Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M. Bergman

Implementation of elementary functions in Ada

Department of Numerical Mathematics Report NM-R8709 April

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Implementation of Elementary Functions in Ada

M. Bergman
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Unlike many other languages, Ada does not define elementary mathematical functions. Therefore a package of basic mathematical functions has been designed and implemented, which meets requirements like portability, general usefulness and efficiency. For these purposes Ada offers a number of interesting and useful features, which will be discussed in brief. Further, a detailed description is given of the way the elementary functions have been implemented.

1980 Mathematics Subject Classification: 69D49, 65-04.
Key Words and Phrases: Ada, elementary mathematical functions, portability, scientific libraries.
Note: The work described in this report was part of the MAP 750 project 'Pilot Implementations of Basic Modules for Large Portable Numerical Libraries in Ada' part-funded by the Commission of the European Communities.

1. INTRODUCTION

Although Ada has been designed in the first place for embedded real-time systems, it offers such important features that the language can also be widely used in many other areas. One of these areas is the area of the scientific computations.
For many scientific computations it is necessary to have at one's disposal a collection of elementary mathematical functions. However, the basic mathematical functions like *log*, *exp* and *sin*, are not included in the definition of Ada.
Consequently the user is totally dependent of the elementary functions provided by the manufacturers of compilers, if they provide any, or is forced to make the required functions himself. In practice this will result in sets of elementary functions, which differ completely from system to system. It is almost superfluous to say that much of this software is not transportable to other machines without adapting large parts of it.

Building on a number of recommendations given in the *Guidelines for the design of large modular scientific libraries in Ada (Symm et al.*, 1984) the project 'Pilot Implementations of Basic Modules for Large Portable Numerical Libraries in Ada' lays the foundation for the design and implementation of portable and efficient large-scale numerical libraries in Ada. These libraries can be applied for scientific purposes as well as for real-time computer systems. Particularly a number of directives is given with relation to the design and implementation of a portable library of elementary functions. For this purpose much attention is paid to subjects like general usefulness, ease of use, efficiency in use and portability, without obstructing the execution efficiency too much.

In the next chapter some Ada-concepts, mainly concerning the design of a library of elementary functions, will be discussed. For a detailed description we refer to the above mentioned *Guidelines*. The subsequent

chapters contain the following subjects :

chapter 3 deals with the use of attributes,
chapter 4 deals with error handling,
chapter 5 deals with function descriptions and
chapter 6 contains some conclusions.

2. USEFUL ADA-CONCEPTS FOR DESIGNING A LIBRARY

2.1. packages

An important Ada-mechanism that does not exist in block-structured languages like Algol and Pascal, is the *package*. The package offers the opportunity to group sets of declarations, which logically should be joined together, to larger units.
A package consists of two parts : the package specification and the package body. The package specification contains the information which is visible for the user. It may include things like type declarations, constants, subroutines and other package specifications, which are all directly accessible by the user.
On the other hand, the package body contains the implementation details which can be hidden for the user. Hence it is not possible for the user to use data, subprograms or other packages which are local with respect to the package body.

With the help of the package construct we can design a package consisting of the elementary functions in an elegant way. Such a package could look like :

```
-- the package specification :
package MATH_FUNCTIONS is
   function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE;
                                  .
                                  .
                                  .
   function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE;
end MATH_FUNCTIONS;

-- the package body :
package body MATH_FUNCTIONS is
   -- local declarations
   function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE is
                                  ...
   end SQRT;
                                  .
                                  .
                                  .
   function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE is
                                  ...
   end ARCCOTH;
end MATH_FUNCTIONS;
```

The above assumes that the floating-point type FLOAT_TYPE is known to the package.

2.2. generics

The package of elementary functions in this form is not very general and lacks flexibility. At least one may expect that for each predefined floating-point type a package of basic functions is provided. By defining a package as we did before, we have to write a new specification and body completely for each built-in floating-point type. This is cumbersome and brings along much superflous work as the algorithms, used by the functions in the package, in general are independent of the precision of the floating-point type. The points of difference in the implementations of the package bodies of the various floating-point types are in fact of minor importance.
Besides it is fairly complicated for a user to make himself a package of elementary functions for a self-defined precision. If he has access to the sources of existing packages, which is mostly undesirable, he can copy them and make the required adaptions. Otherwise he would have to make a package by using a package for a predefined type and use explicit conversions for input parameters and results.

To avoid these problems, we can use the Ada *generics mechanism*. In Ada it is possible to define a template, with or without parameters, of a package or subprogram. After this we can make as many instantiations as wanted by calling the template with appropriate parameters. As formal parameters not only objects of the usual types, built-in or user-defined, can be used, but also types themselves and subprograms.
When a formal generic parameter is specified to be a floating-point type, it is possible to make instances for each desired floating-point type without changes. The specification of such a package could look like :

```
-- specification of the generic package :
generic
        type FLOAT_TYPE is digits <>;
package GENERIC_MATH_FUNCTIONS is
        function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE;
                .
                .
                .
        function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE;
end GENERIC_MATH_FUNCTIONS;
```

'**type** FLOAT_TYPE **is digits** <>;' indicates that the generic formal parameter FLOAT_TYPE consists of a floating-point type. Hence, when instantiating GENERIC_MATH_FUNCTIONS a floating-point type should be supplied as an actual parameter.
The package body of GENERIC_MATH_FUNCTIONS is the same as the body of MATH_FUNCTIONS (except for the package name).

An instantiation of this package could be given as follows :

```
package FLOAT_MATH_FUNCTIONS is new GENERIC_MATH_FUNCTIONS (FLOAT);
```

for a predefined type, and

```
type DIGITS_3 is digits 3;
package DIG_3_MATH_FUNCTIONS is new GENERIC_MATH_FUNCTIONS (DIGITS_3);
```

for a user-defined type DIGITS_3 which has three significant decimal digits.

2.3. exceptions

During execution of a program it may happen that the normal course of the execution is disturbed by the occurrence of some unusual situation. For example an attempt to divide by zero, the arising of overflow, or a function argument not belonging to the domain of the function. In all these cases the normal program execution should be stopped and some 'repairing' actions should be carried out to handle the unusual situation.
To handle these situations the Ada *exception mechanism* can be used. When, during execution of a program, an anomaly occurs, this situation is detected and the normal execution is stopped ("raise an exception"). Next, some actions can be performed to catch this situation ("handle the exception"). However it is not necessary to handle an exception on the same level as where it occurs; this may also be done on another level.

Beside a number of predefined exceptions like NUMERIC_ERROR (for the detection of, for example, overflow) and CONSTRAINT_ERROR (for the detection of, for example, an attempt to violate an index constraint), it is possible for a user to define his own exceptions. Unlike the predefined exceptions, which can be raised automatically, the user-defined exceptions can only be raised explicitly.

Using the package, generic, and exception mechanism, a preliminary version of the specification of the generic package of elementary functions, as proposed by the *Guidelines*, looks like :

```
generic
   type FLOAT_TYPE 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 : FLOAT_TYPE) return FLOAT_TYPE;
   function LOG(X: FLOAT_TYPE;
         BASE : FLOAT_TYPE := EXP_1) return FLOAT_TYPE;
   function EXP(X: FLOAT_TYPE;
         BASE : FLOAT_TYPE := EXP_1) return FLOAT_TYPE;
   function SIN(X: FLOAT_TYPE;
         CYCLE : FLOAT_TYPE := 2.0 * PI) return FLOAT_TYPE;
   function COS(X: FLOAT_TYPE;
         CYCLE : FLOAT_TYPE := 2.0 * PI) return FLOAT_TYPE;
   function TAN(X: FLOAT_TYPE;
         CYCLE : FLOAT_TYPE := 2.0 * PI) return FLOAT_TYPE;
   function COT(X: FLOAT_TYPE;
         CYCLE : FLOAT_TYPE := 2.0 * PI) return FLOAT_TYPE;
   function ARCSIN(X : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCCOS(X : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCTAN(X : FLOAT_TYPE;
         Y : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function ARCCOT(X : FLOAT_TYPE;
         Y : FLOAT_TYPE := 1.0) 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;
-------------------------------------------------------------------
-- Declare exception.
-------------------------------------------------------------------
   ARGUMENT_ERROR : exception;
-------------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
```

In the future this specification might be adapted slightly. It is probable that each of the trigonometrical functions will be split up in a function with one parameter and a function with two parameters, both without default parameters.

3. ATTRIBUTES

An important resource for approximating many elementary functions is the evaluation by means of polynomials. Such approximations consist of truncated powerseries, of which the contribution by the higher order terms decreases rapidly. The number of terms, used by a polynomial, depends on the desired precision.

In the *Guidelines* a proposal is made to design the algorithms for the basic functions in such a way that an accuracy of at most thirty-five significant digits can be attained. However, the software which would proceed from just considering this maximum precision, would be very inefficient and would mean a waste of computer time. Even the function values for smaller precisions would be evaluated with the maximum accuracy by using a needlessly large number of terms for the approximating polynomials.

To avoid this one may consider to design different algorithms for the different precisions. A more elegant solution consists of using *attributes* of a floating-point type. In Ada attributes are predefined functions which can be helpful for obtaining some information about properties of types. Which attributes a certain type has, depends on the kind of type.

To the attributes of a floating-point type belong, amongst others, the DIGITS-attribute, which denotes the number of decimal digits in the declaration of a floating-point type, and the MANTISSA-attribute, which denotes the number of bits in the mantissa of the representation of a floating-point type.

For the implementation of the elementary functions, it is now possible to use one algorithm per function if this is desired. When an approximation by means of a polynomial has to be performed, the number of terms to achieve the desired precision may depend on the generic (floating-point type) parameter. In this way we can distinguish a number of branches each delivering a value by evaluating a polynomial with a specific number of terms. Each polynomial evaluation uses no more terms than strictly necessary to acquire an efficient and accurate approximation.

Using attributes many elementary functions will get the following form :

```
function BASIC_FUN (X : FLOAT_TYPE; ... ) return FLOAT_TYPE is
  -- local declarations
begin
  -- statements
  case FLOAT_TYPE'DIGITS is
    when  1 .. 3  => -- simple approximation
    when  4 .. 6  => -- less simple approximation
```

```
        when 33 .. 35 => -- approximation for maximum accuracy
        when others  => null;
      end case;
      -- statements
    end BASIC_FUN;
```

The bounds, which determine what branch has to be taken, differ from function to function. On attempting to get a polynomial evaluation of more than thirtyfive digits, the others-branch will be chosen and nothing will be done. This will lead to either a function value, which is completely wrong, or to the raising of some exception.
A good optimising compiler might take care of leaving out the dead branches when the package is instantiated.

4. ERROR HANDLING

Many elementary functions will be constructed on the basis of a fixed procedure. This procedure can roughly be summarized as follows :

* a check for correctness of the function argument(s)

* the reduction of the argument to a related argument in a small interval

* the evaluation of the function value for this reduced argument

* the reconstruction of the function value for the original argument

Almost every function starts with a test to see if the function arguments are in the specified domain of a function. For example the SQRT starts with a check to see if the argument is non-negative, and the SIN checks if it has a non-zero cycle.
When a wrong argument is met, the exception ARGUMENT_ERROR is explicitly raised. This is in complete agreement with the *Guidelines* which state that an exception should be raised by a function only when the final result is exceptional.
This implies that in the case of overflow an exception is raised only when the final function result also overflows. In most cases such an exception can be propagated, but it is also permitted for a basic function to handle it or to raise another appropriate exception.
On the other hand, care should be taken to avoid situations in which an exception is not raised automatically, but instead special values are returned by the hardware. In these cases intermediate results will not be correct and the final result might be completely wrong.

Although it appears from the *Language Reference Manual* (1983) that the occurrence of underflow in computations is not possible, a great number of basic functions checks the magnitude of its arguments. Mainly this is done from the point of view of efficiency. When an argument is less than a certain threshold, which depends on the precision, a special value is returned. This special value on its turn may depend on the argument. For example, for very small values of the argument the SIN returns this (argument) value and the COS returns one in this case.

5. FUNCTION DESCRIPTIONS

The package body of GENERIC_MATH_FUNCTIONS contains, beside the bodies of the elementary functions, a number of auxiliary functions, the package MACH_DEP_CONSTANTS containing machine-dependent entities and an instantiation of a package of primitive functions. This package is also generic and has a floating-point type as generic parameter. The instantiation is given with the formal parameter of GENERIC_MATH_FUNCTIONS as actual parameter. The package has some useful functions, which can be used to decompose a floating-point number in its (binary) mantissa and exponent (or simply yield one of these) or to construct a floating-point number from these parts. It will appear that this kind of primitive functions is indispensable for a number of elementary functions.

Now we will describe a number of details which play a part in the computations of the function values of the various basic functions. For this purpose it is assumed that the input parameters are all in the prescribed function domains. Hence, it is not necessary to repeat each time that the correctness of the arguments is tested.

5.1. SQRT

The SQRT routine calculates values of the square root function for non-negative arguments.

The implementation of the SQRT has been taken literally from the *Guidelines*. Therefore we refer to these *Guidelines* and to *Winter* (1986) for more information.

5.2. LOG

The LOG routine calculates values of the base-*BASE* logarithm for positive arguments. *BASE* must be a positive floating-point number which may not be one. The default value of *BASE* is e.

The logarithm with a base other than e is computed from the natural logarithm through the identity ${}^{b}log(x) = \frac{ln(x)}{ln(b)}$, where b is a positive floating-point number unequal to one. For some values of b, viz. integer values from two up to ten, the term $\frac{1}{ln(b)}$ has been computed in advance and therefore only $ln(x)$ has to be computed.
Hence, the computation of the logarithm function can be restricted to one for the natural logarithm. First the argument x is rewritten as $f*2^m$, $\frac{1}{2} \le f < 1$ and m an integer. Then g and n, an integer multiple of 1/2, are determined such that $f = g*2^{-n}$ and $2^{-\frac{1}{4}} \le g < 2^{\frac{1}{4}}$. Now $ln(x)$ can be expressed in m, n and g : $ln(x) = (m-n)*ln(2) + ln(g)$.
For the computation of $ln(g)$ we define $s = \frac{g-a}{g+a}$, $a = \frac{1}{2}\sqrt{2}$, if $2^{-\frac{3}{4}} < f \le 2^{-\frac{1}{4}}$ and $s = \frac{g-1}{g+1}$ otherwise. Next $ln(g)$ can be evaluated using respectively $ln(g) = ln(a*\frac{1+s}{1-s})$ and $ln(g) = ln(\frac{1+s}{1-s})$. The evaluation is performed by a near-minimax approximation using the variable $z = 2s$. Finally, the logarithm for the original argument is reconstructed by reapplying the above identities.

In general the computation of $ln(x)$ is numerically well-behaved. In most cases the term $(m-n)*ln(2)$ dominates and small disturbances in $ln(g)$ do not affect the final result. However, when $m-n$ vanishes, the precision of $ln(g)$ becomes vital. Since $ln(g) = 2s + s*\sum_{i=1}^{\infty} a_i s^{2i}$ the precision of $ln(g)$ depends on the precision of z. By computing z directly from f, without determing g and s explicitly, the value of z is calculated as accurate as possible.

5.3. EXP

The EXP routine calculates values of $BASE^X$ where *BASE* is a positive floating-point number and X is arbitrary. *BASE* may also be zero but then X must be positive. The default value of *BASE* is e, the base of the natural logarithm.

Although it seems obvious to compute $BASE^X$ in terms of the exponential base-e and logarithm function, this should be avoided because large errors might disturb the final result. To see this let $y = X * ln(BASE)$ and $z = e^y$. Then the relative error in $z = BASE^X$ is approximately the absolute error in y, which is proportional to the magnitude of y because of the finite word length of the computer. The only way to decrease the relative error in z is to compute $ln(BASE)$, y and z in an extended precision. However, in Ada it is not allowed to use attributes of generic actual parameters for type declarations, since these attributes are not static. Therefore it is not possible to define a local floating-point type with a larger accuracy and perform (a part of) the computations in this extended precision.
The alternative is to use some pseudo extended precision. In *Software manual for the elementary functions (Cody and Waite, 1980)* an algorithm for the computation of the power function is given, which uses this extended precision. Our implementation of this function has been taken from the *Software manual* to a great extent. Therefore we refer to the *Software manual* for a more detailed description of the power function.

The power function, for bases other than e, is computed by determining respectively $^2log(BASE)$, $X * {}^2log(BASE)$
and $2^{X * {}^2log(BASE)}$. For the first and third step a near-minimax polynomial approximation is used, except when the base equals e. In that case only the third step uses such an approximation and the first and second step are combined to form $X * {}^2log(e)$ in a pseudo extended precision.

5.4. SIN

The SIN routine calculates values of the sine function for an arbitrary argument and given non-zero cycle. The default value of the cycle is 2π.

The computation of $sin(x)$ or $sin(x, cycle)$ involves three steps : the reduction of the given argument x to a related argument $\tilde{x}$, the evaluation of $sin(\tilde{x})$ over a small interval symmetric around the origin, and the reconstruction of $sin(x)$ or $sin(x, cycle)$ from these results.
$sin(\tilde{x})$ is computed by a routine SIN_COS, which can also be used to calculate the cosine using the identity $cos(x) = sin(x + \frac{\pi}{2})$.
The evaluation of $sin(\tilde{x})$ is accomplished by a polynomial approximation, which has been derived from a Chebyshev series of the sine and cosine function.

The critical step in the computation of $sin(x)$ is the argument reduction. For this purpose a special routine REDUCE_RANGE has been designed, which reduces the argument modulo cycle and scales the result. More specifically, it determines the reduced argument $\tilde{x}$, $-\frac{\pi}{4} \le \tilde{x} \le +\frac{\pi}{4}$, and an integer N, $0 \le N < 4$, such that $x - \tilde{x} * \frac{cycle}{2\pi} - N * \frac{cycle}{4}$ is a multiple of $cycle$. Under the assumption that x is error-free, this reduction is exact.
However there might be cases where the argument can not be used for calculating the function value with useful accuracy (e.g. for a call of $sin(10^{REAL'DIGITS})$). The problem here is that the function body can not be made aware that the user expects a smaller precision than normally since all functions have the same floating-point type for parameter(s) and function result.
Two possible solutions for this problem have been rejected by the *Guidelines*. The first of these solutions consists of providing a second generic parameter which specifies a number of digits that may be lost without

raising an exception SIGNIFICANCE_ERROR. The other solution consists of restricting the arguments of SIN, COS, TAN and COT to the range $[-2\pi, +2\pi]$. However, neither of these solutions is supported.

5.5. COS

The COS routine calculates values of the cosine function for an arbitrary argument and given non-zero cycle. The default value of the cycle is 2π.

The cosine is computed in a similar way as the sine function. Therefore we refer to the SIN routine for more information.

5.6. TAN

The TAN routine calculates values of the tangent function for an arbitrary argument and given non-zero cycle. The default value of the cycle is 2π.

The computation of $tan(x)$ or $tan(x,cycle)$ involves three steps : the reduction of the given argument x to a related argument $\tilde{x}$, the evaluation of $tan(\tilde{x})$ over a small interval symmetric around the origin, and the reconstruction of $tan(x)$ or $tan(x,\ cycle)$ from these results.
For the reduction of the argument to the interval $[-\frac{\pi}{4}, +\frac{\pi}{4}]$ we use the same routine, REDUCE_RANGE, as for the reduction of the argument of the sine and cosine. Further, by setting $y = |\frac{2}{\pi}\tilde{x}|$ and using $tan(-\tilde{x}) = -tan(\tilde{x})$ we find $tan(\tilde{x}) = \pm tan(\frac{\pi}{2}y)$, $y \in [0, \frac{1}{2}]$. Since the approximating Chebyshev series of $tan(\frac{\pi}{2}y)$ still converges too slowly for y near $\frac{1}{2}$, we have to reduce the interval once more by using the identity $tan(\frac{\pi}{2}y) = \frac{1 - tan((\frac{1}{2} - y) * \frac{\pi}{2})}{1 + tan((\frac{1}{2} - y) * \frac{\pi}{2})}$. This relation will be used if $y > \frac{1}{4}$, thus reducing the interval to [0, 1/4]. Although the convergence of the Chebyshev series of the tangent function on this interval is still too slow, we can speed it up by taking a Chebyshev series of $(y-1)*(y+1)*tan(\frac{\pi}{2}y)$. Hence $tan(\frac{\pi}{2}y)$ will not be computed directly, but first $(y-1)*(y+1)*tan(\frac{\pi}{2}y)$ is computed and the result is divided by (y^2-1).
Finally $tan(x)$ is constructed from these results and the formula $tan(x) = \begin{cases} tan(\tilde{x}) & N\ even \\ -\frac{1}{tan(\tilde{x})} & N\ odd \end{cases}$ where N is determined by REDUCE_RANGE.

The tangent function is very sensitive to small errors in the argument. Especially for arguments near its zeros and singularities a small relative error in the argument may cause a large relative error in the final result. To prevent the loss of significant digits as much as possible, the argument reduction must be performed very accurately. By using the routine REDUCE_RANGE the argument will be reduced without loss of precision, providing that the argument is error-free.
For very large arguments the same remarks can be made as for very large arguments of the sine and cosine function.

5.7. COT

The COT routine calculates values of the cotangent function for an arbitrary argument and given non-zero cycle. The default value of the cycle is 2π.

The cotangent is computed directly from the tangent through the identity $cot(x) = \frac{1}{tan(x)}$.

5.8. ARCSIN

The ARCSIN routine calculates values of the inverse sine function for an argument in the range [−1, +1].

The computation of $arcsin(x)$ involves three steps : the reduction of the given argument x to a related argument $\tilde{x}$, the evaluation of $arcsin(\tilde{x})$, and the reconstruction of $arcsin(x)$ from these results.
$arcsin(\tilde{x})$ is computed by means of a routine ARCSIN_COS, which can also be used to calculate the arccosine using the equations $arccos(x) = \begin{cases} \frac{\pi}{2} - arcsin(x) & \text{if } 0 \le |x| \le \frac{1}{2}\sqrt{2} \\ arcsin(\sqrt{1-x^2}) & \text{if } \frac{1}{2}\sqrt{2} < |x| \le 1. \end{cases}$

Only for arguments close to −1 or +1 is the final value of the arcsine sensitive to small argument errors. However, even when the argument is free of error, small errors introduced during the argument reduction can result in a substantial loss of accuracy in the final result. Hence, care has to be taken during the argument reduction process.
The argument reduction is performed in two steps. First the computation of the arcsine is reduced to one for non-negative arguments through the identity $arcsin(-x) = -arcsin(x)$. Next, this interval is reduced further through the equation $arcsin(x) = a + arcsin(x * cos(a) - \sqrt{1-x^2} * sin(a))$, $0 \le sin(a) \le x$, where a is an arbitrary fixed number, $0 \le a < \frac{\pi}{2}$, for which the sine and cosine values are known. With an appropriate choice of a it is possible to make the expression $\tilde{x} = x * cos(a) - \sqrt{1-x^2} * sin(a)$ as small as we want. By providing a list of values of a with corresponding sine and cosine values, we can choose a from this list, such that $sin(a)$ is the largest sine value not exceeding x. In this way we can reduce the argument range to $[0, sin(\frac{\pi}{32})]$.
In *Single and double-length computations of elementary functions (Hemker et al.,* 1973) it is shown that the error introduced by this reduction is not excessive.
Next, $arcsin(\tilde{x})$ is approximated on $[0, sin(\frac{\pi}{32})]$ using a polynomial derived from a Chebyshev series of $(\frac{arcsin(\tilde{x})}{\tilde{x}} - \frac{\pi}{2}) * \sqrt{1-x^2}$. By taking this series expansion we obtain a reasonably fast convergence and preserve a high relative precision near zero.

5.9. ARCCOS

The ARCCOS routine calculates values of the inverse cosine function for an argument in the range [−1, +1].

The arccosine function is computed in a similar way as the arcsine. Therefore we refer to the ARCSIN routine for more detailed information.

5.10. ARCTAN

The ARCTAN routine calculates values of the inverse tangent function with the result in $(-\pi, +\pi]$. This is defined as the argument (in radians) in $(-\pi, +\pi]$ of the cartesian point (y, x). For calls with $y > 0$, $arctan(\frac{x}{y})$ is computed with the result in $(-\frac{\pi}{2}, +\frac{\pi}{2})$. The default value of the second argument is one.

The computation of $arctan(x)$ or $arctan(x,y)$ involves three steps : the reduction of the given argument(s) to a related argument $\tilde{x}$, the evaluation of $arctan(\tilde{x})$ over a small interval symmetric around the origin and the reconstruction of $arctan(x)$ or $arctan(x,y)$ from these results.
The argument reduction is performed in several steps. First the identities $arctan(-x) = -arctan(x)$ and $arctan(x) = \frac{\pi}{2} - arctan(\frac{1}{x})$ are used to obtain a non-negative argument which does not exceed one. After this we use the identity $arctan(x) = a + arctan(\frac{x-\tilde{a}}{x+\tilde{a}})$, where $\tilde{a} = tan(a)$, to reduce the argument range to $[-tan(\frac{\pi}{24}), +tan(\frac{\pi}{24})]$. The value of a depends on the argument x and can be equal to $\frac{\pi}{24}$, $\frac{3}{24}\pi$ or $\frac{5}{24}\pi$. Now $arctan(\tilde{x})$ is evaluated on $[-tan(\frac{\pi}{24}), +tan(\frac{\pi}{24})]$ using a near-minimax polynomial approximation.

The function value is relatively insensitive to small errors introduced during the reduction of the argument(s). Thus small errors will not lead to a severe loss of significance in determining the function value. Nevertheless, the reduction of the argument(s) and the reconstruction of the arctangent for the original argument(s) have to be done carefully.

5.11. ARCCOT

The ARCCOT routine calculates values of the inverse cotangent function with the result in $(-\pi, +\pi]$. This is defined as the argument (in radians) in $(-\pi, +\pi]$ of the cartesian point (x, y). For calls with $y > 0$, $arccot(\frac{x}{y})$ is computed with the result in $(0, \pi)$. The default value of the second argument is one.

The arccotangent is computed directly from the arctangent through the identity $arccot(x) = arctan(\frac{1}{x})$ or $arccot(x, y) = arctan(y, x)$.

5.12. SINH

The SINH routine calculates values of the hyperbolic sine function for an arbitrary argument.

The way $sinh(x)$ is computed depends on the size of the argument. For arguments x with $|x| \leq 1$ we approximate $sinh(x)$ by means of a polynomial and for the other arguments the computation is formulated in terms of the exponential function. Both alternatives have in common that they restrict the computation to one for non-negative arguments by using the identity $sinh(-x) = -sinh(x)$.
We will first discuss the approximation for arguments that do not exceed one. For these arguments a formulation in terms of the exponential function, $sinh(x) = \frac{e^x - e^{-x}}{2}$, may cause a severe loss of accuracy because the subtraction concerns two nearly equal quantities. This problem is solved by using a polynomial approximation, derived from a Chebyshev series of the hyperbolic sine for arguments less than or equal one. This series converges so fast on [0, 1] that it is not necessary to reduce this interval any further.

For arguments greater than one it is possible to compute $sinh(x)$ through its definition, i.e. $sinh(x) = \frac{e^x - e^{-x}}{2}$. However there are arguments that may cause overflow in the computation of e^x but for

which the hyperbolic sine should not overflow. Rewriting the computation for large arguments as $sinh(x) = e^{x-ln2}$ is not sufficient (the negative exponential term can be ignored). It is easy to see that in this case the relative error in $sinh(x)$ equals the absolute error in $x - ln2$. Since $ln2$ is not exactly representable, this absolute error is proportional to the magnitude of x, even when x is error-free. To avoid this problem we reformulate the computation of $sinh(x)$, $x > 1$, to $sinh(x) = \frac{v}{2} * (e^{x-ln v} - \frac{e^{-(x-ln v)}}{v^2})$, where $ln v$ is an exact machine number slightly larger than $ln2$. Using this numerically stable expression, the relative error in the result only depends on the error in x.

5.13. COSH

The COSH routine calculates values of the hyperbolic cosine function for an arbitrary argument.

In the computation of cosh(x) the same problem may arise for large arguments as for the hyperbolic sine. Therefore we reformulate the computation of the hyperbolic cosine for arguments greater than one in a similar way. Thus, by defining $cosh(x) = \frac{v}{2} * (e^{|x|-ln(v)} + \frac{e^{-(|x|-ln(v))}}{v^2})$, $|x| > 1$ and $ln v$ exactly representable, we assure that the relative error in the result only depends on the error in x.

However, for arguments not exceeding one, the definition of $cosh(x) = \frac{e^x + e^{-x}}{2}$ is used, which is numerically more stable.

5.14. TANH

The TANH routine calculates values of the hyperbolic tangent function for an arbitrary argument.

The computation of the hyperbolic tangent is reduced to one for non-negative arguments through the identity $tanh(-x) = -tanh(x)$. The computation of $tanh(x)$, $x \geq 0$, can be accomplished in several ways depending on the magnitude of x. For arguments beyond a certain threshold $tanh(x) = 1$ to machine precision. For smaller arguments $tanh(x)$ will be computed using the exponential function. In this case we use a stable computation which consists of approximating $tanh(x)$ by $1 - \frac{2}{e^{2x}+1}$. However, the argument may not be too small because a subtraction error may disturb the final result. Such an error may occur whenever $x < \frac{ln3}{2}$. Therefore we will use the latter expression only if the argument is greater than or equal to $\frac{ln3}{2}$. Finally, for arguments less than $\frac{ln3}{2}$ we will use a near-minimax polynomial approximation.

Since this series converges too slowly for arguments near $\frac{ln3}{2}$ we have to reduce the interval [0, $\frac{ln3}{2}$]. By using the identity $tanh(x) = \frac{tanh(\tilde{x}) + tanh(a)}{1 + tanh(\tilde{x}) * tanh(a)}$, $\tilde{x} = x - a$, and choosing appropriate values for a, we can reduce the interval to [0, 1/8]. After evaluating $tanh(\tilde{x})$ on this interval we can reconstruct the hyperbolic tangent for the original argument by reapplying the above identities.

5.15. COTH

The COTH routine calculates values of the hyperbolic cotangent function for a non-zero argument.

The hyperbolic cotangent is computed directly from the hyperbolic tangent through the identity $coth(x) = \frac{1}{tanh(x)}$.

5.16. ARCSINH

The ARCSINH routine calculates values of the inverse hyperbolic sine function for an arbitrary argument.

The computation of the inverse hyperbolic sine can be reduced to one for non-negative arguments through the identity $arcsinh(-x) = -arcsinh(x)$. Again, the way $arcsinh(x)$ is computed, depends on the magnitude of the argument. For arguments greater than one $arcsinh(x)$ can be determined with the aid of the *log* and *sqrt* function : $arcsinh(x) = log(x) + log(1 + \sqrt{1 + \frac{1}{x^2}})$. In this expression both terms on the right side should not be combined, since this may cause overflow for very large arguments. By writing the equation in this form, we obtain a numerically stable expression, which will not overflow even for very large arguments.

For small arguments the above approximation is not usable since $log(x) \simeq -log(1 + \sqrt{1 + \frac{1}{x^2}})$ and this may cause a severe loss of precision. Therefore we evaluate $arcsinh(x)$, $x \leq 1$, by means of a poynomial, derived from a Chebyshev series of the inverse hyperbolic sine.

However, since this series converges too slowly on [0, 1], we have to reduce this interval. To reduce the interval to [0, 1/8] the following transformation is used : $arcsinh(x) = arcsinh(a) + arcsinh(x * \sqrt{1+a^2} - a * \sqrt{1+x^2})$, $0 \leq a \leq 1$, where a is an arbitrary fixed number of which the inverse hyperbolic sine values are known. With an appropriate choice of a, it is possible to make the expression $\tilde{x} = x * \sqrt{1+a^2} - a * \sqrt{1+x^2}$, smaller than 1/8. By providing a list of inverse hyperbolic sine values for arguments that are multiples of 1/8, a can be chosen such that it is the largest integer multiple of 1/8 not exceeding x.

In a similar way as for the ARCSIN it can be shown that the error introduced by this reduction is not excessive. After reducing the argument to $\tilde{x}$, $arcsinh(\tilde{x})$ is evaluated on [0, 1/8] using a near-minimax polynomial approximation. Finally, the function result for the original argument is reconstructed by reapplying the above identities.

5.17. ARCCOSH

The ARCCOSH routine calculates values of the inverse hyperbolic cosine function for an argument of at least one.

For arguments greater than $\sqrt{2}$ $arccosh(x)$ can be computed by means of the *log* and *sqrt* function : $arccosh(x) = log(x) + log(1 + \sqrt{1 - \frac{1}{x^2}})$. In this expression both terms on the right side should not be combined, since this may cause overflow for very large arguments. By writing the equation in this form, we obtain a numerically stable expression which will not overflow even for very large arguments.

For small arguments this approximation is not usable because large errors may occur in calculating the logarithm for arguments near one. Therefore we compute $arccosh(x)$, $1 \leq x \leq \sqrt{2}$, by means of the inverse hyperbolic sine through $arccosh(x) = arcsinh(\sqrt{x^2 - 1})$.

5.18. ARCTANH

The ARCTANH routine calculates values of the inverse hyperbolic tangent for arguments in the open interval (–1, +1).

The computation of the inverse hyperbolic tangent can be reduced to one for non-negative arguments through the identity $arctanh(-x) = -arctanh(x)$.

The way $arctanh(x)$ is computed, depends on the magnitude of the argument. For arguments greater than or equal to 1/2 $arctanh(x)$ can be determined by means of the *log* function : $arctanh(x) = \frac{1}{2} * log\frac{1+x}{1-x}$. For smaller arguments this approximation is not usable since the subtraction in the denumerator of the *log*-argument may cause a loss of accuracy. Therefore we evaluate $arctanh(x)$, $0 \leq x < 1/2$, by means of a near-

minimax polynomial approximation. However, since this approximation converges too slowly on [0, 1/2) we have to reduce this interval. By using the identity $arctanh(x) = arctanh(a) + arctanh(\frac{x-a}{1-a*x})$, where a is fixed and $arctanh(a)$ is known, the interval can be reduced to $[0, \frac{1-\sqrt{1-b^2}}{b}]$ where $b = 2-\sqrt{3}$.
After reducing the argument to a related argument $\tilde{x}$ in this interval, $arctanh(\tilde{x})$ is evaluated using a near-minimax polynomial approximation. Finally, the function result for the original argument is reconstructed by reapplying the above identities.

5.19. ARCCOTH

The ARCCOTH routine calculates values of the inverse hyperbolic cotangent for arguments greater than one.

The inverse hyperbolic cotangent is computed directly from the inverse hyperbolic tangent through the identity $arccoth(x) = arctanh(\frac{1}{x})$.

6. CONCLUSIONS

Although we should not say too much of the quality of the Ada package of elementary functions as long as it has not been tested thoroughly, some remarks can be made at this time.

When we tried to make an instance of GENERIC_MATH_FUNCTIONS, we ran into the predefined exception STORAGE_ERROR raised on compilation. After some experimenting this turned out to be the case for each floating-point type that was used as a generic parameter. What happened was that the compiler used at the CWI gathered the code of all elementary functions and tried to compile the whole at once. This happened despite the fact that the function bodies were given separate from the package body. This should mean that the compiler should compile the function bodies separated from the package body (but this did not happen).
To overcome this problem in an ad hoc fashion, we made some hand-made non-generic instances of the package. Instead of a generic parameter we provided the specification of the package with a type declaration of type FLOAT_TYPE, which was specified in an other package. In this way the number of changes which had to be made in the package specification, was limited. The disadvantage of this solution was that for each static instance a copy of the package had to be made by the user instead of by the machine.

A second problem concerned the use of default parameters. During compilation of the hand-made versions of the package an error message was received concerning the default parameters. According to the compiler the specification of functions with a default parameter, which was not a named number, did not conform to the corresponding function specification in the package specification.
This compiler bug was avoided by splitting up these functions in a function with the same name and same parameters, only without the default value, and a function with the same name of which the parameter with the default value had been omitted. When a function call of the initial function with a default parameter is made, (i.e. when the function is called without the parameter which has a default value), the latter function is called automatically. This function on its turn calls the former function with the default value mentioned explicitly as actual parameter.

Finally we encountered a problem with the predefined type LONG_FLOAT. When a hand-made version of the package had been made, it appeared the compiler interchanged the built-in type FLOAT and LONG_FLOAT in complicated expressions. Consequently the result was less accurate than required.
This problem was solved by splitting up these expressions into simpler expressions. Therefore it was sometimes necessary to adjust large parts of the package body.

REFERENCES

Abramowitz, M. and Stegun, I.A. (eds.) *Handbook of Mathematical functions*, Dover, New York, 1965

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

Barnes, J.G.P. *Programming in Ada*, Addison-Wesley, London, 1982

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

Habermann, A.N. and Perry, D.E. *Ada for experienced programmers*, Addison-Wesley, 1983

Hemker, P.W., Hoffmann, W., Kampen, S.P.N. van, Oudshoorn, H.L. and Winter, D.T. *Single and double-length computations of elementary functions*, MC report NW 7/73, 1973

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-N 8401, 1984

Winter, D.T. *The implementation of standard functions in Ada*, in : Ford, B., Kok, J. and Rogers, M.W. (eds.) *Scientific Ada*, Cambridge University Press, 1986