A PORTABLE GENERIC ELEMENTARY FUNCTION PACKAGE IN
ADA AND AN ACCURATE TEST SUITE

by

Ping Tak Peter Tang

Mathematics and Computer Science Division

November 1990

This work was supported in part by the Strategic Defense Initiative Organization, Office of the Secretary of Defense, under PMA 2300 and also in part by the STARS Program Office, under AF Order RESD-632.
# Contents

Abstract 1

1 Introduction 1

2 Design and Implementation of the Function Package 2
  2.1 Software Engineering Issues 3
    2.1.1 Generic Packages and Range Constraints 4
    2.1.2 Accurate Implementation and Extra-Precise Arithmetic 6
    2.1.3 Portability and Efficiency 8
    2.1.4 Modularity and Maintenance 9
  2.2 Design of the Package 10
  2.3 Algorithms 12
  2.4 Error Analysis 16

3 An Accurate and Efficient Test Suite 18
  3.1 Testing Methodology 19
    3.1.1 Extra-Precise Representation 20
    3.1.2 Backward Argument Reduction 20
    3.1.3 Perturbation 21
    3.1.4 Breakpoints in Primary Intervals 21
    3.1.5 Testing Inverses 23
    3.1.6 Testing Ratio of Functions 23
  3.2 Structure of the Test Suite 24
    3.2.1 Kernel Tests 24
    3.2.2 Application of the Kernels 25

4 Test Results 26
  4.1 Portability of the Function Package 27
  4.2 Calibration of the Test Package 27
  4.3 Testing Various Function Packages 28
    4.3.1 Testing Tang's Function Package 28
    4.3.2 Testing Vendors' Function Packages 30

5 Conclusion 33

Acknowledgments 33

References 34
A Portable Generic Elementary Function Package in Ada and an Accurate Test Suite

Ping Tak Peter Tang

Abstract

A comprehensive set of elementary functions has been implemented portably in Ada. The high accuracy of the implementation has been confirmed by rigorous analysis. Moreover, we present new test methods that are efficient and offer a high resolution of 0.005 unit in the last place. These test methods have been implemented portably here and confirm the accuracy of our implemented functions. Reports on the accuracy of other function libraries obtained by our test programs are also presented.

1 Introduction

Since the adoption of IEEE Standard 754 for binary floating-point arithmetic, there has been a revival of interest in implementing highly accurate elementary function libraries (see [2], [8], [17], [24], [25], and [26]). Our recent work ([24], [25]) has shown that lucid error analysis can be performed on these implementations to yield tight error bounds. Typically, those bounds do not exceed the maximum error observed by more than 0.02 units in the last place (ulp).

The algorithms presented in these recent works seem to be best implemented in a machine-dependent manner, not only because some of the operations required are quite low level (such as extracting the exponent value of a floating-point number), but also because some of the constants needed must be made available exactly to every single significant bit. Two questions arise. First, can a portable implementation in some appropriate high-level language offer accuracy comparable to a machine-dependent implementation in a low-level language? Second, can the accuracy of the portable implementation be analyzed based on the semantics of the language of choice, with minimal assumptions about the specific hardware used? In other words, can one provide a portable implementation where the analysis is also portable? With the language Ada, we answer both questions affirmatively.

This paper concerns a library of elementary functions that we have implemented in Ada. The main feature that separates Ada from other languages is the floating-point
model ([4] and [1]) that it supports. The semantics of floating-point operations and conversions is the foundation of our portable implementation and analysis. Moreover, Ada's facility of "generic package" allows us to implement, in just one source code, a library of 29 functions that can accommodate precisions ranging from roughly 6 decimal digits to 33. In particular, the library can support IEEE single, double, and double-extended precisions; IBM 370 single, double, and quadruple precisions; and VAX single, double (both D- and G-formats), and quadruple (H-format) precisions.

A portable elementary function library, even one that has been analyzed rigorously, achieves only half of our goal, however. Constructing a test program that can accurately evaluate the accuracy of a function library is much more challenging than constructing the library in the first place, especially when the library in question is highly accurate (the functions in [24], [25], and [26] are typically accurate to around 0.55 ulp). For the functions exp and log, we proposed in [21] portable and efficient test methods that offer a resolution of 0.005 ulp. By carefully extending our work there, we have implemented a test package for the function library, which, to the best of our knowledge, is the only practical test program that offers such high resolution. (For a discussion of commonly used test methods, see [21].) This portable test package can test libraries with accuracies ranging from 6 to 33 digits. Indeed, this test package is likely to contribute more than our elementary function library, since different compiler vendors may eventually choose to optimize the latter for specific machines by assembly coding and other means. Our vision is that while our function package will set a goal for accuracy, our test package will serve as a benchmark that evaluates the quality of libraries which often claim to be "new and improved."

The rest of the paper is organized as follows. Section 2 describes the implementation of the function package—its structure, the algorithms used, and error analysis techniques. Section 3 describes the underlying techniques in implementing the test suite. Section 4 reports our experience in compiling our function packages on a number of different hardware/software environments. We also report test results from running our test suite on our functions as well as on other math libraries provided by Ada vendors. Section 5 presents some concluding remarks about our experience with implementing mathematical software in Ada.

2 Design and Implementation of the Function Package

The creators of Ada tried to equip it to be a fine language for portable numeric programming by incorporating into Ada a formal floating-point model [4] that is further supported by numerous environmental inquiries [1]. It is therefore somewhat ironic that Ada has not included any of the commonly used elementary functions such as exp, log, and sin. Since these functions are indispensable in many engineering and scientific computations, compiler vendors have typically supplied them in the form of math libraries. Unfortunately, these libraries differ in (1) the functions that they provide; (2) the calling conventions, which include the parameter profile of the functions, the name of the func-
tions, and even the name of the library itself; and (3) the use or avoidance of genericity. Thus, most Ada codes that use those elementary functions are not portable.

This undesirable situation led to the pioneering works of [15] and [14] and the more recent work by the SIGAda Numerics Working Group which proposes a standard specification for a package of elementary functions for Ada [18] and [13]. At present, the proposal is on its way through a hierarchy of committees toward achieving the status of a secondary ISO standard. A detailed rationale of the proposed standard has been written by Dritz [7]. To describe in a self-contained manner our portable implementation of the proposed standard, we summarize its features that concern us the most.

The package as specified consists of 29 functions that can be classified into three families: exponential, trigonometric, and hyperbolic.

The exponential family consists of 5 functions: \( \text{Sqrt}(x) \), \( \text{Exp}(x) \), \( x^y \), \( \text{Log}(x) \), and \( \text{Log}(x,\text{Base}) \). The \( \text{Sqrt}(x) \), \( \text{Exp}(x) \), and \( \text{Log}(x) \) are the usual square root function \( \sqrt{x} \), exponential function \( e^x \), and the natural logarithm function \( \log x \). The two-argument log function \( \text{Log}(x,\text{Base}) \) allows the user to specify an arbitrarily chosen base, e.g., 2.0 or 10.0. The exponentiation operator allows one to compute \( x^y \).

The trigonometric family consists of 16 functions. The four usual trigonometric functions \( \sin(x) \), \( \cos(x) \), \( \tan(x) \), and \( \cot(x) \) and the four inverses \( \text{Arcsin}(x) \), \( \text{Arccos}(x) \), \( \text{Arctan}(Y,X) \), and \( \text{Arccot}(X,Y) \) constitute 8 of the 16. (The function \( \text{Arctan}(Y,X) \) is the analogue of the Fortran intrinsic \( \text{Atan2}(Y,X) \).) These 8 functions are approximations to the corresponding mathematical functions, with \( 2\pi \) being the period (for the forward functions). The specification also requires 8 other functions that allow the user to specify a period (e.g., 360.0). The calling sequences are \( \text{Sin}(X,\text{Cycle}) \), \( \text{Cos}(X,\text{Cycle}) \), ..., and \( \text{Arccot}(Y,X,\text{Cycle}) \). It will become clear later on why the first 8 functions are not eliminated by assigning suitable default values to the parameter \( \text{Cycle} \) in the last 8 functions (see also [7]).

Finally, the hyperbolic family consists of 8 functions: the commonly used \( \sinh(x) \), \( \cosh(x) \), \( \tanh(x) \), and \( \coth(x) \) and the corresponding inverses \( \text{Arccosh}(x) \), \( \text{Arccosh}(x) \), \( \text{Arctanh}(x) \), and \( \text{Arccoth}(x) \).

In the remainder of this section, we discuss several software engineering issues concerning portable generic implementations of the proposed standard, the design of our function package, and algorithms. The section concludes with a presentation of a method of analysis to estimate the accuracy of the implementation.

### 2.1 Software Engineering Issues

Our goal is to implement a portable library that satisfies the proposed standard specifications. Consequently, we must satisfy three requirements: (1) portability (self-imposed), (2) genericity, and (3) functionality (of all 29 functions). These requirements give rise to several software engineering issues that must be settled regardless of the specific algorithms for the individual functions.
2.1.1 Generic Packages and Range Constraints

The proposed specification requires that the library of elementary functions be generic and provide an accuracy comparable to the base type of the generic actual type. What type, then, should be used inside the generic package for computations? The generic actual type is unsuitable because it may have a range constraint that can be violated during intermediate calculations in the package. This violation would then cause a constraint error to be raised, even if the final result would have been perfectly acceptable had the exception not occurred.

Clearly, a constraint-free type with a precision comparable to that of the base type of the generic actual type must be used within the generic package body. Let WORKING_FLOAT be such an ideal type, and let FLOAT.TYPE be the generic formal type. How can the type WORKING_FLOAT be declared? One may try declaring

type WORKING_FLOAT is digits FLOAT.TYPE'BASE'DIGITS.

Unfortunately, this approach does not work because FLOAT.TYPE'BASE'DIGITS is a non-static expression. A workable solution, but impractical for portable implementations, would be to perform a case analysis as follows:

```haskell
case FLOAT.TYPE'BASE'DIGITS is
  when 1 =>
    declare
      type WORKING_FLOAT is digits 1;
    begin
      -- complete calculation of F in this case
      -- approximation accurate to at least 1 digit
    end;
  when 2 =>
    declare
      type WORKING_FLOAT is digits 2;
    begin
      -- complete calculation of F in this case
      -- approximation accurate to at least 2 digits
    end;
  when SYSTEM.MAX_DIGITS =>
    declare
      type WORKING_FLOAT is digits SYSTEM.MAX_DIGITS;
    begin
      -- complete calculation of F in this case
      -- approximation accurate to at least
      -- SYSTEM.MAX_DIGITS digits
```
For any portable implementation intended to accommodate systems whose attribute `SYSTEM.MAX_DIGITS` is 15 or larger, the solution just proposed leads to a huge code. As a practical solution, we condense the many cases as follows:

```plaintext
case FLOAT_TYPE'BASE'DIGITS is
when 1..6 =>
  declare
    type WORKING_FLOAT is digits
    (6+SYSTEM.MAX_DIGITS - abs(6-SYSTEM.MAX_DIGITS))/2;
    -- the expression above is MIN( 6, SYSTEM.MAX_DIGITS )
  begin
    -- complete calculation of F in this case
    -- approximation accurate to at least 6 digits
  end;
when 7..15 =>
  declare
    type WORKING_FLOAT is digits
    (15+SYSTEM.MAX_DIGITS - abs(15-SYSTEM.MAX_DIGITS))/2;
  begin
    -- complete calculation of F in this case
    -- approximation accurate to at least 15 digits
  end;
when 27..33 =>
  declare
    type WORKING_FLOAT is digits
    (33+SYSTEM.MAX_DIGITS - abs(33-SYSTEM.MAX_DIGITS))/2;
  begin
    -- complete calculation of F in this case
    -- approximation accurate to at least 33 digits
  end;
when others =>
  -- cannot handle this case
end case;
```

This method guarantees that

```
WORKING_FLOAT'DIGITS \geq FLOAT_TYPE'BASE'DIGITS
```

always. Note that equality for (1) holds whenever the right boundary of the case coincides with a predefined type of the machine on which the code runs; and whenever that is the
case, we can avoid using a type that is unnecessarily more accurate. Thus, we have chosen the cases such that, on all the Ada systems with which we have experience,

\[ \text{WORKING\_FLOAT\_DIGITS} = \text{FLOAT\_TYPE\_BASE\_DIGITS} \]  

(2)

for all possible FLOAT\_TYPEs.

2.1.2 Accurate Implementation and Extra-Precise Arithmetic

A typical implementation of an elementary function involves three steps: argument reduction to a primary interval, approximation of the function on that primary interval, and reconstruction of the function value. For accurate implementations, it is standard practice to perform argument reduction and reconstruction in an extra-precise data type. When such a data type is unavailable, extra precision is simulated in software by using the working-precise data type (cf. [5], [10]). Therefore, an elementary function package following that practice would try to exploit extra-precise data types whenever they are available, and would resort to simulation when they are not.

In principle, a portable generic package is able to detect at elaboration time whether an extra-precise data type is available and, consequently, can ensure that appropriate actions be taken when the function is invoked later. In practice, however, because of technicalities in constructing a portable generic package, such an approach will lead to a huge code jammed with several sets of (otherwise unnecessary) constants used for argument reduction and reconstruction and with many complicated branches.

Fortunately, two practical alternatives exist: one slightly more efficient, the other noticeably more accurate. We have chosen the latter. In what follows, we describe and compare the two methods, examine the cost of the method that we adopted, and explain how to switch from the adopted method to the other.

Two Methods

In Subsection 2.1.1, we described how we could declare the type WORKING\_FLOAT that corresponds to the same predefined floating type of the generic actual type. Thus, an acceptably accurate implementation can compute solely in WORKING\_FLOAT and perform simulation of extra precision at critical places. This method has one advantage and two disadvantages.

- **Advantage:**
  The cost is minimal. The implementation is no more expensive than an acceptably accurate implementation need be. The price of using simulation has to be paid even when the package is not required to be generic or portable.

- **Disadvantages:**
1. This method leads to a large body of code. The reason is that the three steps in the implementation — argument reduction, approximation, and reconstruction — must be included in each of the different cases of WORKING_FLOATs (cf. Subsection 2.1.1).

2. When extra precision is available without simulation, this method is less accurate than that of using the readily available higher-precision arithmetic. The reason is that, because of the high cost, extra-precision simulation is done only in a few of the many places where higher precision would enhance accuracy noticeably.

To overcome the disadvantages, we have taken an approach that uses unsimulated extra precision whenever it is available. Because of portability and genericity, the only convenient extra-precise type, named, say, COMMON_FLOAT, is the type with the maximum allowable number of digits. Moreover, because we cannot determine a priori whether WORKING_FLOAT leaves us with any extra-precise type, the implementation must simulate extra-precision operations as well. Let us consider the advantages and disadvantages of this approach.

- Advantages:
  1. The resulting code is compact. The reason is that the code for both argument reduction and reconstruction need appear only once. These two steps compute solely in COMMON_FLOAT and work for all possible generic actual types.
  2. The accuracy is enhanced in general. When

      COMMON_FLOAT'DIGITS > WORKING_FLOAT'DIGITS,

      the result obtained would be more accurate than that obtained from computations done solely in WORKING_FLOAT, even with the help of extra-precision simulation.

- Disadvantages:
  1. Work is duplicated in this approach. When

      COMMON_FLOAT'DIGITS > WORKING_FLOAT'DIGITS,

      the simulation of extra precision is rendered redundant by the use of the type COMMON_FLOAT.
  2. The approach may be unaffordably inefficient. It is conceivable that in some systems, operations in COMMON_FLOAT are extremely inefficient. For example, operations on the H-format (113 significant bits) data types supported by the VMS operating systems may be implemented in software on some particular machines.
We have chosen the second approach because of the higher accuracy it offers and the compact code that results from it. We explain below why the approach is justified on most systems, and we show how our code can be modified slightly to give the first approach.

Cost

How often is work being duplicated in our approach? On systems with only two pre-defined floating types, duplication occurs only half of the time. In those undesirable cases, the cost of the unnecessary effort is only approximately five multiplications in COMMON_FLOAT. Moreover, operations in COMMON_FLOAT are efficient whenever they are implemented in hardware. Consequently, on machines with only two predefined floating types, both supported in hardware, our implementation is justified. All but two Ada systems that we know of fall into this category.

Alternative

On systems such as the IBM/370 or VAX under VMS, there are usually three predefined floating types. If calculations in the widest format are excessively expensive, implementors can easily incorporate the argument reduction and the reconstruction into the approximation step of the code. By doing so, all calculations will be performed in the base type of the generic actual type.

2.1.3 Portability and Efficiency

The issues of portability and efficiency are best illustrated by two specific examples. In calculating \( e^x \), a first step usually involves transforming the argument \( x \) to a value \( r \) in the interval \([-\log 2/2, \log 2/2]\) via

\[
x = n \cdot \log x + r,
\]

where \( n \) is an integer. The mathematical relationship

\[
e^x = 2^n e^r
\]

allows us to reduce the problem to finding an approximation on a small interval and scaling a floating-point value by an integer power of 2. The latter, on most machines, is a trivial low-level operation; it involves modification of the exponent field of a floating-point number. To accomplish the same task by portable Ada, however, is extremely inefficient, as it involves executing a loop \( n \) times. (Even with the use of repeated squaring, significant overhead must be paid to avoid spurious overflow/underflow.) Another example is that of calculating \( \log(x) \). Again, the usual first step is to decompose the floating-point number \( x \) into

\[
x = 2^n * y,
\]
where \( 1/\sqrt{2} \leq y \leq \sqrt{2} \), typically. Hence,

\[
\log(x) = n \times \log 2 + \log(y)
\]
can be computed easily, provided an approximation to \( \log \) on \([1/\sqrt{2}, \sqrt{2}]\) is found. Clearly, decomposing a floating-point \( x \) into \( n \) and \( y \) is an inexpensive low-level operation, but not in portable Ada. One must not misinterpret the situation to conclude that Ada is incapable of expressing low-level operations efficiently—for its "unchecked conversion" and "representation" facilities can be used to manipulate floating-point objects efficiently. The crux of the matter is portability; any Ada code (or C code, for that matter) that exploits the specific floating-point representation of the hardware cannot be portable across different hardwares. The dilemma seems to be portability but inefficiency on the one hand and efficiency but machine dependency on the other.

Our solution to this problem is to identify a set of commonly used low-level functionalities and define a separate package specification for them. The package of elementary functions can then be implemented portably based on the package GPF (for generic primitive functions) of low-level functionalities.

The SIGAda Numerics Working Group is currently completing the proposed standard specification for GPF. The idea is that these low-level (or primitive) functionalities are so commonly used that each system should have a machine-specific implementation that is an efficient as possible. With the package GPF, the elementary function package can achieve efficiency without sacrificing portability.

### 2.1.4 Modularity and Maintenance

Modularity is a principle of good software design. Consequently, we have heavily used the facilities of "sub units" and "separate bodies" in our implementation. The structure of our package consists of the "spec" which is the proposed standard and a "body" which instantiates the generic primitive functions with Common_Float (see Section 2.1.1) and declares the essential objects (such as functions and commonly used constants). The actual implementation of the functions is given in "separate bodies." This methodology saves much recompilation time during our development.

Library maintenance is a crucial consideration in software engineering. We have therefore implemented our function library based on "kernel" functions. The idea is that instead of implementing every function independently, we implement a number of crucial functions—in the sense that many other functions can be implemented based on straightforward use of these crucial ones (hence the term "kernel"). Consequently, any future improvements of these kernel functions will automatically improve the whole library. Moreover, because the implementation of the nonkernel functions is simple, errors are much less likely to occur. In the next section, we describe the specific design of the package and, in particular, the collection of kernels.
2.2 Design of the Package

The idea of kernel functions can be best illustrated by specific examples. Consider the function \( \sinh(x) \) which is mathematically given by

\[
\sinh(x) = \frac{1}{2}(e^x - e^{-x}).
\]

Suppose the underlying hardware has, say, 24 significant bits. Clearly, for \( x > 12\log_2 \),

\[
\sinh(x) = \frac{1}{2}e^x
\]

for all practical purposes. Hence, the following implementation may seem reasonable for such input arguments:

```c
if (x > Threshold)
    return( 0.5*Exp(x) );
else
    return( 0.5*(z - 1.0/z) );
```

There is one problem, however. For those \( x \)'s such that \( \exp(x) \) overflows but \( \frac{1}{2}\exp(x) \) does not, the implementation above will overflow prematurely. The problem can be solved easily if, instead of using the function \( \exp \), we have at our disposal a kernel procedure \( KP_{Exp} \) that returns a floating-point value \( z \) and an integer \( N \) such that \( 2^N \cdot z \) approximates \( e^x \) accurately. Thus the value \( \frac{1}{2}e^x \) can be obtained without premature overflow by scaling \( z \) with \( 2^{N-1} \). Clearly, \( \exp \) would now be obtained by calling \( KP_{Exp} \) and then scaling the value \( z \) by \( 2^N \).

The example of \( \sinh(x) \) illustrates the need for another kernel function. Suppose \( x \approx \epsilon \), the unit roundoff of the underlying hardware, and hence there is no danger of premature overflow. The straightforward implementation

```c
z := Exp(x);
return( 0.5 * (z - 1.0/z) );
```

will yield highly inaccurate result caused by cancellation between \( z \) and \( 1.0/z \). One way to handle this case is to find polynomial or rational approximations to \( \sinh(x) \) for \( x \) near 0 (see [5], for example). Our approach is different. The problem here is similar to \( \tanh(x) \) for \( x \) near 0; and both problems can be solved if there is a kernel function, say, \( KP_{Em1} \), that delivers the value \( e^x - 1 \) accurately. The formulae

\[
\sinh(x) = \frac{1}{2}\left\{(e^x - 1) + (e^{-x} - 1) \right\} \div \left\{(e^x - 1) + 2 \right\}
\]

and

\[
\tanh(x) = (e^{2x} - 1) \div \left\{(e^{2x} - 1) + 2 \right\}
\]

clearly avoid catastrophic cancellations for those input values in question. Moreover, \( KP_{Exp} \) and \( KP_{Em1} \) can be implemented by sharing common resources. The details of these implementations are given in [24] and [26].
Another example would be the implementation of the trigonometric functions Sin, Cos, Tan, and Cot. These four functions can all be easily constructed by using kernel functions KF.Sin and KF.Cos that yield sin and cos, respectively, for arguments in $[-\pi/4, \pi/4]$. The reason is that as soon as one can decompose an input argument $x$ into

$$x = n \cdot \left(\pi/2\right) + r, \quad |r| \leq \pi/4,$$

sin $x$ and cos $x$ are given by one of the four values $\sin(r)$, $-\sin(r)$, $\cos(r)$, or $-\cos(r)$, depending on the value $i = n \mod 4$. (Details can be found in [23].) Clearly, tan($x$) and cot($x$) are appropriate ratios of sin and cos. By suitable transformations of the reduced argument $r$ in

$$x = n \cdot \text{Cycle} + r,$$

the two kernel functions KF.Sin and KF.Cos can be used to implement the other four trigonometric functions with user-specified periods.

Of the 29 functions required by the proposed standard, 27 are constructed based on kernel functions. (The functions Sqrt($X$) and $X^{**}Y$ are implemented as independent units.) The eight kernels below, in the form of procedures and functions, are the building blocks of the whole library.

- KP.Exp($X$, $N$, $Z_1$, $Z_2$) and KF.Em1($X$).

  The kernel procedure KP.Exp produces an integer $N$ and floating-point values $Z_1$ and $Z_2$ such that

  $$e^x \approx 2^N (Z_1 + Z_2),$$

  where $Z_1$, when expressed in base 2, has no more than 10 significant bits. The idea here is that the unrounded sum $Z_1 + Z_2$ approximates $2^{-N}e^X$ to 4 or 5 bits more than the type Float.Type'Base. Moreover, although scaling by $2^N$ may not be exact on hexadecimal machines in general, the trailing zeros in $Z_1$ guarantee that scaling it by $2^N$ will be error free.

  The kernel function KF.Em1 returns the value $e^X - 1$ to the precision of the base type of Float.Type.

  These two kernels are used to implement Exp($X$), Sinh($X$), Cosh($X$), Tanh($X$), and Coth($X$).

- KP.Log($X$, $N$, $Z_1$, $Z_2$) and KF.Llp($X$).

  The kernel procedure KP.Log produces an integer $N$ and floating-point values $Z_1$ and $Z_2$ such that

  $$\log(X) \approx N \log 2 + Z_1 + Z_2.$$

  The idea is that $Z_1 + Z_2$ approximates $\log(2^{-N}X)$ to extra precision. Moreover, the trailing zeros in $Z_1$ (see the previous description of KP.Exp) allows accurate computation in special situations such as $\log_{10}$ and $\log_2$. 

---

11
In the spirit of KF.Em1, the kernel function KF.L1p returns the value \( \log(1 + x) \) accurately.

These two kernels are used to implement \( \log(X) \), \( \log(X, \text{Base}) \), \( \text{Arcsinh}(X) \), \( \text{Arccosh}(X) \), \( \text{Arctanh}(X) \), and \( \text{Arccoth}(X) \).

- **KP.Pi2Rd(X,R1,R2,I)**.
  The kernel procedure KP.Pi2Rd returns accurate reduced arguments \( R_1 \) and \( R_2 \),
  \[
  X \approx N(\pi/2) + (R_1 + R_2), \quad |R_1 + R_2| \leq \pi/4,
  \]
  and the value \( I = n \mod 4 \). This kernel is used in the forward trigonometric functions.

- **KF.Sin(X) and KF.Cos(X)**.
  The two kernel functions KF.Sin and KF.Cos return \( \sin(X) \) and \( \cos(X) \), respectively, for \( |X| \leq \pi/4 \). These two kernel functions are used in all the forward trigonometric functions.

- **KP.Atn(Y,X,Z1,Z2)**.
  The kernel procedure KP.Atn produces floating-point values \( Z_1 \) and \( Z_2 \) such that
  \[
  \arctan(Y/X) \approx Z_1 + Z_2, \quad 0 < Y < X.
  \]
  This kernel is used in all the inverse trigonometric functions.

### 2.3 Algorithms

We now describe the essence of the algorithms used to implement the kernel functions.

- **KP.Exp and KF.Em1**.
  Our previous work [24], [26] contains the detailed description and analysis of the algorithm; we summarize the key features here.
  The algorithm consists of three steps:

  **Reduction**: For a given value \( x \), find an integer \( n \), such that
  \[
  x = m(\log 2/32) + r,
  \]
  where \( |r| \leq \log 2/64 \) and \( m = 32n + j, j = 0, 1, 2, \ldots, 31 \).

  **Approximation**: Approximate \( e^r \) by a polynomial
  \[
  1 + p(r) \approx e^r.
  \]
Reconstruction: Reconstruct $e^x$ or $e^x - 1$ via

$$
e^x = \frac{2^j}{32}e^r = 2^n \cdot \frac{2^j}{32} \cdot e^r 
\approx 2^n (2^{j/32} + 2^{j/32}p(r));
\quad e^x - 1 \approx 2^n (2^{j/32} + 2^{j/32} p(r) - 2^{-n}).$$

Ada's semantics of literal conversions allows us to store the 32 values $2^{j/32}, j = 0, 1, \ldots, 31$, to extra precision. The idea is that each of those values is stored in two variables, the first of which has the 12 leading bits of $2^{j/32}$ expressed as a base-16 literal. Hence, for any generic formal type whose safe numbers have at least 12 bits, that leading part is guaranteed to be converted without error. (For all practical Ada systems, safe numbers contain at least 21 bits.) Provided the second part is expressed as a long literal, we are able to represent the 32 values each to roughly 12 bits more than the generic formal type.

Another fine illustration of use for Ada's peculiar feature is found in the reduction step. The simple formula

$$x = m(\log 2^{j/32}) + r$$

suggests $r$ is obtained by

$$r := x - m(\log 2^{j/32}).$$

But it is well known (see [5]) that absolute error made in $r$ will become relative error in the final computed result. Hence the number of bits we need to know about $\log 2^{j/32}$ must be at least working precision plus the number of bits in the exponent field of working precision (see [24] for details). Once again, by using Ada's various attribute information, we can select two appropriate constants that give us sufficient information about $\log 2^{j/32}$. Moreover, by using Ada's "sequence of statements" in the package body, we can make the selection once and for all at elaboration time.

- KP.Log and KFLlp.

Our previous work [25] contains the detailed description of the algorithm and analysis. We summarize the key features here.

The algorithm consists of three steps:

**Reduction:** For a given value $x$, decompose $x$ (or $1 + x$ for KFLlp) into an integer $n$ and $f_1$ and $f_2$ such that

$$x(1 + x) = 2^n(f_1 + f_2),$$

where $1 \leq f_1 + f_2 < 2$, $f_1 = 1 + j/64$, $j = 0, 1, \ldots, 64$, and $|f_2| \leq 1/128$.  

13
Approximation: Approximate \( \log(1 + f_2/f_1) \) by a polynomial in \( r \) where

\[
r = \frac{2f_2}{2f_1 + f_2}.
\]

The idea is that

\[
\log \left(1 + \frac{f_2}{f_1}\right) = \log \left(\frac{1 + \frac{r}{2}}{1 - \frac{r}{2}}\right) \\
= 2 \left(\frac{r}{2} + \frac{1}{3} \left(\frac{r}{2}\right)^3 + \frac{1}{5} \left(\frac{r}{2}\right)^5 + \ldots\right) \\
\approx r + r^3p(r^2).
\]

Reconstruction: Reconstruct \( \log(x) \) or \( \log(1 + x) \) via

\[
\log(2^n(f_1 + f_2)) = n \log 2 + \log(f_1) + \log(1 + f_2/f_1) \\
\approx n \log 2 + \log(f_1) + r + r^3p(r^2).
\]

\* KP\_Pi2Rd.

The kernel procedure KP\_Pi2Rd reduces a given input \( x \) to the interval \([−π/4, π/4]\) via

\[
x = n(π/2) + r, \quad |r| ≤ π/4.
\]

In addition to \( r \), the value \( n \mod 4 \) is also returned. We first explain the need for this procedure.

Given machine numbers \( x \) and \( p \), the operator “rem” provided in the package GPF yields an exact remainder \( r \) such that

\[
x = np + r, \quad |r| ≤ p/2.
\]

For those who are familiar with the IEEE standard, this “rem” is indeed the required remainder operation. Since we implemented our function package using GPF, and since the “rem” operator it provides is likely to be efficient, why would we need the procedure KP\_Pi2Rd when, say,

\[
P := 1.5707963267948966192313; \quad π/2
\]

\[
R := X \text{ rem } P;
\]

seem sufficient?

A simple example explains the situation. For argument’s sake, assume we are implementing \( \cos \) on a decimal machine with 6 digits. Suppose \( x = 1.57079 \). Since

\[
π/2 - x = 10^{-6} × 6.32679489661\ldots,
\]
\[ \cos(x) \text{ rounded to 6 decimal digits is} \]
\[ 10^{-6} \times 6.32679. \]

If we employ the technique
\[
P := 1.5707963267948966192313; \\
r := x \text{ rem } P;
\]
then \( r \) would be zero! The reason is that \( P \), being a machine number, is exactly equal to \( x \). The resulting computed value of \( \cos(x) \) would carry no correct significant digits. Clearly the reduction must be performed with a value that approximates \( \pi/2 \) to at least double the working precision. In fact, the number of digits needed for \( \pi/2 \) increases as the magnitude of \( x \) increases. For example, the reduction scheme in [19] uses a value of \( \pi/2 \) that is 4000 bytes long.

In the proposed standard for the function package, the SIGAda Numerics Working Group adopted a compromise that, in effect, requires an accurate reduction with respect to \( \pi/2 \) only when
\[ |x| \leq 2^{L/2}, \]
where
\[ L = \text{Float_Type'Base'Hantissa}. \]

\( \text{XP}\_\text{Pi2Rd} \) is the procedure that performs an accurate reduction in this range.

The algorithm has four steps, the first of which is carried out in the sequence of statements.

**Initialization:** From a table storing a large number of digits of \( \pi/2 \), create

\[ \text{Piby2}_j, \quad j = 1, 2, \ldots, M, \]

each with no more than \( L/2 \) bits.

**Integer Quotient:** Given \( |x| \leq 2^{L/2} \), let

\[ N := \text{round_to_integer}(x \cdot \text{two_by_Pi}). \]

Clearly, \( |N| \leq 2^{L/2} \).

**Exact subtraction:** Since \( |N| \leq 2^{L/2} \), Ada's model guarantees \( N \cdot \text{Piby2}_j \) to be exact for all \( j \)'s. This step starts by \( r := x \) and then keeps updating \( r \) by

\[ r := r - N \cdot \text{Piby2}_j \]

as long as \( \frac{1}{2} \leq |r/(N \cdot \text{Piby2}_j)| \leq 2, \quad j = 1, 2, \ldots. \)

**Refinement:** Use the value of \( r \) and the remaining \( N \cdot \text{Piby2}_j \) to create an accurate reduced argument. Standard techniques such as those in [6] are used here.
• KF.Sin and KF.Cos.

The two kernel functions KF.Sin and KF.Cos are simple odd and even polynomial approximations to \( \sin(r) \) and \( \cos(r) \), respectively, for \( |r| \leq \pi/4 \). Our work in [23] has tabulated various coefficients suitable for a range of required accuracies.

• KP.Atn.

The kernel procedure KP.Atn calculates arctan\((y/x)\) for given inputs \( x \) and \( y \), where \( 0 < y < x \). Whenever \( y/x \leq 1/16 \), a simple polynomial approximation is used. Otherwise, the following table-lookup approach is used.

**Reduction:** Let \( c_j \) be chosen such that

\[
\left| c_j - \frac{y}{x} \right| \leq 2^{-9}.
\]

One way to obtain \( c_j \) is

\[
j := \text{Integer}(256.0 * y/x),
\]
\[
c_j := \text{Float}(j)/256.0.
\]

Next, calculate \( r = (c_j x - y)/(x + c_j y) \).

The numerator is calculated accurately by performing \( c_j \ast x \) exactly. This is accomplished by using the primitive Leading.Part in the package GPF to split \( x \) into two pieces \( x_1 \) and \( x_2 \) so that \( c_j \ast x_1 \) and \( c_j \ast x_2 \) are both exact.

**Approximation:** Odd polynomials are used to approximate arctan\((r)\):

\[
p(r) \approx \arctan(r).
\]

**Reconstruction:** The value arctan\((y/x)\) is reconstructed by

\[
\arctan\left(\frac{y}{x}\right) = \arctan(c_j) + \arctan\left(\frac{y - c_j}{1 + \frac{y}{x}c_j}\right)
\]
\[
= \arctan(c_j) + \arctan(r)
\]
\[
\approx \arctan(c_j) + p(r).
\]

The values \( \arctan(c_j) \) are calculated beforehand and stored in a table that gives extra precision.

2.4 Error Analysis

How can one evaluate the accuracy of elementary function implementations? In the past, mathematical analyses were lacking even in important works such as [5]. In more recent work such as [2] and [8], the accuracy of the implementation is verified solely by extensive high-solution testing (about which we will discuss more in Section 3). Although good
reports from extensive testing give a strong vote of confidence for the implementations, no practical testing on, say, double-precision implementations (i.e., $\approx 10^{18}$ arguments) can be exhaustive. Why, then, has there been a lack of mathematical analysis to complement testing? We believe there are two main reasons. First, any such analysis would inevitably require specific assumptions about the underlying arithmetic. And in the past, computer arithmetic was so diversified that almost every different system required a different analysis. Second, rounding error analysis in the past had not been carried out tightly enough to yield worthwhile results; for example, a theoretical error bound of $4 \times 10^{-11}$ is not exciting news to an implementation that exhibits error no worse than 0.7 ulp under extensive testing. But the situation has now changed.

Since the adoption of IEEE Standard 754, the vast majority of floating-point engines have conformed to that standard. Hence an analysis based on the accuracy properties of IEEE floating point will be widely applicable. Furthermore, our recent work ([24], [26], and [25]) has shown that by analyzing independently the three aspects of an implementation that are largely uncorrelated with each other, rigorous and tight error bounds are obtained. In a typical case, an error bound of 0.57 ulp is obtained for an implementation whose largest error observed is 0.55 ulp. Finally, this analysis is applicable for implementation in arithmetic satisfying Ada’s model (instead of IEEE). Our work [22] on Ada is a straightforward translation of the one tailored to IEEE arithmetic [24].

We summarize the key features of our method here and refer the readers to our previously cited work for details.

Suppose that the function to be implemented is $f$ and the input argument is $x$. The three common steps of implementation are as follows:

**Reduction:**

$r = G(x), \quad \hat{r} \in I_P$,

where $G$ is the reduction process, and $I_P$ is a small interval.

**Approximation:**

$p(r) \approx f(r)$

where $p$ is a simple formula, usually a polynomial.

**Reconstruction:**

$f(x) = H(f(r)) \approx H(p(r))$

where $H$ is the reconstruction formula.

Because of inexact computations, we obtain $\hat{r}$ instead of $r$, $\hat{p}$ instead of $p$, and $\hat{H}$ instead of $H$. Hence, the computed result is

$\hat{H}(\hat{p}(\hat{r}))$
instead of

\[ H(f(r)). \]

The goal is to estimate the difference

\[ |H(f(r)) - \hat{H}(\hat{p}(\hat{r}))|. \]

We apply the triangular inequality. Thus,

\[
|H(f(r)) - \hat{H}(\hat{p}(\hat{r}))| \leq |H(f(r)) - H(f(\hat{r}))| + |H(f(\hat{r})) - H(p(\hat{r}))| \\
+ |H(p(\hat{r})) - \hat{H}(\hat{p}(\hat{r}))| \\
\leq E_1 + E_2 + E_3.
\]

\( E_1 \) is the error when reduction is the only error; \( E_2 \) is the error when approximation is the only error; and \( E_3 \) is the rounding errors in computing \( p \) and \( H \).

In most situations,

\[
E_1 \leq \text{constant} \cdot |f(r) - f(\hat{r})| \\
\approx \text{constant} \cdot |f'(r)| \cdot |r - \hat{r}|
\]

and

\[
E_2 \leq \text{constant} \cdot \max_{t \in I_P} |f(t) - p(t)|.
\]

\( E_1 \) can be easily estimated because the reduction process \( G \) is usually so simple that \( |r - \hat{r}| \) can be estimated tightly.

\( E_2 \) can also be easily estimated since the numerical value

\[
\max_{t \in I_P} |f(t) - p(t)|
\]

is obtained when the polynomial is sought, usually by the use of the Remez algorithm.

The rounding error

\[ E_3 = |H(p(\hat{r})) - \hat{H}(\hat{p}(\hat{r}))| \]

in calculating the polynomial and reconstruction is usually the most difficult to estimate tightly. With the use of table-lookup algorithms, however, the analysis is greatly simplified. The reason is that the magnitude of the reduced argument \( r \) (or \( \hat{r} \)) is typically so small that rounding errors associated with \( r^k \), \( k \geq 2 \), are practically zero. The simplicity of the analysis in [24] and [25] illustrates the situation. We conclude this section by tabulating in Table 1 the estimate for the various kernels.

3 An Accurate and Efficient Test Suite

The recent work on accurate elementary function implementation and the various implementation efforts induced by the proposed function package standard for Ada give rise to
Table 1: Error Estimates for Kernel Functions

<table>
<thead>
<tr>
<th>Kernel Function</th>
<th>Error Bounds in ulp</th>
</tr>
</thead>
<tbody>
<tr>
<td>KP.Exp</td>
<td>0.55</td>
</tr>
<tr>
<td>KF.Exp</td>
<td>0.58</td>
</tr>
<tr>
<td>KP.Log</td>
<td>0.57</td>
</tr>
<tr>
<td>KF.Llp</td>
<td>0.57</td>
</tr>
<tr>
<td>KF.Sin</td>
<td>0.98</td>
</tr>
<tr>
<td>KF.Cos</td>
<td>0.91</td>
</tr>
<tr>
<td>KP.Atn</td>
<td>0.57</td>
</tr>
</tbody>
</table>

A need for test programs of sufficiently high resolution that will separate the high-quality libraries from the low, and those that conform to the proposed specification from those that do not. In the past, test programs such as those in the ELEFUNT test package [5] have been low in resolution; and the only high-resolution tests available inevitably compute functions in extended precision. That method is impractically slow (cf. [16]) when extended precision is software simulated in a way that provides arbitrarily high precision (for example, in [3]). Our recent work [21] has shown that by applying table-driven techniques, making moderate assumptions on the floating arithmetic in use, and exploiting Ada’s language support for floating-point computation, one can implement efficient test programs for exp and log that exhibit a resolution of 0.005 ulp.

By extending the principles used in [21], we have implemented tests for $e^x$, log $x$, log$_2 x$, $x^y$, sin $x$, cos $x$, tan $x$, cot $x$, arcsin $x$, arccos $x$, and atan($y, x$). In what follows, we summarize our methodology, techniques, and implementation.

3.1 Testing Methodology

Let $F$ be the implementation of an elementary function $f$, say $e^x$. To test the accuracy of $F(T)$ efficiently and accurately, we compute independently $f(T)$ to a precision that is roughly 10 more bits than the working one. (Clearly, being able to evaluate $f(T)$ to arbitrarily high precision is not our objective.) Let $\epsilon$ denote one ulp of 1 in working precision, and let $\delta$ denote $2^{-10}$. Hence, for some chosen test argument $T$, our method is to obtain a value

$$f(T)(1 + O(\delta \epsilon))$$

so that we can compute the relative error

$$\frac{f(T) - F(T)}{f(T)} \approx \frac{f(T)(1 + O(\delta \epsilon)) - F(T)}{f(T)(1 + O(\delta \epsilon))}$$

accurately.
3.1.1 Extra-Precise Representation

To represent the value

\[ f(T)(1 + O(\delta e)) \]

in an environment that offers \(\epsilon\) in precision, we must use more than one working-precision variable. The standard representation we use for these values is a pair of working-precision values, say, \((X, x)\), such that \(x = O(\delta X)\) and \(X\) has no more than half the significant digits of the floating-point format, that is, the lower half of all the significant bits are zero. The use of this representation will become apparent in the subsequent discussion.

3.1.2 Backward Argument Reduction

Most elementary function implementations first reduce an input argument to some primary interval \(I_P\):

\[ T - \eta \in I_P. \]

Since \(\eta\) depends on \(T\), it cannot be calculated before \(T\) is given. The actual reduction process, because of computation error, usually turns out to be

\[ T - \hat{\eta} \]

instead. The error made in calculating \(\eta\) contributes to a major source of error in the implemented function. Although, in principle, \(\eta\) can be approximated as accurately as one wishes, the cost for an extremely accurate \(\eta\) in function implementation is too expensive, for such calculation must be repeated every time the function is invoked. But the situation in testing is different.

The set of \(T\)'s that corresponds to the same \(\eta\) is usually an interval. For example, \(\eta\) in \(\exp\) can be \(n\log 2\), and hence all \(T\)'s in

\[ n\log 2 + [-\log 2/2, \log 2/2] \]

correspond to the same \(\eta = n\log 2\). Thus, in testing we can choose to concentrate on test arguments with the same \(\eta\) before moving on to other locations of the domain. Consequently, we can afford to spend considerable time on calculating a very accurate approximation to \(\eta\). The typical scenario of backward argument reduction is as follows. Instead of generating test arguments randomly over the whole test domain, we first decide which set of \(T\)'s correspond to the same \(\eta\) we wish to test. Next, we calculate a very accurate approximation, say, \(P_1 + P_2 \approx \eta\). At this point, we are ready to generate test argument \(T\) by

\[ T := P_1 + P_2 + (\text{random } r \in I_P). \]

Hence, the accurate reduced argument is given by

\[ r + \text{correction}, \]
where
\[
\text{correction} = T - P_1 - P_2 - r.
\]

After sufficient testing is performed on this set of \(T\)'s, we choose a different subinterval.

### 3.1.3 Perturbation

From various perturbation formulae, the value \(f(T + \gamma)\) can be obtained to \(O(\delta \epsilon)\) accuracy provided \(f(T)\) is known to \(O(\delta \epsilon)\) accuracy in the first place. We must deal with two types of perturbations:

1. **\(O(\delta)\)-perturbations.** Here we typically have to evaluate \(f(T + \gamma)\) to \(O(\delta \epsilon)\) accuracy, where \(f(T + \gamma)(1 + O(\delta \epsilon))\) is known and \(|\gamma| = O(\delta T)\). To accomplish this, we usually exploit special properties of the underlying function, for example,

\[
e^{T + \gamma} = e^T e^\gamma = e^T + e^T (e^{\gamma} - 1) = e^T + \text{correction},
\]

or

\[
\sin(T + \gamma) = \sin(T) \cos(\gamma) + \cos(T) \sin(\gamma) = \sin(T) + (\sin(T)(\cos(\gamma) - 1) + \cos(T) \sin(\gamma)) = \sin(T) + \text{correction}.
\]

In both cases
\[
|\text{correction}| = O(\delta \cdot f(T)).
\]

Thus, provided \(f(T)\) is known to \(O(\delta \epsilon)\), and "correction" can be calculated to working precision (i.e., \(O(\epsilon)\) accuracy only), \(f(T + \gamma)\) can be calculated to \(O(\delta \epsilon)\) accuracy.

2. **\(O(\epsilon)\)-perturbation.** Here we have to evaluate \(f(T + \gamma)\) to \(O(\delta \epsilon)\) accuracy, where \(f(T)(1 + O(\delta \epsilon))\) is known and \(|\gamma| = O(\delta T)\). Because \(\gamma\) is so small, it suffices to use

\[
f(T + \gamma) \approx f(T) + f'(T) \cdot \gamma,
\]

provided \(f'(T)\) can be easily calculated and is moderate in magnitude.

### 3.1.4 Breakpoints in Primary Intervals

To apply the perturbation methods, we must have some accurate values of \(f(T)\) to begin with. For a number of functions, we begin with a basic set of points \(\{b_j\}\), called breakpoints, in their primary intervals \(I_j\)'s. Before testing begins, the values \(f(b_j)\) must be known to \(O(\delta \epsilon)\) accuracy. Two methods are used to obtain this set of values.
1. **Tabulation.** In situations where the number of breakpoints are few, say $\leq 32$, we simply calculate the values $f(b_j)$ beforehand to high accuracy (by say, "bc" on Unix systems) and tabulate the values as literals (in leading and trailing pieces) in the source program.

2. **Tabulation and Calculations.** In some situations, a large number of breakpoints are required ($\approx 1000$, say). Tabulating 1000 literals in source programs make them bulky and error prone. We solve this problem by choosing our breakpoints in such a way that it suffices to tabulate the values for a small subset of all the breakpoints. The remaining values required can be computed by the perturbation techniques. An example illustrates the ideas here. The breakpoints used in testing the trigonometric functions are

$$b_j = j/1024, \quad j = 0, 1, \ldots, 863,$$

and we need accurate values for $\sin(b_j)$ and $\cos(b_j)$. First, the values $\sin(c_k)$ and $\cos(c_k)$

$$c_k = k/32, \quad k = 0, 1, \ldots, 26,$$

are tabulated to $O(\delta \epsilon)$ accuracy. Then the values $\sin(d_l)$ and $\cos(d_l)$ where

$$d_l = l/1024, \quad l = 0, 1, \ldots, 31,$$

are calculated to $O(\delta \epsilon)$ accuracy. The calculations are easy because $d_l$ and $1-d_l^2/2$, $l = 0, 1, \ldots, 31$, are representable exactly in 21 bits. Hence as long as

$$\text{Float\_Type\_base\_digits} \geq 6,$$

these terms can be computed without error. Now

$$\sin(d_l) = d_l + O(d_l^3),$$
$$\cos(d_l) = 1 - d_l^2/2 + O(d_l^4).$$

The $O(d_l^3)$ and $O(d_l^4)$ terms above can be calculated to full machine precision, that is, $O(\epsilon)$ precision, by applying Horner recurrence to the truncated Taylor series of $\sin x - x$ and $\cos x - 1 + x^2/2$, respectively. Finally, each $b_j$, $j = 0, 1, \ldots, 863$, can be expressed as $b_j = c_k + d_l$. Careful calculations based on

$$\sin(b_j) = \sin(c_k + d_l) = \sin(c_k) \cos(d_l) + \cos(c_k) \sin(d_l)$$
$$\cos(b_j) = \cos(c_k + d_l) = \cos(c_k) \cos(d_l) - \sin(c_k) \sin(d_l),$$

give values of $\sin(b_j)$ and $\cos(b_j)$ to $O(\delta \epsilon)$ accuracy. Clearly, these calculations are performed in the sequence of statements in the test package body.
3.1.5 Testing Inverses

Knowing \( f \) at an argument to \( T \) to \( O(\delta \epsilon) \) accuracy sometimes allows us to calculate \( f^{-1} \) at some other argument \( S \) by applying an \( O(\epsilon) \)-type of perturbation. Suppose we have \((X, x)\) where

\[
X + x = f(T)(1 + O(\delta \epsilon)) = f(T) + \eta.
\]

Let \( g \) denote \( f^{-1} \). Then

\[
g(X + x) = g(f(T) + \eta) = g(f(T)) + O(\eta \cdot g'(f(T))) = T + O(\eta/f'(T)).
\]

Provided \( f'(T) \) is such that

\[
O(\eta/f'(T)) = O(\delta \epsilon T),
\]

we have \( g(X + x) \) to \( O(\delta \epsilon) \) accuracy.

An obvious test argument \( S \) for \( g \) (i.e., \( f^{-1} \)) is the rounded sum \( X + x \) thus: \( S := X + x \). Clearly, \( S \) is an \( O(\epsilon) \) perturbation of \( X + x \) at which \( g \) is known to \( O(\delta \epsilon) \) accuracy. This method is applicable to testing \( \arctan, \arcsin, \) and \( \arccos \) by knowing accurate values of \( \sin \) and \( \cos \).

3.1.6 Testing Ratio of Functions

Knowing both \( f \) and \( g \) at an argument \( T \) to \( O(\delta \epsilon) \) accuracy allows us to test the accuracy of an implementation of \( f/g \). Suppose

\[
X + x = f(T)(1 + O(\delta \epsilon)) \quad \quad Y + y = g(T)(1 + O(\delta \epsilon)),
\]

and let \( S \) be the value obtained by invoking an implementation of \( f/g \). The observation is that, although

\[
\frac{X + x}{Y + y} = \frac{f(T)(1 + O(\delta \epsilon))}{g(T)(1 + O(\delta \epsilon))},
\]

there is no need to evaluate the extra-precise quotient. The reason is that we are interested only in the relative error

\[
\left( \frac{X + x}{Y + y} - S \right) \cdot \frac{Y + y}{X + x} = \frac{(X + x) - S(Y + y)}{X + x}.
\]

Clearly, the crucial calculation is the subtraction. By representing \( S \) as \( S = Z + z \), where \( Z \) has at least half of the least significant bits zeros, we can evaluate the numerator by

\[
(X - Z \ast Y) + (x - (Z \ast y + z \ast Y + z \ast y)).
\]
The crucial observation is that
\[ X - Z \cdot Y \]
is exact. This method is applicable to testing tan and cot from sin and cos, and also to testing log₆ from log.

3.2 Structure of the Test Suite

In a manner similar to that used for the elementary function package, we have implemented our test package based on various test procedures we identified as kernels. We now describe them briefly and point out how the rest of the tests are constructed from the kernels.

3.2.1 Kernel Tests

As explained earlier, the key to our high-resolution test is obtaining function values to \( O(\delta \epsilon) \) accuracy thus:
\[ W + w = f(T)(1 + O(\delta \epsilon)). \]
Hence, the kernels we implemented are really procedures that produced a list of test arguments \( T_j \)'s chosen randomly from some prescribed interval in the whole test domain and the corresponding \( O(\delta \epsilon) \) accurate function values \((W_j, w_j)'s\).

- Testing exp.
  Given \( n \), this kernel generates a set of test arguments
  \[ T \in n \log 2 + [- \log 2/2, \log 2/2], \]
  and the corresponding \( O(\delta \epsilon) \) accurate values for \( e^T \).

- Testing log.
  Given \( n \), this kernel generates a set of test arguments
  \[ T \in 2^n \cdot [1/\sqrt{2}, \sqrt{2}] \]
  and the corresponding \( O(\delta \epsilon) \) accurate values for \( \log(2^{-n} \cdot T) \). Clearly, an accurate \( \log(T) \) can be easily constructed by performing
  \[ n \log 2 + \log(2^{-n}T) \]
carefully.

- Testing sin and cos.
  This kernel generates a set of test arguments
  \[ T \in [0, \pi/4] \]
  and the corresponding \( O(\delta \epsilon) \) accurate values for \( \sin(T) \) and \( \cos(T) \).
3.2.2 Application of the Kernels

Using the techniques of backward argument reduction, we can apply the three kernels to testing Exp, Log, Sin, and Cos. By using the other techniques described previously, we can do considerably more.

- Testing $X^{**}Y$.

  Using the exp kernel, we have $Z$ and $(U, u)$ such that
  \[ U + u = e^Z (1 + O(\delta \varepsilon)). \]

  Now, by tabulation (or by modification to the log kernel), we have $X$ and $(V, v)$ such that
  \[ V + v = \log(X) (1 + O(\delta^2 \varepsilon)). \]

  Since $X^Y = e^{Y \log X}$, we would naturally create our test argument $Y$ so that $Y \log(X) \approx Z$. Therefore, we set
  \[ Y := Z/(V + v). \]

  Now with $X$ and $Y$ as the test arguments for the exponentiation operator $"^{**}"$, we must obtain to $O(\delta \varepsilon)$ accuracy the value $e^{Y \log X}$. Since
  \[ V + v = \log(X) (1 + O(\delta^2 \varepsilon)), \]

  it suffices to obtain $e^{Y(V + v)}$ to $O(\delta \varepsilon)$ accuracy. But $Y(V + v)$ is an $O(\varepsilon)$ perturbation of $Z$, and $e^Z$ is already known to $O(\delta \varepsilon)$ accuracy. Consequently, by carefully calculating the amount of perturbation
  \[ Z - Y(V + v), \]

  an $O(\delta \varepsilon)$ accurate value of $X^Y$ can be obtained.

- Testing $\log(X, \text{Base})$, Tan($X$), and Cot($X$).

  Simple applications of the kernels for log, sin, and cos together with the method of testing ratios of functions allow us to test these functions.

- Testing Arctan($Y, X$).

  The test kernels for sin and cos and the method of testing inverses allow us to test this function.

  The test kernel provides $T$, $(S, s)$, and $(C, c)$ where $T \in [0, \pi/4]$ and
  \[
  S + s = \sin(T)(1 + O(\delta \varepsilon)) \\
  C + c = \cos(T)(1 + O(\delta \varepsilon)).
  \]
Clearly,
\[ \tan(T) = \frac{S + s}{C + c} (1 + O(\delta \epsilon)), \]
and
\[ \tan^{-1} \left( \frac{S + s}{C + c} \right) = T (1 + O(\delta \epsilon)), \]
as the slope of \( \tan^{-1} \) is less than 1 in magnitude. Thus, a natural pair of test arguments would be \( X := C + c \) and \( Y := S + s \), the rounded sum of \( C + c \) and \( S + s \), respectively. \( Y/X \) is clearly an \( O(\epsilon) \)-perturbation of \((S + s)/(C + c)\). Moreover, the derivative of \( \tan^{-1}(t) \) is given by a simple formula \( 1/(1 + t^2) \). Thus

\[
\begin{align*}
\arctan(Y, X) &= \tan^{-1}(Y/X) \\
&= \tan^{-1} \left( \frac{S + s}{C + c} + \eta \right) \\
&= \left( T + \frac{\eta}{1 + T^2} \right) (1 + O(\delta \epsilon)).
\end{align*}
\]

The perturbation
\[ \eta = \frac{Y}{X} - \frac{S + s}{C + c} \]
can be obtained by careful calculation. This tests \( \arctan(Y, X) \) for \( 0 < Y \leq X \). The whole domain can be tested by applying the following:

\[
\begin{align*}
\arctan(X, Y) &= \frac{\pi}{2} - \tan^{-1}(Y/X) \\
\arctan(X, -Y) &= \frac{\pi}{2} + \tan^{-1}(Y/X) \\
\arctan(Y, -X) &= \pi - \tan^{-1}(Y/X)
\end{align*}
\]

- Testing \( \arcsin \) and \( \arccos \).

The crux is to be able to test the two functions \( \arcsin \) and \( \arccos \) in the domain \([0, \pi/4]\). Testing \( \arcsin \) in that domain is a simple application of the technique of testing inverses. The infinite slope of \( \arccos \) at zero causes that method to break down—but only in a small neighborhood of zero. Testing \( \arccos \) near zero can be implemented directly, that is, without using tests for \( \cos \).

4 Test Results

In this section, we report our experience with our function package and test suite.
Table 2: Successful Compilations of Tang's Function Package

<table>
<thead>
<tr>
<th>Machine</th>
<th>Compiler</th>
<th>Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequent Symmetry</td>
<td>Verdix</td>
<td>Argonne National Lab</td>
</tr>
<tr>
<td>Encore Multimax</td>
<td>Verdix</td>
<td>Argonne National Lab</td>
</tr>
<tr>
<td>Sun 3</td>
<td>Alsys</td>
<td>Argonne National Lab</td>
</tr>
<tr>
<td>SPARCstation</td>
<td>Verdix</td>
<td>Boeing, Seattle</td>
</tr>
<tr>
<td>DEC MicroVAX</td>
<td>DEC Ada</td>
<td>Boeing, Seattle and NAG, Oxford</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>IBM Ada</td>
<td>IBM, Toronto</td>
</tr>
<tr>
<td>IBM 370</td>
<td>Telesoft</td>
<td>IBM, Toronto</td>
</tr>
</tbody>
</table>

4.1 Portability of the Function Package

We have compiled our function package on a number of hardware/software environments. The machines we have used include Sun 3, Sun Sparcstation, IBM RS/6000 workstation, IBM 370, IBM PC/AT, Sequent Symmetry, DEC MicroVAX, and Encore Multimax. Compilers we have used include Verdix, DEC Ada, IBM Ada, Alsys, and Meridian Adavantage. All compilations were performed on our source code without any changes. Only one failure was experienced. That occurred on the IBM PC/AT with the Meridian Adavantage compiler whose 64K-limit on object sizes caused the failure. Table 2 tabulates the hardware/software environments in which compilation was successful.

4.2 Calibration of the Test Package

To confirm the accuracy of our test package, we apply it to a function package in a precision lower than the highest one available. The function package under test is implemented typically as

\[ \text{sgl}(F(\text{dble}(T))) \]

Hence, for all practical purposes, the functions under test are perfect. Consequently, the test package, when run on sufficiently many arguments over the entire domain, should report a maximum error of 0.5ulp + noise. The smaller the noise, the more accurate the test package is. Note that ulp is measured with respect to the actual hardware precision, not with respect to the attribute \( \text{Float_Type'Base'Epsilon} \).

Table 3 tabulates the environments used for the calibrations, and Table 4 tabulates the calibration results. Note that \( \text{Arctan}(Y, X) \) and \( \text{Log}(X, \text{Base}) \) are unavailable in Environments 1 and 2.

In addition to calibrating the test package, we also cross-examine the test results: When reporting the maximum error observed, our test package also gives the exact test
Table 3: Environments Used for Calibrating Tang's Test Package

<table>
<thead>
<tr>
<th>Environment</th>
<th>Machine</th>
<th>Precision</th>
<th>Coprocessor</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequent Symmetry</td>
<td>IEEE Single</td>
<td>Intel 80387</td>
<td>Verdix</td>
</tr>
<tr>
<td>2</td>
<td>Encore Multimax</td>
<td>IEEE Single</td>
<td>NS 38021</td>
<td>Verdix</td>
</tr>
<tr>
<td>3</td>
<td>Sun 3</td>
<td>IEEE Single</td>
<td>MC 68881</td>
<td>Alys</td>
</tr>
</tbody>
</table>

argument at which such error occurred together with the exact function valued produced by the implementation under test. Thus, we can verify the accuracy of the reported error by an independent method (we used "bc" in Unix). The results of the test package have always been correct to 3 significant digits.

4.3 Testing Various Function Packages

We now report on the results by applying our test package on our function package as well as a number of other function packages. Most of the other function packages tested are supplied by compiler vendors. At this point, only the package supplied by Alys claims to conform to the proposed function standard. Other compiler vendors typically supply function packages with their own specifications. Consequently, we wrote several simple interfaces to construct a partial implementation of the package GEF. We will discuss these implementations when we present the corresponding test results. Throughout the test reports, ulp is measured with respect to the underlying machine precision, e.g., \( 1 \text{ulp} = 2^{-23} \) for IEEE single precision and \( 1 \text{ulp} = 2^{-112} \) for VAX H-format. Because of our limited access to some of the systems discussed in Section 4.1, we were able to perform tests on only a subset of those environments. For simplicity's sake, we have concentrated on testing the various function packages in the highest precision available in the particular environments.

4.3.1 Testing Tang's Function Package

Table 5 tabulates the various test environments and Table 6 tabulates the test results.

Note that the Alys compiler offers two alternatives: one exploiting the floating-point instructions provided by the math coprocessor MC68881, and the other performing all floating-point operations by software simulation. From the viewpoint of accuracy, the main difference between the two options is that using the floating-point support from MC68881 yields a more accurate library because basic arithmetic operations are carried out to 64 bits of precision; software simulation only provides 53 bits of accuracy. This difference accounts for the slight variation in the reported errors. The large error reported about \( X^{\ast\ast}Y \) in VAX H-format does not warrant alarm. Error analysis shows that in order for the exponentiation operator to be fully accurate, \( \log(X) \) must be calculated to at least working precision plus the number of bits in the exponent field of working precision.
Table 4: Calibration Results

<table>
<thead>
<tr>
<th>Function</th>
<th>Maximum Error in Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Exp(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>X**Y</td>
<td>0.500</td>
</tr>
<tr>
<td>Log(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Log(X,2.0)</td>
<td>0.500</td>
</tr>
<tr>
<td>Log(X,10.0)</td>
<td>0.500</td>
</tr>
<tr>
<td>Log(X,Base)</td>
<td>N/A</td>
</tr>
<tr>
<td>Sin(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Cos(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Tan(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Cot(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Arcsin(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Arccos(X)</td>
<td>0.500</td>
</tr>
<tr>
<td>Arctan(Y,1.0)</td>
<td>0.500</td>
</tr>
<tr>
<td>Arctan(Y,X)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5: Environments Used for Testing Tang's Function Package

<table>
<thead>
<tr>
<th>Environment</th>
<th>Machine</th>
<th>Precision</th>
<th>Coprocessor</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequent Symmetry</td>
<td>IEEE Double</td>
<td>Intel 80387</td>
<td>Verdix S5.5.2</td>
</tr>
<tr>
<td>2</td>
<td>Encore Multimax</td>
<td>IEEE Double</td>
<td>NS 38021</td>
<td>Verdix 5.5</td>
</tr>
<tr>
<td>3</td>
<td>DEC MicroVAX</td>
<td>VAX H-format</td>
<td>N/A</td>
<td>DEC Ada 2.1</td>
</tr>
<tr>
<td>4</td>
<td>Sun 3</td>
<td>IEEE Double</td>
<td>MC 68881</td>
<td>Alsys 5.2</td>
</tr>
<tr>
<td>5</td>
<td>Sun 3</td>
<td>IEEE Double</td>
<td>Software</td>
<td>Alsys 5.2</td>
</tr>
</tbody>
</table>
Table 6: Test Results on Tang's Function Package

<table>
<thead>
<tr>
<th>Function</th>
<th>No. of Test Arguments ($\times 10^6$)</th>
<th>Maximum Error in Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Exp(X)</td>
<td>1.5</td>
<td>0.51</td>
</tr>
<tr>
<td>X**Y</td>
<td>1.5</td>
<td>0.51</td>
</tr>
<tr>
<td>Log(X)</td>
<td>2.0</td>
<td>0.54</td>
</tr>
<tr>
<td>Log(X, 2.0)</td>
<td>2.0</td>
<td>0.84</td>
</tr>
<tr>
<td>Log(X, 10.0)</td>
<td>2.0</td>
<td>0.91</td>
</tr>
<tr>
<td>Log(X, Base)</td>
<td>2.0</td>
<td>1.25</td>
</tr>
<tr>
<td>Sin(X)</td>
<td>2.0</td>
<td>0.52</td>
</tr>
<tr>
<td>Cos(X)</td>
<td>2.0</td>
<td>0.52</td>
</tr>
<tr>
<td>Tan(X)</td>
<td>2.0</td>
<td>1.31</td>
</tr>
<tr>
<td>Cot(X)</td>
<td>2.0</td>
<td>1.31</td>
</tr>
<tr>
<td>Arctan(x)</td>
<td>1.0</td>
<td>1.16</td>
</tr>
<tr>
<td>Arccos(x)</td>
<td>1.0</td>
<td>1.15</td>
</tr>
<tr>
<td>Arctan(Y, 1.0)</td>
<td>1.0</td>
<td>0.51</td>
</tr>
<tr>
<td>Arctan(Y, X)</td>
<td>1.0</td>
<td>0.96</td>
</tr>
</tbody>
</table>

Since providing that kind of extra accuracy may be impractically expensive, the SIGAda Numerics Working Group adopted a compromise that, in effect, allows as many as $L - 5$ bits in the final answer to be lost, where $L$ is the number of bits in the exponent field. The implication is that log(X) now needs only 5 extra bits of accuracy. For VAX H-format, the proposed standard allows roughly 10 bits to be lost; and the error exhibited here is well below that threshold. In fact, our exponentiation operator is the most accurate among all those tested. We present test results in the remainder of this section; details of the algorithm used for X**Y will be presented in a different article.

4.3.2 Testing Vendors' Function Packages

We report the test results obtained by running our test package on the various function packages provided by Verdix, DEC, and Alsys. As mentioned earlier, Alsys is the only vendor that provides a function package that supplies all the functionalities required by the proposed standard. Neither Verdix's nor DEC's math libraries provide an arbitrary-base logarithm; but they both provide the special cases log2 and log10. Verdix's library provides the exponentiation operator but not the 2-argument arctan; DEC's library provides the latter but not the former. Table 7 tabulates the specific environments in which the vendors' libraries were tested. Table 8 tabulates the test results.
Table 7: Environments Used in Testing Vendors' Function Packages

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Machine</th>
<th>Floating Point</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verdix S5.5.2</td>
<td>Sequent Symmetry</td>
<td>Intel 30387</td>
<td>IEEE Double</td>
</tr>
<tr>
<td>Verdix 5.5</td>
<td>Encore Multinax</td>
<td>NS 38021</td>
<td>IEEE Double</td>
</tr>
<tr>
<td>DEC Ada 2.1</td>
<td>DEC MicroVAX</td>
<td>VAX</td>
<td>VAX H-format</td>
</tr>
<tr>
<td>Alsys 5.2</td>
<td>Sun 3</td>
<td>MC 68881</td>
<td>IEEE Double</td>
</tr>
<tr>
<td>Alsys 5.2</td>
<td>Sun 3</td>
<td>Software</td>
<td>IEEE Double</td>
</tr>
</tbody>
</table>

Table 8: Test Results on Vendors' Function Packages

<table>
<thead>
<tr>
<th>Function</th>
<th>No. of Test Arguments ($\times 10^6$)</th>
<th>Verdict Maximum Error</th>
<th>Alsys Maximum Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exp(X)</td>
<td>1.5</td>
<td>0.72</td>
<td>1.22</td>
</tr>
<tr>
<td>X**Y</td>
<td>1.5</td>
<td>0.82</td>
<td>$3.33 \times 10^4$</td>
</tr>
<tr>
<td>Log(X)</td>
<td>2.0</td>
<td>0.50</td>
<td>0.67</td>
</tr>
<tr>
<td>Log(X, 2.0)</td>
<td>2.0</td>
<td>1.00</td>
<td>1.14</td>
</tr>
<tr>
<td>Log(X, 10.0)</td>
<td>2.0</td>
<td>0.50</td>
<td>1.09</td>
</tr>
<tr>
<td>Log(X, Base)</td>
<td>2.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sin(X)</td>
<td>2.0</td>
<td>$1.69 \times 10^9$</td>
<td>2.26 x 10^9</td>
</tr>
<tr>
<td>Cos(X)</td>
<td>2.0</td>
<td>$1.69 \times 10^9$</td>
<td>2.12 x 10^9</td>
</tr>
<tr>
<td>Tan(X)</td>
<td>2.0</td>
<td>$5.12 \times 10^{13}$</td>
<td>2.26 x 10^9</td>
</tr>
<tr>
<td>Cot(X)</td>
<td>2.0</td>
<td>$5.12 \times 10^{13}$</td>
<td>2.03 x 10^9</td>
</tr>
<tr>
<td>Arcsin(X)</td>
<td>1.0</td>
<td>0.73</td>
<td>2.27</td>
</tr>
<tr>
<td>Arccos(X)</td>
<td>1.0</td>
<td>0.95</td>
<td>1.22</td>
</tr>
<tr>
<td>Arctan(Y, 1.0)</td>
<td>1.0</td>
<td>0.50</td>
<td>2.53</td>
</tr>
<tr>
<td>Arctan(Y, X)</td>
<td>1.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Verdix's Libraries

The library for the Sequent Symmetry is accurate except for the forward trigonometric functions. The accuracy is largely attributable to the 64-bit accurate functions provided by the coprocessor 30387. Since that chip provides 64-bit accurate $\log_2 x$ and $2^x$, the exponentiation operator implemented by

$$x^y = 2^{y \log_2 x}$$

is accurate to 53 bits, that is, to full IEEE double precision. The 1.0-ulp accuracy of $\text{Log}(X,2.0)$ is strange because if the 30387 $\log_2$ function is used, the result should be 0.5. This mystery cannot be answered without knowing the construction of the function library.

The result of the trigonometric functions on the Sequent is interesting. The error of $10^9$ clearly suggests that the underlying period is a 66-bit approximation to $\pi$, which is how the 30387 trigonometric functions are implemented. So the $10^9$ error is expected; but not so with the $10^{13}$ error exhibited by the functions $\text{Tan}(X)$ and $\text{Cot}(X)$. That error suggests that a 53-bit approximation to $\pi$ was used as the underlying period. Indeed, not only did we confirm that $\text{Tan}(X)$ and $\text{Cot}(X)$ are using a 53-bit approximation to $\pi$ (and hence not exploiting the tan x function already provided by the 30387 chip), but we also discovered a bug. When $\text{Tan}(X)$ was invoked at

$$X := 16\#1.921FB54442D18\#; \quad \pi \text{ rounded to 53 bits},$$

instead of getting 0.0 as the answer, an irrecoverable exception was generated that aborted the program. A bug report was subsequently filed with the vendor.

Verdix's library on the Encore does not get the extra-precision support since the NS38021 coprocessor supports only 53 bits. The exponentiation operator is clearly unacceptable. The uniform error exhibited by the trigonometric functions indicates that a uniform period has been used.

DEC's Library

DEC's library is clearly exemplary and lives up to the reputation of DEC's Fortran runtime library. That the exponentiation operator $X**Y$ is not provided is unfortunate.

Alsys's Libraries

Except for $X**Y$, the Alsys library using MC68881 is similar to the Verdix library using Intel 30387. The Alsys library that does not use MC68881 also exhibits large errors in $X**Y$.

To conclude this section, we offer in Table 9 a brief comparison of the typical accuracy properties of the various software-implemented functions, that is, those not supplied directly by coprocessors.
Table 9: Brief Comparison of Functions Packages

<table>
<thead>
<tr>
<th>Function</th>
<th>Maximum Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tan</td>
</tr>
<tr>
<td>Exp(X)</td>
<td>0.51</td>
</tr>
<tr>
<td>X**Y</td>
<td>0.51</td>
</tr>
<tr>
<td>Log(X)</td>
<td>0.56</td>
</tr>
<tr>
<td>Log(X, 2.0)</td>
<td>0.85</td>
</tr>
<tr>
<td>Log(X, 10.0)</td>
<td>0.93</td>
</tr>
<tr>
<td>Log(X, Base)</td>
<td>1.25</td>
</tr>
<tr>
<td>Sin(X)</td>
<td>0.98</td>
</tr>
<tr>
<td>Cos(X)</td>
<td>0.98</td>
</tr>
<tr>
<td>Tan(X)</td>
<td>1.50</td>
</tr>
<tr>
<td>Cot(X)</td>
<td>1.50</td>
</tr>
<tr>
<td>Arcsin(X)</td>
<td>1.20</td>
</tr>
<tr>
<td>Arccos(X)</td>
<td>1.20</td>
</tr>
<tr>
<td>Arctan(Y, 1.0)</td>
<td>0.51</td>
</tr>
<tr>
<td>Arctan(Y, X)</td>
<td>0.96</td>
</tr>
</tbody>
</table>

5 Conclusion

We have shown that with moderate assumptions about the underlying arithmetic, accurate elementary functions can be implemented portably. We have shown that such implementations can be analyzed rigorously. Moreover, high-resolution test methods can be implemented portably as well. We have also illustrated the usefulness of some of Ada's features that support floating-point computations. Since many of those features, such as base-16 literals and their accurate conversions, can be implemented on languages other than Ada, our work would encourage such considerations. Finally, our test package and the results presented here are likely to encourage vendors to improve the qualities of some of their functions.

Acknowledgments

A number of people have used the function and test packages in their institutions: Alan Adamson at IBM, Toronto; Michael Carter at Boeing Computing Services, Seattle; Graham Hodgson at NAG, Oxford; and Jon Squire at Westinghouse, Baltimore. Ken Dritz's expertise in Ada has been invaluable to the author. Special thanks are due to Gil Myers, whose leadership in the SIGAda Numerics Working Group has motivated most of this work.
References


