INTERNATIONAL STANDARD

ISO/IEC 13813

First edition
1998-08-01

Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)

Technologies de l'information — Langages de programmation — Paquetages génériques de déclarations de types réel et complexe et opérations de base pour Ada (y compris les types vecteur et matrice)

Reference number
ISO/IEC 13813:1998(E)

Contents

Annexes

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13813 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces.*

Annexes A, B, C, D and E form an integral part of this International Standard. Annexes F, G and H are for information only.

Introduction

The generic packages described here are intended to provide the basic real and complex scalar, vector, and matrix operations from which portable, reusable applications can be built. This International Standard serves a broad class of applications with reasonable ease of use, while demanding implementations that are of high quality, capable of validation and also practical given the state of the art.

The specifications included in this International Standard are presented as compilable Ada specifications in annexes A, B, C, D and E with explanatory text in numbered sections in the main body of text. The explanatory text is normative, with the exception of notes (labeled as such).

The word "may," as used in this International Standard, consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could" or "might") are used to express ability, possibility, capacity or consequentiality.

Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)

1 Scope

This International Standard defines the specifications of three generic packages of scalar, vector and matrix operations called `GENERIC_COMPLEX_TYPES`, `GENERIC_REAL_ARRAYS` and `GENERIC_COMPLEX_ARRAYS`, the specification of a package of related exceptions called `ARRAY_EXCEPTIONS` and the specification of a generic package of complex input and output operations called `COMPLEX_IO`. A package body is not required for `ARRAY_EXCEPTIONS`; bodies of the other packages are not provided by this International Standard.

The specifications of non-generic packages called `COMPLEX_TYPES`, `REAL_ARRAYS` and `COMPLEX_ARRAYS` are also defined, together with those of analogous packages for other precisions. This International Standard does not provide the bodies of these packages.

This International Standard specifies certain fundamental scalar, vector and matrix arithmetic operations for real, imaginary and complex numbers. They were chosen because of their utility in various application areas; moreover, they are needed to support a generic package for complex elementary functions.

This International Standard is applicable to programming environments conforming to ISO/IEC 8652.

NOTE — This International Standard is specifically designed for applicability in programming environments conforming to ISO/IEC 8652:1987. Except for the packages and generic packages dealing with arrays, comparable facilities are specified in ISO/IEC 8652:1995; specifications for the generic array packages comforming to ISO/IEC 8652:1995 are provided in annex G.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 8652, *Information technology — Programming languages — Ada.*

ISO/IEC 11430, *Information technology — Programming languages — Generic package of elementary functions for Ada.*

ISO/IEC 11729, *Information technology — Programming languages — Generic package of primitive functions for Ada.*

ISO/IEC 13814, *Information technology — Programming languages — Generic package of complex elementary functions for Ada.*

3 Types and operations provided

The following record type, scalar type and four array types are exported by the packages provided by this International Standard:

COMPLEX	IMAGINARY
REAL_VECTOR	REAL_MATRIX
COMPLEX_VECTOR	COMPLEX_MATRIX

Type COMPLEX provides a cartesian representation of complex scalars; type IMAGINARY is provided to represent pure imaginary scalars; two composite types with elements of type REAL are provided, REAL_VECTOR and REAL_MATRIX, to represent real vectors and matrices; and two composite types with elements of type COMPLEX are provided, COMPLEX_VECTOR and COMPLEX_MATRIX, to represent complex vectors and matrices.

The following twenty-four operations are provided:

"+"	"-"	"*"	"/"
"<"	"<="	">"	">="
"**"	"abs"	CONJUGATE	TRANSPOSE
RE	IM	SET_RE	SET_IM
COMPOSE_FROM_CARTESIAN	MODULUS	ARGUMENT	COMPOSE_FROM_POLAR
UNIT_VECTOR	IDENTITY_MATRIX	GET	PUT

These are the usual mathematical operators (+, -, * and /) for real, complex and imaginary scalars, and for real and complex vectors and matrices (together with analogous componentwise operations for vectors); the relational operators (<, <=, > and >=) for imaginary scalars; the exponentiation operator (**) for complex and imaginary scalars, and for real and complex vectors; the absolute value operator (abs) for real, imaginary and complex scalars, and for real and complex vectors and matrices; the conjugate operation (CONJUGATE) for complex and imaginary scalars, and for complex vectors and matrices; the transpose operation (TRANSPOSE) for real and complex matrices; the cartesian component-part operations (RE, IM, SET_RE, SET_IM and COMPOSE_FROM_CARTESIAN) for complex scalars, vectors and matrices (and, where applicable, for imaginary scalars), for selecting component-parts and for composing from component-parts; the polar component-part operations (MODULUS, ARGUMENT and COMPOSE_FROM_POLAR) for complex scalars, vectors and matrices, for selecting component-parts and for composing from component-parts; the initializing operations (UNIT_VECTOR and IDENTITY_MATRIX) for real and complex vectors and matrices; and the input/output operations (GET and PUT) for complex scalars.

4 Instantiations

This International Standard describes generic packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS and COMPLEX_IO. Each package has a generic formal parameter, which is a generic formal floating-point type named REAL. At instantiation, this parameter determines the precision of the arithmetic.

This International Standard also describes non-generic packages COMPLEX_TYPES, REAL_ARRAYS and COMPLEX_ARRAYS, which provide the same capability as instantiations of the packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS. It is required that non-generic packages be constructed for each precision of floating-point type defined in package STANDARD.

Depending on the implementation, the user may or may not be allowed to specify a generic actual type having a range constraint (see clause 5). If allowed, such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when a scalar argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return a scalar value (or to construct a composite value with a scalar component or element) outside the user's range. Allowing the generic actual type to have a range constraint also has some implications for implementers.

5 Implementations

Portable implementations of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS and COMPLEX_IO are strongly encouraged. However, implementations are not required to be portable. In particular, an implementation of this International Standard in Ada may use pragma INTERFACE or other pragmas, unchecked conversion, machine-code insertions, or other machine-dependent techniques as desired.

An implementation is allowed to make reasonable assumptions about the environment in which it is to be used, but only when necessary in order to match algorithms to hardware characteristics in an economical manner. For example, an implementation is allowed to limit the precision it supports (by stating an assumed maximum value for SYSTEM.MAX_DIGITS), since portable implementations would not, in general, be possible otherwise. All such limits and assumptions shall be clearly documented. By convention, an implementation of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS or COMPLEX_IO is said not to conform to this International Standard in any environment in which its limits or assumptions are not satisfied, and this International Standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

For any of the generic packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS or COMPLEX_IO, an implementation may impose a restriction that the generic actual type shall not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction shall be documented, and the effects of violating the restriction shall be one of the following:

- Compilation of a unit containing an instantiation of that generic package is rejected.

- CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of that generic package.

Conversely, if an implementation does not impose the restriction, then such a range constraint shall not be allowed, when included with the user's actual type, to interfere with the internal computations of the subprograms; that is, if the arguments and result (of functions), or their components, are within the range of the type, then the implementation shall return the result (if any) and shall not raise an exception (such as CONSTRAINT_ERROR).

Any of the restrictions discussed above may in fact be inherited from implementations of the package GENERIC_ELEMENTARY_FUNCTIONS of ISO/IEC 11430 and the package GENERIC_PRIMITIVE_FUNCTIONS of ISO/IEC 11729, if used. The dependence of an implementation on such inherited restrictions should be documented.

Implementations of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS shall function properly in a tasking environment. Apart from the obvious restriction that an implementation of these packages shall avoid declaring variables that are global to the subprograms, no special constraints are imposed on implementations. With the exception of COMPLEX_IO, nothing in this International Standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means (for example) of preserving the sign of an infinitesimal quantity that has underflowed to zero. Implementations of these packages may exploit that capability, when available, so as to exhibit continuity in the results of ARGUMENT as certain limits are approached. At the same time, implementations in which that capability is unavailable are also allowed. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this International Standard, implementations are allowed the freedom not to exploit the capability, even when it is available. This International Standard does not specify the signs that an implementation exploiting signed zeros shall give to zero results; it does, however, specify that an implementation exploiting signed zeros shall yield a scalar result (or a scalar element of a composite result) for ARGUMENT that depends on the sign of a zero imaginary component of a scalar argument (or a corresponding scalar element of a composite argument). An implementation shall exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in these packages. In addition, an implementation shall document its behavior with respect to signed zeros.

In implementations of GENERIC_COMPLEX_TYPES and GENERIC_COMPLEX_ARRAYS, all operations involving mixed real and complex arithmetic are required to construct the result by using real arithmetic (instead of by converting real values to complex values and then using complex arithmetic). This is to facilitate conformance with IEEE arithmetic.

6 Exceptions

The ARGUMENT_ERROR exception is declared in GENERIC_COMPLEX_TYPES and GENERIC_COMPLEX_ARRAYS. This exception is raised by a subprogram in these generic packages when the argument(s) of the subprogram violate one or more of the conditions given in the subprogram's definition (see clause 8).

NOTE — These conditions are related only to the mathematical definition of the subprogram and are therefore implementation independent.

The ARRAY_INDEX_ERROR exception is declared in GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS. This exception is raised by a subprogram in these generic packages when the argument(s) of the subprogram violate one or more of the conditions for matching elements of arrays (as in predefined equality); that is, for dyadic array operations, the bounds of the given left and right array operands need not be equal, but their appropriate vector lengths or row and/or column lengths (for matrices) shall be equal.

The ARGUMENT_ERROR and ARRAY_INDEX_ERROR exceptions are declared as renamings of exceptions of the same name declared in the ELEMENTARY_FUNCTIONS_EXCEPTIONS package of ISO/IEC 11430 and in the ARRAY_EXCEPTIONS package of this International Standard, respectively. These exceptions distinguish neither between different kinds of argument errors or array index errors, nor between different subprograms. The ARGUMENT_ERROR exception does not distinguish between instantiations of either GENERIC_COMPLEX_TYPES, GENERIC_COMPLEX_ARRAYS, the GENERIC_ELEMENTARY_FUNCTIONS package of ISO/IEC 11430 or the GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS package of ISO/IEC 13814. The ARRAY_INDEX_ERROR exception does not distinguish between different instantiations of either GENERIC_REAL_ARRAYS or GENERIC_COMPLEX_ARRAYS.

Besides ARGUMENT_ERROR and ARRAY_INDEX_ERROR, the only exceptions allowed during a call to a subprogram in these packages are predefined exceptions, as follows:

— Virtually any predefined exception is possible during the evaluation of an argument of a subprogram in these packages. For example, NUMERIC_ERROR, CONSTRAINT_ERROR, or even PROGRAM_ERROR could be raised if an argument has an undefined value; and, as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when the value of an argument lies outside the range of the user's generic actual type. Additionally, STORAGE_ERROR could be raised, e.g. if insufficient storage is available to perform the call. All these exceptions are raised before the body of the subprogram is entered and therefore have no bearing on implementations of these packages.

— For the subprograms in COMPLEX_IO only, any of the exceptions declared (by renaming) in TEXT_IO may be raised in the appropriate circumstances. For example, TEXT_IO.LAYOUT_ERROR will be raised during an output operation to a string if the given string is too short to hold the formatted output. Additionally, TEXT_IO.DATA_ERROR will be raised during the evaluation of arguments of an input operation if the components of the complex value obtained are not of the type REAL, or, for implementations of COMPLEX_IO not based on an instantiation of TEXT_IO.FLOAT_IO, if the input sequence does not have the required syntax. Implementations of COMPLEX_IO which make use of an instantiation of TEXT_IO.FLOAT_IO shall make every attempt to raise TEXT_IO.DATA_ERROR in the presence of invalid input sequence syntax; however, this International Standard recognizes the difficulty in handling all possible invalid input sequences for these types of implementations.

— Also, as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when a subprogram in these packages attempts to return a scalar value (or to construct a composite value with a scalar component or element) outside the range of the user's generic actual type. The exception raised for this reason shall be propagated to the caller of the subprogram.

— Whenever the arguments of a subprogram are such that a scalar result (or a scalar component or element of a composite result) permitted by the accuracy requirements would exceed REAL'SAFE_LARGE in absolute value, as formalized below in clause 10, an implementation may raise (and shall then propagate to the caller) the exception specified by Ada for signaling overflow.

— Once execution of the body of a subprogram has begun, an implementation may propagate STORAGE_ERROR to the caller of the subprogram, but only to signal the unexpected exhaustion of storage. Similarly, once execution of the body of a subprogram has begun, an implementation may propagate PROGRAM_ERROR to the caller of the subprogram, but only to signal errors made by the user of these packages.

No exception is allowed during a call to a subprogram in these packages except those permitted by the foregoing rules. In particular, for arguments for which all scalar results (or scalar components or elements of all composite results) satisfying the accuracy requirements remain less than or equal to `REAL'SAFE_LARGE` in absolute value, a subprogram shall locally handle an overflow occurring during the computation of an intermediate result, if such an overflow is possible, and not propagate an exception signaling that overflow to the caller of the subprogram.

The only exceptions allowed during an instantiation of `GENERIC_COMPLEX_TYPES`, `GENERIC_REAL_ARRAYS`, `GENERIC_COMPLEX_ARRAYS` or `COMPLEX_IO`, including the execution of the optional sequence of statements in the body of the instance, are `CONSTRAINT_ERROR`, `PROGRAM_ERROR` and `STORAGE_ERROR`, and then only for the following reasons. The raising of `CONSTRAINT_ERROR` during instantiation is only allowed when the implementation imposes the restriction that the generic actual type shall not have a range constraint, and the user violates that restriction (it may, in fact, be an inescapable consequence of the violation). The raising of `PROGRAM_ERROR` during instantiation is only allowed for the purpose of signaling errors made by the user – for example, violation of this same restriction, or of other limitations of the implementation. The raising of `STORAGE_ERROR` during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

NOTE — In ISO/IEC 8652:1987, the exception specified for signaling overflow or division by zero is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

7 Arguments outside the range of safe numbers

ISO/IEC 8652 fails to define the result safe interval of any basic or predefined operation of a real subtype when the absolute value of one of its operands exceeds the largest safe number of the operand subtype. (The failure to define a result in this case occurs because no safe interval is defined for the operand in question.) In order to avoid imposing requirements that would, consequently, be more stringent than those of Ada itself, this International Standard likewise does not define the result of a contained subprogram when the absolute value of one of its scalar arguments (or one of the scalar components or elements of composite arguments) exceeds `REAL'SAFE_LARGE`. All of the accuracy requirements and other provisions of the following clauses are understood to be implicitly qualified by the assumption that scalar subprogram arguments (or scalar components or elements of composite subprogram arguments) are less than or equal to `REAL'SAFE_LARGE` in absolute value.

8 Method of specification of subprograms

Some of the subprograms have two or more overloaded forms. For each form of a subprogram covered by this International Standard, the subprogram is specified by its parameter and result type profile, the domain of its argument(s) if restricted, its range if restricted, and the accuracy required of its implementation. The meaning of, and conventions applicable to, the domain, range and accuracy specifications are described below.

The specification of each subprogram covered by this International Standard includes, where necessary, a characterization of the argument values for which the subprogram is mathematically defined. It is expressed by inequalities or other conditions which the arguments shall satisfy to be valid. Whenever the arguments fail to satisfy all the conditions, the implementation shall raise `ARGUMENT_ERROR`. It shall not raise that exception if all the conditions are satisfied.

Inability to deliver a result for valid arguments because the scalar result (or a scalar component or element of the composite result) overflows, for example, shall not raise `ARGUMENT_ERROR`, but shall be treated in the same way that Ada defines for its predefined floating-point operations (see clause 10).

The usual mathematical meaning of the "range" of a function is the set of values into which the function maps the values in its domain. Some of the subprograms covered by this International Standard (for example, `ARGUMENT`) are mathematically multivalued, in the sense that a given argument value can be mapped by the subprogram into many different result values. By means of range restrictions, this International Standard imposes a uniqueness requirement on the results of multivalued functions, thereby reducing them to single-valued functions.

The range of each subprogram result is shown, where necessary, in the specifications. Range definitions take the form of inequalities limiting the results of a subprogram. An implementation shall not exceed a limit of the range when

that limit is a safe number of `REAL` (like `0.0`, `1.0`, or `CYCLE/2.0` for certain values of `CYCLE`). On the other hand, when a range limit is not a safe number of `REAL` (like π, or `CYCLE/2.0` for certain other values of `CYCLE`), an implementation may exceed the range limit, but may not exceed the safe number of `REAL` next beyond the range limit in the direction away from the interior of the range; this is, in general, the best that can be expected from a portable implementation. Effectively, therefore, range definitions have the added effect of imposing accuracy requirements on implementations above and beyond those presented as error bounds in the specifications (see clause 9).

9 Accuracy requirements

Because they are implemented on digital computers with only finite precision, the subprograms provided in these generic packages can, at best, only approximate the corresponding mathematically defined operations.

The accuracy requirements contained in this International Standard define the latitude that implementations are allowed in approximating the intended precise mathematical result with floating-point computations. Accuracy requirements of two kinds are stated in the specifications. Additionally, range definitions impose requirements that constrain the values implementations may yield, so the range definitions are another source of accuracy requirements (in that context, the precise meaning of a range limit that is not a safe number of `REAL`, as an accuracy requirement, is discussed in clause 8). Every result returned by a subprogram is subject to all of the subprogram's applicable accuracy requirements, except in the one case described in clause 12. In that case, the scalar result (or scalar components or elements of the composite result) will satisfy a small absolute error requirement in lieu of the other accuracy requirements defined for the subprogram.

The accuracy requirements on array operations are defined in terms of corresponding accuracy requirements on their (real or complex) scalar elements, unless the mathematical definition of the operation includes an inner product (indicated in the specifications as such). The accuracy of operations involving inner products is beyond the scope of this International Standard, except that an implementation shall document what, if any, extended-precision accumulation of intermediate results is used to implement such inner products.

The first kind of (scalar) accuracy requirement used in the specifications is a "maximum relative error requirement." It is specified by bounds on appropriate measures of the relative error in the computed result of a subprogram, which shall hold (except as provided by the rules in clauses 10 and 12) for all arguments satisfying the conditions in the domain definition, whenever those measures are defined.

Three forms of measure are used in the specifications; they depend on the type (real, imaginary or complex) of the scalar result. In the real or imaginary case, the measure is the usual "relative error"; in the complex case, the measure used for each component-part is, whenever possible, a "component-part error," but in cases where substantial cancellation may be involved this is relaxed to a "box error."

For a real result, if the mathematical result is α and the computed result is x, then the relative error $rel\_err(x)$ is defined in the usual way:

$$rel\_err(x) = |\alpha - x|/|\alpha|$$

provided the mathematical result is finite and nonzero.

For a complex result, if the mathematical result is $\alpha + i\beta$ and the computed result is $x + iy$, then the component-part errors $real\_comp\_err(x)$, $imag\_comp\_err(y)$ are defined as:

$$real\_comp\_err(x) = |\alpha - x|/|\alpha|$$

provided the mathematical component-part α is finite and nonzero, and

$$imag\_comp\_err(y) = |\beta - y|/|\beta|$$

provided the mathematical component-part β is finite and nonzero; and the box errors $real\_box\_err(x)$, $imag\_box\_err(y)$ are defined as:

$$real\_box\_err(x) - |\alpha - x|/\max(|\alpha|, |\beta|)$$

$$imag\_box\_err(y) = |\beta - y|/\max(|\alpha|, |\beta|)$$

provided the mathematical component-parts α, β are finite and not both zero.

In all other cases, the above measures of the relative error are not defined (i.e., when the mathematical result, or a component-part of the mathematical result, is infinite or zero).

The second kind of (scalar) accuracy requirement used in the specifications is a stipulation, usually in the form of an equality, that the implementation shall deliver "prescribed results" for certain special arguments. It is used for two purposes:

- to define the computed result when one of the measures of the relative error is undefined, i.e., when the mathematical result (or a component-part of the mathematical result) is zero; and
- to strengthen the accuracy requirements at special argument values.

When such a prescribed result (or component-part of a prescribed result) is a safe number of `REAL` (like `0.0`, `1.0` or `CYCLE/2.0` for certain values of `CYCLE`), an implementation shall deliver that value. On the other hand, when a prescribed result (or component-part of a prescribed result) is not a safe number of `REAL` (like π, or `CYCLE/2.0` for certain other values of `CYCLE`), an implementation may deliver any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them. Complex results need not have the same kind of accuracy requirement for both of their component-parts. Where all results of an operation are prescribed, the operation is specified as "exact."

Range definitions in the specifications, are an additional source of accuracy requirements, as stated in clause 8. As an accuracy requirement, a range definition has the effect of eliminating some of the values permitted by the maximum relative error requirements, e.g. those outside the range.

10 Overflow

Floating-point hardware is typically incapable of representing numbers whose absolute value exceeds some implementation-defined maximum. For the type `REAL`, that maximum will be at least `REAL'SAFE_LARGE`. For the subprograms defined by this International Standard, whenever the maximum relative error requirements permit a scalar result (or a scalar component or element of a composite result) whose absolute value is greater than `REAL'SAFE_LARGE`, the implementation may

- yield any result permitted by the maximum relative error requirements, or
- raise the exception specified by Ada for signaling overflow.

In addition, some of the functions are allowed to signal overflow for certain arguments for which neither component of the result can overflow. This freedom is granted for operations involving either an inner product or complex exponentiation. Permission to signal overflow in these cases recognizes the difficulty of avoiding overflow in the computation of intermediate results, given the current state of the art.

NOTES

1 The rule permits an implementation to raise an exception, instead of delivering a result, for arguments for which the mathematical result (or a component-part of the mathematical result) is close to but does not exceed `REAL'SAFE_LARGE` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result (or a component-part of the mathematical result) does exceed `REAL'SAFE_LARGE` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

2 The rule is motivated by the behavior prescribed by ISO/IEC 8652 for the predefined operations. That is, when the set of possible results of a predefined operation includes a number whose absolute value exceeds the implementation-defined maximum, the implementation is allowed to raise the exception specified for signaling overflow instead of delivering a result.

3 In ISO/IEC 8652:1987, the exception specified for signaling overflow is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

11 Infinities

An implementation shall raise the exception specified by Ada for signaling division by zero in the following specific cases where the corresponding mathematical results, or component-parts thereof, are infinite:

a) division by (real, imaginary or complex) zero;

b) array operations whose mathematical definition involves division of an element by (real or complex) zero;

c) exponentiation of (real, imaginary or complex) zero by a negative (integer) exponent;

d) array operations whose mathematical definition involves exponentiation of (real or complex) zero by a negative (integer) exponent;

NOTE — In ISO/IEC 8652:1987, the exception specified for signaling division by zero is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

12 Underflow

Floating-point hardware is typically incapable of representing nonzero numbers whose absolute value is less than some implementation-defined minimum. For the type `REAL`, that minimum will be at most `REAL'SAFE_SMALL`. For the subprograms defined by this International Standard, whenever the maximum relative error requirements permit a scalar result (or a scalar component or element of a composite result) whose absolute value is less than `REAL'SAFE_SMALL` and a prescribed result is not stipulated, the implementation may yield for that scalar result (or a scalar component or element of that composite result)

a) any value permitted by the maximum relative error requirements;

b) any nonzero value less than or equal to `REAL'SAFE_SMALL` in magnitude (and having the correct sign, unless the maximum relative error requirements permit values with either sign); or

c) zero.

NOTES

1 Whenever the behavior on underflow is as described in 12 b) or 12 c), the maximum relative error requirements are, in general, unachievable and are waived.

2 The rule permits an implementation to deliver a scalar result (or component or element of a composite result) violating the maximum relative error requirements for arguments for which the mathematical result (or component-part of the result) equals or slightly exceeds `REAL'SAFE_SMALL` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result (or component-part of the result) is less than `REAL'SAFE_SMALL` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

13 Generic Complex Types Package

The generic package `GENERIC_COMPLEX_TYPES` defines operations and types for scalar complex arithmetic. One generic formal parameter, the floating-point type `REAL`, is defined for `GENERIC_COMPLEX_TYPES`. The corresponding generic actual parameter determines the precision of the arithmetic to be used in an instantiation of this generic package.

The Ada package specification for `GENERIC_COMPLEX_TYPES` is given in annex A.

13.1 Types

Two types are defined and exported by `GENERIC_COMPLEX_TYPES`. The type `COMPLEX` provides a cartesian representation of a complex number; it is declared as a record with two components which represent the real and imaginary parts. The type `IMAGINARY` is provided to represent a pure imaginary number; it is declared as a private type whose full type declaration reveals it to be derived from type `REAL`.

13.2 Constants

```
i: constant IMAGINARY := 1.0;
j: constant IMAGINARY := 1.0;
```

Each constant represents the imaginary unit value.

Each constant is exact.

13.3 COMPLEX selection, conversion and composition operations

```
function RE (X : COMPLEX) return REAL;
function IM (X : COMPLEX) return REAL;
```

Each function returns the specified cartesian component-part of `X`.

Each function is exact.

```
procedure SET_RE (X  : in out COMPLEX;
                  RE : in     REAL);
procedure SET_IM (X  : in out COMPLEX;
                  IM : in     REAL);
```

Each procedure resets the specified (cartesian) component of `X`; the other (cartesian) component is unchanged.

Each procedure is exact.

```
function "+" (LEFT  : REAL;
              RIGHT : IMAGINARY) return COMPLEX;
function "-" (LEFT  : REAL;
              RIGHT : IMAGINARY) return COMPLEX;
```

Each operation returns the `COMPLEX` result of applying the appropriate standard mathematical operation for arithmetic between real and imaginary numbers. This is also the standard mathematical operation for composing a complex number from real and imaginary numbers.

The real component-part of the result is exact. The imaginary component-part of the result shall satisfy the accuracy requirement of the appropriate unary operation for real arithmetic, as defined by Ada.

```
function "+" (LEFT  : IMAGINARY;
              RIGHT : REAL) return COMPLEX;
function "-" (LEFT  : IMAGINARY;
              RIGHT : REAL) return COMPLEX;
```

Each operation returns the `COMPLEX` result of applying the appropriate standard mathematical operation for arithmetic between real and imaginary numbers. This is also the standard mathematical operation for composing a complex number from real and imaginary numbers.

The real component-part of the result shall satisfy the accuracy requirement of the appropriate unary operation for real arithmetic, as defined by Ada. The imaginary component-part of the result is exact.

```
function COMPOSE_FROM_CARTESIAN (RE : REAL) return COMPLEX;
function COMPOSE_FROM_CARTESIAN (RE, IM : REAL) return COMPLEX;
```

Each function constructs a COMPLEX result (in cartesian representation) formed from given cartesian component-parts (when only the real component-part is given, a zero imaginary component-part is assumed).

Each function is exact.

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

Each function calculates and returns the specified polar component-part of X (where MODULUS(X) $\geq$ 0.0 and $-$CYCLE/2.0 $\leq$ ARGUMENT(X, CYCLE) $\leq$ CYCLE/2.0). CYCLE defines the period of ARGUMENT(X, CYCLE); when no CYCLE is given, a period of 2π is assumed ($-\pi \leq$ ARGUMENT(X) $\leq \pi$). The exception ARGUMENT_ERROR is raised for CYCLE $\leq$ 0.0.

The function MODULUS returns 0.0 when X = (0.0, 0.0).

For the function ARGUMENT, special cases are defined as follows:

a) When X.RE $\geq$ 0.0 and X.IM = 0.0, ARGUMENT returns 0.0.

b) When X.RE $<$ 0.0 and X.IM = 0.0, two cases can arise:

1) for an implementation exploiting signed zeros, ARGUMENT returns $-$CYCLE/2.0 (or $-\pi$) when X.IM is a negatively signed zero and CYCLE/2.0 (or π) when X.IM is a positively signed zero;

2) for an implementation not exploiting signed zeros, ARGUMENT returns CYCLE/2.0 (or π).

Otherwise, for the function MODULUS (and its renaming "abs"), the maximum relative error is 3.0·REAL'BASE'EPSILON. For the function ARGUMENT, the maximum relative error is 4.0 · REAL'BASE'EPSILON.

```
function COMPOSE_FROM_POLAR (MODULUS, ARGUMENT : REAL) return COMPLEX;
function COMPOSE_FROM_POLAR
            (MODULUS, ARGUMENT, CYCLE : REAL) return COMPLEX;
```

Each function constructs a COMPLEX result (in cartesian representation) formed from given polar component-parts. The period of ARGUMENT is specified by CYCLE; when no CYCLE is given, a period of 2π is assumed. The exception ARGUMENT_ERROR is raised for CYCLE $\leq$ 0.0.

For the functions COMPOSE_FROM_POLAR, the usual mathematical definitions apply, e.g., for MODULUS $<$ 0.0, the (cartesian) COMPLEX result is formed from |MODULUS| and the rotation of ARGUMENT by CYCLE/2.0 (or π); for |ARGUMENT| $>$ CYCLE/2.0 (or |ARGUMENT| $> \pi$), the (cartesian) COMPLEX result is formed by reducing ARGUMENT according to the period CYCLE (or 2π).

For these functions, special cases are defined as follows:

a) when MODULUS = 0.0, the result is (0.0, 0.0);

b) when ARGUMENT is an integral multiple of CYCLE (or an integral multiple of 2π), the real component-part of the result is MODULUS and the imaginary component-part of the result is 0.0;

c) when ARGUMENT is the sum of CYCLE/4.0 and an integral multiple of CYCLE (or the sum of $\pi/2$ and an integral multiple of 2π), the real component-part of the result is 0.0 and the imaginary component-part of the result is MODULUS;

d) when ARGUMENT is an odd integral multiple of CYCLE/2.0 (or an odd integral multiple of π), the real component-part of the result is −MODULUS and the imaginary component-part of the result is 0.0;

e) when ARGUMENT is the sum of −CYCLE/4.0 and an integral multiple of CYCLE (or the sum of $-\pi/2$ and an integral multiple of 2π), the real component-part of the result is 0.0 and the imaginary component-part of the result is −MODULUS.

Otherwise, for the function COMPOSE_FROM_POLAR with CYCLE specified, the maximum relative error in the cartesian component-parts is 3.0 · REAL'BASE'EPSILON. For the function COMPOSE_FROM_POLAR with CYCLE omitted, the maximum relative error in the cartesian component-parts is 3.0 · REAL'BASE'EPSILON when |ARGUMENT| is less than or equal to some documented implementation-dependent threshold, which shall be not less than

$$\text{REAL'MACHINE\_RADIX}^{\lfloor \text{REAL'MACHINE\_MANTISSA}/2 \rfloor}.$$

For larger values of |ARGUMENT|, degraded accuracy is allowed. An implementation shall document its behavior for large |ARGUMENT|.

13.4 COMPLEX arithmetic operations

```
function "+" (RIGHT : COMPLEX) return COMPLEX;
function "-" (RIGHT : COMPLEX) return COMPLEX;
function CONJUGATE (X : COMPLEX) return COMPLEX;
function "+" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT, RIGHT : COMPLEX) return COMPLEX;
```

Each operation applies the standard mathematical operation for complex arithmetic. This is also the standard mathematical operation for complex identity, negation, conjugation, addition and subtraction.

The real component-part of the result of CONJUGATE is exact. Otherwise, each cartesian component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada.

```
function "*" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT, RIGHT : COMPLEX) return COMPLEX;
```

Each operation applies the standard mathematical operation for complex arithmetic. This is also the standard mathematical operation for complex multiplication and division. The exception specified by Ada for signaling division by zero is raised when division by complex zero is attempted.

For complex multiplication, the maximum box error is 5.0 · REAL'BASE'EPSILON.

For complex division, the maximum box error is 13.0 · REAL'BASE'EPSILON.

```
function "**" (LEFT  : COMPLEX;
               RIGHT : INTEGER) return COMPLEX;
```

This operation returns the result of applying the standard mathematical operation for complex exponentiation by an integer power. The exception specified by Ada for signaling division by zero is raised when LEFT = (0.0, 0.0), and RIGHT < 0.

For this operation special cases are defined as follows:

a) when LEFT = (0.0, 0.0), and RIGHT > 0, the result is (0.0, 0.0);

b) when LEFT = (1.0, 0.0), the result is (1.0, 0.0);

c) when RIGHT = 0, the result is (1.0, 0.0);

d) when RIGHT = 1, the result is LEFT.

Otherwise, the following shall hold:

a) For an implementation which obtains the result by converting LEFT to a polar representation, exponentiating the modulus and multiplying the argument by RIGHT, and reconverting to a cartesian representation, an accuracy requirement is not specified.

b) For all other implementations, the box error of the result is obtained by applying the sequence of complex multiplications defined by RIGHT, assuming arbitrary association of the factors, and to the final complex division when RIGHT < 0.

Clause 10 applies when the arguments are such that computation of an intermediate result could signal overflow.

13.5 Mixed REAL and COMPLEX arithmetic operations

```
function "+" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "+" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
function "-" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the appropriate standard mathematical operation for arithmetic between real and complex numbers.

The real component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada. The imaginary component-part of the result is exact.

```
function "*" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "*" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
function "/" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
```

Each operation returns the result of applying the appropriate standard mathematical operation for arithmetic between real and complex numbers. The exception specified by Ada for signaling division by zero is raised when division by (real or complex) zero is attempted.

Each cartesian component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada. Each operation constructs the mathematical result by using real arithmetic (instead of by using complex arithmetic, after converting real values to complex values).

13.6 Mixed IMAGINARY and COMPLEX arithmetic operations

```
function "+" (LEFT  : IMAGINARY;
              RIGHT : COMPLEX) return COMPLEX;
function "+" (LEFT  : COMPLEX;
```

```
                RIGHT : IMAGINARY) return COMPLEX;
function "-" (LEFT  : IMAGINARY;
                RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT  : COMPLEX;
                RIGHT : IMAGINARY) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the appropriate standard mathematical operation for arithmetic between imaginary and complex numbers.

The real component-part of the result is exact. The imaginary component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada.

```
function "*" (LEFT  : IMAGINARY;
                RIGHT : COMPLEX) return COMPLEX;
function "*" (LEFT  : COMPLEX;
                RIGHT : IMAGINARY) return COMPLEX;
function "/" (LEFT  : IMAGINARY;
                RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT  : COMPLEX;
                RIGHT : IMAGINARY) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the appropriate standard mathematical operation for arithmetic between imaginary and complex numbers. The exception specified by Ada for signaling division by zero is raised when division by (imaginary or complex) zero is attempted.

Each cartesian component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada. Each operation constructs the mathematical result by using real arithmetic (instead of by using complex arithmetic, after converting real values to complex values).

13.7 IMAGINARY selection, conversion and composition operations

```
function IM (X : IMAGINARY) return REAL;
```

This function returns the REAL representation of X.

This function is exact.

```
procedure SET_IM (X  : out IMAGINARY;
                  IM : in  REAL);
```

This procedure sets the IMAGINARY representation of X.

This procedure is exact.

```
function COMPOSE_FROM_CARTESIAN (IM : IMAGINARY) return COMPLEX;
```

This function constructs a COMPLEX result (in cartesian representation) formed from the given IMAGINARY value (a zero real component-part is assumed).

This function is exact.

13.8 IMAGINARY ordinal and arithmetic operations

```
function "<" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
function "<=" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
function ">" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
function ">=" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
```

Each operation returns the result of applying the appropriate standard mathematical relational operation between real numbers to the REAL representations of LEFT and RIGHT.

Each result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada.

```
function "+" (RIGHT : IMAGINARY) return IMAGINARY;
function "-" (RIGHT : IMAGINARY) return IMAGINARY;
function CONJUGATE (X : IMAGINARY) return IMAGINARY renames "-";
function "abs" (RIGHT : IMAGINARY) return REAL;
function "+" (LEFT, RIGHT : IMAGINARY) return IMAGINARY;
function "-" (LEFT, RIGHT : IMAGINARY) return IMAGINARY;
function "*" (LEFT, RIGHT : IMAGINARY) return REAL;
function "/" (LEFT, RIGHT : IMAGINARY) return REAL;
```

Each operation applies the standard mathematical operation for imaginary arithmetic. This is also the standard mathematical operation for imaginary identity, negation (conjugation), absolute value, addition, subtraction, multiplication and division. The exception specified by Ada for signaling division by zero is raised when division by (imaginary) zero is attempted.

Each result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada.

```
function "**" (LEFT  : IMAGINARY;
               RIGHT : INTEGER) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the standard mathematical operation for imaginary exponentiation by an integer power. The exception specified by Ada for signaling division by zero is raised when LEFT $= 0.0$ and RIGHT < 0.

For this operation special cases are defined as follows:

a) when LEFT $= 0.0$ and RIGHT > 0, the result is $(0.0, 0.0)$;

b) when RIGHT $= 0$, the result is $(1.0, 0.0)$;

c) when RIGHT $= 1$, the result is $(0.0, \text{LEFT})$.

Otherwise, the following shall hold:

a) When RIGHT is even, the real component-part of the result shall satisfy the accuracy requirement of real exponentiation by an integer power, as defined by Ada. The imaginary component-part of the result is 0.0.

b) When RIGHT is odd, the real component-part of the result is 0.0. The imaginary component-part of the result shall satisfy the accuracy requirement of the appropriate operation for real exponentiation by an integer power, as defined by Ada.

13.9 Mixed REAL and IMAGINARY arithmetic operations

```
function "*" (LEFT  : REAL;
              RIGHT : IMAGINARY) return IMAGINARY;
function "*" (LEFT  : IMAGINARY;
              RIGHT : REAL) return IMAGINARY;
function "/" (LEFT  : REAL;
              RIGHT : IMAGINARY) return IMAGINARY;
function "/" (LEFT  : IMAGINARY;
              RIGHT : REAL) return IMAGINARY;
```

Each operation returns the REAL or IMAGINARY result of applying the appropriate standard mathematical operation for arithmetic between real and imaginary numbers. The exception specified by Ada for signaling division by zero is raised when division by (real or imaginary) zero is attempted.

Each result shall satisfy the accuracy requirement of the appropriate operation for real arithmetic, as defined by Ada.

14 Array Exceptions Package

The ARRAY_EXCEPTIONS package defines one exception, ARRAY_INDEX_ERROR, which is raised by a subprogram in the generic array packages when the argument(s) of that subprogram violate one or more of the conditions for matching elements of arrays (see clause 6).

The Ada package specification for ARRAY_EXCEPTIONS is given in annex B.

15 Generic Real Arrays Package

The generic package GENERIC_REAL_ARRAYS defines operations and types for real vector and matrix arithmetic. One generic formal parameter, the floating-point type REAL, is defined for GENERIC_REAL_ARRAYS. The corresponding generic actual parameter determines the precision of the arithmetic to be used in an instantiation of this generic package.

The Ada package specification for GENERIC_REAL_ARRAYS is given in annex C.

15.1 Types

Two types are defined and exported by GENERIC_REAL_ARRAYS. The composite type REAL_VECTOR is provided to represent a vector with elements of type REAL; it is defined as an unconstrained, one-dimensional array with an index of type INTEGER. The composite type REAL_MATRIX is provided to represent a matrix with elements of type REAL; it is defined as an unconstrained, two-dimensional array with indices of type INTEGER.

15.2 REAL_VECTOR arithmetic operations

```
function "+" (RIGHT : REAL_VECTOR) return REAL_VECTOR;
function "-" (RIGHT : REAL_VECTOR) return REAL_VECTOR;
function "abs" (RIGHT : REAL_VECTOR) return REAL_VECTOR;
```

Each operation returns the result of applying the appropriate operation to each element of RIGHT. This is also the standard mathematical operation for vector identity, negation and absolute value.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

```
function "+" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
function "-" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
function "/" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
```

Each operation returns the result of applying the appropriate operation to each element of LEFT and the matching element of RIGHT. This is also the standard mathematical operation for vector addition, subtraction, multiplication and division. The index range of the result is LEFT'RANGE. The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH $\neq$ RIGHT'LENGTH. The exception specified by Ada for signaling division by zero is raised when division by zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

```
function "**" (LEFT  : REAL_VECTOR;
               RIGHT : INTEGER) return REAL_VECTOR;
```

This operation returns the result of applying the standard mathematical operation for exponentiation by an integer power to each element of LEFT. The index range of the result is LEFT'RANGE. The exception specified by Ada for signaling division by zero is raised if for some integer I (in the index range of LEFT), LEFT(I) = 0.0 and RIGHT < 0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of exponentiation by an integer power, as defined by Ada.

```
function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL;
```

This operation returns the inner (dot) product of LEFT and RIGHT. The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH ≠ RIGHT'LENGTH.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of LEFT and RIGHT are such that computation of an intermediate result could signal overflow.

15.3 REAL_VECTOR scaling operations

```
function "*" (LEFT  : REAL;
              RIGHT : REAL_VECTOR) return REAL_VECTOR;
```

This operation applies the standard mathematical operation for scaling a vector RIGHT by a real number LEFT. The index range of the vector result is RIGHT'RANGE.

Each array element of the result shall satisfy the (scalar) accuracy requirement of multiplication, as defined by Ada.

```
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : REAL) return REAL_VECTOR;
function "/" (LEFT  : REAL_VECTOR;
              RIGHT : REAL) return REAL_VECTOR;
```

Each operation applies the standard mathematical operation for scaling a vector LEFT by a real number RIGHT. The index range of the vector result is LEFT'RANGE. The exception specified by Ada for signaling division by zero is raised when division by zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

15.4 Other REAL_VECTOR operations

```
function UNIT_VECTOR (INDEX : INTEGER;
                      ORDER : POSITIVE;
                      FIRST : INTEGER := 1) return REAL_VECTOR;
```

This function returns a "unit vector" with ORDER elements and a lower bound of FIRST. All elements are set to 0.0 except for the INDEX element which is set to 1.0. The exception ARRAY_INDEX_ERROR is raised if INDEX < FIRST or INDEX > FIRST + ORDER − 1; the exception CONSTRAINT_ERROR is raised if FIRST + ORDER − 1 > INTEGER'LAST.

This function is exact.

15.5 REAL_MATRIX arithmetic operations

```
function "+" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
function "-" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
function "abs" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

Each operation returns the result of applying the appropriate operation to each element of RIGHT. This is also the standard mathematical operation for matrix identity, negation and absolute value. The index ranges of the result are those of RIGHT.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

```
function TRANSPOSE (X : REAL_MATRIX) return REAL_MATRIX;
```

This function returns the transpose of a matrix X. The index ranges of the result are X'RANGE(2) and X'RANGE(1) (first and second index respectively).

This function is exact.

```
function "+" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
function "-" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

Each operation returns the result of applying the appropriate operation to each element of LEFT and the matching element of RIGHT. This is also the standard mathematical operation for matrix addition and subtraction. The index ranges of the result are those of LEFT. The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH(1) $\neq$ RIGHT'LENGTH(1) or LEFT'LENGTH(2) $\neq$ RIGHT'LENGTH(2).

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

```
function "*" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

This operation applies the standard mathematical operation for matrix multiplication. The index ranges of the result are LEFT'RANGE(1) and RIGHT'RANGE(2) (first and second index respectively). The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH(2) $\neq$ RIGHT'LENGTH(1).

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of LEFT and RIGHT are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL_MATRIX;
```

This operation applies the standard mathematical operation for multiplication of a (column) vector LEFT by a (row) vector RIGHT. The index ranges of the matrix result are LEFT'RANGE and RIGHT'RANGE (first and second index respectively).

Each array element of the result shall satisfy the (scalar) accuracy requirement of multiplication, as defined by Ada.

```
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : REAL_MATRIX) return REAL_VECTOR;
```

This operation applies the standard mathematical operation for multiplication of a (row) vector LEFT by a matrix RIGHT. The index range of the (row) vector result is RIGHT'RANGE(2). The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH $\neq$ RIGHT'LENGTH(1).

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of LEFT and RIGHT are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : REAL_VECTOR) return REAL_VECTOR;
```

This operation applies the standard mathematical operation for multiplication of a matrix `LEFT` by a (column) vector `RIGHT`. The index range of the (column) vector result is `LEFT'RANGE(1)`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

15.6 REAL_MATRIX scaling operations

```
function "*" (LEFT  : REAL;
              RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

This operation applies the standard mathematical operation for scaling a matrix `RIGHT` by a real number `LEFT`. The index ranges of the matrix result are those of `RIGHT`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of multiplication, as defined by Ada.

```
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : REAL) return REAL_MATRIX;
function "/" (LEFT  : REAL_MATRIX;
              RIGHT : REAL) return REAL_MATRIX;
```

Each operation applies the standard mathematical operation for scaling a matrix `LEFT` by a real number `RIGHT`. The index ranges of the matrix result are those of `LEFT`. The exception specified by Ada for signaling division by zero is raised when division by zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation, as defined by Ada.

15.7 Other REAL_MATRIX operations

```
function IDENTITY_MATRIX (ORDER : POSITIVE;
                          FIRST_1, FIRST_2 : INTEGER := 1) return REAL_MATRIX;
```

This function returns a square "identity matrix" with `ORDER`^{2} elements and lower bounds of `FIRST_1` and `FIRST_2` (for the first and second index ranges respectively). All elements are set to 0.0 except for the main diagonal, whose elements are set to 1.0. The exception `CONSTRAINT_ERROR` is raised if `FIRST_1` + `ORDER` $-$ 1 $>$ `INTEGER'LAST` or `FIRST_2` + `ORDER` $-$ 1 $>$ `INTEGER'LAST`.

This function is exact.

16 Generic Complex Arrays Package

The generic package `GENERIC_COMPLEX_ARRAYS` defines operations and types for complex and mixed real and complex vector and matrix arithmetic. Four generic formal type parameters are defined for `GENERIC_COMPLEX_ARRAYS`, including the floating-point type `REAL` which determines the precision of the arithmetic to be used in an instantiation of this generic package. The other generic formal type parameters are `REAL_VECTOR`, `REAL_MATRIX` and `COMPLEX`; a cartesian representation for the `COMPLEX` type is required throughout. Twenty-two generic formal subprogram parameters are also defined for `GENERIC_COMPLEX_ARRAYS`.

The Ada package specification for `GENERIC_COMPLEX_ARRAYS` is given in annex D.

16.1 Types

Two types are defined and exported by GENERIC_COMPLEX_ARRAYS. The composite type COMPLEX_VECTOR is provided to represent a vector with elements of type COMPLEX; it is defined as an unconstrained, one-dimensional array with an index of type INTEGER. The composite type COMPLEX_MATRIX is provided to represent a matrix with elements of type COMPLEX; it is defined as an unconstrained, two-dimensional array with indices of type INTEGER.

16.2 COMPLEX_VECTOR selection, conversion and composition operations

```
function RE (X : COMPLEX_VECTOR) return REAL_VECTOR;
function IM (X : COMPLEX_VECTOR) return REAL_VECTOR;
```

Each function returns a vector of the specified cartesian component-parts of X. The index range of the result is X'RANGE.

Each function is exact.

```
procedure SET_RE (X  : in out COMPLEX_VECTOR;
                  RE : in     REAL_VECTOR);
procedure SET_IM (X  : in out COMPLEX_VECTOR;
                  IM : in     REAL_VECTOR);
```

Each procedure resets the specified (cartesian) component of each of the elements of X; the other (cartesian) component of each of the elements is unchanged. The exception ARRAY_INDEX_ERROR is raised if X'LENGTH ≠ RE'LENGTH and if X'LENGTH ≠ IM'LENGTH.

Each procedure is exact.

```
function COMPOSE_FROM_CARTESIAN
   (RE : REAL_VECTOR) return COMPLEX_VECTOR;
function COMPOSE_FROM_CARTESIAN
   (RE, IM : REAL_VECTOR) return COMPLEX_VECTOR;
```

Each function constructs a vector of COMPLEX results (in cartesian representation) formed from given vectors of cartesian component-parts (when only the real component-parts are given, imaginary component-parts of zero are assumed). The index range of the result is RE'RANGE. The exception ARRAY_INDEX_ERROR is raised if RE'LENGTH ≠ IM'LENGTH.

Each function is exact.

```
function MODULUS (X : COMPLEX_VECTOR) return REAL_VECTOR;
function "abs" (RIGHT : COMPLEX_VECTOR) return REAL_VECTOR
   renames MODULUS;
function ARGUMENT (X : COMPLEX_VECTOR) return REAL_VECTOR;
function ARGUMENT (X     : COMPLEX_VECTOR;
                   CYCLE : REAL) return REAL_VECTOR;
```

Each function calculates and returns a vector of the specified polar component-parts of X. The index range of the result is X'RANGE. Each array element of the result shall satisfy the (scalar) range definition of the appropriate function.

CYCLE defines the period of ARGUMENT; when no CYCLE is given, a period of 2π is assumed. The exception ARGUMENT_ERROR is raised for CYCLE ≤ 0.0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate function.

```
function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_VECTOR) return COMPLEX_VECTOR;
function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_VECTOR;
    CYCLE             : REAL) return COMPLEX_VECTOR;
```

Each function constructs a vector of COMPLEX results (in cartesian representation) formed from given vectors of polar component-parts. Each element of ARGUMENT is assumed to have a period of CYCLE (and is reduced accordingly); when no CYCLE is given, a period of 2π is assumed. The index range of the result is MODULUS'RANGE. The exception ARRAY_INDEX_ERROR is raised if MODULUS'LENGTH $\neq$ ARGUMENT'LENGTH; the exception ARGUMENT_ERROR is raised for CYCLE ≤ 0.0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate function.

16.3 COMPLEX_VECTOR arithmetic operations

```
function "+" (RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "-" (RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

Each operation returns the result of applying the appropriate operation to each element of RIGHT. This is also the standard mathematical operation for vector identity and negation. The index range of the result is RIGHT'RANGE.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function CONJUGATE (X : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

This function returns the result of applying the standard mathematical operation for complex conjugation to each element of X. The index range of the result is X'RANGE.

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex conjugation.

```
function "+" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "-" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "/" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

Each operation returns the result of applying the appropriate operation to each element of LEFT and the matching element of RIGHT. This is also the standard mathematical operation for vector addition, subtraction, multiplication and division. The index range of the result is LEFT'RANGE. The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH $\neq$ RIGHT'LENGTH. The exception specified by Ada for signaling division by zero is raised when division by (complex) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function "**" (LEFT  : COMPLEX_VECTOR;
               RIGHT : INTEGER) return COMPLEX_VECTOR;
```

This operation returns the result of applying the standard mathematical operation for complex exponentiation by an integer power to each element of LEFT. The index range of the result is LEFT'RANGE. The exception specified by Ada for signaling division by zero is raised if for some integer I (in the index range of LEFT), LEFT(I) $= (0.0, 0.0)$ and RIGHT < 0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex exponentiation by an integer power.

```
function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX;
```

This operation returns the inner (dot) product of LEFT and RIGHT; no complex conjugation is performed. The exception ARRAY_INDEX_ERROR is raised if LEFT'LENGTH $\neq$ RIGHT'LENGTH.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of LEFT and RIGHT are such that computation of an intermediate result could signal overflow.

16.4 Mixed REAL_VECTOR and COMPLEX_VECTOR arithmetic operations

```
function "+" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "+" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
function "-" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "-" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
function "/" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "/" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
```

Each operation returns the result of applying the appropriate operation to each element of **LEFT** and the matching element of **RIGHT**. This is also the standard mathematical operation for vector addition, subtraction, multiplication and division. The index range of the result is **LEFT'RANGE**. The exception **ARRAY_INDEX_ERROR** is raised if **LEFT'LENGTH** $\neq$ **RIGHT'LENGTH**. The exception specified by Ada for signaling division by zero is raised when division by (real or complex) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for mixed real and complex arithmetic.

```
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX;
```

Each operation returns the inner (dot) product of **LEFT** and **RIGHT**. The exception **ARRAY_INDEX_ERROR** is raised if **LEFT'LENGTH** $\neq$ **RIGHT'LENGTH**.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of **LEFT** and **RIGHT** are such that computation of an intermediate result could signal overflow.

16.5 COMPLEX_VECTOR scaling operations

```
function "*" (LEFT  : COMPLEX;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for scaling a vector **RIGHT** by a complex number **LEFT**. The index range of the result is **RIGHT'RANGE**.

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex multiplication.

```
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : COMPLEX) return COMPLEX_VECTOR;
function "/" (LEFT  : COMPLEX_VECTOR;
              RIGHT : COMPLEX) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for scaling a vector LEFT by a complex number RIGHT. The index range of the result is LEFT'RANGE. The exception specified by Ada for signaling division by zero is raised when division by (complex) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function "*" (LEFT  : REAL;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for scaling a complex vector RIGHT by a real number LEFT. The index range of the result is RIGHT'RANGE.

Each array element of the result shall satisfy the (scalar) accuracy requirement of mixed real and complex multiplication.

```
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL) return COMPLEX_VECTOR;
function "/" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for scaling a complex vector LEFT by a real number RIGHT. The index range of the result is LEFT'RANGE. The exception specified by Ada for signaling division by zero is raised when division by (real) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for mixed real and complex arithmetic.

16.6 Other COMPLEX_VECTOR operations

```
function UNIT_VECTOR (INDEX : INTEGER;
                      ORDER : POSITIVE;
                      FIRST : INTEGER := 1) return COMPLEX_VECTOR;
```

This function returns a "unit vector" with ORDER elements and a lower bound of FIRST. All elements are set to $(0.0, 0.0)$ except for the INDEX element which is set to $(1.0, 0.0)$. The exception ARRAY_INDEX_ERROR is raised if INDEX $<$ FIRST or INDEX $>$ FIRST + ORDER − 1; the exception CONSTRAINT_ERROR is raised if FIRST + ORDER − 1 $>$ INTEGER'LAST.

This function is exact.

16.7 COMPLEX_MATRIX selection, conversion and composition operations

```
function RE (X : COMPLEX_MATRIX) return REAL_MATRIX;
function IM (X : COMPLEX_MATRIX) return REAL_MATRIX;
```

Each function returns a matrix of the specified cartesian component-parts of X. The index ranges of the result are those of X.

Each function is exact.

```
procedure SET_RE (X  : in out COMPLEX_MATRIX;
                  RE : in     REAL_MATRIX);
procedure SET_IM (X  : in out COMPLEX_MATRIX;
                  IM : in     REAL_MATRIX);
```

Each procedure resets the specified (cartesian) component of each of the elements of X; the other (cartesian) component of each of the elements is unchanged. The exception ARRAY_INDEX_ERROR is raised if X'LENGTH(1) ≠ RE'LENGTH(1) or X'LENGTH(2) ≠ RE'LENGTH(2) and if X'LENGTH(1) ≠ IM'LENGTH(1) or X'LENGTH(2) ≠ IM'LENGTH(2).

Each procedure is exact.

```
function COMPOSE_FROM_CARTESIAN
   (RE : REAL_MATRIX) return COMPLEX_MATRIX;
function COMPOSE_FROM_CARTESIAN
   (RE, IM : REAL_MATRIX) return COMPLEX_MATRIX;
```

Each function constructs a matrix of COMPLEX results (in cartesian representation) formed from given matrices of cartesian component-parts (when only the real component-parts are given, imaginary component-parts of zero are assumed). The index ranges of the result are those of RE. The exception ARRAY_INDEX_ERROR is raised if RE'LENGTH(1) ≠ IM'LENGTH(1) or RE'LENGTH(2) ≠ IM'LENGTH(2).

Each function is exact.

```
function MODULUS (X : COMPLEX_MATRIX) return REAL_MATRIX;
function "abs" (RIGHT : COMPLEX_MATRIX) return REAL_MATRIX
   renames MODULUS;
function ARGUMENT (X : COMPLEX_MATRIX) return REAL_MATRIX;
function ARGUMENT (X     : COMPLEX_MATRIX;
                   CYCLE : REAL) return REAL_MATRIX;
```

Each function calculates and returns a matrix of the specified polar component-parts of X. The index ranges of the result are those of X. Each array element of the result shall satisfy the (scalar) range definition of the appropriate function.

CYCLE defines the period of ARGUMENT; when no CYCLE is given, a period of 2π is assumed. The exception ARGUMENT_ERROR is raised for CYCLE ≤ 0.0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate function.

```
function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_MATRIX) return COMPLEX_MATRIX;
function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_MATRIX;
    CYCLE             : REAL) return COMPLEX_MATRIX;
```

Each function constructs a matrix of COMPLEX results (in cartesian representation) formed from given matrices of polar component-parts. Each element of ARGUMENT is assumed to have a period of CYCLE (and is reduced accordingly); when no CYCLE is given, a period of 2π is assumed. The index ranges of the result are those of MODULUS. The exception ARRAY_INDEX_ERROR is raised if MODULUS'LENGTH(1) ≠ ARGUMENT'LENGTH(1) or MODULUS'LENGTH(2) ≠ ARGUMENT'LENGTH(2); the exception ARGUMENT_ERROR is raised for CYCLE ≤ 0.0.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate function.

16.8 COMPLEX_MATRIX arithmetic operations

```
function "+" (RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

Each operation returns the result of applying the appropriate operation to each element of RIGHT. This is also the standard mathematical operation for matrix identity and negation. The index ranges of the result are those of RIGHT.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function CONJUGATE (X : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

This function returns the result of applying the standard mathematical operation for complex conjugation to each element of `X`. The index ranges of the result are those of `X`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex conjugation.

```
function TRANSPOSE (X : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

This function returns the transpose of a matrix `X`. The index ranges of the result are `X'RANGE(2)` and `X'RANGE(1)` (first and second index respectively).

This function is exact.

```
function "+" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

Each operation applies the appropriate standard mathematical operation for matrix addition or subtraction. The index ranges of the result are those of `LEFT`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(1)` $\neq$ `RIGHT'LENGTH(1)` or `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH(2)`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function "*" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

This operation applies the standard mathematical operation for matrix multiplication. The index ranges of the result are `LEFT'RANGE(1)` and `RIGHT'RANGE(2)` (first and second index respectively). The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH(1)`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_MATRIX;
```

This operation applies the standard mathematical operation for multiplication of a (column) vector by a (row) vector. The index ranges of the matrix result are `LEFT'RANGE` and `RIGHT'RANGE` (first and second index respectively).

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex multiplication.

```
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_VECTOR;
```

This operation applies the standard mathematical operation for multiplication of a (row) vector by a matrix. The index range of the (row) vector result is `RIGHT'RANGE(2)`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH` $\neq$ `RIGHT'LENGTH(1)`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

This operation applies the standard mathematical operation for multiplication of a matrix by a (column) vector. The index range of the (column) vector result is `LEFT'RANGE(1)`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

16.9 Mixed `REAL_MATRIX` and `COMPLEX_MATRIX` arithmetic operations

```
function "+" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "+" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;
```

Each operation applies the appropriate standard mathematical operation for matrix addition or subtraction. The index ranges of the result are those of `LEFT`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(1)` $\neq$ `RIGHT'LENGTH(1)` or `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH(2)`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for mixed real and complex arithmetic.

```
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for matrix multiplication. The index ranges of the result are `LEFT'RANGE(1)` and `RIGHT'RANGE(2)` (first and second index respectively). The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(2)` $\neq$ `RIGHT'LENGTH(1)`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_MATRIX;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for multiplication of a (column) vector by a (row) vector. The index ranges of the matrix result are `LEFT'RANGE` and `RIGHT'RANGE` (first and second index respectively).

Each array element of the result shall satisfy the (scalar) accuracy requirement of mixed real and complex multiplication.

```
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_VECTOR;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_MATRIX) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for multiplication of a (row) vector by a matrix. The index range of the (row) vector result is `RIGHT'RANGE(2)`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH` ≠ `RIGHT'LENGTH(1)`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

```
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
```

Each operation applies the standard mathematical operation for multiplication of a matrix by a (column) vector. The index range of the (column) vector result is `LEFT'RANGE(1)`. The exception `ARRAY_INDEX_ERROR` is raised if `LEFT'LENGTH(2)` ≠ `RIGHT'LENGTH`.

This operation involves an inner product; an accuracy requirement is not specified.

Clause 10 applies when the elements of `LEFT` and `RIGHT` are such that computation of an intermediate result could signal overflow.

16.10 `COMPLEX_MATRIX` scaling operations

```
function "*" (LEFT  : COMPLEX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for scaling a matrix `RIGHT` by a complex number `LEFT`. The index ranges of the result are those of `RIGHT`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of complex multiplication.

```
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : COMPLEX) return COMPLEX_MATRIX;
function "/" (LEFT  : COMPLEX_MATRIX;
              RIGHT : COMPLEX) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for scaling a matrix `LEFT` by a complex number `RIGHT`. The index ranges of the result are those of `LEFT`. The exception specified by Ada for signaling division by zero is raised when division by (complex) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for complex arithmetic.

```
function "*" (LEFT  : REAL;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for scaling a complex matrix `RIGHT` by a real number `LEFT`. The index ranges of the result are those of `RIGHT`.

Each array element of the result shall satisfy the (scalar) accuracy requirement of mixed real and complex multiplication.

```
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL) return COMPLEX_MATRIX;
function "/" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL) return COMPLEX_MATRIX;
```

Each operation applies the standard mathematical operation for scaling a complex matrix `LEFT` by a real number `RIGHT`. The index ranges of the result are those of `LEFT`. The exception specified by Ada for signaling division by zero is raised when division by (real) zero is attempted.

Each array element of the result shall satisfy the (scalar) accuracy requirement of the appropriate operation for mixed real and complex arithmetic.

16.11 Other `COMPLEX_MATRIX` operations

```
function IDENTITY_MATRIX (ORDER : POSITIVE;
                          FIRST_1, FIRST_2 : INTEGER := 1) return COMPLEX_MATRIX;
```

This function returns a square "identity matrix" with $\texttt{ORDER}^2$ elements and lower bounds of `FIRST_1` and `FIRST_2` (for the first and second index ranges respectively). All elements are set to $(0.0, 0.0)$ except for the main diagonal, whose elements are set to $(1.0, 0.0)$. The exception `CONSTRAINT_ERROR` is raised if $\texttt{FIRST\_1} + \texttt{ORDER} - 1 > \texttt{INTEGER'LAST}$ or $\texttt{FIRST\_2} + \texttt{ORDER} - 1 > \texttt{INTEGER'LAST}$.

This function is exact.

17 Generic Complex Input/Output Package

The generic package `COMPLEX_IO` defines procedures for the formatted input and output of scalar complex values. Exceptional conditions are reported by raising the appropriate exception defined in `TEXT_IO`.

Five generic formal parameters are defined for `COMPLEX_IO`, including the floating-point type `REAL` which determines the precision of the arithmetic to be used in an instantiation of this generic package. The other generic formal parameters are the type `COMPLEX` and subprograms to compose and decompose scalar complex values.

The Ada package specification for `COMPLEX_IO` is given in annex E.

```
procedure GET (FILE  : in  FILE_TYPE;
               ITEM  : out COMPLEX;
               WIDTH : in  FIELD := 0);
procedure GET (ITEM  : out COMPLEX;
               WIDTH : in  FIELD := 0);
```

Each procedure inputs a complex number from the indicated source. The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value; optionally, the pair of components may be separated by a comma and/or surrounded by a pair of parentheses. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter `WIDTH` is zero, then

a) line and page terminators are also allowed in these places;

b) the components shall be separated by at least one blank or line terminator if the comma is omitted; and

c) reading stops when the right parenthesis has been read, if the input sequence includes a left parenthesis, or when the imaginary component has been read, otherwise.

If a nonzero value of `WIDTH` is supplied, then

a) the components shall be separated by at least one blank if the comma is omitted; and

b) exactly `WIDTH` characters are read, or the characters (possibly none) up to a line terminator, whichever comes first (blanks are included in the count).

The value of type COMPLEX that corresponds to the input sequence is returned in the parameter ITEM.

The exception TEXT_IO.DATA_ERROR is raised if the input sequence does not have the required syntax, or if the components of the complex value obtained are not of type REAL. For an implementation of GET which uses invocation(s) of GET from an instantiation of TEXT_IO.FLOAT_IO, nonstandard behavior is permitted in the presence of invalid input sequence syntax. If nonstandard behavior is exhibited by an implementation, it shall be documented.

```
procedure PUT (FILE : in FILE_TYPE;
               ITEM : in COMPLEX;
               FORE : in FIELD := DEFAULT_FORE;
               AFT  : in FIELD := DEFAULT_AFT;
               EXP  : in FIELD := DEFAULT_EXP);
procedure PUT (ITEM : in COMPLEX;
               FORE : in FIELD := DEFAULT_FORE;
               AFT  : in FIELD := DEFAULT_AFT;
               EXP  : in FIELD := DEFAULT_EXP);
```

Each procedure outputs the value of the parameter ITEM as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically, each procedure

a) outputs a left parenthesis;

b) outputs the value of the real component of the parameter ITEM with the format defined by the corresponding PUT procedure of an instance of TEXT_IO.FLOAT_IO using the given values of FORE, AFT, and EXP;

c) outputs a comma;

d) outputs the value of the imaginary component of the parameter ITEM with the format defined by the corresponding PUT procedure of an instance of TEXT_IO.DATA_ERROR using the given values of FORE, AFT, and EXP; and

e) outputs a right parenthesis.

```
procedure GET (FROM : in  STRING;
               ITEM : out COMPLEX;
               LAST : out POSITIVE);
```

The procedure reads a complex value from the beginning of the given string, following the same rule as the GET procedure that reads a complex value from a file, but treating the end of the string as a line terminator. The value of type COMPLEX that corresponds to the input sequence is returned in the parameter ITEM; the index value such that FROM(LAST) is the last character read is returned in LAST.

The exception TEXT_IO.DATA_ERROR is raised if the input sequence does not have the required syntax, or if the components of the complex value obtained are not of the type REAL. For an implementation of GET which uses invocation(s) of GET from an instantiation of TEXT_IO.FLOAT_IO, nonstandard behavior is permitted in the presence of invalid input sequence syntax. If nonstandard behavior is exhibited by an implementation, it shall be documented.

```
procedure PUT (TO   : out STRING;
               ITEM : in  COMPLEX;
               AFT  : in  FIELD := DEFAULT_AFT;
               EXP  : in  FIELD := DEFAULT_EXP);
```

This procedure outputs the value of the parameter ITEM to the given string as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

a) a left parenthesis, the real component, and a comma are left justified in the given string, with the real component having the format defined by the PUT procedure (for output to a file) of an instance of TEXT_IO.FLOAT_IO using a value of zero for FORE and the given values of AFT and EXP;

b) the imaginary component and a right parenthesis are right justified in the given string, with the imaginary component having the format defined by the PUT procedure (for output to a file) of an instance of TEXT_IO.FLOAT_IO using a value for FORE that completely fills the remainder of the string, together with the given values of AFT and EXP.

The exception TEXT_IO.LAYOUT_ERROR is raised if the given string is too short to hold the formatted output.

18 Standard non-generic packages

In addition to the generic type packages, analogous non-generic packages are required to define standard scalar complex and imaginary types and standard real and complex vector and matrix types. Non-generic packages shall be provided for all precisions defined in package STANDARD. The same floating-point type shall be used to generate real and complex packages of the same precision.

The packages COMPLEX_TYPES, REAL_ARRAYS and COMPLEX_ARRAYS shall always be provided; these packages shall define the same types, constants (COMPLEX_TYPES only) and subprograms as GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS, respectively, except that the predefined type FLOAT shall replace type REAL throughout.

Names of the other non-generic packages (where defined) shall be assigned as follows:

- if the predefined floating-point type SHORT_FLOAT is supported by a host implementation of Ada, then this type shall be used to generate the packages SHORT_COMPLEX_TYPES, SHORT_REAL_ARRAYS and SHORT_COMPLEX_ARRAYS;

- if the predefined floating-point type LONG_FLOAT is supported by a host implementation of Ada, then this type shall be used to generate the packages LONG_COMPLEX_TYPES, LONG_REAL_ARRAYS and LONG_COMPLEX_ARRAYS; and

- if other predefined floating-point types are supported (e.g., LONG_LONG_FLOAT), package names shall be assigned by considering the predefined types in order of ascending (for LONG-types) or descending (for SHORT-types) precision and matching the prefix of each floating-point type with that of the corresponding package names.

Each non-generic package shall define the same types, constants (if applicable) and subprograms as the corresponding generic package, except that the appropriate predefined type shall replace type REAL throughout.

Annex A
(normative)
Ada specification for GENERIC_COMPLEX_TYPES

```
with ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic

  type REAL is digits <>;

package GENERIC_COMPLEX_TYPES is

-- TYPES --

  type COMPLEX is
    record
      RE, IM : REAL;
    end record;

  type IMAGINARY is private;

-- CONSTANTS --

  i: constant IMAGINARY;
  j: constant IMAGINARY;

-- SUBPROGRAMS for COMPLEX TYPES --

  -- COMPLEX selection, conversion and composition operations --

  function RE (X : COMPLEX) return REAL;
  function IM (X : COMPLEX) return REAL;

  procedure SET_RE (X  : in out COMPLEX;
                    RE : in     REAL);
  procedure SET_IM (X  : in out COMPLEX;
                    IM : in     REAL);

  function "+" (LEFT  : REAL;
                RIGHT : IMAGINARY) return COMPLEX;
  function "-" (LEFT  : REAL;
                RIGHT : IMAGINARY) return COMPLEX;

  function "+" (LEFT  : IMAGINARY;
                RIGHT : REAL) return COMPLEX;
  function "-" (LEFT  : IMAGINARY;
                RIGHT : REAL) return COMPLEX;

  function COMPOSE_FROM_CARTESIAN (RE : REAL) return COMPLEX;
  function COMPOSE_FROM_CARTESIAN (RE, IM : REAL) return COMPLEX;

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

```
                    CYCLE : REAL) return REAL;

function COMPOSE_FROM_POLAR (MODULUS, ARGUMENT : REAL) return COMPLEX;
function COMPOSE_FROM_POLAR
            (MODULUS, ARGUMENT, CYCLE : REAL) return COMPLEX;

-- COMPLEX arithmetic operations --

function "+" (RIGHT : COMPLEX) return COMPLEX;
function "-" (RIGHT : COMPLEX) return COMPLEX;
function CONJUGATE (X : COMPLEX) return COMPLEX;

function "+" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "*" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT, RIGHT : COMPLEX) return COMPLEX;
function "**" (LEFT  : COMPLEX;
               RIGHT : INTEGER) return COMPLEX;

-- Mixed REAL and COMPLEX arithmetic operations --

function "+" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "+" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
function "-" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
function "*" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "*" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;
function "/" (LEFT  : REAL;
              RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT  : COMPLEX;
              RIGHT : REAL) return COMPLEX;

-- Mixed IMAGINARY and COMPLEX arithmetic operations --

function "+" (LEFT  : IMAGINARY;
              RIGHT : COMPLEX) return COMPLEX;
function "+" (LEFT  : COMPLEX;
              RIGHT : IMAGINARY) return COMPLEX;
function "-" (LEFT  : IMAGINARY;
              RIGHT : COMPLEX) return COMPLEX;
function "-" (LEFT  : COMPLEX;
              RIGHT : IMAGINARY) return COMPLEX;
function "*" (LEFT  : IMAGINARY;
              RIGHT : COMPLEX) return COMPLEX;
function "*" (LEFT  : COMPLEX;
              RIGHT : IMAGINARY) return COMPLEX;
function "/" (LEFT  : IMAGINARY;
              RIGHT : COMPLEX) return COMPLEX;
function "/" (LEFT  : COMPLEX;
              RIGHT : IMAGINARY) return COMPLEX;
```

```
-- SUBPROGRAMS for IMAGINARY TYPES --

  -- IMAGINARY selection, conversion and composition operations --

  function IM (X : IMAGINARY) return REAL;

  procedure SET_IM (X  : out IMAGINARY;
                    IM : in  REAL);

  function COMPOSE_FROM_CARTESIAN (IM : IMAGINARY) return COMPLEX;

  -- IMAGINARY ordinal and arithmetic operations --

  function "<" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
  function "<=" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
  function ">" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;
  function ">=" (LEFT, RIGHT : IMAGINARY) return BOOLEAN;

  function "+" (RIGHT : IMAGINARY) return IMAGINARY;
  function "-" (RIGHT : IMAGINARY) return IMAGINARY;
  function CONJUGATE (X : IMAGINARY) return IMAGINARY renames "-";
  function "abs" (RIGHT : IMAGINARY) return REAL;

  function "+" (LEFT, RIGHT : IMAGINARY) return IMAGINARY;
  function "-" (LEFT, RIGHT : IMAGINARY) return IMAGINARY;
  function "*" (LEFT, RIGHT : IMAGINARY) return REAL;
  function "/" (LEFT, RIGHT : IMAGINARY) return REAL;

  function "**" (LEFT  : IMAGINARY;
                 RIGHT : INTEGER) return COMPLEX;

  -- Mixed REAL and IMAGINARY arithmetic operations --

  function "*" (LEFT  : REAL;
                RIGHT : IMAGINARY) return IMAGINARY;
  function "*" (LEFT  : IMAGINARY;
                RIGHT : REAL) return IMAGINARY;
  function "/" (LEFT  : REAL;
                RIGHT : IMAGINARY) return IMAGINARY;
  function "/" (LEFT  : IMAGINARY;
                RIGHT : REAL) return IMAGINARY;

-- EXCEPTIONS --

  ARGUMENT_ERROR: exception
                  renames ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;

-- IMAGINARY private definitions --

private

  type IMAGINARY is new REAL;
  i: constant IMAGINARY := 1.0;
  j: constant IMAGINARY := 1.0;

end GENERIC_COMPLEX_TYPES;
```

Annex B
(normative)
Ada specification for ARRAY_EXCEPTIONS

```
package ARRAY_EXCEPTIONS is

  ARRAY_INDEX_ERROR: exception;

end ARRAY_EXCEPTIONS;
```

Annex C
(normative)
Ada specification for GENERIC_REAL_ARRAYS

```
with ARRAY_EXCEPTIONS;
generic

  type REAL is digits <>;

package GENERIC_REAL_ARRAYS is

-- TYPES --

  type REAL_VECTOR is array (INTEGER range <>) of REAL;
  type REAL_MATRIX is array (INTEGER range <>,
                             INTEGER range <>) of REAL;

-- SUBPROGRAMS for REAL_VECTOR TYPES --

  -- REAL_VECTOR arithmetic operations --

  function "+" (RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "-" (RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "abs" (RIGHT : REAL_VECTOR) return REAL_VECTOR;

  function "+" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "-" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "/" (LEFT, RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "**" (LEFT  : REAL_VECTOR;
                 RIGHT : INTEGER) return REAL_VECTOR;

  function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL;

  -- REAL_VECTOR scaling operations --

  function "*" (LEFT  : REAL;
                RIGHT : REAL_VECTOR) return REAL_VECTOR;
  function "*" (LEFT  : REAL_VECTOR;
                RIGHT : REAL) return REAL_VECTOR;
  function "/" (LEFT  : REAL_VECTOR;
                RIGHT : REAL) return REAL_VECTOR;

  -- Other REAL_VECTOR operations --

  function UNIT_VECTOR (INDEX : INTEGER;
                        ORDER : POSITIVE;
                        FIRST : INTEGER := 1) return REAL_VECTOR;

-- SUBPROGRAMS for REAL_MATRIX TYPES --

  -- REAL_MATRIX arithmetic operations --

  function "+" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

```
   function "-" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
   function "abs" (RIGHT : REAL_MATRIX) return REAL_MATRIX;
   function TRANSPOSE (X : REAL_MATRIX) return REAL_MATRIX;

   function "+" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
   function "-" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
   function "*" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;

   function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL_MATRIX;
   function "*" (LEFT  : REAL_VECTOR;
                 RIGHT : REAL_MATRIX) return REAL_VECTOR;
   function "*" (LEFT  : REAL_MATRIX;
                 RIGHT : REAL_VECTOR) return REAL_VECTOR;

   -- REAL_MATRIX scaling operations --

   function "*" (LEFT  : REAL;
                 RIGHT : REAL_MATRIX) return REAL_MATRIX;
   function "*" (LEFT  : REAL_MATRIX;
                 RIGHT : REAL) return REAL_MATRIX;
   function "/" (LEFT  : REAL_MATRIX;
                 RIGHT : REAL) return REAL_MATRIX;

   -- Other REAL_MATRIX operations --

   function IDENTITY_MATRIX (ORDER : POSITIVE;
                             FIRST_1, FIRST_2 : INTEGER := 1) return REAL_MATRIX;

-- EXCEPTIONS --

   ARRAY_INDEX_ERROR: exception renames ARRAY_EXCEPTIONS.ARRAY_INDEX_ERROR;

end GENERIC_REAL_ARRAYS;
```

Annex D
(normative)
Ada specification for GENERIC_COMPLEX_ARRAYS

```
with ARRAY_EXCEPTIONS, ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic

  type REAL is digits <>;
  type REAL_VECTOR is array (INTEGER range <>) of REAL;
  type REAL_MATRIX is array (INTEGER range <>,
                             INTEGER range <>) of REAL;

  type COMPLEX is private;

  with function RE (X : COMPLEX) return REAL is <>;
  with function IM (X : COMPLEX) return REAL is <>;
  with procedure SET_RE (X  : in out COMPLEX;
                         RE : in     REAL) is <>;
  with procedure SET_IM (X  : in out COMPLEX;
                         IM : in     REAL) is <>;
  with function COMPOSE_FROM_CARTESIAN
     (RE, IM : REAL) return COMPLEX is <>;

  with function MODULUS (X : COMPLEX) return REAL is <>;
  with function ARGUMENT (X : COMPLEX) return REAL is <>;
  with function ARGUMENT (X     : COMPLEX;
                          CYCLE : REAL) return REAL is <>;
  with function COMPOSE_FROM_POLAR
     (MODULUS, ARGUMENT : REAL) return COMPLEX is <>;
  with function COMPOSE_FROM_POLAR
     (MODULUS, ARGUMENT, CYCLE : REAL) return COMPLEX is <>;

  with function "-" (RIGHT : COMPLEX) return COMPLEX is <>;
  with function CONJUGATE (X : COMPLEX) return COMPLEX is <>;

  with function "+" (LEFT, RIGHT : COMPLEX) return COMPLEX is <>;
  with function "-" (LEFT, RIGHT : COMPLEX) return COMPLEX is <>;
  with function "*" (LEFT, RIGHT : COMPLEX) return COMPLEX is <>;
  with function "/" (LEFT, RIGHT : COMPLEX) return COMPLEX is <>;
  with function "**" (LEFT  : COMPLEX;
                      RIGHT : INTEGER) return COMPLEX is <>;

  with function "+" (LEFT  : REAL;
                     RIGHT : COMPLEX) return COMPLEX is <>;
  with function "-" (LEFT  : REAL;
                     RIGHT : COMPLEX) return COMPLEX is <>;
  with function "*" (LEFT  : REAL;
                     RIGHT : COMPLEX) return COMPLEX is <>;
  with function "/" (LEFT  : REAL;
                     RIGHT : COMPLEX) return COMPLEX is <>;
  with function "/" (LEFT  : COMPLEX;
                     RIGHT : REAL) return COMPLEX is <>;

package GENERIC_COMPLEX_ARRAYS is
```

```
-- TYPES --

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

-- SUBPROGRAMS for COMPLEX_VECTOR types --

  -- COMPLEX_VECTOR selection, conversion and composition operations --

  function RE (X : COMPLEX_VECTOR) return REAL_VECTOR;
  function IM (X : COMPLEX_VECTOR) return REAL_VECTOR;

  procedure SET_RE (X  : in out COMPLEX_VECTOR;
                    RE : in     REAL_VECTOR);
  procedure SET_IM (X  : in out COMPLEX_VECTOR;
                    IM : in     REAL_VECTOR);

  function COMPOSE_FROM_CARTESIAN
     (RE : REAL_VECTOR) return COMPLEX_VECTOR;
  function COMPOSE_FROM_CARTESIAN
     (RE, IM : REAL_VECTOR) return COMPLEX_VECTOR;

  function MODULUS (X : COMPLEX_VECTOR) return REAL_VECTOR;
  function "abs" (RIGHT : COMPLEX_VECTOR) return REAL_VECTOR
     renames MODULUS;
  function ARGUMENT (X : COMPLEX_VECTOR) return REAL_VECTOR;
  function ARGUMENT (X     : COMPLEX_VECTOR;
                     CYCLE : REAL) return REAL_VECTOR;

  function COMPOSE_FROM_POLAR
     (MODULUS, ARGUMENT : REAL_VECTOR) return COMPLEX_VECTOR;
  function COMPOSE_FROM_POLAR
     (MODULUS, ARGUMENT : REAL_VECTOR;
      CYCLE             : REAL) return COMPLEX_VECTOR;

  -- COMPLEX_VECTOR arithmetic operations --

  function "+" (RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "-" (RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function CONJUGATE (X : COMPLEX_VECTOR) return COMPLEX_VECTOR;

  function "+" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "-" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "/" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "**" (LEFT  : COMPLEX_VECTOR;
                 RIGHT : INTEGER) return COMPLEX_VECTOR;

  function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX;

  -- Mixed REAL_VECTOR and COMPLEX_VECTOR arithmetic operations --

  function "+" (LEFT  : REAL_VECTOR;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "+" (LEFT  : COMPLEX_VECTOR;
```

```
                RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
  function "-" (LEFT  : REAL_VECTOR;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "-" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
  function "*" (LEFT  : REAL_VECTOR;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "*" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;
  function "/" (LEFT  : REAL_VECTOR;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "/" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;

  function "*" (LEFT  : REAL_VECTOR;
                RIGHT : COMPLEX_VECTOR) return COMPLEX;
  function "*" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL_VECTOR) return COMPLEX;

  -- COMPLEX_VECTOR scaling operations --

  function "*" (LEFT  : COMPLEX;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "*" (LEFT  : COMPLEX_VECTOR;
                RIGHT : COMPLEX) return COMPLEX_VECTOR;
  function "/" (LEFT  : COMPLEX_VECTOR;
                RIGHT : COMPLEX) return COMPLEX_VECTOR;

  function "*" (LEFT  : REAL;
                RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
  function "*" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL) return COMPLEX_VECTOR;
  function "/" (LEFT  : COMPLEX_VECTOR;
                RIGHT : REAL) return COMPLEX_VECTOR;

  -- Other COMPLEX_VECTOR operations --

  function UNIT_VECTOR (INDEX : INTEGER;
                        ORDER : POSITIVE;
                        FIRST : INTEGER := 1) return COMPLEX_VECTOR;

-- SUBPROGRAMS for COMPLEX_MATRIX TYPES --

  -- COMPLEX_MATRIX selection, conversion and composition operations --

  function RE (X : COMPLEX_MATRIX) return REAL_MATRIX;
  function IM (X : COMPLEX_MATRIX) return REAL_MATRIX;

  procedure SET_RE (X  : in out COMPLEX_MATRIX;
                    RE : in     REAL_MATRIX);
  procedure SET_IM (X  : in out COMPLEX_MATRIX;
                    IM : in     REAL_MATRIX);

  function COMPOSE_FROM_CARTESIAN
     (RE : REAL_MATRIX) return COMPLEX_MATRIX;
  function COMPOSE_FROM_CARTESIAN
     (RE, IM : REAL_MATRIX) return COMPLEX_MATRIX;
```

```
function MODULUS (X : COMPLEX_MATRIX) return REAL_MATRIX;
function "abs" (RIGHT : COMPLEX_MATRIX) return REAL_MATRIX
   renames MODULUS;

function ARGUMENT (X : COMPLEX_MATRIX) return REAL_MATRIX;
function ARGUMENT (X     : COMPLEX_MATRIX;
                   CYCLE : REAL) return REAL_MATRIX;

function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_MATRIX) return COMPLEX_MATRIX;
function COMPOSE_FROM_POLAR
   (MODULUS, ARGUMENT : REAL_MATRIX;
    CYCLE             : REAL) return COMPLEX_MATRIX;

-- COMPLEX_MATRIX arithmetic operations --

function "+" (RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function CONJUGATE (X : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function TRANSPOSE (X : COMPLEX_MATRIX) return COMPLEX_MATRIX;

function "+" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "*" (LEFT, RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;

function "*" (LEFT, RIGHT : COMPLEX_VECTOR) return COMPLEX_MATRIX;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_VECTOR;
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;

-- Mixed REAL_MATRIX and COMPLEX_MATRIX arithmetic operations --

function "+" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "+" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "-" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
function "*" (LEFT  : COMPLEX_MATRIX;
              RIGHT : REAL_MATRIX) return COMPLEX_MATRIX;

function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_MATRIX;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_VECTOR) return COMPLEX_MATRIX;
function "*" (LEFT  : REAL_VECTOR;
              RIGHT : COMPLEX_MATRIX) return COMPLEX_VECTOR;
function "*" (LEFT  : COMPLEX_VECTOR;
              RIGHT : REAL_MATRIX) return COMPLEX_VECTOR;
function "*" (LEFT  : REAL_MATRIX;
              RIGHT : COMPLEX_VECTOR) return COMPLEX_VECTOR;
```

```
  function "*" (LEFT  : COMPLEX_MATRIX;
                RIGHT : REAL_VECTOR) return COMPLEX_VECTOR;

  -- COMPLEX_MATRIX scaling operations --

  function "*" (LEFT  : COMPLEX;
                RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
  function "*" (LEFT  : COMPLEX_MATRIX;
                RIGHT : COMPLEX) return COMPLEX_MATRIX;
  function "/" (LEFT  : COMPLEX_MATRIX;
                RIGHT : COMPLEX) return COMPLEX_MATRIX;

  function "*" (LEFT  : REAL;
                RIGHT : COMPLEX_MATRIX) return COMPLEX_MATRIX;
  function "*" (LEFT  : COMPLEX_MATRIX;
                RIGHT : REAL) return COMPLEX_MATRIX;
  function "/" (LEFT  : COMPLEX_MATRIX;
                RIGHT : REAL) return COMPLEX_MATRIX;

  -- Other COMPLEX_MATRIX operations --

  function IDENTITY_MATRIX (ORDER : POSITIVE;
                            FIRST_1, FIRST_2 : INTEGER := 1) return COMPLEX_MATRIX;

-- EXCEPTIONS --

  ARGUMENT_ERROR: exception
                  renames ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;
  ARRAY_INDEX_ERROR: exception renames ARRAY_EXCEPTIONS.ARRAY_INDEX_ERROR;

end GENERIC_COMPLEX_ARRAYS;
```

Annex E
(normative)
Ada specification for COMPLEX_IO

```
with TEXT_IO; use TEXT_IO;
generic

  type REAL is digits <>;

  type COMPLEX is private;

  with function RE (X : COMPLEX) return REAL is <>;
  with function IM (X : COMPLEX) return REAL is <>;
  with function COMPOSE_FROM_CARTESIAN(RE, IM : REAL) return COMPLEX is <>;

package COMPLEX_IO is

  DEFAULT_FORE : FIELD := 2;
  DEFAULT_AFT  : FIELD := REAL'DIGITS - 1;
  DEFAULT_EXP  : FIELD := 3;

  procedure GET (FILE  : in  FILE_TYPE;
                 ITEM  : out COMPLEX;
                 WIDTH : in  FIELD := 0);
  procedure GET (ITEM  : out COMPLEX;
                 WIDTH : in  FIELD := 0);

  procedure PUT (FILE : in FILE_TYPE;
                 ITEM : in COMPLEX;
                 FORE : in FIELD := DEFAULT_FORE;
                 AFT  : in FIELD := DEFAULT_AFT;
                 EXP  : in FIELD := DEFAULT_EXP);
  procedure PUT (ITEM : in COMPLEX;
                 FORE : in FIELD := DEFAULT_FORE;
                 AFT  : in FIELD := DEFAULT_AFT;
                 EXP  : in FIELD := DEFAULT_EXP);

  procedure GET (FROM : in  STRING;
                 ITEM : out COMPLEX;
                 LAST : out POSITIVE);
  procedure PUT (TO   : out STRING;
                 ITEM : in  COMPLEX;
                 AFT  : in  FIELD := DEFAULT_AFT;
                 EXP  : in  FIELD := DEFAULT_EXP);

end COMPLEX_IO;
```

Annex F
(informative)
Rationale

F.1 Abstract

This annex, a revision of [6], outlines the history, purpose, features and development of International Standard ISO/IEC 13813 and provides a rationale for its features. Based on recommendations made jointly by the Ada-Europe Numerics Working Group and the ACM SIGAda Numerics Working Group, the real and complex types and operations standard is the third of four ISO standards to address the interrelated issues of portability, efficiency and robustness of numerical software written in ISO/IEC 8652:1987. Its purpose, features and development are outlined in this commentary.

F.2 Introduction

The absence from ISO/IEC 8652:1987 of predefined types and operations for complex arithmetic and for real and complex vector and matrix arithmetic has been one of the deterrents to the portability of scientific and engineering applications software written in that language. Whilst particular vendors have provided proprietary packages, this has done little to solve the broader problem of portability for applications packages using these operations. This is because of the lack of commonality among different packages: they differ in the number of operations implemented, their names and parameter type profiles, the handling of exceptional conditions, the precision of the types and the accuracy of the operations, and even the use (or avoidance) of genericity.

International Standard ISO/IEC 13813 defines a collection of generic packages: namely, one package for complex and imaginary types and operations; another package for real vector and matrix types and operations; a third package for complex vector and matrix types and operations, including mixed real and complex operations; and a fourth package for the input and output of complex scalar values. (The reasons for adopting this form of packaging are discussed in clause F.16.)

F.3 What basic operations are included?

In this clause we discuss the types and operations that are included in the packages as a whole — the subdivision of the facilities into separate packages is discussed later in this annex. Floating-point types and their basic operations are defined by ISO/IEC 8652:1987, but types and operations involving vectors and matrices are not; for complex arithmetic, not even the complex types and their basic operations are defined by ISO/IEC 8652:1987. Real and complex arithmetic have many features in common; where possible this commonality has been retained. Also, certain extensions have been included for complex arithmetic, e.g. conjugation.

There are the usual mathematical operations for vector and matrix arithmetic; for example, for vectors `X`, `Y` the operation `X + Y` is defined, and for `X * Y` two operations are defined — the scalar product of `X` and `Y` returning a scalar result and the matrix product of `X` and `Y` returning a matrix result. Additionally, applications, particularly in signal processing, use componentwise products; hence there is a third componentwise product of `X` and `Y` returning a vector result (this is defined by applying the multiplication operation to matching components of `X` and `Y`).

This International Standard provides both the usual mathematical operations for vectors and matrices as well as the componentwise operations for vectors (in some cases they are one and the same, e.g. `X + Y`). Hence, the operations `+`, `-`, `*`, `/`, `**` are provided as appropriate for vectors and matrices; additionally, operations for scaling vectors and matrices are provided (e.g. `X * Y` for a vector `X` and a scalar `Y`). Both left-hand and right-hand operations are provided where necessary, thus `X * Y` and `Y * X` are both provided.

Operator notation is used throughout for clarity and ease of use. This has the consequence that procedured forms are not used and the optimizers of compilation systems may sometimes be relied upon to avoid unnecessary intermediate copying of array results (e.g. `(X + Y) * X`; the intermediate result `X + Y` is calculated, but it is assumed that the implementation will not copy the vector result, where it is inefficient to do so, before calculating `(X + Y) * X`). This

choice of operator notation also has the consequence that only straightforward operations can be included; operations involving array sections which are common in the Basic Linear Algebra Subprograms (BLAS) are not part of this standard; such operations would require additional parameters to define the array section (e.g. the row number and/or its bounds).

The choice of basic operations for scalars has been strictly limited—in the real case to those provided for floating-point types, and in the complex and imaginary case, to include only limited extensions (e.g. `CONJUGATE`). Operations such as `MIN`, `MAX`, `SIGN` were considered, but were rejected for this standard since it was felt that they could be more appropriately provided elsewhere or in other ways. This choice has the added advantage of keeping the real and complex operations as similar as possible since `MIN`, `MAX` and `SIGN` are inappropriate for complex values. This design philosophy is also partly responsible for the inclusion of certain operations, such as `GET` and `PUT` for the input and output of complex scalars, respectively.

Several operations have been provided for the selection, composition and construction of complex values from floating-point values. The `COMPLEX` type in `GENERIC_COMPLEX_TYPES` can be manipulated directly since it is not private; however, the `RE`, `IM`, `SET_RE` and `SET_IM` operations are required for use as generic actual parameters and have useful vector and matrix forms. The traditional composition method from cartesian and polar coordinates is provided by `COMPOSE_FROM_CARTESIAN` and `COMPOSE_FROM_POLAR`, respectively, and these procedures have vector and matrix analogs as well. However, a more natural and abstract method is provided for complex scalars through the use of mixed real and imaginary arithmetic operations with the imaginary constant `i` (or equivalently `j`). For example, the construction of a complex number can be accomplished with the mixed real and imaginary operations `*` and `+` (i.e., one would write `X + i * Y`). An optimized implementation need not perform real arithmetic during this composition process since all that is required is the conversion of `Y` to the type `IMAGINARY` and construction of the complex scalar `(X, Y)`. Thus, application code involving complex scalars can be written in a concise, readable way, without sacrificing efficiency and performance.

The abstraction of expressing complex numbers in mathematical terms requires certain pure imaginary, mixed real and imaginary and mixed real and complex operations to be defined. A full set of pure imaginary arithmetic operations are provided, both unary and binary, including integer exponentiation. Ordinal arithmetic operations on the private type `IMAGINARY` are also required for completeness. The analog of the `REAL` selection and composition operations are provided for `IMAGINARY` types; `SET_IM` is defined as a procedure to be consistent with the `COMPLEX` operation of the same name. Both left-hand and right-hand `COMPLEX` composition operations are provided, i.e., mixed real and imaginary `+` and `-`, but note there is no mixed real and imaginary `COMPOSE_FROM_CARTESIAN` function. The other mixed real and imaginary arithmetic operations (`*` and `/`) are also provided, as well as the set of mixed imaginary and complex arithmetic operations. Vector and matrix forms of the imaginary operations are not provided.

For reasons of efficiency, it is often desirable not to use full complex arithmetic when only one of the operands is complex, but instead, use operations on the components (which are real). "Mixed arithmetic" is provided for this purpose. For example, for a complex value `X` and a real value `Y`, then `X * Y` need only involve two real multiplications rather than the four that would be required for the product of two complex operands.

Operations that provide combinations of more primitive operations could also have been included. Consider, for example, the scalar product `*`; in the complex case the elements of one of the vectors are sometimes conjugated. Hence, `X * Y` and a combined operation `X * CONJUGATE(Y)` could both have been provided; since the types of the operands no longer distinguish the operations, some other name would have had to be found for one of the operations. Analogous combined operations occur for `TRANSPOSE`, and also for both conjugation and transposition (Hermitian transposition in the complex case). Inclusion of all combined operations would have had a dramatic effect on the size of the packages, hence only the primitive operations are provided, and it is left to the user to combine them in an appropriate way.

F.4 Selecting an array index subtype

The choice of `INTEGER` for the array index subtype in `GENERIC_REAL_ARRAYS` and `GENERIC_COMPLEX_ARRAYS` is consistent with established norms in the application areas for which these packages are intended. This is not to say that other possible choices were not considered. The type `LONG_INTEGER` was considered but rejected since it would require significantly more overhead than `INTEGER` and the latter was deemed to have sufficient range for all but pathological

applications. Defining a generic formal parameter INDEX_TYPE to allow GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS to be instantiated with any discrete type was also discussed. This would have allowed creation of arrays with enumeration type values for indices, for example. This idea was ruled out for several reasons:

— Each discrete type supported by an implementation of these packages would require non-generic packages corresponding to GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS to be provided for all of the floating-point precisions defined in package STANDARD. This would have a dramatic, multiplicative effect on the number of packages defined by this International Standard.

— Array packages involving an enumeration index subtype would require attributes such as PRED and SUCC to be used as part of an implementation, perhaps limiting their efficiency. Note that GENERIC_COMPLEX_ARRAYS and GENERIC_REAL_ARRAYS do not require the array operands in dyadic operations to have identical ranges; hence, some form of index offsetting may be required to form the result of an operation.

— The packages defined in this International Standard are intended for general use and extensions such as a generalized index type were deemed to be more appropriate for a set of related packages.

F.5 The use of overloadings versus default values

Close scrutiny of this International Standard will show a subtle overloading of one of the conversion functions: COMPOSE_FROM_CARTESIAN is provided to compose the real and imaginary parts (given as real values) into the complex representation. It is overloaded with the additional form:

```
function COMPOSE_FROM_CARTESIAN (RE : REAL) return COMPLEX;
```

rather than defining a single function with a default for the real imaginary part:

```
function COMPOSE_FROM_CARTESIAN (RE : REAL;
                                 IM : REAL := 0.0) return COMPLEX;
```

(There is another form of the COMPOSE_FROM_CARTESIAN function, namely,

```
function COMPOSE_FROM_CARTESIAN (IM : IMAGINARY) return COMPLEX;
```

but it is not relevant to this discussion.)

The reason is that the separate one-parameter form is also needed as the complex equivalent of

```
function "+" (X : REAL) return REAL;
```

Such functions are used as generic actual parameters whenever a generic formal parameter of the form

```
function CONVERT (X : REAL) return SCALAR_TYPE;
```

is required to convert from real to a general scalar type (real or complex).

Overloadings of the ARGUMENT and COMPOSE_FROM_POLAR functions are provided to allow for arbitrary angular unit values, e.g.,

```
function COMPOSE_FROM_POLAR (MODULUS, ARGUMENT : REAL) return COMPLEX;
function COMPOSE_FROM_POLAR
            (MODULUS, ARGUMENT, CYCLE : REAL) return COMPLEX;
```

where the two-parameter form has the usual radian angular measure. The primary reason to utilize overloading instead of a single function with a default value is quite different than that for COMPOSE_FROM_CARTESIAN and has to do with the transcendental nature of 2π. Since 2π cannot be exactly represented, in terms of accuracy, the most one could expect from an implementation is the use of an internal representation of 2π of high precision (perhaps better precision than provided by the generic parameter type REAL), which ignores user-specified values for CYCLE in a small, arbitrary interval about 2π. The additional overhead involved in the interval comparison and its arbitrary nature make this approach somewhat unattractive.

The issue of accuracy cannot easily be ignored for the function COMPOSE_FROM_POLAR with CYCLE omitted, since amplification of errors in the computed result is unavoidable for large values of ARGUMENT, hence the relaxation of the accuracy requirement for such values exceeding an implementation-dependent threshold. This error amplification stems from the inability, when reducing ARGUMENT to its principal value, to produce the exact remainder of a cycle whose length cannot be precisely determined (see the rationale in ISO/IEC 11430 for a more detailed discussion). Choosing an overloading scheme for the ARGUMENT and COMPOSE_FROM_POLAR functions nicely hides the accuracy considerations for 2π from the user, thereby making it the responsibility of implementors of GENERIC_COMPLEX_TYPES (as it should be). It has the added benefit of parameter profile consistency with COMPOSE_FROM_CARTESIAN and the trigonometric functions defined in both ISO/IEC 11430 and ISO/IEC 13814. This last benefit is significant because the trigonometric functions in ISO/IEC 11430 can be used to implement COMPOSE_FROM_POLAR.

F.6 Should constants be included?

Early drafts of this International Standard specifically excluded constants, although complex zero, complex one, complex i (the square root of -1), and empty vectors and matrices were considered for inclusion. To include such vectors (and matrices) as constants involves constructs of the form:

```
NULL_VECTOR : constant REAL_VECTOR (1..0) := (others => 0.0);
```

Such constructs were considered to be rather unusual, and a sufficiently strong case could not be found for their inclusion. (The choice of bounds 1..0 was in any case arbitrary and other choices were possible). Further, problems with naming constants for different precisions arose: should SHORT_COMPLEX_ZERO, COMPLEX_ZERO, LONG_COMPLEX_ZERO, etc., be provided with different names in different packages (and in a generic package, what name should be chosen)? Alternatively, functions returning a complex zero, with overloadings for the different precisions, were considered. The initial conclusion was that the inclusion of such constants did not add to the functionality of the packages, and that they could be provided by appropriate aggregates (e.g. $(0.0, 0.0)$) or by calls to the composition functions (e.g. COMPOSE_FROM_CARTESIAN (0.0)). Indeed, the COMPLEX type is a visible, rather than a private, type specifically to allow the writing of such attributes (see also clause F.9). The marginal increase in convenience to the user was considered insufficient for the inclusion of an arbitrary number of constants.

The standardization process for ISO/IEC 8652:1995 and the concurrent wider attention given to Ada numerics significantly altered the landscape of this International Standard. One of the conclusions made during this revision of the Ada language was the importance of the standardized support for the expression of complex scalars in the usual mathematical way, i.e., as X + i * Y rather than (X,Y), where X and Y are of type REAL. As a result, the IMAGINARY type and the constant i (and equivalently j) were included in ISO/IEC 8652:1995, and the ACM SIGAda Numerics Working Group voted to adopt this modification to this International Standard as well. Thus, the constants i and j of type IMAGINARY are defined in GENERIC_COMPLEX_TYPES, and shall similarly be defined in the standard non-generic packages, e.g., COMPLEX_TYPES. These are the only constants defined in this International Standard.

F.7 Why define a type IMAGINARY?

One of the main goals for an object-based language such as ISO/IEC 8652:1987 is to provide facilities for the representation and manipulation of objects which is as natural and consistent as possible with real-world constructs. This was the primary technical motivation to standardize facilities for expressing a complex number as a mathematical object represented by X + i * Y (or equivalently X + j * Y) in this International Standard (although facilities for the two-component record notation (X,Y) are also provided). Use of a more abstract notation for a complex number (or any object) serves to distinguish it from its underlying storage representation.

Requisite to this abstract notation is the definition of a constant `i` to represent the imaginary unit value. However, there are subtle difficulties with the obvious approach of defining `i` to be a constant of type `COMPLEX` (with an imaginary component of unit value). Constructing `X + i * Y` would involve two mixed real and complex operations which consist only of degenerate component operations. The suppression of promoting `REAL` values to `COMPLEX` values (as prescribed by this International Standard) ameliorates the situation by reducing the number of component operations; however, the systematic avoidance of degenerate operations is still a primary concern to ensure efficiency. Implementations must take algorithmic measures, such as argument prescreening, or rely upon an optimizing compiler to produce efficient code.

For implementations which are compatible with IEEE arithmetic, there is the additional concern of corrupted results when infinite or signed zero components are encountered. For example, using the standard formula for complex multiplication, the product `(2.0*i)*(∞*i)` yields `NaN+∞*i` (where "NaN" is not "Not-a-Number") since $0.0*\infty =$ `NaN` and $x +$ `NaN` $=$ `NaN` for any real number x. Argument prescreening can be applied to obtain the true result of $-\infty$, but at the cost of additional overhead. An example of corrupted results with signed zero components occurs when calculating the sum $(2.0 * \texttt{i}) + (-0.0 + 2.0 * \texttt{i})$; the true result has a real component of -0.0 but applying the usual complex addition algorithm yields a zero of positive sign.

The inclusion of a type `IMAGINARY` provides a solution to the difficulties noted above. It permits the definition of distinct mixed complex and imaginary operations which remove the extraneous zero real component from consideration, thereby avoiding the possibility of corrupted results. Defining a set of mixed real and imaginary operations makes the construction of the canonical form of a complex number trivial — the mixed real and (unit) imaginary multiplication is simply a type conversion and the mixed real and imaginary addition reduces to setting the cartesian components of the complex result. It is clearly possible for the real component of a complex number to vanish, resulting in an unconverted number of type `COMPLEX` which lies on the imaginary axis. This is consistent with the relationship between other types, e.g., complex numbers which lie on the real axis and real numbers which do not have a fractional part.

`GENERIC_COMPLEX_TYPES` defines a full set of pure imaginary operations (a separate package for type `IMAGINARY` is unwarranted since it primarily supports the writing of `COMPLEX` expressions), including ordinal operators (`"<"`, `"<="`, `">"`, `">="`) which are not predefined since type `IMAGINARY` is private. The intent is to mirror the treatment of real numbers as much as possible; however, there is variation in the result type of operations since the set of imaginary numbers do not form an algebraic field (the set of imaginary numbers is not closed under multiplication). For example,

```
function "*" (LEFT, RIGHT : IMAGINARY) return REAL;
```

has the result type one would mathematically expect, as does

```
function "abs" (RIGHT : IMAGINARY) return REAL;
```

Defining an `"abs"` function with a result type of `IMAGINARY`, analogous to its definition along the real axis, would lead to anomalous results of the type

$$1.0 + \texttt{abs}(1.0 * \texttt{i}) = 1.0 + 1.0 * \texttt{i},$$

$$1.0 + \texttt{abs}(0.0 + 1.0 * \texttt{i}) = 2.0 + 0.0 * \texttt{i}$$

which illustrate the need to maintain consistency between the results of mathematically identical complex and imaginary operations.

The type `IMAGINARY` is private, with its full type declaration revealing it as derived from type `REAL`, for at least two reasons:

- The result type of the imaginary multiplication and division operations is `REAL`; defining `IMAGINARY` to be private suppresses the derivation of these operations with an incorrect (`IMAGINARY`) result type.

- Implicit conversion of real literal values to type `IMAGINARY` is suppressed, which allows overload resolution to work correctly by avoiding various ambiguous expressions.

The difficulties noted above could be overcome by defining `IMAGINARY` as a one-component, visible record type and indeed proposals made early in the standardization process for ISO/IEC 8652:1995 included such a definition with a

component named VALUE (IM was ruled out because it could be confused with the COMPLEX component of the same name). However, this definition makes it somewhat awkward to represent and access imaginary values; positional aggregates would be decidedly non-intuitive and named aggregates would seem redundant.

Once a type IMAGINARY is defined, it might appear natural to alter the COMPLEX type definition so that its IM component is of type IMAGINARY. If this design change was implemented, the canonical aggregate representation of a complex number would no longer be available; for example, one would write $(1.0, 1.0 * \mathtt{i})$ instead of $(1.0, 1.0)$. This would adversely affect current applications and data sets which rely upon this canonical form, perhaps hampering the acceptance of this International Standard.

F.8 The use of operator notation versus function notation

Because of the convenience to the user of operator notation, this has been chosen whenever possible rather than the use of function notation. In particular, overloading of the same operator token has been used extensively for related vector and matrix operations – but such overloading can require the use of type qualification to resolve possible ambiguities when such operators are used in combination.

For example, consider vectors X and Y and a matrix A, then the expression (X * Y) * A is ambiguous. The relevant operations are

```
a) function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL;
b) function "*" (LEFT, RIGHT : REAL_VECTOR) return REAL_MATRIX;
c) function "*" (LEFT : REAL; RIGHT : REAL_MATRIX) return REAL_MATRIX;
d) function "*" (LEFT, RIGHT : REAL_MATRIX) return REAL_MATRIX;
```

with the formal parameters LEFT, RIGHT replaced by the actual scalars, vectors and matrices in the appropriate way. Both a and c or b and d can be combined, and qualification is required to resolve this ambiguity, i.e., either REAL'(X*Y)*A or REAL_MATRIX'(X*Y)*A can be specified. Such qualification can be avoided if, rather than overloading "*" so comprehensively, function notation with distinct names were used instead:

```
a1) function INNER_PRODUCT (X, Y : REAL_VECTOR) return REAL;
b1) function OUTER_PRODUCT (X, Y : REAL_VECTOR) return REAL_MATRIX;
```

The expressions then become INNER_PRODUCT (X * Y) * A or OUTER_PRODUCT (X * Y) * A. No qualification is needed to distinguish the two expressions – the distinct function names are sufficient. Thus qualification can be avoided, but at the expense of function notation.

In many cases, the operator notation is the most convenient – only when ambiguity arises is it necessary to qualify intermediate results, and in such cases there is little difference in convenience between the two forms. Overall, therefore, operator notation was chosen as the most convenient.

F.9 Complex arithmetic

These packages are intended to define portable, accurate and robust implementations of complex arithmetic. The inclusion of accuracy requirements (discussed in clause F.10) not only controls the accuracy of the operations +, -, *, / and ** for complex types, but also imposes an implicit requirement to avoid intermediate overflow in the calculation of component parts when the final complex result does not itself overflow (in its components). Hence the simple implementation of z_1/z_2, where $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$, by the formula

$$z_1/z_2 = (x_1 + iy_1)(x_2 - iy_2)/({x_2}^2 + {y_2}^2)$$

is not sufficiently robust since ${x_2}^2$ or ${y_2}^2$ could overflow prematurely, although the accuracy requirements would impose a result within certain bounds of the true mathematical result.

The performance and accuracy of complex arithmetic is fundamentally dependent on the form of representation of the complex value, i.e., whether a cartesian or polar representation is chosen. The algorithms for complex arithmetic

in polar form are very different from their cartesian counterparts, and often involve the use of elementary mathematical functions, which can, at best, only approximate the result in finite arithmetic. This International Standard requires that *a cartesian representation be used throughout*, in order that reasonable accuracy requirements can be given. Functionality for conversion to and from polar form is included by the functions `ARGUMENT`, `MODULUS` and `COMPOSE_FROM_POLAR`, but the internal form of the complex value is required to be cartesian as is the form of algorithms implementing the complex operations `+`, `-`, `*`, `/` and `**`.

Having chosen a cartesian representation for the complex types, it is no longer necessary to define such types as private (which might be the case if both cartesian and polar forms were allowed). By explicitly defining complex types as records (with real and imaginary components), a user is able to initialize complex types (particularly arrays) directly using aggregates (although complex types imported to generic packages will still require the use of the composition functions). The visible record structure also prevents additional components from being added to the representation.

F.10 Accuracy requirements

A number of ways can be devised for measuring the error in a computed complex value. For a true result $\zeta = \alpha + i\beta$ and a calculated result $z = x + iy$, three bounds on the error were considered:

— the relative error in a component

$$|\alpha - x| \leq n\epsilon|\alpha|$$
$$|\beta - y| \leq p\epsilon|\beta|$$

— the box error in a component

$$|\alpha - x| \leq m\epsilon \max(|\alpha|, |\beta|)$$
$$|\beta - y| \leq m\epsilon \max(|\alpha|, |\beta|)$$

— the circular error in a component

$$|\alpha - x| \leq c\epsilon|\zeta|$$
$$|\beta - y| \leq c\epsilon|\zeta|$$

where n, p, m and c are small, ϵ is the machine precision, and the appropriate component(s) α, β are nonzero.

The use of relative error measures is uniform with other standards. Where possible, the tight bound on the relative error is desirable, but for some operations cancellation may occur (e.g. complex multiplication) and the bound needs to be relaxed somewhat. Both box error and circular error serve this purpose, and the relationship between them is obvious. Box error is more straightforward to calculate (in test programs) and was therefore chosen.

This International Standard specifies error requirements for all complex scalar operations (but note the two special cases discussed below), and for subprograms where no arithmetic operation should be performed, e.g., `COMPOSE_FROM_CARTESIAN`, an exact result is required. Error requirements may differ on the real and imaginary components of the result of a single complex operation; this is to ensure (of an implementation) the most stringent accuracy that is feasibly obtainable. Whenever possible, this International Standard appeals to the accuracy requirements of the indicated operation for real arithmetic, and never imposes an accuracy requirement more stringent than is defined in ISO/IEC 8652:1987.

There are two instances in which the relative error requirement on a complex scalar operation is either relaxed or removed:

a) For the `COMPOSE_FROM_POLAR` function with natural cycle (`CYCLE` parameter omitted), degraded accuracy is allowed when |`ARGUMENT`| is greater than some documented implementation-dependent threshold, which shall be not less than

$$\texttt{REAL'MACHINE\_RADIX}^{\lfloor \texttt{REAL'MACHINE\_MANTISSA}/2 \rfloor}.$$

This latitude is given to implementations because of the difficulty in accurately reducing the `ARGUMENT` parameter to its principal value when `ARGUMENT` is large in magnitude relative to the length of the natural cycle.

b) For complex exponentiation, there is no relative error requirement for implementations which calculate the result by first converting the complex base to polar form, exponentiating its modulus and multiplying the argument by the integer exponent, and reconverting to a cartesian representation. This latitude is given to implementations because this International Standard makes no provision for the accuracy of operations performed on complex numbers represented in polar form.

It is instructive to study the subprogram results permitted by the accuracy requirements when a component of the true result is near zero and a maximum relative error requirement is specified for that component (a similar analysis applies for the component of a result which has a maximum box error requirement). Let the maximum relative error for a certain component of a subprogram result be given by $n \cdot$`REAL'BASE'EPSILON` (where $n > 1.0$) and consider a sequence of true results for which this component moves from $n \cdot$ `REAL'BASE'EPSILON` towards zero. For values of this result component less than a certain threshold dependent upon n, the maximum relative error allows computed component values to be less than `REAL'SAFE_SMALL` which, as this threshold is crossed, immediately widens the acceptable range of component values to include all values between zero and `REAL'SAFE_SMALL`. A second threshold (also dependent upon n) is crossed when the maximum relative error requirement permits computed component values of either sign. Thus, if the true result is close enough to one of the axes in the complex plane, a subprogram is permitted to return a result close to but on the wrong side of the axis, thereby placing the computed result in an adjacent quadrant.

Having defined measures as above on the errors in complex values, these measures can be used for vector and matrix operations whenever the operation is defined in terms of operations on its elements. However, some vector and matrix operations involve scalar products (i.e., a summation of products), where destructive cancellation is likely. Various techniques exist for either minimizing or even eliminating this cancellation, but all can have significant computational cost. This International Standard therefore does not specify error requirements for such scalar products; this allows implementations to choose an appropriate balance between computational cost and accuracy (but the choice shall be clearly documented).

F.11 Naming and renaming conventions

A scheme for the naming of parameters in a uniform way has been adopted in this International Standard. Precedents set by other standard packages have guided the choices made. For example, `GENERIC_COMPLEX_TYPES`, `GENERIC_REAL_ARRAYS` and `GENERIC_COMPLEX_ARRAYS` utilize the same parameter naming convention utilized in package `STANDARD`, namely, `RIGHT` for unary operators and `LEFT`, `RIGHT` for binary operators. Similarly, `COMPLEX_IO` adopts the parameter naming conventions of package `TEXT_IO`.

Early drafts of this standard did not conform to these conventions, and the choice to use `LEFT`, `RIGHT` was not a popular one (among the working group members responsible for drafting this standard). It was felt that they convey no useful information about the parameters, and would not be widely accepted in the numerics community. Earlier parameter naming conventions utilized single-character names (for brevity) that were based on the object represented, e.g., `V,W` for vectors. However, once again the need to be consistent with decisions made during the standardization process for ISO/IEC 8652:1995 overruled the objections.

Renamings of certain operations and types were considered, for example `RE` renamed to `REAL_PART` for selecting the real component-part of a complex number. Such additions were rejected because they did not add to the functionality of the packages, and the case could be made for many such renamings – the choice to include or reject some would be a matter of taste. Renamings of operations can always be declared external to these packages to provide names more familiar to a specific user group.

There are only five concessions to renaming provided in these packages:

– The inclusion of `"abs"` as a renaming of `MODULUS` for all types that `MODULUS` is defined, namely `COMPLEX`, `COMPLEX_VECTOR` and `COMPLEX_MATRIX`. `MODULUS` and `ARGUMENT` provide the polar components of a complex number; it just happens that the absolute value `"abs"` of a complex value is also its `MODULUS` – and the alternative forms were retained for uniformity with the real case. Note that for type `IMAGINARY` there is an `"abs"` operation but no

MODULUS; this mirrors the situation for real types and suppresses the inclination to pair MODULUS with an ARGUMENT operation for type IMAGINARY which would have little utility.

— The inclusion of CONJUGATE as a renaming of the unary "-" operation with operand of type IMAGINARY. Conjugation and negation are mathematically equivalent when applied to numbers lying on the imaginary axis and the alternative form was retained for uniformity with the COMPLEX type.

— The ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR exception is renamed to ARGUMENT_ERROR in GENERIC_COMPLEX_TYPES and GENERIC_COMPLEX_ARRAYS to export direct visibility to this exception from an application using a single instantiation of one of these generic packages.

— The ARRAY_EXCEPTIONS.ARRAY_INDEX_ERROR exception is renamed to ARRAY_INDEX_ERROR in GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS to export direct visibility to this exception from an application using a single instantiation of one of these generic packages.

— The inclusion of both i and j to represent the imaginary unit value. The concession to include j was made primarily to accommodate the engineering community, which cannot use i since it typically represents an electrical current value. Note however that j is independently defined and not a renaming of i (renaming of constants is not allowed in ISO/IEC 8652:1987).

F.12 Genericity

The Ada language defines two facilities by which packages (with types and basic operations) can be defined so that similar packages can be produced (possibly with constraints) namely, genericity and derivation. Derivation is inappropriate for composite types (e.g., the complex types or the vector and matrix types). Given a vector type:

```
type REAL_VECTOR is array (INTEGER range <>) of REAL;
```

a derived type NEW_VECTOR with

```
type NEW_VECTOR is new REAL_VECTOR;
```

generates a new array type, but with the old REAL components. Hence a user cannot derive a new set of scalar and array types for which the new scalar type and the new array types are consistent.

There remains the question of whether the ability to produce similar packages is needed. Without such an ability, distinct types to represent different entities (e.g. distance, time, velocity, etc.) cannot be defined a significant loss of functionality. On the other hand, many users will only wish to use a set of predefined standard types throughout their applications in order to share development across applications. Hence, both a generic package and non-generic packages defining standard types have been defined.

Unlike ISO/IEC 11430 in which a generic package is given and implementations *may* provide instances, this International Standard *requires* both the generic packages (defining types and basic operations, i.e., GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS) and their equivalent non-generic packages (defining the standard types and operations, e.g. COMPLEX_TYPES and REAL_ARRAYS, etc.) to be provided. The reason that the non-generic packages are required is to ensure that standard types are always provided by an implementation of this standard, and to avoid one group of users generating their own types which would be different types from those generated by another group of users.

Each of the non-generic packages must export identical type names, e.g., REAL_VECTOR or COMPLEX, regardless of the precision of the floating-point type utilized. This approach enables a user to easily switch precisions in an application; however, if two precisions are used simultaneously, either fully qualified type specifications or renamings of types must be used.