Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M. Bergman

Testing of elementary functions in Ada

Department of Numerical Mathematics Report NM-R8909 May

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 Research (N.W.O.).

Testing of Elementary Functions in Ada

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

The elementary mathematical functions are not comprised in the language Ada. Since these functions are essential for much software a collection of basic functions has been designed and implemented. In the next stadium the functions have to be tested thoroughly. In this paper a number of test methods is described together with the tests that are used to examine the elementary functions. Further, the results of the tests are interpreted in two ways: they are compared with the results of the tests of non-Ada libraries and they are interpreted taking into account the model number arithmetic of Ada.

1980 Mathematics Subject Classification: 69D49, 65-04.
Key Words and Phrases: Ada, elementary mathematical functions, portability, scientific libraries, testing.

1. Introduction

Many programming languages contain standard declarations of the elementary functions. In Ada this is not the case. In spite of many useful features and concepts the elementary functions are not part of the language. The need arising from this lack of a collection of basic functions can be met by supplying the functions in the form of additional software. However, the absence of commonly adopted specifications will cause many different designs and implementations. Programs built on such manufacturer-supplied or user-supplied functions will, in general, suffer from insufficient portability. As portability is one of the most important aims of Ada, it is necessary to have a uniform specification for the elementary functions.

An important step towards a common specification was made in the project *Pilot Implementations of Basic Modules for Large Portable Numerical Libraries in Ada.* Part of the project, which later in this paper will be referred to as PIA, was the design and implementation of a portable library of elementary functions. Starting from the PIA design an implementation of the elementary functions has been made. The next step consists of the design and implementation of tests that examine these standard functions. In the tests various aspects of the functions will have to be tested carefully, like:

* the accuracy of the functions for valid arguments
* specific function-dependent properties like odd or even parity
* the behaviour of the functions for special acceptable arguments
* the behaviour of the functions for incorrect arguments.

For all tests it is essential to be constructed cautiously in order not to contaminate the results of the functions through inaccuracies of the test procedures. As the word *elementary* indicates, the elementary functions are the basis of much software. This means that they should be accurate and reliable. Test programs that introduce large errors are useless for determining the qualities of the

Report NM-R8909
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

functions being tested.

A second requirement that should be met by the tests is that they be machine-independent. One of the goals of the above-mentioned project was to achieve a library of elementary functions that should be highly portable. To measure the performances of the functions on various machines objectively, it is necessary to submit the functions to the same tests on every machine. For that reason the tests should be portable as well, so that they can easily be executed on different machines without large adaptations.

A number of tests agree with tests occurring in the *Software Manual for the Elementary Functions (Cody and Waite, 1980)*. In this book implementation schemes and accessory test programs in Fortran are given for a number of elementary functions. However, it does not provide test programs for every standard function since Cody and Waite consider a smaller set of elementary functions. Moreover, a number of test programs can be extended as a result of different specifications of some basic functions. What we present and discuss is a portable package of self-contained test programs written in Ada. This package should be able to test the collection of elementary functions straightforward irrespective of the underlying hardware.

2. Test subjects

Good test programs should demonstrate not only the capabilities and quality of the subprograms being tested but also their limitations and deficiencies. In a way we could say that test results which show bad performance, are more helpful to us than those that do not signal any irregular behaviour. When, after testing of a subprogram, the results appear to be unsatisfactory, or even wrong, the subprogram can be corrected.

On the other hand, the absence of errors in a program that has been tested, does not guarantee that the program is error-free. Since in most cases it is not possible or not feasible to test all aspects of a program, a selection has to be made. As with every selection this implies that certain facets are left out. By taking a deliberate choice of what is tested and what is not, we keep the possibility that undetected errors occur small and at the same time we get a good view of the capabilities of a program.

An example of a property of some elementary functions that has not been tested, is the monotonicity. Since it is extremely difficult to test whether a function is monotonic or not, and the efforts to examine this would be disproportionately complex in comparison with the other tests, a test for monotonicity has been omitted.

In the next subsections the tests will be discussed that are used to examine the performance of the elementary functions.

2.1. Accuracy tests

One of the properties of the elementary functions that should be tested thoroughly is the accuracy. Since much software relies on these functions, it is a prerequisite that they are well-behaved on their domains and the function results are as accurate as possible.

For measuring the accuracy of a function it is necessary to understand what exactly has to be measured. For this purpose we need some basic calculus. Let $y = f(x)$ be a differentiable function. Then $\left|\frac{dy}{y}\right| = \left|x\frac{f'(x)}{f(x)}\right|\left|\frac{dx}{x}\right|$ i.e. the relative error $\left|\frac{dy}{y}\right|$ in the function value (the so-called *transmitted error*) is obtained by multiplying the relative error $\left|\frac{dx}{x}\right|$ in the function argument (the so-called *inherited error*) by some quantity which solely depends on the numerical properties of the function f. Since the relative error $\left|\frac{dy}{y}\right|$ has only been constructed from factors that are not directly attributable to the

subprogram, this kind of error should be disregarded when the accuracy is tested. Only the errors generated by the program itself, for example errors due to the truncation of series expansions, cancellation of significant digits in forming a sum and rounding errors, should be taken into account. These errors are grouped together in the so-called *generated error.*

2.1.1. Accuracy test methods.

There are several methods for determining the accuracy of a program. The first and most simple method consists of taking a number of input values, computing the corresponding function values and comparing these with 'correct' results as can be found in tables etc. The disadvantage of this method may be clear. When a rather small number of test arguments is used, this procedure is fairly arbitrary, and when a large number of input arguments is used this method becomes tedious and cumbersome. Besides, it is not plausible that tables used for comparison will contain an entry for every argument for which the function value has to be determined. This implies that one is either restricted to arguments that occur in tables or one should extrapolate table values to obtain an estimate of the correct answer.
Finally, this way of comparing results is not very reliable since it is a human activity and deficiencies can easily go unnoted (the results may be up to 35 significant digits!).

A second method for measuring the generated error consists of comparing the computed results with results obtained for the same arguments by a different algorithm in higher precision arithmetic. A different algorithm should be used to avoid the occurrence of systematic programming errors. However, also against this method a number of objections can be made despite the fact that such tests give very reliable accuracy statistics.
First the elementary functions are not included in Ada. If a package of such functions is provided in an Ada environment it is machine-dependent, which makes it unsuited to be used in test programs. Besides, the package itself would have to be verified.
A consequence is that one would have to provide the higher accuracy standard functions in the form of additional software. The effort to accomplish such tests would be much greater than the effort to produce the subprograms being tested. Even if the higher precision arithmetic had been available one should have a detailed knowledge of the underlying computer arithmetic which, again, does not improve the portability of the test programs.
All these drawbacks make that this method of measuring the accuracy is also not suitable for testing the elementary functions.

The last method that is described here uses mathematical identities to measure the generated error. An example of such an identity is $log(x^2) = 2\,log(x)$. Unfortunately the error that is measured in this way is not just confined to the generated error, since these identities involve operations beyond the control of the subprograms being tested. When calculations of identities contaminate the function error excessively, the measured error will be less suitable for obtaining an impression of the generated error. In most cases a rough estimate can be made of the difference between the measured error and the generated error, so that it is possible to determine in advance whether the use of a specific identity should be rejected or not.
The following example illustrates the preference of a mathematical identity to another, apparently equivalent, identity. Consider the testing of the square root function on the interval (0.25, 1). Both the identity $\sqrt{x}^2 = x$ and $\sqrt{x^2} = x$ can be employed to estimate the error made by the square root. In the left-hand side of the former identity the function maps the interval (0.25, 1) into the interval (0.5, 1). Since the first interval contains roughly twice as many floating-point numbers as the second interval, the square root has to map an average of two different arguments into each result. When further the result is squared, at most one of the possible original arguments can be got back. Consequently the computation of $\sqrt{x}^2$ will on average return x for only about half of the arguments,

irrespective of the precision of the square root function.
When using the identity $\sqrt{x^2} = x$ the left-hand side is computed by first squaring the argument. This operation maps each argument in the interval (0.25, 1) into a unique argument in the interval (0.0625, 1). Because in Ada this mapping is required to be accurate to within rounding error for model numbers, the computation of $\sqrt{x^2}$ may be expected to return x in most of the cases where the square root is accurate.
Therefore the identity $\sqrt{x^2} = x$ is more appropriate for this test and gives a more reliable estimate of the error than the identity $\sqrt{x^2} = x$.

There is another case in which the use of mathematical identities leads to unreliable results: an identity that is already used to compute a function. To illustrate this assume that the hyperbolic cosine is computed by first calculating the hyperbolic sine and then using the identity $\cosh^2(x) - \sinh^2(x) = 1$. Now this identity is no longer useful for measuring the accuracy of either the hyperbolic sine or the hyperbolic cosine since the computed function values automatically satisfy the identity, whatever the accuracy might be. The results that would be produced by using such equalities would be misleading.

2.1.2. Test intervals.

Most elementary functions are computed following a fixed concept: first the argument is reduced to a related argument in a small interval on which the function value can be computed in a fast and efficient way. Then the function value is calculated for this reduced argument and finally the function value for the original argument is constructed from the intermediate results. Each of these steps contributes to the final error made by the function. By selecting appropriate test intervals it is possible to get a view of the influences of each of these contributions.
A test interval that lies in the *primary range* of the function, i.e. the interval in which no argument reduction is required, will give an estimate of the accuracy of the main approximation regardless of the reduction of the argument. It is often not possible to measure the additional errors induced by the argument reduction and by the construction of the final function value separately. However, one can observe the effects of these steps by selecting several test intervals, containing different size arguments, such that the arguments have to be reduced more or less.
For elementary functions to which the above-mentioned scheme is not applicable, the bounds of the test intervals must be chosen on other grounds, depending on the function being tested. In chapter 5 the accuracy tests for each elementary function are discussed, a description is given of the test intervals together with an explanation why these intervals have been selected.

2.1.3. Error statistics.

Once a mathematical identity and a test interval have been chosen, the question remains how to measure the accuracy of the function on this interval. For this purpose two quantities are introduced: MRE (Maximum Relative Error) and RMS (Root Mean Square). The MRE is an indicator for the worst-case behaviour of the function for all selected arguments, whereas the RMS indicates the overall quality on the interval. The MRE and the RMS are defined as follows: MRE $= \max |RE|$ and RMS $= \sqrt{\frac{\sum RE^2}{N}}$, where N denotes the number of arguments selected from the test interval and max and $\sum$ are to be taken over these N arguments. For each argument the relative error RE is defined by $RE = \left| \frac{F(x) - f(x)}{f(x)} \right|$, $f(x) \neq 0$, where $F(x)$ is the computed function value and $f(x)$ is the 'correct' value.
Besides the decimal value of the MRE and the RMS the number of base β digits will be given that is

lost in the computation, where β denotes the radix of the floating-point system.
The following example shows that the results ought to be interpreted carefully. Let $F(x) = 1 - \epsilon$ and $F(y) = 1 + \epsilon$ for a small $\epsilon > 0$. Further assume that both results have an error of one unit in the last digit of the significand. Now the *RE* for $F(x)$ will report one more correct base β digit than the *RE* for $F(y)$ will do.

The MRE and the RMS would be sufficient to measure the accuracy of the elementary functions for almost every implementation involving another programming language. On the other hand in Ada it is not quite satisfactory to observe just the values of these quantities. By doing so we do not take account of the way the accuracy of the predefined floating-point types (and also fixed-point types) and the attended predefined operations, like addition and multiplication, is defined by the language.

Ada defines the accuracy required from any predefined operation giving a real result in terms of *model numbers*. With each real type, i.e. a floating-point type or a fixed-point type, a set of numbers called model numbers is associated. An implementation of a real type must include at least the model numbers and represent them exactly.
In the definition of any user-defined floating-point type the minimum number of significant decimal digits must be specified. This number is denoted by *D* in the remainder of this section. Now the set of model numbers of the user-defined floating-point type consists of zero and all numbers of the form *sign* * *mantissa* * $2^{exponent}$. In this notation *sign* is either $+1$ or -1; *mantissa* is a binary mantissa with *B* digits after the point; and *exponent* is an integer number in the range $-4*B$ .. $+4*B$ such that the integer part of *mantissa* is zero and the first digit of its fractional part is not a zero. The length of the binary mantissa depends on the value of *D*.
The model numbers of a predefined floating-point type are defined in terms of the number *D* of decimal digits returned by the attribute DIGITS.

Normally, the hardware does not provide a different floating-point type for every user-defined floating-point type. Therefore, it is inevitable to map every user-defined floating-point type internally to some predefined type. This mapping takes place automatically, so the user is not concerned in choosing the appropriate built-in type. The type on which the user-defined type is mapped, is called the *base type* of the floating-point type. The model numbers associated with the base type of a floating-point type include at least the set of model numbers of the type.

Since the range of model numbers only depends on the floating-point type definition, it is independent of the implementation. Considering this and the fact that error bounds on the predefined operations are given in terms of model numbers, it is obvious to relate the results of the accuracy tests to the model numbers. To achieve this we will express the MRE in ϵ, where ϵ denotes the absolute value of the difference between the model number 1.0 and the next model number above.
However, here we have two possibilities: either we use the epsilon of the set of model numbers associated with the floating-point type in question, or we use the epsilon of the set of model numbers of the base type of the floating-point type. In other words we use the value obtained through either the attribute EPSILON or through the attribute BASE'EPSILON.

The current specification of a generic package of elementary functions *(ACM SIGAda Numerics Working Group, February 1989)* relates to a number of detailed decisions. It defines the names of the various functions, the number of the parameters of each function, the names of the parameters etc. Other points that are covered by the specification are the domain and range definitions of the functions. Finally, for every function it gives accuracy requirements that should be met by the implementation.
For this specification, proposed by the co-operating Ada-Europe Numerics Working Group and the ACM SIGAda Numerics Working Group, the choice for relating the accuracy requirements to either

FLOAT_TYPE'EPSILON or FLOAT_TYPE'BASE'EPSILON received much discussion. Both approaches have their advantages.
The advantage of the EPSILON formulation can be summarized as:

* it permits improved efficiency of execution (the approximations are better matched to the requested precision)

The advantages of the BASE'EPSILON formulation can be summarized as:

* it requires implementations to deliver the accuracy available on the hardware

* it permits the analysis of the accuracy of a program using GENERIC_ELEMENTARY_FUNCTIONS in the same way that we can analyze for the arithmetic operations

* it improves efficiency of implementation (only a few approximations are needed to cover all available precisions)

The Working Groups have agreed to propose the BASE'EPSILON formulation for standardisation. This implies that for every function the MRE will be formulated in terms of a factor times the BASE'EPSILON value of the floating-point type. For all functions except the "**" operator, the factor will be some function-dependent constant. For the "**" function the factor is not a constant, but depends on the arguments.
To verify whether a function meets its accuracy requirements, we will express the MRE also in BASE'EPSILON. Besides, we will state what is the accuracy constraint for every function and whether the measured accuracy is as required or not.

2.2. Special tests

The accuracy of the elementary functions is not the only aspect that should be tested. The behaviour of the functions for specific valid arguments and some basic analytical properties should also be examined. Naturally, special tests may differ from function to function. For this reason it is an impractical task to set up a global scheme for these tests. Instead the decision which supplemental tests have to be added will be taken for each function separately. Chapter 5 contains an exhaustive survey of these special tests. Here a selection of frequently occurring additional tests is mentioned:

* tests for odd or even symmetry of a function
* for odd functions, comparing function values with their arguments for small arguments
* tests for large arguments for which a substantial amount of argument reduction has to take place
* tests for the behaviour of a function in the neighbourhood of horizontal or vertical asymptotes.

2.3. Error tests

Up to now all tests dealt with correct function arguments. However, sometimes the submitted arguments are not acceptable to a function. In such a case the function should be able to detect the invalid arguments and then perform some appropriate actions. An example of this is an argument outside the function domain in which case the exception ARGUMENT_ERROR should be raised.
Beside this exception there are other events after which a function can not terminate normally. A well-known example of this is overflow during the computation of a function value, possibly in intermediate results. In Ada, an exception might then be raised implicitly. In the case of overflow the predefined exception NUMERIC_ERROR (or possibly in the future the predefined exception

CONSTRAINT_ERROR) might be raised.

For every function the error tests consist of a number of function calls with special selected arguments. Some of the arguments should trigger error situations. They are chosen such that they lie outside the function domain or cause overflow. The function that is called should be able to signal these exceptional conditions.

Other critical arguments are chosen such that they should not cause any trouble. Bounds belonging to the function domain and for some functions the largest or smallest representable floating-point number are examples of valid arguments that should not raise any exception. For such arguments a function should terminate normally and return correct values.

3. TEST OBJECTS

For the implementation of the elementary functions several specific Ada concepts have been used. First the elementary functions are grouped in a package named GENERIC_ELEMENTARY_FUNCTIONS. In the second place the package has been made generic with floating-point type FLOAT_TYPE as a formal parameter. The arguments and the function values of all elementary functions are of this type. By instantiating the generic package with a predefined or user-defined floating-point type a collection of basic functions can be obtained of any desired precision up to 35 digits.

Finally the package provides an exception named ARGUMENT_ERROR that will be raised when a function argument lies outside the function domain. The exception is a renaming of an exception, also called ARGUMENT_ERROR, that is defined in the package ELEMENTARY_FUNCTIONS_EXCEPTIONS.

The specification of the generic package of elementary functions is as follows:

```
with ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <>;
package GENERIC_ELEMENTARY_FUNCTIONS is
- -
- - Declare the basic mathematical functions
   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;
   - -
   - - Declare exception, to be raised for implementation-independent range constraints
   ARGUMENT_ERROR : exception renames
                        ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```

4. THE TEST PACKAGE

For the testing of the elementary functions a number of test programs are provided. All the test programs are presented in the form of parameterless procedures that have been collected into a generic package. The package, named TEST_ELEMENTARY_FUNCTIONS_GENERIC, has two generic parameters. The first one is a floating-point type FLOAT_TYPE and the second one an object of type BOOLEAN, named BASE_EPSILON_WANTED. The latter has a default value TRUE.

At instantiation of the test package one should use the same floating-point type as actual parameter that is used to instantiate the generic package of elementary functions. When the results of the accuracy tests need to be expressed in FLOAT_TYPE'EPSILON rather than in FLOAT_TYPE'BASE'EPSILON, a second actual parameter should be supplied which has the value FALSE (see section 2.1.3). If the second parameter is omitted the default value TRUE is used and the accuracy tests are expressed in FLOAT_TYPE'BASE'EPSILON. The full specification of the generic test package is:

```
generic
  type FLOAT_TYPE is digits <>;
  BASE_EPSILON_WANTED : BOOLEAN := TRUE;
package TEST_ELEMENTARY_FUNCTIONS_GENERIC is
  procedure TEST_SQRT;
  procedure TEST_LOG;
  procedure TEST_EXP;
  procedure TEST_POWER;
  procedure TEST_SIN_COS;
  procedure TEST_SIN_COS_1;
  procedure TEST_SIN_COS_360;
```

```
   procedure TEST_TAN_COT;
   procedure TEST_TAN_COT_1;
   procedure TEST_TAN_COT_360;
   procedure TEST_ARCSIN_ARCCOS;
   procedure TEST_ARCTAN_ARCCOT;
   procedure TEST_SINH_COSH;
   procedure TEST_TANH_COTH;
   procedure TEST_ARCSINH_ARCCOSH;
   procedure TEST_ARCTANH_ARCCOTH;
end TEST_ELEMENTARY_FUNCTIONS_GENERIC;
```

4.1. Auxiliary subprograms

In the test programs the accuracy test part, the special test part and the error test part all need the support of additional subprograms. The accuracy tests use a random number generator, the special tests often use information about the floating-point arithmetic and the error tests use a special procedure to intercept and handle exceptions. Moreover, many tests express the results in the form of the fraction of base β digits that is lost in the computations, with β the radix of the floating-point representation. This requires some knowledge of the floating-point numbers.
Other subprograms that have to be supplied are the I/O-procedures for integer and floating-point types. In Ada these procedures are provided in the form of generic procedures, which have to be instantiated by the user with the appropriate actual types.

4.1.1. EXTENDED_S01BA.

EXTENDED_S01BA is a package that has been built upon GENERIC_ELEMENTARY_FUNCTIONS. It combines the elementary functions with comprehensive error handling facilities. Comparison of the specification of EXTENDED_S01BA with that of GENERIC_ELEMENTARY_FUNCTIONS shows that in the former every elementary function has an additional parameter FAIL of type ERROR_RECORD. Although the name of this type suggests that it concerns a record type the actual representation of objects of type ERROR_RECORD is hidden from the user. The type, that is declared in NAG_P01AA, is a so-called *private type*. This means that beside a number of basic operations like equality, inequality and assignment, only functions and procedures declared in the specification of NAG_P01AA can be used to manipulate objects of this type. For convenience the term *error record* will be used in the rest of this paper to refer to an object of type ERROR_RECORD.

Error records are very useful when program control cannot terminate normally due to some exceptional situation. An error record contains a lot of information on the sort of error and the way to deal with it. It indicates, among other things, what kind of error has occurred, the error condition, the library unit in which the exception was raised and the unit that called the subprogram in which the error occurred. After creating an error record through an appropriate procedure call the error record has a default value which is initially 'blank'. When an elementary function is called it first checks whether the error record permits continuation. If it does and the execution of the rest of the function can terminate normally, the error record will remain unchanged. However, if the execution of the function is interrupted to signal an exception the contents of an error record will be changed. Depending on its initial value the exception will be propagated or special values will be returned in which case the execution of a program can proceed normally.
Since an error record is the last parameter in the specification of an elementary function and default values are present, it may be omitted in a function call.

The specification of the package EXTENDED_S01BA is as follows:

```
package EXTENDED_S01BA is
  function SQRT (X        : FLOAT_TYPE;
                 FAIL     : ERROR_RECORD := DEFAULT_RECORD) return FLOAT_TYPE;
  function LOG  (X        : FLOAT_TYPE;
                 FAIL     : ERROR_RECORD := DEFAULT_RECORD) return FLOAT_TYPE;
  function LOG  (X, BASE  : FLOAT_TYPE;
                 FAIL     : ERROR_RECORD := DEFAULT_RECORD) return FLOAT_TYPE;
                 .
                 .
  -- other elementary functions
                 .
                 .
  function ARCTANH (X     : FLOAT_TYPE;
                    FAIL  : ERROR_RECORD := DEFAULT_RECORD) return FLOAT_TYPE;
  function ARCCOTH (X     : FLOAT_TYPE;
                    FAIL  : ERROR_RECORD := DEFAULT_RECORD) return FLOAT_TYPE;
end EXTENDED_S01BA;
```

To test the elementary functions the functions defined in this package will be used. The additional parameter is omitted in the accuracy tests and the special tests. On the other hand the error tests can be executed more easily when the functions of EXTENDED_S01BA are used. Now the testing of the functions for arguments that (might) cause trouble is nothing more than creating an error record or clearing an existing one and then calling the desired function with an additional parameter. For convenience default values are assigned to the fields of the error record except for the field containing the name of the calling unit. A consequence of choosing default values is that an exception that has been raised by an elementary function will be propagated until it is caught by a user-provided exception handler. The subprogram containing this handler is explained in section 4.1.6.
Although it is possible, by choosing appropriate initial values for the fields of an error record, to obtain function results under any circumstances, this approach has been rejected. The error tests are intended to demonstrate the capability of the elementary functions to signal errors and not the capability of the extended functions to handle errors.

4.1.2. INIT_MACHAR *and* MACHAR.

INIT_MACHAR and MACHAR are two packages that supply information about the characteristics of the floating-point arithmetic. To be more specific: the package MACHAR, which only needs a specification, contains environmental constants that are dynamically determined by calling the appropriate functions provided by the package INIT_MACHAR. This construction of two packages offers the opportunity to obtain a set of constants which are fixed for every floating-point type. The following parameters are presented by MACHAR:

IBETA -- an integer denoting the radix of the floating-point representation
IT -- an integer denoting the number of base IBETA digits in the mantissa of a floating-point number
IRND -- an integer denoting whether floating-point addition chops or rounds
NGRD -- an integer denoting the number of guard digits used in multiplication
MACHEP -- the largest negative integer such that $1 + \text{IBETA}^{\text{MACHEP}} \neq 1$, except that MACHEP is bounded below by $-(\text{IT} + 3)$
NEGEP -- the largest negative integer such that $1 - \text{IBETA}^{\text{NEGEP}} \neq 1$, except that NEGEP is bounded below by $-(\text{IT} + 3)$

IEXP -- an integer denoting the number of bits (or decimal places when IBETA = 10) reserved for the representation of the exponent of a floating-point number, including sign bit
MINEXP -- the smallest negative integer such that $\text{IBETA}^{\text{MINEXP}} > 0$
MAXEXP -- an integer denoting the largest positive exponent in the representation of a floating-point number
EPS -- the smallest positive floating-point number such that $1 + \text{EPS} \neq 1$
EPSNEG -- a small positive floating-point number such that $1 - \text{EPSNEG} \neq 1$. In particular, $\text{EPSNEG} = \text{IBETA}^{\text{NEGEP}}$ or $\text{EPSNEG} = \text{IBETA}^{\text{NEGEP}/2}$ depending on the values of IBETA and IRND
XMAX -- the largest representable floating-point number
XMIN -- the smallest positive representable floating-point number

Many computations to determine the values for these parameters are transcriptions in Ada of parts of an updated version of the Fortran subroutine MACHAR as mentioned in the *Software manual.*

It is not strictly necessary to include all above-mentioned parameters in the package MACHAR. Several can be omitted since they can be accessed directly in Ada by means of attributes. However, this approach has not been followed for two reasons. In the first place there already existed a (Fortran) program that determined environmental parameters, of which some could be obtained by attributes but other could not. Second, on one of our Ada machines some of the values acquired through attributes appeared to be completely wrong. For instance, if IBETA would have been gained by the use of an attribute, it would have been two, pretending a binary representation whereas in fact this machine is hexadecimal. For reasons of reliability and clarity all parameters are computed without the use of attributes.

Many algorithms in INIT_MACHAR fail whenever floating-point types are examined having lower precision than a built-in floating-point type. As mentioned before, in Ada such user-defined types are, invisibly to the user, mapped on predefined types. All computations are performed in the precision of the base type after which the result is converted to the user-defined type. As a consequence of this mapping it will not always be the environmental parameters of the user-defined type that are determined, but instead those of the base type. A parameter like IBETA will still be computed correctly, but most other parameters are computed for the predefined type. To solve this problem a second version of the algorithms of INIT_MACHAR has been made available which is used for such user-defined types. In this version almost every parameter is determined either by hand or by a very simple algorithm.
Since all computations are performed in an extended precision, a simulation of a binary machine can be made. That is, the radix IBETA can be fixed to two, independent of the radix of the actual representation, and the number of bits in the mantissa, IT, can be derived from the decimal precision. IRND and NGRD can safely be fixed to one. The rest of the parameters are determined from these parameters and straightforward programming. The parameters gained in this way are tuned to one another, so they can be applied in any expression instead of the values given by attributes.
The choice in which way the parameters are computed, is made internally depending of the precision of the floating-point type in question.

4.1.3. RAN *and* RANDL

Any accuracy test of an arbitrary elementary function is performed by sampling a number of arguments from an interval and then checking some mathematical identity. To form a good picture of the behaviour of the function on this interval, the arguments should be drawn at random. For this purpose a random generator called RAN has been used that generates random numbers uniformly distributed over (0, 1). Our random generator is the one suggested in the *Software manual.* In the Ada version it consists of a package RAN_SEED containing an integer seed for the random generator, and a

parameterless function RAN. The generated numbers are random up to 29 bits. For the floating-point types on which we have performed tests, this suffices. Floating-point types with a larger mantissa can also use this random generator on the condition that not all lower bits are zero. If this turns out to be the case another random generator should be used. One possibility to obtain larger random numbers is to use one or more 29 bits random numbers and some scaling to produce these numbers.

Sometimes the use of a random generator producing uniformly distributed numbers, leads to an unbalanced partition of the test arguments over the test interval. Especially for large intervals this can play an important role. For small intervals the difference between a uniform distribution and a logarithmic distribution has less importance. The logarithmic random generator, RANDL, is composed of the exponential function and RAN. If $x = a*e^{y*log(b/a)}$, y uniformly distributed over (0, 1), x will be logarithmically distributed over (a, b). The factor $log(b/a)$ has to be passed as parameter to RANDL.
With one restriction, the numbers generated by RANDL are random to the precision of the exponential function. However, if the precision of a floating-point number exceeds 29 bits and the logarithm of a random number, that is generated by RANDL, is required, the use of RANDL becomes less meaningful and therefore should be avoided.

4.1.4. INTEGER_IO *and* FLOAT_IO.

For the input and output of integer values an instance, named INTEGER_IO, has been made of the generic package INTEGER_IO, which can be found in the standard package TEXT_IO. The integer type that has been used as actual parameter, is the predefined type INTEGER.
The input and output procedures for the floating-point type are obtained by instantiating the generic package FLOAT_IO, which can be found in TEXT_IO as well. The result is a package, named FLOAT_IO, that contains these procedures for the demanded floating-point type.

4.1.5. TEST_AUX_TYPES.

The package TEST_AUX_TYPES is an auxiliary subprogram that contains additional types required by the test programs. The tests as presented here only need one extra type. This is the type FUNCTION_TYPE that is used in the error tests. The error tests for all elementary functions are performed through the procedure ERROR_TEST, that will be discussed in detail in the next subsection. Since ERROR_TEST needs to know on which function the error test has to be performed, some additional information must be supplied. This has been done in the form of a parameter of type FUNCTION_TYPE.

4.1.6. ERROR_TEST.

As stated above the procedure ERROR_TEST is used by all elementary functions to monitor the effect of special or exceptional arguments. It contains an exception handler that catches and handles exceptions that are possibly raised. If an exception is raised, it is handled by reporting the status of the error record. The advantage of the use of this procedure may be clear: now it is not necessary to provide an exception handler for each situation in which an exception might occur. Instead of this we can just call ERROR_TEST with the appropriate parameters and look whether or not a function value is returned.
The full specification of ERROR_TEST is:

```
procedure ERROR_TEST (F     : FUNCTION_TYPE;
                      TITLE : STRING;
                      ARG1  : FLOATING;
                      ARG2  : FLOATING := TWO_PI;
                      ARG3  : FLOATING := TWO_PI;
                      FAIL  : ERROR_RECORD := DEFAULT_RECORD);
```

The meaning of the parameters is:

F	--	an indicator of which function must be called
TITLE	--	some extra comment that can be added to the error report
ARG1, ARG2, ARG3	--	the arguments for the function; if a function has less than three parameters the other(s) can be omitted
FAIL	--	an error record that is passed to the function and might be changed if an exception is raised, depending on its initial state

4.1.7. Portability and modifiability of the test package.

One of the design goals of the PIA project was to create a library of standard functions that was highly portable. To verify whether the elementary functions are really portable, they ought to be tested by the same test programs on every machine where they have been installed. A consequence of this is that test programs should also be portable to a large extent. To meet this requirement a set of transportable, self-contained test programs has been designed and implemented.

To enhance the flexibility of the test package the bodies of several subprograms have been replaced by body stubs. The bodies of every test procedure, EXTENDED_S01BA, INIT_MACHAR and the random generator RAN have been made separate from the body of TEST_ELEMENTARY_FUNCTIONS_GENERIC and each of them can be compiled separately. Whenever parts of the test programs, EXTENDED_S01BA, INIT_MACHAR, or RAN need to be modified, it is sufficient to recompile the library units that have been changed and to load again the main program that calls test procedures.
As mentioned the function body of RAN has also been made separate. By so doing it is fairly simple to replace this random generator by some other if this is desired. Naturally the specification of this function ought to remain unchanged. Hence the first few lines of this separate unit should always contain:

```
separate (TEST_ELEMENTARY_FUNCTIONS_GENERIC)
function RAN return FLOAT_TYPE is
```

The consequence of altering the specification is that all occurrences of RAN in the test programs must be changed as well. The use of parameters in RAN can be circumvented by using the integer variable SEED in RAN_SEED or, if necessary, by adding other variables to this package.

5. FUNCTION TESTS

The subsequent sections present the tests that have been used to examine the various standard functions. For each function they include accuracy tests, special tests and error tests.
The accuracy test part describes the mathematical identities and the intervals that have been selected to determine the accuracy of the functions. The use of every identity will be explained by indicating its purpose. Further, the suitability of an identity is shown by demonstrating its numerical stability through a rough estimate of the error that can be expected.
The special test part describes the supplemental tests used to examine some function-dependent properties. Besides mentioning these tests and their purposes it will also be stated what results are to be

expected.
Finally, in the error test part arguments are presented that may lead to erroneous conditions in which an exception can be raised. In the case of an exception the status of the error record will be indicated, that can be expected after an exception has been handled appropriately (starting with the default error record and the default handler). For tests on overflow the assumption has to be made that the exception NUMERIC_ERROR (or possibly in the future CONSTRAINT_ERROR) is raised in overflow situations for the floating-point types and integer types in question. If a machine, instead of raising NUMERIC_ERROR, returns incorrect values in situations where numbers are too large to be represented, the tests for overflow become useless and should be neglected.
All sections start by naming the procedure that must be called to execute the tests for a specific function. Further, in all sections β will refer to the radix in the representation of a floating-point number. The identifiers in italics like *it*, *xmin* and *xmax* correspond with the parameters provided by MACHAR (see section 4.1.2).

5.1. SQRT

The tests for the square root function are performed by invoking the procedure TEST_SQRT.

The accuracy tests are based on the following mathematical identities and intervals:
Test 1: $\sqrt{x^2} = x$ on $(1/\sqrt{\beta}, 1)$ and
Test 2: $\sqrt{x^2} = x$ on $(1, \sqrt{\beta})$.
The first test determines the accuracy of the square root when the argument lies in the primary range and no argument reduction is necessary. The second test uses the same identity, but on an interval where the argument has to be reduced. Moreover the effect is measured of the multiplication by $1/\sqrt{\beta}$ in the algorithm of the SQRT, due to the odd exponent in the base β representation of the arguments.
Both tests are extensively described in the *Software manual* and therefore more details can be found there.

The special tests consist of six function calls with the following arguments: 0.0, *xmin*, *xmax*, $1.0 - \epsilon_1$, 1.0 and $1.0 + \epsilon_2$ where $\epsilon_1 = epsneg$ and $\epsilon_2 = eps$ (see section 4.1.2). The square root of zero should be equal to zero. The computation for the arguments *xmin* and *xmax* should terminate normally in both cases. The computation for the arguments close to one and one itself should return one or near-one in all cases.

The error tests consist of two function calls with the arguments zero and minus one. The argument zero should not trigger an error situation. The second argument should trigger an error. Although the type of an error record is not a record type but a private type, we will give the status of an error record in the form of a record aggregate. For the argument -1.0 the error record should state (1, "SQRT", "ARGUMENT ERROR") for the error number, the library unit in which the exception was raised and the error category, respectively.

5.2. LOG

The tests for the logarithm are performed by invoking the procedure TEST_LOG.

The accuracy tests are based on the following mathematical identities and intervals:
Test 1: $log(x) = -\sum_{k=1}^{\infty} \frac{(1-x)^k}{k}$ on $(1-\epsilon, 1+\epsilon)$, where $\epsilon = \beta^{-it/3}$,
Test 2: $log(x) = log(\frac{17}{16}x) - log(\frac{17}{16})$ on $(1/\sqrt{2}, 15/16)$,

Test 3: ${}^2log(x) = {}^2log(\frac{17}{16}x) - {}^2log(\frac{17}{16})$ on $(1/\sqrt{2}, 15/16)$,

Test 4: ${}^{10}log(x) = {}^{10}log(\frac{11}{10}x) - {}^{10}log(\frac{11}{10})$ on $(1/\sqrt{10}, 9/10)$ and

Test 5: $log(x^2) = 2\,log(x)$ on (16, 240).

The series expansion in the first test is truncated after four terms.

The first test determines the accuracy of the logarithm for arguments close to one where the function is fairly sensitive to small errors in the argument. The second test demonstrates the behaviour of the function on the primary interval. The third test is the analogue of the second one for the base 2 logarithm and the fourth test for the base 10 logarithm. In both the third test and the fourth test the choice of the interval is such that the arguments lie in the primary range. The last test measures the error in the logarithm for arguments outside this range, in particular when x and x^2 have different exponents in the floating-point representation.

Test 1, 2, 4 and 5 are fully described in the *Software manual* and therefore we refer to this source for more details. The third test is new, but in fact all comments made for the second test are also applicable to this case. The only difference is that wherever *log* occurs, one should substitute ${}^2log$.

The special tests start with a cursory check of the property $log(x) = -log(1/x)$ for x in the range (15, 17). The results of this test should be zero or negligible with respect to the floating-point precision. Second is a series of function calls with arguments 1.0, *xmin* and *xmax* and with argument pairs (1.0, 2.0) and (1.0, 10.0). In both argument pairs the first argument denotes the argument of the LOG function and the second one the base. The function should return zero for the argument one and both argument pairs and terminate normally for the other, returning proper values. These additional tests can also be found in the *Software manual.*

The error tests consist of a series of function calls which all should trigger an error situation. The arguments are presented to the function in pairs (x, b), where x denotes the argument of the logarithm and b the base. The argument pairs are: $(-2.0, e)$, $(0.0, e)$, $(8.0, -1.0)$, $(8.0, 0.0)$ and $(8.0, 1.0)$. The first two pairs cause an ARGUMENT_ERROR due to non-positive function arguments. The other pairs will also cause an ARGUMENT_ERROR, this time due to an illegal value for the base of the logarithm. The status of the error record after catching the exception can be read from table 5.1.

Arguments		Error record		
x	*base*	error number	library unit	error category
-2.0	e	1	LOG	argument error
0.0	e	1	LOG	argument error
8.0	-1.0	2	LOG	argument error
8.0	0.0	2	LOG	argument error
8.0	1.0	3	LOG	argument error

TABLE 5.1 Error diagnosis for LOG

5.3. EXP

The tests for the exponential function with base e are performed by invoking the procedure TEST_EXP.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $e^{x-1/16} = e^x * e^{-1/16}$ on $(1/16 - log(\sqrt{2}), log(\sqrt{2}))$,

Test 2: $e^{x-45/16} = e^x * e^{-45/16}$ on $(log(4*xmin*\beta^{it}), -5\,log(2))$ and

Test 3: $e^{x-45/16} = e^x * e^{-45/16}$ on $(10\,log(2), q)$, where $q = max(100\,log(2), log(0.9\,xmax))$.
These three tests have been extracted from the *Software manual* where they were intended to examine the exponential function on respectively the primary interval, an interval with values in the neighbourhood of the smallest value acceptable to the function and an interval with values in the neighbourhood of the largest acceptable function argument. However, the algorithm used to compute the exponential function differs from the algorithm mentioned in the *Software manual*, so that primary ranges do not agree. Consequently, using the first test unaltered, may result in values for the MRE and RMS that are slightly larger than those reported by the *Software manual*. The other tests should give comparable results. More specific details can be found in the *Software manual*.

The special tests start with a cursory check of the property $e^x * e^{-x} = 1$ for x in the range $(0, \beta)$. The results of this test should all be zero or near-zero with respect to the floating-point precision.
Second is a series of function calls with special arguments: 0.0, an argument very close to the smallest acceptable input value and an argument very close to the largest acceptable input value. In the first case EXP should return 1.0, whereas it should return proper function values for the other arguments.
Finally there is a test to detect systematic errors. In the tests above systematic errors will go unnoted whereby the calculated function value is too large by a constant factor. To check for this situation the identity $e^x = (e^{x/2})^2$, $x = 0.5*[log(xmax)]$ is used, where [ ] denotes the entier function. In the absence of systematic errors both sides should be equal.

The error tests consists of two function calls of which the second should trigger an error situation. The arguments of these calls are $-1.0/\sqrt{xmin}$ and $1.0/\sqrt{xmin}$ respectively. The first argument should result in zero or a value close to the smallest positive floating-point number, whereas the second one should cause overflow. The status of the error record after the function has been called with the argument $1.0/\sqrt{xmin}$ should state (3, "EXP", "OVERFLOW ERROR").

*5.4. "**"*

The tests for the "**" operator are performed by invoking the procedure TEST_POWER.

The accuracy tests are based on the following mathematical identities and intervals:
Test 1: $x^{1.0} = x$ on $(1/\beta, 1)$,
Test 2: $(x^2)^{1.5} = x^3$ on $(1/\beta, 1)$,
Test 3: $(x^2)^{1.5} = x^3$ on $(1, e^{log(xmax)/3})$ and
Test 4: $x^y = (x^2)^{y/2}$, $x \in (0.01, 10)$ and $y \in (-c, c)$,
where $c = min(log(xmax), -log(xmin))/log(100)$.
The first test determines the accuracy of the power function algorithm for an exponent 1.0, i.e. the accuracy of $e^{log(x)}$. The effect of the additional error induced by the non-trivial multiplication in forming the intermediate result $y*log(x)$ for moderate y is measured in the second test. In the third test substantial argument reduction is required for large values of the base. This may increase the error in forming $log(x)$ and $e^{y\,log(x)}$. The last test is the most demanding. In this test the power varies as well. Consequently, this may even increase the error in $y*log(x)$.
All tests are described in the *Software manual*, so that more specific details can be found there.

The special tests start with a cursory check on the property $x^y = (1/x)^{-y}$ for both x and y in (1, 11). The error, measured relatively to the result of x^y, should be zero or at least be small with respect to the floating-point precision.
The equality $x ** 0.0 - 1.0 = 0.0$, $x > 0.0$, is used for the arguments FLOAT_TYPE'SMALL, 1.0 and FLOAT_TYPE'LARGE. For all arguments the results should be zero.
The same arguments are used to verify the equality $0.0 ** x = 0.0$, $x > 0.0$. Again the results should be zero.

Finally these arguments and their opposites are used to check the equality $1.0 ** y - 1.0 = 0.0$. Also in this case all results should be zero.

The error tests consist of a series of function calls of which some should trigger an error condition. The following argument pairs are presented to the function (the first argument denotes the exponent and the second the base): (*minexp*, β), (*maxexp* + 1, β), (2.0, 0.0), (2.0, −2.0), (−2.0, 0.0), (0.0, 0.0) and (*xmax*, *xmax*). The first three pairs are perfectly legal and should deliver correct function values. The base in the fourth pair is negative and should cause an ARGUMENT_ERROR. The fifth pair deals with a zero base, a boundary value for the base, and a negative exponent, which should also cause an ARGUMENT_ERROR. In the sixth pair both the exponent and the base are zero, the exponent being the largest unacceptable value for this base. Again ARGUMENT_ERROR should be raised. Finally, the last pair should clearly result in overflow.

The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record given in table 5.2.

Arguments		Error record		
x	*base*	error number	library unit	error category
minexp	β	-	--	--
maxexp + 1	β	-	--	--
2.0	0.0	-	--	--
2.0	-2.0	1	EXP	argument error
-2.0	0.0	2	EXP	argument error
0.0	0.0	2	EXP	argument error
xmax	*xmax*	3	EXP	overflow error

TABLE 5.2 Error diagnosis for EXP

5.5. SIN

The tests for the sine function are performed by invoking one of the procedures:

TEST_SIN_COS for the sine with cycle 2π,
TEST_SIN_COS_1 for the sine with cycle 1 and
TEST_SIN_COS_360 for the sine with cycle 360.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $sin(x) = 3\,sin(x/3) - 4\,sin^3(x/3)$ on $(0, cycle/4)$ [test 1 in the test procedures],
Test b: $sin(x) = 3\,sin(x/3) - 4\,sin^3(x/3)$ on $(3\,cycle, 3.25\,cycle)$ [test 3 in the test procedures] and
Test c: $sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$ on $(0, \pi/4)$ [test 5 in TEST_SIN_COS].

The Taylor series in test c is truncated after $m + 1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [ ] denotes the entier function.

In the *Software manual* the first test was intended to test the function over the primary range in which case no argument reduction is necessary. However, our implementation uses another primary range and consequently the computation of $sin(x)$ can involve some argument reduction. Nevertheless, the MRE and RMS should be comparable to those obtained by the algorithms in the *Software manual.* The second test measures the effect of the argument reduction for arguments outside the primary range. Both tests can be found in the *Software manual.*

The third test is new. It examines the function over the primary range. The error introduced by

truncating the series after $m+1$ terms, is bounded in magnitude by $\frac{x^{2m+3}}{(2m+3)!}$, m defined as above. Hence the following rough estimate for the relative error between $sin(x)$ and the truncated Taylor series can be made:

$$0 \leqslant \frac{\left| sin(x) - \sum_{k=0}^{m} \frac{(-1)^k x^{2k+1}}{(2k+1)!} \right|}{sin(x)} \leqslant \frac{\frac{x^{2m+3}}{(2m+3)!}}{sin(x)} \leqslant 2\frac{\frac{x^{2m+3}}{(2m+3)!}}{x} \leqslant 2\frac{(\pi/4)^{2m+2}}{(2m+3)!}$$

This should be much less than the rounding error for the floating-point type under consideration. Therefore, this series can safely be used. By evaluating carefully it is possible to limit the additional error due to the computation of the series to one rounding.
However, if the algorithm for the sine function uses the same Taylor series, systematic errors might go undetected. In such cases no great value should be attached to the results.

The special tests of the procedures all contain a cursory check on the parity of the function. The parity check is performed by sampling a handful of arguments from the interval (0, 6 *cycle*). Good implementations should return a zero value for each argument.
The equalities below will be examined for three different values of *cycle*: 2π, 1.0 and 360.0. The equality sin(*x*, *cycle*) = 0.0, $x = k * cycle / 2.0$ and k integer, is verified for small positive values of k. When *cycle* is exactly representable, i.e. when *cycle* = 1.0 or *cycle* = 360.0, the results must be zero. Otherwise the results should be zero or at least a small multiple of the machine epsilon.
The equality sin(*x*, *cycle*) − 1.0 = 0.0, $x = (4k+1) * cycle / 4.0$ and k integer, is checked for small values of k. Also here the results should be zero or at least a small multiple of the machine epsilon whenever *cycle* is not exactly representable.
The equality sin(*x*, *cycle*) + 1.0 = 0.0, $x = (4k+3) * cycle / 4.0$ and k integer, is examined for small values of k. The same results as in the previous test can be expected here.
The following tests only apply to the procedure TEST_SIN_COS. To assure that the sine uses the correct cycle the equation $(sin(a+c) - sin(a-c))/(c+c)$, $a = 6\pi$ and $c = \beta^{-it/2}$, is used. The result must approximately be one.
The equation $sin(x) - x \approx 0$, $|x| \ll 1$, is used to compare function values with their arguments for small arguments. The results of this test should be small with respect to the working precision. Another, very small, argument is used to check for underflow during the evaluation of $sin(x)$. Strictly speaking this test is redundant in Ada, since this language does not consider underflow to be exceptional. The function call with the single argument zero must return zero. Finally, for three consecutive large arguments the sine is computed. Although the relative errors in the arguments are small, the calculated function values may differ considerably.

There is only one error test, which appears in the procedure TEST_SIN_COS. The function call with argument one and cycle zero should trigger an error condition. For these arguments the error record should state (1, "SIN", "ARGUMENT ERROR").

5.6. COS

The tests for the cosine function are performed by invoking one of the procedures:

TEST_SIN_COS	for the cosine with cycle 2π,
TEST_SIN_COS_1	for the cosine with cycle 1 and
TEST_SIN_COS_360	for the cosine with cycle 360.

The accuracy tests are based on the following mathematical identities and intervals:
Test a: $cos(x) = 4\,cos^3(x/3) - 3\,cos(x/3)$ on (0.5 *cycle*, 0.75 *cycle*) [test 2 in the test procedures],
Test b: $cos(x) = 4\,cos^3(x/3) - 3\,cos(x/3)$ on (3.5 *cycle*, 3.75 *cycle*) [test 4 in the test procedures] and

Test c: $cos(x) = \sum_{k=0}^{\infty} \frac{(-x)^{2k}}{(2k)!}$ on $(0, \pi/4)$ [test 6 in TEST_SIN_COS].

The Taylor series in test c is truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [ ] denotes the entier function.

Analogously to the sine function, the same identity is used to examine the cosine on two intervals differing an integer multiple of *cycle*. Both tests measure the effect of the argument reduction for arguments that do not lie too far outside the primary range. The analysis for the second test is identical to the one for the first test and can be found in the *Software manual.*

The third test examines the function over the primary interval. The error introduced by truncating the series after $m+1$ terms is bounded in magnitude by $\frac{x^{2m+2}}{(2m+2)!}$, m defined as above. Hence the following rough estimate for the relative error between $cos(x)$ and the truncated Taylor series can be made:

$$0 \leqslant \frac{\left| cos(x) - \sum_{k=0}^{m} \frac{(-x)^k}{(2k)!} \right|}{cos(x)} \leqslant \frac{\frac{x^{2m+2}}{(2m+2)!}}{cos(x)} \leqslant 2\frac{\frac{x^{2m+2}}{(2m+2)!}}{x} \leqslant 2\frac{(\pi/4)^{2m+1}}{(2m+2)!}$$

This should be much less than the rounding error for the floating-point type under consideration. Thus the error due to truncating the Taylor series, will not unduly magnify the measured error in this test. If the series is evaluated carefully, the additional error made in this evaluation can be limited to one rounding.

Again, the results of this test are of little value whenever the algorithm that computes the cosine uses the same series. In this case systematic errors might go undetected and the results may be too optimistic.

The equalities below will be examined for three different values of *cycle*: 2π, 1.0 and 360.0. The equality cos(*x*, *cycle*) − 1.0 = 0.0, $x = k * cycle$ and k integer, is verified for small positive values of k. When *cycle* is exactly representable, i.e. when *cycle* = 1.0 or *cycle* = 360.0, the results must be zero. Otherwise the results should be zero or at least a small multiple of the machine epsilon.

The equality cos(*x*, *cycle*) = 0.0, $x = (2k+1) * cycle / 4.0$ and k integer, is checked for small values of k. Also here the results should be zero or at least a small multiple of the machine epsilon whenever *cycle* is not exactly representable.

The equality cos(*x*, *cycle*) + 1.0 = 0.0, $x = (2k+1) * cycle / 2.0$ and k integer, is examined for small values of k. The same results as in the previous test can be expected here.

The procedure TEST_SIN_COS has some additional tests. There is a test that verifies the parity of the cosine on the interval $(0, 6\pi)$ cursorily. Good implementations should return a zero value for each argument. Finally, the function call with the single argument zero should deliver the value one.

TEST_SIN_COS contains the only error test. Analogously to the sine function, a zero cycle is presented to the cosine, which should trigger an error condition. For these arguments the error record should state (1, "COS", "ARGUMENT ERROR").

5.7. TAN

The tests for the tangent function are performed by invoking one of the procedures:

TEST_TAN_COT for the tangent with cycle 2π,
TEST_TAN_COT_1 for the tangent with cycle 1 and
TEST_TAN_COT_360 for the tangent with cycle 360.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $tan(x) = \frac{2\,tan(x/2)}{1 - tan^2(x/2)}$ on $(0, cycle/4)$,

Test 2: $tan(x) = \frac{2\,tan(x/2)}{1 - tan^2(x/2)}$ on (0.4375 *cycle*, 0.5625 *cycle*) and

Test 3: $tan(x) = \frac{2\,tan(x/2)}{1 - tan^2(x/2)}$ on (3 *cycle*, 3.125 *cycle*),

The first test determines the accuracy of the tangent on an interval where at most one step is required in the argument reduction. Since our implementation uses a primary range that is only about half as large as the one in the algorithm included in the *Software manual*, an additional argument reduction in the computation of $tan(x)$ might be necessary. Nevertheless, the results of this test may exceed those reported by the *Software manual* only slightly. In the second test the arguments are larger and their reduction requires one or two steps. Besides, the test interval is selected such that $|tan(x/2)| \geq 1$. The third test requires substantial argument reduction for both $tan(x)$ and $tan(x/2)$. Like in the first test the function values are bounded above by 1.
These three tests are also mentioned in the *Software manual*, so more specific details can be found there.

The test procedures start with a check on the parity of the function. This test is performed by sampling a handful of arguments from the interval (0, 3 *cycle*). Good implementations should return a zero value for each of the arguments.
The equality tan(x, *cycle*) = 0.0, $x = k * cycle / 2.0$, and k integer, is verified for small positive values of k. The results must be zero when *cycle* is exactly representable and zero or a small multiple of the machine epsilon otherwise.
TEST_TAN_COT supplies checks on other properties of the function as well. The equation $tan(x) - x \approx 0$, $|x| << 1$, is used to compare function values with their arguments for small arguments. The results of this test should be zero or at least small with respect to the working precision. Another, very small, argument is used to check for underflow during the evaluation of $tan(x)$, although in Ada this test is redundant.
Then there is an extra test to illustrate the loss in significance for large arguments in the neighbourhood of an asymptote.
Finally, the function is called with a single argument zero in which case the function must return zero as well.

The error tests for the procedure TEST_TAN_COT consist of two function calls of which at least one should trigger an error. The argument pair (0.5, 0.0), an argument 0.5 and a cycle 0.0, should result in an ARGUMENT_ERROR. The argument $\pi/2$ might, but does not have to, result in an overflow error depending on the accuracy of the reduction scheme and the presence of wobbling precision. With exact range reduction the reduced argument will cause overflow, but small perturbations will produce large, though representable, function values. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record given in table 5.3.

Arguments		Error record		
x	*cycle*	error number	library unit	error category
0.5	0.0	1	TAN	argument error
$\pi/2$	2π	(2)	(TAN)	(overflow error)

TABLE 5.3 Error diagnosis for TAN

5.8. COT

The tests for the cotangent function are performed by invoking one of the procedures:

TEST_TAN_COT for the cotangent with cycle 2π,
TEST_TAN_COT_1 for the cotangent with cycle 1 and
TEST_TAN_COT_360 for the cotangent with cycle 360.

The cotangent is computed directly from the tangent through the identity $cot(x) = 1/tan(x)$. This makes an extensive testing of the function superfluous. For this reason there is only one accuracy test, just to demonstrate the correct working of the function. This test is based on the identity $cot(x) = \frac{cot^2(x/2) - 1}{2\,cot(x/2)}$ on $(3\,cycle, 3.125\,cycle)$. It is described in detail in the *Software manual*, so more information can be found there.

The special tests for the cotangent only consist of a check on the equality $cot(x, cycle) = 0.0$, $x = (2k+1) * cycle / 4.0$ and k integer, for small values of k. The results must be zero when *cycle* is exactly representable and zero or a small multiple of the machine epsilon otherwise.

The error tests, that are only included in TEST_TAN_COT, consist of two function calls, which should both trigger an error condition. First the function is called with an argument 0.5 and a cycle 0.0. This should result in an ARGUMENT_ERROR. Second the function is called with an argument 0.0 and this should result in overflow. Similarly, arguments very close to integer multiples of π may result in overflow or large function values depending on the accuracy of argument reduction scheme. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record in table 5.4.

Arguments		Error record		
x	*cycle*	error number	library unit	error category
0.5	0.0	1	COT	argument error
0.0	2π	2	COT	overflow error

TABLE 5.4 Error diagnosis for COT

5.9. ARCSIN

The tests for the inverse sine function are performed by invoking the procedure TEST_ARCSIN_ARCCOS.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $arcsin(x) = \sum_{k=0}^{\infty} a_{2k+1} x^{2k+1}$, $a_{2k+1} = \frac{1}{2k+1} \prod_{i=1}^{k} \frac{2i-1}{2i}$ on $(-0.125, 0.125)$
[test 1 in TEST_ARCSIN_ARCCOS] and

Test b: $arcsin(x) = \frac{\pi}{2} - 2 \sum_{k=0}^{\infty} a_{2k+1} y^{2k+1}$, a_{2k+1} as in test a and $y = \sqrt{\frac{1-x}{2}}$ on (0.75, 1.0)
[test 3 in TEST_ARCSIN_ARCCOS].

The Taylor series in both tests are truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [] denotes the entier function.

The algorithm of the arcsine adopted in the *Software manual* differs from the one that is tested here. Therefore both tests have lost their original meaning and do not serve any special purpose, but merely

determine the overall quality of the ARCSIN.
Usually the computation of the arcsine is very accurate. For that reason it is necessary for an accuracy test not to introduce an excessive error that can not be eliminated in some way. The few mathematical identities involving the arcsine do not meet this requirement and that is why they have been rejected. An alternative is to use a truncated Taylor series. The error introduced by truncating this series after $m+1$ terms can roughly be estimated as follows:

$$0 \leqslant \left| \frac{arcsin(x) - \sum_{k=0}^{m} \frac{(-1)^k x^{2k+1}}{(2k+1)!}}{arcsin(x)} \right| \leqslant \frac{\left| \sum_{k=m+1}^{\infty} a_{2k+1} x^{2k+1} \right|}{|x|} \leqslant \sum_{k=m+1}^{\infty} \frac{|x|^{2k}}{2k+1} \approx \frac{|x|^{2m+2}}{2m+3}$$

This should be much less than the rounding error for the floating-point type under consideration. Further, if the series is evaluated carefully, the additional error made in this evaluation can be limited to one rounding.
The results of the tests are of little value whenever the algorithm that computes the arcsine uses the same series. In that case systematic errors might go undetected.

The special tests start with a check on the parity of the function. The test is performed by sampling a handful of arguments from the interval $(-1, 0)$. Good implementations should return a zero value for each of the arguments.
Second, the equation $arcsin(x) - x \approx 0$, $|x| \ll 1$, is used to compare function values with their arguments for small arguments. The results of this test should be zero or at least small with respect to the working precision. Although redundant, there is also a check for underflow during the computation of the arcsine for very small arguments.
The equality *arcsin*(0.0, *cycle*) = 0.0 is checked for *cycle* = 2π, 1.0 and 360.0. In the case of the regular cycle of 2π, the function is called with only one argument. The results of all function calls should be zero.
The equality *arcsin*(1.0, *cycle*) = *cycle* / 4.0 is verified for *cycle* = 2π, 1.0 and 360.0. For *cycle* = 2π the function is called with a single argument. When *cycle* is a model number the returned values must be exact. Otherwise the function values may differ from the exact values only slightly.
The equality *arcsin*(−1.0, *cycle*) = −*cycle* / 4.0 is checked for *cycle* = 2π, 1.0 and 360.0. The function call for *cycle* = 2π is issued with a single argument. The same results as in the previous test can be expected here.

Arguments		Error record		
x	*cycle*	error number	library unit	error category
1.0	2π	-	--	--
-1.0	2π	-	--	--
1.5	2π	1	ARCSIN	argument error
-1.5	2π	1	ARCSIN	argument error
0.5	0.0	2	ARCSIN	argument error

TABLE 5.5 Error diagnosis for ARCSIN

The error tests consist of a number of function calls of which some should trigger an error situation. The arguments −1.0 and 1.0 are perfectly legal and the function should return correct results. On the other hand, the arguments 1.5 and −1.5 lie outside the function domain and should lead to an ARGUMENT_ERROR. Finally, the function is called with an argument 0.5 and a cycle 0.0, which should

result in the same error. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record in table 5.5.

5.10. ARCCOS

The tests for the inverse cosine function are performed by invoking the procedure TEST_ARCSIN_ARCCOS.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $arccos(x) = \frac{\pi}{2} - \sum_{k=0}^{\infty} a_{2k+1} x^{2k+1}$, $a_{2k+1} = \frac{1}{2k+1} \prod_{i=1}^{k} \frac{2i-1}{2i}$ on $(-0.125, 0.125)$
[test 2 in TEST_ARCSIN_ARCCOS],

Test b: $arccos(x) = 2 \sum_{k=0}^{\infty} a_{2k+1} y^{2k+1}$, a_{2k+1} as in test a and $y = \sqrt{\frac{1-x}{2}}$ on $(0.75, 1.0)$
[test 4 in TEST_ARCSIN_ARCCOS] and

Test c: $arccos(x) = \pi - 2 \sum_{k=0}^{\infty} a_{2k+1} y^{2k+1}$, a_{2k+1} as in test a and $y = \sqrt{\frac{1+x}{2}}$ on $(-1.0, -0.75)$
[test 6 in TEST_ARCSIN_ARCCOS].

The Taylor series in all tests are truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [ ] denotes the entier function.

The algorithm of the arccosine adopted in the *Software manual* differs from the one that is tested here. Therefore the tests have lost their original meaning and do not serve any special purpose, but merely determine the overall quality of the ARCCOS.

Usually the computation of the arccosine is very accurate. For that reason it is necessary for an accuracy test not to introduce excessive errors that can not be eliminated in some way. The few mathematical identities involving the arccosine do not meet this requirement and that is why they have been rejected. An alternative is to use a truncated Taylor series. The series that have been used here, are in fact Taylor series for the arcsine. Therefore the remarks and comments that have been made on the Taylor series for the arcsine can be applied here as well.

The supplemental tests consist of a series of function calls with special arguments. The equality *arccos*(1.0, *cycle*) = 0.0 is checked for *cycle* = 2π, 1.0 and 360.0. In the case of the regular cycle of 2π, the function is called with only one argument. The results of all function calls should be zero.

The equality *arccos*(0.0, *cycle*) = *cycle* / 4.0 is verified for *cycle* = 2π, 1.0 and 360.0. For *cycle* = 2π the function is called with a single argument. When *cycle* is a model number the returned values must be exact. Otherwise the function values may differ from the exact values only slightly.

The equality *arccos*(−1.0, *cycle*) = −*cycle* / 2.0 is checked for *cycle* = 2π, 1.0 and 360.0. The function call for *cycle* = 2π is issued with a single argument. The same results as in the previous test can be expected here.

The error tests consist of a series of function calls of which some should trigger an error situation. The arguments −1.0 and 1.0 are perfectly legal and the function should return correct results. On the other hand, the arguments 1.5 and −1.5 lie outside the function domain and should lead to an ARGUMENT_ERROR. Finally, the function is called with an argument 0.5 and a cycle 0.0, which should result in the same error. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record in table 5.6.

Arguments		Error record		
x	*cycle*	error number	library unit	error category
1.0	2π	-	--	--
-1.0	2π	-	--	--
1.5	2π	1	ARCCOS	argument error
-1.5	2π	1	ARCCOS	argument error
0.5	0.0	2	ARCCOS	argument error

TABLE 5.6 Error diagnosis for ARCCOS

5.11. ARCTAN

The tests for the inverse tangent function are performed by invoking the procedure TEST_ARCTAN_ARCCOT.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $arctan(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$ on $(-0.0625, 0.0625)$,

Test 2: $arctan(x) = arctan(1/16) + arctan(\frac{x - 1/16}{1 + x/16})$ on $(0.0625, 2-\sqrt{3})$,

Test 3: $2\,arctan(x) = arctan(\frac{2x}{1-x^2})$ on $(2-\sqrt{3}, \sqrt{2}-1)$ and

Test 4: $2\,arctan(x) = arctan(\frac{2x}{1-x^2})$ on $(\sqrt{2}-1, 1)$.

The Taylor series in the first test is truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [] denotes the entier function.

The first test determines the accuracy of the function over the primary range where no range reduction is required. In the second test some argument reduction might be necessary in algorithms that differ from the one adopted by the *Software manual*. The function that was tested here, uses a smaller primary interval and as a consequence, some arguments may lie outside this range in this test. Nevertheless, the measured MRE and RMS should be of the same order of magnitude as those reported in the *Software manual*. The third and fourth test examine the range reduction scheme. In both tests the arguments of both sides of the identity lie outside the primary range. Moreover, in the last test the expression $2x/(1-x^2)$ always exceeds one, in which case an additional step in the reduction is involved.

All tests are mentioned in the *Software manual*, so more specific details can be found there.

The special tests start with a check on the parity of the function. The test is performed by sampling a handful of arguments from the interval (0, 5). Good implementations should return a zero value for each of the arguments.

Second, the equation $arctan(x) - x \approx 0$, $|x| \ll 1$, is used to compare function values with their arguments for small arguments. The results of this test should be zero or at least be small to the considered floating-point precision.

Third is a test that examines the relation between $arctan(y/x)$ and $arctan(y,x)$ for $x \in (0, 1)$ and $y \in (-2, 2)$. It determines $arctan(y/x) - arctan(y,x)$ which ideally should be zero, and $arctan(y/(-x)) - arctan(y, -x)$ which should be $\pm\pi$. Then there is a test for underflow for a very small argument.

The equality $arctan(0.0, 1.0, cycle) = 0.0$, is verified for $cycle = 2\pi$, 1.0 and 360.0. For each value of *cycle* the result must be zero.

The equality $arctan(0.0, x, cycle) = 0.0$, $x > 0.0$, is checked for $x =$ FLOAT_TYPE'SMALL, 1.0 and

FLOAT_TYPE'LARGE and *cycle* = 2π, 1.0 and 360.0. In all cases the function must deliver the value zero.
The equality *arctan*(0.0, *x*, *cycle*) = *cycle* / 2.0, $x < 0.0$, is checked for the same values of *cycle* and for the values of *x* with opposite sign. Whenever *cycle* is a model number the results must be exact. Otherwise the results may differ from the exact values only slightly.
The equality *arctan*(*y*, 0.0, *cycle*) = *cycle* / 4.0, $y > 0.0$, is verified for *y* = FLOAT_TYPE'SMALL, 1.0 and FLOAT_TYPE'LARGE and for the same values of *cycle*. The same results as in the previous test can be expected here.
Finally, the equality *arctan*(*y*, 0.0, *cycle*) = $-$*cycle* / 4.0, $y < 0.0$, is verified for the same values of *cycle* and the values of *y* with opposite sign. The remarks made in the previous test, on the results that can be expected also apply here.
In all above tests the cycle argument is omitted in the function call when it has a value of 2π.

The error tests consist of a series of function calls of which some should trigger an error situation. The argument *xmax* is legal and the function value should be $\pi/2$. The same result should be obtained when the function is called with the argument pair (1.0, 0.0). The pair (*xmin*, *xmax*) is also acceptable and the function should return zero or a very small value. The function value for the argument pair (*xmax*, *xmin*) should be $\pi/2$ again. The function call with two zero arguments is not legal and should raise an exception. Finally, the function is called with an argument 1.0 and a cycle 0.0 which should result in an ARGUMENT_ERROR as well. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record in table 5.7.

Arguments			Error record		
y	*x*	*cycle*	error number	library unit	error category
xmax	1.0	2π	-	--	--
1.0	0.0	2π	-	--	--
xmin	*xmax*	2π	-	--	--
xmax	*xmin*	2π	-	--	--
0.0	0.0	2π	1	ARCTAN	argument error
1.0	1.0	0.0	2	ARCTAN	argument error

TABLE 5.7 Error diagnosis for ARCTAN

5.12. ARCCOT
The tests for the inverse cotangent are performed by invoking the procedure TEST_ARCTAN_ARCCOT.

The inverse cotangent is computed directly from the inverse tangent through the identity *arccot*(*x*,*y*) = *arctan*(*y*,*x*). This makes an extensive testing superfluous. For this reason there is only one accuracy test merely to demonstrate that the function works properly. The test is based on the identity $2\,arccot(x) = arccot(\frac{x^2-1}{2x})$ on (1, 6). If we use this identity then we measure $E = [2\,arccot(x) - arccot(y)]/[2\,arccot(x)]$, $y = \frac{x^2-1}{2x}$. If the relative errors in the evaluation of *arccot*(*x*), *arccot*(*y*) and *y* are denoted by δ, ϵ and η respectively, then

$$E = \frac{2\,arccot(x)(1+\delta) - arccot(y)(1+\epsilon+a\eta)}{2\,arccot(x)(1+\delta)}, \text{ where } a = y/[(1+y^2)arccot(y)]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - \epsilon - a\eta$. For the interval (1, 6) *a*

monotonically increases from 0.0 to 0.93. Hence E measures primarily the relative error in $arccot(x)$ contaminated by small multiples of the errors in y and $arccot(y)$. The MRE and RMS for this test should be comparable to those of the fourth test of the arctangent.

The special tests consist of a series of function calls with special arguments. The equality $arccot(0.0, 1.0, cycle) = cycle / 4.0$, is verified for $cycle = 2\pi$, 1.0 and 360.0. Whenever *cycle* is a model number the results must be exact. Otherwise the results may differ from the exact values only slightly.
The equality $arccot(0.0, y, cycle) = cycle / 4.0$, $y > 0.0$, is checked for $y =$ FLOAT_TYPE'SMALL, 1.0 and FLOAT_TYPE'LARGE and $cycle = 2\pi$, 1.0 and 360.0. The same results as in the previous test may be expected here.
The equality $arccot(0.0, y, cycle) = -cycle / 4.0$, $y < 0.0$, is checked for the same values of *cycle* and for the values of y with opposite sign. Whenever *cycle* is a model number the results must be exact. Otherwise the results may differ from the exact values only slightly.
The equality $arccot(x, 0.0, cycle) = 0.0$, $x > 0.0$, is verified for $x =$ FLOAT_TYPE'SMALL, 1.0 and FLOAT_TYPE'LARGE and for the same values of *cycle*. In all cases the results must be exact.
Finally, the equality $arccot(x, 0.0, cycle) = cycle / 2.0$, $x < 0.0$, is verified for the same values of *cycle* and the values of x with opposite sign. Whenever *cycle* is a model number the results must be exact. Otherwise the results may differ from the exact values only slightly.
In all above tests the cycle argument is omitted in the function call when it has a value of 2π.

The error tests consist of two function calls, which should both trigger an error situation. First the function is called with two zero arguments which should result in an ARGUMENT_ERROR. Second there is a function call with arguments one and a cycle zero. This should raise the same exception. The status of the error record after the function has been called with the above arguments, should be in accordance with the corresponding error record in table 5.8.

Arguments			Error record		
y	x	*cycle*	error number	library unit	error category
0.0	0.0	2π	1	ARCCOT	argument error
1.0	1.0	0.0	2	ARCCOT	argument error

TABLE 5.8 Error diagnosis for ARCCOT

5.13. SINH
The tests for the hyperbolic sine are performed by invoking the procedure TEST_SINH_COSH.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $sinh(x) = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!}$ on (0, 0.5) [test 1 in TEST_SINH_COSH] and

Test b: $sinh(x) = c(sinh(x+1) + sinh(x-1))$, $c = 1/[2\,cosh(1)]$, on $(3, log(xmax))$
[test 3 in TEST_SINH_COSH].

The Taylor series in the first test is truncated after $m+1$ terms where $m = 2 - [log(eps) * 3]/20$ and [ ] denotes the entier function.
The first test determines the accuracy of the hyperbolic sine on an interval where the function is computed by means of a polynomial approximation rather than through the exponential function. However, if the SINH is computed by the same series this test is of little value. In that case systematic errors might go undetected. In contrast with the first test, the second one examines the function when

the function values are computed with the aid of the exponential function. Both tests are described in the *Software manual*, so more details can be found there.

The special tests start with a check on the parity of the hyperbolic sine. The test is performed by sampling a handful of arguments from the interval (0, 3). Good implementations should return zero for all arguments.
Second, function values are compared with their arguments for small arguments through the equation $sinh(x) - x \approx 0$, $|x| \ll 1$. The results of this test should be zero and at least be small with respect to the precision of the floating-point type in question.
Then there is a function call with a very small argument, that may cause underflow in the computation of $sinh(x)$, although this is beyond question in Ada. Finally the function is called with the argument zero. The result must also be zero.

The error tests for the SINH consist of two function calls of which the second should trigger an error situation due to overflow. The argument $log(xmax) + 0.125$ is legal and the function should return a value of the same order of magnitude as *xmax*. On the other hand, the argument β^{it} should result in overflow. In this case the error record should state (1, "SINH", "OVERFLOW ERROR").

5.14. COSH

The tests for the hyperbolic cosine are performed by invoking the procedure TEST_SINH_COSH.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $cosh(x) = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$ on (0, 0.5) [test 2 in TEST_SINH_COSH] and

Test b: $cosh(x) = c\,(cosh(x+1) + cosh(x-1))$, $c = 1/[2\,cosh(1)]$, on $(3, log(xmax))$
[test 4 in TEST_SINH_COSH].

The Taylor series in the first test is truncated after $m+1$ terms where $m = 2 - [log(eps) * 3]/20$ and [ ] denotes the entier function.
The first test determines the accuracy of the hyperbolic cosine on an interval where the function is computed directly from its definition $cosh(x) = (e^x + e^{-x}) / 2$. The second test determines the accuracy of the function on an interval where the function is computed by a somewhat different algorithm to avoid intermediate overflow for large arguments. Both tests are described in the *Software manual*, so more specific details can be found there.

The special tests start with a check on the parity of the hyperbolic cosine. The test is performed for some arguments drawn randomly from the interval (0, 3). The results of these test should be zero. The function call with an argument zero should deliver the result one.

The error tests for the COSH consist of two function calls of which the second should trigger an error situation due to overflow. The argument $log(xmax) + 0.125$ is legal and the function should return a value of the same order of magnitude of *xmax*. On the other hand, the argument β^{it} should result in overflow. In this case the error record should state (1, "COSH", "OVERFLOW ERROR").

5.15. TANH

The tests for the hyperbolic tangent are performed by invoking the procedure TEST_TANH_COTH.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $tanh(x) = \frac{tanh(x - 1/8) + tanh(1/8)}{1 + tanh(x - 1/8)tanh(1/8)}$ on $(0.125, 0.5\,log(3))$ and

Test 2: $tanh(x) = \dfrac{tanh(x-1/8) + tanh(1/8)}{1 + tanh(x-1/8)tanh(1/8)}$ on $(0.125 + 0.5\,log(3), d)$,
where $d = log(2) + \dfrac{it+1}{2} log(\beta)$.

The first test determines the accuracy of the hyperbolic tangent on an interval where the function is computed directly through a polynomial approximation. In the second test the function values are calculated by means of the exponential function. Both tests are described in the *Software manual*, so more specific details can be found there.

The special tests start with a check on the odd parity of the hyperbolic tangent. The test is performed by sampling a handful of arguments from the interval (0, 1). Good implementations should return zero for every argument.
Next, the equation $tanh(x) - x \approx 0$, $|x| \ll 1$, is used to compare function values with their arguments for small input values. The result of this test should ideally be zero and must at least be small with respect to the precision of the floating-point type in question.
Third is a test that compares $tanh(x)$ with 1 for arguments $x \gg 1$. For this test the results should also be zero or nearly zero. Then there is a check for underflow for a very small argument, although in fact this is redundant in Ada. Finally, there is a series of function calls with the arguments 0.0, *xmin* and *xmax*. The hyperbolic tangent must return zero for the first argument and proper values for the other ones.

There are no error tests for the hyperbolic tangent.

5.16. COTH

The tests for the hyperbolic cotangent are performed by invoking the procedure TEST_TANH_COTH.

The hyperbolic cotangent is computed directly from the hyperbolic tangent through the identity $coth(x) = \dfrac{1}{tanh(x)}$. This makes an extensive testing superfluous. For this reason there is only one accuracy test, merely to demonstrate that no programming errors have been made. The test is based on the identity $coth(x) = \dfrac{coth^2(x/2) + 1}{2\,coth(x/2)}$ on $(1, 4)$. Thus we measure $E = [coth(x) - (coth^2(x/2)+1)/(2\,coth(x/2))] / coth(x)$. By perturbating x slightly we can accomplish that both x and $x/2$ are error-free. This can easily be done by the statements '$y := x * 0.5$; $x := y + y$;'. Now assume relative errors δ and ϵ are made in the evaluations of $coth(x)$ and $coth(x/2)$ respectively. Then

$$E = \frac{coth(x)(1+\delta) - (coth^2(x/2)(1+\epsilon)^2 + 1)/(2\,coth(x/2)(1+\epsilon))}{coth(x)(1+\delta)}$$

Simplifying and keeping only terms linear in δ and ϵ gives $E = \delta - a\epsilon$, where $a = 1/cosh(x)$. For the interval under consideration, a monotonically decreases from 0.65 to 0.04. Therefore E measures primarily the relative error in $coth(x)$ contaminated by small multiples of the error in $coth(x/2)$. The MRE and RMS in this test might be slightly larger than those reported for the second accuracy test of the hyperbolic tangent.

There are no special tests for the hyperbolic cotangent.

The error tests consist of two function calls of which at least one should trigger an error situation. The argument of the first call is zero and this should lead to an exception due to overflow. The second call takes *xmin* as argument and this may cause overflow depending on the value returned by the

hyperbolic tangent. If overflow occurs the error record should state (1, "COTH", "OVERFLOW ERROR").

5.17. ARCSINH

The tests for the inverse hyperbolic sine are performed by invoking the procedure TEST_ARCSINH_ARCCOSH.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $arcsinh(x) = \sum_{k=0}^{\infty} a_{2k+1} x^{2k+1}$, $a_{2k+1} = \frac{(-1)^k}{2k+1} \prod_{i=1}^{k} \frac{2i-1}{2i}$, on $(-0.25, 0.25)$,

Test 2: $2\,arcsinh(x) = arcsinh(2x\sqrt{x^2+1})$ on $(1/\sqrt{2}, 1)$ and

Test 3: $2\,arcsinh(x) = arcsinh(2x\sqrt{x^2+1})$ on $(1, log(xmax))$.

The Taylor series in the first test is truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [] denotes the entier function.

The first test checks the accuracy of the inverse hyperbolic sine on an interval where the function is calculated through a polynomial approximation. In this test approximately half of the randomly drawn arguments need some moderate range reduction.

The relative difference between $arcsinh(x)$ and the truncated series can roughly be estimated as follows:

$$0 \leqslant \left| \frac{arcsinh(x) - \sum_{k=0}^{\infty} a_{2k+1} x^{2k+1}}{arcsinh(x)} \right| \leqslant \frac{\left| \sum_{k=m+1}^{\infty} a_{2k+1} x^{2k+1} \right|}{|arcsinh(x)|} \leqslant \frac{|a_{2m+3}|\,|x|^{2m+3}}{|arcsinh(x)|} \approx |a_{2m+3}|\,|x|^{2m+2}$$

Since $|a_{2m+3}|$ is always smaller than one, this should be much less than the rounding error for the floating-point type in question. By evaluating the truncated series carefully, the additional error due to this process can be limited to one rounding. If the function uses the same Taylor series one should beware of undetected systematic errors.

This test should report a loss of about one base β digit or less for the MRE and a loss of a small fraction of a digit for the RMS.

The identity $2\,arcsinh(x) = arcsinh(2x\sqrt{x^2+1})$ is used in the accuracy tests for arguments outside the primary range. If we use this identity then we measure $E = [2\,arcsinh(x) - arcsinh(y)] / [2\,arcsinh(x)]$, where $y = 2x\sqrt{x^2+1}$. If δ and ϵ are the relative errors in the evaluation of $arcsinh(x)$ and $arcsinh(y)$ respectively, and η denotes the relative error in the evaluation of y then

$$E = \frac{2\,arcsinh(x)(1+\delta) - arcsinh(y)(1+\epsilon+a\eta)}{2\,arcsinh(x)(1+\delta)}, \text{ where } a = y / [arcsinh(y)\sqrt{y^2+1}]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - \epsilon - a\eta$. For the second test the arguments x are drawn from the interval $(1/\sqrt{2}, 1)$, so that y lies in the interval $(\sqrt{3}, 2\sqrt{2})$. In this case a monotonically decreases from 0.66 to 0.53. Thus E measures primarily the relative error in $arcsinh(x)$ contaminated by some small multiples of the errors in y and $arcsinh(y)$. The error in y is kept small by evaluating it carefully.

In the third test the arguments x exceed 1.0 so that y exceeds $2\sqrt{2}$. Now a is always smaller than 0.53 and vanishes when y goes to infinity. Also in this case, E measures primarily the relative error in $arcsinh(x)$ contaminated by small multiples of the errors in y and $arcsinh(y)$. This test should report a loss of about one base β digit for the MRE, and a loss of a small fraction of a digit for the RMS. Only for binary machines it is acceptable that the MRE reports a loss of about two bits.

The special tests start with a check on the odd parity of the inverse hyperbolic sine. The test is

performed by sampling a handful of arguments from the interval (0, 2). Good implementations should return zero for each of the arguments.
Second, the equation $arcsinh(x) - x \approx 0$, $|x| \ll 1$, is used to compare function values with their arguments for small arguments. The results of this test should ideally be zero and must at least be small to the precision of the floating-point type in question.
Then there is a test for underflow for a very small argument, although this can not occur in Ada. Finally, the function is called with an argument zero. The result should be zero as well.

The error test consists of one single function call with the argument *xmax*, which should not trigger an error situation.

5.18. ARCCOSH

The tests for the inverse hyperbolic cosine are performed by invoking the procedure TEST_ARCSINH_ARCCOSH.

The accuracy tests are based on the following mathematical identities and intervals:

Test a: $arccosh(x) = \sum_{k=0}^{\infty} a_{2k+1} y^{2k+1}$, $a_{2k+1} = \frac{(-1)^k}{2k+1} \prod_{i=1}^{k} \frac{2i-1}{2i}$ and $y = \sqrt{x^2-1}$,
on $(1, \sqrt{17/16})$, [test 4 in TEST_ARCSINH_ARCCOSH],

Test b: $arccosh(x) = arcsinh(1) + arcsinh(\sqrt{2(x^2-1)} - x)$ on $(\sqrt{2}, 2)$
[test 5 in TEST_ARCSINH_ARCCOSH] and

Test c: $2\,arccosh(x) = arccosh(2x^2-1)$ on $(2, log(xmax))$
[test 6 in TEST_ARCSINH_ARCCOSH].

The Taylor series in the third test is truncated after $m+1$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [ ] denotes the entier function.
The first test describes the inverse hyperbolic cosine on an interval where the function is computed by means of the inverse hyperbolic sine. The remarks on the truncated series in the first accuracy test of the inverse hyperbolic sine, also apply to the series in test a, since they are essentially the same. The fact that y may be slightly perturbed due to the evaluation of the square root, should hardly affect the results of this test. Therefore the results of this test should be comparable to those obtained in the first test of the inverse hyperbolic sine, i.e. the measured MRE should be about one base β digit and the RMS should be a small fraction of it.

The second test determines the accuracy of the function for moderate size arguments, when it is computed through the logarithmic function and the square root. If we use the above-mentioned identity then we measure $E = [arccosh(x) - [arcsinh(1) + arcsinh(y)]] / arccosh(x)$ where $y = \sqrt{2(x^2-1)} - x$. If we assume that the relative errors in $arccosh(x)$, $arcsinh(y)$ and y are denoted by δ, ϵ and η respectively, then

$$E = \frac{arccosh(x)(1+\delta) - [arcsinh(1) + arcsinh(y)(1+\epsilon+a\eta)]}{arccosh(x)(1+\delta)}, \text{ where } a = y \,/\, [arcsinh(y)\sqrt{y^2+1}]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - b\epsilon - c\eta$, where $b = arcsinh(y) / arccosh(x)$ and $c = a*b$. For the interval $(\sqrt{2}, 2)$ b and c are approximately the same size. Each vanishes for $x = \sqrt{2}$ and each is bounded above by 0.33. Thus E measures primarily the relative error in $arccosh(x)$ perturbed by small multiples of the errors in $arcsinh(y)$ and y. To avoid that the addition of $arcsinh(1)$ introduces extra error, this constant has been broken into a sum of two constants that are added to $arcsinh(y)$ in an appropriate order.
The results for this test might be slightly larger than those for test a, but in general the MRE should still report a loss of about one base β digit and the RMS a small fraction of it.

The third test examines the function for large arguments where the function uses the same approximation as in test b. Using the identity $2\,arccosh(x) = arccosh(2x^2-1)$, we measure:
$E = [2\,arccosh(x) - arccosh(y)] / [2\,arccosh(x)]$, where $y = 2x^2 - 1$. If the relative errors in $arccosh(x)$, $arccosh(y)$ and y are denoted by δ, ϵ and η respectively then

$$E = \frac{2\,arccosh(x)(1+\delta) - arccosh(y)(1+\epsilon+a\eta)}{2\,arccosh(x)(1+\delta)}, \text{ where } a = y / [arccosh(y)\sqrt{y^2-1}]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - \epsilon - a\eta$. For the interval under consideration, a never exceeds 0.38 and vanishes when y goes to infinity. Thus E measures primarily the relative error in $arccosh(x)$ perturbed by small multiples of the errors in y and $arccosh(y)$. The results of this test should be comparable with those of the third test of the inverse hyperbolic sine. The measured MRE should report a loss of about one base β digit, except for binary machines that may report a loss of about two bits for the MRE, and a small fraction of it for the RMS.

The special tests for the inverse hyperbolic cosine only consist of a function call with the argument one. The result of this function call should be zero.

The error tests consist of a series of function calls some of which should trigger an error situation. The arguments *xmax* and 1.0 are perfectly legal and should not cause any trouble. However, the argument 0.5 is not acceptable since it is smaller than one. This should raise an ARGUMENT_ERROR. After the function has been called with this argument, the error record should state (1, "ARCCOSH", "ARGUMENT ERROR").

5.19. ARCTANH

The tests for the inverse hyperbolic tangent are performed by invoking the procedure TEST_ARCTANH_ARCCOTH.

The accuracy tests are based on the following mathematical identities and intervals:

Test 1: $arctanh(x) = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{2k+1}$ on $(-0.0625, 0.0625)$,

Test 2: $arctanh(x) = arctanh(1/16) + arctanh((x-1/16)/(1-x/16))$
on $(0.0625, 2+\sqrt{3} - \sqrt{6+4\sqrt{3}})$ and

Test 3: $2\,arctanh(x) = arctanh(\frac{2x}{x^2+1})$ on $(2+\sqrt{3} - \sqrt{6+4\sqrt{3}}, 2-\sqrt{3})$.

The Taylor series in the first test is truncated after $m+2$ terms where $m = [^{10}log(\beta^{it})] + 1$ and [] denotes the entier function.
The first test examines the function on the primary interval where argument reduction is not necessary, and the function is calculated through a polynomial approximation. The error introduced by truncating the series after $m+2$ terms can roughly be estimated as follows:

$$0 \leqslant \frac{\left| arctanh(x) - \sum_{k=0}^{m+1} \frac{x^{2k+1}}{2k+1} \right|}{|arctanh(x)|} \leqslant \frac{\left| \sum_{k=m+2}^{\infty} \frac{x^{2k+1}}{2k+1} \right|}{|x|} \leqslant \sum_{k=m+2}^{\infty} \frac{|x|^{2k}}{2k+1} \leqslant \frac{2}{2m+5} \frac{1}{16^{2m+4}}$$

This should be much less than the rounding error for the floating-point type in question. If the evaluation of the series is done carefully, the additional error in this process can be limited to one rounding. Again, care should be taken when the polynomial approximation uses the same series, since systematic errors might go undetected in that case.
This test should report a loss of one base β digit for the MRE and a loss of a small fraction of a digit for the RMS.

The second test determines the accuracy of the function on an interval where the function still uses the same approximation, but where the arguments might have to be reduced. If we use the above-mentioned identity, we measure $E = [arctanh(x) - [arctanh(1/16) + arctanh(y)]] / arctanh(x)$, where $y = (x - 1/16) / (1 - x/16)$. In this case we can perturbate x to a nearby model number to achieve exact representation of both the numerator and the denominator of y. The Ada statement '$x := ((1.0 + x*0.0625) - 1.0)*16.0;$' can be used for this purpose except for machines on which the arithmetic registers carry more significance than the storage registers.
By breaking the constant $arctanh(1/16)$ into a sum of two constants that are added in an appropriate order, the computation of E is further stabilised. If δ, ϵ and η are the relative errors in the evaluation of $arctanh(x)$, $arctanh(y)$ and y respectively, then

$$E = \frac{arctanh(x)(1+\delta) - [arctanh(1/16) + arctanh(y)(1+\epsilon+a\eta)]}{arctanh(x)(1+\delta)}, \text{ where } a = y / [(1-y^2)arctanh(y)]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - b\epsilon - c\eta$, where $b = arctanh(y)/arctanh(x)$ and $c = a*b$. For the interval under consideration, b and c are approximately of the same order of magnitude. They both vanish for $x = 1/16$ and they are bounded by 0.55. Thus E measures primarily the relative error in $arctanh(x)$ contaminated by small multiples of the errors in y and $arctanh(y)$. The measured MRE and RMS may be slightly larger than those of the first test, but should not differ too much.

The third and last accuracy test determines the accuracy of the function when it is computed through the logarithmic function. Using the above identity, we measure:
$E = [2\,arctanh(x) - arctanh(y)] / [2\,arctanh(x)]$, where $y = 2x/\sqrt{x^2+1}$. If δ, ϵ and η denote the relative errors in the evaluation of $arctanh(x)$, $arctanh(y)$ and y respectively, then

$$E = \frac{2\,arctanh(x)(1+\delta) - arctanh(y)(1+\epsilon+a\eta)}{2\,arctanh(x)(1+\delta)}, \text{ where } a = y / [(1-y^2)arctanh(y)]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - \epsilon - a\eta$. For the test interval a is approximately one. Hence E measures the relative error in $arctanh(x)$ contaminated by the errors in $arctanh(y)$ and y. The MRE and RMS may be slightly larger than those of the second test, but the MRE should still report a loss of about one base β digit and the RMS a small fraction of it.

The special tests start with a check on the odd parity of the inverse hyperbolic tangent. The test is performed by sampling a handful of arguments from the interval (0, 1). Good implementations should return a zero value for each of the arguments.
Second, the equation $arctanh(x) - x \approx 0$, $|x| \ll 1$, is used to compare the function values with their arguments for small arguments. The results of this test should ideally be zero and must at least be small with respect to the precision of the floating-point type in question.
Next is a test for underflow for a very small argument, although this can not occur in Ada. Finally, the function call with an argument zero should deliver the value zero.

The error tests consist of two function calls of which the second should trigger an error situation. The boundary argument 1.0 lies within the function domain and should not cause any trouble. On the other hand, the argument 3.0 is illegal since it lies outside the function domain. This should raise the exception ARGUMENT_ERROR. After the function has been called with this argument the error record should state (1, "ARCTANH", "ARGUMENT ERROR").

5.20. ARCCOTH

The tests for the inverse hyperbolic cotangent are performed by invoking the procedure TEST_ARCTANH_ARCCOTH.

The inverse hyperbolic cotangent is computed directly from the inverse hyperbolic tangent through the identity $arccoth(x) = arctanh(1/x)$. This makes an extensive testing of the function superfluous. For this reason there is only one accuracy test, merely to demonstrate that no programming errors have been made. The test is based on the identity $2\,arccoth(x) = arccoth((x^2+1)/2x)$ on (3, 6). Thus we measure $E = [2\,arccoth(x) - arccoth(y)] / [2\,arccoth(x)]$, where $y = (x^2+1)/2x$. If we assume δ, ϵ and η to be the relative errors made in the evaluation of $arccoth(x)$, $arccoth(y)$ and y respectively, then

$$E = \frac{2\,arccoth(x)(1+\delta) - arccoth(y)(1+\epsilon+a\eta)}{2\,arccoth(x)(1+\delta)}, \text{ where } a = y / [(1-y^2)arccoth(y)]$$

Simplifying and keeping only terms linear in δ, ϵ and η gives $E = \delta - \epsilon - a\eta$. For the interval under consideration, a varies from -1.35 for $x = 3$ to -1.08 for $x = 6$. Thus E measures primarily the relative error in $arccoth(x)$ contaminated by the errors in $arccoth(y)$ and y. The MRE of this test should report a loss of slightly more than one base β digit or less, and the RMS a loss of a fraction of it.

There are no special tests for the inverse hyperbolic cotangent.

The error tests consist of two function calls of which the second should trigger an error situation. The boundary argument 1.0 lies within the function domain and should not cause any trouble. On the other hand, the argument 0.5 is illegal since it exceeds the function domain. This should raise an ARGUMENT_ERROR. After the function has been called with this argument, the error record should state (1, "ARCCOTH", "ARGUMENT ERROR").

6. PERFORMANCE

The generic package of elementary functions has been designed in such a manner that it can be instantiated with floating-point types of a precision up to 35 significant digits. Here, instantiations for three floating-point types have been made: for type REAL, LONG_REAL and REAL_03. REAL and LONG_REAL are library (sub-)types which have an accuracy of about six digits and fifteen digits respectively. Type REAL_03 is a user-defined type that is by definition a floating-point type with three digits accuracy.

The instantiations have been made on an Alliant FX/4 and on a DG with compiler version 2.50. The Alliant FX/4 is a binary machine that provides two floating-point types: type SHORT_FLOAT and type FLOAT, having an accuracy of six digits and fifteen digits respectively. Both REAL and REAL_03 are mapped on type SHORT_FLOAT, whereas LONG_REAL is mapped on type FLOAT.

The DG is a hexadecimal machine that also provides two floating-point types: type FLOAT and type LONG_FLOAT, having an accuracy of six digits for type FLOAT and fifteen digits for type LONG_FLOAT. On this machine REAL and REAL_03 are mapped on type FLOAT. Type LONG_REAL is mapped on type LONG_FLOAT. Since it was not possible to instantiate the generic package directly, static instantiations had to be made. However, this was not the only problem. On the DG the instance for type LONG_REAL appeared not to be usable without large adaptations. In addition, it would be necessary to modify the test programs themselves. Therefore, the tests for type LONG_REAL have been omitted. Tests have only been performed using the instantiations with type REAL and REAL_03.

By testing the package of elementary functions on two machines with different representations of floating-point numbers, we can observe some important points. First of all, we can verify the portability of both the package of elementary functions and the package of test programs. Second, it is

possible to study the effect of unequal radices and the wobbling precision of one of the representations.

We shall first discuss the results of the accuracy tests for the user-defined type REAL_03. In the initial approach the implementation used lower-accuracy approximations for user-defined types having lower precision than a built-in floating-point type. Therefore, the results for this type are not very surprising. Every test reports no loss for the RMS and in most cases no loss for the MRE as well. The few occasions where the measured MRE does not equal zero, can be neglected (a maximal loss of 0.05 bits in the test of the cotangent). The explanation for these results has already been mentioned: the type REAL_03 is internally mapped onto some predefined type and all computations are performed in this type, after which the result is converted to type REAL_03. On the machines where the elementary functions have been tested, type REAL_03 was mapped on type FLOAT and SHORT_FLOAT respectively, which both have an accuracy of approximately six decimal digits. Thus the computations were performed in an extended precision of about three extra digits. This holds for both the elementary functions and the test procedures. As a consequence, the errors introduced by inaccuracies and rounding in the test procedures can be expected to be eliminated for the most part, thanks to the extended precision. In general only the errors due to inaccuracies in the approximations will be measured, and these turn out to be insignificant.

However, in the earlier mentioned standardisation proposal, the accuracy requirements are now related to the precision of the associated predefined type on which a user-defined type (REAL_03) is mapped. In that case, the errors of the elementary functions of a REAL_03 instance must satisfy the (more severe) accuracy requirements of the base type.

The reported values for the MRE and the RMS do not change fundamentally. For every test the reported loss is zero or so small that it can be ignored safely. If we express the values for the MRE in the epsilon of the base type of type REAL_03, the results are not trivial. On the binary machine the MRE is always less than a half times the BASE'EPSILON of type REAL_03. On the other, hexadecimal, machine the reported loss for the MRE can be more than two times the BASE'EPSILON of REAL_03. This can probably be attributed to the wobbling precision which causes loss of significance.

In either case the results largely satisfy the accuracy requirements of the base type of REAL_03.

6.1. Results for type REAL

In this section we will discuss the test results of the elementary functions for type REAL. For every test procedure the results of the accuracy tests are reported together with the results of the corresponding tests as stated in the *Software manual.* This offers the opportunity to compare our results.

Table 6.1 shows the results of the accuracy tests for the SQRT function. The measured MRE and RMS are quite acceptable. The special tests and the error tests were also performed correctly on both

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	0.49	0.00	2	Alliant FX/4	2	PIA (Ada)	0.99	0.00
	CDC 6400	2	Cody & Waite	0.50	0.26		CDC 6400	2	Cody & Waite	2.00	1.31
	CDC 7600	2	FTN 4.6	0.50	0.00		CDC 7600	2	FTN 4.6	0.99	0.00
	DG v2.50	16	PIA (Ada)	0.50	0.00		DG v2.50	16	PIA (Ada)	0.00	0.00
	IBM/370	16	Cody & Waite	0.75	0.41		IBM/370	16	Cody & Waite	1.00	0.81
	IBM/370	16	FTX 2.2	0.75	0.24		IBM/370	16	FTX 2.2	0.00	0.00

TABLE 6.1 Typical results for SQRT tests

machines.

The results for the LOG function are demonstrated in table 6.2. The MRE in the first test on the binary machine is slightly larger than usually, but acceptable. The remainder part of the results is common for both the binary and the hexadecimal machine. On both machines the function passes the auxiliary tests with extreme arguments, and also preserves the identity $log(x) = -log(1/x)$. Finally, it detects and handles error conditions properly.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	1.62	0.00	4	Alliant FX/4	2	PIA (Ada)	2.56	0.52
	CDC 6400	2	Cody & Waite	1.00	0.00		CDC 6400	2	Cody & Waite	3.28	1.29
	CDC 7600	2	FTN 4.7	2.00	0.09		CDC 7600	2	FTN 4.7	2.53	0.56
	GP L3055	10	Cody & Waite	1.31	0.81		GP L3055	10	Cody & Waite	1.13	0.57
	DG v2.50	16	PIA (Ada)	1.00	0.13		DG v2.50	16	PIA (Ada)	1.01	0.49
	IBM/370	16	Cody & Waite	1.00	0.19		IBM/370	16	Cody & Waite	1.07	0.59
	IBM/370	16	FTX 2.2	1.00	0.29		IBM/370	16	FTX 2.2	1.04	0.51
2	Alliant FX/4	2	PIA (Ada)	1.81	0.05	5	Alliant FX/4	2	PIA (Ada)	1.00	0.00
	CDC 6400	2	Cody & Waite	1.67	0.00		CDC 6400	2	Cody & Waite	0.97	0.00
	CDC 7600	2	FTN 4.7	1.74	0.02		CDC 7600	2	FTN 4.7	0.98	0.00
	GP L3055	10	Cody & Waite	1.34	0.83		GP L3055	10	Cody & Waite	1.00	0.42
	DG v2.50	16	PIA (Ada)	0.99	0.50		DG v2.50	16	PIA (Ada)	0.58	0.17
	IBM/370	16	Cody & Waite	0.99	0.57		IBM/370	16	Cody & Waite	0.53	0.10
	IBM/370	16	FTX 2.2	1.00	0.53		IBM/370	16	FTX 2.2	0.50	0.09
3	Alliant FX/4	2	PIA (Ada)	1.99	0.12						
	CDC 6400	2	Cody & Waite	-.--	-.--						
	CDC 7600	2	FTN 4.7	-.--	-.--						
	GP L3055	10	Cody & Waite	-.--	-.--						
	DG v2.50	16	PIA (Ada)	1.09	0.53						
	IBM/370	16	Cody & Waite	-.--	-.--						
	IBM/370	16	FTX 2.2	-.--	-.--						

TABLE 6.2 Typical results for LOG tests

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	1.65	0.04	3	Alliant FX/4	2	PIA (Ada)	1.62	0.00
	CDC 7600	2	Cody & Waite	1.50	0.00		CDC 7600	2	Cody & Waite	1.55	0.05
	CDC 7600	2	FTN 4.7	1.63	0.06		CDC 7600	2	FTN 4.7	1.31	0.00
	DG v2.50	16	PIA (Ada)	1.00	0.57		DG v2.50	16	PIA (Ada)	1.07	0.64
	IBM/370	16	Cody & Waite	1.00	0.67		IBM/370	16	Cody & Waite	1.00	0.56
	IBM/370	16	FTX 2.2	1.00	0.68		IBM/370	16	FTX 2.2	1.00	0.55
2	Alliant FX/4	2	PIA (Ada)	1.61	0.00						
	CDC 7600	2	Cody & Waite	1.55	0.00						
	CDC 7600	2	FTN 4.7	1.37	0.00						
	DG v2.50	16	PIA (Ada)	1.00	0.61						
	IBM/370	16	Cody & Waite	1.00	0.55						
	IBM/370	16	FTX 2.2	1.00	0.53						

TABLE 6.3 Typical results for EXP tests

Table 6.3 illustrates the results of the accuracy tests for the EXP function. On both machines the accuracy of the function is quite good. The measured MRE and RMS appear to be as expected.
The function passes the supplemental tests with special arguments, and the results for examining the identity $e^x * e^{-x} = 1$ are satisfactory.
Finally, the function handles error situations properly.

The accuracy of the "**" function is excellent on both machines. The results of the accuracy tests are shown in table 6.4. The results of the first test show that the function tests explicitly whether the power is one.
The additional tests in which the identity $x^y = (1/x)^{-y}$ is examined, appear to be satisfactory. The measured relative errors are zero or small with respect to the precision of type REAL.
Finally, the function handles invalid arguments properly.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	0.00	0.00	3	Alliant FX/4	2	PIA (Ada)	0.99	0.01
	CDC 6400	2	Cody & Waite	0.37	0.00		CDC 6400	2	Cody & Waite	1.00	0.00
	CDC 7600	2	FTN 4.6	1.00	0.00		CDC 7600	2	FTN 4.6	9.92	8.58
	DG v2.50	16	PIA (Ada)	0.00	0.00		DG v2.50	16	PIA (Ada)	0.99	0.52
	IBM/370	16	Cody & Waite	0.99	0.43		IBM/370	16	Cody & Waite	1.00	0.20
	IBM/370	16	Argonne	0.00	0.00		IBM/370	16	Argonne	1.00	0.48
2	Alliant FX/4	2	PIA (Ada)	0.99	0.00	4	Alliant FX/4	2	PIA (Ada)	1.11	0.00
	CDC 6400	2	Cody & Waite	0.99	0.00		CDC 6400	2	Cody & Waite	4.21	1.59
	CDC 7600	2	FTN 4.6	2.23	0.18		CDC 7600	2	FTN 4.6	10.00	6.69
	DG v2.50	16	PIA (Ada)	0.99	0.40		DG v2.50	16	PIA (Ada)	1.11	0.53
	IBM/370	16	Cody & Waite	1.00	0.21		IBM/370	16	Cody & Waite	0.99	0.45
	IBM/370	16	Argonne	1.00	0.48		IBM/370	16	Argonne	1.00	0.26

TABLE 6.4 Typical results for "**" tests

Table 6.5 shows the results of the accuracy tests for the SIN and COS function with the regular cycle of 2π. On both machines the measured MRE and RMS are typical for good implementations of the sine and cosine function. The MRE and RMS for the accuracy tests that are not included in the *Software manual* are also in accordance with the expectations.
In the additional tests the parity of both the SIN and the COS is preserved. This also holds for the small angle approximation, but for one exception. This occurs when this property is checked for type REAL_03. In this case not all arguments return zero. An explanation for that must be sought in the argument reduction process that reduces arguments given an arbitrary cycle. Without making any distinction in cycles the reduction process scales arguments and maps them to the interval $[-\frac{\pi}{4}, \frac{\pi}{4}]$.
Here, the SIN function examines this new argument and decides which approximation has to be used.
The results of the remaining special tests are all satisfactory.
Finally, the error tests are handled correctly as well.

The test procedures for the SIN and COS with cycles 1 and 360 give the expected results. These results are shown in table 6.6 and 6.7 respectively. In all runs of both test programs the measured RMS is almost identical to the RMS reported by the corresponding tests with cycle 2π. The measured MRE is somewhat larger than the MRE in the corresponding tests with cycle 2π, but not more than 0.5 bit on the binary machine and 0.25 on the hexadecimal machine. The parity of the SIN is also preserved here. The results of the remaining supplemental tests of the sine and cosine show the values

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	1.75	0.00	4	Alliant FX/4	2	PIA (Ada)	1.60	0.00
	CDC 6400	2	Cody & Waite	2.00	0.73		CDC 6400	2	Cody & Waite	2.39	0.68
	PDP/11	2	DOS 8.02	1.99	0.10		PDP/11	2	DOS 8.02	12.63	7.66
	Varian 72	2	FORT E3	1.87	0.00		Varian 72	2	FORT E3	12.69	7.31
	GP L3055	10	Cody & Waite	1.07	0.51		GP L3055	10	Cody & Waite	1.38	0.58
	DG v2.50	16	PIA (Ada)	1.00	0.57		DG v2.50	16	PIA (Ada)	0.97	0.37
	IBM/370	16	Cody & Waite	1.08	0.71		IBM/370	16	Cody & Waite	1.11	0.70
	IBM/370	16	Argonne	1.18	0.69		IBM/370	16	Argonne	1.16	0.69
2	Alliant FX/4	2	PIA (Ada)	1.97	0.00	5	Alliant FX/4	2	PIA (Ada)	0.99	0.00
	CDC 6400	2	Cody & Waite	-.--	-.--		CDC 6400	2	Cody & Waite	-.--	-.--
	PDP/11	2	DOS 8.02	-.--	-.--		PDP/11	2	DOS 8.02	-.--	-.--
	Varian 72	2	FORT E3	-.--	-.--		Varian 72	2	FORT E3	-.--	-.--
	GP L3055	10	Cody & Waite	-.--	-.--		GP L3055	10	Cody & Waite	-.--	-.--
	DG v2.50	16	PIA (Ada)	0.99	0.39		DG v2.50	16	PIA (Ada)	0.91	0.47
	IBM/370	16	Cody & Waite	-.--	-.--		IBM/370	16	Cody & Waite	-.--	-.--
	IBM/370	16	Argonne	-.--	-.--		IBM/370	16	Argonne	-.--	-.--
3	Alliant FX/4	2	PIA (Ada)	1.60	0.00	6	Alliant FX/4	2	PIA (Ada)	0.38	0.00
	CDC 6400	2	Cody & Waite	2.20	0.80		CDC 6400	2	Cody & Waite	-.--	-.--
	PDP/11	2	DOS 8.02	1.74	0.09		PDP/11	2	DOS 8.02	-.--	-.--
	Varian 72	2	FORT E3	13.54	8.55		Varian 72	2	FORT E3	-.--	-.--
	GP L3055	10	Cody & Waite	2.28	0.77		GP L3055	10	Cody & Waite	-.--	-.--
	DG v2.50	16	PIA (Ada)	0.99	0.38		DG v2.50	16	PIA (Ada)	0.62	0.12
	IBM/370	16	Cody & Waite	1.16	0.72		IBM/370	16	Cody & Waite	-.--	-.--
	IBM/370	16	Argonne	1.16	0.70		IBM/370	16	Argonne	-.--	-.--

TABLE 6.5 Typical results for SIN/COS tests

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.02	0.06	3	Alliant FX/4	2	PIA (Ada)	2.06	0.06
	DG v2.50	16	PIA (Ada)	1.17	0.62		DG v2.50	16	PIA (Ada)	1.07	0.58
2	Alliant FX/4	2	PIA (Ada)	1.99	0.08	4	Alliant FX/4	2	PIA (Ada)	2.04	0.08
	DG v2.50	16	PIA (Ada)	1.09	0.59		DG v2.50	16	PIA (Ada)	1.13	0.57

TABLE 6.6 Typical results for SIN/COS tests with period 1.0

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.00	0.00	3	Alliant FX/4	2	PIA (Ada)	1.98	0.00
	DG v2.50	16	PIA (Ada)	1.15	0.61		DG v2.50	16	PIA (Ada)	1.24	0.59
2	Alliant FX/4	2	PIA (Ada)	1.82	0.00	4	Alliant FX/4	2	PIA (Ada)	1.92	0.02
	DG v2.50	16	PIA (Ada)	1.08	0.59		DG v2.50	16	PIA (Ada)	1.10	0.59

TABLE 6.7 Typical results for SIN/COS tests with period 360.0

that were expected.

The measured errors for the TAN and COT are acceptable, although not outstanding. They are demonstrated in table 6.8. The TAN function preserves parity, but the test for small angle approximation returns non-zero values for all arguments on the hexadecimal machine. The reason for this is probably the same as for the SIN with cycle 2π. The test of the TAN with argument 11.0 revealed some problems. On the hexadecimal machine three hexadecimal digits out of six were lost and on the binary machine ten bits out of twenty-four. After looking at it more closely, it appeared that small errors in the argument reduction process were responsible for the loss in significance. The reduction scheme reduces arguments irrespective of the cycle. The case in which the cycle is given in radians is not considered separately. This can result in extra errors in the reduction of the arguments due to the fact that the argument is reduced modulo a cycle that is only represented to within working precision. Since the argument 11.0 lies in the neighbourhood of a vertical asymptote, small contaminations in the arguments have large effects on the final function value. The remaining special tests show the loss of accuracy for increasing arguments. Although the arguments are not very large the errors increase rapidly. The explanation for this is probably the same as for the previous test. We conclude here, that the range reduction for the radians case must be performed more accurately than for the general cycle case.
The error tests for zero-cycles and the argument zero for the COT are handled properly on both machines. For the argument $\pi/2$ the results were different. On the binary machine the argument lead to an error situation due to overflow. On the other hand, the hexadecimal machine accepted the argument and returned a value in order of magnitude 10^6. The difference may be explained from the wobbling precision and subsequent perturbations during the argument reduction.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.85	0.84	3	Alliant FX/4	2	PIA (Ada)	2.57	0.81
	CDC 6400	2	Cody & Waite	2.48	0.69		CDC 6400	2	Cody & Waite	2.48	0.69
	CDC 6400	2	FTN 4.6	2.70	1.06		CDC 6400	2	FTN 4.6	15.93	10.53
	DG v2.50	16	PIA (Ada)	1.24	0.78		DG v2.50	16	PIA (Ada)	1.21	0.78
	IBM/370	16	Cody & Waite	1.17	0.67		IBM/370	16	Cody & Waite	1.38	0.74
	IBM/370	16	FTX 2.2	1.25	0.73		IBM/370	16	FTX 2.2	4.97	3.61
2	Alliant FX/4	2	PIA (Ada)	2.37	0.43	4	Alliant FX/4	2	PIA (Ada)	2.62	0.85
	CDC 6400	2	Cody & Waite	2.39	0.62		CDC 6400	2	Cody & Waite	2.49	0.67
	CDC 6400	2	FTN 4.6	2.52	0.87		CDC 6400	2	FTN 4.6	-.--	-.--
	DG v2.50	16	PIA (Ada)	1.19	0.76		DG v2.50	16	PIA (Ada)	1.24	0.79
	IBM/370	16	Cody & Waite	1.30	0.88		IBM/370	16	Cody & Waite	1.29	0.82
	IBM/370	16	FTX 2.2	4.10	2.75		IBM/370	16	FTX 2.2	4.82	3.48

TABLE 6.8 Typical results for TAN/COT tests

The results for the TAN and COT function with cycle 1 and 360 are normal. The MRE as well as the RMS for all tests are comparable to the corresponding results obtained by the test program for the cycle 2π. The parity of the TAN function is also preserved here. The rest of the special tests is passed correctly. Typical results for the TAN and COT with cycles 1 and 360 are shown in table 6.9 and 6.10 respectively.

The ARCSIN and the ARCCOS seem to have accuracy problems on several test intervals. The first, second and fourth test report a MRE and a RMS that are larger than normal for binary machines.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.56	0.85	3	Alliant FX/4	2	PIA (Ada)	2.58	0.82
	DG v2.50	16	PIA (Ada)	1.37	0.85		DG v2.50	16	PIA (Ada)	1.37	0.84
2	Alliant FX/4	2	PIA (Ada)	2.14	0.45	4	Alliant FX/4	2	PIA (Ada)	2.90	0.90
	DG v2.50	16	PIA (Ada)	1.33	0.80		DG v2.50	16	PIA (Ada)	1.34	0.84

TABLE 6.9 Typical results for TAN/COT tests with period 1.0

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.62	0.81	3	Alliant FX/4	2	PIA (Ada)	2.54	0.80
	DG v2.50	16	PIA (Ada)	1.28	0.58		DG v2.50	16	PIA (Ada)	1.30	0.85
2	Alliant FX/4	2	PIA (Ada)	2.44	0.49	4	Alliant FX/4	2	PIA (Ada)	2.60	0.87
	DG v2.50	16	PIA (Ada)	1.24	0.81		DG v2.50	16	PIA (Ada)	1.29	0.85

TABLE 6.10 Typical results for TAN/COT tests with period 360.0

The MRE in the first and fourth test is about twice as big as common (a loss of one bit). The RMS should be negligible for these tests, but here it is slightly less than half a bit in the first and fourth test and a fraction of bit in the third test. Although the results of these tests on the hexadecimal machine are also somewhat larger than normal, the difference with common results is fairly small. For that rea-

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.16	0.48	4	Alliant FX/4	2	PIA (Ada)	2.19	0.45
	CDC 6400	2	Cody & Waite	0.99	0.00		CDC 6400	2	Cody & Waite	1.00	0.00
	CDC 6400	2	FTN 4.7	1.00	0.00		CDC 6400	2	FTN 4.7	1.90	0.00
	DG v2.50	16	PIA (Ada)	1.25	0.77		DG v2.50	16	PIA (Ada)	1.15	0.57
	IBM/370	16	Cody & Waite	1.00	0.64		IBM/370	16	Cody & Waite	0.99	0.29
	IBM/370	16	FTX 2.2	1.00	0.38		IBM/370	16	FTX 2.2	0.99	0.30
2	Alliant FX/4	2	PIA (Ada)	0.46	0.00	5	Alliant FX/4	2	PIA (Ada)	1.63	0.03
	CDC 6400	2	Cody & Waite	0.47	0.00		CDC 6400	2	Cody & Waite	0.66	0.00
	CDC 6400	2	FTN 4.7	0.47	0.00		CDC 6400	2	FTN 4.7	0.72	0.00
	DG v2.50	16	PIA (Ada)	0.87	0.64		DG v2.50	16	PIA (Ada)	0.88	0.55
	IBM/370	16	Cody & Waite	0.87	0.69		IBM/370	16	Cody & Waite	0.93	0.71
	IBM/370	16	FTX 2.2	0.87	0.75		IBM/370	16	FTX 2.2	0.68	0.34
3	Alliant FX/4	2	PIA (Ada)	1.98	0.28						
	CDC 6400	2	Cody & Waite	1.24	0.03						
	CDC 6400	2	FTN 4.7	1.24	0.00						
	DG v2.50	16	PIA (Ada)	1.00	0.70						
	IBM/370	16	Cody & Waite	1.00	0.72						
	IBM/370	16	FTX 2.2	1.00	0.75						

TABLE 6.11 Typical results for ARCSIN/ARCCOS tests

son the results on this machine are acceptable. Table 6.11 shows the results for the accuracy tests of the ARCSIN and ARRCOS.

In the supplemental tests the parity of the ARCSIN and the small argument approximation is preserved. The results of the other special tests all show the prescribed values.
Finally, the ARCSIN as well as the ARCCOS detect and handle error situations correctly.

The ARCTAN and ARCCOT report errors that are quite normal for these functions. The parity of the ARCTAN and the small argument approximation are preserved and the test of $arctan(y/x)$ versus $arctan(y,x)$ was also passed successfully.
The tests with the special arguments were all passed correctly.
Both functions return correct values for the special, valid, argument pairs and signal the situations where invalid arguments are passed to the functions. The subsequent exceptions are handled properly. The error statistics of the accuracy tests are demonstrated in table 6.12.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	1.00	0.00	4	Alliant FX/4	2	PIA (Ada)	1.34	0.00
	CDC 6400	2	Cody & Waite	0.84	0.00		CDC 6400	2	Cody & Waite	2.34	0.05
	CDC 7600	2	FTN 4.8	0.84	0.00		CDC 7600	2	FTN 4.8	1.95	0.02
	DG v2.50	16	PIA (Ada)	0.91	0.00		DG v2.50	16	PIA (Ada)	1.00	0.66
	IBM/370	16	Cody & Waite	1.00	0.38		IBM/370	16	Cody & Waite	1.00	0.67
	IBM/370	16	FTX 2.2	1.00	0.37		IBM/370	16	FTX 2.2	1.01	0.68
2	Alliant FX/4	2	PIA (Ada)	1.96	0.14	5	Alliant FX/4	2	PIA (Ada)	2.03	0.16
	CDC 6400	2	Cody & Waite	1.00	0.00		CDC 6400	2	Cody & Waite	-.--	-.--
	CDC 7600	2	FTN 4.8	1.00	0.00		CDC 7600	2	FTN 4.8	-.--	-.--
	DG v2.50	16	PIA (Ada)	1.00	0.41		DG v2.50	16	PIA (Ada)	1.04	0.62
	IBM/370	16	Cody & Waite	1.00	0.69		IBM/370	16	Cody & Waite	-.--	-.--
	IBM/370	16	FTX 2.2	1.04	0.73		IBM/370	16	FTX 2.2	-.--	-.--
3	Alliant FX/4	2	PIA (Ada)	1.80	0.00						
	CDC 6400	2	Cody & Waite	2.50	0.75						
	CDC 7600	2	FTN 4.8	1.86	0.39						
	DG v2.50	16	PIA (Ada)	0.92	0.53						
	IBM/370	16	Cody & Waite	0.76	0.33						
	IBM/370	16	FTX 2.2	0.76	0.30						

TABLE 6.12 Typical results for ARCTAN/ARCCOT tests

The results of the accuracy tests of the SINH and COSH, that are shown in table 6.13, are satisfactory. The fact that the first and the second test report no loss of significance in the MRE and RMS, points to an identical approximation in the test procedure and the functions being tested. The remaining test results are normal.
The parity for both the SINH and the COSH is preserved in the additional tests as well as the small argument approximation for the SINH. Both functions returned correct values for the argument zero.
Finally, all error tests were passed correctly.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	0.00	0.00	3	Alliant FX/4	2	PIA (Ada)	2.42	0.97
	CDC 6400	2	Cody & Waite	1.00	0.00		CDC 6400	2	Cody & Waite	2.36	0.98
	CDC 6400	2	FTN 4.6	1.48	0.08		CDC 6400	2	FTN 4.6	2.50	1.00
	DG v2.50	16	PIA (Ada)	0.00	0.00		DG v2.50	16	PIA (Ada)	1.21	0.78
	IBM/370	16	Cody & Waite	1.00	0.48		IBM/370	16	Cody & Waite	1.26	0.77
	IBM/370	16	FTX 2.2	1.00	0.45		IBM/370	16	FTX 2.2	1.25	0.76
2	Alliant FX/4	2	PIA (Ada)	0.00	0.00	4	Alliant FX/4	2	PIA (Ada)	1.99	0.22
	CDC 6400	2	Cody & Waite	1.00	0.00		CDC 6400	2	Cody & Waite	2.04	0.98
	CDC 6400	2	FTN 4.6	1.00	0.41		CDC 6400	2	FTN 4.6	2.19	0.99
	DG v2.50	16	PIA (Ada)	0.00	0.00		DG v2.50	16	PIA (Ada)	1.12	0.72
	IBM/370	16	Cody & Waite	1.00	0.85		IBM/370	16	Cody & Waite	1.25	0.77
	IBM/370	16	FTX 2.2	1.00	0.76		IBM/370	16	FTX 2.2	1.24	0.75

TABLE 6.13 Typical results for SINH/COSH tests

The error statistics for the accuracy tests of the TANH and COTH are normal. The results can be found in table 6.14. The parity and the small argument approximation are preserved for the TANH. The test of $tanh(x) - 1 = 0$ for large arguments and the test for special arguments were also passed successfully.
Both functions detected and handled error situations correctly.

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	2.04	0.11	3	Alliant FX/4	2	PIA (Ada)	1.76	0.09
	CDC 6400	2	Cody & Waite	1.75	0.03		CDC 6400	2	Cody & Waite	-.--	-.--
	CDC 6400	2	FTN 4.6	3.90	1.81		CDC 6400	2	FTN 4.6	-.--	-.--
	DG v2.50	16	PIA (Ada)	1.12	0.66		DG v2.50	16	PIA (Ada)	1.00	0.71
	IBM/370	16	Cody & Waite	1.05	0.72		IBM/370	16	Cody & Waite	-.--	-.--
	IBM/370	16	FTX 2.2	1.05	0.72		IBM/370	16	FTX 2.2	-.--	-.--
2	Alliant FX/4	2	PIA (Ada)	1.22	0.00						
	CDC 6400	2	Cody & Waite	1.73	0.00						
	CDC 6400	2	FTN 4.6	2.25	0.00						
	DG v2.50	16	PIA (Ada)	0.98	0.66						
	IBM/370	16	Cody & Waite	1.00	0.61						
	IBM/370	16	FTX 2.2	1.00	0.60						

TABLE 6.14 Typical results for TANH/COTH tests

Table 6.15 shows the results of the accuracy tests for the ARCSINH and ARCCOSH. Although no material for comparison was present, we believe the results to be acceptable. The additional tests for parity and small argument approximation indicate preservation of these properties. Both functions returned correct values for the special arguments.
Finally, the error situations for invalid arguments were signaled and handled properly.

The results of the accuracy tests for the ARCTANH and ARCCOTH are listed in table 6.16. Also for these functions no other results were known, but the measured errors are as expected. Therefore we consider the results to be acceptable. Parity and small argument approximation are preserved in the

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	1.90	0.00	4	Alliant FX/4	2	PIA (Ada)	1.92	0.00
	DG v2.50	16	PIA (Ada)	1.07	0.64		DG v2.50	16	PIA (Ada)	1.00	0.58
2	Alliant FX/4	2	PIA (Ada)	2.34	0.62	5	Alliant FX/4	2	PIA (Ada)	2.36	0.55
	DG v2.50	16	PIA (Ada)	1.32	0.84		DG v2.50	16	PIA (Ada)	1.20	0.77
3	Alliant FX/4	2	PIA (Ada)	1.89	0.20	6	Alliant FX/4	2	PIA (Ada)	1.75	0.21
	DG v2.50	16	PIA (Ada)	0.94	0.33		DG v2.50	16	PIA (Ada)	0.93	0.31

TABLE 6.15 Typical results for ARCSINH/ARCCOSH tests

Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS	Test	Machine	β	Library or program	Reported loss of base β digits in MRE	RMS
1	Alliant FX/4	2	PIA (Ada)	0.33	0.00	3	Alliant FX/4	2	PIA (Ada)	1.92	0.06
	DG v2.50	16	PIA (Ada)	0.05	0.00		DG v2.50	16	PIA (Ada)	1.28	0.71
2	Alliant FX/4	2	PIA (Ada)	0.99	0.00	4	Alliant FX/4	2	PIA (Ada)	2.24	0.22
	DG v2.50	16	PIA (Ada)	0.99	0.35		DG v2.50	16	PIA (Ada)	1.18	0.64

TABLE 6.16 Typical results for ARCTANH/ARCCOTH tests

supplemental tests. The inverse hyperbolic tangent returned zero for the argument zero, as it should. Lastly, the error tests were performed correctly.

6.2. Accuracy results in an Ada context

In the previous section the results of the accuracy tests of the PIA subprograms were compared with the results of the tests of Fortran subprograms. Although in general the differences between the measured errors of the various implementations were small, the comparison of the results of the Ada tests with the results of the Fortran tests is not quite satisfactory, as was discussed in section 2.1.3. Therefore it was suggested to relate the error statistics also to the model numbers. By expressing the MRE in the epsilon of the model numbers of its base type, we obtain accuracy results in an Ada context.

Again, it is not worth it to list the results of the accuracy tests for type REAL_03. As mentioned before the MRE of any test is either zero or so small that it can be ignored safely. Therefore only the results of the tests for type REAL will be stated.

On both the DG and the Alliant the attribute DIGITS yields the value six for type REAL. This implies that both machines have the same sets of model numbers for this type. The value of ϵ is ϵ = REAL'BASE'EPSILON = REAL'EPSILON = 2^{-20}.

Table 6.17 shows the results of the tests with the MRE expressed in ϵ. The tests in the table correspond with the accuracy tests as described in chapter 5. On the binary machine the measured errors are very small in all tests. The hexadecimal machine shows larger errors. Nevertheless, the results are satisfactory. The difference in the results is probably caused by the wobbling precision of the hexadecimal machine.

		Alliant FX/4 $\beta = 2$	DG v.2.50 $\beta = 16$			Alliant FX/4 $\beta = 2$	DG v.2.50 $\beta = 16$
Function	Test	MRE	MRE	Function	Test	MRE	MRE
SQRT	1	0.08 ε	0.25 ε	COT	1	0.38 ε	1.95 ε
	2	0.12 ε	0.00 ε	*cycle* 1	1	0.46 ε	2.55 ε
				cycle 360	1	0.37 ε	2.26 ε
LOG	1	0.19 ε	1.00 ε				
	2	0.22 ε	0.97 ε	ARCSIN	a	0.27 ε	1.99 ε
	3	0.24 ε	1.27 ε		b	0.24 ε	1.00 ε
	4	0.36 ε	1.02 ε				
	5	0.12 ε	0.31 ε	ARCCOS	a	0.08 ε	0.69 ε
					b	0.28 ε	1.53 ε
EXP	1	0.19 ε	1.00 ε		c	0.19 ε	0.73 ε
	2	0.19 ε	0.99 ε				
	3	0.19 ε	1.20 ε	ARCTAN	1	0.12 ε	0.77 ε
					2	0.24 ε	0.99 ε
"**"	1	0.00 ε	0.00 ε		3	0.21 ε	0.80 ε
	2	0.12 ε	0.97 ε		4	0.15 ε	1.00 ε
	3	0.12 ε	0.97 ε				
	4	0.13 ε	1.34 ε	ARCCOT	1	0.25 ε	1.12 ε
SIN	a	0.21 ε	0.99 ε	SINH	a	0.00 ε	0.00 ε
	b	0.19 ε	0.97 ε		b	0.33 ε	1.79 ε
	c	0.12 ε	0.78 ε				
cycle 1	a	0.25 ε	1.60 ε	COSH	a	0.00 ε	0.00 ε
	b	0.26 ε	1.23 ε		b	0.24 ε	1.38 ε
cycle 360	a	0.25 ε	1.54 ε				
	b	0.24 ε	1.93 ε	TANH	1	0.25 ε	1.40 ε
					2	0.14 ε	0.95 ε
COS	a	0.24 ε	0.98 ε				
	b	0.18 ε	0.92 ε	COTH	1	0.21 ε	1.00 ε
	c	0.08 ε	0.35 ε				
cycle 1	a	0.24 ε	1.27 ε	ARCSINH	1	0.23 ε	1.22 ε
	b	0.25 ε	1.44 ε		2	0.31 ε	2.40 ε
cycle 360	a	0.22 ε	1.26 ε		3	0.23 ε	0.84 ε
	b	0.23 ε	1.31 ε				
				ARCCOSH	a	0.23 ε	1.00 ε
TAN	1	0.45 ε	1.96 ε		b	0.32 ε	1.72 ε
	2	0.32 ε	1.72 ε		c	0.21 ε	0.83 ε
	3	0.37 ε	1.77 ε				
cycle 1	1	0.37 ε	2.76 ε	ARCTANH	1	0.07 ε	0.08 ε
	2	0.27 ε	2.46 ε		2	0.12 ε	0.99 ε
	3	0.37 ε	2.76 ε		3	0.23 ε	2.18 ε
cycle 360	1	0.38 ε	2.17 ε				
	2	0.34 ε	1.93 ε	ARCCOTH	1	0.29 ε	1.64 ε
	3	0.36 ε	2.27 ε				

TABLE 6.17 Accuracy results expressed in REAL'BASE'EPSILON

7. CONCLUSIONS AND RECOMMENDATIONS

Upon the whole the library of elementary functions has an accuracy that can be qualified as satisfactory to good. With a few exceptions the error statistics reported by the test programs are of normal magnitude on both the binary and the hexadecimal machine.

The substantial loss of significance in the computation of the tangent in the neighbourhood of an asymptote can be reduced by improving the argument reduction process. By providing a separate, accurate reduction scheme when the cycle is given in radians, the errors in this process can be kept

small. Although the function is very sensitive to small errors in the neighbourhood of an asymptote it is possible to limit the loss of precision, particularly when the arguments are not too large.
Another case that should be looked at more closely is the computation of the ARCSIN and ARCCOS. Although the test results are not alarming, they are larger than normal. The application of some fine tuning would probably improve the error statistics here as well.

REFERENCES

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

ACM SIGAda Numerics Working Group, *Draft 1.0 of a Proposed Standard for a Generic Package of Elementary Functions for Ada*, February 1989

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

Bergman, M. *Implementation of elementary functions in Ada*, CWI Report NM-R8709, 1987

Cody, W.J. *ELEFUNT Test results under X1.4 on the Encore Multimax*, Technical Memorandum ANL/MCS-TM-68, 1986

Cody, W.J. *ELEFUNT Test results under FX/Fortran Version 1.0 on the Alliant FX/8*, Technical Memorandum ANL/MCS-TM-78, 1986

Cody, W.J. *ELEFUNT Test results under NS32000 Fortran V2.5.3 on the Sequent Balance*, Technical Memorandum ANL/MCS-TM-80, 1986

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

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

Kok, J. *Proposal for standard mathematical functions in Ada*, CWI Report NM-R8718, 1987

Symm, G.T., Wichmann, B.A., Kok, J. and Winter, D.T. *Guidelines for the design of large modular scientific libraries in Ada*, in : Ford, B., Kok, J. and Rogers, M.W. (eds.) *Scientific Ada*, Cambridge University Press, 1986

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