Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Kok

Proposal for standard mathematical packages in Ada

Department of Numerical Mathematics Report NM-R8718 November

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

Proposal for Standard Mathematical Packages in Ada

Jan Kok
Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

On behalf of the Ada-Europe Numerics Working Group we propose Ada packages of mathematical types, constants, operators and subprograms to be added to the standard Ada program library. These include packages of elementary mathematical functions, of mathematical constants, of random number generators, and of (cartesian and polar) complex types and related arithmetic operations.

1980 Mathematics Subject Classification: 69D49, 65-04.
Key Words & Phrases: Ada, high level language, elementary functions, mathematical packages, standard functions, scientific libraries.
Notes:
1. This report will be submitted for publication elsewhere.

1. INTRODUCTION

Unlike many other programming languages the language Ada (ANSI-MIL-STD 1815A, 1983) does not provide standard declarations of the traditional, well-known, elementary mathematical functions. Definitions can quite well in several ways be expressed in Ada. The absence of a common definition for elementary functions has therefore been the cause of proliferation of different designs of manufacturer-supplied packages. This clearly collides with the portability aim of the language Ada, for which otherwise several language features have particularly been designed for the enhancement of the portability of software.

Among early activities for obtaining a common specification for the elementary functions that would be acceptable to the numerical community we mention a design by R. Firth and the proposal given in the NPL/CWI Guidelines (Symm et al., 1984) that was also published in a separate article in *Ada Letters* (Kok & Symm, 1984). An initial portable implementation (function bodies) was produced by Whitaker & Eicholtz (1982).

Since 1986 co-operative activities for achieving a commonly adopted specification of mathematical functions have taken place frequently, building on work for the core design of a planned numerical library in Ada in the project *Pilot implementations of basic modules for large portable numerical libraries in Ada* (MAP 750). During a stay of *Ada-Europe Numerics Working Group* members at Argonne National Laboratory a first draft was produced of specifications of mathematical packages, including an Ada specification for elementary functions. This draft was subsequently presented and discussed in meetings of the SIGAda

Report NM-R8718
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Numerics Users Committee Working Group in Pittsburgh and of the Ada-Europe Numerics Working Group in Brussels.

The present paper is the result evolved from this draft, taking into account discussions in Ada-Europe Numerics WG meetings in Brussels and in SIGAda Numerics Users Committee WG meetings at several places in the USA, other comments received and also a couple of elaborate working papers by Ada-Europe Numerics WG members.

In this paper the *Ada-Europe Numerics Working Group* presents an Ada specification of the elementary functions, as the first of a set of mathematical packages of facilities for which international adoption of an Ada specification and semantic specification is considered to be highly desirable. Ada specifications of other desirable mathematical facilities, including a basic random number generator, floating-point primitives and declarations for composite data structures like *vectors*, *matrices* and *complex numbers* are also given in Chapter 2.

It is widely agreed that a common Ada specification for the elementary functions should be accompanied by a detailed statement of the accuracies to be expected of the results of all elementary functions, related to the employed precision. Work on such accuracy statements is still ongoing, in particular in the SIGAda Numerics Users Committee WG.

For clearness the following proposal only contains Ada package declarations, followed by a (usually mathematical) semantic specification for all facilities. In a preliminary version of the first draft mentioned above a start was made of a Rationale for the Ada specifications presented, reviewing the background and discussing the possibilities considered and the choices made. It is expected that an updated version, in agreement with the final contents of the proposal, will be published that will also contain systematically the results of the many discussions.

2. PROPOSAL

In Section 2.1 we propose (generic) package declarations as additions to the standard Ada program library. These packages should be present in the Ada program library whenever floating-point arithmetic is provided.

Separate Ada package declarations are presented for the following areas :-

- basic mathematical provisions like elementary mathematical functions, mathematical constants, and a uniform random number generator,

- auxiliary or supporting facilities:

  - one package containing declarations of basic auxiliary functions, like the sign of a number or the maximum of two values,

  - a second (generic) package for providing floating-point primitive functions together with environment details which cannot be obtained through the Ada floating-point type attributes or the standard package SYSTEM,

- composite mathematical types and the related operators, viz. COMPLEX, VECTOR and MATRIX, together with arithmetic complex operations and complex functions based on a library package providing a standard floating-point type REAL that satisfies minimal accuracy requirements.

Section 2.2 contains the semantic specifications for all subprogram declarations proposed.

2.1. Proposed Ada package specifications

The declarations presented are intended for floating-point types. In most cases analogous definitions could be given for fixed-point types.

2.1.1. Basic mathematical packages

We propose standard Ada specifications for mathematical facilities of general usefulness, viz. for common mathematical constants, elementary functions, and a generator of randomly distributed floating-point numbers.

a) Elementary functions

The following generic package contains function declarations for a set of elementary (mathematical) functions.

```
with MATHEMATICAL_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <>;
package GENERIC_ELEMENTARY_FUNCTIONS is
   --
   -- Declare the basic mathematical functions
   function SQRT     (X       : FLOAT_TYPE) return FLOAT_TYPE;
   function LOG      (X       : FLOAT_TYPE) return FLOAT_TYPE;
   function LOG      (X, BASE : FLOAT_TYPE) return FLOAT_TYPE;
   function EXP      (X       : FLOAT_TYPE) return FLOAT_TYPE;
   function "**"     (X, Y    : FLOAT_TYPE) return FLOAT_TYPE;

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

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

   function SINH     (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function COSH     (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function TANH     (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function COTH     (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCSINH  (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCCOSH  (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCTANH  (X         : FLOAT_TYPE) return FLOAT_TYPE;
   function ARCCOTH  (X         : FLOAT_TYPE) return FLOAT_TYPE;
   --
   -- Exception, to be raised for implementation-independent range constraints
   ARGUMENT_ERROR : exception renames MATHEMATICAL_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```

A standard instance for FLOAT should be called ELEMENTARY_FUNCTIONS. Where an implementation provides additional pre-defined floating-point types SHORT_FLOAT or LONG_FLOAT, it is recommended that analogous instances SHORT_ELEMENTARY_FUNCTIONS or LONG_ELEMENTARY_FUNCTIONS should also be provided.

b) Declaration of exception for mathematical functions

The following package provides one exception declaration for all instances of the above generic package of elementary functions.

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

c) Mathematical constants

The package MATHEMATICAL_CONSTANTS contains number declarations through which names are introduced for the values of the available constants with optimal accuracy. We assume that (universal expressions containing) real literals are given that are accurate enough for all possible real types.

```
package MATHEMATICAL_CONSTANTS is
   --
   -- Number declarations for the mathematical constants
   PI            : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   TWO_PI        : constant := 2 * PI;
   HALF_PI       : constant := PI / 2;
   ONE_OVER_PI   : constant := 1.0 / PI;
   SQRT_PI       : constant := 1.7724_53850_90551_60272_98167_48334_11452;

   NATURAL_E     : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ONE_OVER_E    : constant := 1.0 / NATURAL_E;

   GAMMA         : constant := 0.5772_15664_90153_28606_06512_09008_24024;

end MATHEMATICAL_CONSTANTS;
```

d) Random numbers

The (generic) package GENERIC_RANDOM_GENERATOR contains procedures for generating pseudo-random numbers with a uniform distribution lying strictly between 0.0 and 1.0, for initialising a repeatable sequence of random numbers, for copying one random seed to another, and for storing and retrieving the internal representation of a seed value. The facility for generating random numbers is also available as a function that gives no user-control over the seed.

```
generic
   type FLOAT_TYPE is digits <>;
   STORE_SIZE : in INTEGER;
package GENERIC_RANDOM_GENERATOR is

   type RANDOM_SEED is limited private;

   procedure RANDOM                (SD          : in out RANDOM_SEED;
                                    NUMBER      : out FLOAT_TYPE);

   procedure SET_RANDOM_SEED       (N           : in INTEGER;
                                    SD          : out RANDOM_SEED);

   procedure COPY_RANDOM_SEED      (OLD_SEED : in RANDOM_SEED;
                                    NEW_SEED: out RANDOM_SEED);
```

```
function RANDOM                          return FLOAT_TYPE;

type STORE_TYPE is array (1 .. STORE_SIZE) of INTEGER;

procedure SAVE_RANDOM_SEED     (SD       : in RANDOM_SEED;
                                STORE    : out STORE_TYPE);
procedure RESTORE_RANDOM_SEED (STORE     : in STORE_TYPE;
                                SD       : out RANDOM_SEED);

private

   -- implementation dependent: the seed may be implemented as some
   -- number of integer components; this number will vary according to
   -- the integer wordlength.
   --
   -- Implementation details are omitted.

end GENERIC_RANDOM_GENERATOR;
```

A standard instantiation for the generic random generator package should be provided for the pre-defined type FLOAT:

```
with GENERIC_RANDOM_GENERATOR;
package RANDOM_GENERATOR is new GENERIC_RANDOM_GENERATOR(FLOAT, 4);
-- Assuming that the value 4 is suitable for the size of STORE_TYPE
```

Where an implementation provides additional pre-defined floating-point types SHORT_FLOAT and/or LONG_FLOAT, it is recommended that analogous instantiations SHORT_RANDOM_GENERATOR and/or LONG_RANDOM_GENERATOR should also be provided.

2.1.2. Auxiliary or supporting facilities

The packages in this section contain subprograms that are expected to be applied in several areas of mathematical software implementation. The existence of such facilities is needed for the portable implementation of mathematical modules like the elementary functions.

a) Basic auxiliary functions

The following package contains templates (*generic subprogram declarations*) for obtaining the *maximum* or *minimum* of two values, for transferring the *sign* of a given first value to a given second value of any integer or floating-point type, for *interchanging* the values of two objects of the same type, and for establish-

ing whether a given value of an integer type is *odd* or *even*.

```
package BASIC_AUX_FUNCTIONS is

   generic
      type ORDERED_TYPE is private;
      with function "<"    (A, B : ORDERED_TYPE) return BOOLEAN is <>;
   function GENERIC_MIN              (A, B : ORDERED_TYPE) return ORDERED_TYPE;

   generic
      type ORDERED_TYPE is private;
      with function "<"    (A, B : ORDERED_TYPE) return BOOLEAN is <>;
   function GENERIC_MAX              (A, B : ORDERED_TYPE) return ORDERED_TYPE;

   generic
      type INTEGER_TYPE  is range <>;
   function GENERIC_SIGN_INTEGERS (S    : INTEGER_TYPE;
                                   V    : INTEGER_TYPE := 1) return INTEGER_TYPE;

   generic
      type FLOAT_TYPE    is digits <>;
   function GENERIC_SIGN_FLOATS   (S    : FLOAT_TYPE;
                                   V    : FLOAT_TYPE := 1.0) return FLOAT_TYPE;

   generic
      type FIXED_TYPE    is delta <>;
   function GENERIC_SIGN_FIXEDS   (S    : FIXED_TYPE;
                                   V    : FIXED_TYPE := 1.0) return FIXED_TYPE;

   generic
      type ITEM_TYPE     is private;
   procedure GENERIC_SWAP            (A, B : in out ITEM_TYPE);

   generic
      type INTEGER_TYPE  is range <>;
   function GENERIC_ODD              (A    : INTEGER_TYPE) return BOOLEAN;

end BASIC_AUX_FUNCTIONS;
```

b) Primitive functions

The following generic package contains modules for manipulating the fraction part and the exponent part of the (unbiased) floating-point machine number representation of values of a given floating-point type, together with other facilities related to the spacing and rounding of floating-point values.

```
generic
   type FLOAT_TYPE        is digits <>;
   type EXPONENT_TYPE     is range <>;
package GENERIC_PRIMITIVE_FUNCTIONS is

   RADIX : constant INTEGER := FLOAT_TYPE'MACHINE_RADIX;

   function  EXPONENT                  (X      : FLOAT_TYPE) return EXPONENT_TYPE;

   function  SCALE                     (X      : FLOAT_TYPE;
                                        EXP    : EXPONENT_TYPE) return FLOAT_TYPE;

   function  FRACTION                  (X      : FLOAT_TYPE) return FLOAT_TYPE;

   procedure SPLIT_FLOAT_VALUE         (X      : in FLOAT_TYPE;
                                        FRACT  : out FLOAT_TYPE;
                                        EXP    : out EXPONENT_TYPE);

   function  SYNTHESIZE                (X      : FLOAT_TYPE;
                                        EXP    : EXPONENT_TYPE) return FLOAT_TYPE;

   function  NEXT_AFTER                (X, Y   : FLOAT_TYPE) return FLOAT_TYPE;

   function  ABS_SPACING               (X      : FLOAT_TYPE) return FLOAT_TYPE;

   function  RECIPROCAL_REL_SPACING    (X      : FLOAT_TYPE) return FLOAT_TYPE;

   function  REMAINDER                 (X      : FLOAT_TYPE; Y : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function  REAL_INT                  (X      : FLOAT_TYPE) return FLOAT_TYPE;

   REPRESENTATION_ERROR : exception;

end GENERIC_PRIMITIVE_FUNCTIONS;
```

2.1.3. Composite mathematical types and operations

a) Standard types and operations

The packages REAL_TYPES, CARTESIAN_COMPLEX_TYPES and POLAR_COMPLEX_TYPES contain definitions of standard types. The standard types provided are (real and complex) scalars, vectors and matrices.

The base is a standard (sub)type called REAL which is a floating-point type satisfying the following minimal requirements upon its precision. When only one floating-point type is supported (this should be the pre-defined type FLOAT), that is chosen. Where there is a choice of pre-defined floating-point precisions then

the standard type REAL is that which corresponds most closely to (but at least provides) the single precision (32-bit precision) of the IEEE standard 754 (1985). If a second supported type is a longer type, LONG_REAL in a similar package LONG_REAL_TYPES should correspond to the pre-defined type whose representation is closest to double precision (64-bits) of the IEEE standard 754.

Where an analogous package LONG_REAL_TYPES (based on LONG_REAL) can be provided, it is recommended that this package and the derived complex packages (called LONG_CARTESIAN_COMPLEX_TYPES and LONG_POLAR_COMPLEX_TYPES) are provided with the defined standard type names (for example REAL, COMPLEX, COMPLEX_VECTOR, REAL_MATRIX, etc.) prefixed by LONG_ (except for the 'renaming' subtype declarations for VECTOR and MATRIX). Moreover, these packages should contain functions LENGTHEN and SHORTEN for conversions between REAL and LONG_REAL, or between COMPLEX and LONG_COMPLEX.

```
package REAL_TYPES is

    subtype REAL        is a floating-point type;
    type REAL_VECTOR is array (INTEGER range <>) of REAL;
    type REAL_MATRIX is array (INTEGER range <>, INTEGER range <>) of REAL;
    subtype VECTOR      is REAL_VECTOR;
    subtype MATRIX      is REAL_MATRIX;

    -- basic auxiliary functions

    function MIN     (X, Y : REAL) return REAL;
    function MAX     (X, Y : REAL) return REAL;
    function SIGN    (S    : REAL;
                      V    : REAL := 1.0) return REAL;
    procedure SWAP (X, Y : in out REAL);

    pragma INLINE   ( MIN, MAX, SIGN, SWAP );

end REAL_TYPES;
```

Cartesian and polar representations for complex types are provided in the packages CARTESIAN_COMPLEX_TYPES and POLAR_COMPLEX_TYPES. These packages also contain selection and composition operators so that the types may be imported into other (generic) packages as private types, and operators for complex arithmetic.

```
with REAL_TYPES; use REAL_TYPES;
package CARTESIAN_COMPLEX_TYPES is

-- cartesian types

    type COMPLEX is private;
    type COMPLEX_VECTOR is array (INTEGER range <>) of COMPLEX;
    type COMPLEX_MATRIX is array (INTEGER range <>, INTEGER range <>) of COMPLEX;

-- selection, conversion and composition operations
    function COMPOSE_CARTESIAN (REAL_PART : REAL;
```

```
                                    IMAG_PART : REAL := 0.0) return COMPLEX;
    function COMPOSE_POLAR          (MODULUS  : REAL;
                                     ARGUMENT : REAL := 0.0) return COMPLEX;
    function WIDEN                  (X        : REAL) return COMPLEX;

    function REAL_PART (X    : COMPLEX) return REAL;
    function IMAG_PART (X    : COMPLEX) return REAL;
    function RE        (X    : COMPLEX) return REAL renames REAL_PART;
    function IM        (X    : COMPLEX) return REAL renames IMAG_PART;

    function ARGUMENT  (X    : COMPLEX) return REAL;
    function MODULUS   (X    : COMPLEX) return REAL;
    function "abs"     (X    : COMPLEX) return REAL renames MODULUS;

-- cartesian operations

    -- unary operations
    function "+"       (X    : COMPLEX) return COMPLEX;
    function "-"       (X    : COMPLEX) return COMPLEX;
    function CONJ      (X    : COMPLEX) return COMPLEX;

    -- COMPLEX arithmetic operations
    function "+"       (X, Y : COMPLEX) return COMPLEX;
    function "-"       (X, Y : COMPLEX) return COMPLEX;
    function "*"       (X, Y : COMPLEX) return COMPLEX;
    function "/"       (X, Y : COMPLEX) return COMPLEX;
    function "**"      (X    : COMPLEX; N : INTEGER) return COMPLEX;

    -- mixed REAL/COMPLEX arithmetic operations
    function "+"       (X : REAL;    Y : COMPLEX) return COMPLEX;
    function "+"       (X : COMPLEX; Y : REAL   ) return COMPLEX;
    function "-"       (X : REAL;    Y : COMPLEX) return COMPLEX;
    function "-"       (X : COMPLEX; Y : REAL   ) return COMPLEX;
    function "*"       (X : REAL;    Y : COMPLEX) return COMPLEX;
    function "*"       (X : COMPLEX; Y : REAL   ) return COMPLEX;
    function "/"       (X : REAL;    Y : COMPLEX) return COMPLEX;
    function "/"       (X : COMPLEX; Y : REAL   ) return COMPLEX;

-- basic auxiliary utilities
    procedure SWAP     (X, Y : in out COMPLEX);

-- pragma
    pragma INLINE      ("+", "-", "*", "/", "**", "abs", CONJ, SWAP,
                        COMPOSE_CARTESIAN, COMPOSE_POLAR, WIDEN,
                        REAL_PART, IMAG_PART, RE, IM, ARGUMENT, MODULUS);

private
```

```
    type COMPLEX is
                record
                    RE, IM : REAL := 0.0;
                end record;

end CARTESIAN_COMPLEX_TYPES;


with REAL_TYPES; use REAL_TYPES;
package POLAR_COMPLEX_TYPES is

-- polar types

    type COMPLEX is private;
    type COMPLEX_VECTOR is array (INTEGER range <>) of COMPLEX;
    type COMPLEX_MATRIX is array (INTEGER range <>, INTEGER range <>) of COMPLEX;

-- selection, conversion and composition operations
    function COMPOSE_CARTESIAN (REAL_PART : REAL;
                                IMAG_PART : REAL := 0.0) return COMPLEX;
    function COMPOSE_POLAR     (MODULUS   : REAL;
                                ARGUMENT  : REAL := 0.0) return COMPLEX;
    function WIDEN             (X         : REAL) return COMPLEX;

    function REAL_PART (X    : COMPLEX) return REAL;
    function IMAG_PART (X    : COMPLEX) return REAL;
    function RE        (X    : COMPLEX) return REAL renames REAL_PART;
    function IM        (X    : COMPLEX) return REAL renames IMAG_PART;

    function ARGUMENT  (X    : COMPLEX) return REAL;
    function MODULUS   (X    : COMPLEX) return REAL;
    function "abs"     (X    : COMPLEX) return REAL renames MODULUS;

-- polar operations

    -- unary operations
    function "+"       (X    : COMPLEX) return COMPLEX;
    function "-"       (X    : COMPLEX) return COMPLEX;
    function CONJ      (X    : COMPLEX) return COMPLEX;

    -- COMPLEX arithmetic operations
    function "+"       (X, Y : COMPLEX) return COMPLEX;
    function "-"       (X, Y : COMPLEX) return COMPLEX;
    function "*"       (X, Y : COMPLEX) return COMPLEX;
    function "/"       (X, Y : COMPLEX) return COMPLEX;
    function "**"      (X    : COMPLEX; N : INTEGER) return COMPLEX;
```

```
   -- mixed REAL/COMPLEX arithmetic operations
   function "+"      (X : REAL;    Y : COMPLEX) return COMPLEX;
   function "+"      (X : COMPLEX; Y : REAL   ) return COMPLEX;
   function "-"      (X : REAL;    Y : COMPLEX) return COMPLEX;
   function "-"      (X : COMPLEX; Y : REAL   ) return COMPLEX;
   function "*"      (X : REAL;    Y : COMPLEX) return COMPLEX;
   function "*"      (X : COMPLEX; Y : REAL   ) return COMPLEX;
   function "/"      (X : REAL;    Y : COMPLEX) return COMPLEX;
   function "/"      (X : COMPLEX; Y : REAL   ) return COMPLEX;

-- basic auxiliary utilities
   procedure SWAP    (X, Y : in out COMPLEX);

-- pragma
   pragma INLINE     ("+", "-", "*", "/", "**", "abs", CONJ, SWAP,
                      COMPOSE_CARTESIAN, COMPOSE_POLAR, WIDEN,
                      REAL_PART, IMAG_PART, RE, IM, ARGUMENT, MODULUS);

private

   type COMPLEX is
                record
                   R, THETA : REAL := 0.0;
                end record;

end POLAR_COMPLEX_TYPES;
```

b) Generic complex functions

A single generic package is provided for COMPLEX functions which can be instantiated for either cartesian or polar COMPLEX types, but this assumes that the composition operators satisfy any conditions imposed on their results by the particular COMPLEX representation.

```
with MATHEMATICAL_EXCEPTIONS;
generic
   type REAL_TYPE is digits <>;
   type COMPLEX_TYPE is private;
   -- selection, conversion and composition operations
   with function COMPOSE_CARTESIAN (REAL_PART : REAL_TYPE;
                                    IMAG_PART : REAL_TYPE := 0.0) return REAL_TYPE is <>;
   with function COMPOSE_POLAR     (MODULUS   : REAL_TYPE;
                                    ARGUMENT  : REAL_TYPE := 0.0) return REAL_TYPE is <>;
```

```
    with function REAL_PART (X : COMPLEX_TYPE) return REAL_TYPE is <>;
    with function IMAG_PART (X : COMPLEX_TYPE) return REAL_TYPE is <>;
    with function ARGUMENT  (X : COMPLEX_TYPE) return REAL_TYPE is <>;
    with function MODULUS   (X : COMPLEX_TYPE) return REAL_TYPE is <>;
    -- mathematical functions
    with function SQRT      (X : REAL_TYPE) return REAL_TYPE is <>;
    with function LOG       (X : REAL_TYPE) return REAL_TYPE is <>;
    with function EXP       (X : REAL_TYPE) return REAL_TYPE is <>;
    with function SIN       (X : REAL_TYPE) return REAL_TYPE is <>;
    with function COS       (X : REAL_TYPE) return REAL_TYPE is <>;
    with function SINH      (X : REAL_TYPE) return REAL_TYPE is <>;
    with function COSH      (X : REAL_TYPE) return REAL_TYPE is <>;

package GENERIC_COMPLEX_FUNCTIONS is

    function SQRT (X : COMPLEX_TYPE) return COMPLEX_TYPE;
    function LOG  (X : COMPLEX_TYPE) return COMPLEX_TYPE;
    function EXP  (X : COMPLEX_TYPE) return COMPLEX_TYPE;
    function SIN  (X : COMPLEX_TYPE) return COMPLEX_TYPE;
    function COS  (X : COMPLEX_TYPE) return COMPLEX_TYPE;

    -- Declare exception to be raised for implementation-independent range constraints
    ARGUMENT_ERROR : exception renames MATHEMATICAL_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_COMPLEX_FUNCTIONS;
```

Standard instantiations for the generic complex function packages should be provided for the precision REAL (package REAL_TYPES) and a related type COMPLEX. Such instantiations can be given as follows:

```
-- For cartesian representation of COMPLEX
with REAL_TYPES, CARTESIAN_COMPLEX_TYPES, ELEMENTARY_FUNCTIONS_REAL;
use REAL_TYPES, CARTESIAN_COMPLEX_TYPES, ELEMENTARY_FUNCTIONS_REAL;
with GENERIC_COMPLEX_FUNCTIONS;
package CARTESIAN_COMPLEX_FUNCTIONS is new GENERIC_COMPLEX_FUNCTIONS (REAL, COMPLEX);

-- For polar representation of COMPLEX
with REAL_TYPES, POLAR_COMPLEX_TYPES, ELEMENTARY_FUNCTIONS_REAL;
use REAL_TYPES, POLAR_COMPLEX_TYPES, ELEMENTARY_FUNCTIONS_REAL;
with GENERIC_COMPLEX_FUNCTIONS;
package POLAR_COMPLEX_FUNCTIONS is new GENERIC_COMPLEX_FUNCTIONS (REAL, COMPLEX);
```

Here the employed package ELEMENTARY_FUNCTIONS_REAL is assumed to be an instance of GENERIC_ELEMENTARY_FUNCTIONS (see Section 2.1.1.a) for the floating-point type REAL_TYPES.REAL.

Where an implementation supports additional pre-defined floating-point types such that a package LONG_REAL_TYPES is also provided, it is recommended that analogous instantiations LONG_CARTESIAN_COMPLEX_FUNCTIONS and / or LONG_POLAR_COMPLEX_FUNCTIONS (for the precision LONG_REAL and the type LONG_COMPLEX) should also be provided.

2.2. Semantic specifications

In all cases care has been taken that every function is not only applicable for (one of) the pre-defined floating-point types but also for user-defined floating-point types made by a type definition like

type REAL_N **is digits** N { **range** A .. B };

This is solved by presenting packages (or in some cases: subprograms) which are generic with the generic parameter FLOAT_TYPE. Instantiation will then give all facilities for one real type or for several real types if this is desired. However, especially in the case of the elementary functions the use of floating-point types with an additional range constraint for actual generic parameters should always be avoided, since this might obstruct the performance of mathematical algorithms implemented for approximating the elementary functions.

The range of applicability of some generic modules extends beyond the area of floating-point and integral types, viz. when the generic type parameter is specified as **private**.

For a few modules the semantic specification given is simply an Ada *subprogram body*.

2.2.1. Basic mathematical packages

a) Elementary functions

For the elementary functions package template the generic parameter FLOAT_TYPE can be a user-defined type (or subtype) as explained in Section 2.2. However, the package would be virtually useless if arbitrary additional *range constraints* were applied. The user should note that any range constraint on the actual type substituted for FLOAT_TYPE may invalidate the mathematical functions implementation. Users who do want to impose range constraints for their own computations are advised to declare the subtypes with restricted ranges after the instantiation of GENERIC_ELEMENTARY_FUNCTIONS.

The following table specifies the requirements for the parameter ranges of all and the result ranges of some of the functions. The range constraints for the function arguments (in general the function *domains*) must be observed by the user and upon violation the implementation should raise the exception ARGUMENT_ERROR. All specified range constraints are *implementation-independent* (the specifications given here review the *mathematical* requirements). Other restrictions upon the arguments which depend upon the *range* of the *floating-point type employed* can be specified, and in those cases violation might cause raising of the pre-defined exception NUMERIC_ERROR. The result-range specifications for some inverse functions serve to identify the dominant range of multiple-valued functions.

Function	Argument and range
SQRT	X ≥ 0.0
LOG	X > 0.0, BASE > 0.0 **and** BASE /= 1.0
EXP	X unrestricted
"**"	X ≥ 0.0, Y unrestricted when X > 0.0, Y > 0.0 when X = 0.0
all trigonometric functions	CYCLE /= 0.0
SIN	X unrestricted
COS	X unrestricted
TAN	X unrestricted
COT	X /= 0.0
ARCSIN	**abs** X ≤ 1.0, $-\pi/2.0 \le$ ARCSIN(X) $\le \pi/2.0$, $-$CYCLE/4.0 ≤ ARCSIN(X, CYCLE) ≤ CYCLE/4.0
ARCCOS	**abs** X ≤ 1.0, $0.0 \le$ ARCCOS(X) $\le \pi$, 0.0 ≤ ARCCOS(X, CYCLE) ≤ CYCLE/2.0
ARCTAN	**not** (X = 0.0 **and** Y = 0.0), $-\pi/2.0 \le$ ARCTAN(Y) $\le \pi/2.0$, $-\pi <$ ARCTAN(Y, X) $\le \pi$, $-$CYCLE/2.0 < ARCTAN(Y, X, CYCLE) ≤ CYCLE/2.0
ARCCOT	**not** (X = 0.0 **and** Y = 0.0), $0.0 \le$ ARCCOT(X) $\le \pi$, $-\pi <$ ARCCOT(X, Y) $\le \pi$, $-$CYCLE/2.0 < ARCCOT(X, Y, CYCLE) ≤ CYCLE/2.0
SINH	X unrestricted
COSH	X unrestricted
TANH	X unrestricted
COTH	X /= 0.0
ARCSINH	X unrestricted
ARCCOSH	X ≥ 1.0, ARCCOSH(X) ≥ 0.0
ARCTANH	**abs** X < 1.0
ARCCOTH	**abs** X > 1.0

The elementary functions have their usual mathematical meaning, which should be sufficient information. The formal parameter name for floating-point arguments is X, or X and Y for two-argument functions. However, for the function ARCTAN the formal parameter names are Y and X (in this order), since a legitimate use of the *arctan* function is to compute the *argument* (i.e. the angle) of a cartesian point (X1, Y1). The appropriate call is then: ARCTAN(Y1, X1), or better: ARCTAN(X => X1, Y => Y1).

For LOG and each of the trigonometric functions two specifications are given, one (the general form) with a last parameter for choosing the *logarithm base* (for LOG), or the *cycle* for the argument (or result) of the trigonometric function, and one specification without this last parameter assuming the traditional values of *base* and *cycle*. Provision of a single specification for each function (with a default expression for the last parameter) (see *Symm et al.*, 1984) is avoided, the present specifications allowing more accurate implementations of the default cases.

- For the trigonometric functions the traditional, one-parameter specifications (two-parameter specifications for ARCTAN and ARCCOT) provide the *radian* versions of the functions. All functions are overloaded, however, with versions having an additional last parameter called CYCLE of type FLOAT_TYPE. This allows scaling to degrees or to multiples of 2π to be chosen for the arguments of SIN, COS, TAN and COT, and for the function results of ARCSIN, ARCCOS, ARCTAN and ARCCOT. For X in degrees the *sin* of X is obtained by SIN(X, 360.0). The *sin* of 2πX is obtained by SIN(X, 1.0).
  Note that for computing *arctan*(Y1) with Y1 in degrees Ada's concept of *named parameter association* must be used:
  ARCTAN(Y => Y1, CYCLE => 360.0)

- A similar design is given for the *log* function with arbitrary base. The one-parameter case LOG(X) gives the natural logarithm *ln*.

It should be guaranteed that all elementary functions will not overflow or underflow during computations if the mathematical result is within the range of safe numbers of the provided floating-point type. In particular, if one of the pre-defined exceptions NUMERIC_ERROR or CONSTRAINT_ERROR could be raised in this case, it should not propagate beyond the function body.

If a result is outside the range of safe numbers of the floating-point type, the called function may raise NUMERIC_ERROR.

For the situation that a mathematical boundary of a function range is exactly representable (for any accuracy definition), it should be guaranteed that returned values do not exceed such boundaries. This applies to

SQRT : SQRT(X) >= 0.0,

EXP and "**" (for real exponent) : EXP(X) >= 0.0,

SIN, COS : -1.0 <= {SIN(X), COS(X)} <= 1.0,

COTH : COTH(X) >= 1.0.

The effect is that the implementation of a mathematical relationship like ARCSIN (SIN (X)) will never raise ARGUMENT_ERROR.

The inverse trigonometric functions are not mentioned in this list, since most of the boundaries of function ranges depend on CYCLE, and this can always be an approximation of a not exactly representable real value. For the special cases where CYCLE = 2π some of the guarantees are hardly implementable.

The implementer should provide further information about the accuracy that can be expected.

b) Exception for the mathematical functions package

Different instances of GENERIC_ELEMENTARY_FUNCTIONS do not create new exceptions, but only new names for the same exception declared by MATHEMATICAL_EXCEPTIONS.

c) Mathematical constants

The mathematical constants have their obvious meaning. The accuracy of the literals given should be appropriate for all supported *real* types. Therefore, on systems supporting real types of higher precision than is expected in the present package specification re-editing of the package source code prior to installation is required.

d) Random numbers

The generic package GENERIC_RANDOM_GENERATOR is a template for uniform pseudo-random number generators. It contains the declaration of a type RANDOM_SEED which is **limited private** to guarantee the integrity of the seed. The user has to declare objects of this type, for use as parameters of the following procedures. Implementations should guarantee that seeds will always be initialised (randomly).

The procedure RANDOM generates a pseudo random result uniformly distributed in the (open) range 0 < NUMBER < 1 ; the implementation should protect against the simultaneous updating of the supplied seed.

The procedure SET_RANDOM_SEED can be used to initialize a seed for generating a repeatable sequence. A value of N should always initialize the seed to the same hidden value. The implementation should protect against simultaneous updating of the seed.

The procedure COPY_RANDOM_SEED can be used for copying seed values. The implementation should protect against simultaneous accessing of the actual parameters.

The function RANDOM generates a pseudo random result uniformly distributed in the (open) range 0 < NUMBER < 1 . The implementation does NOT protect against the simultaneous updating of the supplied seed. The function cannot be used to obtain reproducible sequences.

The procedure SAVE_RANDOM_SEED can be used for saving seed values, such that a sufficient (internal) representation is stored in an array variable of type STORE_TYPE, which is an array type of STORE_SIZE INTEGERs declared in the package. Here, STORE_SIZE refers to the second generic formal parameter, an in parameter of type INTEGER. The instantiation of GENERIC_RANDOM_GENERATOR must be made with a suitable value for STORE_SIZE.
The procedure RESTORE_RANDOM_SEED reconstructs RANDOM_SEED values from values stored in an STORE_TYPE variable. For reconstructing RANDOM_SEED values on machines with (considerably) differing values of INTEGER´LAST, re-formatting of the STORE_TYPE values must be performed by the user.
The implementation should protect against simultaneous accessing of the actual RANDOM_SEED parameters.

2.2.2. Auxiliary and supporting facilities

a) Basic auxiliary functions

The package BASIC_AUX_FUNCTIONS provides templates for *min* and *max* functions for any type for which an ordering has been declared, *sign* functions for any integer or floating-point type (the function result is the absolute value of the second parameter with the sign of the first parameter), a *swap* procedure for inter-

changing the values of two variables of the same type, and an *odd* function for any integer type.

For example, a *min* function for a given type ANY_TYPE can be obtained (provided that the "<" operator is defined for values of ANY_TYPE) through:

```
function MIN is new BASIC_AUX_FUNCTIONS.GENERIC_MIN ( ANY_TYPE );
A := MIN ( B, C ); -- with A, B, C of type ANY_TYPE
```

A package body will suffice here to define the meaning of the auxiliary subprograms.

```
package body BASIC_AUX_FUNCTIONS is

   function GENERIC_MIN (A, B : ORDERED_TYPE) return ORDERED_TYPE is
   begin
      if A < B then
         return A;
      else
         return B;
      end if;
   end GENERIC_MIN;

   function GENERIC_MAX (A, B : ORDERED_TYPE) return ORDERED_TYPE is
   begin
      if A < B then
         return B;
      else
         return A;
      end if;
   end GENERIC_MAX;

   function GENERIC_SIGN_INTEGERS (S : INTEGER_TYPE;
         V : INTEGER_TYPE := 1) return INTEGER_TYPE is
   begin
      if S < 0 then
         return - abs V;
      else
         return abs V;
      end if;
   end GENERIC_SIGN_INTEGERS;

   function GENERIC_SIGN_FLOATS (S : FLOAT_TYPE;
         V : FLOAT_TYPE := 1.0) return FLOAT_TYPE is
   begin
      if S < 0.0 then
         return - abs V;
      else
         return abs V;
```

```
        end if;
    end GENERIC_SIGN_FLOATS;

    function GENERIC_SIGN_FIXEDS (S : FIXED_TYPE;
            V : FIXED_TYPE := 1.0) return FIXED_TYPE is
    begin
        if S < 0.0 then
            return – abs V;
        else
            return abs V;
        end if;
    end GENERIC_SIGN_FIXEDS;

    procedure GENERIC_SWAP (A, B : in out ITEM_TYPE) is
        T : constant ITEM_TYPE := A;
    begin
        A := B;
        B := T;
    end GENERIC_SWAP;

    function GENERIC_ODD (A : INTEGER_TYPE) return BOOLEAN is
    begin
        return INTEGER_TYPE 'POS(A) mod 2 = 1;
    end GENERIC_ODD;

end BASIC_AUX_FUNCTIONS;
```

b) Primitive functions

The functions given here manipulate machine-representations of floating-point values. Their meaning is defined for machine numbers, not model numbers only, and the results depend on the representation of floating-point numbers, in particular the machine radix.

We assume that every nonzero floating-point number can be uniquely represented in the form

$$X = \pm f * B^e, \quad 1/B \leq f < 1,$$

where B is the radix for the machine representation, f is the fraction or significand, and e is the signed exponent. Floating-point zero is represented with zero fraction and zero exponent. Then the definitions are as follows. In every case the result is as defined, provided only that the result is representable as a machine number; otherwise, an appropriate exception is raised.

EXPONENT(X) returns the unbiased, signed integer exponent e of X. Specification of this function makes the implicit assumption that the exponent range is within the range of type INTEGER. INTEGER of 16 bits is adequate to represent the 15-bit exponent recommended as a minimum for double-extended precision in the ANSI/IEEE Standard (1985), for example.

Note that EXPONENT(0.0) = 0.0.

SCALE(X, EXP) returns $X \times B^{EXP}$ without computing B^{EXP}.

FRACTION(X) returns the signed fraction f of the machine number X.
It can be computed as SCALE(X, –EXPONENT(X)).
Note that FRACTION(0.0) = 0.0.

SPLIT_FLOAT_VALUE(X, FRACT, EXP) combines the functionality of EXPONENT and FRACTION to return both the fraction f and the exponent e for the machine number X.

SYNTHESIZE(X, EXP) is defined as SCALE(X, EXP–EXPONENT(X)) for X /= 0.0. For X = 0.0 the result will be 0.0.
The parameter values in a call of SYNTHESIZE may be such that they do not represent a machine number. In that case the defined exception REPRESENTATION_ERROR should be raised instead.

NEXT_AFTER(X, Y) returns the next representable neighbor of X in the direction of Y. If X = Y, the result is X.

ABS_SPACING(X) returns the absolute spacing in the neighborhood of the machine number X. This value can be determined, for example, by the computation
MAX(X – NEXT_AFTER(X, XMIN), NEXT_AFTER(X, XMAX) – X),
where XMAX is the largest representable machine number, and XMIN is the largest representable negative machine number.
Note that for X extremely small in magnitude, the result may underflow.

RECIPROCAL_REL_SPACING(X) returns the absolute value of X / ABS_SPACING(X), with the obvious underflow and overflow problems associated with extremely small or extremely large X.

REMAINDER is defined by the mathematical relation $REMAINDER = X - Y \times N$, where N is the integer nearest the exact value X/Y; whenever $|N - X/Y| = 1/2$, then N is even.
Requirement on second parameter: Y /= 0.0.

REAL_INT delivers the integer part of the real value X as a real, without intermediate conversion to INTEGER which might cause overflow. Rounding takes place to the nearest integral value, but a value is returned of the same floating-point type as the parameter.
Requirement on parameter: X unrestricted.

We note that several of these functions are redundant and could be removed. The functions EXPONENT and SCALE are fundamental, but the functions FRACTION, SPLIT_FLOAT_VALUE, and SYNTHESIZE are easily constructed from the first two. Interfacing to corresponding system-provided facilities is an obvious implementation of some of these functions.

NEXT_AFTER(X, Y) is another fundamental function that can be used to construct others.

We define the functions ABS_SPACING and RECIPROCAL_REL_SPACING in terms of machine numbers, but analogous (and possibly more useful) functions might be defined in terms of model numbers as suggested in the Guidelines.

2.2.3. Composite mathematical types and operations

a) Standard types and operations

The package REAL_TYPES provides a floating-point type REAL, unconstrained array types VECTOR and MATRIX constructed from this REAL and four basic auxiliary subprograms for REALs. If a similar package LONG_REAL_TYPES can be provided, it has the same provisions for LONG_REAL.

The two packages CARTESIAN_COMPLEX_TYPES and POLAR_COMPLEX_TYPES provide private type definitions, selection and composition operations and arithmetic operations so that complex types can be imported into other generic packages as generic parameters. The declaration parts of the two packages are deliberately identical.

For *polar complex* types (i.e. POLAR_COMPLEX_TYPES.COMPLEX and LONG_POLAR_COMPLEX_TYPES.LONG_COMPLEX) there should be no range constraints on the type of the components R and THETA (for *modulus* and *argument* of a complex number) to avoid violation of such constraints by intermediate computations (in arithmetic complex operations provided by the *polar* packages). However, it is assumed that the representation of a *polar complex* value should always be unique. Therefore the argument THETA of a polar result will always be transformed to the range $-$ PI <= THETA <= PI , and the radius (or modulus) R of a polar result will always be transformed to the range 0.0 <= R. Here, the representable REAL value $-$ PI is always different from PI $- 2\pi$ (not representable). It can therefore be expected that the pre-defined equality operator gives the expected mathematical result.

No exceptions need be raised by the packages since it is expected that the COMPOSE_ functions as well as the arithmetic operators check that the resulting *polar complex* value is in the above ranges for the R and THETA components and if not transform those components. The pre-defined exception NUMERIC_ERROR may be raised if (the components of) the results of any operator are outside the range of safe numbers.

It should be guaranteed that all complex operators will not overflow or underflow during computations if (the components of) the mathematical results are within the range of safe numbers. In particular, if the pre-defined exception NUMERIC_ERROR could be raised in this case, it should not propagate beyond the operator body.

b) Generic complex functions

The generic package GENERIC_COMPLEX_FUNCTIONS provides a template for the elementary complex functions in terms of a generic floating-point type parameter and a related generic type parameter for a *complex* type. The generic formal parameters have been chosen such that instances can be made for all supported accuracies and also for both representations of COMPLEX.

If the first generic *actual* parameter is REAL from REAL_TYPES, then all other required generic actual parameters (in particular COMPLEX) can be obtained from either CARTESIAN_COMPLEX_TYPES or POLAR_COMPLEX_TYPES, and from an appropriate package ELEMENTARY_FUNCTIONS_REAL that can quite well be an instance of GENERIC_ELEMENTARY_FUNCTIONS (see Section 2.1.1.a) with REAL_TYPES.REAL as the generic actual parameter (cf. the instantiations given at the end of Section 2.1.3.b). In that case, for all generic subprogram parameters an explicit association can be omitted, in which case the default is used. To preserve integrity, it is recommended to use the defaults.

If the first generic actual parameter is LONG_REAL from LONG_REAL_TYPES, then all other required generic actual parameters (in particular LONG_COMPLEX) can be obtained from either LONG_CARTESIAN_COMPLEX_TYPES or LONG_POLAR_COMPLEX_TYPES, and from an appropriate package ELEMENTARY_FUNCTIONS_LONG_REAL that can again be an instance of GENERIC_ELEMENTARY_FUNCTIONS, with LONG_REAL as the generic actual parameter.

The exception ARGUMENT_ERROR is declared in the package MATHEMATICAL_EXCEPTIONS (see Section 2.1.1.b). It should be raised only for calls of the function LOG with **abs** X = 0.0. The pre-defined exception NUMERIC_ERROR may be raised if (the components of) the results of any function are outside the range of safe numbers.

It should be guaranteed that all complex functions will not overflow or underflow during computations if (the components of) the mathematical results are within the range of safe numbers. In particular, if the pre-defined exception NUMERIC_ERROR could be raised in this case, it should not propagate beyond the function body.

ACKNOWLEDGEMENTS

Starting the work on this proposal was made possible by financial support of the Commission of the European Communities and the US Department of the Navy.

Graham S. Hodgson (NAG, Oxford) contributed the Ada specifications given in the Sections titled "**Random numbers**" and "**Composite mathematical types and operations**". W.J. Cody (Argonne National Laboratory) supplied the semantic specifications for the primitive functions. Some other specifications were derived from results of the EC-supported MultiAnnual Programme project (MAP 750) *Pilot implementations of basic modules for large portable numerical libraries in Ada* (see, e.g., *Kok* 1987).

The following individuals contributed to this Proposal through their comments and by taking part in discussions in workshops of the SIGAda Numerics Users Committee Working Group organised by Gil Myers, Ada-Europe Numerics Working Group meetings, working group meetings at Argonne National Laboratory and meetings of the MAP 750 project team. Inclusion in the list of names does not imply that the named individual or his or her company approves of the present state of this Proposal or of any of its details.

J.L. Adda (SEMA.METRA, Montrouge), C.A. Addison (CMI, Bergen), J.G.P. Barnes (Alsys Ltd), M. Bergman (CWI, Amsterdam), L. Bonami (SIEMENS a.g., München), J. Carroll (NIHE, Dublin), D. Clarkson (IMSL), W.J. Cody (Argonne National Laboratory (ANL), Argonne), P. Cohen (AJPO), T.J. Dekker (Universiteit van Amsterdam), L.M. Delves (University of Liverpool), B. Doman (University of Liverpool), K.W. Dritz (ANL, Argonne), E. Edberg (NDRI, Linköping), B. Ford (NAG, Oxford), F. Forest (Informatique Internationale (II), Rungis), T.J. Froggatt (Systems Designers, Camberley), P. Gendre (CISI, Rungis), G.S. Hodgson (NAG, Oxford), Th. Kalamboukis (Athens SE&BS), K. Köhler (NDRI, Linköping), A.R. Klumpp (JPL, Pasadena), P. Leroy (ALSYS s.a., Paris), R.F. Mathis (Fairfax), H. Mumm (NOSC, San Diego), G. Myers (NOSC, San Diego), C. Pursglove (University of Liverpool), K. Rehmer (Magnavox, Ft Wayne), J.P. Rosen (ENST, Paris), K. Schmidt (Universität Marburg), L. Shanbeck (QTC), B.T. Smith (ANL, Argonne), J. Squire (Westinghouse, Baltimore), L. Steenman-Clark (NAG, Oxford), G.T. Symm (NPL, Teddington), C. Ullrich (Universität Karlsruhe), G. Völksen (Universität Marburg), R.P. Wehrum (SIEMENS, München), B.A. Wichmann (NPL, Teddington), D.T. Winter (CWI, Amsterdam).

REFERENCES

ANSI/IEEE Std 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*, July 1985.

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

Ford, B., Kok, J. and Rogers, M.W. (eds.) *Scientific Ada*, Cambridge University Press, 1986.

Kok, J. *Design and implementation of elementary functions in Ada*, CWI Report NM-R8710, April 1987.

Kok, J. and Symm, G.T. A proposal for standard basic functions in Ada, *Ada Letters*, Vol. IV.3, Nov/Dec 1984, 44-52.

Symm, G.T., Wichmann, B.A., Kok, J., and Winter, D.T. *Guidelines for the design of large modular scientific libraries in Ada*, NPL Report DITC 37/84 and CWI Report NM-N8401, March 1984 (also in: *Ford et al.*, 1986).

Whitaker, W.A. and Eicholtz, T.C. *An Ada implementation of the Cody-Waite "Software manual for the elementary functions"*, US Air Force, 1982.