which Ada offers here. This would avoid the need for separate instantiations in all dependent packages, possibly yielding many copies of an instance. The standard instantiation would read:

```
with GENERIC_MATH_FUNCTIONS;
package STD_MATH_FUNCTIONS is
   new GENERIC_MATH_FUNCTIONS(FLOAT);
      -- or possibly with LONG_FLOAT.
```

A particular installation may, for reasons of efficiency, effect an instantiation of the above generic package by calling an equivalent non-generic version (which may even be on a special-purpose chip). As far as the user is concerned, the fact that this is not an instantiation in the normal sense will not be evident and will not matter.

In the generic package above it is assumed, for simplicity, that only one floating-point type is needed by the implementation. We leave the generalisation, for different particular needs, to the reader. The situation where a different floating-point type (with presumably a higher accuracy) is needed by the package body (the implementation of the functions) but not by the package specification (the visible part) is a subject of continuing concern to the Ada-Europe Numerics Working Group.

4. CONTENTS OF THE PROPOSED PACKAGE

Unlike many other common programming languages, Ada does not include elementary functions (other than the usual arithmetical operations) in the language definition. However, provision for these functions can be readily made through packages written in Ada and such packages, if sufficiently standardized, may be viewed as extensions to the language.

For the contents of the basic mathematical functions package we have chosen those functions which are available in most languages, assuming that other (special) mathematical functions are less frequently required. We have included in this package number declarations for the mathematical constants PI and e (the base of natural logarithms, which we have named EXP_1). Also, one exception declaration is proposed, viz ARGUMENT_ERROR. This exception can be raised by functions which detect that the argument given is not in the prescribed domain (see below).

The source text which follows contains all the function specifications, and hence provides complete information regarding standard naming, types or subtypes used for parameters, formal names of parameters and possible defaults, and the type of each function result.

In this package specification we have taken the opportunity to incorporate more general versions of the usual EXP, LN (natural logarithm), SIN, COS, TAN, COT, ARCTAN and ARCCOT functions (as explained in the following section). We may regard their declarations as being overloaded with both the usual functions (calls with default second parameter) and related functions, viz. a power or logarithm with arbitrary base, circular functions with arbitrary period, and inverse circular functions ARCTAN and ARCCOT with two parameters for accurate results near PI/2. Correspondingly, we have taken the more general name LOG instead of LN. Traditional calls like SIN(X) are allowed and yield the usual results. We consider it appropriate to have complete sets of circular and hyperbolic functions, as otherwise additional packages might arise with less logical structure.

The complete package declaration reads as follows:

```
-------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is
   ----------------------------------------------------------------
   -- Declare constants.                                         --
   ----------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ----------------------------------------------------------------
   -- Declare the basic mathematical functions.                  --
   ----------------------------------------------------------------
   function SQRT(X : REAL) return REAL;
   function LOG(X : REAL; BASE : REAL := EXP_1) return REAL;
   function EXP(X : REAL; BASE : REAL := EXP_1) return REAL;
   function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function TAN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COT(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function ARCSIN(X : REAL) return REAL;
   function ARCCOS(X : REAL) return REAL;
   function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
   function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
   function SINH(X : REAL) return REAL
   function COSH(X : REAL) return REAL;
   function TANH(X : REAL) return REAL;
   function COTH(X : REAL) return REAL;
   function ARCSINH(X : REAL) return REAL;
   function ARCCOSH(X : REAL) return REAL;
   function ARCTANH(X : REAL) return REAL;
   function ARCCOTH(X : REAL) return REAL;

   ----------------------------------------------------------------
   -- Declare exception.                                         --
   ----------------------------------------------------------------
   ARGUMENT_ERROR : exception;
   ----------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
-------------------------------------------------------------------
```

The 35 digits given for the declarations of the two mathematical constants should be sufficient for most purposes. The results to be expected from function calls follow from the usual mathematical definitions, except for the extended calls (i.e. the calls with a second actual parameter) of ARCTAN and ARCCOT. For full details of the latter see Symm et al. (1984), where we have also given guidelines (for the package body) regarding the delivered accuracy and the raising of exceptions. No textual error messages should be issued.

The requirements for the parameters of all the functions are:

Function	Argument and range
SQRT	X >= 0.0
LOG	X > 0.0, BASE > 0.0 and /= 1.0
EXP	X unrestricted, BASE > 0.0
SIN	X unrestricted, CYCLE /= 0.0
COS	X unrestricted, CYCLE /= 0.0
TAN	X unrestricted, CYCLE /= 0.0
COT	X unrestricted, CYCLE /= 0.0
ARCSIN	**abs** X <= 1.0
ARCCOS	**abs** X <= 1.0
ARCTAN	**not** (X = 0.0 **and** Y = 0.0)
ARCCOT	**not** (X = 0.0 **and** Y = 0.0)
SINH	X unrestricted
COSH	X unrestricted
TANH	X unrestricted
COTH	X unrestricted
ARCSINH	X unrestricted
ARCCOSH	X >= 1.0
ARCTANH	**abs** X < 1.0
ARCCOTH	**abs** X > 1.0

We recommend that implementations raise the exception ARGUMENT_ERROR of the package if an argument violates these requirements. Note that predefined exceptions can still be raised during computations, e.g. when REAL has an uncommon range constraint or if division by zero occurs. We assume that overflow (arising, for example, from too large an argument for EXP) will cause the raising of NUMERIC_ERROR if Ada implementations are used for the function bodies. Unfortunately, if the functions are hardware-provided, this cannot be expected.

No exception is proposed for the situation where the argument in a function call is such that the function cannot be evaluated with useful accuracy. Incorporation of such an exception, to be called SIGNIFICANCE_ERROR, was considered but was rejected. The problem is that it is difficult, or at least rather cumbersome in Ada, to specify the accuracy expected for each function (see Symm et al., 1984, Chapter 4, for more details).

5. DISCUSSION

The function names are mostly in agreement with those familiar in Algol 60 and Pascal (and in mathematics of course) and we have ignored the sometimes anomalous naming which arises, due to implicit type conventions, in Fortran. One exception is the logarithmic function LOG whose definition is more general than that of the natural logarithm indicated by 'ln' in other languages (see previous section).

We have no particular preference for the names of the formal parameters of the standard basic functions. These have been chosen to be as simple as possible, except for the names of the second parameters of the more general functions. The name E for the mathematical constant e was rejected as a potential source of much confusion.

We considered, but rejected, the possibility of letting the package consist of several subpackages, for e.g. transcendental, circular, inverse circular and hyperbolic functions, all packages possibly with their own subtypes for arguments and their own exceptions. In such a structure the ease of use would be completely lost.

For some of the functions (e.g. LOG, SIN) we propose more general declarations, thus combining related functions which differ only by a scale factor. The user of the traditional form of the basic functions need not be troubled by this, as calls with one parameter are allowed. The most natural alternative, of providing the more general versions (e.g. SIN for arbitrary or specific other periods) in a dependent special purpose package, would be very wasteful, since the number of bodies needed would be doubled. Moreover, we have shown (in Appendix C of Symm et al., 1984) that portable general implementations can be readily designed. The Ada Style alternative (of providing generic functions, such as a SIN with a generic in parameter for the period) differs too much from the way basic functions are usually available, and is actually not needed.

6. CONCLUSIONS

We have already mentioned our recommendation that a standard instance of the proposed package should be available at every installation, with the common floating-point type as the generic actual parameter. Also, one might substitute for this instance an equivalent non-generic version. We assert that portable and efficient implementations of the package constituents can be written in Ada and that the proposal presented here will then supply an essential tool for programming numerical computations in Ada.

REFERENCES

ANSI/MIL-STD 1815 A. Reference manual for the Ada programming language, January 1983.

Cody, W.J. Floating-point parameters, models and standards. The relationship between numerical computation and programming languages, edited by J.K. Reid, North Holland, Amsterdam, 1982, 51-64.

Cody, W.J. and Waite, W. Software manual for the elementary functions, Prentice Hall, New Jersey, 1980.

Ford, B. Parametrization of the environment for transportable numerical software. ACM Trans. Math. Softw., 1978, 4, 100-103.

Nissen, J.C.D. and Wallis, P.J.L., eds. Portability and style in Ada, Cambridge University Press, Cambridge, 1984.

Rice, J.R. Remarks on software components and packages in Ada. ACM SIGSOFT Software Engineering Notes, 1983, 8(2), 9-10.

Symm, G.T., Wichmann, B.A., Kok, J. and Winter, D.T. Guidelines for the design of large modular scientific libraries in Ada, NPL Report DITC 37/84 and CWI Report NM-N8401, March 1984.

Wallis, P.J.L. Ada floating-point arithmetic as a basis for portable numerical software. Proceedings of the 6th Symposium on Computer Arithmetic, IEEE, Computer Society Press, Silver Spring, 1983, 79-81.

Whitaker, W.A. and Eicholtz, T.C. An Ada implementation of the Cody-Waite "Software manual for the elementary functions", US Air Force, 1982.