Elementary Functions Packages for Ada

by
Robert F. Mathis
9712 Ceralene Drive, Fairfax, VA 22032-1704
(703)425-5923, Mathis@C.ISI.EDU

This paper incorporates some of the discussions by the ACM/SIGAda Working Group on Ada Numerics, the ISO/TC97/SC22/WG9 Rapporteur Group on an elementary functions package, and the Ada-Europe Ada Numerics Working Group. This is not a report from any one of those groups nor does it attempt to reflect all the discussions or potential resolutions of the issues. This paper is the author's attempt to provide information on the current thinking of these groups about a standard specification for an elementary functions package for Ada so that a broader group can assess the utility and trade-offs involved in such standard specifications and libraries.

The following people have been involved in developing and influencing the ideas presented here: Jim Cody, Paul Cohen, Sandy Cohen, Ken Dritz, Brian Ford, Graham Hodgson, Jan Kok, Gil Myers, Brian Smith, Jon Squire, and Bill Whitaker. Any confusion or inaccuracies are the author's. These people represent a combination of interests in the Ada language ranging from numerical analysis to embedded application development.

The SIGAda Numerics Working Group (SIGAda NUMWG) has met at the SIGAda meetings in Pittsburgh, PA (July, 1986), Charleston, WV (November, 1986), Fort Lauderdale, FL (January, 1987), and Seattle, WA (August, 1987), several times at Argonne National Laboratory near Chicago, IL and at various locations in the Washington, DC area. The Numerics Working Group of Ada-Europe has met many times on this and related topics.

A number of proposals for elementary functions in Ada ([FIRTH 1982], [WHITAKER & EICHOLTZ 1982], [WITTE 1983], [SYMM, WICHMANN, KOK & WINTER 1984], [KOK & SYMM 1984], and [KOK 1987]) have been studied by the SIGAda NUMWG. These were considered along with recent work in numerical analysis about the calculation of the elementary functions and about the properties of floating point arithmetic. Various aspects of Ada's real arithmetic, type mechanism, generic packages, and library structures have also been considered as they relate to the specification and implementations of a potentially widely used package.

The proposed package specification is given at the very end of the paper as a convenient reference point.

1 Background

From the very beginning of the development of the requirements that lead to Ada, the HOLWG and other advisors rationalized that such standard packages would be early natural developments. Thus elementary functions were left out of the requirements for the language. There was also some hope that by motivating people to develop such standard elementary packages in Ada rather than using packages that had been developed from FORTRAN, the perspective of the programmer would be altered. The SIGAda NUMWG set itself the two challenges of developing an Ada-oriented specification and being at least a step better than most previous approaches to implementing the elementary functions. This has lead to an attempt to specify more precisely the results and behavior of the routines in an Ada-oriented machine-independent way.

Ada has a number of features which make it well suited for numeric software. The most important of these features is user declared precision in floating point types. The FORTRAN 77 concepts of REAL and DOUBLE PRECISION correspond roughly to the Ada predefined types of FLOAT and LONG_FLOAT. In both languages, their direct use leads to a number of problems in transporting numeric software; but Ada's user declared floating point types have predictable properties on different implementations. It was the intention of the language designers that user declared types be efficiently handled by the underlying hardware. This means an Ada program relying on the properties of these user declared floating point types should be both transportable and efficient.

The set of elementary mathematical functions seem a natural candidate for a standard library for Ada. These functions exist in other programming languages and are widely used. They are relatively well understood in terms of functionality. That is not to say, however, that the construction of such a library is well understood. Many issues are involved in specifying a math library. Which functions to provide, what arguments those functions should use, in what ways the functions can be generalized or customized, how to fit these general functions with the type and generic mechanisms of Ada, the level of integration of these routines into the language, and how various errors are to be handled — these are among the choices facing the designer.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1987 ACM 0-89791-243-8 / 87 / 0012 / 0095 $1.50

The Numerics Working Groups have thought of the elementary functions as being in the support base of the transportability of numeric programs. The specification of the package MATHEMATICAL_CONSTANTS is a good example. The values of the constants pi and e are known to more digits than any implementation of floating point uses. The only way to express them in Ada is by explicitly listing the digits. They need to be given to at least as many digits as SYSTEM.MAX_DIGITS, but this varies. The suggestion for MATHEMATICAL_CONSTANTS is therefore minimal.

Programs using the packages suggested here need to be transportable in the usual Ada sense, but the bodies of the routines implementing the elementary functions need not themselves be transportable. Our goal is to have a small set of flexible specifications which make simple things straightforward for casual users and provide a framework within which more sophisticated users can develop specialized variants.

The work of the group has been in three main areas — functionality, packaging, and accuracy. Functionality concerns which functions to provide and for what units of measuring angles. In the packaging area various organizations for the library and its use were considered including how to handle generics, types (for precisions, ranges, and names), exceptions, constants, user implementations, and user enclosing shells. In the area of accuracy, the group has investigated what can be expected from the best available algorithms and how that can be specified in machine independent Ada terminology. Test cases and validation procedures for implementations have also been considered.

2 Choice of Functions

One of the first decisions in the design of an elementary functions package is which functions should be provided and the units or bases on which they should work. The SIGAda NUMWG decided to concentrate initially on the traditional elementary functions — square root, logarithm, exponential, power, basic trigonometric circular functions (sine, cosine, tangent, and cotangent), their inverses (or "arc" functions), the corresponding hyperbolic functions, and their inverses. These are basically the same functions suggested in various European proposals ([SYMM, WICHMANN, KOK & WINTER 1984], [KOK & SYMM 1984], and [KOK 1987]). This resulted in the twenty basic functions which are shown in the specification of the package.

Left for later were other standard mathematical and numerical analysis functions — construction utilities, random number generators, complex arithmetic, elementary functions for complex numbers, matrix manipulation, and other topics. We felt that if the general approach for the real-valued elementary functions could be agreed on, then these others would follow. We also decided to work initially with only floating point types.

After deciding on the functions to include, names had to be chosen. Over the years, various abbreviations have been used for the names of these functions and even the FORTRAN names for these functions have not remained fixed. Seventeen circular functions and more than fifty-five possible names for them were considered. The chosen names are conservative and well known to people who have previously used mathematical functions. We decided not to use underscores ("_") in the names. But while we chose to retain the full prefix "arc" rather than just "a" as in some other languages, we elected to designate the hyperbolic functions traditionally, i.e., with the suffix "h".

The names for the arguments were also chosen conservatively. In environments where named parameter associations are used, it would be appropriate to consider renaming. In one instance the names were chosen to have a particular meaning. The tangent of an angle can be thought of as the slope (Y/X) of a line through the origin making that angle with the X-axis. If X=1 then the value of the tangent is the corresponding value of Y. The arc-tangent function takes the cartesian (X, Y) coordinates of a point on the line as two arguments, or the slope of the line as a single argument, and returns the corresponding angle. In Ada this can be accomplished in a single function specification by giving a default value of 1.0 for the argument corresponding to X. To be able to call this function with positional notation and take advantage of the default value, Y has to be the first argument. Since the cotangent corresponds to X/Y, the X comes first in that case and Y has the default value of 1.0.

We concluded that the other trigonometric functions (secant, cosecant, haversine, and so forth) need not be included since they are traditionally defined in terms of the more common functions. The proposed list was chosen as a compromise similar to the ones taken by other programming languages. The working group considers this a minimal package.

In all these cases about choosing names it was felt that the choices made would have been high on anyone's list of options (even if not necessarily first) and that they would therefore be relatively obvious guesses by anybody looking for routines with these functionalities. Ada provides for renaming which should make it possible for projects and individuals to adapt these packages to their preferences.

3 Units, Bases, Domains and Ranges

The next question concerns the choice of units for the functions (radians or degrees for the trigonometric functions, base ten or natural logarithms, for example). Radian measure is so commonly used in mathematics and numerical programming, it was decided that this should be the simplest to use. Early in their discussions the Numerics Working Groups of both SIGAda and Ada-Europe decided that it should also be possible to call the trigonometric functions for different angular measures. This was expressed by a second parameter, CYCLE, which gave the measure of a full circle in the units being used for the first parameter. For example,

```
SIN ( X, CYCLE => TWO_PI )
```

for angles measured in radians and

```
SIN ( X, CYCLE => 360.0 )
```

for angles measured in degrees. One way of thinking about the measure of an angle is to consider it as some fraction of a full circle; the ratio of the first parameter to the second gives this fractional measure.

We considered a number of different units for measuring angles and cycles (360.0 degrees, 400.0 grads, 6400.0 mils,

1.0 bams, and so on). Although this short list might seem to cover all possibilities, it does not. There always seem to be cycles that are not included in a short list and so we decided that separate functions for different units was not a reasonable approach.

If all of the potentially useful cycles (two pi radians, 360.0 degrees, etc.) were equally likely to be encountered in practice, we might be content to put them on an equal footing by endowing the trigonometric functions with two required parameters (X and CYCLE). But since radian measure (cycle of two pi radians) occurs far more frequently than other cycles, we decided to make the use of radians more convenient by making the CYCLE parameter optional, with the understanding that when it is omitted a cycle of two pi radians is implied.

There are two good ways of presenting the Ada user with the appearance of a function that can be called with one argument or two, the behavior in the former case being as if a particular value of the second argument were implicitly provided. The most obvious of these two ways, illustrated with the sine function, is to use the Ada mechanism of a default value for a parameter:

```
function SIN ( X : FLOAT_TYPE;
      CYCLE : FLOAT_TYPE := TWO_PI )
   return FLOAT_TYPE ;
```

This method was used in [SYMM, WICHMANN, KOK & WINTER 1984]. Somewhat less obvious is to use subprogram overloading to declare two different, but similarly named, functions

```
function SIN ( X : FLOAT_TYPE )
   return FLOAT_TYPE;
function SIN ( X, CYCLE : FLOAT_TYPE )
   return FLOAT_TYPE;
```

with the implied cycle of two pi radians built into the body of the one-argument function. After considerable debate, The SIGAda NUMWG chose this latter method, which is illustrated in the proposed package specification. It is the method employed in [KOK 1987]. (The BASE parameter of the LOG function has a similar history.)

Why this choice? It has to do, essentially, with the recognition that the one-argument forms of the trigonometric functions, with a built-in cycle of two pi radians, have qualitatively different properties from the two-argument forms (with a user-supplied cycle), and that to extract the maximum practical accuracy from both sets of functions requires different argument-reduction techniques. The resulting two sets of implementations are accommodated by the chosen specification method with the fewest risks and compromises.

Note that all of the usual user-supplied alternate cycles are exactly representable in typical floating point systems. It is therefore possible to perform exact argument reduction for any of these user-supplied cycles, beginning with the calculation of the exact remainder of X and CYCLE. The remainder will range from zero up to (but not including) CYCLE. At the cost of one roundoff error in the entire argument reduction step, this can be reduced to the range 0.0 to 1.0 by dividing by cycle, and transformed without further loss of accuracy to the appropriate principal domain for each function's reduce argument. No domain restrictions are necessary.

On the other hand, two pi is not exactly representable. To proceed as in the user-supplied cycle case with the nearest representable value to two pi would introduce unacceptable error in the argument reduction step for argument only a few cycles away from zero. Better techniques are known ([CODY & WAITE 1980] and [MILLER 1984]) for this unique problem, having the benefits of calculating in higher precision without actually doing so (higher precision than that used for FLOAT_TYPE might not be available). These techniques have the property that as much precision as desired may be achieved in the argument reduction step, but at a rapidly escalating cost. Since the precision needed to accurately reduce the argument increases with the magnitude of the argument, practicality ultimately dictates a limit on the domain of the function if meaningful results are to be obtained. Thus, the one-argument and two-argument forms of the functions have very different usable domains and somewhat different accuracy requirements (i.e., achievable accuracy), and these are most appropriately attached to separate specifications for the two versions of each trigonometric function. (Another difference between the one-argument and two-argument cases is that, while the two-argument case accommodates a variety of cycles by dividing by CYCLE in the argument reduction step, the one-argument case has a single value of the cycle to handle and can therefore omit the canonicalization of dividing by the cycle. The different range of reduced argument values that results in this case calls for a different approximation method with coefficients tailored to this case.)

There has been considerable investigation into the calculation of the values of these elementary functions. There have evolved two different approaches — one for angles measured in terms of radians and another for angles measured as rational fractions of a circle. The one-argument function should work with the best of the radian oriented techniques and the two-argument function with the best of the rational fraction approaches.

The committee realized that the differences between the one-argument and two-argument forms described above could be accommodated in a single function body, thus permitting the other choice of specification method to be made. We decided against this, however, because it would have necessitated certain compromises (not detailed here) and entailed certain risks (elaborated on below), the latter being the more serious. Since Ada gives a function body no way of distinguishing between a call in which the default value of an optional argument was used and a call in which the same value was explicitly passed, the specialized behavior desired for the one-argument case would have to be inferred whenever the function body detects the CYCLE parameter to have the value given by the default expression in the specification. There is a slight risk that the high-precision named number representing two pi

```
TWO_PI : constant :=
   2.0 * MATHEMATICAL_CONSTANTS.PI ;
```

will be converted to the precision of FLOAT_TYPE differently in the two contexts where it is used (the language

only requires that the results of both conversions lie in the same "safe interval," which may contain several machine numbers). Even worse, the user may not understand the default mechanism and optional arguments and may call a trigonometric function with an explicit CYCLE which is some other approximation of two pi than the one given in the default expression. In either of these cases, the function body will likely fail to recognize the call as one corresponding to the default cycle and the desired specialized behavior will not be provided.

Some consideration was also given to requiring only one-argument versions of the trigonometric functions. Providing an exact remainder function ("rem") were made available to users, alternate (non-radian) measures could still be obtained (at a cost of a single roundoff error during argument reduction). The user would essentially have to perform the argument reduction before calling the trigonometric function, as in (for degrees)

```
RESULT := SIN ( ( X rem 360.0 ) *
   MATHEMATICAL_CONSTANTS.PI/180.0 ) ;
```

Since this is burdensome, prone to error, hard to understand, and no better than the primary choice, it was rejected. It does, however, illustrate a relevant general principle — users can usually do better argument reduction outside of the call to the standard function because of special knowledge about the problem situation.

In the arc-tangent and arc-cotangent functions the second parameter is optional and has an explicit default value of 1.0. In both cases this second parameter corresponds to the denominator in a fraction. The third parameter, CYCLE, is optional as discussed above, a value needing to be supplied in the call only for cycles of other than two pi radians. To specify another value for CYCLE and to take advantage of the default value for the second parameter requires explicit use of a named parameter association, for example

```
ARCTAN ( Y, CYCLE => 360.0 )
```

Named parameter associations are appropriate for the other function calls where alternate values are being specified. In particular, if a program segment uses two different types of angular measure, the code will be clarified by making the different cycles explicit.

4 Packaging and Generics

After the question of what functionality to provide comes the question of how to make that available in the language. The SIGAda NUMWG considered a number of possibilities, but finally settled on the specification given at the end of this paper. This is a relatively cohesive, single-level, generic package.

There are a number of roughly equivalent ways to write some packages. The following examples show how a simple package containing only a sine function might be done. The packages are not explicitly generic to make the examples a little cleaner. The conclusion is that the package specifications should be written to provide the capabilities needed by the programmer. Actual implementations may be provided through other means. It is also possible for a programmer to use a general package in a way more suitable to his own application.

In the following examples it is assumed that the type names are visible. These packages would normally be written as generic over the types involved and then there would have to be instantiations of the generics being used.

Assume that we wanted to implement either a single or double argument style package in terms of the other. For the purposes of this example, assume the following simplified package specifications:

```
package SINGLE_PARM is
   function SIN ( X: FLOAT_TYPE ) return
FLOAT_TYPE;
end SINGLE_PARM;

package DOUBLE_PARM is
   function SIN ( X, CYCLE: FLOAT_TYPE ) return
FLOAT_TYPE;
end DOUBLE_PARM;
```

Then the body of either of these packages could be implemented in terms of the other one, but of course not simultaneously, as follows:

```
-- DOUBLE_PARM in terms of SINGLE_PARM
with SINGLE_PARM,
      MATHEMATICAL_CONSTANTS;
package body DOUBLE_PARM is
   function SIN ( X, CYCLE: FLOAT_TYPE )
      return FLOAT_TYPE is
   begin
      -- there are better argument reduction and
      -- conversion methods,
      -- but this illustrates the relationship
      return SINGLE_PARM.SIN (( X / CYCLE ) *
         2.0*MATHEMATICAL_CONSTANTS.PI );
   end SIN;
end DOUBLE_PARM;

-- SINGLE_PARM in terms of DOUBLE_PARM
with DOUBLE_PARM,
      MATHEMATICAL_CONSTANTS;
package body SINGLE_PARM is
   function SIN ( X: FLOAT_TYPE )
      return FLOAT_TYPE is
   begin
      return DOUBLE_PARM.SIN ( X, CYCLE =>
         2.0*MATHEMATICAL_CONSTANTS.PI );
   end SIN;
end SINGLE_PARM;
```

In much the same way, it is also possible to write packages with two sine functions, or which have different types for the arguments and return values, or which have default cycles for units other than radians, or which have special argument reduction, or which have various forms of error handling. This is the way the standard functions will likely be used — as a conceptual basis for packages actually used. This standard specification can be implemented in terms of vendor supplied routines or the vender supplied routines can be thought of as implemented in terms of this standard package.

5 Exceptions

There was general agreement to use exceptions rather that TEXT_IO messages to indicate problems, as this gives the

user the most flexibility (albeit at the risk of failing to handle the exception). It appears most implementations provide an indication to the user of the line where an unhandled exception was raised. Many installations further provide a traceback of the sequence of callers and the line that did the calling. This means that most implementations will provide textual output automatically for users who do not include exception handlers in their code. It is also possible for users or implementations to provide versions of these elementary functions that produce textual error messages. Functions with textual error messages can be derived from or built on the ones described here, but not the other way around.

Besides the existing predefined exceptions, it was decided that all other exceptions arising during the execution of these routines were ultimately due to an improper or out of domain argument. There was some discussion about the possibility of another exception for loss of significance internally in the calculation of the value of the function. There were numerous examples of where it would be difficult to distinguish between these two types of problems. Hence there needs to be only one exception — ARGUMENT_ERROR. This exception is defined in a separate package and then used in the generic elementary functions package.

The generic package for the elementary mathematical functions uses this exception ARGUMENT_ERROR which is defined in the separate package through a **renames** clause. If there should be two instantiations of the generic package named in **use** clauses, the exception name will be hidden rather than overloaded. The single exception which is raised by either package can still be referenced by its full name, MATHEMATICAL_EXCEPTIONS.ARGUMENT_ERROR. This requires that the exceptions package be named in a **with** clause by any program segment that has an exception handler for ARGUMENT_ERROR. This technique is similar to what is used in IO_EXCEPTIONS and is better than having different names for the exceptions in each package and for each instantiation.

6 Accuracy Specifications and Testing

Another major emphasis in our discussions has been on validation of implementations against accuracy requirements rather than imposing a standard implementation. We felt this was in keeping with the Ada philosophy. These accuracy specifications are in Ada terms rather than machine terms. We are also working on a set of test routines. These accuracy specifications and other details about the expected performance of the implementing routines will be described in a separate paper.

As an indication of some of the issues involved, consider the computation of EXP(X) and the desire to compute the value for very negative X. Mathematically exp(x) can never return zero, but depending on how underflow is handled there may be a value for which EXP(X) returns a zero (or the function may be flat after some point). A symmetric limit expressed in terms of the Ada model may be too conservative and a useful bound may not be expressible in terms of the Ada model.

As another example, consider the sine function which is very smooth. Most computation methods have good error characteristics except near multiples of two pi radians. [MILLER 1984] How should the error bounds on the sine be expressed so that they are useful to the majority of programmers while at the same time being accurate and tight enough to require good implementations?

7 Conclusions

The SIGAda NUMWG, the ISO/TC97/SC22/WG9 Rapporteur Group, and the Ada-Europe Ada Numerics Working Group will be presenting these suggested specifications to the appropriate standards groups and implementors of commercial function packages. We have discussed ways to implement and use these specifications. We hope that our goals of Ada-oriented, accurate, and easy to use specifications have been met. We also hope that these will lead naturally to standard specifications for other packages and libraries of mathematical routines. We welcome further comments from interested users.

8 References

[CODY & WAITE 1980] William J. Cody, Jr., and William Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

[FIRTH 1982] Robert Firth, *Preliminary Draft Specification of a Basic Mathematical Library for the High Order Programming Language Ada*, Royal Military College of Science, Shrivenham, Swindon, Wiltshire, England, March, 1982.

[FORD, KOK & ROGERS 1986] Brian Ford, Jan Kok, and Mike W. Rogers, *Scientific Ada*, Cambridge University Press, Cambridge, England, 1986.

[KOK 1987] Jan Kok, *Design and Implementation of Elementary Functions in Ada*, Report NM-R8710, Centrum voor Wiskunde en Informatica, Amsterdam, April, 1987.

[KOK & SYMM 1984] Jan Kok and George T. Symm, "A Proposal for Standard Basic Functions in Ada," *Ada Letters*, Vol. IV, No. 3, Nov-Dec, 1984, pp. iv.3-44 to iv.3-52.

[MILLER 1984] Webb Miller, *The Engineering of Numerical Software*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[SYMM, WICHMANN, KOK & WINTER 1984] George T. Symm, Brian A. Wichmann, Jan Kok, and Dik T. Winter, *Guidelines for the Design of Large Modular Scientific Libraries in Ada*, Final Report for the Commission of European Communities, Note NM-N8401, Centre for Mathematics and Computer Science, Amsterdam, Netherlands, March, 1984 (also available as NPL Report DITC 37/84, National Physical Laboratory, Teddington, Middlesex, UK) (edited and reprinted as Chapter 10, pp. 209-319 in [FORD, KOK & ROGERS 1986]).

[WHITAKER & EICHOLTZ 1982] William A. Whitaker and T. C. Eicholtz, *An Ada Implementation of the Cody-Waite "Software Manual for the Elementary Functions"*, US Air Force Armament Laboratory, Eglin AFB, FL, 1982.

[WITTE 1983] Bruno Witte, "General Requirements for an Elementary Math Functions Library," *Report on Ada Program Libraries Workshop*, Naval Postgraduate School, Monterey, CA, November 1-3, 1983; Edited by Joseph A. Goguen and Karl N. Levitt, SRI International, Menlo Park, CA, 1983, pp. 100.i-xii.

-- Proposed Package Specifications

```
package MATHEMATICAL_EXCEPTIONS is
   ARGUMENT_ERROR : exception ;
end MATHEMATICAL_EXCEPTIONS;

package MATHEMATICAL_CONSTANTS is
PI              : constant     := 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
NATURAL_E       : constant     := 2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
   -- to be specified to more digits than SYSTEM.MAX_DIGITS
   -- this package may contain additional definitions
end MATHEMATICAL_CONSTANTS ;

with MATHEMATICAL_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <> ;
package GENERIC_ELEMENTARY_FUNCTIONS is

   function SQRT( X: FLOAT_TYPE )         return FLOAT_TYPE;
   function LOG  ( X: FLOAT_TYPE )        return FLOAT_TYPE;
   function LOG  ( X, BASE: FLOAT_TYPE ) return FLOAT_TYPE;
   function EXP  ( X: FLOAT_TYPE )        return FLOAT_TYPE;
   function "**" ( X, Y: FLOAT_TYPE )     return FLOAT_TYPE;

   function SIN  ( X: FLOAT_TYPE )           return FLOAT_TYPE;
   function SIN  ( X, CYCLE: FLOAT_TYPE )    return FLOAT_TYPE;
   function COS  ( X: FLOAT_TYPE )           return FLOAT_TYPE;
   function COS  ( X, CYCLE: FLOAT_TYPE )    return FLOAT_TYPE;
   function TAN  ( X: FLOAT_TYPE )           return FLOAT_TYPE;
   function TAN  ( X, CYCLE: FLOAT_TYPE )    return FLOAT_TYPE;
   function COT  ( X: FLOAT_TYPE )           return FLOAT_TYPE;
   function COT  ( X, CYCLE: FLOAT_TYPE )    return FLOAT_TYPE;

   function ARCSIN ( X: FLOAT_TYPE )           return FLOAT_TYPE;
   function ARCSIN ( X, CYCLE: FLOAT_TYPE )    return FLOAT_TYPE;
   function ARCCOS( X: FLOAT_TYPE )            return FLOAT_TYPE;
   function ARCCOS( X, CYCLE: FLOAT_TYPE )     return FLOAT_TYPE;
   function ARCTAN( Y: FLOAT_TYPE; X: FLOAT_TYPE := 1.0 )                          return FLOAT_TYPE;
   function ARCTAN( Y: FLOAT_TYPE; X: FLOAT_TYPE := 1.0; CYCLE: FLOAT_TYPE )      return FLOAT_TYPE;
   function ARCCOT ( X: FLOAT_TYPE; Y: FLOAT_TYPE := 1.0 )                         return FLOAT_TYPE;
   function ARCCOT ( X: FLOAT_TYPE; Y: FLOAT_TYPE := 1.0; CYCLE: FLOAT_TYPE )     return FLOAT_TYPE;

   function SINH   ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function COSH   ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function TANH   ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function COTH   ( X: FLOAT_TYPE ) return FLOAT_TYPE;

   function ARCSINH  ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function ARCCOSH  ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function ARCTANH  ( X: FLOAT_TYPE ) return FLOAT_TYPE;
   function ARCCOTH  ( X: FLOAT_TYPE ) return FLOAT_TYPE;

   ARGUMENT_ERROR : exception renames MATHEMATICAL_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```