Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

G.T. Symm, B.A. Wichmann, J. Kok, D.T. Winter

Guidelines for the design of large modular scientific libraries in ADA
Final report for the Commission of the European Communities

Department of Numerical Mathematics Note NM-N8401 March

GUIDELINES FOR THE DESIGN OF LARGE MODULAR SCIENTIFIC LIBRARIES IN ADA
Final report for the Commission of the European Communities

G.T. SYMM, B.A. WICHMANN
Division of Information Technology and Computing
National Physical Laboratory, Teddington, Middlesex, England

J. KOK, D.T. WINTER
Centre for Mathematics and Computer Science, Amsterdam

The new programming language Ada has been designed primarily for real-time, embedded computer applications development. However, it is envisaged that it will also be widely used in large-scale scientific computation. Several features of the language require careful consideration if large portable and modular scientific algorithms libraries are to be implemented succesfully. Accordingly, in this report we attempt to identify the problems associated with the overall design and implementation of such libraries in Ada and make recommendations for their solution. The problem areas considered are precision, basic mathematical functions, composite data types, information passing, error handling, working-space organisation and real-time environment.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 69D49, 65-Ø4.
KEY WORDS & PHRASES: Ada programming language, scientific software.

Note NM-N8401
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

CONTENTS

1. INTRODUCTION

As is well known the new programming language Ada (ANSI/MIL-STD 1815 A, 1983) has been designed primarily for real-time, embedded computer applications development. However, in view of the scale of effort that has been invested in its design, it is envisaged that it will also be widely used in other areas, including the important one of large-scale scientific computation. Preliminary evaluations of the suitability of Ada for this purpose (Cox and Hammarling, 1980; Hammarling and Wichmann, 1982) have indicated that several features of the language require careful consideration if large portable and modular scientific algorithms libraries are to be implemented successfully. Accordingly, we here attempt to identify the problems associated with the overall design and implementation of such libraries in Ada and make recommendations for their solution.

Our main object is to draw up a set of practical guidelines for the benefit of those wishing to develop large scientific libraries in Ada (and hence, indirectly, also for those concerned with the construction of specialised applications packages). We have in mind here libraries comparable with the NAG FORTRAN Library (Ford et al., 1979), the NUMAL Library in Algol 60 (Hemker, 1981) and the NAG Algol 68 Library (NAG, 1983). We believe the provision of guidelines to be essential if the risk of incompatibility is to be avoided.

In preparing this report we have taken full account of existing guidelines (Nissen and Wallis, 1984) concerning portability and style in Ada. The latter, which originated from the work of the Portability Subgroup of Ada-Europe, are intended to assist programmers in the design and preparation of portable Ada software. The present guidelines, on the other hand, seek to ensure that individually compiled modules of large scientific libraries can retain this portability while also being compatible with each other and with users' programs. (Incidentally, the need for portability rules out the possibility of simply providing interfaces with existing libraries in other languages, though it is recognised that mixed-language programming may well be necessary during the early stages of a transition towards the adoption of Ada. This and other topics, which are not covered by the present project but which clearly require further study, are listed in an appendix (Appendix G) to this report.)

Throughout this report, references to the Language Reference Manual (ANSI/MIL-STD 1815 A, 1983) are abbreviated to LRM xxx, where xxx specifies the precise location within the manual. Further details of the manual and all other references are gathered together, in alphabetical order of author, at the end of the report.

The plan of our report is as follows. In Chapter 2, we outline the basic problems which face designers of large modular scientific libraries in Ada. In Chapters 3 to 9, we discuss each problem area in turn, deriving solutions to the problems through examples of Ada code. We then summarise our recommendations in Chapter 10.

Some examples of program code are included in Appendices C, D and E, in order to avoid unnecessary interruptions in the main text, while in further appendices we summarise:

- features (assumed or desired) of a target implementation, together with what we consider to be deficiencies in the Ada language as far as scientific computing is concerned (Appendix A),

- the proposed contents of basic packages for scientific computation (Appendix B)

- the IEC floating-point standard and its relationship with Ada (Appendix F) and

- topics which we consider to require further study, such as interfaces with other languages, as mentioned above (Appendix G).

We add here that the topics considered in these appendices have not been relegated through lack of importance but are, rather, gathered there for convenience of reference. Indeed, some of the topics included are of considerable importance and we recommend, in particular, that implementors should study Appendix A while anyone interested in Ada numerics might find Appendix G useful.

Note that, while preparing this report, we have not had regular access to an Ada compiler but most of the Ada code included in the text has been verified by means of a syntax checker. In relation to this, a sequence of statements is sometimes indicated by a single statement, in the form of a procedure call, describing the action involved, e.g.

```
SIMPLE_APPROXIMATION;
```

rather than by a comment:

```
-- sequence of statements
```

since the latter is not acceptable to the syntax checker where at least one statement is necessary. On the other hand, the notation "...", which is never acceptable to the syntax checker, is used occasionally, as in the Language Reference Manual, to cover an obvious gap in the code.

Acknowledgements

We wish to thank the Commission of the European Communities for supporting this work and the reviewers, appointed by the Commission, for their constructive criticisms of our earlier interim reports.

We gratefully acknowledge the help of Hans-Stephan Jansohn (GMD Forschungsstelle Karlsruhe) with the testing of a sorting procedure (Chapter 9) and Andreas Schneider (ROLM Corporation, Mühlheim) with the compilation of a least-squares package (Appendix E).

We also thank our colleagues Sven Hammarling (now with NAG Limited, Oxford), Maurice Cox and Geoff Miller, NPL, and Piet Hemker, Mathematisch Centrum, for their assistance with the work and their contributions to this report.

2. THE PROBLEMS

In this chapter we outline the problems, as we see them, which face designers of large modular scientific libraries in Ada.

a) Precision

The first and most fundamental problem in the design of large scientific libraries in Ada is concerned with precision.

Every object in the language has a type (or, more specifically, has a value of some type), where a type is characterised by a set of values and a set of operations applicable to those values (LRM 3.2, 3.3). In particular, for floating-point computation, the language includes at least one predefined type FLOAT. An implementation may also have predefined types such as SHORT_FLOAT and LONG_FLOAT, which have (substantially) less and more accuracy, respectively, than FLOAT (LRM 3.5.7). These and all other predefined identifiers are contained in the package STANDARD to which the user may be assumed to have access (LRM C). The user is also permitted to declare his own floating-point types, e.g.

```
type REAL is digits D;
```

where D is any number of decimal digits supported by the implementation, i.e. any positive integer not exceeding SYSTEM.MAX_DIGITS (LRM 13.7.1(4)). In this case, the type REAL is derived by the implementation from one of the predefined types which has at least D digits of precision. Note that (from LRM 3.5.7(12)) there is always one predefined floating-point type (call it LONGEST_FLOAT) which corresponds to the highest possible value of D, i.e. such that the attribute LONGEST_FLOAT'DIGITS (LRM 3.5.8) equals SYSTEM.MAX_DIGITS. Note also that explicit type conversions are allowed between closely related types (LRM 4.6); for example, REAL(2*J) represents the integer expression 2*J in the floating-point form of the type REAL.

The user must decide how best to use these facilities and, since the rules of the language require that types must match on a function or procedure call (LRM 6.4.1(1)), the choices are particularly important in the design of large numerical libraries. In such libraries, separately compiled program units must be compatible with each other, with units of other libraries and with users' units. Also intercommunication between units, of any kind, should involve as little recompilation as possible. In Ada a compilation unit (LRM 10.1) can be a subprogram (i.e. procedure or function) declaration or body, a package declaration or body, a generic declaration or a generic instantiation. Alternatively, it can be a subunit (LRM 10.2), which is the separate body of a subprogram, package or task declared within another compilation unit. In either case it may be preceded by a context clause.

The main problem arises from the strong type-checking rules of the language whereby any two type definitions specify distinct types even if their descriptions are identical (LRM 3.3.1(8)). For example, if

```
type REALA is digits 6;
type REALB is digits 6;
A : REALA;
B : REALB;
```

then A and B are of different types. Similarly, if one compilation unit declares

```
type REAL is digits 10;
X : REAL;
```

while another declares

```
type REAL is digits 10;
Y : REAL;
```

then X and Y are of different types and the two units are incompatible.

Ways around this difficulty and other problems associated with precision are discussed in Chapter 3 of these Guidelines.

b) Basic functions

The basic mathematical functions, which, in Fortran and other languages, are denoted by SQRT, EXP, SIN, etc., are not (apart from ABS, which is covered by the operator **abs**, represented by a reserved word) included in the Ada language and must therefore be provided in a library package (LRM 7). Ideally, such a package would already be available, in some universally accepted form, to the designer of large scientific libraries. Unfortunately, although some proposals have been made (e.g. Firth, 1982; Whitaker and Eicholtz, 1982), this is not yet the case and we must therefore design our own package. In so doing, we hope that we may influence the ultimate design of a universal package in such a way that it is compatible with the remainder of our guidelines.

If all computations could be carried out successfully in terms of the predefined type FLOAT, the required package might have a specification of the form:

```
package MATH_FUNCTIONS is

   function SQRT(X : FLOAT) return FLOAT;
   function EXP(X : FLOAT) return FLOAT;
   function SIN(X : FLOAT) return FLOAT;

   -- etc.

end MATH_FUNCTIONS;
```

In practice, however, types SHORT_FLOAT, LONG_FLOAT and, more generally, user-defined real types must also be accommodated. How this may be achieved is clearly dependent upon the way in which the precision problem is solved (in Chapter 3 of these Guidelines).

Problems relating to the package MATH_FUNCTIONS and its contents are discussed in Chapter 4.

c) Composite data types

Composite data types, such as COMPLEX, VECTOR and MATRIX, whose values consist of component values (LRM 3.3(2)), are not predefined in the Ada language and must therefore be provided in appropriate packages.

For example, COMPLEX may be provided as a record type (LRM 3.7), with its associated operators (cf. Wichmann, 1984), in a package of the form:

```
package COMPLEX_OPERATORS is

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

   function "+"(X : COMPLEX) return COMPLEX;
   function "-"(X : COMPLEX) return COMPLEX;
   function "abs"(X : COMPLEX) return REAL;
   function ARG(X : COMPLEX) return REAL;
   function "+"(X,Y : COMPLEX) return COMPLEX;
   function "-"(X,Y : COMPLEX) return COMPLEX;
   function "*"(X,Y : COMPLEX) return COMPLEX;
   function "/"(X,Y : COMPLEX) return COMPLEX;
   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX;

end COMPLEX_OPERATORS; -- specification
```

where it is assumed that a type REAL is already available. If it is further assumed that the basic mathematical functions, in the package MATH_FUNCTIONS, are applicable to such REAL variables, then the package body (LRM 7.3), corresponding to the above specification, could take the form:

```
with MATH_FUNCTIONS;
package body COMPLEX_OPERATORS is

   function "+"(X : COMPLEX) return COMPLEX is
   begin
      return X;
   end "+";

   function "-"(X : COMPLEX) return COMPLEX is
   begin
      return (- X.RE, - X.IM);
   end "-";
```

```
      function "abs"(X : COMPLEX) return REAL is
         A,B : REAL;
      begin
         if abs X.RE > abs X.IM then
            A := abs X.RE;
            B := abs X.IM;
         else
            A := abs X.IM;
            B := abs X.RE;
         end if;
         if A > 0.0 then
            return A * MATH_FUNCTIONS.SQRT(1.0 + (B/A)**2);
         else
            return 0.0;
         end if;
      end "abs";

      -- etc.

   end COMPLEX_OPERATORS; -- body
```

Similar packages may be provided for interval arithmetic, using a record type to describe an interval, e.g.

```
   type INTERVAL is
      record
         MIN,MEAN,MAX : REAL;
      end record;
```

but, since these would give rise to similar design problems, they are not considered in detail here.

Other abstract floating-point types, such as representations of multiple length variables as record types, are not considered in detail here either, for several reasons. All manipulations of such variables would have to be done by software and would therefore tend to be extremely slow and inefficient. Such variables could not feature in type conversions and the basic MATH_FUNCTIONS library would not be available to their users. The design of libraries to accommodate such variables is considered to be outside the scope of the present project. Nevertheless, the design of such packages would be useful, after the basic structure of scientific libraries has been established, and is recommended for further study (Appendix G).

Since vectors and matrices are useful in their own right, we consider that these are best packaged separately from their associated operators. Appropriate packages, together with packages for complex arithmetic, are discussed in detail in Chapter 5.

d) Information passing

The Ada language does not define the implementation method for passing parameters of array, record and task types; such parameters may be passed either by copying or by reference (LRM 6.2(7)). A program whose action depends upon which of these methods is used is erroneous since it will have indeterminate properties. Although this freedom of implementation is needed in certain special cases, it is essential, for efficiency in high-quality scientific libraries, that large vectors and matrices should not be copied unnecessarily.

Ada does not permit functions or procedures as parameters in procedure calls but such information may be passed by means of generics (LRM 12) or by means of "reverse communication" (Hammarling and Wichmann, 1982).

As an example of the former, a simple procedure for numerical integration (quadrature) of a function F of a single real variable X, between fixed limits of integration A and B, may have a declaration:

```
generic
   with function F(X : REAL) return REAL;
procedure QUAD(A,B : in REAL; R : out REAL);
```

Then integration of a specific function F1 with declaration:

```
function F1(X : REAL) return REAL;
```

may be achieved by means of an instantiation (LRM 12.3) of the generic procedure:

```
procedure QUAD_F1 is new QUAD(F1);
```

followed by a procedure call:

```
QUAD_F1(A,B,R);
```

Issues raised by such use of generics, the alternative of reverse communication, and other problems relating to procedure parameters are discussed in Chapter 6.

e) Error handling

The Ada concept of exceptions (LRM 11) provides an error handling mechanism which must be fully explored. An exception is an error or other exceptional situation which arises during program execution. Detecting this situation and drawing attention to it, abandoning normal program execution in the process, is called "raising the exception". Executing some actions, in response to the raising of an exception, is called "handling the exception".

Exception names, other than those of a few predefined exceptions such as CONSTRAINT_ERROR and NUMERIC_ERROR, are introduced by exception declarations (LRM 11.1), e.g.

```
SINGULAR : exception;
```

Exceptions can be raised by raise statements (LRM 11.3) or by other statements or operations which propagate the exceptions (LRM 11.4.2(8)). When an exception arises, control can be transferred to a user-provided exception handler (LRM 11.2) at the end of a frame, i.e. at the end of a block statement or at the end of the body of a subprogram, package, task unit or generic unit. This handler acts as a substitute for the remainder of that frame; so that, for example, a handler within a function body may execute a return statement on its behalf.

The handling of an exception raised during execution of a sequence of statements depends on the innermost frame or accept statement that encloses that sequence of statements (LRM 11.4.1). However, if an exception is raised during the elaboration of the declarative part of a frame, or during the elaboration of a package or task declaration, this elaboration is abandoned (LRM 11.4.2). In this case, if the frame is a task body, the task becomes completed and the exception TASKING_ERROR is raised at the point of activation of the task (LRM 9.3). Otherwise, the exception is propagated, if possible, or the program/task is abandoned. In particular, if an exception is raised during the elaboration of the declarative part of a library unit, the execution of the main program is abandoned. It follows that one may sometimes wish to avoid the raising of exceptions in the declarative part of a library unit, possibly by enclosing the necessary declarations in an inner block so that exceptions due to errors in input parameters can be handled in the surrounding body.

Such issues and more general questions regarding error handling in Ada are discussed in Chapter 7.

f) Working-space organisation

In general, working-space must be efficiently organised. In Ada, this organisation may depend upon:

- the types used for claiming large storage areas (e.g. arrays, records or list and tree structures),

- the parameter-passing mechanism (subprograms might make copies of all parameters passed) and other situations where extra copies might be made, and

- the architecture of the machine (e.g. on a machine with paging, for efficiency an algorithm should process contiguous components of arrays and, for two-dimensional arrays, contiguity depends upon the way in which the arrays are stored).

Programs might be made to run more efficiently by using information about the working-space (e.g. the size for different types). In Ada, this information is provided by attributes and by the package SYSTEM.

Storage which is no longer required may be reclaimed, to be used again, by a garbage collector. However, in Ada, the existence of a garbage collector is implementation-dependent and software which relies upon it should therefore make this clear. In any case, the programmer may prefer to do his own tidying-up, e.g. in a real-time program where he may achieve better timing control by so doing (Barnes, 1982, p.253). For access types, he may use the predefined generic procedure UNCHECKED_DEALLOCATION which has the specification:

```
generic
   type OBJECT is limited private;
   type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

with a typical instantiation of the form:

```
procedure FREE is new
   UNCHECKED_DEALLOCATION(object_type_name, access_type_name);
```

All aspects of working-space organisation are discussed in Chapter 8.

g) Real-time environment

Ada has been specifically designed for real-time computation and the needs of real-time users must therefore be taken into account. For example, it may be required that a program should continue to run in all circumstances - no matter what errors may arise during its execution. This may be achieved by the inclusion of an exception handler of the form:

```
when others =>
   -- sequence of statements
```

where the sequence of statements carries out appropriate remedial action to enable the computation to continue in the event of any unforeseen error arising.

In real-time situations, such as process control, a result of a computation may be required at a particular time; the precise response moment may not be known in advance but when it arrives the answer must be immediate. This requirement can affect the choice of an algorithm or the way in which it is implemented. For example, if an iterative process consists of several parts (which may run concurrently), of which the results are normally added together at the end of the process (when each part has reached a specified accuracy), it would be preferable in this case to keep a running total (with an estimate of its accuracy) to be used in the event of a rendezvous being met before the iteration is complete.

Issues such as these are discussed in Chapter 9.

3. PRECISION

In this chapter we consider the problems concerned with the accuracy of real types in Ada, introduced in section (a) of Chapter 2. Our discussion takes the form of a series of notes, labelled alphabetically for easy reference.

a) Hardware types

The predefined types FLOAT, SHORT_FLOAT and LONG_FLOAT correspond to the hardware. Since one view of numerical packages is to consider them as additions to the hardware, one might conclude that all library software should be written in terms of these predefined types. However, this would not be a good idea for reasons of portability. The language does not state any specific accuracy for FLOAT and, since this is the name assigned if there is only one floating-point type, the actual accuracy is likely to vary considerably. On some machines LONG_FLOAT would be more appropriate than FLOAT for library use, while on others SHORT_FLOAT might suffice. Hence the use of the predefined types cannot be recommended in general. (Since the names FLOAT, SHORT_FLOAT and LONG_FLOAT are not reserved in Ada, one could possibly redeclare them to achieve the portability that would otherwise be lacking, but this idea is rejected since it would be rather misleading.)

b) Derived types

It may appear that the type compatibility rules make it very difficult to write any portable library software at all. Yet, if LONG_FLOAT is available as well as FLOAT (see section (b) of Appendix A), one can certainly imitate standard FORTRAN practice by declaring

```
type REAL is new FLOAT;
type DOUBLE is new LONG_FLOAT;
```

and writing all program units in terms of these derived types (LRM 3.4). Alternatively, if SHORT_FLOAT is available, one may declare

```
type REAL is new SHORT_FLOAT;
type DOUBLE is new FLOAT;
```

and use these derived types in all program units. Hence, by introducing the same names, REAL and DOUBLE, in each case, we have a possible solution to the problem of providing portable software. This solution is, of course, restricted to implementations which support at least two predefined floating-point types and is based upon the assumption (which may not be acceptable to many) that two levels of precision are sufficient for library purposes.

c) Attributes

In Ada, most of the properties of a real type can be accessed by its attributes, which are defined as part of the language (LRM 3.5.8, 3.5.10). This enables one, when writing software, to anticipate the problems of moving code to another machine. For instance, an approximation may be known to be good for 10 digits but not more, in which case one can write

```
if REAL'DIGITS <= 10 then
   SIMPLE_APPROXIMATION;
else
   MORE_COMPLEX_CASE;
end if;
```

where, if the static condition is TRUE, the code for the MORE_COMPLEX_CASE (though it must be valid) need not be compiled (cf. section (e) below). Careful use of these facilities permits one to write code which is robust and numerically correct across almost all conceivable machines. In this, one is aided by the fact that the numerical properties of real types are well defined in terms of model numbers (LRM 4.5.7), although these have their limitations (Wallis, 1983). See also section (d) of Appendix A and Appendix F, where the Ada model is compared with the IEC floating-point standard (IEC, 1982).

d) User-defined types

The contrary view to that expressed in section (a) above is that of the applications programmer who wishes (not unnaturally) to ignore details of the specific hardware in use. His concern is to program in a portable manner knowing that, for example, 10 digits of accuracy will suffice for his particular application. He therefore declares

```
type MY_REAL is digits 10;
```

whereupon the problem is that, since MY_REAL is dependent upon the application, numerical library packages (written in terms of a different real type) cannot be called directly.

One approach to this problem is the use of generics, as in the input-output system (LRM 14.3). There, for example, the output procedure PUT may be made available for MY_REAL, as declared above, by instantiating the generic package FLOAT_IO, which is inside the package TEXT_IO, thus:

```
with TEXT_IO;
procedure MAIN is
   ...
   package MY_IO is new TEXT_IO.FLOAT_IO(MY_REAL); use MY_IO;
   X : MY_REAL;
begin
   ...
   PUT(X);
   ...
end MAIN;
```

It is assumed here that the declaration of MY_REAL either lies within the procedure MAIN, before its use in the instantiation, or is visible there through a previous context clause (cf. section (f) below).

As a consequence of the need to instantiate the generic, this solution has some disadvantages. It is very unlikely that the instantiation of a generic will be a cheap operation for the compiler. At worst, it could amount to an overhead comparable with the recompilation of the instantiated body. With a large mathematical library, such an overhead might not be acceptable. Moreover, the body of the instantiated package could need to call other packages which would themselves need to be instantiated. The compiler overhead for such an activity is likely to be even greater than that for the ordinary text.

In practice, perhaps such generic packages will be precompiled (see section (a) of Appendix A) for each of the relevant predefined types, such as the hardware types of section (a) above, and the appropriate version selected at instantiation. However, the conclusion here is that generics need to be used with care, at least within the context of a large library. The advantage of generics is that they do allow one to write a subprogram or package for any accuracy and let the user select the appropriate accuracy. Thus they are ideal for the user who is prepared to tailor a system to his own specific requirements.

e) Use of generics

On the assumption that some use is made of generics, subprograms or packages can call any low-level routines that may be provided for the hardware types by means of tests on the attributes and conversions. A simple example might be

```
generic
   type REAL is digits <>;
function SQRT(X : REAL) return REAL;         -- specification

function SQRT(X : REAL) return REAL is       -- body
begin
   if REAL'DIGITS <= FLOAT'DIGITS then
      return REAL(SQRT(FLOAT(X)));
   else
      return REAL(SQRT(LONG_FLOAT(X)));
   end if;
end SQRT;
```

Note the use of explicit conversion and the two distinct calls of the overloaded function SQRT. Of course, for a specific instantiation of this generic, a compiler should optimise the code so that no condition is tested or code produced for the other leg. Note, however, that the condition involving REAL'DIGITS is no longer static (cf. section (c) above) when REAL is a generic actual parameter (LRM 4.9, 12.1(12)).

Unfortunately, the code given here is not fully portable, being again restricted to implementations which support LONG_FLOAT as well as FLOAT. Moreover, no allowance is made for the possibility that REAL'DIGITS > LONG_FLOAT'DIGITS for which an exception could be raised (see Chapter 7).

f) Library design

One conclusion from the arguments above is that, for a large library, the use of existing subroutines by new routines necessitates the use of a standard set of real types. Such standard types may be collected together in one package:

```
   package REAL_TYPES is
      type REAL is digits D;          -- an implementation choice
      ...
   end REAL_TYPES;
```

Then each library package may operate in terms of these, for example:

```
   with REAL_TYPES; use REAL_TYPES;
   package LIBRARY_PACK is                     -- specification

      function FUN(X : REAL) return REAL;

      -- other functions, etc.

   end LIBRARY_PACK;
```

However, if the corresponding package body is written for only the standard types, with their specified accuracy, this approach lacks generality. There may well be a need for functions, such as FUN, of higher accuracy and the textual bodies of these functions will often admit such accuracy.

It is preferable therefore to implement LIBRARY_PACK by means of a generic package:

```
   generic
      type REAL is digits <>;
   package GENERIC_LIBRARY_PACK is

      function FUN(X : REAL) return REAL;

      -- other functions, etc.

   end GENERIC_LIBRARY_PACK; -- specification
```

The body of this package, written for any (sufficiently high) accuracy, takes the form:

```
   package body GENERIC_LIBRARY_PACK is

      function FUN(X : REAL) return REAL is
         ...

      -- other functions, etc.

   end GENERIC_LIBRARY_PACK; -- body
```

Then the library package specification above may be replaced by the instantiation:

```
   package LIBRARY_PACK is new GENERIC_LIBRARY_PACK(REAL);
```

in which case:

```
   use LIBRARY_PACK;
```

permits one to call, for example, FUN(X) for X : REAL.

At the same time, this approach allows a sophisticated user, who is not satisfied with the package REAL_TYPES, to declare his own real type and to call the library package for this type:

```
with GENERIC_LIBRARY_PACK;
procedure MAIN is
   type MY_REAL is digits ...;
   ...
   package MY_LIBRARY_PACK is new GENERIC_LIBRARY_PACK(MY_REAL);
   X,Y : MY_REAL;
begin
   ...
   Y := MY_LIBRARY_PACK.FUN(X);
   ...
end MAIN;
```

This construction is discussed further in the next chapter with reference to the basic mathematical functions.

Note that in some cases it may be very difficult to produce, and highly inefficient to execute, code of arbitrary precision (within the accuracy supported by the target machine). In such cases, the non-generic form of the package, as first described, may be used, with its body specialised to a particular machine precision. The effect of calling an instantiation of the generic form of the package for type REAL could then be simply to call the more efficient non-generic form. An example of this practice is described in section (h) of the next chapter.

Note also that within a program library the simple names of all library units must be distinct identifiers (LRM 10.1(3)). It is important therefore that library designers should all use the same names for basic packages, such as REAL_TYPES. For ease of reference, our proposals for the names of such packages (and their contents) are summarised in Appendix B to this report.

Finally, in this chapter, we note that if the package (GENERIC_)LIBRARY_PACK requires higher precision, than that of type REAL, for internal working (e.g. for the accurate accumulation of inner products), this may be provided by including

```
type LONG_REAL is digits N; -- where N > D
```

in the package REAL_TYPES and making the generic version of the package generic with respect to

```
type LONG_REAL is digits <>;
```

as well as type REAL. However, the language does not guarantee that LONG_REAL is any more accurate than REAL, so a check must be made within any procedure that uses both types, and appropriate action taken (such as the use of a different algorithm or the issue of a warning message if LONG_REAL'DIGITS is no bigger than REAL'DIGITS).

4. BASIC FUNCTIONS

As observed in section (b) of Chapter 2, the basic mathematical functions, which are essential for any serious scientific computation, are not included in the Ada language and so must be provided in a library package. The design of such a package provides an excellent vehicle for illustrating the recommendations of the previous chapter and, in the absence to date of any universally accepted package of mathematical functions, provides a useful source of reference for the remaining chapters of these Guidelines.

In this chapter, therefore, the following problems concerning basic functions are identified and discussed:

- contents of a package of basic mathematical functions,
- naming of basic mathematical functions,
- method of use for user-defined types,
- efficiency of execution,
- calling sequences,
- exceptions,
- package specification,
- practical considerations.

Each of these problems is considered in a separate section.

a) Contents of a package of basic mathematical functions

Although large sets of mathematical functions are sometimes required, we propose that only Square Root and the Elementary Transcendental Functions, as given in Abramowitz and Stegun (1965) but omitting the secant and cosecant functions, should be components of a basic Mathematical Functions package (see section (b) below). By permitting some of these functions to have two arguments, with a default value prescribed for the second, we provide a certain amount of flexibility in their range of application (see section (e)). All other functions can be contained in several packages of Special Mathematical or Statistical Functions.

In the basic package, we also include number declarations for PI and the base of natural logarithms e (here named EXP_1). In the specification, in section (g), we give each of these constants to 35 digits, which we consider to be more than sufficient for most purposes. Note that, in any case, the number of digits in such declarations is ultimately restricted (LRM 2.2(9)) by the limitation of line length to 80 characters, imposed in section 2.2 of the Ada-Europe portability guidelines (Nissen and Wallis, 1984). Note also that to change/extend this precision, once given, would alter the specification of the package and therefore necessitate recompilation of all dependent modules; this, of course, should be avoided.

The use of function calls for these constants, e.g.

```
PI : constant REAL := 4.0*ARCTAN(1.0);
```

is not possible here, since the body of the function ARCTAN, which is declared in the same specification, is not available at the time of elaboration of this declaration. Moreover, the value of PI might be required in the body of ARCTAN itself. Both PI and EXP_1 are definitely required, as default parameter values, in the specifications of other functions in this package (see section (e) below).

The further possibility of actually representing the constants by functions, e.g.

```
function PI return REAL;
```

would avoid the necessity of recompilation of dependent library units when more than 35 digits were required. This might have some merit if the body of the basic package could compute the value of PI to the desired accuracy and store it in a local (invisible) variable to be simply fetched by each function call. However, the feasibility of this approach is debatable when the type REAL is a generic parameter, in which case only operations for this type can be used in the computation. This construction is therefore not recommended here.

It must be mentioned that due to the proposed structure of this Mathematical Functions package, following section (f) of Chapter 3, there is no need for (visible) type declarations in the package (see section (c) below). In our opinion, the package obtained through an instantiation with a floating-point type FPT, chosen by the user, should provide all the basic mathematical functions for this type FPT, each of the form:

```
function MATH_FUNCTION(X : FPT) return FPT;
```

when only a single argument is involved. We reject a construction in which every basic function has its specific types and subtypes, to which a user has to accommodate.

Through each instantiation the user receives a package with the familiar basic functions (as an extension of the set of arithmetic operators) for his chosen floating-point type. In this connection we note that such an instantiation is not necessary if the user-defined type is a derived type (like **type** REAL **is new** FLOAT) and an instantiation of GENERIC_MATH_FUNCTIONS (see section (b)) is already available for the parent type.

The package is not subdivided into smaller local packages, each containing some connected basic functions, e.g. the hyperbolic functions, since this would make calls of these functions too verbose.

We do not propose a separate non-generic version of the basic Mathematical Functions package. We propose instead that the program library should contain at least one standard instantiation of this package with FLOAT (or, more appropriately for scientific computation, the library type REAL) as generic actual parameter. (Note that a particular implementation may, through preference, create such an instantiation from an Ada text by expanding the generic declaration as described in section (h) below.)

b) Naming of basic mathematical functions

The package itself should be named:

GENERIC_MATH_FUNCTIONS,

where the prefix "GENERIC_" distinguishes it from a possible instantiation, or a non-generic version, with the (same) name MATH_FUNCTIONS. Its components should be named:

```
PI, EXP_1 (the base e of natural logarithms),
SQRT,
LOG (for an arbitrary base),
EXP (for powers of an arbitrary base),
SIN, COS, TAN, COT, (for an arbitrary period),
ARCSIN, ARCCOS, ARCTAN, ARCCOT,
SINH, COSH, TANH, COTH,
ARCSINH, ARCCOSH, ARCTANH, ARCCOTH.
```

Although we agree with other authors, such as Barnes (1982), that identifiers should be meaningful and that abbreviations should not be used where there is any risk of confusion, we think that for the basic mathematical functions the traditional names above are sufficiently familiar. We use the name EXP_1 rather than E, for the base of natural logarithms, on the grounds that there is a significant risk of misuse of E, e.g. when 1.0*E-1 is written instead of 1.0E-1 (assuming a mixed-type subtraction operation to be available) or when E occurs naturally in a sequence of real variables A, B, C, ... Functions with two arguments are explained in detail in section (e) below.

c) Method of use for user-defined types

In accordance with section (f) of Chapter 3, the package structure should be as follows:

```
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is

   function SQRT(X : REAL) return REAL;

   -- LOG, EXP, etc.

end GENERIC_MATH_FUNCTIONS;
```

Then the package may be made available for any user-defined floating-point type, and also for the standard types FLOAT, SHORT_FLOAT and LONG_FLOAT (if present) with implementation-dependent accuracies, by an instantiation of the package for the type concerned; for example:

```
type REAL_6 is digits 6;
package MATH_FUNCTIONS_6 is new GENERIC_MATH_FUNCTIONS(REAL_6);

-- and for the standard type FLOAT:

package STD_MATH_FUNCTIONS is new GENERIC_MATH_FUNCTIONS(FLOAT);
```

(For completeness we remark that the program unit containing such an instantiation must include GENERIC_MATH_FUNCTIONS in its context specification.)

For derived types, the package is automatically available from the parent type. For example, if types REAL and DOUBLE are declared as in section (b) of Chapter 3, and if standard instantiations are available as library units (which makes all subprograms in the instances derivable, LRM 3.4(11)) as suggested in section (d) of Chapter 3, then new instantiations for REAL and DOUBLE are not needed.

No allowance is made here for mixed-type expressions, as when a specification like

```
function SQRT(A : AREA) return LENGTH;
```

is needed. We assume that any such application will be effected by the user by means of type conversions or overloadings. Note, however, that some of our functions serve multiple purposes. For example, our trigonometric functions are so designed that they may be evaluated for angles measured in either radians or degrees (see section (e) below).

Finally we remark that it is perfectly acceptable for every instantiation to deliver the same numbers PI and EXP_1 (since they do not depend upon the generic actual parameter).

d) Efficiency of execution

When writing an Ada source text suitable for calculating values of some basic function for every feasible accuracy, the following problems are faced:

- Whatever the machine arithmetic, the algorithm executed must deliver values as specified with maximal accuracy if the argument is inside its range. In agreement with the recommendations of the Ada-Europe Portability Group (Nissen and Wallis, 1984), algorithms must be given for accuracies ranging from **digits** 5 up to **digits** 10 at least, but in the present context we propose an extension of this requirement up to **digits** 35 and suggest a minimum of 10 (see section (b) of Appendix A).

- An exception SIGNIFICANCE_ERROR might be raised for calls when the argument cannot be used for calculating the value of the basic function with useful accuracy (e.g. for a call of SIN(10.0 ** REAL'DIGITS)). The problem here is that the function body cannot be made aware that the user (the function call) expects a smaller precision than normally, as would be the case if the type provided for the function result had a less stringent accuracy constraint than the type for the parameter. Here all functions have the same floating-point type for parameter(s) and function result. A possible, but somewhat arbitrary, solution is to raise SIGNIFICANCE_ERROR only if more than a specified number of digits will be lost. (This number of digits could be controlled by a second generic parameter of the form

  ```
  SIG : in POSITIVE := 1;
  ```

  with a prescribed default value, in this case unity, but this would only work if the end-user were directly responsible for the instantiation.) Since we have found no satisfactory solution to this problem, we reject the use of SIGNIFICANCE_ERROR here. The alternative, restricting calls of the functions SIN, COS, TAN and COT to arguments in the range [- 2*PI, + 2*PI], is not supported either.

- Algorithms may have many branches conditional upon the accuracy of the type REAL (LRM 3.5.8) (and perhaps also upon the machine mantissa, machine exponent and other machine properties).

- Expressions must be built from the elementary operators only, though some basic functions may call other (more basic) ones from the same package.

- If some branching depends on the value of an argument then it should be distinctly separated from branching which depends on attributes of the generic type. In this way optimising compilers will not be prevented from deleting dead branches.

- The standard type FLOAT cannot be used inside the packages for local declarations and calculations, as this might imply an undesirable loss of accuracy in the final results. Alternatively, it might signify a waste of computer time if FLOAT were much more accurate than necessary. The algorithm might use different approximations for different accuracy constraints. For this reason we advise that branching of algorithms is not done through the MACHINE_MANTISSA attribute but through the DIGITS or the MANTISSA attribute (cf. section (c) of Chapter 3).

- As static expressions in floating-point type definitions cannot depend on attributes of the generic actual parameter (LRM 4.9), it is not possible (see section (d) of Appendix A) to make a local floating-point type definition with a (slightly) larger accuracy, e.g.

  ```
  type LOCAL_REAL is digits REAL'DIGITS + 2;
  ```

  for performing the internal calculations. All algorithms for basic functions must simply deliver the best results possible using the user-supplied floating-point type. If this user-supplied type has unexpected additional constraints, then the exception CONSTRAINT_ERROR will be raised upon violation. This exception can also be raised in the package body (elaborated upon instantiation) if the user-defined type is unfit for any calculation at all.

- In the same way static expressions in fixed-point type definitions cannot depend on attributes of the generic actual parameter. So the idea of Wichmann (1984) of using local fixed-point arithmetic for evaluating polynomials cannot apply here, because the appropriate fixed-point types cannot be defined (unless the types are declared inside the different branches). Besides, it will be uncertain whether a fixed-point type with as large a mantissa as that of the floating-point type is supported.

- No exception occurring in intermediate calculations should be propagated to the user's call (provided that the final result would not be exceptional). Only when the final result is exceptional, due to a bad argument of the function call, should an appropriate exception be raised (see section (f) below).

- Program units using the basic Mathematical Functions package should not each make their own instantiation of GENERIC_MATH_FUNCTIONS, as this might imply that several copies are made. Consider for example:

```
generic
   type REAL is digits <>;
package GENERIC_CHOLESKY is
   type SYMMATRIX is array(INTEGER range <>) of REAL;
   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX);
end GENERIC_CHOLESKY; -- specification

with GENERIC_MATH_FUNCTIONS;
package body GENERIC_CHOLESKY is

   package MATH_FUNCTIONS is
      new GENERIC_MATH_FUNCTIONS(REAL);
   use MATH_FUNCTIONS;

   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX) is

      -- Local declarations

   begin
      DECOMPOSE_MAT;
   end CHOLESKY_DECOMPOSITION;

end GENERIC_CHOLESKY; -- body
```

Such a package, which itself must be instantiated, would require an instantiation of the basic Mathematical Functions package and so would all other similar numeric packages.

A solution might be that a numeric package (in the above and following examples for the Cholesky decomposition of symmetric positive-definite matrices, which needs the SQRT function) is given as a generic package with, as generic parameters (besides the user-supplied floating-point type), those basic mathematical functions which it uses. These generic subprogram parameters must be declared with themselves as defaults, in which case we have

```
generic
   type REAL is digits <>;
   with function SQRT(X : REAL) return REAL is <>;
package GENERIC_CHOLESKY is
   type SYMMATRIX is array(INTEGER range <>) of REAL;
   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX);
end GENERIC_CHOLESKY; -- specification
```

with a body of the form:

```
package body GENERIC_CHOLESKY is

   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX) is

      -- Local declarations

   begin
      DECOMPOSE_MAT;             -- using SQRT
   end CHOLESKY_DECOMPOSITION;

end GENERIC_CHOLESKY; -- body
```

Such a generic package can be used in the following way:

```
with GENERIC_MATH_FUNCTIONS, REAL_TYPES; use REAL_TYPES;
with GENERIC_CHOLESKY; -- and other numeric packages, etc.
procedure MAIN is

   -- Instantiations:

   package MATH_FUNCTIONS is
      new GENERIC_MATH_FUNCTIONS(REAL);
   use MATH_FUNCTIONS;

   package MY_CHOLESKY is new GENERIC_CHOLESKY(REAL);

      -- Note that the name SQRT is visible, through
      -- the use clause, and that SQRT can be used
      -- as the generic actual parameter since it
      -- has the correct subprogram specification.

   -- etc.

begin
   MAIN_PROGRAM_STATEMENTS;
end MAIN;
```

Unfortunately, this solution violates the "black box" principle of library software by making the (possible) use of the function SQRT apparent to the user when there should really be no need for him to know that this function is used. We would prefer the contents of the package body to be completely hidden from the user so that any changes within the body, such as the use of some other function than SQRT, would not affect dependent library units.

e) Calling sequences

Assuming the availability of the instantiation:

```
type REAL_6 is digits 6; -- as an example
package MATH_FUNCTIONS_6 is
   new GENERIC_MATH_FUNCTIONS(REAL_6);
```

and the use clause:

```
use MATH_FUNCTIONS_6;
```

it follows from the full declarations given in section (g), below, that each of the basic mathematical functions can be called, taking SQRT as an example, in each of the following ways:

```
MATH_FUNCTIONS_6.SQRT(REAL_6_EXPRESSION) -- as a primary

SQRT(REAL_6_EXPRESSION) -- when the component SQRT of the
                        -- package is visible

SQRT(X => REAL_6_EXPRESSION) -- using the name of the
                             -- formal parameter.
```

Similar calls apply to those functions with two arguments, the second of which has a prescribed default value.

The declarations of LOG and EXP take the form:

```
function LOG(X : REAL; BASE : REAL := EXP_1) return REAL;
function EXP(X : REAL; BASE : REAL := EXP_1) return REAL;
```

where the second parameter, with the default value EXP_1, gives the base of the logarithm or the power respectively. Thus, for example, a call of LOG(X) gives the value of the natural logarithm ln X, while EXP(X,A) gives the value of A^X. Note that EXP(X,A) is used in preference to A**X (overloading **) to avoid confusion with the predefined operator ** which yields only integer powers (corresponding to repeated multiplication).

The declarations of the trigonometric functions are

```
function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
function TAN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
function COT(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
```

where the second argument CYCLE gives the complete angle at a point in the units of the first argument X, i.e. CYCLE := 2.0*PI (the default value) when X is measured in radians, CYCLE := 360.0 when X is measured in degrees, etc. Note that the second argument represents the period of the functions SIN and COS but is twice the period of the functions TAN and COT.

The declarations of ARCTAN and ARCCOT allow a particular function call for arguments close to infinity. Their declarations read:

```
function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
```

and are such that, for example, a call:

```
ARCTAN(REAL_EXPRESSION)
```

delivers the normal arctangent value in the range [- PI/2, PI/2], whereas:

```
ARCTAN(REAL_EXPR1, REAL_EXPR2)
```

delivers the angle between the X-axis and the radius vector of the Cartesian point (REAL_EXPR2, REAL_EXPR1) (note the different orders of the coordinates and the parameters of ARCTAN) lying in the range (- PI, PI]. This would also be delivered, but possibly less accurately, by

```
ARCTAN(REAL_EXPR1/REAL_EXPR2).
```

The ranges of the arguments of all the functions in the package are specified in Appendix C.

f) Exceptions

Any exceptional situation which arises can lead to the raising of an exception, this raising being done either automatically or by an explicit raise statement. Exceptions which may be raised automatically (or explicitly) are the predefined exceptions (see LRM 11.1 and section (a) of Chapter 7):

NUMERIC_ERROR (for errors in the use of floating-point arithmetic, especially "overflow"),

CONSTRAINT_ERROR (for "out-of-range" values, as might occur with function calls and array indexing),

STORAGE_ERROR (self-explanatory) and

PROGRAM_ERROR (usually for programming errors, such as calling a subprogram before its body has been elaborated, reaching the end of a function call without having executed a return statement, and so on).

Other exceptions, which may be raised only explicitly, must be declared explicitly. We propose that the basic mathematical functions package contains only one such exception:

ARGUMENT_ERROR (for arguments which are outside the domain of the relevant function, e.g. negative arguments for SQRT).

We propose that an exception is raised by a function only if its final result would be exceptional. More specifically, if NUMERIC_ERROR is not raised automatically but special values are returned by the hardware, then the function body should not raise an exception, as it might be the user's wish to continue the calculations with these special values. In most cases an exception that is raised automatically (usually NUMERIC_ERROR or CONSTRAINT_ERROR) can be propagated, but it is permitted for a basic function to handle such an exception or to raise another exception as appropriate. This may be compared with the IEEE recommendations for binary floating-point arithmetic (IEEE, 1981): they advise that exceptions (like invalid operations, division by zero, overflow, underflow) must be detected by the hardware, but that the user should have the means to enable and disable the corresponding traps.

g) Package specification

The complete generic package declaration is as follows:

```
-------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is
   ----------------------------------------------------------------
   -- Declare constants.                                         --
   ----------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ----------------------------------------------------------------
   -- Declare the basic mathematical functions.                  --
   ----------------------------------------------------------------
   function SQRT(X : REAL) return REAL;
   function LOG(X : REAL; BASE : REAL := EXP_1) return REAL;
   function EXP(X : REAL; BASE : REAL := EXP_1) return REAL;
   function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function TAN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COT(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function ARCSIN(X : REAL) return REAL;
```

```
    function ARCCOS(X : REAL) return REAL;
    function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
    function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
    function SINH(X : REAL) return REAL;
    function COSH(X : REAL) return REAL;
    function TANH(X : REAL) return REAL;
    function COTH(X : REAL) return REAL;
    function ARCSINH(X : REAL) return REAL;
    function ARCCOSH(X : REAL) return REAL;
    function ARCTANH(X : REAL) return REAL;
    function ARCCOTH(X : REAL) return REAL;
    ------------------------------------------------------------------
    -- Declare exceptions.                                          --
    ------------------------------------------------------------------
    ARGUMENT_ERROR : exception;
    ------------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
---------------------------------------------------------------------
```

For the package body, guidelines about the delivered accuracy and the raising of exceptions are given in sections (d) and (f) above. No textual error messages should be issued. We advise that all the program components of the package body are given as body stubs with separate subunits, assuming that facilities for partial loading are available (see Appendix A and section (h) of Chapter 8).

h) Practical considerations

As noted in section (f) of Chapter 3, and mentioned in section (a) above, a particular implementation may, for reasons of efficiency (see section (b) of Appendix A), effect an instantiation of a generic package by calling an equivalent non-generic version (which may even be on a special-purpose chip). As far as the user is concerned, the fact that this is not an instantiation in the normal sense will not be evident and will not matter.

In the present case, the non-generic version will have the specification:

```
with REAL_TYPES; use REAL_TYPES;
package MATH_FUNCTIONS is

   -- Declarations as in the generic package above

end MATH_FUNCTIONS; -- specification
```

and the body:

```
package body MATH_FUNCTIONS is

   function SQRT(X : REAL) return REAL is separate;
   function LOG(X : REAL; BASE : REAL := EXP_1) return REAL
      is separate;

   -- etc.

end MATH_FUNCTIONS; -- body
```

Then each function will have a separate body, typically of the form:

```
separate (MATH_FUNCTIONS)
function MATH_FUNCTION(X : REAL) return REAL is

   -- Local declarations

begin

   -- Sequence of statements

end MATH_FUNCTION;
```

This may be preceded by a context clause if necessary.

Example bodies for the functions SQRT, SIN and COS are given in Appendix C.

5. COMPOSITE DATA TYPES

In this chapter we discuss the provision of composite data types such as COMPLEX, VECTOR and MATRIX.

a) Complex operators

Since complex variables are seldom used without complex arithmetic, we propose that the type COMPLEX should be provided, as a record type (cf. Wichmann, 1984), alongside its associated operators in a package of the form:

```
package COMPLEX_OPERATORS is

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

   function "+"(X : COMPLEX) return COMPLEX;
   function "-"(X : COMPLEX) return COMPLEX;
   function "abs"(X : COMPLEX) return REAL;
   function ARG(X : COMPLEX) return REAL;
   function "+"(X,Y : COMPLEX) return COMPLEX;
   function "-"(X,Y : COMPLEX) return COMPLEX;
   function "*"(X,Y : COMPLEX) return COMPLEX;
   function "/"(X,Y : COMPLEX) return COMPLEX;
   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX;

end COMPLEX_OPERATORS; -- specification
```

where it is assumed that a floating-point type REAL is already available, e.g. through a context clause:

```
with REAL_TYPES; use REAL_TYPES;
```

such as was introduced in section (f) of Chapter 3. If it is further assumed that the basic mathematical functions applicable to such REAL variables are available in a package MATH_FUNCTIONS, e.g. through an instantiation of the generic package described in Chapter 4:

```
package MATH_FUNCTIONS is new GENERIC_MATH_FUNCTIONS(REAL);
```

then the package body, corresponding to the above specification, could take the form:

```
with MATH_FUNCTIONS;
package body COMPLEX_OPERATORS is

   use MATH_FUNCTIONS;

   function "+"(X : COMPLEX) return COMPLEX is
   begin
      return X;
   end "+";
```

```
function "-"(X : COMPLEX) return COMPLEX is
begin
   return (- X.RE, - X.IM);
end "-";

function "abs"(X : COMPLEX) return REAL is
   A,B : REAL;
begin
   if abs X.RE > abs X.IM then
      A := abs X.RE;
      B := abs X.IM;
   else
      A := abs X.IM;
      B := abs X.RE;
   end if;
   if A > 0.0 then
      return A*SQRT(1.0 + (B/A)**2);
   else
      return 0.0;
   end if;
end "abs";

function ARG(X : COMPLEX) return REAL is
begin
   return ARCTAN(X.IM, X.RE);
end ARG;

function "+"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE + Y.RE, X.IM + Y.IM);
end "+";

function "-"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE - Y.RE, X.IM - Y.IM);
end "-";

function "*"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE*Y.RE - X.IM*Y.IM, X.IM*Y.RE + X.RE*Y.IM);
end "*";

function "/"(X,Y : COMPLEX) return COMPLEX is
   A,B : REAL;
begin
   if abs Y.RE > abs Y.IM then
      A := Y.IM/Y.RE;
      B := A*Y.IM + Y.RE;
      return ((X.RE + A*X.IM)/B, (X.IM - A*X.RE)/B);
   else
      A := Y.RE/Y.IM;
      B := A*Y.RE + Y.IM;
      return ((A*X.RE + X.IM)/B, (A*X.IM - X.RE)/B);
   end if;
end "/";
```

```
      function "**"(X : COMPLEX; N : INTEGER) return COMPLEX is
         CMOD,CARG,R,THETA : REAL;
      begin
         CMOD := abs X;
         CARG := ARG(X);
         R := CMOD**N;
         THETA := N*CARG;
         return (R*COS(THETA), R*SIN(THETA));
      end "**";

   end COMPLEX_OPERATORS; -- body
```

The complex division function in this package could perhaps raise an explicit exception if the denominator Y should vanish but it would seem better to rely upon the outcome of the real divisions within it (i.e. upon whether or not they raise an exception).

Note that there are no explicit type conversions between types REAL and COMPLEX but that, given

```
   R,I : REAL;
   C   : COMPLEX;
```

we may write

```
   C := (R,I);
```

or, equivalently,

```
   C := COMPLEX'(R,I);
```

to form a complex number from two real numbers, and

```
   R := C.RE;
   I := C.IM;
```

to extract the real and imaginary parts of a complex number.

b) Use of generics for complex operators

Following our proposals in section (f) of Chapter 3, we might consider making a generic form of the above package:

```
   generic
      type REAL is digits <>;
   package GENERIC_COMPLEX_OPERATORS is

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

      -- etc.

   end GENERIC_COMPLEX_OPERATORS; -- specification
```

in which case the corresponding package body would take the form:

```
with GENERIC_MATH_FUNCTIONS;
package body GENERIC_COMPLEX_OPERATORS is

   package MATH_FUNCTIONS is new GENERIC_MATH_FUNCTIONS(REAL);
   use MATH_FUNCTIONS;

   function "+"(X : COMPLEX) return COMPLEX is
   begin
      return X;
   end "+";

   -- etc.

end GENERIC_COMPLEX_OPERATORS; -- body
```

The particular instantiation:

```
package COMPLEX_OPERATORS is new GENERIC_COMPLEX_OPERATORS(REAL);
```

would then serve the same purpose as the non-generic package, in section (a) above, for the same REAL type. The generic form would also satisfy the needs of the sophisticated programmer wishing to use some floating-point type other than type REAL. However, this construction cannot be recommended for general use, since it necessitates an instantiation of the basic mathematical functions package within an instantiation of the complex operators package (cf. section (d) of Chapter 3).

Another construction, which may be preferable, is obtained by following the example in section (d) of Chapter 4 and making the package GENERIC_COMPLEX_OPERATORS generic with respect to each of the mathematical functions which it uses, viz. SQRT, ARCTAN, SIN and COS. In this case we have

```
generic
   type REAL is digits <>;
   with function SQRT(X : REAL) return REAL is <>;
   with function ARCTAN(X,Y : REAL) return REAL is <>;
   with function SIN(X : REAL; CYCLE : REAL := 2.0*PI)
      return REAL is <>;
   with function COS(X : REAL; CYCLE : REAL := 2.0*PI)
      return REAL is <>;
package GENERIC_COMPLEX_OPERATORS is

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

   -- etc.

end GENERIC_COMPLEX_OPERATORS; -- specification
```

with a body of the form:

```
package body GENERIC_COMPLEX_OPERATORS is

   ...
```

```
      function "abs"(X : COMPLEX) return REAL is
         A,B : REAL;
      begin
         ...         -- using SQRT
      end "abs";

      function ARG(X : COMPLEX) return REAL is
      begin
         return ARCTAN(X.IM, X.RE);
      end ARG;

      ...

      function "**"(X : COMPLEX; N : INTEGER) return COMPLEX is
         CMOD,CARG,R,THETA : REAL;
      begin
         ...         -- using SIN and COS (with defaults)
      end "**";

   end GENERIC_COMPLEX_OPERATORS; -- body
```

Provided that the necessary MATH_FUNCTIONS are visible, e.g. through a use clause, this package may be instantiated exactly as above. We observe here, however, that to proceed in this way in general could lead to very long lists of generic function parameters.

c) Complex functions

Corresponding to the basic mathematical functions considered in Chapter 4, we might also have a package of basic complex functions with the specification:

```
   with COMPLEX_OPERATORS; use COMPLEX_OPERATORS;
   package COMPLEX_FUNCTIONS is

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

   end COMPLEX_FUNCTIONS; -- specification
```

and the package body:

```
   with REAL_TYPES, MATH_FUNCTIONS;
   package body COMPLEX_FUNCTIONS is

      use REAL_TYPES, MATH_FUNCTIONS;

      function SQRT(X : COMPLEX) return COMPLEX is
         YR,YI : REAL;
         ABS_X : constant REAL := abs X;
      begin
         if ABS_X = 0.0 then
            return (0.0, 0.0);
         elsif X.RE >= 0.0 then
            YR := SQRT((X.RE + ABS_X)/2.0);
            YI := X.IM/(2.0*YR);
         return (YR, YI);
      else
         declare
            SIGN : REAL;
         begin
            if X.IM >= 0.0 then
               SIGN := 1.0;
            else
               SIGN := - 1.0;
            end if;
            YI := SIGN*SQRT((abs X.RE + ABS_X)/2.0);
            YR := X.IM/(2.0*YI);
            return (YR, YI);
         end;
      end if;
   end SQRT;

   function LOG(X : COMPLEX) return COMPLEX is
   begin
      return (LOG(abs X), ARG(X));
   end LOG;

   function EXP(X : COMPLEX) return COMPLEX is
      EXP_RE : constant REAL := EXP(X.RE);
   begin
      return (EXP_RE*COS(X.IM), EXP_RE*SIN(X.IM));
   end EXP;

   function SIN(X : COMPLEX) return COMPLEX is
   begin
      return (SIN(X.RE)*COSH(X.IM), COS(X.RE)*SINH(X.IM));
   end SIN;

   function COS(X : COMPLEX) return COMPLEX is
   begin
      return (COS(X.RE)*COSH(X.IM), - SIN(X.RE)*SINH(X.IM));
   end COS;

end COMPLEX_FUNCTIONS; -- body
```

d) Use of generics for complex functions

Unfortunately, a record type cannot be passed as a generic parameter (LRM 12.1.2). Therefore, we cannot make the above package of complex functions generic with respect to the type COMPLEX unless we make this type private (see section (d) of Appendix A). Then, of course, this type is no longer necessarily defined by its real and imaginary parts, but may, for example be given, in polar form, by its modulus and argument, thus:

```
type COMPLEX is
   record
      CMOD,CARG : REAL;
   end record;
```

Since the bodies of the functions within the package require the real and imaginary parts and the modulus and argument of the type COMPLEX, it would appear to be necessary to make the package generic also with respect to functions which extract these parts. Similarly, since the bodies require to form a complex number from its real and imaginary parts, or from its modulus and argument, the package must also be generic with respect to functions which do this, e.g.

```
function C_TO_COMP(R : REAL; I : REAL := 0.0)
   return COMPLEX;
function P_TO_COMP(M : REAL; A : REAL := 0.0)
   return COMPLEX;
```

The specification of the generic package might therefore take the form:

```
with REAL_TYPES; use REAL_TYPES;
generic
   type COMPLEX is private;
   with function RE(X : COMPLEX) return REAL is <>;
   with function IM(X : COMPLEX) return REAL is <>;
   with function "abs"(X : COMPLEX) return REAL is <>;
   with function ARG(X : COMPLEX) return REAL is <>;
   with function C_TO_COMP(R : REAL; I : REAL := 0.0)
      return COMPLEX is <>;
   with function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX is <>;
package GENERIC_COMPLEX_FUNCTIONS is

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

end GENERIC_COMPLEX_FUNCTIONS; -- specification
```

The body of this package could then take the form:

```
with MATH_FUNCTIONS;
package body GENERIC_COMPLEX_FUNCTIONS is

   use MATH_FUNCTIONS;

   function SQRT(X : COMPLEX) return COMPLEX is
      ABS_X : constant REAL := abs X;
   begin
      if ABS_X = 0.0 then
         return C_TO_COMP(0.0, 0.0);
      else
         return P_TO_COMP(SQRT(ABS_X), 0.5*ARG(X));
      end if;
   end SQRT;

   function LOG(X : COMPLEX) return COMPLEX is
   begin
      return C_TO_COMP(LOG(abs X), ARG(X));
   end LOG;

   function EXP(X : COMPLEX) return COMPLEX is
   begin
      return P_TO_COMP(EXP(RE(X)), IM(X));
   end EXP;
```

```
      function SIN(X : COMPLEX) return COMPLEX is
      begin
         return C_TO_COMP(SIN(RE(X))*COSH(IM(X)),
            COS(RE(X))*SINH(IM(X)));
      end SIN;

      function COS(X : COMPLEX) return COMPLEX is
      begin
         return C_TO_COMP(COS(RE(X))*COSH(IM(X)),
            - SIN(RE(X))*SINH(IM(X)));
      end COS;

   end GENERIC_COMPLEX_FUNCTIONS; -- body
```

For the type COMPLEX defined in the package COMPLEX_OPERATORS, those functions which are required as generic parameters, but which are not included in the package COMPLEX_OPERATORS, may be included in a package COMPLEX_PARTS, thus:

```
   with REAL_TYPES, COMPLEX_OPERATORS;
   use REAL_TYPES, COMPLEX_OPERATORS;
   package COMPLEX_PARTS is

      function RE(X : COMPLEX) return REAL;
      function IM(X : COMPLEX) return REAL;
      function C_TO_COMP(R : REAL; I : REAL := 0.0)
         return COMPLEX;
      function P_TO_COMP(M : REAL; A : REAL := 0.0)
         return COMPLEX;

   end COMPLEX_PARTS; -- specification
```

with the body:

```
   with MATH_FUNCTIONS; use MATH_FUNCTIONS;
   package body COMPLEX_PARTS is

      function RE(X : COMPLEX) return REAL is
      begin
         return X.RE;
      end RE;

      function IM(X : COMPLEX) return REAL is
      begin
         return X.IM;
      end IM;

      function C_TO_COMP(R : REAL; I : REAL := 0.0)
         return COMPLEX is
      begin
         return (R, I);
      end C_TO_COMP;

      function P_TO_COMP(M : REAL; A : REAL := 0.0)
         return COMPLEX is
      begin
         return (M*COS(A), M*SIN(A));
      end P_TO_COMP;

   end COMPLEX_PARTS; -- body
```

We might then have an instantiation:

```
with COMPLEX_OPERATORS, COMPLEX_PARTS;
use COMPLEX_OPERATORS, COMPLEX_PARTS;
package COMPLEX_FUNCTIONS is
   new GENERIC_COMPLEX_FUNCTIONS(COMPLEX);
```

Clearly, the contents of the package COMPLEX_PARTS may be included in the package COMPLEX_OPERATORS, and we now recommend this; then the above instantiation simplifies to

```
with COMPLEX_OPERATORS; use COMPLEX_OPERATORS;
package COMPLEX_FUNCTIONS is
   new GENERIC_COMPLEX_FUNCTIONS(COMPLEX);
```

A corresponding package of COMPLEX_POLAR_OPERATORS permits the alternative instantiation:

```
with COMPLEX_POLAR_OPERATORS; use COMPLEX_POLAR_OPERATORS;
package COMPLEX_FUNCTIONS is
   new GENERIC_COMPLEX_FUNCTIONS(COMPLEX);
```

for the user who wishes to work in polar coordinates; we recommend this construction, whereby the one generic package provides the required functions for either of the two COMPLEX types. We note in passing that the use of generics in this manner is a subject of considerable importance in the design of scientific libraries and is currently being further investigated by L.M.Delves and C.Pursglove as part of the activities of the Ada-Europe Numerics Working Group.

The packages COMPLEX_OPERATORS and COMPLEX_POLAR_OPERATORS are presented in full in Appendix D, where in each case the efficiency of the package is enhanced by the use of the pragma INLINE (LRM 6.3.2). The packages may be extended to include operations between REAL and COMPLEX arguments.

e) Vectors and matrices

Packages similar to those proposed for complex arithmetic might be provided for vectors and matrices, but we consider that these types, being useful in their own right, are best packaged separately from their associated operators. Thus for a given

```
type REAL is digits D;
```

where D has some appropriate value for scientific computation, we define

```
type VECTOR is array (INTEGER range <>) of REAL;
type MATRIX is array
   (INTEGER range <>, INTEGER range <>) of REAL;
```

and we group these three types together in one package, as suggested in section (f) of Chapter 3:

```
   package REAL_TYPES is
      type REAL is digits D;
      type VECTOR is array (INTEGER range <>) of REAL;
      type MATRIX is array
         (INTEGER range <>, INTEGER range <>) of REAL;
   end REAL_TYPES;
```

In this case, the context clause:

```
   with REAL_TYPES; use REAL_TYPES;
```

attached to a library unit, gives immediate access, within that unit, to all three types, as, for example, in the package LEAST_SQUARES in Appendix E.

Though types VECTOR and MATRIX are defined in terms of type REAL, there is little to be gained by defining them in a generic package thus:

```
   generic
      type REAL is digits <>;
   package GENERIC_REAL_TYPES is
      type VECTOR is array (INTEGER range <>) of REAL;
      type MATRIX is array
         (INTEGER range <>, INTEGER range <>) of REAL;
   end GENERIC_REAL_TYPES;
```

This could be useful for a programmer wishing to manipulate vectors and matrices with a particular precision other than D digits, but it would not be very helpful in the construction of library packages. Suppose, for example, that one were to make a linear algebra package generic with respect to the type REAL:

```
   generic
      type REAL is digits <>;
   package GENERIC_LINEAR_ALGEBRA is
      ...
   end GENERIC_LINEAR_ALGEBRA;
```

Then within this package, which manipulates vectors and matrices, one would require an instantiation:

```
   package REAL_TYPES is new GENERIC_REAL_TYPES(REAL);
   use REAL_TYPES;
```

giving access to the types VECTOR and MATRIX. Unfortunately, these types would not be directly available outside an instantiation:

```
   package LINEAR_ALGEBRA is new GENERIC_LINEAR_ALGEBRA(REAL);
```

and another instantiation of GENERIC_REAL_TYPES in a user's program would yield a different set of REAL_TYPES. The user would therefore not have access to subprograms in the LINEAR_ALGEBRA package with VECTOR or MATRIX parameters, unless for actual parameters of the types

```
   LINEAR_ALGEBRA.REAL_TYPES.VECTOR
```

and

```
   LINEAR_ALGEBRA.REAL_TYPES.MATRIX
```

or by using type conversions for the array parameters. For general purposes, of course, one instantiation of the package GENERIC_REAL_TYPES for the appropriate type REAL:

```
   package REAL_TYPES is new GENERIC_REAL_TYPES(REAL);
```

would provide a package with the properties of the preceding non-generic form.

If the LINEAR_ALGEBRA package above and the user's program are both to have access to the same types VECTOR and MATRIX, the generic package must take the form:

```
   generic
      type REAL is digits <>;
      type VECTOR is array (INTEGER range <>) of REAL;
      type MATRIX is array
         (INTEGER range <>, INTEGER range <>) of REAL;
   package GENERIC_LINEAR_ALGEBRA is
      ...
   end GENERIC_LINEAR_ALGEBRA;
```

in which case the instantiation:

```
   with REAL_TYPES; use REAL_TYPES;
   package LINEAR_ALGEBRA is
      new GENERIC_LINEAR_ALGEBRA(REAL,VECTOR,MATRIX);
```

in the user's program will give him access to the types REAL, VECTOR and MATRIX from the package REAL_TYPES.

Finally, in this chapter, we propose that a complex vector should be represented as a vector of complex components, thus:

```
   type CO_VECTOR is array (INTEGER range <>) of COMPLEX;
```

and not as a pair of vectors:

```
   type CO_VECTOR(SIZE : INTEGER) is
      record
         RES,IMS : VECTOR(1 .. SIZE);
      end record;
```

since a complex vector is more often accessed element by element than by its real and imaginary parts. Similarly, we suggest the declaration:

```
   type CO_MATRIX is array
      (INTEGER range <>, INTEGER range <>) of COMPLEX;
```

for a complex two-dimensional array.

6. INFORMATION PASSING

Software interface problems arise whenever two (or more) items of software are to be used in conjunction with each other. In this chapter we consider such problems in detail, beginning with the particular problems which arise when one item is a library procedure and the other a function (or procedure) to be supplied by the user, in which case the former has to be designed extremely carefully in order to accommodate the latter in a flexible but straightforward manner.

Problems in which the user has to specify a mathematical function to a library procedure in this way occur in many areas of numerical analysis including the solution of differential and integral equations, function approximation, and the location of zeros and extrema of functions.

Consider the following model problem:

> It is required to design a mathematical library procedure to find a zero z of a function f(x), for real x in an interval [a, b], to an absolute accuracy e > 0. The function f and the values of a, b and e are to be specified by the user.

We discuss, in the following sections (a) - (d), the solution of this problem for functions f(x) of varying complexity. In each section, we begin by describing a solution in FORTRAN, which will be familiar to many readers, and then describe the corresponding solution in Ada.

In section (e), we discuss the various possibilities which are available for parameter association together with the use of defaults.

a) Solution of model problem for simple functions

If the function f(x) has a simple explicit expression in terms of x, a FORTRAN library subroutine for the solution of the model problem might take the form:

```
SUBROUTINE ZERO(F, A, B, E, Z)
REAL F, A, B, E, Z
EXTERNAL F
...
code for determining Z from F
...
RETURN
END
```

where F is declared as EXTERNAL in the calling (sub)program. (In practice ZERO would have additional parameters, to indicate cases of failure, etc.) The user would be asked to supply a function subprogram which would return the value of F corresponding to any specified value of X in [A, B]. Subroutine ZERO would operate according to some iterative process, making repeated calls of F for values of X selected by the process until it was deemed that a zero Z had been determined to the prescribed tolerance E. In its simplest form the subprogram would appear as:

```
      REAL FUNCTION F(X)
      REAL X
      ...
      code for determining F from X
      ...
      RETURN
      END
```

For straightforward problems this approach is ideal. For example, to determine the zero z of the function $g(x) = e^x - x - 3$, within the interval [0, 2] to an absolute accuracy of 10^{-6}, the user would simply supply the subprogram:

```
      REAL FUNCTION G(X)
      REAL X
      G = EXP(X) - X - 3.0
      RETURN
      END
```

and make the call:

```
      CALL ZERO(G, 0.0, 2.0, 1.0E-6, Z)
```

In Ada, as already mentioned in section (d) of Chapter 2, functions may not be passed as procedure parameters in the normal way (see section (d) of Appendix A) but may be passed by means of generics (LRM 12). Consequently, for the model problem above, an appropriate Ada procedure might have the generic specification:

```
   generic
      with function F(X : REAL) return REAL;
   procedure GENERIC_ZERO(A,B,E : in REAL; Z : out REAL);
```

where it is assumed, as it will be throughout this Chapter, that type REAL is available, e.g. through the context clause:

```
   with REAL_TYPES; use REAL_TYPES;
```

The body of this procedure must contain the code for determining the zero Z from the function F. Then, in the manner of the example given in section (d) of Chapter 2, the zero of a specific function g(x), with the specification:

```
   function G(X : REAL) return REAL;
```

may be obtained to the required accuracy by instantiating the generic procedure, thus:

```
   procedure ZERO_G is new GENERIC_ZERO(G);
```

and making the call:

```
   ZERO_G(A, B, E, Z);
```

with appropriate values for A, B and E.

For the specific example above, the body of the function G will have the form:

```
function G(X : REAL) return REAL is
begin
   return EXP(X) - X - 3.0;
end G;
```

and the procedure call will be simply:

```
ZERO_G(0.0, 2.0, 1.0E-6, Z);
```

b) Solution of model problem using global variables

For many applications the simple approach used above is impracticable since the user-specified function depends upon additional information, as for example with the function:

$$h(x) = \sum_{j=1}^{n} c_j \exp(d_j x)$$

for specified values of n and the coefficients c_j, d_j, j = 1,...,n.

The only way to supply such information to the function subprogram written in the above form is to declare variables that are global to it. In FORTRAN, because of its lack of block structure, the global variables have to be simulated through the use of COMMON statements. For example, for the function h(x) above, the user could supply the subprogram:

```
   REAL FUNCTION H(X)
   REAL X, S
   INTEGER J
   COMMON / CONSTS / N, C(10), D(10)
   S = 0.0
   DO 10 J = 1, N
      S = S + C(J)*EXP(D(J)*X)
10 CONTINUE
   H = S
   RETURN
   END
```

The user's main program must then contain an identical COMMON statement and assign appropriate values to the constants N and C(J), D(J), J = 1,...,N.

Note the severe restriction that arrays in COMMON storage must be specified of fixed length. If, in the example, a value of n larger than 10 were required, the main program, the function subprogram and any other affected program units would have to be modified accordingly and recompiled.

In Ada, this solution may be simulated by using a data package (LRM 7.2):

```
package CONSTS is
   N   : INTEGER := 10;
   C,D : array (1 .. N) of REAL;
end CONSTS;
```

in which case the body of the function representing h(x) might have the form:

```
   with CONSTS; use CONSTS;
   function H(X : REAL) return REAL is
      SUM : REAL := 0.0;
   begin
      for J in 1 .. N loop
         SUM := SUM + C(J)*EXP(D(J)*X);
      end loop;
      return SUM;
   end H;
```

Here the user must assign a value to N (unless n = 10) and the values of the coefficients to the arrays C and D in the package CONSTS, whereafter he may instantiate the generic package, thus:

```
   procedure ZERO_H is new GENERIC_ZERO(H);
```

and call ZERO_H as required. Not surprisingly, this Ada solution is no better than the FORTRAN solution.

A slight improvement over this crude simulation of FORTRAN practice may be obtained by packaging the function, thus:

```
   package FUN is
      N : constant INTEGER := 10;
      procedure INITIALISE(X,Y : in VECTOR);
      function H(X : REAL) return REAL;
   end FUN; -- specification
```

where the vectors X and Y, of the type VECTOR introduced in Chapter 5, are to contain the prescribed coefficients of the series for h(x). The body of this package may have the form:

```
   package body FUN is
      C,D : VECTOR(1 .. N);
      ACTUAL_N : POSITIVE;

      procedure INITIALISE(X,Y : in VECTOR) is
      begin
         ACTUAL_N := X'LAST;
         if ACTUAL_N > N or X'LAST /= Y'LAST then

            -- raise an appropriate exception

         end if;
         C(1 .. ACTUAL_N) := X;
         D(1 .. ACTUAL_N) := Y;
      end INITIALISE;

      function H(X : REAL) return REAL is
         SUM : REAL := 0.0;
      begin
         for J in 1 .. ACTUAL_N loop
            SUM := SUM + C(J)*EXP(D(J)*X);
         end loop;
         return SUM;
      end H;

   end FUN; -- body
```

In this case, the number n of terms in the series for h(x) is implicit in the lengths of the vectors of coefficients. Naming these vectors XN and YN, the user may initialise the arrays C and D, which are now private to the package body, by the procedure call:

```
   FUN.INITIALISE(XN,YN);
```

and instantiate the generic package, thus:

```
   procedure ZERO_H is new GENERIC_ZERO(FUN.H);
```

whereafter the call:

```
   ZERO_H(A, B, E, Z);
```

yields the required zero Z. However, if the number of terms n is larger than the constant value 10, it is necessary to recompile the package FUN.

To overcome this difficulty, we remove N from the specification of the package FUN, thus:

```
   package FUN is
      procedure INITIALISE(X,Y : in VECTOR);
      function H(X : REAL) return REAL;
   end FUN; -- specification
```

and modify the body of the package to:

```
   package body FUN is
      type VECPTR is access VECTOR;
      C,D : VECPTR;

      procedure INITIALISE(X,Y : in VECTOR) is
      begin
         C := new VECTOR'(X);
         D := new VECTOR'(Y);
      end INITIALISE;

      function H(X : REAL) return REAL is
         SUM : REAL := 0.0;
      begin
         for J in 1 .. C'LAST loop
            SUM := SUM + C(J)*EXP(D(J)*X);
         end loop;
         return SUM;
      end H;

   end FUN; -- body
```

In this case, changes may be made in the vectors XN and YN, including changes in length, without any recompilation of the package FUN being required.

In the preceding arguments, we have assumed that the function H(X) has to be compiled outside the main program. However, this will often not be necessary, in which case a much simpler construction, using the block structure of the language, may be adopted as follows:

```
declare

   -- N is imported to this block

   C,D : array (1 .. N) of REAL;

   function H(X : REAL) return REAL is
      SUM : REAL := 0.0;
   begin
      for J in 1 .. N loop
         SUM := SUM + C(J)*EXP(D(J)*X);
      end loop;
      return SUM;
   end H;

   procedure ZERO_H is new GENERIC_ZERO(H);

begin
   ...
   -- Initialise C and D
   ...
   ZERO_H(A, B, E, Z);
   ...
end;
```

This is the construction which we recommend.

c) Parametric solution

The following alternative approach avoids the use of COMMON storage in FORTRAN, but requires a different structure for the function subprogram. Suppose the function subprogram were to take the form:

```
      REAL FUNCTION F(X, WRK, LWRK, IWRK, LIWRK)
      REAL X, WRK(LWRK)
      INTEGER LWRK, LIWRK, IWRK(LIWRK)
      ...
      RETURN
      END
```

where the real and integer working-space arrays WRK and IWRK are at the disposal of the user. Within these arrays he can store any information relating to the definition of his mathematical function. (We could also add a LOGICAL working-space array if we so wished.) For the example function h(x) above, IWRK(1) could contain n and elements 1 to 2n of WRK could contain the values of the coefficients. These values would have to be initialised before the call to the subroutine ZERO. The dimensions LWRK and LIWRK, of WRK and IWRK respectively, would need to be set appropriately.

The disadvantages of this alternative approach are that

(i) the user is required to pack information (n and the 2n coefficients in the above example), which to him is in meaningful terms, into the anonymity of working-space arrays, and

(ii) he has to code the function subprogram in terms of elements of the working-space arrays, thus losing all clarity in the process.

The first disadvantage is certainly tiresome for the user, but the second may necessitate a major reprogramming effort. For example, in practice, it is not uncommon for each function value to involve extensive computations such as matrix manipulations or the solution of systems of differential equations.

This alternative solution to the model problem, for non-trivial functions f(x), may be implemented in Ada with a function specification:

```
function F(X : REAL; WRK : VECTOR; IWRK : INTEGER_VECTOR)
   return REAL;
```

assuming the availability of appropriate types VECTOR and INTEGER_VECTOR. In this case, the vector lengths LWRK and LIWRK can be extracted from the vectors themselves, by calling upon the appropriate attributes, e.g.

```
LWRK := WRK'LENGTH;
```

within the function body. Otherwise, this solution suffers from the same disadvantages as the FORTRAN version, so it is not recommended. We note also that, in other contexts, the passing of working-space parameters can have undesirable effects (see section (b) of Chapter 8).

d) Reverse communication solution

The rigid specification of the structure of each of the function subprograms described above implies that the user has to program his mathematical function within a set of rules that are outside his control. Ideally, however, a library routine of the type under discussion should not constrain the user at all but should permit him to construct his code in any way he chooses and, perhaps, even more importantly, to use existing code that he may already have available. This may be achieved by means of reverse communication, whereby, in the present context, the control by the library routine over the form of the user's mathematical function may be replaced by full control by the user.

Because of the nature of serial computers, a FORTRAN subroutine such as ZERO would necessarily make successive calls to the user-specified function F. Thus a likely internal structure for ZERO would be:

```
      ...
      DO 20 IT = 1, ITMAX
         (i)   tests to determine whether the process has converged
         (ii)  code to produce a new value of X
         (iii) call to user function to provide the value FX
               of the function corresponding to X
   20 CONTINUE
      ...
```

Now suppose that steps (i) and (ii) above are replaced by a call to a subroutine with declaration part:

```
    SUBROUTINE ZERO2(..., FX, X, INFORM, ...)
    REAL ..., FX, X, ...
    INTEGER ..., INFORM, ...
```

where X is the new estimate of the zero, FX is the value of the function corresponding to the previous value of X, and INFORM indicates the status of the process, e.g. whether a failure of some kind has occurred or whether the process has converged. The unidentified arguments include A, B, etc. and some working-space parameters used to preserve information between calls of ZERO2. (In FORTRAN 77 the SAVE facility could be used to avoid these working-space parameters.)

The situation as seen by the user is so far unchanged. However, now suppose that the declarative part of ZERO is completely removed, the user being requested instead to write <u>in-line code</u> of the form:

```
    ...
    DO 20 IT = 1, ITMAX
       CALL ZERO2( ..., FX, X, INFORM, ... )
       ...
       code to examine INFORM and evaluate FX from X
       ...
 20 CONTINUE
    ...
```

This reverse communication approach has the following advantages:

a) The fact that he is supplying in-line code implies that the user's mathematical function can depend on any or all of the information available in his program.

b) The form of the mathematical function specification is arbitrary: subroutine, function subprogram, in-line code, etc.

c) The user can easily incorporate his own termination requirements: iteration count, absolute or relative error tolerance, etc.

Its disadvantages are:

d) The user has to supply a few lines of in-line code, surrounding the relevant procedure call (to ZERO2 in this case).

e) The zero-finding algorithm is broken up, making its components visible unnecessarily (and inhibiting parallel computation).

f) Working-space is needed to preserve information.

g) Integrity checks are difficult to implement.

Implementation of reverse communication in Ada may proceed along similar lines by introducing a procedure with the specification:

```
procedure ZERO2(...; FX : in REAL;
   X : out REAL; INFORM : out INTEGER; ...);
```

and asking the user to write in-line code of the form:

```
...
for IT in 1 .. ITMAX loop
   ZERO2(..., FX, X, INFORM, ...);
   ...
   ... -- code to examine INFORM and evaluate FX from X
   ...
end loop;
...
```

This implementation has all the advantages of the FORTRAN solution and avoids the passing of unnecessary array parameters which is involved in the parametric solution and which can be costly in Ada if passing is done by copying. However, it also has the disadvantages listed above and we feel that reverse communication is not required in Ada, where the use of generics and the nested block structure of the language provide all that is needed (see, for example, the code at the end of section (b) above).

We note that reverse communication may be required for mixed language programming but this latter topic is not considered in the present report (see section (a) of Appendix G). The problems of interfacing FORTRAN subroutines into Ada programs are currently being studied by C.G. van der Laan in association with the Ada-Europe Numerics Working Group.

e) Parameter association

In subprogram calls, for each parameter an actual parameter is associated with a corresponding formal parameter (LRM 6.4(3)). This association is said to be "named" if the formal parameter is named explicitly, e.g.

```
HEADER => TITLE,
```

otherwise it is said to be "positional". For positional association, each actual parameter corresponds to the formal parameter with the same position in the formal part, whereas named associations may be given in any order (though, once a named association has been given, all following associations must also be named). If a default is given for an **in** parameter (in the formal part), then an association for that parameter can be omitted, in which case the default is used.

No rules are given for the order of evaluation of parameter associations and, even if the parameter-passing mechanism is call-by-copying, the copying-in may be performed in a different order from the copying-out. One might expect that the order of evaluation would be changed if the order of named associations were changed, but this is not necessarily the case. Therefore no subprogram call should depend upon a specific order of evaluation of its parameter associations.

If a formal parameter has a default and its association is omitted from the subprogram call, then for all following parameters the named association must be used. Consequently, it is convenient if all parameters with defaults are given at the end of the formal part. This is contrary to the common practice of specifying **in** parameters at the beginning of the calling sequence.

Finally, we note that formal parameters cannot be used in default expressions in the same formal part as their own specifications (LRM 6.1(5)) and that a type conversion is allowed as an actual parameter (not only for mode **in** but also for modes **out** and **in out**) if the conversion exists for the two types (see also section (i) of Chapter 8).

7. ERROR HANDLING

The Ada exception mechanism provides an elegant and disciplined way of handling error situations. The mechanism has three components: detection of the error, location of the appropriate software to handle the error, and the error handling software itself. However, like all language features, the exception mechanism can be misused. This chapter therefore illustrates the recommended use of exceptions in the design of mathematical libraries. Some pitfalls are noted as appropriate.

a) The predefined exceptions

The misuse of a language construct in Ada, such that no semantics for an operation can be defined, results in the raising of a predefined exception. We consider here the three such exceptions which are most likely to arise in the present context. We discuss TASKING_ERROR later, in section (d) of Chapter 9, and we refer the reader to the LRM 11.1 for details of PROGRAM_ERROR.

- CONSTRAINT_ERROR

A typical example of an undefined operation occurs when an array subscript value lies outside the bounds of the array, in which case the exception CONSTRAINT_ERROR is raised. This situation is clearly caused by a programming bug, which should, ideally, never arise in high-quality software. In other contexts, however, the CONSTRAINT_ERROR exception can arise in software which does not contain such obvious programming bugs.

Consider, for example, the mathematical function SQRT whose specification in the proposed library is:

```
function SQRT(X : REAL) return REAL;
```

If the argument is negative, then the (semantic) specification states that ARGUMENT_ERROR is raised. This can be accomplished by including an initial test in the body of SQRT:

```
if X < 0.0 then
   raise ARGUMENT_ERROR;
end if;
```

However, a reasonable alternative strategy is to use a subtype constraint on the formal parameter:

```
subtype POS is REAL range 0.0 .. REAL'LAST;
function SQRT(X : POS) return REAL;
```

In this case, the constraint is checked before the function is called and the exception CONSTRAINT_ERROR is raised. The subtype POS is used to check a pre-condition on the parameter - such checks being essential for robust real-time software.

The main difference between this last situation and an array bound violation is that here interface checking is necessary in large systems and an occasional violation is to be expected. For instance, a variable which logically must be positive may computationally have a negative value due to rounding errors. Hence CONSTRAINT_ERROR can be raised in "working" software.

Care must be exercised with range constraints used for real types, since range constraints are defined in terms of relational operators which give only approximate results for such types depending upon their accuracy. Consider, for instance:

```
subtype RATIO is REAL range 0.0 .. SQRT(2.0);
```

The mathematical value of the square root of two is certainly not a model number of type REAL. In consequence, values near to the upper bound will give indeterminate results (LRM 4.5.7(10)). In contrast, there should be no problems with the lower bound since 0.0 is a model number (regardless of the accuracy of type REAL).

There is another reason for being cautious about the use of real range constraints, namely the cost in both space and time that the checking of such constraints implies. By contrast with the situation with constraints on integer values, there is little chance here that an optimising compiler will remove "unnecessary" constraint checking.

- NUMERIC_ERROR

The predefined exception NUMERIC_ERROR is very important for mathematical software. Although it is theoretically possible to write software that never overflows, it is substantially simpler not to make the checks that this implies. Because FORTRAN provides no mechanism for controlling overflow, the majority of high-quality packages in that language avoid overflow by careful coding. This approach is satisfactory in many cases but it is virtually impossible to prove that overflow can never arise. Hence in sensitive real-time contexts (e.g. controlling a chemical plant) one must allow for overflow.

The Ada definition does not require the NUMERIC_ERROR exception to be raised on overflow - it merely advises that this is highly desirable. However, we do not believe that it is sensible nowadays to consider a high-quality scientific library on machines which cannot recognise overflow in floating-point arithmetic.

The package COMPLEX_OPERATORS, in section (a) of Chapter 5, has a function to calculate the modulus of a complex value. This function could be written as:

```
function "abs"(C : COMPLEX) return REAL is
begin
   return SQRT(C.RE**2 + C.IM**2);
end "abs";
```

Unfortunately, this simple algorithm has a defect. The expression SQRT(C.RE**2 + C.IM**2) can overflow even if the result is in range (for instance, when the magnitude of C.RE exceeds the square root of the largest number, REAL'LAST, and C.IM is zero). This difficulty can be avoided by careful (but awkward) programming, but can more easily be overcome by handling the exception NUMERIC_ERROR, thus:

```
function "abs"(C : COMPLEX) return REAL is
begin
   return SQRT(C.RE**2 + C.IM**2);
exception
   when NUMERIC_ERROR =>
      declare
         X : REAL := abs C.RE;
         Y : REAL := abs C.IM;
      begin
         if X > Y then
            return X * SQRT(1.0 + (Y/X)**2);
         else
            return Y * SQRT(1.0 + (X/Y)**2);
         end if;
      end;
end "abs";
```

The handler itself uses the cautious approach, so that a value is returned by the function even in cases where NUMERIC_ERROR is raised by the simple algorithm (i.e. cases which might imply restarting the whole computation). Of course, the cautious coding of the handler could be used in the main body, but the method given above is much more efficient if NUMERIC_ERROR is not raised. Also, the main body above is much easier to understand and can act as a logical description of the objective in all cases.

It must be admitted that this example is not entirely satisfactory because the algorithm above has another defect which is not caused by overflow. This concerns underflow. If the real and imaginary parts have very small (but non-zero) values, then the square can underflow to give zero. In these circumstances, the value **abs** Z could be computed as zero even though Z is non-zero. The cautious coding given in Chapter 5 avoids this pitfall.

We advocate that high-quality numerical software should require that NUMERIC_ERROR be raised in overflow situations, although the Ada language does not require this, and several machines cannot efficiently implement our requirement. The reason for our view is a desire to ensure high reliability in all software and to be able to prove small algorithms formally correct. If NUMERIC_ERROR is not raised, then most algorithms will malfunction in extreme cases in such a way that no remedial action is possible. Formal correctness can only be achieved if all values (arguments and results) are in the range of safe numbers. However, almost no computation can be shown to keep to this range; hence the need to raise NUMERIC_ERROR to show the presence of overflow. Even if an algorithm, e.g. a sine routine, keeps its result within the range of safe numbers, its argument value could be outside the range.

One might assume that Ada arithmetic will be adequately behaved if the attribute MACHINE_OVERFLOWS is true. Unfortunately, this is not the case (see section (d) of Appendix A). The current wording (LRM 13.7.3) implies that if MACHINE_OVERFLOWS is true then every real operation gives a result in the model interval defined in LRM 4.5.7, or if this interval is not defined, NUMERIC_ERROR is raised. (This is the highly desirable situation in LRM 4.5.7(7)).

However, it is very unlikely that MACHINE_OVERFLOWS will ever be true according to this definition. To see why this is so, consider the example of double length (say FLOAT) on the IBM series. The machine has 14 hexadecimal places. With the Ada floating-point model, this gives at most 53 binary places (= 4 * 14 - 3). Using the formula in LRM 3.5.7, this implies FLOAT'DIGITS = 15 and FLOAT'MANTISSA = 51 (=B in LRM 3.5.7). FLOAT'SAFE_LARGE will therefore have 51 leading binary 1's in its representation. Meanwhile, the largest machine number clearly has 56 non-zero bits in the mantissa. The difference is caused by two factors (a) use of hexadecimal (3 bits lost), (b) specification of FLOAT in decimal digits rather than binary places (2 bits lost). (A further loss could arise if the machine exponent range were unsymmetric, with more positive than negative values). As a result, there are 31 machine numbers greater than FLOAT'SAFE_LARGE. Moreover, since the IBM arithmetic is in some loose sense "well-behaved", these 31 numbers can result from a real operation <u>and</u> be the "correct" result.

The conclusion to be drawn from this is that the MACHINE_OVERFLOWS attribute should take into account that the underlying hardware may give more precision than required (in the same way that the concept of safe numbers extends the exponent range).

In fact, it does appear possible to define the attribute in an abstract manner, as in the case of the model numbers and safe numbers, which gives the desired properties. Define ideal numbers to be those with unbounded exponents but with the same mantissa length as the model numbers. This is an infinite set, of course, with

{model numbers} $\subseteq$ {safe numbers} $\subset$ {ideal numbers}

Define an ideal interval analogously to the model (safe) interval of LRM 4.5.7. Then if MACHINE_OVERFLOWS is true, every operation either gives a result within the ideal interval or NUMERIC_ERROR is raised. The 31 numbers noted above do not now cause a problem because the result is bounded by the next ideal number (which is <u>not</u> a machine number). Further issues concerning numerics are considered in Appendix F.

- STORAGE_ERROR

The storage required for an Ada program consists of two quite separate parts: storage for the program instructions (and literals) and storage for the data objects. The storage for program instructions and literals is outside the user's control. Consequently, if the program is to run at all, the machine's memory must be sufficient for these. The storage required for data objects is quite different. In general, it is not possible to determine the total storage needed before the program is executed. For example, the size of an array could depend upon values read in by the program. An Ada system could well allocate a fixed amount of storage for data, so that the storage could become exhausted. This would raise the exception STORAGE_ERROR. Entering a subprogram, elaborating declarations and allocating space for objects of an access type are the main actions likely to raise the STORAGE_ERROR exception. The pattern of subprogram calls will determine the main characteristics of the storage needed (and all subprograms should be well documented in this respect), but information on this may be lacking, perhaps because it depends upon the data. In practice, it may be best to run a program with a diagnostic tool to determine its storage characteristics.

It might seem impossible to handle this particular exception because the handler itself would require storage. Fortunately, however, the raising of an exception in itself never requires extra storage.

The possibility might be considered of providing two variants of an algorithm - a fast version using substantial amounts of storage, and a slow version using less storage. The fast version could be attempted and then, if STORAGE_ERROR were raised, the handler could use the slow version. There are only a few circumstances where such a method is likely to be effective since the program would need to contain the instructions for both variants. A more practical method would be to have different bodies for the same package specification, the selection being made by the library builder (for the specific machine or library).

- Suppressing exceptions

Consider a function for matrix multiplication:

```
function MATRIX_PRODUCT(M1,M2 : MATRIX) return MATRIX;
```

For the most obvious implementation, a compiler is likely to generate a time consuming check on the validity of the use of every array reference, whereas a single test that the rows of M1 match the columns of M2 would suffice. By placing this test outside the main loop, the check that the compiler would otherwise perform can be safely suppressed:

```
function MATRIX_PRODUCT(M1,M2 : MATRIX) return MATRIX is
   P : MATRIX(M1'RANGE(1), M2'RANGE(2));
   S : REAL;
   pragma SUPPRESS(INDEX_CHECK);
begin
   if M1'FIRST(2) /= M2'FIRST(1) or
      M1'LAST(2) /= M2'LAST(1) then
      raise CONSTRAINT_ERROR;
   end if;
   for I in M1'RANGE(1) loop
      for J in M2'RANGE(2) loop
         S := 0.0;
         for K in M1'RANGE(2) loop
            S := S + M1(I,K) * M2(K,J);
         end loop;
         P(I,J) := S;
      end loop;
   end loop;
   return P;
end MATRIX_PRODUCT;
```

To perform this form of hand optimisation requires substantial care. Each operation which could require a check must be analysed to ensure that the check is unnecessary.

There does not seem to be any case for the general suppression of the NUMERIC_ERROR exception. Random number generators occasionally use integer multiplication and division ignoring overflow. However, an efficient algorithm which avoids the possibility of overflow is available (Wichmann and Hill, 1982).

b) Existing practices

The machinery for handling error situations in most current languages is cumbersome. Consider for example the case of Pascal, where the method commonly adopted is to perform a non-local GOTO on detecting an error, so that the current algorithm is abandoned. Remedial action can be taken before or after the execution of the GOTO. The method is clearly inflexible, especially in view of the lack of separate compilation in Pascal. A Pascal program using this method would need to be restructured for Ada. Merely replacing the GOTO by the raising of an exception is unlikely to give the best Ada solution.

Large scientific libraries must be able to handle error conditions, and a high-quality library must adopt a consistent method which is both flexible and easy to use. The need for such consistency is evident with respect to the user interface to the library and in the necessity for some library routines to call other routines.

The Numerical Algorithms Group FORTRAN Library (Ford et al., 1979) is an example of a high-quality product which has adopted a consistent technique. This method involves an additional parameter IFAIL which controls both the remedial action and the reporting of potential failures. In Ada terms, IFAIL is an **in out** parameter. The input value determines whether a failure should terminate the program (a hard failure) or whether the program should continue (a soft failure). A recent addition also permits control of the reporting of the failure. The output value indicates the nature of the error in the case of a soft failure.

Since almost anything that can be done in FORTRAN can also be done in Ada, the NAG method of handling failures could be used in an Ada library. However, this would be inappropriate for the following reasons:

a) The existence of the exception mechanism renders the additional parameter unnecessary and leads to a simplified user interface.

b) In a real-time context, the printing out of error or warning messages is inappropriate (there may be no printing device).

c) The soft failure condition is dangerous since the user can easily forget to inspect IFAIL to see if a failure has arisen. (The NAG documentation is careful to draw attention to this danger.)

d) The dual input/output function of the IFAIL parameter can be a source of confusion. The input function can be handled elegantly in Ada by means of an **in** parameter with a default value.

It should also be noted that NAG uses the IFAIL logic to handle errors in input values (such as range constraint violations) where in Ada we might raise CONSTRAINT_ERROR. Similarly, where we might raise NUMERIC_ERROR in Ada, for example in the function EXP for large arguments, the corresponding NAG FORTRAN library code would detect the condition and use IFAIL to handle the situation.

The error-handling mechanism adopted in the Numerical Algorithms

Group Algol 68 Library (NAG, 1982) also involves an additional parameter, NAGFAIL, which enables the user to influence the action to be taken in the event of failure. This mechanism, which has evolved over several years, is currently being considered, in the context of scientific libraries in Ada, by L.M.Delves and the Ada-Europe Numerics Working Group.

c) Recommended Ada practice

Following the above remarks, we adopt here a simple philosophy for the use of exceptions in Ada. The general pattern advocated is that required by defensive programming of adding the test:

```
if pre-conditions not satisfied then
   raise condition violated;
end if;
```

This protects a package/subprogram against misuse which might otherwise inhibit its continued correct operation. It should, however, be noted that it is not necessarily possible to place such a check at the start of a subprogram.

The conclusion here is that each package should declare exceptions corresponding to each class of misuse. A package A may call subprograms in package B. Therefore the question arises as to whether the exceptions that B can raise should be handled by A. This is only necessary if such exceptions would be meaningless to a user of A. For instance, the exception NUMERIC_ERROR does not need to be handled if this is a reasonable response for a user of A (and is in the semantic specification of A). On the other hand, if A is a curve-fitting package and B a matrix package which can raise the exception SINGULAR, then the latter needs to be hidden from the user of A. Hence, in this case, A can handle the exception either to use a different approach or to raise another more appropriate exception.

A further problem arises when an exception may be raised during the evaluation of an expression, where a user might wish to handle the exception in order to make some amendments and to return into the expression to continue its evaluation.

Here, a possibility in some languages is for the subprogram to be "told" beforehand what its reaction should be in the event of an error, in which case "raise an exception" might be replaced by "issue a message and continue with an acceptable value". In Ada, this approach can be adopted by providing an error-mending subprogram as a generic parameter (with the raising of an exception as the default action) to a generic subprogram or even to a complete generic package of subprograms. The disadvantage of this method is that it does not discriminate between the different places where various exceptional events may occur, unless perhaps a long list of generic parameters is provided.

A more satisfactory solution in Ada is for a subprogram which might possibly raise an exception, e.g. SQRT in the following assignment statement:

```
RESULT := A * B + SQRT(C) - D;
```

to be replaced by a local subprogram, e.g. LOCAL_SQRT with the following body:

```
function LOCAL_SQRT(X : REAL) return REAL is
begin
   return SQRT(X);
exception
   when ARGUMENT_ERROR =>
      PUT(MESSAGE); -- using TEXT_IO
      return 0.0;
end LOCAL_SQRT;
```

In this case, the user can replace each SQRT call by a call of LOCAL_SQRT or some other re-definition of SQRT. Note that this example (deliberately) does not handle NUMERIC_ERROR, to show that this exception is not expected and should not be handled inside the expression evaluation.

Finally, we indicate the important difference between exceptions raised in the sequence of statements of a body (or a block statement) and exceptions raised in a declarative part, e.g. in an initialisation such as

```
SQRT_X : constant REAL := SQRT(X);
```

In the former case, the raised exception can be handled in the same body (or block statement). However, in the latter case the exception immediately propagates to the place where the subprogram was called, if it is a subprogram body, or to the surrounding declarative part if it is a package body or a task body (LRM 11.4.2). This suggests that it is advisable to avoid initialisations that are exception prone. On the other hand, examples have been given (see LRM 11.6(10,11)) where the canonical order of certain actions can be changed by an implementation for the sake of optimisation, and this may lead to unexpected values for objects used in an exception handler. The LRM advises one to initialise (by declaration) objects that might otherwise be uninitialised in an exception handler as a result of such an optimisation. Whilst we agree with this advice, we recommend that the expressions involved should not be complicated.

8. WORKING-SPACE ORGANISATION

A comprehensive treatment of the efficient use of working-space would require detailed knowledge of particular compilers and (target) machines. Since hardware capabilities are currently increasing rapidly and Ada compilers are still under development, such a treatment of this subject is not feasible at present. Here, therefore, only general aspects of working-space organisation are considered, these aspects being classified as follows:

- Explicitly allocated storage, associated with type and object declarations in Ada programs. The user can claim storage for data in several ways, some of which would be preferred with respect to Ada style (Nissen and Wallis, 1984), some (not necessarily the same) with respect to efficiency. In the following sections we discuss:

  - choice of data types (transparent or private),
  - use of parameters and generic parameters,
  - representation clauses,
  - use of relevant attributes and pragmas.

- Implicitly used storage, depending on:

  - running system (storage overheads for Ada style declarations),
  - use of the heap,
  - machine architecture,
  - use of generics and subunits,
  - implicit copying for parameter passing, assignment statements with array-type objects and values, and results of function calls.

The subject of length of code of compiled units is not directly addressed here, though section (h) contains some related discussion. For problems that are particularly connected with the use of tasks, see Chapter 9.

In the sequel the term "storage unit" is used, as in the LRM 13, to denote (mostly addressable) storage places in the target machine. No assumptions are made about the number of storage units needed for standard type or user-defined scalar, real and composite type objects, not even if this amount can in some way be controlled by using the pragma STORAGE_UNIT (see section (d) below).

Also the term "heap" is used to denote that part of the working-space which is reserved for dynamic storage allocation (see section (f) below). With reference to the automatic raising of the exception STORAGE_ERROR, see section (a) of Chapter 7.

a) Choice of data types (transparent or private)

Regarding integer and real type objects, it is to be expected that different type definitions (differing in range constraint for integer types, or differing in floating-point accuracy definition for real types) will require different numbers of storage units. However, it should not be assumed that subtype objects will require fewer storage units than objects of their host type. On the contrary, additional range constraints may require more working-space, e.g. for:

```
   A,B : INTEGER range F .. G;
```

which is equivalent to

```
   A : INTEGER range F .. G;
   B : INTEGER range F .. G;
```

A and B belong to different subtypes, even if F and G do not have side-effects, and each object has its own range constraint. (If A and B were of the same subtype, the working-space for storing the range constraint might be associated with the subtype.)

For composite types, the number of storage units needed will usually be the sum of those needed for all components, with additional space for dope vectors (with array types) and discriminant values (with record types). However, a particular implementation may allow for space optimisation by packing more composite type object components in one storage unit, and it can do so either automatically or when instructed v an application of the pragma PACK (see section (d) below). (Note that this pragma cannot be given for objects of anonymous array type, as it can only be applied for named types.)

Use of access type objects will also require some extra storage units (and there arises here the problem of efficient use of the heap, which is discussed in section (f) below).

A minor topic is the claiming of storage within a package body, either by the declaration of composite-type objects in the declarative part or by allocators in the sequence of statements of the package body. While users should be warned if a package body will claim a large amount of storage, we recommend that users should not have access to this storage for updating (although subprograms of the package body can be allowed to update it). Therefore, the object declarations should be placed in the package body, so that the objects are not visible to the user. Programmers should be aware of the simultaneous use of such storage by tasks (see section (c) of Chapter 9 regarding shared-variable updates).

In general it is clear, from the application, what kinds of type definitions are needed for particular purposes. However, in the construction of libraries it would be convenient (to say the least) if all useful algorithms could be made available for as many applications as possible without much extra labour. Copying a matrix from one composite type object to another, in order to be able to call some library subprogram, would generally be unacceptable. Possible alternatives here are:

i. Connect each kind of matrix that requires a different storage method with a different data type; then all subprograms needed will be copied for each data type.

ii. Choose a common data type for all imaginable matrix structures, in which case the matrix-handling subprograms will use a local package of subprograms for the storage method. (This common data type could be a private type declared in the library package, though it might be inefficient to update or read such objects.)

iii. Give one subprogram (for each problem) with generic parameters for the data type and the storage method, leaving it to the user to provide the actual parameters.

For each possibility, the working-space to be claimed for storing the (relevant) matrix coefficients can be minimised. The first solution will lead to a large set of specialised subprograms and the second will need a large set of storage method subprograms, but in either case the matrix handling may be coded very efficiently. In case iii, the gain in generality will be achieved at the expense of inefficient access to matrix coefficients.

For case ii, making the data type visible will give further problems. Only an array of REALs will be needed (together with some zero-dimensional objects containing information about the structure) for (square) matrix classes like:

full,
symmetric, and possibly (positive- or negative-) (semi-)definite,
triangular, possibly strictly triangular (or with unit diagonal),
banded (and again symmetric, etc.).

For sparse matrices, however, part of the storage will be needed for INTEGER indices, and also for access values when list structures are used with dynamic storage allocation.

As case i appears to be the most advantageous, we do not discuss the other cases further but recommend the use of different data types for different storage methods.

Note that documentation of library subprograms should contain sufficient information to allow a programmer to estimate the amount of working-space to be claimed for their execution. Moreover, this information should include similar information for all auxiliary subprograms which may be invoked. In the case of generic subprograms, the space used may depend upon the generic parameters (space required for REAL, etc.).

b) Use of parameters and generic parameters

When values of a parameter type occupy only a few storage units, it is immaterial whether or not copies are made for passing parameter values. However, if we assume here that the parameter passing mechanism is call-by-copying, then it is probable that in several cases superfluous copies will be made. For example:

Let the following declarations be valid:

```
type VECTOR is array (INTEGER range <>) of REAL;
function "+"(A,B : VECTOR) return VECTOR;
X1,X2 : VECTOR(1 .. M);
Y1    : VECTOR(1 .. N);
```

and consider the two cases:

i. mode **in**:

Calls of a function ZZ, declared by

```
function ZZ(A : in VECTOR) return REAL;
```

like

```
ZZ(X1(1 .. 2)), ZZ(X1 + X2) or even
ZZ(X1(1 .. 10) + Y1(3 .. 12)),
```

might implicitly give the following copies:

3 for each call of "+" (when each operand is copied, and the result must be stored),
1 for passing the parameter value to ZZ.

ii. mode **out** (or **in out**):

In this case actual parameters can only be (parts of) objects, so A + B is not a possible parameter, but for a subprogram declaration like

```
procedure WW(A : out VECTOR);
```

calls like

```
WW(X1);
```

can still involve copying twice and claiming extra working-space for 1 copy (the second time the parameter is only updated). Note that for **out** parameters, if the parameter passing is by copying, then when copying-out is taking place any components of the parameter which are not updated will be destroyed; so for array type parameters in general mode **in out** is to be preferred to mode **out**. See section (a) of Appendix A.

It should be made as simple as possible for (intelligent) compilers to decide when copying can be avoided. Therefore, aliasing with subprogram parameters should be avoided (even if the intended use of it would not make a program erroneous). See section (i), below, for implicit copying in general. A program is erroneous if its results depend upon the way in which an implementation passes composite-type parameters (i.e. by reference or by copying). It follows that the FORTRAN practice of passing working-space parameters to subroutines might not have the desired effect in Ada. This practice is therefore not recommended (see also section (c) of Chapter 6).

For generic parameters, the situation is different. Parameter association takes place upon elaboration of a generic instantiation (LRM 12.3), and the instance is a declaration containing the generic actual parameters as a fixed environment. For **in** parameters, the generic actual parameter can be expected to be copied. For **in out** parameters the actual parameters are to be used as variables by the instance, hence no copying should occur (LRM 12.3(8)). The association is explained as merely a renaming of variables (LRM 12.3.1). Other kinds of generic parameters do not affect the working-space.

c) Representation clauses

Type representation clauses (LRM 13) can be used to control the numbers of storage units needed for objects of some types. For a record type, for example, they can indicate the size and relative position of each distinct component within the total amount of storage needed for an object of this type. Expressions in such representation clauses may contain constants like SYSTEM.STORAGE_UNIT and SYSTEM.MEMORY_SIZE to obtain some degree of hardware independence.

These representation clauses might be useful for the definition of abstract data types, for which composite type definitions would otherwise waste too much storage space. However, their actual purpose is the reverse one, viz. to adjust type declarations to match available hardware types. Since this use is fully machine-dependent, type representation clauses should be avoided in the interests of portability. Address clauses should not be used either.

d) Use of relevant attributes and pragmas

For abstract data types defined as private types, the attributes:

FIRST_BIT, LAST_BIT, POSITION, SIZE, STORAGE_SIZE

can be used to estimate the size of the working-space needed. By using representation clauses (see the previous section) these attributes might even be controlled to some extent. Together with the constants STORAGE_UNIT and MEMORY_SIZE of the package SYSTEM, these attributes should make it possible to calculate, in advance, whether or not a subprogram can execute. However, the Ada language does not provide inquiry functions for obtaining the size of the free space dynamically, so the possibilities here are rather limited.

The pragma PACK may be used for instructing an implementation to minimise gaps in storage areas for all objects of some composite type, especially if the area for the components has already been restricted by the pragma PACK or by representation clauses. This applies in general to record types. It should not be expected that an array of BOOLEANs will be packed in the same way as in many implementations of Pascal for the type PACKED ARRAY [subrange] OF boolean;.

There may be some use for the pragma STORAGE_UNIT, but it is hardly possible to give general advice here. Its function is to initialise the constant STORAGE_UNIT in the package SYSTEM, and the meaning of this constant is the number of bits per storage unit. If the installation value would cause many gaps in storage for composite-type objects, then perhaps better values for SYSTEM.STORAGE_UNIT might be found, but we expect this situation to be very exceptional. We note that the use of this pragma does not influence the hardware representation of standard types.

For the effects of

pragma OPTIMIZE(SPACE);

one should consult the implementation reference manuals. Code including this pragma will certainly not be portable.

e) Running system (storage overheads for Ada style declarations)

For the claiming of large storage areas one can choose array types, record types containing array-type components or dynamic data structures like lists, trees, etc., created using access types. (Little can be said about the use of files except that there will probably be some implementation-dependent working-space for file buffers.)

Array objects will require extra space for their dope vectors (or other descriptors) so that an array of one-dimensional array-type components will probably require more space than an equivalent two-dimensional array.

If record types are used with array-type components, with the aim of forcing a lower bound of 1 on the index of each array-type object, as in:

```
type ANON_VECT is array (INTEGER range <>) of REAL;
type VECTOR(SIZE : NATURAL) is -- indexing from zero
   record
      ELEM : ANON_VECT(1 .. SIZE); -- null vector if SIZE = 0
   end record;
```

then discriminants may again require some space. Moreover, if a discriminant controlling the size of an array-type component has a default (the effect being that values of different sizes can be assigned to such objects), it can be imagined that these objects will always occupy some minimal space. The same applies to discriminants selecting some record variant.

The overhead for access types for dynamic data structures is obvious (see section (f) below).

Additional (range) constraints for individual objects (i.e. objects of anonymous subtype) will also use extra space. A declaration like

```
X1 : array (A .. B) of RESTRICTED_REAL;
```

or better:

```
type VECTOR is array (INTEGER range <>) of RESTRICTED_REAL;
X1 : VECTOR(A .. B);
```

(assuming that: **subtype** RESTRICTED_REAL **is** REAL **range** C .. D;) is therefore preferable to:

```
X1 : array (A .. B) of REAL range C .. D;.
```

Whether types are private (or not) should not influence the working-space during execution, although access to objects of such types will be more laborious.

f) Use of the heap

We consider here two topics:

- dynamic storage allocation and
- storage management in real-time programming.

i. Dynamic storage allocation.

Dynamic storage allocation is obtained by allocators for objects of some access type (LRM 4.8). The effect of an allocator is that sufficient working-space is claimed for storing values of the base type. This space remains "claimed" by the program as long as objects of the access type give access to it. So the preservation of such storage places need not be related to the block structure of the executing program. The storage becomes "free" or "garbage" when no objects have access to it any more, and this occurs when either:

I. other access values or **null** are assigned to all objects that formerly had access to the storage,

II. the appropriate instance of UNCHECKED_DEALLOCATION (LRM 13.10.1) is called for one object, and other objects that had the same access value no longer use this access, or

III. values of the access type become inaccessible, through control leaving the unit containing the access-type declaration.

The danger of dynamic storage allocation is that either garbage storage is not reclaimed, when new storage claims are made, or it is expensive to find out which storage can be reclaimed. Deallocating storage by method I is expensive - either in working-space, because the storage is not reused, or in time, because it is not easy to discern that such storage can no longer be accessed. The explicit returning of storage by method II is unsafe, as it does not guarantee that deallocated storage will not be used via other access objects. Finally, recycling of storage is difficult if later storage claims require storage units of a different size from those of the deallocated storage.

Since the use of dynamic storage allocation may cause very inefficient use of the whole working-space, it should be used with great care in scientific libraries. Although a garbage collector is not necessarily available in Ada (LRM 4.8(7)), its presence is desirable since it simplifies the use of allocators (see section (b) of Appendix A). Dynamic storage allocation can be used in library subprograms if the claims by a subprogram are not (i.e. cannot be) intermingled with claims by the user and all storage can be reclaimed afterwards (see method III above). Otherwise, if dynamic storage must be given to the calling user program, then the access type should be **limited private** to the user (thus preventing the user from copying accesses) and the package containing the type declaration should also provide an instance of UNCHECKED_DEALLOCATION for explicitly returning storage by the user.

Note that making the access type **limited private** does indeed prevent the user from making copies, but this does not imply that copies cannot be made at all. Parameter passing for scalars and access types is done by copy-in, copy-out for mode **in out**, and appropriately for other modes (LRM 6.2(6)). Moreover, for private types the parameter passing mechanism is the same as for the corresponding full types (LRM 6.2(8)). Thus, though a type may be **limited private**, when an object of this type is passed as a parameter, a copy is made. Problems may arise in two cases. First, if an object is accessible both as a parameter and as a global variable, the results may be unpredictable; however, this kind of aliasing should always be avoided. A much more serious problem arises in the presence of exceptions, as illustrated by the following example:

```
package STORAGE_MANAGER is

   type STORAGE is limited private;

   procedure ALLOCATE(P : in out STORAGE);
   -- The body is roughly as follows:
   -- begin
   --    if P /= null then
   --       DEALLOCATE(P);
   --    end if;
   --    P := new STORAGE_CELL;
   -- end ALLOCATE;

   procedure DEALLOCATE(P : in out STORAGE);
   -- The body is roughly as follows:
   -- begin
   --    P.REF_COUNT := P.REF_COUNT - 1;
   --    if P.REF_COUNT = 0 then
   --       UNCHECKED_DEALLOCATION(P);
   --    end if;
   -- end DEALLOCATE;

   procedure COPY(FROM : STORAGE; TO : in out STORAGE);
   -- The body is roughly as follows:
   -- begin
   --    if TO /= null then
   --       DEALLOCATE(TO);
   --    end if;
   --    TO := FROM;
   --    if TO /= null then
   --       TO.REF_COUNT := TO.REF_COUNT + 1;
   --    end if;
   -- end COPY;

   procedure STORE(P : STORAGE; I : INTEGER);

   function FETCH(P : STORAGE) return INTEGER;

private

   type STORAGE_CELL is
      record
         REF_COUNT : INTEGER := 1;
         CONTENTS  : INTEGER;
      end record;

   type STORAGE is access STORAGE_CELL;

end STORAGE_MANAGER;
```

```
   with SEQ_INT_IO; -- instance of SEQUENTIAL_IO
   with STORAGE_MANAGER;
   use SEQ_INT_IO, STORAGE_MANAGER;
   procedure READ_ITEM(FILE : in FILE_TYPE; P : in out STORAGE) is
      I : INTEGER;
   begin
      ALLOCATE(P); -- new entry on heap
      READ(FILE, I); -- read integer entry
      STORE(P, I); -- store in P
   exception
      when END_ERROR => -- no more entries on file
         DEALLOCATE(P); -- P no longer needed
         raise; -- signal end to caller
   end READ_ITEM;
```

Though it may not be apparent at first sight, this example will not work, on account of the use of the **raise** statement to signal the end of data to the caller. The language does not guarantee that if an exception occurs (as is the case here) then copy-back will be performed. Thus, at the end of the data on the file, the actual parameter to READ_ITEM will become unreliable (it will contain a "dangling" reference) and may no longer be used.

Further, implicit declamation of storage (i.e. removal of all accesses to it) can be avoided by using the pragma CONTROLLED (see below). This virtually prevents inefficient garbage collection for the attentive user, especially if the package itself includes some bookkeeping of freed storage.

According to the LRM 13.10, 13.10.1, a storage declaiming procedure can be made for every access type by instantiating the predefined generic library procedure:

```
   generic
      type OBJECT is limited private;
      type NAME is access OBJECT;
   procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

For any type declaration such as

```
   type LINK is access CELL; -- for some type CELL
```

a generic instantiation can be given, thus:

```
   procedure FREE is new UNCHECKED_DEALLOCATION(CELL, LINK);
```

Then a call:

```
   FREE(LINK_VARIABLE);
```

will deallocate the storage for the object designated by LINK_VARIABLE.

One can prevent the automatic storage reclamation for all objects of a type designated by one access type, by giving the pragma CONTROLLED (LRM 4.8) immediately after the access type declaration. Then the storage will only become free when the unit containing the access type declaration is left (method III above). If this is always correctly used a garbage collector is no longer needed.

ii. Storage management in real-time programming.

In real-time situations, the interrupts which can be caused by sudden garbage collections may be unacceptable for the running processes. In the first place, use of the heap should be avoided. (It is uncertain when a particular implementation might want to use the heap, but in general the programmer should abstain from the use of access types, and also of record types with defaults for discriminants that are used for constraints on array-type components.) If, however, the programmer must use access types, then he can produce as little garbage as possible by keeping superfluous storage cells in a "free list" and reissuing them to access-type objects whenever requested. This might imply that all free storage cells should have the same type and subtype (the same discriminant values) or that several free lists should be kept. Of course, a free list cannot be kept beyond the scope of the variable containing the head of the list but, if this unit is to be left, then UNCHECKED_DEALLOCATION can be used. We add a warning that this practice is error prone and should be avoided if possible.

g) Machine architecture

Special architecture of machines can greatly influence the choice between different algorithms and may also affect implementation decisions (which might themselves further influence choice of algorithms). Such architectural properties might be those of

paging machines,
vector processors or
distributed systems.

With respect to working-space, a choice could be the storage of matrix components by rows or by columns, whereupon the processing of the complete matrix would be performed with different efficiencies on the various machines. In Ada, it seems highly probable that the storage of two-dimensional arrays will be implemented row-wise but a programmer might still think it wise to store matrices transposed in two-dimensional arrays. We would like to encourage implementations that allow the user to choose the way of storing two-dimensional arrays. Users of interfaces to FORTRAN subroutines would be greatly helped by this feature (see section (a) of Appendix A).

In order to minimise thrashing on a paging machine, algorithms should be implemented in such a way that storage locations holding elements of vectors and matrices are referenced sequentially. For example, an LU-decomposition (producing rows of the upper-triangular matrix U and columns of the lower-triangular matrix L, or perhaps also rows of the latter but with a different order of storage for intermediate results) might be preferred to a Gaussian elimination when the number of matrix coefficients exceeds the size of a page; deciding what is best can be complicated. Similarly, computing A * x or A(transpose) * x might require different storage methods or different algorithms, while, on a vector processor, the latter case would impose completely different requirements on the implementation (see section (f) of Chapter 9 for further discussion).

Another example concerning storage is connected with a vector of complex numbers. The question arises as to whether this should be a vector of complex-type components:

```
type CO_VECTOR is array (INTEGER range <>) of COMPLEX;
```

or whether it should consist of a vector of real parts and a vector of imaginary parts:

```
type CO_VECTOR (SIZE : INTEGER) is
   record
      RES, IMS : VECTOR(1 .. SIZE);
   end record;
```

For the reason given in section (e) of Chapter 5, the former of these possibilities is recommended. We note that our main aim is to provide (guidelines for writing) specifications of packages and their constituents. These specifications should appear as natural as possible to the user. Implementors may provide different bodies for machines with different architectures, and we can provide hints for their labours, but it is not possible to imagine all bodies, e.g. bodies for which a type MATRIX may even be a task type.

h) Use of generics and subunits

In this section we discuss topics which deal with the size of the working-space occupied by a loaded and executing program. These topics are:

- the use of shared code for different instances of the same generic package and
- the possibilities of partial loading.

i. The use of shared code.

It is clear that if an implementation duplicates the code for each instantiation of a generic package, this will lead to a waste of space. Take for example a zero-finding subprogram that requires a function parameter. In Ada we are forced to make the zero-finder a generic subprogram (see section (a) of Chapter 6). Now if more than one instance of that subprogram is made, we find ourselves with multiple copies of one and the same subprogram, differing only in the calls of the actual supplied function. In the case of a simple zero-finder this might not lead to trouble, as for a simple function such a subprogram will be quite short. However, the problem will become serious if the subprogram concerned is not a simple one but perhaps a package for solving differential equations or some yet more complicated problem. One way to overcome this difficulty is through the concept of reverse communication (see section (d) of Chapter 6), in which case the subprogram provided by the library performs one step only, and the caller is required to call the subprogram often enough to obtain a fair answer to his problem (then calls of the subprogram defining the problem are made by the user, hence the problem-solving subprogram need not be generic). However, this is not the solution we require; indeed, it avoids the problem rather than solving it.

On the other hand, in some cases it might be preferable to use multiple copies of the code for multiple instances of the generic. This is especially true if the generic parameter is, for example, an abstract floating-point type, on which basic operations are performed by calls to routines, which must be provided as generic parameters along with the type, rather than by machine instructions. In this case, repeated use of the same piece of code could lead to huge overheads in time.

As, up to now, it is not clear what different implementations of Ada will do with instances of generics, any further discussion of this topic here could well prove to be premature.

ii. Partial loading.

The concept of partial loading also has a forceful impact on the space requirements for a program. Suppose a package is defined with many subprograms, some of which are always needed, while others are needed only in special cases. If, in this case, all modules are always loaded into memory, this leads to waste of space (except perhaps on some machines using virtual memory, where library routines are stored in shared instruction space).

Now, very sophisticated systems may be able to load only those parts of a program that are actually needed, but we believe that most systems will require assistance when selecting the loadable parts. The major feature of Ada which should help in this matter is the concept of separate compilation. It is anticipated that if all modules within a package are compiled separately, using a body stub in the package body, most systems will be able to detect the parts to be loaded.

Note that it is not permitted for designators of subunits to be operator symbols (see section (d) of Appendix A). However, this difficulty may be circumvented by the following construct (unfortunately introducing a new identifier):

In the package declaration:

```
function ADD(A,B : A_TYPE) return A_TYPE;
function "+"(A,B : A_TYPE) return A_TYPE renames ADD;
```

in the package body:

```
function ADD(A,B : A_TYPE) return A_TYPE is separate;
```

and as a subunit:

```
separate (A_PACKAGE)
function ADD(A,B : A_TYPE) return A_TYPE is
begin
   -- sequence of statements
end ADD;
```

In conclusion, we cannot be sure that Ada programs will be processed in this way. Hence, as a general recommendation, we advise that packages should be kept fairly small by combining only closely related subprograms, all of which are needed in most cases. Moreover, the bodies should be compiled separately (i.e. with body stubs and subunits) to give aid to those systems which are particularly sophisticated.

i) Implicit copying

As has already been indicated in section (b) above, implicit copying of values may be invoked by implementations, possibly together with the claiming of extra working-space. The main situations are:

i. Type conversion:

For numeric types the effect on working-space is negligible. For array types no implicit type conversion of the components is allowed (LRM 4.6). If the components have different subtypes, extra checks can be made if the subtypes differ in range constraint (for numeric type components), otherwise they should be convertible (for array type components). Hence, we do not expect copying to occur here.

ii. Assignment statement:

Again only composite-type objects and values are considered. An assignment might be implemented by copying to guarantee the correctness of, for example,

```
X1(5 .. 9) := X1(3 .. 7);
```

and also (assuming a vector-"+") of

```
X1 := X2 + X3;
```

Here the "+" requires extra storage for delivering the result but, hopefully, the assignment will not make an extra copy before copying into the storage of X1 (see section (b) of Appendix A).

iii. Parameter passing:

It is clearly stated (LRM 6.2(7)) that the language does not define which of the two mechanisms (call-by-copying or call-by-reference) should be adopted by implementations for the passing of composite-type values, nor indeed whether an implementation should be consistent (in the chosen mechanism).

If the mechanism is call-by-copying, then a subprogram will have extra storage for each parameter passed. A copy-in is made upon subprogram entry, and for out and in out parameters, at the return, a copy-out is made (though possibly not for an abnormal exit). See section (a) of Appendix A.

We conclude with an example in which copying is highly probable, even if the prevailing parameter-passing mechanism is call-by-reference. Consider the declarations (cf. LRM 13.6):

```
type DESCRIPTOR is
   record
      -- components of a descriptor, e.g.
      ELEM : DESCR_COMP;
   end record;

type PACKED_DESCRIPTOR is new DESCRIPTOR;
```

```
   for PACKED_DESCRIPTOR use
      record
         -- component clauses for some or for all components
      end record;

   X : PACKED_DESCRIPTOR;

   procedure USE_DESCRIPTOR(Y : in out DESCRIPTOR);

   procedure USE_DESCR_COMP(Z : in out DESCR_COMP);
```

and the following calls:

```
   USE_DESCRIPTOR(DESCRIPTOR(X));
   USE_DESCR_COMP(X.ELEM);
```

Unlike Pascal, Ada does not prohibit this kind of parameter passing but it cannot be performed without copying (-in and -out).

9. REAL-TIME ENVIRONMENT

In a real-time processing environment, new problems arise in the design of large scientific libraries. These concern:

- the need for scientific calculations to be performed during running processes, which cannot themselves be interrupted for this purpose and which cannot be kept waiting indefinitely for the results, and

- the possibility of designing and implementing new algorithms for use on multi-processor systems.

For the first class of problems, several questions must be considered, such as:

- Will the calling task (i.e. the process that requests a calculation) be suspended during the calculation?

- Can the calling task obtain advance information about the computation time required?

- Can the computation be performed without interrupting the calling task (assuming that a separate processor is available for the required computation), and if so, will the result become available to the calling task in the permitted time?

- In the latter case (for which we assume a "mailbox" construct to be the most useful), will one result (possibly not very accurate, but a result) become available, or will the task performing the calculation continue to put improved results in the mailbox as long as the calling task does not destroy the mailbox?

With reference to the second class of problems, we note that algorithms for distributed computation will be highly dependent upon machine architecture and we question whether Ada is the appropriate language for describing such algorithms.

An overall problem is the action to be taken in the event of an exception (already addressed in general in Chapter 7) when the exception is a hardware failure (graceful degradation) or a raised NUMERIC_ERROR.

The above subjects are discussed in the following sections.

a) Libraries for real-time use

For libraries to be used in a real-time processing environment, requirements for this kind of processing must have precedence over those for batch-processing. These requirements usually stem from the fact that a running process (issuing a calculation request) cannot itself be interrupted, or can be kept waiting for only a limited (and probably very small) period ("duration"). Therefore, such a process should not call a library subprogram at all, unless it (or, more precisely, its programmer) knows in advance when the answer will become available and that the response time will be acceptably short. Aspects of particular importance are:

i. duration of a calculation,

ii. documentation of the duration for calls of a library subprogram,

iii. reliability with respect to getting an answer and getting it within the promised period.

Considering the duration of (scientific) subprograms, we can distinguish three classes of these:

A. Those for which the computation time is essentially constant. Standard arithmetic, basic mathematical functions and most of the special mathematical functions belong to this class.

B. Those for which the computation time depends upon the size of the problem. We have in mind here most of the vector and matrix manipulations, and methods for which the computation time is a simple function of the accuracy demanded.

C. Those for which the computation time depends upon the data of the problem (and possibly also upon the size of the problem).

Correct and clear information in the subprogram documentation, which is of course a general requirement and not only one for subprograms used by real-time processes, is obviously indispensable here. As for reliability, documentation must be abundantly clear about the nature of the results in exceptional situations (NUMERIC_ERROR raised, singular matrix, required accuracy not obtained, etc.).

We foresee that (subprogram bodies in) scientific libraries for use in a real-time environment will often be different from those for use in batch processing. Probably the only packages that can be shared by both libraries will be standard instantiations of the GENERIC_MATH_FUNCTIONS package (Chapter 4). The execution time for all mathematical functions is fixed and negligibly small (at least we expect this time to be short enough for calls by on-line processes). It is unlikely that other packages of related scientific subprograms will contain only entities that belong to class A, since the above subdivision into three classes does not coincide with any usual structuring of scientific libraries. For many algorithms belonging to the classes B and C the computation time may turn out to exceed the allowed response time. In such cases algorithms written for use in batch-processing will have to be adapted to satisfy the requirements of on-line use. It follows that in general services requested by tasks should be rendered by tasks (to ensure synchronization).

Especially in real-time processing, there may be some demand for mathematical functions for fixed-point types, but a separate package is not needed here if floating-point arithmetic is available, since type conversion is allowed (LRM 4.6(7)). One important reason for designing separate packages for most other scientific problems is that the relationship of a calculation to a calling task (which may possibly accept a less accurate answer at a certain moment, or allow for the updating of a previous inaccurate answer) will lead to the selection of different methods. Examples are given in section (e) below.

b) Use of language features regarding tasks

An executing process, described by a "task" (LRM 9), can call a library subprogram when it needs some scientific calculation. In the present context such a subprogram might well be replaced by another task whose "entry" can be called (this task is sometimes called a "server task"). This allows greater freedom in the use of such an auxiliary unit, e.g. the calling process may continue its own execution if it is known that the required answer will come back at a specified moment. Later, in section (e), we present some examples for several practical situations. Here, we summarise the language tools.

"Entries" (LRM 9.5) are the principal means of communication between tasks. An entry (perhaps from a family of entries) can be called in the same way as a procedure is called. This may cause the calling process to be suspended, viz. if the entry call is not immediately accepted by the task whose entry it is. However, the caller may decide to cancel the call if it waits too long: "timed entry call" (LRM 9.7.3) or to issue the call only if the task with the entry is ready for accepting the call: "conditional entry call" (LRM 9.7.2).

On the other hand, a task can wait (at an "accept statement" (LRM 9.5)) till it receives an entry call for its entry, or it can cycle along a series of accept statements until one of its entries is called ("selective wait"). If none of its entries is called it can decide to do something else or it can cancel its waiting for entry calls if it waits too long ("delay alternative") (LRM 9.7.1).

If an entry call is accepted, then the caller and the called task are synchronized (they have a "rendezvous" till the end of the accept statement). They can communicate by means of the parameters passed by the entry call, which can be used in the sequence of statements of the accept statement. Even if this communication is empty, there has still been an instant of synchronization.

In the example in section (f) below, every SORTER waits at a WAKE_UP accept statement, until this entry receives a call. In the rendezvous it obtains the index of the start position in the array X. Next it calls the SEIZE entry of the GUARD of an array component. The GUARD will only accept this entry call if the GUARD has not already been SEIZEd by another SORTER. Otherwise, it can only accept a RELEASE, and care has been taken that this RELEASE will only be called by the SORTER that SEIZEd (this should have been ensured by issuing and checking secret permissions).

Tasks start executing when their declaration is elaborated and they terminate (approximately, see LRM 9.4(6)) when their sequence of statements has been performed. Alternatively, they may terminate at a "terminate alternative" in a cycle of accept statements, if their entries can no longer be called. Tasks can also be aborted, but this should be done only in extreme circumstances (LRM 9.10(10)).

Attributes T'CALLABLE and T'TERMINATED (for any visible task T) can be used to inquire after the status of a task. The attribute E'COUNT (for an entry E of a task T) can be used inside the body of T to obtain the number of E entry calls that are waiting for an accept statement. If several entry calls for the same entry are waiting, they are always accepted in the order of arrival (LRM 9.5(15)), notwithstanding the possibly different priorities of the calling tasks.

A task may (but need not) have an associated priority, which is implementation-dependent (LRM 9.8). Such priorities can affect the order of allocation of processing resources to parallel tasks. In scientific programs the results of a computation (obtained from ser er tasks) should not depend upon the scheduling of tasks which may execute in parallel, or the program will be erroneous. Therefore priorities are of little use here, though it is expected that running processes which require on-line calculations will invariably have higher priorities than their server tasks.

c) Variables shared by tasks

A variable is "shared" by two tasks if it is accessible to both (LRM 9.11(2)). If the two tasks read or write such a shared variable, then nothing is known about the order in which they perform their operations, unless the two tasks are synchronized by a rendezvous. If the result of a computation depends upon an unknown order of performed operations, the program is erroneous; therefore proper synchronizations must be used.

Synchronization of two tasks is needed not only if the tasks have to meet, to communicate information to each other, but also if the tasks have to avoid each other, because they both need to use the same accessible variable. With reference to the latter case, it would appear that shared variables can be partially updated, e.g. a floating-point variable might have its mantissa updated but its exponent not yet, when the contents are read by another task. Such uncertainties cannot be allowed in scientific computations; we therefore strongly recommend that implementations make the updating of numeric-type objects an indivisible operation (see section (a) of Appendix A).

The first kind of synchronization is simply achieved by direct communication, i.e. one task calls an entry of the other to receive its latest information. The second case, however, when two tasks must avoid each other, is more complicated.

In the elaborate GENERIC_SORT example in section (f) below, a SORTER (task) may only read an array component if its right SORTER neighbour is finished with it. However, it must also be ensured that an update issued by the right neighbour has effectively been performed on the shared variable, not only on a local copy (see LRM 9.11(8)). This is accomplished by performing all accesses to the array through a special UPDATES task and by locking array components for use by one SORTER at a time. The guaranteed order of accesses and updates is as follows:

```
step 1: UPDATES.PUT into X(I);  by right Sorter,
step 2: RELEASE;    to GUARD(I)  by right Sorter,
step 3: SEIZE;      to GUARD(I)  by  left Sorter,
step 4: UPDATES.GET from X(I);  by  left Sorter.
```

Here, the right Sorter guarantees the order 1 - 2, GUARD(I) guarantees the order 2 - 3, the left Sorter guarantees the order 3 - 4, and the access via UPDATES guarantees that step 4 delivers the value that was passed to X(I) in step 1.

Note that this result would not have been guaranteed if the four steps had been:

```
step 1: X(I) := ITEM;              by right Sorter,
step 2: RELEASE;  to GUARD(I)      by right Sorter,
step 3: SEIZE;    to GUARD(I)      by  left Sorter,
step 4: ITEM := X(I);              by  left Sorter.
```

In this case, the assignment statement X(I) := ITEM; in step 1 need not be effected in the shared variable itself before step 4 takes place. The Ada language offers a means of enforcing this updating, synchronous with the assignment statement, by:

```
pragma SHARED (variable_simple_name); -- (LRM 9.11(9))
```

the effect of which is to make every use of the named variable a synchronization point. However, the applicability of this pragma is restricted to certain variables of scalar or access type (LRM 9.11(10,11)).

Our advice here is that tasks should not access shared variables for simultaneous reading and updating. The best solution to this problem is to perform all accesses through a central task. We do not object to direct access of shared variables for reading purposes only (i.e. shared data).

d) Exceptions

Exceptions can be raised during the activation of a task or they can be raised in or propagated to activated tasks.

If an exception is raised during the activation of a task (i.e. the elaboration of the declarative part of the task body), the task becomes "completed" and the exception TASKING_ERROR is raised (in the surrounding frame) (LRM 9.3(3,7)).

If an exception is raised in or propagated to a task body, and the task does not handle the exception, the task becomes completed and TASKING_ERROR is raised at the point of activation of the task (i.e. at the first **begin** of the body containing the task body in its declarative part, or at the place where the allocator is evaluated for an access variable accessing a task type). TASKING_ERROR is also raised if an entry of a completed task is called (or if the task is completed before the entry call is accepted).

If an exception is raised during a rendezvous (i.e. in an accept statement) the exception propagates to the calling task and also to the control point following the accept statement in the called task (LRM 11.5). TASKING_ERROR is raised in the calling task if the called task is aborted during the rendezvous. Termination of a calling task during a rendezvous (by an abort statement) is not perceived by the called task: it completes its rendezvous with a "ghost" (to quote Barnes (1982, p.228): "If the customer dies, too bad - but we must avoid upsetting the server").

For the use of exceptions, our general recommendations of Chapter 7 apply. However, in real-time processing one has to be especially careful. Exception handlers should always be provided, unless the exception (usually TASKING_ERROR) concerns a design error, such as caused by a call of an entry of a completed (or abnormal) task.

The possibility of TASKING_ERROR being raised is diminished if exception handlers are provided for all critical situations. Therefore, if computations can fail, an exception handler should be given, especially inside every accept statement (for correctly ending the rendezvous) and in every eternal loop of a server task as long as its entries can be called. As a side remark, we note that answering the calling task by raising an exception during a rendezvous (provided that the calling task expects this reaction) still has the disadvantage that the exception is also raised in the server task, which would require a trivial exception handler in a block statement surrounding every accept statement. While this could be provided, we do not recommend the raising of an exception as a regular way of communication between a task and its user.

Not only should all expected exceptions (like NUMERIC_ERROR) be handled, but also unexpected exceptions such as STORAGE_ERROR. Even if the cause of this error cannot be removed (and the exception may soon be raised again by the system), the possibility should still be recognised, because the calling program itself must usually continue in any case. (The latter should stop requesting the service that caused the raising of STORAGE_ERROR but should be allowed to accomplish its own service in some truncated form: "graceful degradation" of a real-time system.) Of course, the calling task should be informed that it need not request further services.

Finally, here, we indicate one means of avoiding (but not completely) the calling of an entry of a completed task. One may first enquire whether the task is callable, like (see section (b) above):

```
if SERVER'CALLABLE then        -- using attribute CALLABLE
   SERVER.START_COMPUTATION (X);
end if;
```

Unfortunately, SERVER may terminate between the enquiry and the entry call. This cannot be solved by a conditional entry call, while for a timed entry call the above mismatch is even more likely (the server task may be completed before the waiting task is timed out). We do not like the solution of an exception handler following each entry call. In most cases it is a matter of algorithm design: server tasks should be eternal (see examples in sections (e) and (f) below). Another solution might be synchronization of the completing of tasks, as for shared variable updates (see section (c) above).

e) Calculations by server tasks

In this section, we present several examples of the use of tasks where:

- the task requesting some service can wait for some time,
- the task requesting some service is not suspended,
- the task requesting some service is not suspended and the server task will provide a series of answers with increasing accuracy,
- as a detail, we will assume that a server task can give information about the computation time needed to finish its execution successfully.

i. The calling task can be suspended.

In this case, the language tool is direct communication, i.e. the calling task has a rendezvous with the server task. It should be possible to inform the server task of the allowed time and this might save time if the server task replies at once that it cannot make it. Example:

In the calling task:

```
SERVER.JOB(IN_VALUE, RESULT, TIME_ALLOTTED, CANNOT_BE_DONE);
if CANNOT_BE_DONE then
   ALTERNATIVE_COMPUTATION;
end if;
```

In the task body of SERVER:

```
loop
   select        -- to allow several calls of JOB
      accept JOB(X : in REAL; ANSWER : out REAL;
            ALLOWED : in DURATION;
            I_CANNOT : out BOOLEAN) do
         if ALLOWED > WHAT_I_NEED then
            I_CANNOT := TRUE;
         else
            I_CANNOT := FALSE;
            ANSWER := LOCAL_FUNCTION(X);
         end if;
      end JOB;        -- end of rendezvous
   or
      terminate;
   end select;
end loop;
```

The caller cannot abort the server task if it does not deliver the answer in the allowed interval, since it does not execute statements before the rendezvous is completed (but see ii below). The caller should have an alternative of its own, if a server cannot do the computation.

It is assumed that a physical processor is immediately available for the server task and that the server task is not interrupted by the task scheduler; otherwise it would be difficult to estimate the time needed. A timed entry call may be used if resources for the server task are not guaranteed. Example:

In the calling task:

```
select
   SERVER.JOB(IN_VALUE, RESULT, TIME_ALLOTTED,
      CANNOT_BE_DONE);
   if CANNOT_BE_DONE then
      ALTERNATIVE_COMPUTATION;
   end if;
or
   delay SOME_TIME;
   ALTERNATIVE_COMPUTATION;
end select;
```

The above example applies also to the situation where the server task is engaged in another rendezvous. If this occurs frequently and if enough physical processors are available, it may be avoided by creating several copies of the server task (using a task type).

We refer to section (f) of Chapter 8 for the case where interrupts are caused by the activation of a garbage collector. Actually, we do not expect a garbage collector to be allowed to overrule a vital process, hence many installations may decide not to offer such a service. To avoid STORAGE_ERROR being raised too soon, the user should tidy up his own garbage storage space.

We note that an unconditional entry call and a procedure call look alike and that one might decide to call a subprogram instead of an entry of a server task. The difference, however, is (cf. Barnes, 1982, p.204) that in the case of a procedure the caller is executing the procedure body, whereas in the case of an entry the server task must execute the statements (the body is now an accept statement), presumably using its own processor. We can even imagine that the processor executing the calling task is completely dedicated to this task and is not able to perform a scientific calculation. We conclude that a task is the most appropriate tool for handling a request for auxiliary computations.

ii. The calling task is continuing its execution.

If the server task is ready for an accept statement (otherwise see i above), the calling task might execute the statements:

```
SERVER.START_COMPUTATION(IN_VALUE);
OTHER_ACTIONS;     -- by the calling task, finished after a
                   -- certain time, or using a delay statement
                   -- if more time is permitted to the server task.
select       -- a conditional entry call:
   SERVER.DELIVER(RESULT);
else
   SERVER.CANCEL;
   ALTERNATIVE_COMPUTATION;
end select;
```

The server task might read:

```
task SERVER is
   entry START_COMPUTATION(X : in REAL);
   entry DELIVER(RESULT : out REAL);
   entry CANCEL;
end SERVER; -- specification

task body SERVER is
   X_READ, LOC_RESULT : REAL;
begin
   loop
      select        -- for every service request
         accept START_COMPUTATION(X : in REAL) do
            X_READ := X;
         end START_COMPUTATION;    -- end of first rendezvous

         declare
            READY : BOOLEAN := FALSE;

            task LOCAL_SERVER; -- specification
            task body LOCAL_SERVER is
            begin
               LOC_RESULT := LOCAL_FUNCTION(X_READ);
               READY := TRUE;
            end LOCAL_SERVER; -- body

         begin
            loop
               select
                  when READY =>
                     accept DELIVER(RESULT : out REAL) do
                        RESULT := LOC_RESULT;
                     end DELIVER;
                     exit;
               or
                  accept CANCEL;
                  -- Stop the Local Server (omitted)
                  exit;
               else
                  null;
               end select;
            end loop;
         end; -- of block statement
      or
         terminate;
      end select;
   end loop;
end SERVER; -- body
```

Here we have introduced a local task for doing the calculation, finally delivering the result in a variable of the server task (assuming that this (shared) variable would be updated in time). After accepting a START_COMPUTATION call the SERVER waits selectively for entry calls of DELIVER or CANCEL. We have omitted an elegant termination of the local task if a CANCEL is received and the synchronization of the updates of READY and LOC_RESULT. Note also that the SERVER task cannot serve another task before the calling task has collected the answer (if it might never do so, then the inner loop of SERVER should contain a terminate alternative).

iii. The calling task continues its execution while the server task delivers a series of answers.

In the previous example we used a local task for the calculation that would be performed concurrently with the calling task. Another solution may be obtained by first calling the START_COMPUTATION entry of the server task and by allowing the server task to call a RECEIVE entry of the calling task for sending the answer. As correctness with respect to "deadlocks" is more difficult to prove if there is no clear hierarchy of tasks concerning "caller" and "called", we prefer to avoid this way of programming.

A better solution is by the creation of an "agent" task, usually called a "mailbox", which can receive a result (or in the following example a succession of results) from the server task and which can be inspected by the calling task whenever necessary. Use of a task type for this agent permits the creation of a distinct mailbox for every request of a computation. For more details we refer to Barnes (1982, pp.225-227). When properly used, this construct solves several minor problems that were touched upon in the previous discussion, such as:

- shared variable update of the result (see also section (c) above),

- no task can collect an answer requested by another task,

- other tasks can be served before the requesting task collects its (final) answer.

In the following example the calling task asks for a result with a certain precision, and it can send a signal that no further (more accurate) answers are needed. The server task receives the identity of a mailbox and it puts successive results with known accuracy into it, until it cannot improve the result further or until a closing signal is received.

```
-- The task type might be given in a package that has ITEM
-- as its generic parameter (type ITEM is private;).
-- Here we assume:

type ITEM is
   record
      FX, ACCURACY : REAL;
   end record;

-- The following order of declarations and bodies is not
-- in agreement with the Ada syntax, but for clarity we give
-- every task body immediately after its specification.

task type MAILBOX is
   entry DEPOSIT(X : in ITEM; READY : in BOOLEAN;
      REQUEST_ENDED : out BOOLEAN);
   entry COLLECT(X : out ITEM; READY : out BOOLEAN);
   entry CANCEL;
end MAILBOX; -- specification

task body MAILBOX is
   LOCAL : ITEM;
   DEPOSED : BOOLEAN := FALSE;
   SERVER_READY, CUSTOMER_GONE : BOOLEAN := FALSE;
begin
   loop
      select
         accept DEPOSIT(X : in ITEM; READY : in BOOLEAN;
               REQUEST_ENDED : out BOOLEAN) do
            LOCAL := X;
            SERVER_READY := READY;
            REQUEST_ENDED := CUSTOMER_GONE;
         end DEPOSIT;
         DEPOSED := TRUE;
      or
         when DEPOSED =>
            accept COLLECT(X : out ITEM; READY : out BOOLEAN) do
               X := LOCAL;
               READY := SERVER_READY;
            end COLLECT;
            DEPOSED := FALSE; -- can be deleted
      or
         accept CANCEL;
         CUSTOMER_GONE := TRUE;
      else
         exit when CUSTOMER_GONE and SERVER_READY;
      end select;
   end loop;
end MAILBOX;

-- If the Customer dies without signalling to the mailbox,
-- this might cause the raising of TASKING_ERROR.

type ADDRESS is access MAILBOX;

task SERVER is
   entry REQUEST(A : in ADDRESS; X : in ITEM);
end SERVER; -- specification

task body SERVER is
   REPLY : ADDRESS;
   JOB_X, JOB_FX : ITEM;
   ACC_REQUEST : REAL;
   ENDED : BOOLEAN := FALSE;
begin
   loop -- for every request
      select
         accept REQUEST(A : in ADDRESS; X : in ITEM) do
            REPLY := A;
            JOB_X := X;
         end REQUEST;
         ACC_REQUEST := JOB_X.ACCURACY;
         -- Work on job:
         loop
            LOCAL_ITERATION(JOB_X, JOB_FX);
            exit when JOB_FX.ACCURACY <= ACC_REQUEST;
            select
               REPLY.DEPOSIT(JOB_FX, FALSE, ENDED);
               exit when ENDED;
            else
               null;
            end select;
         end loop;
         REPLY.DEPOSIT(JOB_FX, TRUE, ENDED);
      or
         terminate;
      end select;
   end loop;
end SERVER; -- body

task USER; -- specification

task body USER is
   MY_BOX : ADDRESS;
   MY_ITEM : ITEM;
   GO_ON : BOOLEAN := TRUE;
   SERVER_READY, SATISFIED : BOOLEAN := FALSE;
begin
   ...
   MY_BOX := new MAILBOX;
   SERVER.REQUEST(MY_BOX, MY_ITEM);
   -- Follow series of collects:
   while GO_ON loop
      select
         MY_BOX.COLLECT(MY_ITEM, SERVER_READY);
         -- Use MY_ITEM, including known accuracy
      else
```

```
            null;    -- or other activities
         end select;
         if SATISFIED or SERVER_READY then
            MY_BOX.CANCEL;
            GO_ON := FALSE;
         end if;
      end loop;

      -- The user might wish to keep the mailbox for
      -- further services, but the contained task terminates,
      -- so a new allocation will be needed.

      ...
   end USER; -- body
```

iv. The server task can be interrogated about the time it still needs for its execution.

In the examples given in this section, it has usually been assumed that a calling task will cancel a request if the answer does not become available in time. This would be very wasteful of time if a server task had already been executing for some period. We would therefore like to encourage the design of algorithms for which the time needed to finish their computations is known dynamically (always assuming that a physical processor is available for the server task).

With the mailbox construct of the above example, the server task may continue to put new values into a variable SECONDS_NEEDED of the mailbox, and these values can be read by the calling task, e.g. in the following way:

```
MY_BOX.NEEDED(N_SECONDS);    -- obtains value in mailbox
if N_SECONDS > WHAT_I_ALLOW then
   MY_BOX.CANCEL;
   ALTERNATIVE_COMPUTATION;
else
   delay N_SECONDS;
   MY_BOX.COLLECT(RESULT, SERVER_READY);
end if;
```

f) Use of special architecture of machines

Since the present chapter is particularly related to the new Ada feature of "tasking", one might expect here also a discussion of the use of this feature in the design of algorithms for special machines (e.g. vector processors). Obviously, however, this application has little connection with the subject mentioned in the title of this chapter.

If the architecture of a machine allows for the speeding-up of computations in a deterministic way, e.g. by means of "pipe-lining", this will not require an alternative Ada source code (usually it will not even be possible to write Ada source code for it), and it should be left to the compiler to deliver the most efficient code for the target machine. A pleasant consequence of this is that the Ada source code will stay portable (if it was portable when written for the general method).

However, if a multi-processor system is available, then new algorithms may well emerge (and in fact some have already been designed, see Hibbard et al., 1981), with the characteristic that parts of the computation can be executed concurrently, e.g. for sorting data as in the example below. These new methods may be expressed in Ada by means of tasks and they should compete well with deterministic algorithms.

An example for vector operations is given by E.K. Blum (1982). This example does not show any advantage, because the same effect might be obtained by presenting a deterministic source code to an optimising (here: vectorising) compiler.

At the end of this section we present an example for sorting data stored in a one-dimensional array. Special care has been taken that parallel Sorters do not use the array directly, but only via a special UPDATES task, thus ensuring correct order of execution by synchronization (see section (c) above). We note that the specification of the (generic) procedure GENERIC_SORT is completely independent of the method used.

One conclusion is that if parts of a problem can be solved in a non-deterministic manner, then these subproblems should be solved by separate subprograms, thus allowing for easy replacement of one method by an alternative one for use on a multi-processor system.

```
-- Example of a generic sorting procedure.

-- Sort (to ascending order) with as many processors as possible.
-- Method: from right to left, for each pair of elements a SORTER
-- is created who walks to the right and interchanges any two
-- elements that are out of order.

generic
   type EL_TYPE is private;
   type EL_AR_TYPE is array (INTEGER range <>) of EL_TYPE;
   with function "<"(A,B : EL_TYPE) return BOOLEAN is <>;
procedure GENERIC_SORT(X : in out EL_AR_TYPE); -- specification

-- Body of GENERIC_SORT (the implementation is highly academic):

procedure GENERIC_SORT(X : in out EL_AR_TYPE) is

   LX : constant INTEGER := X'FIRST;
   UX : constant INTEGER := X'LAST;
   UX_1 : constant INTEGER := UX - 1;
   subtype INDEX is INTEGER range LX .. UX;

   task UPDATES is        -- for comments, see task bodies
      entry PUT(N : in INDEX; ITEM : in EL_TYPE);
      entry GET(N : in INDEX; ITEM : out EL_TYPE);
   end UPDATES; -- specification

   task type GUARDS is       -- cf. LRM 9.1(8)
      entry SEIZE;
      entry RELEASE;
   end GUARDS; -- specification

   task type SORTER_TYPE is
      entry WAKE_UP(N : in INDEX);
   end SORTER_TYPE; -- specification

   GUARD : array (INDEX) of GUARDS;
```

```
SORTER : array (LX .. UX_1) of SORTER_TYPE;

-- Task bodies

task body UPDATES is

-- All updates of array X are done using this task,
-- instead of by unreliable shared variable updates.
-- Hence, any reading of X gives most recent values
-- if successive PUTs and GETs are synchronized.

begin
   loop
      select
         accept PUT(N : in INDEX; ITEM : in EL_TYPE) do
            X(N) := ITEM;
         end PUT;
      or
         accept GET(N : in INDEX; ITEM : out EL_TYPE) do
            ITEM := X(N);
         end GET;
      or
         terminate;
      end select;
   end loop;
end UPDATES; -- body

task body GUARDS is       -- cf. LRM 9.7.1(13)
   BUSY : BOOLEAN := FALSE;

-- Every GUARD locks the use of the corresponding place for
-- single use by SEIZE caller, until the SORTER who locked
-- the place calls RELEASE.

begin
   loop
      select
         when not BUSY =>
            accept SEIZE;
            BUSY := TRUE;
      or
         accept RELEASE;
         BUSY := FALSE;
      or
         terminate;
      end select;
   end loop;
end GUARDS; -- body
```

```
   task body SORTER_TYPE is
      NR : INDEX;
      ITEM, ITEM_1 : EL_TYPE;
      CHANGED : BOOLEAN;
   begin
      accept WAKE_UP(N : in INDEX) do
         NR := N; -- this SORTER is informed of its own number
      end WAKE_UP;

      -- First SEIZE before waking up next SORTER, because
      -- the new one may not overtake this one.

      GUARD(NR).SEIZE;
      UPDATES.GET(NR, ITEM);
      if NR > LX then        -- wake up next-left Sorter
         SORTER(NR - 1).WAKE_UP(NR - 1);
      end if;

      -- At each step of the next iteration the SORTER
      -- reads place X(I), the value at X(I-1) is known
      -- from the previous iteration. Both elements are
      -- locked for use by this SORTER only. If necessary,
      -- two values are interchanged and the SORTER moves to
      -- the right, releasing place X(I-1)
      -- for use by the next SORTER.

      for I in NR + 1 .. UX loop
         CHANGED := FALSE;
         GUARD(I).SEIZE;
         UPDATES.GET(I, ITEM_1);
         if ITEM_1 < ITEM then
            UPDATES.PUT(I - 1, ITEM_1);
            CHANGED := TRUE;
            if I = UX then
               UPDATES.PUT(I, ITEM);
               GUARD(I).RELEASE;
            end if;
         else
            UPDATES.PUT(I - 1, ITEM);
            GUARD(I).RELEASE;
         end if;
         GUARD(I - 1).RELEASE;
         exit when not CHANGED;
      end loop;
   end SORTER_TYPE; -- body

begin -- of main procedure : start by waking up first SORTER
   if X'LENGTH > 1 then
      SORTER(UX_1).WAKE_UP(UX_1);
   end if;
end GENERIC_SORT; -- body
```

This example procedure has been successfully run with test data using the Karlsruhe Ada Compiler.

10. SUMMARY OF RECOMMENDATIONS

In this chapter we summarise our recommendations, primarily in the order in which they have appeared in the preceding chapters but with appropriate cross references. Since Chapters 1 and 2 are simply of an introductory nature, we begin with Chapter 3.

a) Precision

In Chapter 3 we discussed the fundamental problems associated with the accuracy of real types in Ada and concluded, in section (f), that, for compatibility, all library subroutines should use the same real types. We therefore recommend that a standard set of real types should be assembled into a library package of the form:

```
package REAL_TYPES is
   type REAL is digits D;
   -- etc. (see section (c) below)
end REAL_TYPES;
```

to be used by all other library packages. In practice, the number D of digits prescribed for type REAL will be implementation dependent but we would recommend (Appendix A) at least 10 digits, if possible, for scientific computation, though 6 digits may have to suffice for many real-time applications.

A library package which uses type REAL may then have a specification of the form:

```
with REAL_TYPES; use REAL_TYPES;
package LIBRARY_PACK is

   function FUN(X : REAL) return REAL;
   -- etc.

end LIBRARY_PACK;
```

in which case we recommend that it should also have a generic form:

```
generic
   type REAL is digits <>;
package GENERIC_LIBRARY_PACK is

   function FUN(X : REAL) return REAL;
   -- etc.

end GENERIC_LIBRARY_PACK;
```

to provide greater programming flexibility. We note that, for efficiency of execution, an instantiation of this generic form, for the particular type REAL used in the non-generic version, may simply result in a call of the latter (cf. section (h) of Chapter 4).

Finally, with reference to Chapter 3, we recommend that attributes should be used wherever appropriate, as described in section (c), to maintain a balance between portability and efficiency of code.

b) Basic functions

In Chapter 4 we discussed, in detail, the design of a package of basic mathematical functions in Ada. In its generic form this package has (from section (g)) the specification:

```
-------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is
   ----------------------------------------------------------------
   -- Declare constants.                                         --
   ----------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ----------------------------------------------------------------
   -- Declare the basic mathematical functions.                  --
   ----------------------------------------------------------------
   function SQRT(X : REAL) return REAL;
   function LOG(X : REAL; BASE : REAL := EXP_1) return REAL;
   function EXP(X : REAL; BASE : REAL := EXP_1) return REAL;
   function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function TAN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COT(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function ARCSIN(X : REAL) return REAL;
   function ARCCOS(X : REAL) return REAL;
   function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
   function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
   function SINH(X : REAL) return REAL;
   function COSH(X : REAL) return REAL;
   function TANH(X : REAL) return REAL;
   function COTH(X : REAL) return REAL;
   function ARCSINH(X : REAL) return REAL;
   function ARCCOSH(X : REAL) return REAL;
   function ARCTANH(X : REAL) return REAL;
   function ARCCOTH(X : REAL) return REAL;
   ----------------------------------------------------------------
   -- Declare exceptions.                                        --
   ----------------------------------------------------------------
   ARGUMENT_ERROR : exception;
   ----------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
-------------------------------------------------------------------
```

The provision of a body for the non-generic version (section (h)) of this package, for a particular machine, is discussed in Appendix C, where the ranges of the arguments of the various functions are also specified. If facilities for partial loading are available (see Appendix A and section (h) of Chapter 8), we recommend that the program components of the package body should be given as body stubs with separate subunits.

Though this package was described in Chapter 4 primarily as an example of library design, it is recommended here as a useful practical package. The double arguments, which appear in certain functions, are explained in section (e) of Chapter 4 and the exceptions which may be raised are discussed in section (f). Further details of error handling, using exceptions, are given in Chapter 7.

c) Composite data types

In Chapter 5 we discussed the provision of composite data types such as COMPLEX, VECTOR and MATRIX. The arguments in sections (a) - (d) led to the conclusion that, since type COMPLEX might be equally well defined in either Cartesian or polar form, a library should contain both versions. Therefore, for work in Cartesian coordinates, we recommend a package with the specification:

```
with REAL_TYPES; use REAL_TYPES;
package COMPLEX_OPERATORS is

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

   function RE(X : COMPLEX) return REAL;
   function IM(X : COMPLEX) return REAL;
   function "abs"(X : COMPLEX) return REAL;
   function ARG(X : COMPLEX) return REAL;
   function C_TO_COMP(R : REAL; I : REAL := 0.0)
      return COMPLEX;
   function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX;
   function "+"(X : COMPLEX) return COMPLEX;
   function "-"(X : COMPLEX) return COMPLEX;
   function "+"(X,Y : COMPLEX) return COMPLEX;
   function "-"(X,Y : COMPLEX) return COMPLEX;
   function "*"(X,Y : COMPLEX) return COMPLEX;
   function "/"(X,Y : COMPLEX) return COMPLEX;
   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX;

   pragma INLINE(RE, IM, "abs", ARG, C_TO_COMP, P_TO_COMP,
      "+", "-", "*", "/", "**");

end COMPLEX_OPERATORS;
```

while, for work in polar coordinates, we recommend a similar package with the specification:

```
package COMPLEX_POLAR_OPERATORS is

   type COMPLEX is
      record
         CMOD,CARG : REAL;
      end record;

   function RE(X : COMPLEX) return REAL;
   function IM(X : COMPLEX) return REAL;
   function "abs"(X : COMPLEX) return REAL;
   -- etc.

end COMPLEX_POLAR_OPERATORS;
```

The complete specifications and bodies of both of these packages are given in Appendix D.

Packages which use type COMPLEX, such as COMPLEX_FUNCTIONS introduced in section (c) of Chapter 5, may be made generic with respect to this type, regarded as a private type rather than a record type, provided that they are also made generic with respect to appropriate functions. Thus, such a package may have a specification:

```
with REAL_TYPES; use REAL_TYPES;
generic
   type COMPLEX is private;
   with function RE(X : COMPLEX) return REAL is <>;
   with function IM(X : COMPLEX) return REAL is <>;
   with function "abs"(X : COMPLEX) return REAL is <>;
   with function ARG(X : COMPLEX) return REAL is <>;
   with function C_TO_COMP(R : REAL; I : REAL := 0.0)
      return COMPLEX is <>;
   with function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX is <>;
package GENERIC_COMPLEX_FUNCTIONS is

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

end GENERIC_COMPLEX_FUNCTIONS;
```

and an instantiation:

```
with COMPLEX_OPERATORS; use COMPLEX_OPERATORS;
package COMPLEX_FUNCTIONS is
   new GENERIC_COMPLEX_FUNCTIONS(COMPLEX);
```

or an instantiation:

```
with COMPLEX_POLAR_OPERATORS; use COMPLEX_POLAR_OPERATORS;
package COMPLEX_FUNCTIONS is
   new GENERIC_COMPLEX_FUNCTIONS(COMPLEX);
```

as described in section (d). We recommend this construction, whereby the one generic package provides the required functions for either of the two COMPLEX types by means of an appropriate instantiation.

In section (e), concerning vectors and matrices, we defined

```
type VECTOR is array (INTEGER range <>) of REAL;
type MATRIX is array
   (INTEGER range <>, INTEGER range <>) of REAL;
```

for vectors and matrices with REAL components.

We recommend that these types should be included in the package REAL_TYPES, introduced earlier, thus:

```
package REAL_TYPES is
   type REAL is digits D;
   type VECTOR is array (INTEGER range <>) of REAL;
   type MATRIX is array
      (INTEGER range <>, INTEGER range <>) of REAL;
end REAL_TYPES;
```

Then packages for linear algebra should be made generic with respect to types VECTOR and MATRIX, as well as type REAL, thus:

```
generic
   type REAL is digits <>;
   type VECTOR is array (INTEGER range <>) of REAL;
   type MATRIX is array
      (INTEGER range <>, INTEGER range <>) of REAL;
package GENERIC_LINEAR_ALGEBRA is
   ...
end GENERIC_LINEAR_ALGEBRA;
```

in which case the instantiation:

```
with REAL_TYPES; use REAL_TYPES;
package LINEAR_ALGEBRA is
   new GENERIC_LINEAR_ALGEBRA(REAL,VECTOR,MATRIX);
```

will take types REAL, VECTOR and MATRIX from the package REAL_TYPES.

Finally, with respect to Chapter 5, we recommend that a complex vector should be represented as a vector of complex components, thus:

```
type CO_VECTOR is array (INTEGER range <>) of COMPLEX;
```

and a complex two-dimensional array similarly:

```
type CO_MATRIX is array
   (INTEGER range <>, INTEGER range <>) of COMPLEX;
```

These two types should be grouped in a library package:

```
package COMPLEX_TYPES is
   type CO_VECTOR is array (INTEGER range <>) of COMPLEX;
   type CO_MATRIX is array
      (INTEGER range <>, INTEGER range <>) of COMPLEX;
end COMPLEX_TYPES;
```

preceded by the context clause:

```
with COMPLEX_OPERATORS; use COMPLEX_OPERATORS;
```

or:

```
with COMPLEX_POLAR_OPERATORS; use COMPLEX_POLAR_OPERATORS;
```

as appropriate.

d) Information passing

In Chapter 6 we discussed interface problems which arise when two (or more) items of software are to be used in conjunction with each other. In particular, we considered the common situation in which a user has to supply a function to a library procedure.

For a simple function, of a single variable, we recommend (section (a)) that the library procedure should be made generic with respect to the function. Thus, for example, a library procedure to find a zero of a function f(x), for real x in an interval [a,b], to some accuracy e, may have a specification:

```
with REAL_TYPES; use REAL_TYPES;
generic
   with function F(X : REAL) return REAL;
procedure GENERIC_ZERO(A,B,E : in REAL; Z : out REAL);
```

Then the zero of a particular function g(x), with the specification:

```
function G(X : REAL) return REAL;
```

may be obtained by instantiating the generic procedure, thus:

```
procedure ZERO_G is new GENERIC_ZERO(G);
```

and making the call:

```
ZERO_G(A, B, E, Z);
```

with appropriate values for A, B and E.

For more complicated functions, involving certain parameters as well as the real variable x, we recommend (section (b)) the use of generics, as described in section (a), together with the block structure of the language. Thus, for example, to find the zero of a function h(x) given by a series of n terms involving prescribed coefficients, we might have:

```
declare

   -- N is imported to this block

   C,D : array (1 .. N) of REAL;

   function H(X : REAL) return REAL is
      SUM : REAL := 0.0;
   begin
      for J in 1 .. N loop
         SUM := SUM + C(J)*EXP(D(J)*X);
      end loop;
      return SUM;
   end H;

   procedure ZERO_H is new GENERIC_ZERO(H);

begin
   ...
   -- Initialise coefficients C and D
   ...
   ZERO_H(A, B, E, Z);
   ...
end;
```

Other techniques considered for the treatment of such functions, such as the passing of working-space parameters (section (c)) and the use of reverse communication (section (d)), are not recommended for general use.

In section (e) of Chapter 6, we recommend that parameters with default values should be placed at the end of the formal part of a subprogram specification, contrary to the common practice of putting in parameters at the beginning.

e) Error handling

In Chapter 7 we discussed the use of exceptions for the handling of error situations. The predefined exceptions - CONSTRAINT_ERROR, NUMERIC_ERROR and STORAGE_ERROR - were considered in section (a) and existing error-handling practices were discussed briefly in section (b). Conclusions were drawn in section (c).

With reference to section (a), we recommend (when an error occurs) the raising of an exception whose name clearly indicates the pre-condition which has been violated. This will not usually be a predefined exception, but, for example, ARGUMENT_ERROR rather than CONSTRAINT_ERROR for a call of SQRT with a negative argument. Thus the raising of predefined exceptions should usually be "translated" into the raising of exceptions belonging to user-oriented packages.

We recommend (section (c)) that exceptions raised in library packages which are used by other library packages should be handled in these other packages to initiate alternative approaches or to raise further, more meaningful exceptions.

We also recommend that initialisations in declarations should be kept simple, to avoid raising exceptions which cannot be handled locally but have to be propagated.

We realise that the exception mechanism does not provide the most user-friendly system of error handling and that control by the end-user has not yet been fully explored. However, this subject is currently being studied by the Ada-Europe Numerics Working Group and we recommend that further attention be given to this subject in the future.

f) Working-space organisation

In Chapter 8 we discussed several general aspects of working-space organisation, considering both explicitly-declared storage (in sections (a) - (d)) and storage which is used implicitly (in sections (e) - (i)).

From section (a), with regard to choice of data types, we recommend the use of different data types for matrices which require different storage methods. We also recommend that documentation of library subprograms should contain sufficient information for the programmer to be able to estimate the amounts of working-space to be claimed for their execution.

From section (b) we recommend that aliasing of subprogram parameters should be avoided and that working-space parameters should not be used.

In section (c) we advise against the use of representation and address clauses.

In section (d), regarding the use of attributes and pragmas, we recommend judicious use of the pragma PACK to instruct an installation to minimise gaps in storage areas for objects of composite types.

Our conclusion from section (e) is that range constraints on array objects should always be separated from their type declarations.

In section (f), with regard to dynamic storage allocation, we recommend that storage should be freed when values of the relevant access type are no longer accessible because the unit containing the declaration of this access type is left. If dynamic storage must be given to a user program, the access type should be a limited private type, to prevent the user from copying accesses. In real-time situations the user should do his own storage management.

Finally, from section (h), we recommend that library packages should not be too large and that body stubs and subunits should be used to permit separate compilation.

g) Real-time environment

In Chapter 9 we discussed the particular problems associated with the design of scientific libraries for use in real-time processing, noting, in particular, that such libraries are likely to have different subprogram bodies from those used for batch processing. Correspondingly, our first recommendation (section (a)) is that services requested by tasks should always be granted by tasks.

From section (c) we recommend that tasks should not access shared variables for simultaneous reading and updating. The best solution to this problem is to perform all accesses through a central task. We do not object to direct access to shared variables for reading purposes only (i.e. shared data).

From section (d) we recommend that all exceptions must be handled when using tasks.

In section (e) we discuss examples of the use of tasks and particularly recommend the use of a "mailbox" construct, whereby results of a server task are sent to an agent task which may be inspected by the calling task whenever necessary.

In section (f) we recommend further investigation of the tasking facilities of Ada as a tool for the development of new algorithms for computations on distributed processors.

APPENDIX A - TARGET IMPLEMENTATION AND LANGUAGE DEFICIENCIES

Here we summarise features of a target implementation under three headings according to the importance which we attach to them. We also list, in a fourth section, features of the Ada language which we consider to be deficiencies as far as scientific computing is concerned; we hope that these deficiencies will be removed from the the language at the first opportunity.

a) Necessary requirements

These requirements are considered essential, and are therefore assumed to hold, for any target implementation to which the preceding guidelines are applicable:

- The exception NUMERIC_ERROR must be raised in overflow situations (cf. LRM 4.5.7(7)). See section (a) of Chapter 7.
- There must be no copying of unconstrained array parameters of mode **out** or **in out** (apart from entry calls). See sections (b) and (i) of Chapter 8.
- Information must be provided as to whether two-dimensional arrays are stored by row or by column. See section (g) of Chapter 8. A choice of storage method would be highly desirable.
- Facilities for pre-compilation must be available. See section (d) of Chapter 3.
- Facilities for partial loading must be available. See section (h) of Chapter 8.
- Updating of a shared scalar-type variable must be an indivisible operation. See section (c) of Chapter 9.

b) Highly desirable features

The following requirements, though not mandatory, are recommended for any target implementation:

- At least 10 digits of precision should be available for floating-point computation, i.e. SYSTEM.MAX_DIGITS should not be less than 10. See section (a) of Chapter 2 and cf. section (a) of Chapter 4. We realise however that for many real-time applications, 6 digits may have to suffice.
- At least two floating-point types should be provided. See section (b) of Chapter 3.
- Instantiations of generics should be performed as efficiently as possible. See section (h) of Chapter 4.
- A garbage collector should be available. See section (f) of Chapter 8.
- There should be no copying of array-type function results. See section (i) of Chapter 8.

c) Useful features

The following features are ideals which might not be easily implemented but which would be very welcome:

- The number of digits in a floating-point type might be unrestricted, i.e. SYSTEM.MAX_DIGITS might be (essentially) unbounded.

- The attribute BASE'DIGITS might give all values from 5 to 100, or thereabouts, for the investigation of algorithms using different (software) floating-point precisions.

d) Language deficiencies

The following features of the Ada language severely restrict its use for large-scale scientific computation:

- Subprograms are not permitted as subprogram parameters. See section (a) of Chapter 6.

- Record types are not permitted as generic parameters. See section (d) of Chapter 5.

- Type declarations in generic packages cannot depend on attributes of generic actual parameters, since these are not static. See section (d) of Chapter 4.

- Designators of subunits must be identifiers (LRM 10.1(3)). See section (h) of Chapter 8.

- The Ada model has limitations for floating-point arithmetic. See section (c) of Chapter 3 and Appendix F.

- The definition of MACHINE_OVERFLOWS is inadequate. See section (a) of Chapter 7.

- Slicing is restricted to one-dimensional arrays (LRM 4.1.2); i.e. slicing of multi-dimensional arrays is not permitted. See Appendix E, where the latter could be very useful.

- The simple names of all subunits that have the same ancestor library unit must be distinct identifiers (LRM 10.2(6)). This is unreasonable and quite unnecessary. The term "ancestor library unit" could be replaced by "(direct) parent unit", though actually "The full names of all compilation units must be distinct" would be quite sufficient.
  At present, the language does not permit the separate compilation of the procedures
  ```
  LIB_UNIT.PAR_UNIT_1.SUB_UNIT
  ```
  and
  ```
  LIB_UNIT.PAR_UNIT_2.SUB_UNIT
  ```
  although their names are distinct. Such a restriction on the names of compilation units conflicts with a clear hierarchical library development.

APPENDIX B - SUMMARY OF BASIC PACKAGES FOR SCIENTIFIC COMPUTATION

Here we summarise the contents of the basic packages which we have recommended in this report. Since library units must have distinct identifiers (LRM 10.1(3)), the names of these packages should not be duplicated by users. The names of packages which have both generic and non-generic versions begin with (GENERIC_), the brackets indicating that the word they enclose is optional.

```
REAL_TYPES
   REAL
   VECTOR
   MATRIX
(GENERIC_)MATH_FUNCTIONS
   PI
   EXP_1
   SQRT
   LOG
   EXP
   SIN
   COS
   TAN
   COT
   ARCSIN
   ARCCOS
   ARCTAN
   ARCCOT
   SINH
   COSH
   TANH
   COTH
   ARCSINH
   ARCCOSH
   ARCTANH
   ARCCOTH
   ARGUMENT_ERROR
COMPLEX_OPERATORS and COMPLEX_POLAR_OPERATORS
   COMPLEX
   RE
   IM
   "abs"
   ARG
   C_TO_COMP
   P_TO_COMP
   "+"   -- both unary and binary versions
   "-"   -- both unary and binary versions
   "*"
   "/"
   "**"
(GENERIC_)COMPLEX_FUNCTIONS
   SQRT
   LOG
   EXP
   SIN
   COS
COMPLEX_TYPES
   CO_VECTOR
   CO_MATRIX
```

APPENDIX C - THE BASIC AND PRIMITIVE FUNCTIONS

Here we illustrate the implementation of the basic mathematical functions package, as specified in Chapter 4, by providing selected parts of the package body. To implement the basic functions with acceptable efficiency in Ada we need to use some primitive functions, so we introduce these first.

a) Primitive functions for Ada

Several proposals have been made for a set of primitive functions for handling the basic representation of floating-point numbers; see, for example, (Ford, 1978), (Brown and Feldman, 1980), (Reid, 1979), and (Cody and Waite, 1980). The last of these references considers the implementation of the basic mathematical functions and in consequence, defines primitive functions in a preliminary chapter. The other three works are broadly similar except that Ford considers also non-numerical aspects of the environment of numerical computation (in FORTRAN). There is one additional proposal which has a formal standards status; that is the set of environmental functions which form an optional part of the IEC floating point standard (IEC, 1982).

These references show the need for a set of primitive functions in any language used for numerical computation. In Ada, some of the requirements are satisfied by attributes such as DIGITS, MANTISSA and SAFE_LARGE. However, there is also a need for functions to decompose a floating-point number into its exponent and mantissa (significand) and to construct such a number from these parts. The early (informal) standardisation of these primitive functions is recommended so that mathematical libraries can have a common base for further development. The proposal made here is based upon the work of Brown and Feldman since this in turn depends upon the Brown model which forms the basis of the definition of floating-point types in Ada.

Following the strategy used for precision (section (f) of Chapter 3), the package is provided in the generic form with the real type as a generic type parameter. The instantiation of this package for the hardware types will then provide the low-level equivalents of the FORTRAN functions proposed by Brown and Feldman. The package then becomes:

```
generic
   type REAL is digits <>;
package GENERIC_PRIMITIVE_FUNCTIONS is

   function EXPONENT(X : REAL) return INTEGER;
      -- gives the exponent of X

   function FRACTION(X : REAL) return REAL;
      -- gives the mantissa as a fraction, such that
      -- 0.5 <= abs FRACTION < 1.0 or FRACTION = 0.0

   function SYNTHESIZE(X : REAL; E : INTEGER) return REAL;
      -- gives FRACTION(X) * 2.0 ** E

   function SCALE(X : REAL; E : INTEGER) return REAL;
      -- gives X * 2.0 ** E
      function ABS_SPACING(X : REAL) return REAL;
         -- gives 2.0**(EXPONENT(X)-REAL'MANTISSA) for
         -- abs X >= REAL'SMALL/REAL'EPSILON
         -- gives REAL'SMALL for
         -- abs X < REAL'SMALL/REAL'EPSILON

      function REC_REL_SPACING(X : REAL) return REAL;
         -- gives SYNTHESIZE(abs X, REAL'MANTISSA)

   end GENERIC_PRIMITIVE_FUNCTIONS; -- specification
```

In practice, for a specific hardware type, such subroutines would be implemented by a few in-line machine instructions. However, such subroutines can be implemented in Ada using unchecked programming.

The above specification is informal but sufficient for most uses. To be more accurate, so that one can prove the correctness of an algorithm using these functions, the results must be defined in terms of the Brown model. The following definitions would be adequate:

EXPONENT: If X is a model number not equal to zero, then the result is the exponent in the Brown model. (The physical exponent could be different due to a bias in the representation.) If X=0.0, then EXPONENT(X)=0. If X is not a model number, then the value is that of the exponent of the next model number above or below X. The specification of this function makes the implicit assumption that the exponent range is within the range of type INTEGER. If INTEGER is 16-bits and the exponent range is that of model numbers, then this gives a restriction that 'DIGITS < 8190.

FRACTION: If X is a model number, then the result must be the fraction with the same sign as the parameter. If X = 0.0, then FRACTION(X) = 0.0. If X is not a model number, then FRACTION(X) returns a result in the appropriate model interval. This means that FRACTION is an operation which supports the model arithmetic, as defined in LRM 4.5.7, in the same way as the predefined operations.

SYNTHESIZE: This function merely gives a fast and more convenient way of calculating FRACTION(X)*2.0**E. Overflow and underflow should be handled in the same way as if the computation had been performed directly in Ada. Hence this function would ordinarily give 0.0 on underflow and NUMERIC_ERROR on overflow. The result is a model number if the exponent is in range. On an IEC system, the result could be a denormalised number (for underflow).

SCALE: It is proposed that this function should not rely upon the model at all but merely provide a quick form of multiplication (or division) by a power of two.

ABS_SPACING: This function gives the absolute spacing in the neighbourhood of X. However, if the value of X is very small, then the result cannot be accommodated within the range of model numbers. (Perhaps we should use safe numbers.)

REC_REL_SPACING: This function gives the reciprocal of the relative spacing around X. The reciprocal is chosen because it is simpler and more often required than the relative spacing itself. The exact semantics is in terms of SYNTHESIZE. Note the uncertainty in the result caused by the application of "abs" (for overlength arguments).

The body of the generic package can be provided, in a version which, though inefficient, serves to clarify the definition, as follows:

```
package body GENERIC_PRIMITIVE_FUNCTIONS is

   function EXPONENT(X : REAL) return INTEGER is
      -- gives the exponent of X
      E: INTEGER := 0;
      Y: REAL := abs X;
   begin
      if Y = 0.0 then
         return 0;
      end if;
      while Y >= 1.0 loop
         E := E + 1;
         Y := Y/2.0;
      end loop;
      while Y < 0.5 loop
         E := E - 1;
         Y := Y*2.0;
      end loop;
      return E;
   end EXPONENT;

   function FRACTION(X : REAL) return REAL is
      -- gives the mantissa as a fraction, such that
      -- 0.5 <= abs FRACTION < 1.0 or FRACTION = 0.0
      Y: REAL := abs X;
   begin
      if Y = 0.0 then
         return 0.0;
      end if;
      while Y >= 1.0 loop
         Y := Y/2.0;
      end loop;
      while Y < 0.5 loop
         Y := Y*2.0;
      end loop;
      if X > 0.0 then
         return Y;
      else
         return - Y;
      end if;
   end FRACTION;

   function SYNTHESIZE(X : REAL; E : INTEGER) return REAL is
      -- gives FRACTION(X)*2.0**E
   begin
      return FRACTION(X)*2.0**E;
   end SYNTHESIZE;
```

```
      function SCALE(X : REAL; E : INTEGER) return REAL is
         -- gives X*2.0**E
      begin
         return X*2.0**E;
      end SCALE;

      function ABS_SPACING(X : REAL) return REAL is
         -- gives 2.0**(EXPONENT(X)-REAL'MANTISSA) for
         -- abs X >= REAL'SMALL/REAL'EPSILON
         -- gives REAL'SMALL for
         -- abs X < REAL'SMALL/REAL'EPSILON
      begin
         if abs X >= REAL'SMALL/REAL'EPSILON then
            return 2.0**(EXPONENT(X)-REAL'MANTISSA);
         else
            return REAL'SMALL;
         end if;
      end ABS_SPACING;

      function REC_REL_SPACING(X : REAL) return REAL is
         -- gives SYNTHESIZE(abs X, REAL'MANTISSA)
      begin
         return SYNTHESIZE(abs X, REAL'MANTISSA);
      end REC_REL_SPACING;

   end GENERIC_PRIMITIVE_FUNCTIONS; -- body
```

In order to illustrate a more practical method of implementing these functions in Ada, we provide below a version using the layout of 64-bit reals adopted by the IEC standard. This implementation avoids the loops in the bodies of EXPONENT and FRACTION which are inherent in the generic version.

To decompose a floating-point value, we must use the actual representation of floating-point numbers on a specific machine. We also need to be able to view a floating-point value in two ways - either as a conventional Ada floating-point value, or as a record containing the sign, exponent and mantissa. In Ada the change from one view to the other is obtained by means of a conversion.

Ada provides a generic function to convert the value of one type into that of another. The types need not have any relationship and it is assumed that the conversion is simply one of changing the type of the same bit pattern. An implementation is likely to place some restrictions upon the types that can be converted, for instance, that they should have the same length. The conversion is by a function so that a new value is obtained which must therefore be assigned to an object of the new type (or otherwise processed as such). This conversion is quite different from the "equivalence" facility in FORTRAN, whereby one storage area can be regarded as being of more than one type.

The generic unit has the specification:

```
   generic
      type SOURCE is limited private;
      type TARGET is limited private;
   function UNCHECKED_CONVERSION(S : SOURCE) return TARGET;
```

Both types are limited private so that they can apply to any type (although an implementation may restrict this freedom).

We can decompose a number entirely within Ada by means of the predefined function UNCHECKED_CONVERSION. However, we must first design a machine-independent specification of the required facility so that we can program the rest of the software in a portable manner.

The procedure is specified as follows:

```
procedure SPLIT_FLOAT_VALUE(EXPO  : out INTEGER;
                            FRACT : out FLOAT;
                            X     : in FLOAT);
   -- sets EXPO and FRACT such that
   --     X = FRACT*2.0**EXPO
```

Then, the separate body of this procedure, which needs to access the UNCHECKED_CONVERSION function, might take the form:

```
with UNCHECKED_CONVERSION;
procedure SPLIT_FLOAT_VALUE(EXPO  : out INTEGER;
                            FRACT : out FLOAT;
                            X     : in FLOAT) is
   type MANTISSA_TYPE is array(1 .. 52) of BOOLEAN;
   type FLOAT_BITS is
      record
         SIGN : BOOLEAN;
         EXPO : INTEGER range 0 .. 2047;
         MANTISSA: MANTISSA_TYPE;
      end record;
   for FLOAT_BITS use
      record
         SIGN at 0 range 0 .. 0;
         EXPO at 0 range 1 .. 11;
         MANTISSA at 0 range 12 .. 63;
      end record;
   FB: FLOAT_BITS;
   function TO_FLOAT_BITS is
      new UNCHECKED_CONVERSION(FLOAT, FLOAT_BITS);
   function TO_FLOAT is
      new UNCHECKED_CONVERSION(FLOAT_BITS, FLOAT);
begin
   if X = 0.0 then
      EXPO := 0;
      FRACT := 0.0;
      return;
   end if;
   FB := TO_FLOAT_BITS(X);
   EXPO := FB.EXPO - 1023;
   FB.EXPO := 1022;       -- to allow for bias
   FRACT := TO_FLOAT(FB);
end SPLIT_FLOAT_VALUE;
```

The technique used here is based upon setting up the type FLOAT_BITS to mirror the actual representation of floating-point values. Specific values for the sizes of the exponent and mantissa fields have been inserted; these sizes are those of the double length of the IEC standard. The conversion is then performed so that the exponent can be extracted as a component of FB. The reverse conversion gives the fraction left. The Ada rules, for the complete expression of the types involved, give an easily understandable program text but it is rather verbose. In contrast, a good compiler could reduce this procedure to just two machine instructions.

One can see that there are substantial pitfalls in implementing this procedure on a new computer system. It is easy to make a mistake with the layout of the parts of the floating-point number so that incorrect results are obtained (but it compiles, since the compiler can do no checking). Hence, we provide a safe (but inefficient) version:

```
procedure SPLIT_FLOAT_VALUE(EXPO  : out INTEGER;
                            FRACT : out FLOAT;
                            X     : in FLOAT) is
   E: INTEGER := 0; -- Exponent so far
   F: FLOAT := abs X; -- Fraction so far
begin
   if X = 0.0 then
      EXPO := 0;
      FRACT := 0.0;
      return;
   end if;
   while F >= 1.0 loop
      F := F/2.0;
      E := E + 1;
   end loop;
   while F < 0.5 loop
      F := F*2.0;
      E := E - 1;
   end loop;
   if X < 0.0 then
      F := - F;
   end if;
   EXPO := E;
   FRACT := F;
end SPLIT_FLOAT_VALUE;
```

We can now implement the six primitive functions in terms of the SPLIT_FLOAT_VALUE when the type REAL is the type FLOAT, i.e. a 64-bit real. (Other floating-point types would require other versions of SPLIT_FLOAT_VALUE.) In this case, the body of the non-generic version becomes:

```
with REAL_TYPES, SPLIT_FLOAT_VALUE;
use REAL_TYPES;
package body PRIMITIVE_FUNCTIONS is

   function EXPONENT(X : REAL) return INTEGER is
      -- gives the exponent of X
      EXPO: INTEGER;
      FRACT: FLOAT;
      Y: FLOAT := FLOAT(X);
   begin
      SPLIT_FLOAT_VALUE(EXPO, FRACT, Y);
      return EXPO;
   end EXPONENT;
```

```
    function FRACTION(X : REAL) return REAL is
       -- gives the mantissa as a fraction, such that
       -- 0.5 <= abs FRACTION < 1.0 or FRACTION = 0.0
       EXPO: INTEGER;
       FRACT: FLOAT;
       Y: FLOAT := FLOAT(X);
    begin
       SPLIT_FLOAT_VALUE(EXPO, FRACT, Y);
       return REAL(FRACT);
    end FRACTION;

    function SYNTHESIZE(X : REAL; E : INTEGER) return REAL is
       -- gives FRACTION(X)*2.0**E
    begin
       return FRACTION(X)*2.0**E;
    end SYNTHESIZE;

    function SCALE(X : REAL; E : INTEGER) return REAL is
       -- gives X*2.0**E
    begin
       return X*2.0**E;
    end SCALE;

    function ABS_SPACING(X : REAL) return REAL is
       -- gives 2.0**(EXPONENT(X)-REAL'MANTISSA) for
       -- abs X >= REAL'SMALL/REAL'EPSILON
       -- gives REAL'SMALL for
       -- abs X < REAL'SMALL/REAL'EPSILON
    begin
       if abs X >= REAL'SMALL/REAL'EPSILON then
          return 2.0**(EXPONENT(X)-REAL'MANTISSA);
       else
          return REAL'SMALL;
       end if;
    end ABS_SPACING;

    function REC_REL_SPACING(X : REAL) return REAL is
       -- gives SYNTHESIZE(abs X, REAL'MANTISSA)
    begin
       return SYNTHESIZE(abs X, REAL'MANTISSA);
    end REC_REL_SPACING;

 end PRIMITIVE_FUNCTIONS;
```

b) The basic functions

We now consider the implementation of the basic functions in Ada as proposed in Chapter 4. The full specification of the package is:

```
---------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTIONS is
   ------------------------------------------------------------------
   -- Declare constants.                                          --
   ------------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ------------------------------------------------------------------
   -- Declare the basic mathematical functions.                    --
   ------------------------------------------------------------------
   function SQRT(X : REAL) return REAL;
   function LOG(X : REAL; BASE : REAL := EXP_1) return REAL;
   function EXP(X : REAL; BASE : REAL := EXP_1) return REAL;
   function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function TAN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function COT(X : REAL; CYCLE : REAL := 2.0*PI) return REAL;
   function ARCSIN(X : REAL) return REAL;
   function ARCCOS(X : REAL) return REAL;
   function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
   function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
   function SINH(X : REAL) return REAL;
   function COSH(X : REAL) return REAL;
   function TANH(X : REAL) return REAL;
   function COTH(X : REAL) return REAL;
   function ARCSINH(X : REAL) return REAL;
   function ARCCOSH(X : REAL) return REAL;
   function ARCTANH(X : REAL) return REAL;
   function ARCCOTH(X : REAL) return REAL;
   ------------------------------------------------------------------
   -- Declare exceptions.                                          --
   ------------------------------------------------------------------
   ARGUMENT_ERROR : exception;
   ------------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
---------------------------------------------------------------------
```

The ranges of the arguments of the functions in this package are shown in the following table:

Function	Argument and range
SQRT	X >= 0.0
LOG	X > 0.0, BASE > 0.0 and /= 1.0
EXP	X unrestricted, BASE > 0.0
SIN	X unrestricted, CYCLE /= 0.0
COS	X unrestricted, CYCLE /= 0.0
TAN	X unrestricted, CYCLE /= 0.0
COT	X unrestricted, CYCLE /= 0.0
ARCSIN	**abs** X <= 1.0
ARCCOS	**abs** X <= 1.0
ARCTAN	**not** (X = 0.0 **and** Y = 0.0)
ARCCOT	**not** (X = 0.0 **and** Y = 0.0)
SINH	X unrestricted
COSH	X unrestricted
TANH	X unrestricted
COTH	X unrestricted
ARCSINH	X unrestricted
ARCCOSH	X >= 1.0
ARCTANH	**abs** X < 1.0
ARCCOTH	**abs** X > 1.0

Since the accuracy of the package is limited by the 35 digits given for the constants, the text of the package should be applicable for all accuracies up to 35 digits. The sample text given below falls a little short of this but illustrates the use of the primitive functions in the coding of the basic functions SQRT, SIN and COS.

Polynomial approximations are used. Such polynomials are typically truncated power series which rely upon the decreasing contributions from the higher-order terms. Given:

```
   Y := A + B*X + C*X**2 + D*X**3;
```

the most effective evaluation method is nested multiplication, i.e.

```
   Y := ((D*X + C)*X + B)*X + A;
```

The routines are coded for the type REAL. The following assumptions are made about this type:

a) its range should contain the subrange 0.25 .. 1.5 (for SQRT),
b) its range should be symmetric about the origin (for SIN and COS),
c) its range should contain 2.0*PI (also for SIN and COS), and
d) its EXPONENT should be in the range of type INTEGER.

No further assumptions are made and the body of the package GENERIC_MATH_FUNCTIONS should raise an exception if the specified type REAL does not satisfy these assumptions.

The choices in the algorithms, necessary to accommodate the widely ranging precisions, are coded as branches. A good optimising compiler might very well omit the dead branches.

The argument reduction is a difficult aspect of sine and cosine. In the routine REDUCE_RANGE the constants C1 and C2 (whose sum is CYCLE) are used for this purpose in such a way that the range reduction is exact. If CYCLE is equal to the default value, precalculated values are used for C1 and C2 (from the package MACH_DEP_CONSTANTS). We note that Cody and Waite (1980) describe a better method of range reduction when CYCLE is 2.0*PI; we have not incorporated this into our version but it could easily be done by changing the calculation of C1PI and C2PI in the body of MACH_DEP_CONSTANTS.

The body of SQRT given here is adequate for any type REAL for which REAL'DIGITS is less than 33. The bodies of SIN and COS and their auxiliary routines are adequate for REAL'DIGITS less than 26. The only machine dependencies are coded in the package MACH_DEP_CONSTANTS.

```
-----------------------------------------------------------------------
with GENERIC_PRIMITIVE_FUNCTIONS;
package body GENERIC_MATH_FUNCTIONS is
   --------------------------------------------------------------------
   package PRIMITIVE_FUNCTIONS is
      new GENERIC_PRIMITIVE_FUNCTIONS(REAL);
   --------------------------------------------------------------------
   function SQRT(X : REAL) return REAL is
   begin
      if X = 0.0 then
         return 0.0;
      elsif X < 0.0 then
         raise ARGUMENT_ERROR;
      else
      declare
         use PRIMITIVE_FUNCTIONS;
         N : INTEGER := EXPONENT(X);
         F : REAL := SYNTHESIZE(X, 0);
         Y : REAL := 0.41732 + 0.59018*F;
      begin
         if N mod 2 = 1 then
            Y := Y*0.70710;
            F := F*0.5;
            N := N + 1;
         end if;
         if REAL'DIGITS > 2 then -- MANTISSA > 14
            Y := 0.5*(Y + F/Y);
         end if;
         if REAL'DIGITS > 7 then -- MANTISSA > 30
            Y := 0.5*(Y + F/Y);
         end if;
         if REAL'DIGITS > 16 then -- MANTISSA > 62
            Y := 0.5*(Y + F/Y);
         end if;
         Y := Y - 0.5*(Y - F/Y);
         return SCALE(Y, N/2);
      end;
   end if;
end SQRT;
-------------------------------------------------------------------
package MACH_DEP_CONSTANTS is
   -- contains machine dependent constants for use in the
   -- trigonometric functions.
   -- NBITS, FACTOR and FACTOR1 are defined as follows:
   -- FACTOR = 2.0**NBITS, FACTOR1 = FACTOR*2.0.
   -- NBITS is limited by the fact that FACTOR - 1.0 should be
   -- representable as an INTEGER, and that (FACTOR - 1.0)*
   -- (FACTOR1 - 1.0) should be representable as REAL.
   NBITS : INTEGER;
   FACTOR, FACTOR1 : REAL;
   -- their inverses
   INV_FACTOR, INV_FACTOR1 : REAL;
   -- C1PI and C2PI contain together PI/4.0, such that
   -- 1. C1PI + C2PI = PI/4.0;
   -- 2. C1PI*FACTOR1 is INTEGER;
   -- 3. abs C2PI <= 0.5/FACTOR.
   C1PI, C2PI : REAL;
   -- UNDERFLOW_THRESHOLD to avoid underflow in calculations.
   UNDERFLOW_THRESHOLD : REAL := 2.0**(- REAL'MACHINE_MANTISSA/2);
end MACH_DEP_CONSTANTS; -- specification
-------------------------------------------------------------------
package body MACH_DEP_CONSTANTS is
begin -- use local blocks to declare temps
   declare
      N1 : constant := INTEGER'SIZE - 1; -- allow for sign bit
      N2 : constant INTEGER := (REAL'MACHINE_MANTISSA - 1)/2;
   begin
      if N1 < N2 then
         NBITS := N1;
         FACTOR := 2.0**N1;
      else
         NBITS := N2;
         FACTOR := 2.0**N2;
      end if;
      FACTOR1 := FACTOR*2.0;
      INV_FACTOR := 1.0/FACTOR;
      INV_FACTOR1 := 1.0/FACTOR1;
   end;
   declare
      TWO_PI : REAL := 2.0*PI;
      PI_OVER_4 : REAL := TWO_PI*0.125;
   begin
      C1PI := REAL(INTEGER(PI_OVER_4*FACTOR1))*INV_FACTOR1;
      C2PI := PI_OVER_4 - C1PI;
   end;
end MACH_DEP_CONSTANTS; -- body
------------------------------------------------------------------
procedure REDUCE_RANGE(X,CYCLE : in REAL; ARG : out REAL;
   N : out INTEGER) is
--
-- This routine reduces X modulo CYCLE, and scales the result.
-- More specifically, at the end:
-- -PI/4.0 <= ARG <= PI/4.0; 0 <= N < 4;
-- (X - ARG*CYCLE/(2.0*PI) - N*CYCLE/4) is a multiple of CYCLE.
--
   use MACH_DEP_CONSTANTS, PRIMITIVE_FUNCTIONS;
   -- factor 2 because of reduction mod CYCLE/4.0
   NX : INTEGER := EXPONENT(X) - EXPONENT(CYCLE) + 2;
   FX : REAL := SYNTHESIZE(X, 0);
   FC : REAL := SYNTHESIZE(CYCLE, 0);
   TWO_PI : REAL := 2.0*PI;
   SGN : INTEGER := 1;
   FF, C1, C2 : REAL;
   NN, N1 : INTEGER;
begin
   if FX < 0.0 then
      SGN := - SGN;
      FX := - FX;
   end if;
   if FC < 0.0 then
      SGN := - SGN;
      FC := - FC;
   end if;
   if TWO_PI*0.125 = FC then
      C1 := C1PI; -- imported from MACH_DEP_CONSTANTS
      C2 := C2PI;
   elsif FC = 0.5 then
      C1 := FC;
      C2 := 0.0;
   else
      C1 := REAL(INTEGER(FC*FACTOR1))*INV_FACTOR1;
      C2 := FC - C1;
   end if;
   -- perform exact range reduction
   if (NX > 0) or else ((NX = 0) and then (FX >= FC)) then
      begin
         NN := NX mod NBITS;
         FF := 2.0**NN;
         NX := NX - NN;
         loop -- retain only low order bits of multiple of CYCLE
            N1 := INTEGER(FX*FF/FC - 0.5); -- truncate
            FX := (FX*FF - N1*C1) - N1*C2;
            exit when NX = 0;
            FF := FACTOR;
            NX := NX - NBITS;
         end loop;
         end;
      else
         N1 := 0;
      end if;
      FX := SCALE(FX, NX + 1); -- adjust for CYCLE/8
      if FX >= FC then
         FX := ((FX - C1) - C2 - C2) - C1; -- be careful here
         N1 := N1 + 1;
      end if;
      -- now abs FX <= FC*0.5
      if SGN < 0 then
         FX := - FX;
         N1 := - N1;
      end if;
      N := N1 mod 4;
      ARG := SCALE((TWO_PI/FC)*FX, - 3);
   end REDUCE_RANGE;
   -----------------------------------------------------------------
   function SIN_COS(X,CYCLE : REAL; COS_WANTED : BOOLEAN)
         return REAL is
      use MACH_DEP_CONSTANTS;
      ARG, ARG2, FX : REAL;
      N : INTEGER;
   begin
      if CYCLE = 0.0 then
         raise ARGUMENT_ERROR;
      end if;
      REDUCE_RANGE(X, CYCLE, ARG, N);
      if abs ARG <= UNDERFLOW_THRESHOLD then
         if COS_WANTED then
            return 1.0;
         else
            return ARG;
         end if;
      end if;
      if COS_WANTED then
         -- shift PI/2
         N := N + 1;
         if N = 4 then
            N := 0;
         end if;
      end if;
      -- apply the following table:
      --    N     Use        Sign
      --    0     sine       as calculated
      --    1     cosine     as calculated
      --    2     sine       inverted
      --    3     cosine     inverted
      ARG2 := ARG*ARG;
      if N mod 2 = 0 then
         -- sine approximations
         case REAL'DIGITS is
            when 1 .. 1 => -- MANTISSA <= 10
               declare
                  CO : constant := -0.1616;
               begin
                  FX := CO*ARG2;
               end;
            when 2 .. 3 => -- MANTISSA <= 18
               declare
                  CO : constant := -0.16662_4;
      C1 : constant := +0.00815_1;
   begin
      FX := (C1*ARG2 + C0)*ARG2;
   end;
when 4 .. 6 => -- MANTISSA <= 27
   declare
      C0 : constant := -0.16666_6507;
      C1 : constant := +0.00833_2036;
      C2 : constant := -0.00019_5040;
   begin
      FX := ((C2*ARG2 + C1)*ARG2 + C0)*ARG2;
   end;
when 7 .. 9 => -- MANTISSA <= 37
   declare
      C0 : constant := -0.16666_66663_16;
      C1 : constant := +0.00833_33287_84;
      C2 : constant := -0.00019_83920_27;
      C3 : constant := +0.00000_27173_49;
   begin
      FX := (((C3*ARG2 + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
   end;
when 10 .. 12 => -- MANTISSA <= 47
   declare
      C0 : constant := -0.16666_66666_66167;
      C1 : constant := +0.00833_33333_23881;
      C2 : constant := -0.00019_84126_32999;
      C3 : constant := +0.00000_27555_27217;
      C4 : constant := -0.00000_00247_56577;
   begin
      FX := ((((C4*ARG2 + C3)*ARG2 + C2)*ARG2 + C1)*ARG2
         + C0)*ARG2;
   end;
when 13 .. 15 => -- MANTISSA <= 57
   declare
      C0 : constant := -0.16666_66666_66666_167;
      C1 : constant := +0.00833_33333_33320_366;
      C2 : constant := -0.00019_84126_98286_530;
      C3 : constant := +0.00000_27557_31337_725;
      C4 : constant := -0.00000_00250_50717_097;
      C5 : constant := +0.00000_00001_58947_433;
   begin
      FX := (((((C5*ARG2 + C4)*ARG2 + C3)*ARG2
         + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
   end;
when 16 .. 18 => -- MANTISSA <= 68
   declare
      C0 : constant := -0.16666_66666_66666_66629_6;
      C1 : constant := +0.00833_33333_33333_32072_1;
      C2 : constant := -0.00019_84126_98412_53481_5;
      C3 : constant := +0.00000_27557_31921_35637_2;
      C4 : constant := -0.00000_00250_52104_77945_0;
      C5 : constant := +0.00000_00001_60583_52470_8;
      C6 : constant := -0.00000_00000_00757_80921_0;
   begin
      FX := ((((((C6*ARG2 + C5)*ARG2 + C4)*ARG2
         + C3)*ARG2 + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
   end;
when 19..21 => -- MANTISSA <= 79
   declare
      C0 : constant := -0.16666_66666_66666_66666_64551;
      C1 : constant := +0.00833_33333_33333_33332_41882;
```

```
            C2 : constant := -0.00019_84126_98412_69826_04680;
            C3 : constant := +0.00000_27557_31922_39731_95420;
            C4 : constant := -0.00000_00250_52108_37949_51033;
            C5 : constant := +0.00000_00001_60590_42200_25880;
            C6 : constant := -0.00000_00000_00764_69011_68013;
            C7 : constant := +0.00000_00000_00002_78872_64214;
         begin
            FX := (((((((C7*ARG2 + C6)*ARG2 + C5)*ARG2
               + C4)*ARG2 + C3)*ARG2 + C2)*ARG2 + C1)*ARG2
               + C0)*ARG2;
         end;
      when 22 .. 25 => -- MANTISSA <= 90
         declare
            C0 : constant :=
               -0.16666_66666_66666_66666_66665_707;
            C1 : constant :=
               +0.00833_33333_33333_33333_33282_006;
            C2 : constant :=
               -0.00019_84126_98412_69841_25918_947;
            C3 : constant :=
               +0.00000_27557_31922_39858_79426_917;
            C4 : constant :=
               -0.00000_00250_52108_38543_49235_944;
            C5 : constant :=
               +0.00000_00001_60590_43834_31740_557;
            C6 : constant :=
               -0.00000_00000_00764_71631_60587_969;
            C7 : constant :=
               +0.00000_00000_00002_81137_84941_828;
            C8 : constant :=
               -0.00000_00000_00000_00816_04793_976;
         begin
            FX := ((((((((C8*ARG2 + C7)*ARG2 + C6)*ARG2
               + C5)*ARG2 + C4)*ARG2 + C3)*ARG2 + C2)*ARG2
               + C1)*ARG2 + C0)*ARG2;
         end;
      when others =>
         null; -- Error, should be trapped in package body.
   end case;
   FX := ((FX + 0.5) + 0.5)*ARG;
else
   -- cosine approximations
   case REAL'DIGITS is
      when 1 .. 2 => -- MANTISSA <= 13
         declare
            C0 : constant := -0.49993;
            C1 : constant := +0.04081;
         begin
            FX := (C1*ARG2 + C0)*ARG2;
         end;
      when 3 .. 4 => -- MANTISSA <= 22
         declare
            C0 : constant := -0.49999_982;
            C1 : constant := +0.04166_141;
            C2 : constant := -0.00136_612;
         begin
            FX := ((C2*ARG2 + C1)*ARG2 + C0)*ARG2;
         end;
      when 5 .. 7 => -- MANTISSA <= 31
         declare
            C0 : constant := -0.49999_99997;
         C1 : constant := +0.04166_66507;
         C2 : constant := -0.00138_87589;
         C3 : constant := +0.00002_44638;
      begin
         FX := (((C3*ARG2 + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
      end;
   when 8 .. 10 => -- MANTISSA <= 41
      declare
         C0 : constant := -0.49999_99999_996;
         C1 : constant := +0.04166_66666_374;
         C2 : constant := -0.00138_88885_093;
         C3 : constant := +0.00002_47998_625;
         C4 : constant := -0.00000_02723_717;
      begin
         FX := ((((C4*ARG2 + C3)*ARG2 + C2)*ARG2 + C1)*ARG2
            + C0)*ARG2;
      end;
   when 11 .. 13 => -- MANTISSA <= 51
      declare
         C0 : constant := -0.49999_99999_99999_7;
         C1 : constant := +0.04166_66666_66630_9;
         C2 : constant := -0.00138_88888_88213_0;
         C3 : constant := +0.00002_48015_82624_3;
         C4 : constant := -0.00000_02755_58556_9;
         C5 : constant := +0.00000_00020_66551_6;
      begin
         FX := (((((C5*ARG2 + C4)*ARG2 + C3)*ARG2
            + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
      end;
   when 14 .. 16 => -- MANTISSA <= 61
      declare
         C0 : constant := -0.49999_99999_99999_9998;
         C1 : constant := +0.04166_66666_66666_6354;
         C2 : constant := -0.00138_88888_88888_0777;
         C3 : constant := +0.00002_48015_87293_6950;
         C4 : constant := -0.00000_02755_73155_6682;
         C5 : constant := +0.00000_00020_87588_6806;
         C6 : constant := -0.00000_00000_11368_0023;
      begin
         FX := ((((((C6*ARG2 + C5)*ARG2 + C4)*ARG2
            + C3)*ARG2 + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
      end;
   when 17 .. 19 => -- MANTISSA <= 72
      declare
         C0 : constant := -0.49999_99999_99999_99999_990;
         C1 : constant := +0.04166_66666_66666_66664_607;
         C2 : constant := -0.00138_88888_88888_88818_771;
         C3 : constant := +0.00002_48015_87301_57820_627;
         C4 : constant := -0.00000_02755_73192_18191_798;
         C5 : constant := +0.00000_00020_87675_49832_433;
         C6 : constant := -0.00000_00000_11470_36128_463;
         C7 : constant := +0.00000_00000_00047_41087_669;
      begin
         FX := (((((((C7*ARG2 + C6)*ARG2 + C5)*ARG2
            + C4)*ARG2 + C3)*ARG2 + C2)*ARG2 + C1)*ARG2
            + C0)*ARG2;
      end;
   when 20 .. 23 => -- MANTISSA <= 84
      declare
         C0 : constant :=
            -0.49999_99999_99999_99999_99999_6;
```

```
               C1 : constant :=
                  +0.04166_66666_66666_66666_66560_8;
               C2 : constant :=
                  -0.00138_88888_88888_88888_84313_5;
               C3 : constant :=
                  +0.00002_48015_87301_58729_39711_8;
               C4 : constant :=
                  -0.00000_02755_73192_23979_53980_7;
               C5 : constant :=
                  +0.00000_00020_87675_69848_93114_2;
               C6 : constant :=
                  -0.00000_00000_11470_74477_90156_8;
               C7 : constant :=
                  +0.00000_00000_00047_79345_98880_5;
               C8 : constant :=
                  -0.00000_00000_00000_15505_51278_5;
            begin
               FX := (((((((C8*ARG2 + C7)*ARG2 + C6)*ARG2
                  + C5)*ARG2 + C4)*ARG2 + C3)*ARG2 + C2)*ARG2
                  + C1)*ARG2 + C0)*ARG2;
            end;
         when 24 .. 25 => -- MANTISSA <= 90
            declare
               C0 : constant :=
                  -0.49999_99999_99999_99999_99999_999;
               C1 : constant :=
                  +0.04166_66666_66666_66666_66666_623;
               C2 : constant :=
                  -0.00138_88888_88888_88888_88886_555;
               C3 : constant :=
                  +0.00002_48015_87301_58730_15824_574;
               C4 : constant :=
                  -0.00000_02755_73192_23985_88554_743;
               C5 : constant :=
                  +0.00000_00020_87675_69878_65008_750;
               C6 : constant :=
                  -0.00000_00000_11470_74559_65909_134;
               C7 : constant :=
                  +0.00000_00000_00047_79477_07262_226;
               C8 : constant :=
                  -0.00000_00000_00000_15618_84882_782;
               C9 : constant :=
                  +0.00000_00000_00000_00040_82966_040;
            begin
               FX := ((((((((C9*ARG2 + C8)*ARG2 + C7)*ARG2
                  + C6)*ARG2 + C5)*ARG2 + C4)*ARG2 + C3)*ARG2
                  + C2)*ARG2 + C1)*ARG2 + C0)*ARG2;
            end;
         when others =>
            null; -- Error, should be trapped in package body.
      end case;
      FX := (FX + 0.5) + 0.5;
   end if;
   if N >= 2 then
      FX := - FX;
   end if;
   return FX;
end SIN_COS;
------------------------------------------------------------------
```

```
   function SIN(X : REAL; CYCLE : REAL := 2.0*PI) return REAL is
   begin
      return SIN_COS(X, CYCLE, COS_WANTED => FALSE);
   end SIN;
   -----------------------------------------------------------------------
   function COS(X : REAL; CYCLE : REAL := 2.0*PI) return REAL is
   begin
      return SIN_COS(X, CYCLE, COS_WANTED => TRUE);
   end COS;
   -----------------------------------------------------------------------
   -- Other functions coded similarly
   -----------------------------------------------------------------------
end GENERIC_MATH_FUNCTIONS;
--------------------------------------------------------------------------
```

In the sample coding of the basic mathematical functions given here, we have included the body of each function inside the package body. However, if facilities for partial loading are available, we recommend the use of body stubs and subunits as outlined in sections (g) and (h) of Chapter 4.

APPENDIX D - OPERATORS ON TYPE COMPLEX

Here we present two complete packages for complex arithmetic, as recommended at the end of section (d) of Chapter 5. The first uses the Cartesian definition of the type COMPLEX, introduced in section (a) of that chapter, and the second uses the alternative polar form, mentioned at the beginning of section (d).

a) Complex operators

```
with REAL_TYPES; use REAL_TYPES;
package COMPLEX_OPERATORS is

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

   function RE(X : COMPLEX) return REAL;
   function IM(X : COMPLEX) return REAL;
   function "abs"(X : COMPLEX) return REAL;
   function ARG(X : COMPLEX) return REAL;
   function C_TO_COMP(R : REAL; I : REAL := 0.0)
      return COMPLEX;
   function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX;
   function "+"(X : COMPLEX) return COMPLEX;
   function "-"(X : COMPLEX) return COMPLEX;
   function "+"(X,Y : COMPLEX) return COMPLEX;
   function "-"(X,Y : COMPLEX) return COMPLEX;
   function "*"(X,Y : COMPLEX) return COMPLEX;
   function "/"(X,Y : COMPLEX) return COMPLEX;
   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX;

   pragma INLINE(RE, IM, "abs", ARG, C_TO_COMP, P_TO_COMP,
      "+", "-", "*", "/", "**");

end COMPLEX_OPERATORS; -- specification


with MATH_FUNCTIONS;
package body COMPLEX_OPERATORS is

   use MATH_FUNCTIONS;

   function RE(X : COMPLEX) return REAL is
   begin
      return X.RE;
   end RE;

   function IM(X : COMPLEX) return REAL is
   begin
      return X.IM;
   end IM;
```

```
function "abs"(X : COMPLEX) return REAL is
   A,B : REAL;
begin
   if abs X.RE > abs X.IM then
      A := abs X.RE;
      B := abs X.IM;
   else
      A := abs X.IM;
      B := abs X.RE;
   end if;
   if A > 0.0 then
      return A*SQRT(1.0 + (B/A)**2);
   else
      return 0.0;
   end if;
end "abs";

function ARG(X : COMPLEX) return REAL is
begin
   return ARCTAN(X.IM, X.RE);
end ARG;

function C_TO_COMP(R : REAL; I : REAL := 0.0)
   return COMPLEX is
begin
   return (R, I);
end C_TO_COMP;

function P_TO_COMP(M : REAL; A : REAL := 0.0)
   return COMPLEX is
begin
   return (M*COS(A), M*SIN(A));
end P_TO_COMP;

function "+"(X : COMPLEX) return COMPLEX is
begin
   return X;
end "+";

function "-"(X : COMPLEX) return COMPLEX is
begin
   return (- X.RE, - X.IM);
end "-";

function "+"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE + Y.RE, X.IM + Y.IM);
end "+";

function "-"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE - Y.RE, X.IM - Y.IM);
end "-";

function "*"(X,Y : COMPLEX) return COMPLEX is
begin
   return (X.RE*Y.RE - X.IM*Y.IM, X.IM*Y.RE + X.RE*Y.IM);
end "*";
```

```
   function "/"(X,Y : COMPLEX) return COMPLEX is
      A,B : REAL;
   begin
      if abs Y.RE > abs Y.IM then
         A := Y.IM/Y.RE;
         B := A*Y.IM + Y.RE;
         return ((X.RE + A*X.IM)/B, (X.IM - A*X.RE)/B);
      else
         A := Y.RE/Y.IM;
         B := A*Y.RE + Y.IM;
         return ((A*X.RE + X.IM)/B, (A*X.IM - X.RE)/B);
      end if;
   end "/";

   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX is
      CMOD,CARG,R,THETA : REAL;
   begin
      CMOD := abs X;
      CARG := ARG(X);
      R := CMOD**N;
      THETA := REAL(N)*CARG;
      return (R*COS(THETA), R*SIN(THETA));
   end "**";

end COMPLEX_OPERATORS; -- body
```

b) Complex polar operators

```
with REAL_TYPES; use REAL_TYPES;
package COMPLEX_POLAR_OPERATORS is

   type COMPLEX is
      record
         CMOD,CARG : REAL;
      end record;

   function RE(X : COMPLEX) return REAL;
   function IM(X : COMPLEX) return REAL;
   function "abs"(X : COMPLEX) return REAL;
   function ARG(X : COMPLEX) return REAL;
   function C_TO_COMP(R : REAL; I : REAL := 0.0)
      return COMPLEX;
   function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX;
   function "+"(X : COMPLEX) return COMPLEX;
   function "-"(X : COMPLEX) return COMPLEX;
   function "+"(X,Y : COMPLEX) return COMPLEX;
   function "-"(X,Y : COMPLEX) return COMPLEX;
   function "*"(X,Y : COMPLEX) return COMPLEX;
   function "/"(X,Y : COMPLEX) return COMPLEX;
   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX;

   pragma INLINE(RE, IM, "abs", ARG, C_TO_COMP, P_TO_COMP,
      "+", "-", "*", "/", "**");

end COMPLEX_POLAR_OPERATORS; -- specification
```

```
with MATH_FUNCTIONS;
package body COMPLEX_POLAR_OPERATORS is

   use MATH_FUNCTIONS;

   function RE(X : COMPLEX) return REAL is
   begin
      return X.CMOD*COS(X.CARG);
   end RE;

   function IM(X : COMPLEX) return REAL is
   begin
      return X.CMOD*SIN(X.CARG);
   end IM;

   function "abs"(X : COMPLEX) return REAL is
   begin
      return X.CMOD;
   end "abs";

   function ARG(X : COMPLEX) return REAL is
   begin
      return X.CARG;
   end ARG;

   function C_TO_COMP(R : REAL; I : REAL := 0.0)
         return COMPLEX is
      A,B : REAL;
   begin
      if abs R > abs I then
         A := abs R;
         B := abs I;
      else
         A := abs I;
         B := abs R;
      end if;
      if A > 0.0 then
         return (A*SQRT(1.0 + (B/A)**2), ARCTAN(I, R));
      else
         return (0.0, 0.0);
      end if;
   end C_TO_COMP;

   function P_TO_COMP(M : REAL; A : REAL := 0.0)
      return COMPLEX is
   begin
      return (M, A);
   end P_TO_COMP;

   function "+"(X : COMPLEX) return COMPLEX is
   begin
      return X;
   end "+";
   function "-"(X : COMPLEX) return COMPLEX is
   begin
      if X.CARG >= 0.0 then
         return (X.CMOD, X.CARG - PI);
      else
         return (X.CMOD, X.CARG + PI);
      end if;
   end "-";

   function "+"(X,Y : COMPLEX) return COMPLEX is
   begin
      return C_TO_COMP(RE(X) + RE(Y), IM(X) + IM(Y));
   end "+";

   function "-"(X,Y : COMPLEX) return COMPLEX is
   begin
      return C_TO_COMP(RE(X) - RE(Y), IM(X) - IM(Y));
   end "-";

   function "*"(X,Y : COMPLEX) return COMPLEX is
      NEW_CARG : REAL := X.CARG + Y.CARG;
   begin
      if NEW_CARG > PI then
         NEW_CARG := NEW_CARG - PI - PI;
      elsif NEW_CARG < - PI then
         NEW_CARG := NEW_CARG + PI + PI;
      end if;
      return (X.CMOD*Y.CMOD, NEW_CARG);
   end "*";

   function "/"(X,Y : COMPLEX) return COMPLEX is
      NEW_CARG : REAL := X.CARG - Y.CARG;
   begin
      if NEW_CARG > PI then
         NEW_CARG := NEW_CARG - PI - PI;
      elsif NEW_CARG < - PI then
         NEW_CARG := NEW_CARG + PI + PI;
      end if;
      return (X.CMOD/Y.CMOD, NEW_CARG);
   end "/";

   function "**"(X : COMPLEX; N : INTEGER) return COMPLEX is
      NEW_CARG : REAL;
      MULT_PI : INTEGER;
   begin
      begin
         NEW_CARG := N*X.CARG;
         if NEW_CARG > 0.0 then
            MULT_PI := INTEGER(NEW_CARG/PI - 0.5);
         else
            MULT_PI := INTEGER(NEW_CARG/PI + 0.5);
         end if;
         MULT_PI := MULT_PI + (MULT_PI rem 2);
      exception
         when NUMERIC_ERROR => MULT_PI := 0; NEW_CARG := 0.0;
      end;
      return (X.CMOD**N, NEW_CARG - REAL(MULT_PI)*PI);
   end "**";

end COMPLEX_POLAR_OPERATORS; -- body
```

APPENDIX E - A LEAST-SQUARES PACKAGE

Here we present a complete package for the least-squares solution of an over-determined system of simultaneous linear algebraic equations, together with a short example program to illustrate its application. A brief description of each subprogram is given, in comment form, immediately after its specification. Further details of the algorithms used are given by Cox and Hammarling (1980).

```
-- ==================================================================
-- |    LEAST SQUARES PACKAGE FOR OVER-DETERMINED LINEAR SYSTEMS    |
-- ==================================================================

-- First written in Preliminary Ada by
-- Maurice G. Cox and Sven J. Hammarling, Winter 1979/80

-- Revised into ANSI Standard Ada by
-- Sven J. Hammarling and George T. Symm, Spring 1983

------------------------ CAUTIONARY MESSAGE -----------------------
--                                                               --
-- At the time that this package was written, no Ada compiler was --
-- available to the authors. Although some syntax checking has   --
-- been carried out, it must not be assumed that the procedures  --
-- and functions in the package are correct.                     --
-- A 'close' Algol 60 counterpart of this package has been       --
-- compiled and tested on several simple problems.               --
--                                                               --
-------------------------------------------------------------------

-- ****************************************************************
-- *                                                              *
-- * NOTE: The following assumptions are made here:-              *
-- *                                                              *
-- * (a) There is available a package REAL_TYPES of types         *
-- *     suitable for scientific computation, including           *
-- *        type REAL, with appropriate accuracy,                 *
-- *        type VECTOR, a one-dimensional array of REALs,        *
-- *        type MATRIX, a two-dimensional array of REALs.        *
-- *                                                              *
-- * (b) There is available a package MATH_FUNCTIONS of basic     *
-- *     mathematical functions, such as SQRT, applicable to      *
-- *     the type REAL.                                           *
-- *                                                              *
-- ****************************************************************


------------------------ SPECIFICATION PART -----------------------

with REAL_TYPES; use REAL_TYPES;
generic
   with procedure GET_ROW (
         XROW, YROW : out VECTOR;
         LAST_ROW   : out BOOLEAN );
package GENERIC_LEAST_SQUARES is
```

```
procedure SIMPLE_LEAST_SQUARES_SOLVER (
      DIG         : in INTEGER;
      BETA        : in out MATRIX; -- See *** Note *** below --
      SS          : in out VECTOR; -- See *** Note *** below --
      M, RANK     : out INTEGER;
      COND        : out REAL;
      SVD, FAIL   : out BOOLEAN );

-- SIMPLE_LEAST_SQUARES_SOLVER solves the
-- following problem. Given an M by N observation matrix X and
-- an M by Q matrix Y of right hand side vectors, determine
-- an N by Q matrix BETA of solution vectors that provides a
-- least squares solution to the over-determined system
-- X*BETA = Y. The Q element vector SS containing the
-- residual sums of squares is also computed.
--
-- The subprogram is designed to be used in circumstances in
-- which the number of rows of X may not be known in advance
-- and in which the rows of (X, Y) are supplied sequentially
-- in some manner, e.g. from an external device.
--
-- The first stage of the computation is to employ orthogonal
-- triangularization; the second stage, which can most
-- efficiently be carried out upon completion of the first,
-- is entered only if the condition number of the triangular
-- matrix obtained from the first stage (this number is equal
-- to the condition number of the specified observation
-- matrix) is such that the solution obtained by solving the
-- triangular system would be inadequate for practical
-- purposes. In this case SVD is returned as TRUE,
-- otherwise it is returned as FALSE. This
-- approach ensures that in cases for which X is clearly
-- of full rank, the solution is obtained efficiently
-- using orthogonal triangularization, whereas in near rank
-- deficient cases the more reliable but more expensive
-- singular value decomposition is employed.
--
-- The rows of (X, Y) are assumed to be supplied by means of
-- the subprogram GET_ROW, each call of which, from
-- procedure SIMPLE_LEAST_SQUARES_SOLVER (via TRIANGULARIZE),
-- returns the current row in XROW and YROW and a BOOLEAN
-- LAST_ROW indicating whether the current row is the
-- last to be processed.
--
-- The integer DIG specifies the number of correct decimal
-- digits in the user's data. If DIG > REAL'DIGITS then DIG
-- is replaced by REAL'DIGITS in the subprogram. On exit,
-- the subprogram returns the value of M, the estimated
-- rank RANK and the condition number COND of X.

-- *** Note ***
-- Comments referring to this note are attached to variables
-- of mode "in out" which are essentially of mode "out" but
-- which are used as working space within the subprogram body.
-- Library documentation should distinguish clearly between
-- such variables and those which require input values.
```

```
procedure INITIALIZE (
      U     : out VECTOR;
      THETA : out MATRIX;
      SS    : out VECTOR );

-- INITIALIZE initializes to zero the upper triangular
-- matrix U, the matrix THETA of right hand side vectors,
-- and the residual sums of squares SS.


procedure TRIANGULARIZE (
      U     : in out VECTOR;
      THETA : in out MATRIX;
      SS    : in out VECTOR;
      M     : out INTEGER );

-- TRIANGULARIZE updates the N by N upper triangular matrix U
-- with a sequence of rows obtained from the user supplied
-- procedure GET_ROW. The updated triangular matrix is
-- overwritten on U. U must be supplied, row by row, as an
-- N*(N + 1)/2 element vector. The N by Q right hand side
-- matrix THETA and the Q element vector SS containing the
-- residual sums of squares are also updated.
-- Each call to GET_ROW must return details of the
-- next observation to be processed. Specifically, the N
-- element vector XROW must contain the row of the
-- observation matrix and the Q element vector YROW the
-- corresponding right hand side values. LAST_ROW must
-- be returned as FALSE if there are more observations to be
-- processed and TRUE otherwise.
--
-- M returns the total number of observations processed.


procedure UPDATE_TRIANGLE (
      XROW, YROW : in out VECTOR;
      U          : in out VECTOR;
      THETA      : in out MATRIX;
      SS         : in out VECTOR );

-- UPDATE_TRIANGLE performs the transformation
--
--         (Q**T)*( U ) => ( U ) ,
--                ( X )    ( 0 )
--
-- where Q is an orthogonal matrix, U an N by N upper
-- triangular matrix and X an N element vector. U must
-- be supplied, row by row, as an N*(N + 1)/2 element vector.
-- The vector X must be supplied in XROW.
-- XROW is destroyed on exit. The transformation
--
--         (Q**T)*( THETA ) => ( THETA ) ,
--                (  YROW )    (  YROW )
--
-- where THETA is an N by Q matrix and YROW a Q element vector,
-- is also performed, and each component of SS is increased
-- additively by the square of the corresponding component
-- of YROW.
```

```
function INVERSE_CONDITION_NUMBER (
      U : in VECTOR ) return REAL;

-- INVERSE_CONDITION_NUMBER returns the reciprocal
-- of the condition number norm(U)*norm(U**(-1)), where norm
-- denotes Euclidean length, of the N by N upper triangular
-- matrix U. If the condition number would overflow then the
-- value zero is returned. U must be supplied, row by row,
-- as an N*(N + 1)/2 element vector.


procedure SOLVE_TRIANGLE (
      U    : in VECTOR;
      K    : in INTEGER;
      BETA : in out MATRIX );

-- SOLVE_TRIANGLE solves the upper triangular system
--
--      R(K)*BETA = THETA
--
-- for BETA, where R(K) is the leading K by K part of an N by N
-- upper triangular matrix R and N = BETA'LENGTH(1). The
-- equations are solved by backward substitution. R must be
-- supplied row by row in the N*(N + 1)/2 element vector U.
-- BETA must be an N by Q matrix. If N is greater than K then
-- rows K + 1 to N of BETA will be set equal to the
-- corresponding rows of THETA. The right hand sides must be
-- supplied in the matrix BETA, which on successful exit
-- will contain the solution vectors.
--
-- The calling subprogram should provide a handler for the
-- exception UNSOLVABLE, which will arise if overflow occurs
-- in computing BETA.


UNSOLVABLE : exception;


procedure BIDIAGONALIZE (
      U     : in out VECTOR;
      THETA : in out MATRIX;
      D, E  : in out VECTOR ); -- See *** Note *** above --

-- BIDIAGONALIZE reduces the upper triangular matrix U to
-- bidiagonal form. That is, U is factorized as
--
--      U = Q*B*(P**T),
--
-- where B is bidiagonal and Q and P are N by N orthogonal
-- matrices. B is overwritten on U and is also returned in
-- the N element vectors D and E such that D(I) = B(I, I)
-- and E(I) = B(I - 1, I). The matrix (Q**T)*THETA is also
-- returned in THETA. U must be supplied row by row as an
-- N*(N + 1)/2 element vector.
```

```
   procedure DIAGONALIZE (
         D, E        : in out VECTOR;
         THETA, PT   : in out MATRIX;
         FAIL        : out BOOLEAN );

   -- DIAGONALIZE reduces an upper bidiagonal matrix B to
   -- diagonal form. That is, B is factorized as
   --
   --          B = Q*D*(P**T),
   --
   -- where D is diagonal with non-negative diagonal elements,
   -- these being the singular values of B, and Q and P are N by N
   -- orthogonal matrices. B must be supplied in the vectors D
   -- and E such that D(I) = B(I, I) and E(I) = B(I - 1, I).
   -- The subprogram arranges the elements of the diagonal
   -- matrix to be in descending order, these being returned in
   -- D. On exit the elements of E will be negligible. The
   -- matrix (P**T)*PT is returned in PT. The matrix
   -- (Q**T)*THETA is returned in THETA. On normal return FAIL
   -- will be FALSE. In the extremely unlikely event that the
   -- QR-algorithm fails to converge in 50N iterations then
   -- FAIL is returned as TRUE. In this case D will not be
   -- quite diagonal, so that some elements of E will still be
   -- significant. A random shift of origin and a return call
   -- of this subprogram is likely to achieve convergence.


   procedure SVD_SOLVE (
         D     : in VECTOR;
         PT    : in MATRIX;
         TOL   : in REAL;
         THETA : in out MATRIX;
         SS    : in out VECTOR;
         RANK  : out INTEGER );

   -- SVD_SOLVE determines the rank of the diagonal matrix
   -- represented by the vector D. The N elements of D are
   -- assumed to be non-negative and to be in descending order
   -- of magnitude. The rank is returned in RANK. An element
   -- D(I) of D is regarded as negligible if D(I) <= TOL*D(1).
   -- The minimal least squares solution for (THETA - D*PT*BETA)
   -- is also returned. BETA is overwritten on THETA. The
   -- vector SS is updated, so that the sums of squares of the
   -- last N - RANK elements of the Jth column of THETA are
   -- added to SS(J).


end GENERIC_LEAST_SQUARES; -- specification

---------------------------------------------------------------------
```

```
------------------------------ BODY PART ---------------------------

with MATH_FUNCTIONS;
package body GENERIC_LEAST_SQUARES is
   use MATH_FUNCTIONS;

   -------------------------------------------------------------------

   -- Specifications of internal auxiliary subprograms:

   -------------------------------------------------------------------

   procedure FORM_PT (
         U  : in VECTOR;
         PT : in out MATRIX ); -- See *** Note *** above --

   -- FORM_PT forms the matrix P**T following subprogram
   -- BIDIAGONALIZE. U must be as supplied by subprogram
   -- BIDIAGONALIZE.


   procedure CHECK (
         T : in BOOLEAN );

   -- CHECK raises the exception CONSTRAINT_ERROR if T is FALSE.


   procedure ROTATION_PARAMETERS (
         A, B : in out REAL;
         C, S : out REAL );

   -- ROTATION_PARAMETERS forms the cosine C and sine S of
   -- the rotation angle whose tangent T is B/A, given the
   -- values of A and B. B is overwritten by T and A by
   -- SQRT(A**2 + B**2).


   procedure PLANE_ROTATION (
         C, S : in REAL;
         X, Y : in out VECTOR );

   -- PLANE_ ROTATION applies a plane rotation with parameters
   -- C and S to the row vectors X and Y and overwrites the result
   -- on X and Y. That is
   --
   --       (  C S )*( X ) => ( X ) .
   --       ( -S C ) ( Y )    ( Y )


   procedure EXTRACT_ROW (
         MAT : in MATRIX;
         I   : in INTEGER;
         ROW : out VECTOR );

   -- EXTRACT_ROW copies the Ith row of the matrix MAT
   -- into the vector ROW.
```

```
procedure REPLACE_ROW (
      ROW : in VECTOR;
      I   : in INTEGER;
      MAT : in out MATRIX );

-- REPLACE_ROW copies the vector ROW into
-- the Ith row of the matrix MAT.


function VECTOR_NORM (
      X : in VECTOR ) return REAL;

-- VECTOR_NORM returns the Euclidean length of the
-- vector X, unless this would overflow, in which case
-- REAL'SAFE_LARGE is returned.


procedure SCLSQS (
      X     : in VECTOR;
      SCALE : in out REAL;
      SUMSQ : in out REAL );

-- SCLSQS is a service routine for VECTOR_NORM
-- and INVERSE_CONDITION_NUMBER.


function TWO_NORM (
      SCALE, SUMSQ : in REAL ) return REAL;

-- TWO_NORM is a service routine for VECTOR_NORM
-- and INVERSE_CONDITION_NUMBER.


procedure SPLIT (
      K, P  : in INTEGER;
      D, E  : in out VECTOR;
      THETA : in out MATRIX );

-- SPLIT is a service routine for DIAGONALIZE.


procedure QR_STEP (
      K, P      : in INTEGER;
      D, E      : in out VECTOR;
      THETA, PT : in out MATRIX );

-- QR_STEP performs one implicit QR step for DIAGONALIZE.


procedure SWAP (
      X, Y : in out VECTOR );

-- SWAP interchanges the vectors X and Y.


procedure COS_AND_SIN (
      T    : in REAL;
      C, S : out REAL );

-- COS_AND_SIN returns the cosine C, always non-negative, and
-- sine S corresponding to a given value of the tangent T.
```

```
function TANGENT (
      A, B : in REAL ) return REAL;

-- TANGENT returns the value B/A, unless A = 0.0 or B/A would
-- overflow, in which case REAL'SAFE_LARGE is returned.


function SIGN (
      R : in REAL ) return REAL;

-- SIGN returns the value 1.0 or - 1.0
-- according as R >= 0 or R < 0.

-------------------------------------------------------------------

-- Bodies of all subprograms:

-------------------------------------------------------------------

procedure SIMPLE_LEAST_SQUARES_SOLVER (
      DIG        : in INTEGER;
      BETA       : in out MATRIX;
      SS         : in out VECTOR;
      M, RANK    : out INTEGER;
      COND       : out REAL;
      SVD, FAIL  : out BOOLEAN ) is
   N           : constant INTEGER := BETA'LENGTH(2);
   U           : VECTOR ( 1 .. N*(N + 1)/2 );
                 -- Upper triangular matrix by rows
   D, E        : VECTOR ( 1 .. N );
                 -- Diagonal and super-diagonal
   PT          : MATRIX ( 1 .. N, 1 .. N );
                 -- Orthogonal matrix
   INV_COND    : REAL;     -- Inverse condition number
   THRESHOLD   : REAL;     -- Relative threshold level
   MW, RANKW   : INTEGER;  -- Working variables
   FAILW       : BOOLEAN;  -- Working variable
begin
   if DIG <= REAL'DIGITS then
      THRESHOLD := 10.0**(- DIG);
   else
      THRESHOLD := REAL'EPSILON;
   end if;
   INITIALIZE ( U, BETA, SS );
   TRIANGULARIZE ( U, BETA, SS, MW );
   M := MW;
   INV_COND := INVERSE_CONDITION_NUMBER ( U );
                 -- between 0 and 1
   FAIL := FALSE;
   if INV_COND > THRESHOLD then
      SOLVE_TRIANGLE ( U, N, BETA );
      RANK := N;
      COND := 1.0/INV_COND;
      SVD := FALSE;
   else
      BIDIAGONALIZE ( U, BETA, D, E );
      FORM_PT ( U, PT );
      DIAGONALIZE ( D, E, BETA, PT, FAILW );
      if FAILW then
         FAIL := TRUE;
         return;
      end if;
      SVD_SOLVE ( D, PT, THRESHOLD, BETA, SS, RANKW );
      RANK := RANKW;
      if RANKW > 0 then
         COND := D(1)/D(RANKW);
      else
         COND := 1.0;
      end if;
      SVD := TRUE;
   end if;
end SIMPLE_LEAST_SQUARES_SOLVER;
```

```
procedure INITIALIZE (
      U     : out VECTOR;
      THETA : out MATRIX;
      SS    : out VECTOR ) is
   N : constant INTEGER := THETA'LENGTH(1);
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   CHECK ( SS'LENGTH = THETA'LENGTH(2) );
   U := ( U'RANGE => 0.0 );
   THETA := ( THETA'RANGE(1) =>
      ( THETA'RANGE(2) => 0.0 ) );
   SS := ( SS'RANGE => 0.0 );
end INITIALIZE;
```

```
procedure TRIANGULARIZE (
      U     : in out VECTOR;
      THETA : in out MATRIX;
      SS    : in out VECTOR;
      M     : out INTEGER ) is
   N        : constant INTEGER := THETA'LENGTH(1);
   Q        : constant INTEGER := THETA'LENGTH(2);
   XROW     : VECTOR ( 1 .. N );
   YROW     : VECTOR ( 1 .. Q );
   LAST_ROW : BOOLEAN := FALSE;
   MW       : INTEGER; -- Working variable
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   CHECK ( SS'LENGTH = Q );
   MW := 0;
   while not LAST_ROW loop
      MW := MW + 1;
      GET_ROW ( XROW, YROW, LAST_ROW );
      UPDATE_TRIANGLE ( XROW, YROW, U, THETA, SS );
   end loop;
   M := MW;
end TRIANGULARIZE;
```

```
procedure UPDATE_TRIANGLE (
      XROW, YROW : in out VECTOR;
      U          : in out VECTOR;
      THETA      : in out MATRIX;
      SS         : in out VECTOR ) is
   N      : constant INTEGER := XROW'LENGTH;
   Q      : constant INTEGER := YROW'LENGTH;
   IDU    : INTEGER := N + 2;
   IU     : INTEGER := - N;
   C, S   : REAL;
   THETAJ : VECTOR ( 1 .. Q );
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   CHECK ( THETA'LENGTH(1) = N and THETA'LENGTH(2) = Q );
   CHECK ( SS'LENGTH = Q );
   for J in 1 .. N loop
      IDU := IDU - 1;
      IU := IU + IDU;
      if XROW(J) /= 0.0 then
         ROTATION_PARAMETERS ( U(IU), XROW(J), C, S );
         PLANE_ROTATION ( C, S,
            U(IU + 1 .. IU + IDU - 2), XROW(J + 1 .. N) );
         EXTRACT_ROW ( THETA, J, THETAJ );
         PLANE_ROTATION ( C, S, THETAJ, YROW );
         REPLACE_ROW ( THETAJ, J, THETA );
      end if;
   end loop;
   for K in SS'RANGE loop
      SS(K) := SS(K) + YROW(K)**2;
   end loop;
end UPDATE_TRIANGLE;

-------------------------------------------------------------------

function INVERSE_CONDITION_NUMBER (
      U : in VECTOR ) return REAL is
   N     : constant INTEGER :=
           (INTEGER(SQRT(REAL(1 + 8*U'LENGTH))) - 1)/2;
   E     : MATRIX ( 1 .. N, 1 .. 1) :=
           ( 1 .. N => ( 1 .. 1 => 0.0 ) );
   EJ    : VECTOR ( 1 .. 1 );
   SCALE : REAL := 0.0;
   SUMSQ : REAL := 1.0;
   NORM  : REAL := VECTOR_NORM ( U );
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   for I in 1 .. N loop
      for J in 1 .. I - 1 loop
         E(J, 1) := 0.0;
      end loop;
      E(I, 1) := NORM;
      SOLVE_TRIANGLE( U, I, E );
      for J in 1 .. I loop
         EJ(1) := E(J, 1);
         SCLSQS ( EJ, SCALE, SUMSQ );
         E(J, 1) := EJ(1);
      end loop;
   end loop;
   NORM := TWO_NORM ( SCALE, SUMSQ );
   if NORM = REAL'SAFE_LARGE then
      return 0.0;
   else
      return 1.0/NORM;
   end if;
exception
   when UNSOLVABLE =>
      return 0.0;
end INVERSE_CONDITION_NUMBER;

-------------------------------------------------------------------

procedure SOLVE_TRIANGLE (
      U    : in VECTOR;
      K    : in INTEGER;
      BETA : in out MATRIX ) is
   N : constant INTEGER := BETA'LENGTH(1);
   Q : constant INTEGER := BETA'LENGTH(2);
   P : INTEGER;
   W : REAL;
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   for I in reverse 1 .. K loop
      P := (I - 1)*(2*N - I)/2 + K;
      for M in reverse I + 1 .. K loop
         W := U(P);
         for J in 1 .. Q loop
            BETA(I, J) := BETA(I, J) - W*BETA(M, J);
         end loop;
         P := P - 1;
      end loop;
      W := U(P);
      for J in 1 .. Q loop
         BETA(I, J) := BETA(I, J)/W;
      end loop;
   end loop;
exception
   when NUMERIC_ERROR =>
      raise UNSOLVABLE;
end SOLVE_TRIANGLE;

-------------------------------------------------------------------

procedure BIDIAGONALIZE (
      U     : in out VECTOR;
      THETA : in out MATRIX;
      D, E  : in out VECTOR ) is
   N                  : constant INTEGER := THETA'LENGTH(1);
   P                  : constant INTEGER := N*(N + 1)/2;
   ST1, ST2, KM2      : INTEGER;
   N1, N2, N3, T      : INTEGER;
   C, S, W            : REAL;
   THETAIP1, THETAI   : VECTOR ( 1 .. THETA'LENGTH(2) );
begin
   CHECK ( U'LENGTH = P );
   CHECK ( D'LENGTH = N );
   CHECK ( E'LENGTH = N );
```

```
-- Start main loop.
-- Kth step puts zeros into Kth column of U,
-- where (as in subsequent comments also) U is
-- regarded as an upper triangular matrix.

   for K in reverse 3 .. N loop
      KM2 := K - 2;
      N1 := K - 1;
      N2 := KM2;
      N3 := K - 3;
      ST1 := 1;
      ST2 := N + 1;
      for I in 1 .. KM2 loop

      -- Set up rotation Q(I,I+1)**T to annihilate U(I,K).
      -- This introduces an unwanted element in U(I,I+1)
      -- which is stored in W.

         ROTATION_PARAMETERS( U(ST2 + N2), U(ST1 + N1), C, S );
         W := S*U(ST1);
         U(ST1) := C*U(ST1);

      -- Apply Q(I,I+1)**T to U and then to THETA.

         PLANE_ROTATION ( C, S,
            U(ST2 .. ST2 + N2), U(ST1 + 1 .. ST1 + N1) );
         EXTRACT_ROW ( THETA, I + 1, THETAIP1 );
         EXTRACT_ROW ( THETA, I, THETAI );
         PLANE_ROTATION ( C, S, THETAIP1, THETAI );
         REPLACE_ROW ( THETAIP1, I + 1, THETA );
         REPLACE_ROW ( THETAI, I, THETA );

      -- Now set up rotation P(I,I+1) to annihilate W = U(I,I+1).
      -- Temporarily store the cos and sin in D(I) and E(I).

         ROTATION_PARAMETERS ( U(ST2), W, D(I), E(I) );
         U(ST1 + N1) := W;
         T := ST2 - ST1;
         ST1 := ST2;
         ST2 := ST2 + T - 1;
         N1 := N1 - 1;
         N2 := N2 - 1;
         N3 := N3 - 1;
      end loop;
      T := 0;

   -- Apply the transformations P(1,2), P(2,3), ...
   -- ..., P(K-2,K-1), row by row, to U.

      for I in 1 .. KM2 loop
         for J in I .. KM2 loop
            T := T + 1;
            W := U(T + 1);
            U(T + 1) := D(J)*W + E(J)*U(T);
            U(T) := D(J)*U(T) - E(J)*W;
         end loop;
         T := T + N - KM2;
      end loop;
   -- Put bidiagonal elements into D and E.

      D(K) := U(ST2);
      E(K) := U(ST1 + 1);
   end loop;
   if N > 1 then
      D(2) := U(N + 1);
      E(2) := U(2);
   end if;
   D(1) := U(1);
end BIDIAGONALIZE;

---------------------------------------------------------------------

procedure DIAGONALIZE (
      D, E       : in out VECTOR;
      THETA, PT  : in out MATRIX;
      FAIL       : out BOOLEAN ) is
   N                : constant INTEGER := D'LENGTH;
   Q                : constant INTEGER := THETA'LENGTH(2);
   MAXIT            : constant INTEGER := 50*N;
   ITER             : INTEGER := 0;
   P                : INTEGER;
   NORM             : REAL := abs D(1);
   TOL              : REAL;
   THETAI, THETAP : VECTOR ( 1 .. Q );
   PTI, PTP         : VECTOR ( 1 .. N );
begin
   CHECK ( E'LENGTH = N );
   CHECK ( THETA'LENGTH(1) = N );
   CHECK ( PT'LENGTH(1) = N and PT'LENGTH(2) = N );

-- Form NORM = max(abs B(I,J)) for the bidiagonal matrix B.
-- Then form the tolerance TOL, for negligible elements,
-- and MAXIT, the maximum permitted number of iterations
-- for the QR algorithm.

   for I in 2 .. N loop
      if NORM < abs D(I) then
         NORM := abs D(I);
      end if;
      if NORM < abs E(I) then
         NORM := abs E(I);
      end if;
   end loop;
   TOL := NORM*REAL'EPSILON;

-- Start main loop. A singular value is found
-- for each value of K.

   for K in reverse 2 .. N loop
      while ITER < MAXIT loop

      -- Now test for negligible elements. If E(P) is
      -- negligible start QR step at Pth row instead of
      -- 1st row. If D(P-1) is negligible force a split
      -- and again start QR step at Pth row. If P = K
      -- then a singular value has been found.
```

```
         P := K;
         while P > 1 loop
            if abs E(P) <= TOL then
               exit;
            end if;
            if abs D(P - 1) < TOL then
               SPLIT ( K, P, D, E, THETA );
               exit;
            end if;
            P := P - 1;
         end loop;
      if P >= K then
         exit;
      end if;
      ITER := ITER + 1;
      QR_STEP ( K, P, D, E, THETA, PT );
      end loop;
      if ITER = MAXIT then
         FAIL := TRUE;
         return;
      end if;
   end loop;

-- Now make singular values non-negative.

   for I in 1 .. N loop
      if D(I) < 0.0 then
         D(I) := - D(I);
         for J in 1 .. Q loop
            THETA(I, J) := - THETA(I, J);
         end loop;
      end if;
   end loop;

-- Now sort the singular values into descending order.
-- Currently a simple ripple sort is used but more
-- efficient techniques could probably be applied.

   for I in 1 .. N loop
      NORM := 0.0;
      P := I;
      for J in I .. N loop
         if D(J) > NORM then
            NORM := D(J);
            P := J;
         end if;
      end loop;
      if NORM = 0.0 then
         exit;
      end if;
      if P > I then
         D(P) := D(I);
         D(I) := NORM;
         EXTRACT_ROW ( THETA, I, THETAI );
         EXTRACT_ROW ( THETA, P, THETAP );
         SWAP ( THETAI, THETAP );
         REPLACE_ROW ( THETAI, I, THETA );
         REPLACE_ROW ( THETAP, P, THETA );
         EXTRACT_ROW ( PT, I, PTI );
         EXTRACT_ROW ( PT, P, PTP );
         SWAP ( PTI, PTP );
         REPLACE_ROW ( PTI, I, PT );
         REPLACE_ROW ( PTP, P, PT );
      end if;
   end loop;
   FAIL := FALSE;
end DIAGONALIZE;

-------------------------------------------------------------------

procedure SVD_SOLVE (
      D     : in VECTOR;
      PT    : in MATRIX;
      TOL   : in REAL;
      THETA : in out MATRIX;
      SS    : in out VECTOR;
      RANK  : out INTEGER ) is
   N       : constant INTEGER := THETA'LENGTH(1);
   Q       : constant INTEGER := SS'LENGTH;
   I       : INTEGER := 1;
   DEL, W  : REAL;
   WORK    : VECTOR ( 1 .. N );
   RANKW   : INTEGER; -- Working variable
begin
   CHECK ( PT'LENGTH(1) = N and PT'LENGTH(2) = N );
   CHECK ( D'LENGTH = N );
   CHECK ( THETA'LENGTH(2) = Q );

-- First determine rank.

   if TOL < REAL'EPSILON or TOL > 1.0 then
      W := REAL'EPSILON;
   else
      W := TOL;
   end if;
   DEL := W*D(1);
   while D(I) > DEL loop
      I := I + 1;
      exit when I = N + 1;
   end loop;
   RANKW := I - 1;
   RANK := RANKW;

-- Now update sums of squares.

   for I in RANKW + 1 .. N loop
      for J in 1 .. Q loop
         SS(J) := SS(J) + THETA(I, J)**2;
      end loop;
   end loop;

-- Now form (D**(-1))*THETA.

   for I in 1 .. RANKW loop
      W := D(I);
      for J in 1 .. Q loop
         THETA(I, J) := THETA(I, J)/W;
      end loop;
   end loop;
```

```
-- Now form P*(D**(-1))*THETA.

   for J in 1 .. Q loop
      for I in 1 .. N loop
         WORK(I) := 0.0;
      end loop;
      for K in 1 .. RANKW loop
         W := THETA(K, J);
         for I in 1 .. N loop
            WORK(I) := WORK(I) + W*PT(K, I);
         end loop;
      end loop;
      for I in 1 .. N loop
         THETA(I, J) := WORK(I);
      end loop;
   end loop;
end SVD_SOLVE;

-----------------------------------------------------------------

procedure FORM_PT (
      U  : in VECTOR;
      PT : in out MATRIX ) is
   N     : constant INTEGER := PT'LENGTH(1);
   KP1   : INTEGER;
   W     : REAL;
   C, S  : VECTOR ( 1 .. N );
begin
   CHECK ( U'LENGTH = N*(N + 1)/2 );
   CHECK ( PT'LENGTH(2) = N );
   PT(1, 1) := 1.0;
   if N > 1 then
      PT(1, 2) := 0.0;
      PT(2, 1) := 0.0;
      PT(2, 2) := 1.0;
   end if;
   for K in 2 .. N - 1 loop
      KP1 := K + 1;
      PT(K, KP1) := 0.0;
      for I in reverse 1 .. K - 1 loop
         COS_AND_SIN (
            U( (I - 1)*(2*N - I)/2 + KP1 ), C(I), S(I) );
         PT(I, KP1) := 0.0;
      end loop;
      for J in 1 .. K loop
         for I in reverse 1 .. K - 1 loop
            W := PT(J, I + 1);
            PT(J, I + 1) := C(I)*W - S(I)*PT(J, I);
            PT(J, I) := C(I)*PT(J, I) + S(I)*W;
         end loop;
         PT(KP1, J) := 0.0;
      end loop;
      PT(KP1, KP1) := 1.0;
   end loop;
end FORM_PT;

-----------------------------------------------------------------
```

```
procedure CHECK (
      T : in BOOLEAN ) is
begin
   if not T then
      raise CONSTRAINT_ERROR;
   end if;
end CHECK;

------------------------------------------------------------------

procedure ROTATION_PARAMETERS (
      A, B : in out REAL;
      C, S : out REAL ) is
   T      : REAL;
   CW, SW : REAL; -- Working variables
begin
   T := TANGENT ( A, B );
   COS_AND_SIN ( T, CW, SW );
   C := CW;
   S := SW;
   A := CW*A + SW*B;
   B := T;
end ROTATION_PARAMETERS;

------------------------------------------------------------------

procedure PLANE_ROTATION (
      C, S : in REAL;
      X, Y : in out VECTOR ) is
   XI, YI : REAL;
begin
   CHECK ( X'FIRST = Y'FIRST );
   CHECK ( X'LAST = Y'LAST );
   for I in X'RANGE loop
      XI := X(I);
      YI := Y(I);
      X(I) := C*XI + S*YI;
      Y(I) := - S*XI + C*YI;
   end loop;
end PLANE_ROTATION;

------------------------------------------------------------------

procedure EXTRACT_ROW (
      MAT : in MATRIX;
      I   : in INTEGER;
      ROW : out VECTOR ) is
   Q : constant INTEGER := MAT'LENGTH(2);
begin
   CHECK ( ROW'LENGTH = Q );
   CHECK ( I > 0 and I <= MAT'LENGTH(1) );
   for J in 1 .. Q loop
      ROW(J) := MAT(I, J);
   end loop;
end EXTRACT_ROW;

------------------------------------------------------------------
```

```
procedure REPLACE_ROW (
      ROW : in VECTOR;
      I   : in INTEGER;
      MAT : in out MATRIX ) is
   Q : constant INTEGER := MAT'LENGTH(2);
begin
   CHECK ( ROW'LENGTH = Q );
   CHECK ( I > 0 and I <= MAT'LENGTH(1) );
   for J in 1 .. Q loop
      MAT(I, J) := ROW(J);
   end loop;
end REPLACE_ROW;
```

```
function VECTOR_NORM (
      X : in VECTOR ) return REAL is
   SCALE : REAL := 0.0;
   SUMSQ : REAL := 1.0;
begin
   SCLSQS ( X, SCALE, SUMSQ );
   return TWO_NORM ( SCALE, SUMSQ );
end VECTOR_NORM;
```

```
procedure SCLSQS (
      X     : in VECTOR;
      SCALE : in out REAL;
      SUMSQ : in out REAL ) is
   ABSXI : REAL;
begin
   for I in X'RANGE loop
      if X(I) /= 0.0 then
         ABSXI := abs X(I);
         if SCALE < ABSXI then
            SUMSQ := SUMSQ*(SCALE/ABSXI)**2 + 1.0;
            SCALE := ABSXI;
         else
            SUMSQ := SUMSQ + (ABSXI/SCALE)**2;
         end if;
      end if;
   end loop;
end SCLSQS;
```

```
function TWO_NORM (
      SCALE, SUMSQ : in REAL ) return REAL is
begin
   return SCALE*SQRT(SUMSQ);
exception
   when NUMERIC_ERROR =>
      return REAL'SAFE_LARGE;
end TWO_NORM;
```

```
procedure SPLIT (
      K, P  : in INTEGER;
      D, E  : in out VECTOR;
      THETA : in out MATRIX ) is
   C, S                : REAL;
   X                   : REAL := E(P);
   THETAI, THETAPM1 : VECTOR ( 1 .. THETA'LENGTH(2) );

-- This routine is called by DIAGONALIZE if D(P-1) is negligible.
-- In this case E(P) can be made negligible
-- thus splitting the bidiagonal matrix.
-- Annihilating E(P) = B(P-1,P) introduces
-- an unwanted element in B(P-1,P+1).
-- This element is chased along and off the
-- end of the (P-1)th row of B by rotations
-- Q(P+1,P-1)**T, Q(P+2,P-1)**T, ..., Q(K,P-1)**T.
-- THETA has to be updated by these transformations.

begin
   E(P) := 0.0;
   for I in P .. K loop
      ROTATION_PARAMETERS( D(I), X, C, S );
      if I < K then
         X := -S*E(I + 1);
         E(I + 1) := C*E(I + 1);
      end if;
      EXTRACT_ROW ( THETA, I, THETAI );
      EXTRACT_ROW ( THETA, P - 1, THETAPM1 );
      PLANE_ROTATION( C, S, THETAI, THETAPM1 );
      REPLACE_ROW ( THETAI, I, THETA );
      REPLACE_ROW ( THETAPM1, P - 1, THETA );
   end loop;
end SPLIT;

------------------------------------------------------------------

procedure QR_STEP (
      K, P      : in INTEGER;
      D, E      : in out VECTOR;
      THETA, PT : in out MATRIX ) is
   C, S, F, G, W, X : REAL;
   PTIM1, PTI       : VECTOR ( 1 .. PT'LENGTH(2) );
   THETAIM1, THETAI : VECTOR ( 1 .. THETA'LENGTH(2) );
begin

-- Form the shift parameters.
-- First plane rotation is chosen to annihilate X in the vector
--
--       ( F ).
--       ( X )

   if K = 2 then
      F := ( D(K - 1) - D(K) )*( D(K - 1) + D(K) ) - E(K)**2;
   else
      F := ( D(K - 1) - D(K) )*( D(K - 1) + D(K) )
           + ( E(K - 1) - E(K) )*( E(K - 1) + E(K) );
   end if;
   F := F/( 2.0*E(K)*D(K - 1) );
   if F >= 0.0 then
      G := SQRT( 1.0 + F**2 );
   else
      G := - SQRT( 1.0 + F**2 );
   end if;
   W := E(K)*( E(K) - D(K - 1)/(F + G) );
   F := ( D(P) - D(K) )*( D(P) + D(K) ) - W;
   X := D(P)*E(P + 1);
   for I in P + 1 .. K loop

   -- Apply initial plane rotation P(P,P+1) and then
   -- chase zeros by rotations Q(P,P+1)**T, P(P+1,P+2),
   -- Q(P+1,P+2)**T, ..., P(K-1,K), Q(K-1,K)**T.
   -- The P matrices have to be accumulated in P**T
   -- and THETA has to be updated with the Q**T matrices.

      ROTATION_PARAMETERS( F, X, C, S );
      if I > P + 1 then
         E(I - 1) := F;
      end if;
      F := C*D(I - 1) + S*E(I);
      E(I) := C*E(I) - S*D(I - 1);
      X := S*D(I);
      D(I) := C*D(I);
      EXTRACT_ROW ( PT, I - 1, PTIM1 );
      EXTRACT_ROW ( PT, I, PTI );
      PLANE_ROTATION( C, S, PTIM1, PTI );
      REPLACE_ROW ( PTIM1, I - 1, PT );
      REPLACE_ROW ( PTI, I, PT );
      ROTATION_PARAMETERS( F, X, C, S );
      D(I - 1) := F;
      F := C*E(I) + S*D(I);
      D(I) := C*D(I) - S*E(I);
      if I < K then
         X := S*E(I + 1);
         E(I + 1) := C*E(I + 1);
      end if;
      EXTRACT_ROW ( THETA, I - 1, THETAIM1 );
      EXTRACT_ROW ( THETA, I, THETAI );
      PLANE_ROTATION( C, S, THETAIM1, THETAI );
      REPLACE_ROW ( THETAIM1, I - 1, THETA );
      REPLACE_ROW ( THETAI, I, THETA );
   end loop;
   E(K) := F;
end QR_STEP;

-------------------------------------------------------------------

procedure SWAP (
      X, Y : in out VECTOR ) is
   T : constant VECTOR ( X'RANGE ) := X;
begin
   X := Y;
   Y := T;
end SWAP;

-------------------------------------------------------------------
```

```
procedure COS_AND_SIN (
      T    : in REAL;
      C, S : out REAL ) is
   CW : REAL; -- Working variable
begin
   CW := 1.0/SQRT ( T**2 + 1.0 );
   C := CW;
   S := CW*T;
exception
   when NUMERIC_ERROR =>
      C := 1.0/abs T;
      S := SIGN(T);
end COS_AND_SIN;

-------------------------------------------------------------------

function TANGENT (
      A, B : in REAL ) return REAL is
begin
   return B/A;
exception
   when NUMERIC_ERROR =>
      return REAL'SAFE_LARGE;
end TANGENT;

-------------------------------------------------------------------

function SIGN (
      R : in REAL ) return REAL is
begin
   if R >= 0.0 then
      return 1.0;
   else
      return - 1.0;
   end if;
end SIGN;

-------------------------------------------------------------------

end GENERIC_LEAST_SQUARES; -- body

----------------------------------------------------------------------
```

```
-- Illustrative use of the package GENERIC_LEAST_SQUARES
-- ====================================================

-- The following example illustrates how the procedures in
-- the package may be utilized to solve a simple least
-- squares problem.

-- It is assumed that each row of (X, Y) is supplied
-- in turn through procedure GET_ROW from the
-- input file. The solution vectors BETA and other
-- useful information will be written to the output file.

-----------------------------------------------------------------------

with REAL_TYPES, TEXT_IO;
package MATH_IO is
   use REAL_TYPES, TEXT_IO;
   package INT_IO is new INTEGER_IO ( INTEGER );
   package BOOL_IO is new ENUMERATION_IO ( BOOLEAN );
   package REAL_IO is new FLOAT_IO ( REAL );
end MATH_IO;

-----------------------------------------------------------------------

with MATH_IO, REAL_TYPES;
use MATH_IO, REAL_TYPES;
procedure GET_ROW (
      XROW, YROW : out VECTOR;
      LAST_ROW   : out BOOLEAN ) is
   use BOOL_IO, REAL_IO;
   N, Q : INTEGER;
begin
   N := XROW'LENGTH;
   Q := YROW'LENGTH;
   GET ( LAST_ROW );
   for J in 1 .. N loop
      GET ( XROW(J) );
   end loop;
   for J in 1 .. Q loop
      GET ( YROW(J) );
   end loop;
end GET_ROW;

-----------------------------------------------------------------------

with GENERIC_LEAST_SQUARES, GET_ROW;
package LEAST_SQUARES is new GENERIC_LEAST_SQUARES ( GET_ROW );

-----------------------------------------------------------------------
```

```
with TEXT_IO, MATH_IO, REAL_TYPES, LEAST_SQUARES;
use MATH_IO;
procedure EXAMPLE_PROGRAM is
   use TEXT_IO, INT_IO, REAL_IO, REAL_TYPES;

   M    : INTEGER; -- Number of rows of observation matrix X
   N    : INTEGER; -- Number of columns of X
   Q    : INTEGER; -- Number of right hand side vectors
   DIG  : INTEGER; -- Number of correct decimal digits
                   -- in user's data
   RANK : INTEGER; -- Estimated rank of X
   COND : REAL;    -- Computed condition number of X
   SVD  : BOOLEAN; -- Returned as TRUE if singular value
                   -- decomposition is employed, otherwise FALSE.
   FAIL : BOOLEAN; -- Failure parameter for
                   -- LEAST_SQUARES.SIMPLE_LEAST_SQUARES_SOLVER

begin

   GET ( N );
   GET ( Q );
   GET ( DIG );

   declare
      BETA : MATRIX ( 1 .. N, 1 .. Q );
             -- Matrix of solution vectors
      SS   : VECTOR ( 1 .. Q );
             -- Residual sums of squares

      procedure SOLVER (
            DIG        : in INTEGER;
            BETA       : in out MATRIX; -- See *** Note *** above --
            SS         : in out VECTOR; -- See *** Note *** above --
            M, RANK    : out INTEGER;
            COND       : out REAL;
            SVD, FAIL  : out BOOLEAN )
         renames LEAST_SQUARES.SIMPLE_LEAST_SQUARES_SOLVER;

   begin
      SOLVER ( DIG, BETA, SS, M, RANK, COND, SVD, FAIL );
      if FAIL then
         PUT ( "LEAST_SQUARES.SIMPLE_LEAST_SQUARES_SOLVER"
            & " has failed to converge" );
         return;
      end if;
      PUT ( "Number of observations processed is " );
      PUT ( M );
      NEW_LINE;
      NEW_LINE;
      PUT ( "Estimated rank is " );
      PUT ( RANK );
      NEW_LINE;
      NEW_LINE;
      PUT ( "Condition number of reduced matrix is " );
      PUT ( COND );
      if SVD then
         NEW_LINE;
         NEW_LINE;
         PUT ( "Singular value decomposition has been used" );
      end if;
      NEW_LINE;
      NEW_LINE;
      PUT ( "Residual sums of squares" );
      NEW_LINE;
      for J in 1 .. Q loop
         PUT ( SS(J) );
      end loop;
      NEW_LINE;
      NEW_LINE;
      PUT ( "Solution vector(s)" );
      NEW_LINE;
      for I in 1 .. N loop
         NEW_LINE;
         for J in 1 .. Q loop
            PUT ( BETA(I, J) );
         end loop;
      end loop;
   end;

end EXAMPLE_PROGRAM;
```

Unfortunately it has not yet been possible to test run this program, but a recent version of the package (of which the above is only a slight modification) has been successfully compiled on a Data General/ROLM Ada Workstation.

APPENDIX F - THE IEC FLOATING-POINT STANDARD AND ADA

The design of floating-point hardware units is a very expensive business in view of their complexity. Moreover, any differences between such units clearly restrict the portability of numerical software from one machine to another. Consequently, the IEEE have established a working group (IEEE Task P754) to develop a standard for binary floating-point arithmetic. Draft 8.0 of this (proposed) standard has been offered for public comment (IEEE, 1981) and is regarded as a bold attempt to overcome many of the inadequacies of current floating-point hardware.

This IEEE proposal has now been revised to Draft 10.0 but it has not yet (in January 1984) been published as an IEEE Standard. However, the International Electrotechnical Commission (IEC) has published a Standard (IEC, 1982), based upon Draft 8.0 of the IEEE proposal. Moreover, both Intel and Motorola have produced silicon implementations based upon early drafts of the IEEE Standard. The differences between the IEC and IEEE Standards, though very annoying, are thought to be relatively minor. The following comments on the IEC Standard therefore relate to the IEEE versions also.

a) Comments on the Standard

The Standard can be approximately described as a conventional 32/64-bit floating-point system with frills. All the interest, difficulties and problems rest upon the frills. Both the 32- and 64-bit formats use the "hidden bit" representation whereby the most significant bit of the mantissa is not stored. This gives an extra bit of precision without any loss of information. The hidden bit has been used by DEC on the PDP11 for some years but is comparatively rare for hardware systems. The ZX81 and BBC micros use the hidden bit format for their software systems.

Let us consider the various aspects of the frills in turn:

1. Overflow. Overflow itself is conventional except that the largest exponent value is reserved for special values (see NaN and infinities below).

2. Underflow. The Standard implements "gradual underflow" whereby the accuracy loss of underflow is gradual rather than sudden. It is a nice technique for reducing the number of significant figures lost due to underflow. One exponent value (the smallest) is reserved for underflowed values (and zero). This implies an ability to handle more negative powers of two than positive ones (so that reciprocation must be treated with care). Numerical software will perform more "gracefully" on a system with gradual underflow than on one with an abrupt cut-off to floating-point zero.

3. Rounding. This is logically very nice but quite complex. One can require computations to be performed exactly (for integer values, say), or rounded down, rounded up or truncated. The complexity of this Standard arises not only from the variety of rounding methods supported, but also from the need to provide a method of setting the current rounding mode. A conforming implementation is required to provide this mode-setting mechanism to the end user. The advantage of the rounding modes is that it becomes practical to explore the rounding characteristics of an algorithm by repeating computations in different modes and comparing the results. Such comparisons are barely practicable with existing hardware.

4. Not a Number (NaN). The Standard introduces special values called NaNs to allow delayed detection of overflow and underflow in a controlled manner. Ordinarily with overflow, one must halt a computation and provide a recovery routine. However, with this Standard, all values calculated from an overflow condition will be distinguished so that recovery can be handled at a more convenient point. Since this mechanism is quite new, it is not clear just how useful it will be. NaNs must be regarded as an experimental frill. On the other hand, there is a considerable potential for NaNs. If an algorithm is inherently stable but has problems in keeping values within range, then NaNs could be used as a method of detection of the need to rescale at points where this is convenient. Of course, use of such methods implies a reliance upon the IEC Standard which makes the software non-portable.

5. Infinities. The floating-point values are extended with two infinite values, minus infinity and plus infinity. This allows one to do interval arithmetic without making overflow a special case.

6. Extended precisions. In addition to the 32/64-bit formats, an extension can be provided to one or both of these. It appears to be the intention that these formats should be used for computations within registers (as on the Intel 8087 chip). The extended formats are not precisely defined so that their use could give more accuracy (or exponent range) for calculations performed entirely within registers.

To summarise, the IEC Standard is quite complex. It has features which numerical analysts can exploit with advantage; however, this would make such software non-portable to conventional floating-point units. The opinion has been expressed that the system is over-complex. Certainly the IEC Standard is too complex in its entirety for programmers who are not numerical analysts. Hence it will be important to provide a system with defaults which give a conventional system. Numerical analysts could then provide additional facilities in such a manner that ordinary computations were unaffected.

b) Relationship with Ada

There is broad agreement between the IEC Standard and Ada as follows:

1. The IEC Standard is a binary, conventional floating-point system in line with the Ada model.

2. Ada allows computations to be performed with more precision than requested, which with an IEC system would allow the use of extended precision.

3. Ada permits gradual underflow.

4. The NaNs can be regarded as machine numbers (though not the only ones) which are not model numbers in the Ada sense.

However, there are some incompatibilities between the full IEC Standard and Ada. In particular, problems arise from the requirement that "the actual environment which the programmer or user of the system sees" should conform to the Standard. This implies that it is not sufficient merely to use an IEC system to implement the Ada floating-point model; one must make the full facilities of the Standard available to the programmer. This clearly conflicts with any language standard which attempts to provide an implementation-independent specification of floating-point arithmetic. Some problems are:

1. How should a Standard Ada system provide:
    a) extended precision,
    b) control over rounding,
    c) infinities, and
    d) literals representing NaNs?

2. The IEC trap handling concept requires that a value should be returned as the result of an operation which raises a trap in lieu of an exception. This conflicts with the Ada mechanism where values are always lost when, for example, the NUMERIC_ERROR exception is raised on the predefined floating-point operations.

3. The IEC trap handler must be able to access information that is lost in Ada, for example information concerning the kind of operation that was being performed when the trap was raised, the corresponding operand values, etc.

APPENDIX G - TOPICS REQUIRING FURTHER STUDY

Here we list a number of topics which are related to the present project but for which, for one reason or another, no guidelines are given in this report.

a) Topics outside the scope of this project

These topics, to which little attention has been given here, are nevertheless of considerable importance, particularly for library implementors:

- Interfaces with existing libraries in other languages (discussed briefly in Chapter 6).
- Libraries for vector and parallel-processing machines (discussed briefly in Chapter 9).
- Libraries using abstract floating-point types.
- Arithmetic using model numbers (see Wallis, 1983).
- Testing of library software.
- Documentation of library software.

b) Topics omitted due to lack of resources

No attention has been given to these topics:

- Fixed-point arithmetic.

  The contractors have little experience of fixed-point computation and, although fixed-point arithmetic is relevant to some specialised real-time computations, no feedback has been forthcoming regarding which issues need to be addressed. Fortunately, with the advent of the IEC Standard for floating-point arithmetic, with silicon implementations such as the Intel 8087, the importance of fixed-point arithmetic may be diminished in the future.

- Tasks as parameters.

REFERENCES

Abramowitz, M. and Stegun, I.A., eds. Handbook of mathematical functions, Dover, New York, 1965.

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

Barnes, J.G.P. Programming in Ada, Addison-Wesley, London, 1982.

Blum, E.K. Programming parallel numerical algorithms in Ada. The relationship between numerical computation and programming languages, edited by J.K. Reid, North Holland, Amsterdam, 1982, 297-304.

Brown, W.S. and Feldman, S.I. Environment parameters and basic functions for floating-point computation. ACM Trans. Math. Softw., 1980, 6, 510-523.

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

Cox, M.G. and Hammarling, S.J. Evaluation of the language Ada for use in numerical computations. NPL Report DNACS 30/80, July 1980.

Firth, R. Draft specification of a basic mathematical library for the high order programming language Ada, Royal Military College of Science, Shrivenham, 1982.

Ford, B. Parameterization of the environment for transportable numerical software. ACM Trans. Math. Softw., 1978, 4, 100-103.

Ford, B., Bentley, J., Du Croz, J.J. and Hague, S.J. The NAG Library "machine". Software Pract. Exper., 1979, 9, 56-72.

Hammarling, S.J. and Wichmann, B.A. Numerical packages in Ada. The relationship between numerical computation and programming languages, edited by J.K. Reid, North Holland, Amsterdam, 1982, 225-244.

Hemker, P.W., ed. NUMAL, a library of numerical procedures in Algol 60, Mathematisch Centrum, Amsterdam, 1981.

Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M. and Sherman, M. Studies in Ada style, Springer-Verlag, New York, 1981.

IEC. Binary floating point arithmetic for microprocessor systems, IEC Standard, Publication 559, International Electrotechnical Commission, Geneva, 1982.

IEEE. A proposed standard for binary floating-point arithmetic. Computer, 1981, 14(3), 51-62.

NAG. A guide for contributors to NAG Algol 68, Numerical Algorithms Group, Oxford, 1982.

NAG. The NAG Algol 68 Mark 3 Library - Contents summary, Numerical Algorithms Group, Oxford, 1983.

Nissen, J.C.D. and Wallis, P.J.L., eds. Portability and style in Ada, Cambridge University Press, Cambridge, 1984.

Reid, J.K. Functions for manipulating floating-point numbers. Harwell Report CSS 76, 1979.

Wallis, P.J.L. Ada floating-point arithmetic as a basis for portable numerical software. Proceedings of the 6th Symposium on Computer Arithmetic, IEEE, Computer Society Press, Silver Spring, 1983, 79-81.

Wichmann, B.A. Tutorial material on the real data types in Ada. NPL Report DITC 34/83, January, 1984.

Wichmann, B.A. and Hill, I.D. A pseudo-random number generator. NPL Report DITC 6/82, June 1982.

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