NASA Technical Memorandum 102256

Basic Mathematical Function Libraries for Scientific Computation

David C. Galant

December 1989

(NASA-TM-102256) BASIC MATHEMATICAL FUNCTION LIBRARIES FOR SCIENTIFIC COMPUTATION (NASA) 28 p CSCL 09B

N90-16423

Unclas
G3/61 0260641

National Aeronautics and
Space Administration

NASA Technical Memorandum 102256

Basic Mathematical Function Libraries for Scientific Computation

David C. Galant, Ames Research Center, Moffett Field, California

December 1989

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

TABLE OF CONTENTS

SUMMARY

Ada[1] packages implementing selected mathematical functions for the support of scientific and engineering applications have been written. The packages provide the Ada programmer with the mathematical function support found in the languages Pascal and Fortran as well as an extended precision arithmetic and a complete complex arithmetic. The algorithms used are fully described and analyzed. Implementation assumes that the Ada type FLOAT objects fully conform to the IEEE 754-1985 standard for single binary floating-point arithmetic, and that INTEGER objects are 32-bit entities. Codes for the Ada packages are included as appendixes.

INTRODUCTION

The packages described in this report were developed as infrastructure for benchmark programs written to test a new computer. They provide selected mathematical functions, elementary transcendental functions, complex arithmetic, and a limited facility for extended precision arithmetic. Other packages for extended precision arithmetic were written to support these functions. The implementation of the functions, coded in Ada [1], assume IEEE 754-1985 [2] single binary floating point arithmetic and 32-bit integers are supported at the hardware level. Project design goals were (1) a portable, robust self contained library in Ada, (2) support for scientific/engineering computation, and (3) accurate, efficiently produced results.

Every means was used to ensure the accuracy and correctness of the functions. Additionally, the algorithms were coded and tested for actual performance. The packages coded are given as appendixes. Individual packages implementing the selected functions are provided as appendixes A through D of this report. The remainder of the report discusses, in detail, the implementation of these functions.

FLOATING-POINT ARITHMETIC

The IEEE 754-1985 standard is a specification of arithmetic formats, operations, and details concerned with the accuracy of results and the treatment of exceptional conditions. For this paper, the relevant specification is single binary floating-point format, which is a 1-bit sign field, an 8-bit biased exponent field e, $0 \le e \le 255$, and a 23-bit fraction field, $f = 0.b_1b_2..b_{23}$. The 32-bit single binary float formatted number X is stored as

1	8	23
s	e	f

[1]Ada is a registered trademark of the U. S. Department of Defense (Ada Joint Program Office).

Its value is decoded from its stored representation by means of the following rules:

e	f	Interpretation
255	0	$(-1)^s \infty$
255	$\neq 0$	NaN, regardless of s
0	0	$(-1)^s$ 0 (zero)
0	$\neq 0$	$(-1)^s 2^{-126}$ 0.f (denormalized number)
>0, <255	—	$(-1)^s 2^{e-127}$ 1.f

For numerical algorithms, the major consequences of the format are

- accuracy is 24 bits, at most
- the smallest representable number ε, such that $(1 + \varepsilon) - 1 > 0$, in floating-point arithmetic is $\varepsilon = 2^{-23} = 1.192\cdots \times 10^{-7}$ (therefore, approximations need barely more than seven digits of relative accuracy).
- the range of normalized floating-point numbers is 2^{-126} ($\approx 1.175494211\ 10^{-38}$) to $2^{127}(2-2^{-22})$ ($\approx 1.701411632\ 10^{38}$).

Some Basic Procedures

While the elementary transcendental functions are all well behaved, their Taylor series are unsatisfactory for numerical calculation. To bound the time necessary for a given accuracy, the range over which an approximation is based must be limited. Range reduction requires either separating the fields of a floating-point number or arithmetic manipulation. In Ada, field separation is most quickly done by instantiating the generic function **unchecked_conversion** available in the Ada package **system** [1]. Converting a FLOAT number to INTEGER without changing the bit pattern, and separation is easily done in integer arithmetic. Conversely, from s, e, and f, an INTEGER with the appropriate bit pattern can be constructed and then converted to a FLOAT number by instantiation of the same generic function.

These fundamental procedures are given as package IEEE754Lib in appendix A. Any floating arithmetic requires similar routines. Machine language code would be more efficient, but portability would be sacrificed.

EXTENDED ARITHMETIC PACKAGE

Digital computer floating-point arithmetic is grainy with gaps between numbers. For example, in IEEE 754-1985 arithmetic, the first number after 2^{23} is $2^{23}+2$. Consequently, results of arithmetic operations may not be exactly expressible. Error also occurs from the truncation of constants. For example, the constant π is expressible only by an infinite sequence of bits, but with IEEE 754-1985 arithmetic, the nearest value is about 3.1415927. A final source of error arises when two nearly equal numbers are subtracted. Fewer bits remain in the difference. Sometimes this is called catastrophic cancellation, but the term is misleading, because the result is exact and is a natural consequence of finite representation.

Examples of how range reduction can lead to the last kind of error are the 2π periodic functions sine and cosine. An argument is reduced to an equivalent value in $[-\pi/4, \pi/4)$. For a large argument, many bits encode which interval of length $\pi/2$ contains it. For such arguments, accuracy can be maintained only with extended values for constants and extra precise arithmetic for this part of the calculation.

Linnainmaa [3] describes a portable, doubled precision arithmetic based on the native floating-point arithmetic. The algorithms depend on the assumption that native floating-point arithmetic is either correctly rounded or correctly chopped. An extended precision number is represented as a pair of single precision numbers `(x_high, x_low)` which, in the **native** floating-point arithmetic, exactly satisfy

```
x_high = x_high + x_low
```

floating-point numbers are separated into pieces, each of which has sufficiently few bits that the result of an arithmetic operation with these numbers is always exactly represented. The separation process for numbers reduces the dynamic range of numbers somewhat, but this is not a problem in the library packages.

Error in extended arithmetic is complicated by truncation, modes of rounding, and extra bits in intermediate results. However, for IEEE 754-1985 single binary float arithmetic with correct rounding, extended multiply is accurate to 47 bits, extended divide to 46.5 bits, and extended add to 48 bits. This is more than sufficient for the library packages. An Ada implementation of this extended arithmetic is appendix B. Quite clearly, this extended arithmetic is applicable to far more than just the libraries discussed here. Furthermore, the same ideas can be used to extend any higher precision arithmetic available as well, including the extended precision itself. However, this extension to multiple precision eventually fails because full double precision is unattainable.

COMPLEX ARITHMETIC PACKAGE

This package carefully implements Cartesian complex arithmetic. A type **complex** (a record of two numbers of type FLOAT), is declared with functions for manipulating these objects. The arithmetic operators are overloaded. Facilities for creating and mixing these new numbers with numbers of type FLOAT are included. Also included are a complex absolute value function and a complex square root function. Special care has been taken to ensure accuracy and absence of false errors. The package, named ComplexArithmetic, is given as appendix C.

THE MATHLIB PACKAGE

The Basic Mathematical Functions Included in the package are

Name	Argument List	Value Type	Function
floor	x : float	integer	nearest integer <= x
ceil	x : float	integer	nearest integer >= x
trunc	x : float	integer	round x toward 0 to integer
xmod	x, y : float	float	(x - trunc(x / y) * y)
urand	—	float	random numbers in (0,1)
urandinit	s1, s2 : integer	---	initialize urand

The first four are simple conversion and manipulation functions. The last two provide a reinitializable 32-bit, portable, uniform random number generator with an extremely long period (about 10^{18}) and very good statistical properties (see [4]). The random number generator depends heavily upon 32-bit integers for efficient implementation.

Transcendental functions Included in the package are

Name	Function	Range
sin(x)	trigonometric sine of x (radians)	$\lvert x\rvert < 2^{24}$
cos(x)	trigonometric cosine of x (radians)	$\lvert x\rvert < 2^{24}$
exp(x)	constant e raised to the power x	x < 88.02969
log(x)	natural logarithm of x	x > 0.0
sqrt(x)	square root of x	x ≥ 0.0
atan(x)	arc tangent of x, radians	all float numbers
atan2(x,y)	proper quadrant arctangent(x/y)	all float numbers

This small set of functions satisfies most needs because these functions form the building blocks for other functions and are the basis for many scientific/engineering calculations. Accuracy has been emphasized over efficiency. All the approximations, except the square root, are rational approximations over intervals. They are optimal in the sense of minimizing the maximum error over the interval of approximation. Errors because of the basic approximations are slightly increased because some exact constants are preserved and the approximations are accurate on an interval slightly larger than the interval used.

The latter is of practical importance because the actual argument is reduced to the interval of approximation. Even with careful range reduction, the reduced argument may fall outside the interval of approximation because of arithmetic error in the reduction phase. All these approximations are found in reference [5].

Individually, the basic approximations are accurate when just single precision binary floating-point arithmetic is used. When necessary, range reduction is performed with extended precision arithmetic and extra precise constants. The implementations avoid false errors, i.e. those not attributable to improper arguments. For out-of-range arguments an exception, **domain_error**, is declared. Whenever an out-of-range argument is passed, an exception is raised and control returns to the calling module.

Algorithms– The algorithms and error analysis for the transcendental function approximations are given and analyzed in this section. In addition to the usual theoretical error bounds, the results from actual use of the implementations based on the approximations are included.

sqrt– This algorithm uses the standard Newton's iteration method with a starting approximation that guarantees accuracy for all arguments at the end of three iterations.
If $x = 0$, then $\mathrm{sqrt}(x) = 0$. Otherwise, let the argument be $x = 2^e\, 1.f$. Set

$$y_0 = \begin{cases} (0.5 * 1.f + C_e)\ 2^{e/2} & \text{,if } e \text{ is even} \\ (0.25 * 1.f + C_o)\ 2^{(e-1)/2} & \text{,if } e \text{ is odd} \end{cases}$$

$$\text{where } C_e = \frac{1}{\sqrt{1/8 + \sqrt{8 + 1/8}}}\quad,\ C_o = 4\ C_e - 1$$

and then

$$\begin{aligned} y_1 &= 0.5\ (y_0 + x/y_0) \\ y_2 &= 0.5\ (y_1 + x/y_1) \\ y_3 &= y_2 - 0.5\ (y_2 - x/y_2) \end{aligned}$$

where y_3 is the approximate square root.

Error Analysis– If y is an approximation to the square root of x, and we define δ by

$$y = \sqrt{x}\left(\frac{1 + \delta}{1 - \delta}\right)$$

then

$$\frac{1}{2}\left(y + \frac{x}{y}\right) = \sqrt{x}\left(\frac{1 + \delta^2}{1 - \delta^2}\right)$$

[6] shows that $\delta(y_0)$ is accurate to more than 4.79 bits. Thus, y_1, y_2, and y_3 are all larger than the true square root and satisfy the above equation with values of δ smaller than 2^{-10}, 2^{-22}, and 2^{-45}, respectively. Hence, the successive relative errors are 0.075, 0.00262, 3.41×10^{-6}, and 5.8×10^{-12}. The final iteration is a small correction to an already accurate value.

sin-cos– First, the range is reduced to the interval $[-\pi/4, \pi/4]$ using

$$x = (4n + j)\ \pi/2 + y,\ |y| \le \pi/4$$

and the relations

$$\sin(x) = \begin{cases} \sin(y), & \text{if } j = 0 \\ \cos(y), & \text{if } j = 1 \\ -\sin(y), & \text{if } j = 2 \\ -\cos(y), & \text{if } j = 3 \end{cases} \qquad \cos(x) = \begin{cases} \cos(y), & \text{if } j = 0 \\ -\sin(y), & \text{if } j = 1 \\ -\cos(y), & \text{if } j = 2 \\ \sin(y), & \text{if } j = 3 \end{cases}$$

The approximations for sin(y) and cos(y) in the reduced range are

$$\sin(y) = y + y^3(-0.16666653 + y^2 \; (0.0083320645 - 0.00019502220 \; y^2))$$

$$\cos(y) = 1 + y^2(-0.5 + y^2 \; (0.041666644 + y^2 \; (0.000024423102y^2 - 0.0013887229)))$$

These approximations are correct to more than 27 and 32 bits, respectively.

Error Analysis– In the sine approximation, rounding error occurs in squaring and cubing y which can generate 1 and 2 bits of error respectively, and truncating coefficients. However, the number added to y never exceeds 0.1 in relative size, so more than 3 bits are shifted off the end, leaving, at most, a single rounding error. Actual accuracy for the sine using rounded arithmetic is more than 23.9 bits.

In the cosine approximation, rounding errors occur in squaring y and the evaluation of the polynomial added to 1. When rounded arithmetic is used the actual accuracy is almost 23.6 bits for the cosine.

Error in the range reduction step occurs as a result of the less than full double precision of a result. Since the error is limited to less than 2 bits, no error occurs until the argument to the sine/cosine routine exceeds 2^{22}. Thus, only in very unusual circumstances will this be a concern.

exp– The exponential function is written

$$\exp(x) = 2^n \; \exp(y)$$
$$y = x - n \; \log_e(2) \; , \qquad |y| \le \log_e(2)/2$$

The multiplication by 2^n is exact in binary arithmetic. In the basic interval, a rational approximation

$$\exp(y) = 1 + y + \frac{y\left[y - y^2 \; P(y^2)\right]}{2 - y - \left[y^2 \; P(y^2)\right]}$$

is used. Here, the final addition yields an error of at most 0.5 bits. This form preserves weak monotonicity for negative arguments. That is, the strong monotonicity property of the original function $x < y$ implies $\exp(x) < \exp(y)$ is preserved only in the form $x < y$ implies $\exp(x) \le \exp(y)$ in the approximation. This is the best that can be expected from floating-point arithmetic. Some applications require preservation of monotonicity. For IEEE 754-1985 single binary float arithmetic, the polynomial is

$$P(x) = 0.16666612 - 0.0027652702\ x$$

and the actual accuracy using rounded arithmetic is better than 23.8 bits.

log– For any positive x, the natural logarithm can be defined from its values over a limited domain as

$$\log_e(x) = n\ \log_e(2) + \log_e(y), \quad \sqrt{0.5} < y < \sqrt{2}$$

Using binary arithmetic, n and y are obtained exactly from the floating-point representation of x. The approximation for log(y) in the reduced domain is

$$\log(y) = \left(y - 1\right) + \frac{v^3}{Q\left(v^2\right)}, \quad v = \frac{y - 1}{y + 1}$$

In this form, the multiplier of any error in v can be as large as 2.0. To reduce this error to at most 0.395, the identity

$$2v = (y - 1) + v(1 - y)$$

is used to convert the approximation to the form

$$\log(y) = (y - 1) + v\left[(1 - y) + \frac{v^2}{Q\left(v^2\right)}\right]$$

The quantity n log(2) is calculated using an extended precision constant. So, if

$$n\ \log_e(2) = (A,\ B)$$

the value of the approximate logarithm is calculated by summing, from right to left, the terms i n

$$\log(x) = A + (y - 1) + B + v\left[(1 - y) + \frac{v^2}{Q\left(v^2\right)}\right]$$

where

$$Q(z) = 1.5000459 - 0.90463380\ z$$

is accurate to more than 25 bits. With IEEE 754-1985 rounded arithmetic, the actual relative accuracy is greater then 23.4 bits in the reduced interval, and more than 24 bits otherwise.

atan– For any x, the arc tangent function can be stably evaluated using the continued fraction

$$\operatorname{atan}(x) = \frac{x}{1 +}\ \frac{x^2}{3 +}\ \frac{4x^2}{5 +}\ \frac{9x^2}{7 +} \cdots \frac{(k\,x)^2}{\left(2\ k + 1\right) +} \cdots$$

However, this is not economical because the amount of work necessary to maintain a given level of accuracy grows unboundedly with the argument. To keep the amount of work small, the range of x is reduced using

$$\mathrm{atan}(x) = \mathrm{sgn}(x)\ \pi/2 - \mathrm{atan}(1/x)$$

for $|x| > 1$, and further reducing the range with

$$\mathrm{atan}(x) = \mathrm{sgn}(x) \left[\frac{\pi}{6} + \mathrm{atan}(y)\right], \quad y = \frac{|x|\sqrt{3} - 1}{|x| + \sqrt{3}}, \quad \tan\left(\frac{\pi}{12}\right) < |x| < 1$$

Compounding of induced errors is avoided with the elaborated approximation

$$\mathrm{atan}(x) = \begin{cases} \mathrm{atan}(x) & , |x| < \tan\left(\frac{\pi}{12}\right) = \frac{\sqrt{3} - 1}{\sqrt{3} + 1} \\ \mathrm{sgn}(x) \left[\frac{\pi}{6} + \mathrm{atan}(y)\right] & , y = \frac{|x|\sqrt{3} - 1}{|x| + \sqrt{3}}, \quad \tan\left(\frac{\pi}{12}\right) < |x| < 1 \\ \mathrm{sgn}(x) \left[\frac{\pi}{3} - \mathrm{atan}(y)\right] & , y = \frac{\sqrt{3} - |x|}{1 + |x|\sqrt{3}}, \quad 1 < |x| < \frac{1}{\tan\left(\frac{\pi}{12}\right)} \\ \mathrm{sgn}(x)\ \frac{\pi}{2} - \mathrm{atan}(y) & , y = 1/x, \quad x > \frac{1}{\tan\left(\frac{\pi}{12}\right)} \end{cases}$$

The basic approximating form is

$$\mathrm{atan}(y) \approx y - y^3/Q(y^2)$$

where $Q(z)$ is a polynomial. The polynomial

$$Q(z) = 3.0000071 + z\ (1.7994048 - 0.19159707\ z)$$

gives results that are accurate to more than 28 bits. When rounded arithmetic is used, the approximation is correct to 24, 22.8, 23.3, and 23.56 bits in the four ranges. Extended precision helps preserve accuracy in forming the numerator in the second range.

atan2– The proper quadrant arc tangent function is defined, using the atan function, by

$$\mathrm{atan2}(x,y) = \begin{cases} \mathrm{sgn}(x)\ \pi/2, & \text{if } y = 0 \\ \mathrm{atan}(x/y), & \text{if } y > 0 \\ \mathrm{sgn}(x)\ \pi + \mathrm{atan}(x/y), & \text{if } y < 0 \end{cases}$$

For this function there are two errors beyond those in atan. The first is the error from forming x/y. The second is from the addition of π. The second can be avoided with an extended precision value of π. But the first induces an error into the argument of the atan function which cannot be avoided or corrected. The relative error is less than 0.5 bits and this is reflected in an increased error not exceeding 0.5 bits.

The entire mathematics package, MathLib, is given in appendix D.

Appendix A. Low Level Routines – Package IEEE754Lib

```
package IEEE754Lib is
------------------------------------------------------------------------------
   type fpcomponents is
      record
         s : integer 0..1;
         e : integer -128 .. 127;
         f : integer 0 .. 8#77777777#;
      end record;
------------------------------------------------------------------------------
   procedure Strip (x : in float; r :out fpcomponents);
   function Assemble ( r : in fpcomponents) return float;
end IEEE754Lib;
------------------------------------------------------------------------------
------------------------------------------------------------------------------
with unchecked_conversion;
package body IEEE754Lib is
   t23     : constant integer := 2**23;
   x_as_integer,
   local_e,
   local_f : integer;
   function float_to_integer is new unchecked_conversion (float, integer);
   function integer_to_float is new unchecked_conversion (integer,float);
   procedure Strip (x : in float; r : out fpcomponents) is
      begin
         if x < 0.0 then r.s := 1; else r.s := 0; end if; -- sign field
         x_as_integer := float_to_integer(abs x);
         local_e := x_as_integer / t23; -- exponent field
         local_f := x_as_integer mod t23; -- fractional field
         if local_e >0 then
            r.e := local_e - 127;
            r.f := local_f + t23;
         else
            r.e :=  -128;
            r.f := local_f;
         end if;
   end Strip;
------------------------------------------------------------------------------
   function Assemble ( r : in fpcomponents) return float is
      begin
         if r.e = -128 then
            x_as_integer := r.f;
         else
            x_as_integer := (r.e + 127) * t23 + (r.f - t23);
         end if;
         if r.s = 1 then
            return -integer_to_float(x_as_integer);
         else
            return  integer_to_float(x_as_integer);
         end if;
   end Assemble;
end IEEE754Lib;
```

Appendix B. The Extended Arithmetic Package – ExtendedArithmetic

```
--------------------------------------------------------------------------------
package ExtendedArithmetic is
-- a portable extended arithmetic that yields slightly dirty double accuracy.
-- It was not implemented using a new data type and overloaded operations,
-- because it is not intended for extensive use.
-- A doubly precise numbers is an ordered pair of type float
-- numbers (x,xx) with the property: x=fl(x+xx) exactly.
--------------------------------------------------------------------------------
-- The procedures are:
--      Emult (a, c, x, xx)           (x,xx) = a * c exactly
--      Empy (a, aa, c, cc, x, xx)    (x,xx) = (a,aa) times (c,cc)
--      Ediv (a, aa, c, cc, x, xx)    (x,xx) = (a,aa) divided by (c,cc)
--      Eadd (a, aa, c, cc, x, xx)    (x,xx) = (a,aa) plus (c,cc)
--all arguments are floating-point numbers.
--------------------------------------------------------------------------------
   procedure Emult(a, c : in float; x, xx : out float);
   procedure Empy(a, aa, c, cc : in float; x, xx : out float);
   procedure Eadd(a, aa, c, cc : in float; x, xx : out float);
   procedure Ediv(a, aa, c, cc : in float; x, xx : out float);
end ExtendedArithmetic;
--============================================================================
package body ExtendedArithmetic is
-- Local variables
   q, qq,
   z, zz,
   a1, a2,
   c1, c2,
   c21, c22,
   u,
   t,
   su        : float;
   w         : constant float := 4097.0; -- IEEE 754 specific
--------------------------------------------------------------------------------
   procedure Emult(a, c : in float; x, xx : out float) is
      begin
         t := a * w;
         a1 := (a - t) + t;
         a2 := a - a1;
         t := c * w;
         c1 := (c - t) + t;
         c2 := c - c1;
         t := c2 * w;
         c21 := (c2 - t) + t;
         c22 := c2 - c21;
         u := a * c;
         x := u;
         xx:= ((((a1*c1 - u) + a1 * c2) + c1 * a2) + c21 * a2) + c22 * a2;
      end Emult;
--------------------------------------------------------------------------------
```

```
   procedure Empy(a, aa, c, cc : in float; x, xx : out float) is
      begin
         Emult(a, c, z, q);
         zz := (a * cc + c * aa) + q;
         u := z + zz;
         x := u;
         xx := (z - u) + zz;
      end Empy;
-------------------------------------------------------------------------------
   procedure Eadd(a, aa, c, cc : in float; x, xx : out float) is
      begin
         z := a + c;
         q := a - z;
         zz := (((q + c) + (a - (q + z))) + aa) + cc;
         u := z + zz;
         x := u;
         xx := (z - u) + zz;
      end Eadd;
-------------------------------------------------------------------------------
   procedure Ediv(a, aa, c, cc : in float; x, xx : out float) is
      begin
         z := a / c;
         Emult(c, z, q, qq);
         zz := ((((a - q) - qq) + aa) - z * cc) / c;
         u := z + zz;
         x := u;
         xx := (z - u) + zz;
   end Ediv;
-------------------------------------------------------------------------------
end ExtendedArithmetic;
```

Appendix C. The Complex Arithmetic Package – ComplexArithmetic

```
--------------------------------------------------------------------------
-- This package provides a complex arithmetic overloading the usual
-- arithmetic operators with its own.  The new operators work on objects
-- of type complex, which is a pair of float numbers, optionally mixed with
float numbers.
-- Emphasis has been placed upon accuracy of the results.
-- A complex square root is included.
--------------------------------------------------------------------------
package ComplexArithmetic is
-- new data type
   type complex is record
      real,
      imag : float;
   end record;
--  Arithmetic operations
   function "+"(x, y : complex) return complex;
   function "+"(x : float; y : complex) return complex;
   function "+"(x : complex; y : float) return complex;
   function "-"(x, y : complex) return complex;
   function "-"(x : float; y : complex) return complex;
   function "-"(x : complex; y : float) return complex;
   function "-"(x : complex) return complex;
   function "*"(x, y : complex) return complex;
   function "*"(x : float; y : complex) return complex;
   function "*"(x : complex; y: float) return complex;
   function "/"(x, y : complex) return complex;
   function "/"(x : float; y : complex) return complex;
   function "/"(x : complex; y : float) return complex;
-- manipulation of complex numbers
-- real part
   function realpart(x : complex) return float;
-- imaginary part
   function imagpart(x : complex) return float;
-- form a complex number from its piees
   function cmplx(x, y : float) return complex;
-- complex conjugate
   function conjg(x : complex) return complex;
-- complex length
   function cabs(x : complex) return float;
-- Square root
   function csqrt(x : complex) return complex;
----------------------------------------------------------------------------
end ComplexArithmetic;
```

```
--=========================================================================
with Mathlib; use Mathlib;
package body ComplexArithmetic is
   function "+"(x, y : complex) return complex is
      begin
         return (x.real + y.real, x.imag + y.imag);
      end;
---------
   function "+"(x : float; y : complex) return complex is

      begin
         return (x + y.real, y.imag);
      end;
---------
   function "+"(x : complex; y : float) return complex is
      begin
         return (x.real + y, x.imag);
      end;
--------------------------------------------------------------------------
   function "-"(x, y : complex) return complex is
      begin
         return (x.real - y.real, x.imag - y.imag);
      end;
--------------------------------------------------------------------------
   function "-"(x : float; y : complex) return complex is
      begin
         return (x - y.real, -y.imag);
      end;
---------
   function "-"(x : complex; y : float) return complex is
      begin
         return (x.real - y, x.imag);
      end;
---------
   function "-"(x : complex) return complex is
      begin
         return (-realpart(x), -imagpart(x));
      end "-";
--------------------------------------------------------------------------
   function "*"(x, y : complex) return complex is
      begin
         return (x.real * y.real - x.imag * y.imag,
                 x.real * y.imag + x.imag * y.real);
      end;
---------
   function "*"(x : float; y : complex) return complex is
      begin
         return (x * y.real, x * y.imag);
      end;
---------
   function "*"(x : complex; y: float) return complex is
      begin
         return (x.real * y, x.imag * y);
      end;
--------------------------------------------------------------------------
```

```
   function "/"(x, y : complex) return complex is
      denom, prodef : float;
      begin
         if y.real = 0.0 and y.imag = 0.0 then
            raise Numeric_Error;
         end if;
         if abs y.real >= abs y.imag then
            prodef := y.imag / y.real;
            denom := 1.0 / (y.real + y.imag * prodef);
            return (denom * (x.real + x.imag * prodef),
                    denom * (x.imag - x.real * prodef));
         else
            prodef := y.real/y.imag;
            denom := 1.0 / (y.imag + y.real * prodef);
            return (denom * (x.imag + x.real * prodef),
                    denom * (x.imag * prodef - x.real));
         end if;
      end;
---------
   function "/"(x : float; y : complex) return complex is
      denom, prodef : float;
      begin

         if y.real = 0.0 and y.imag = 0.0 then
            raise Numeric_Error;
         end if;
         if abs y.real >= abs y.imag then
            prodef := y.imag / y.real;
            denom := x / (y.real + y.imag * prodef);
            return (denom, -denom * prodef);
         else
            prodef := y.real/y.imag;
            denom := x / (y.imag + y.real * prodef);
            return (denom * prodef, -denom);
         end if;
      end;
---------
   function "/"(x : complex; y : float) return complex is
      begin
         return x * (1.0 / y);
      end;
------------------------------------------------------------------------------
   function cabs(x : complex) return float is
      y : float;
      begin
         if x.real = 0.0 and x.imag = 0.0 then
            return 0.0;
         end if;
         if abs x.real >= abs x.imag then
            return (abs x.real) * sqrt((x.imag / x.real) ** 2 + 1.0);
         else
            return (abs x.imag) * sqrt((x.real / x.imag) ** 2 + 1.0);
         end if;
      end cabs;
------------------------------------------------------------------------------
```

```
   function realpart(x : complex) return float is

      begin
         return x.real;
      end;
--------------------------------------------------------------------------
   function imagpart(x : complex) return float is
      begin
         return x.imag;
      end;
--------------------------------------------------------------------------
   function cmplx(x, y : float) return complex is
      begin
         return (x, y);
      end;
--------------------------------------------------------------------------
   function conjg(x : complex) return complex is
      begin
         return (x.real, -x.imag);
      end;
--------------------------------------------------------------------------
   function csqrt(x : complex) return complex is

      q : float;
      begin
         if x.imag = 0.0 then
            if x.real >= 0.0 then
               return (sqrt(x.real), 0.0);
            else
               return (0.0, sqrt(-x.real));
            end if;
         else
            q := sqrt(0.5 * (abs (x.real) + cabs(x)));
            if x.real >= 0.0 then
               return (q, 0.5 * x.imag / q);
            else
               return (-0.5 * x.imag / q, -q);
            end if;
         end if;
      end csqrt;
--------------------------------------------------------------------------
end ComplexArithmetic;
```

```
---------------------------------------------------------------------------
-- Ada does not come with a mathematical function package of any kind, so
-- this one is provided for the most common elementary and transcendental
-- functions.

-- IEEE 754-1985 single binary float format is assumed.

-- A single exception is defined in the package.  This exception, named
-- domain_error, is raised whenever an illegal argument is provided to one of
-- the routines.

package MathLib is
-------------------------------------------------------------------------------
   domain_error        : exception;
   function sin  (x    : float) return float;    -- sine of x
   function cos  (x    : float) return float;    -- cosine of x
   function exp  (x    : float) return float;    -- e to the power x
   function log  (x    : float) return float;    -- natural logarithm of x
   function sqrt (x    : float) return float;    -- square root of x
   function atan (x    : float) return float;    -- arc tangent of x
   function atan2(x, y : float) return float;    -- proper quadrant atan(x/y)
   function xmod (x, y : float) return float;    -- (x - trunc(x / y) * y)
   function floor(x    : float) return integer; -- nearest integer <= x
   function ceil (x    : float) return integer;  --nearest integer >= x
   function trunc(x    : float) return integer;  -- nearest integer in [0, x)
   function urand return float; -- uniform random numbers in (0, 1)
   procedure urandinit(seed1, seed2 : in integer); -- reinitialize urand
end MathLib;

--===========================================================================
   with IEEE754Lib, ExtendedArithmetic; Use IEEE754Lib, ExtendedArithmetic;
package body MathLib is
-------------------------------------------------------------------------------
      r            : fpcomponents;
-- urand default seeds
      s2 : integer := 41569;
      s1 : integer := 46013;
-- atan constants
      tanpiby12 : constant float := 0.26794919;
      sqrt3     : constant float := Assemble((0,   0, 8#67331727#));
      sqrt3lo   : constant float := Assemble((0, -25, 8#41302312#));
      thirdpi   : constant float := 1.0471976;
      sixthpi   : constant float := 0.52359878;
      AtanR     : constant array (0 .. 2) of float := ( -0.19159707,
                                                         1.7994048 ,
                                                         3.0000072 );
-- sine-cosine constants
      FourthPi : constant float := 0.78539816;
      HalfPi   : constant float := Assemble((0,   0, 8#62207732#));
      HalfPiLo : constant float := Assemble((0, -24, 8#50420551#));
      sine     : constant array(0 .. 2) of float :=  (-0.19502220e-3,
                                                       0.83320645e-2,
                                                      -0.16666653   );
      cosine   : constant array (0 .. 2) of float := ( 0.24423102e-4,
                                                      -0.13887229e-2,
                                                       0.41666644e-1);
```

```
-- exp constants
     ln2hi   : constant float := Assemble((0,  -1, 8#54271027#));
     ln2lo   : constant float := Assemble((0, -25, 8#75750717#));
     halfln2 : constant float := 0.34657359;
     domain  : constant float := 87.6831092;
     ExpR    : constant array (0 .. 1) of float := (-0.27652702e-2,
                                                     0.16666612    );
-- log constants
     sqrt2   : constant float := Assemble((0,   0, 8#55202363#));
     LogR    : constant array (0 .. 1) of float := ( -0.90463380,
                                                      1.5000459);
-- sqrt constants
     Ceven   : constant float := 0.48260052;
     Codd    : constant float := 0.93040208;
     function q(y, y2 : float) return float is
        begin
           return y-(y*y2)/(((AtanR(0)*y2+AtanR(1))*y2)+AtanR(2));
     end q;
------------------------------------------------------------------------------
-- uniform random number generator based upon
--    combined linear congruential generators
   procedure urandinit(seed1, seed2 : in integer) is
     begin
        s1 := seed1;
        s2 := seed2;
        if s1 <= 0 then
              s1 := s1 + 2147483563;
              if s1 <= 0 then
                 s1 := s1 + 2147483563;
               end if;
        end if;
        if s2 <= 0 then
           s2 := s2 + 2147483399;
           if s2 <= 0 then
              s2 := s2 + 2147483399;
           end if;
        end if;
        if s1 > 2147483563 then
           s1 := s1 - 2147483563;
        end if;
        if s2 > 2147483399 then
           s2 := s2 - 2147483399;
        end if;
     end urandinit;
-------------------
```

```
 function urand return float is
    z, k : integer;
    begin
    k := s1 / 53668;
    s1 := 40014 * (s1 - k * 53668) - k * 12211;
    if s1 < 0 then
       s1 := s1 + 2147483563;
    end if;
    k := s2 / 52774;
    s2 := 40692 * (s2 - k * 52774) - k * 3791;
    if s2 < 0 then
       s2 := s2 + 2147483399;
    end if;
    z := s1 - s2;
    if z < 1 then
       z := z + 2147483562;
    end if;
    return float(z) * 4.656613e-10;
 end  urand;
--------------------------------------------------------------------------------
 function floor(x : float) return integer is
    y   : float := abs x;
    fix : integer := integer(y);
    z   : float := float(fix);
 begin
    if x >= 0.0 then
       if z <= y then
          return fix;
       else
          return fix - 1;
       end if;
    else
       if z < y then
          return -1-fix;
       else
          return -fix;
       end if;
    end if;
 end floor;
--------------------------------------------------------------------------------
 function ceil(x : float) return integer is
    begin
       return -floor(-x);
    end ceil;
--------------------------------------------------------------------------------
 function trunc(x : float) return integer is
    begin
       if x >= 0.0 then
          return floor(x);
       else
          return ceil(x);
       end if;
   end trunc;
--------------------------------------------------------------------------------
```

```
   function SineR(x : float) return float is
      x2    : float := x * x;
      begin
         return x + x * x2 * ((sine(0) * x2 + sine(1)) * x2 + sine(2));
      end SineR;
-------------------------------------------------------------------------------
   function CosineR(x : float) return float is
      x2     : float := x * x;
      begin
         return 1.0 + x2 * (-0.5 + x2 *
                          ((cosine(0) * x2 + cosine(1)) * x2 + cosine(2)));
      end CosineR;
-------------------------------------------------------------------------------
   procedure ReduceTrigArg( x : float;
                           yy : out float;
                           jj : out integer) is
      y, whi, wlo : float;
      j           : integer;
      z           : float := abs x;
      begin
         j := trunc(z / HalfPi);
         empy(float(j), 0.0, HalfPi, HalfPiLo, whi, wlo);
         y := (z - whi) - wlo;
         if y > FourthPi then
            y := (y - HalfPi) - HalfPiLo;
            j := j + 1;
         end if;
         if x < 0.0 then
            yy := -y;
            jj := (4-j mod 4) mod 4;
         else
            yy := y;
            jj := j mod 4;
         end if;
   end ReduceTrigArg;
-------------------------------------------------------------------------------
   function sin(x : float) return float is
      y : float;
      j : integer;
      begin
         if abs x >= 16777216.0 then
            raise domain_error;
         end if;
        ReduceTrigArg(x, y, j);
         case j is
            when 0 => return  SineR(y);
            when 1 => return  CosineR(y);
            when 2 => return -SineR(y);
            when 3 => return -CosineR(y);
            when others => null;
         end case;
      end sin;
-------------------------------------------------------------------------------
```

```
   function cos(x:float) return float is
      y : float;
      j : integer;
      begin
         if abs x >= 16777216.0 then
            raise domain_error;
         end if;
         ReduceTrigArg(x, y, j);
         case j is
            when 0 => return  CosineR(y);
            when 1 => return -SineR(y);
            when 2 => return -CosineR(y);
            when 3 => return  SineR(y);
            when others => null;
         end case;
      end cos;
-----------------------------------------------------------------------------
   function exp(x : float) return float is
      n    : integer;
      fhi,
      flo,
      y,
      y2,
      z    : float;
      begin
         if x >= domain then
            raise domain_error;  --overflow will occur.
         elsif x <= -domain then
            return 0.0;
         end if;
         n := integer(x / ln2hi);
         y := x - float(n) * ln2hi;
         if y > 0.5 * ln2hi then
            n := n + 1;
         elsif y < -0.5 *  ln2hi then
            n := n-1;
         end if;
         Empy(float(n), 0.0, ln2hi, ln2lo, fhi, flo);
         y := (x - fhi) - flo;
         y2 := y * y;
         z := y - y2 * (ExpR(0) * y2 + ExpR(1));
         return (1.0 + y + y * z / (2.0 - z)) * 2.0 ** integer(n);
      end exp;
-----------------------------------------------------------------------------
```

```
function log(x : float) return float is
   y, ln,
   v, v2  : float;
   n      : integer;
   begin
      if x <= 0.0 then
         raise domain_error;
      end if;
      Strip(x, r);
      n := r.e;
      y := Assemble((r.s,0,r.f));
      if y > sqrt2 then
         y := y * 0.5;
         n := n + 1;
      elsif y < 0.5 * sqrt2 then
         y := y + y;
         n := n - 1;
      end if;
      v  := ( y - 1.0) / ( y + 1.0);
      v2 := v*v;
      ln := float(n);
      return ((((v2 / (LogR(0) * v2 + LogR(1)) + (1.0 - y)) * v
              + ln2lo * ln) + (y - 1.0)) + ln * ln2hi);
   end log;
```

```
function atan(x : float) return float is
   y,
   y2,
   ylo,
   arctan    : float;
   begin
      if abs x <= tanpiby12 then
         y := x;
         y2 := y * y;
         arctan := q(y,y2);
         return arctan;
      elsif abs x in tanpiby12 .. 1.0 then
         empy(abs(x), 0.0, sqrt3, sqrt3lo, y, ylo);
         y := ((y-1.0) + ylo) / (sqrt3 + abs x);
         y2 := y * y;
         arctan := q(y, y2);
         arctan := sixthpi + arctan;
         if x >= 0.0 then
            return arctan;
         else
            return -arctan;
         end if;
      elsif abs x in 1.0 .. 1.0/tanpiby12 then
         y := (sqrt3 - abs x) / (1.0 + sqrt3 * abs x);
         y2 := y * y;
         arctan := q(y,y2);
         arctan := thirdpi - arctan;
         if x >= 0.0 then
            return arctan;
         else
            return -arctan;
         end if;
      else
         y := 1.0 / x;
         y2 := y * y;
         arctan := q(y,y2);
         if x >= 0.0 then
            return (HalfPi - arctan) + HalfPiLo;
         else
            return -(HalfPi + arctan);
         end if;
      end if;
end atan;
------------------------------------------------------------------------------
```

```
    function atan2(x, y : float) return float is
       z      : float;
       begin
          if y = 0.0 then
             if x >= 0.0 then
                return HalfPi;
             else
                return -HalfPi;
             end if;
          end if;
          z := x/y;
          if y > 0.0 then
                return atan(z);
          else
             if x > 0.0 then
                return   (2.0 * HalfPi + atan(z)) + 2.0 * HalfPiLo;
             else
                return -((2.0 * HalfPi - atan(z)) + 2.0 * HalfPiLo);
             end if;
          end if;
       end atan2;
--------------------------------------------------------------------------------
    function sqrt(x : float) return float is
       y      : float;
       begin
          if x <0.0 then
             raise domain_error;
          elsif x = 0.0 then
             return 0.0;
          end if;
          Strip(x,r);
          y := Assemble((r.s, 0, r.f));
          if r.fe mod 2 = 0 then
             y := (0.5 * y + Ceven) * 2.0 ** integer(r.e/2);
          else
             y := (0.25 * y + Codd) * 2.0 ** integer((r.e - 1) / 2);
          end if;
          y := 0.5 * (y + x / y);
          y := 0.5 * (y + x / y);
          return y - 0.5 * (y - x / y);
       end sqrt;
--------------------------------------------------------------------------------
       function xmod(x, y : float) return float is
          z : float;
          begin
             z := x - float(trunc(x/y)) * y;
             if z < 0.0 then
                z := z + abs y;
             end if;
          return z;
       end xmod;
                            end MathLib;
```

REFERENCES

1. ANSI/MIL-STD-1815A, Military Standard Ada™ Programming Language, 22 January 1983.

2. ANSI/IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, Inc., New York, N. Y., 1985.

3. Linnainmaa, Seppo: Software for Doubled-Precision Floating Point Computations, ACM Transactions of Mathematical Software, vol. 7, No. 3, 1981, pp. 272-283.

4. L'Ecuyer, Pierre: Efficient and Portable Combined Random Number Generators. Communications of the ACM, vol. 31, No. 6, 1988, pp. 742-749, 774.

5. Manos, Paul; and Turner, L. Richard: Constrained Chebyshev Approximations to Some Elementary Functions Suitable for Evaluation with Floating-Point Arithmetic. NASA TN D-6698, 1972.

6. Kahan, William: Implementation of Algorithms. Part 1, AD-769 124, NTIS, Springfield, VA., 1973.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA TM-102256	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle
Basic Mathematical Function Libraries for Scientific Computation

5. Report Date
December 1989

6. Performing Organization Code

7. Author(s)
David C. Galant

8. Performing Organization Report No.
A-90015

10. Work Unit No.
549-03-61

9. Performing Organization Name and Address
Ames Research Center
Moffett Field, CA 94035-1000

11. Contract or Grant No.

13. Type of Report and Period Covered
Technical Memorandum

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes
Point of Contact: David C. Galant, Ames Research Center, MS 244-4, Moffett Field, CA 94035-1000
(415) 604-4851 or FTS 464-4851

16. Abstract

Ada* packages implementing selected mathematical functions for the support of scientific and engineering applications have been written. The packages provide the Ada programmer with the mathematical function support found in the languages Pascal and Fortran as well as an extended precision arithmetic and a complete complex arithmetic. The algorithms used are fully described and analyzed. Implementation assumes that the Ada type FLOAT objects fully conform to the IEEE 754-1985 standard for single binary floating-point arithmetic, and that INTEGER objects are 32-bit entities. Codes for the Ada packages are included as appendixes.

*Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

17. Key Words (Suggested by Author(s)) Scientific computation Transcendental functions Ada packages	18. Distribution Statement Unclassified – Unlimited Subject category – 61

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 29	22. Price A03