

# Tutorial for the NAG Fortran 90 Library

## Contents

<b>0</b>	<b>How to Use this Tutorial</b>	0.2.2
<b>1</b>	<b>Example 1: Displaying Information About the Library</b>	0.2.2
1.1	Accessing the Library .....	0.2.2
1.2	Finding the Documentation You Need .....	0.2.2
1.3	A First Program .....	0.2.3
1.4	Remarks on Programming Style and Conventions .....	0.2.3
1.5	USE Statements and Compile-time Errors .....	0.2.4
<b>2</b>	<b>Example 2: Evaluating a Special Function</b>	0.2.4
2.1	Arguments of Real Type .....	0.2.4
2.2	Precision and Kind Values .....	0.2.6
2.3	Portable Programming for Real Data .....	0.2.6
2.4	Arguments of intent(in) and Passing Constants as Arguments ...	0.2.7
2.5	A First Look at Run-time Errors .....	0.2.8
<b>3</b>	<b>Example 3: Summary Statistics of Univariate Data</b>	0.2.9
3.1	A First Look at Array Arguments .....	0.2.9
3.2	Passing Array Sections as Arguments .....	0.2.10
3.3	Using Allocatable Arrays .....	0.2.10
3.4	Optional Arguments and Argument Keywords .....	0.2.11
<b>4</b>	<b>Example 4: Eigenvalues and Eigenvectors</b>	0.2.13
4.1	Generic Procedures for Different Data Types .....	0.2.13
4.2	Reading and Writing Two-dimensional Arrays .....	0.2.15
<b>5</b>	<b>Example 5: Solving Systems of Linear Equations</b>	0.2.16
5.1	More on Genericity .....	0.2.16
5.2	Arguments of intent(inout) .....	0.2.18
5.3	Sensitivity of Numerical Results .....	0.2.19
5.4	Handling Warning Exits from the Library .....	0.2.21
<b>6</b>	<b>Example 6: Finding a Solution of a Single Nonlinear Equation</b>	0.2.23
6.1	A First Look at Procedure Arguments .....	0.2.23
6.2	Embedding User-supplied Procedures in Modules .....	0.2.24
6.3	Using Modules to Share Data .....	0.2.25
<b>7</b>	<b>Example 7: One-dimensional Quadrature</b>	0.2.26
7.1	An Array-valued User-supplied Function .....	0.2.26
7.2	Handling Failure Exits from the Library .....	0.2.28
7.3	An Argument Which is an Array Pointer .....	0.2.29
<b>8</b>	<b>Example 8: Interpolation</b>	0.2.30
8.1	Using a Structure to Communicate Between Procedures .....	0.2.30
8.2	Derived Types and Precision .....	0.2.32
<b>9</b>	<b>Example 9: Nonlinear Least-squares</b>	0.2.34
9.1	A Procedure Argument with an Optional Argument .....	0.2.34
9.2	Using a Structure for Option Setting .....	0.2.41
<b>10</b>	<b>Concluding Remarks</b>	0.2.42

## 0 How to Use this Tutorial

This tutorial describes those aspects of the Fortran 90 language that are important when calling NAG *f*90 procedures. It also explains any relevant conventions that have been adopted in the design and documentation of the Library.

It is not a self-contained introduction to Fortran 90; it assumes that you already have some general knowledge of the language.

The tutorial presents a series of simple example programs, illustrating the use of NAG *f*90. The numerical or statistical problems which have been chosen for the examples should be simple to understand. They have been selected to highlight certain features of the Fortran 90 language or of NAG *f*90. This tutorial does not aim to instruct you in the numerical or statistical background of the chosen problems.

The examples are intended to be followed in order, one after another. If you are tempted to skip ahead, in order to learn about a particular feature, be prepared to backtrack.

You will need to refer to the documentation of the procedures which are used in the examples.

If you run the example programs on your own compiler and machine, you should not expect to get exactly the same results as those which are reproduced in this document, especially when numerical values are printed to high precision.

There are several references in this Tutorial to the behaviour of the NAGWare *f*90 compiler, and examples of its error messages; other compilers may behave differently, and will issue different messages.

All the information presented in this Tutorial is also presented in a more condensed form in the **Essential Introduction**. That document is designed as a reference document for programmers who are reasonably familiar with Fortran 90. You are definitely *not* expected to read the Essential Introduction before reading this Tutorial. But after working through the Tutorial, you should find the Essential Introduction helpful for checking up on specific points.

## 1 Example 1: Displaying Information About the Library

### 1.1 Accessing the Library

In order to use NAG *f*90, you will need to know how to compile a Fortran 90 program, link it to NAG *f*90 and execute it. The details may vary from one computing system to another, and in some respects from one installation to another. You must refer to local information provided by your installation.

Assume that you have this information, and want to check whether you can indeed execute a program which calls NAG *f*90 procedures.

The simplest task that you can ask the Library to do is to display information about the Library as it is installed on your system (for example, the type of machine, the compiler, the precision, the release). This task is performed by the procedure `nag_lib_ident`, and it is in the module `nag_lib_support` (1.1), which provides general support facilities for the Library. The term *procedure* is used to refer to both subroutines and functions.

At this stage, as far as the modules in NAG *f*90 are concerned, you can think of them as ‘packages’ which bundle together small groups of related procedures, and in some cases definitions of derived types or named constants as well.

### 1.2 Finding the Documentation You Need

Each module in the Library is documented in a separate *module document*. A module document is the basic unit of documentation for NAG *f*90: it contains sections which describe the individual procedures in the module.

Therefore, in order to use the procedure `nag_lib_ident`, you must refer to the module document for the module `nag_lib_support` (1.1), which is the first module document in Chapter 1.

### 1.3 A First Program

The procedure `nag_lib_ident` is trivial: it is a subroutine with no arguments which writes information to the standard output unit. Section 2 of the procedure specification is headed **Usage** and summarises how to call the procedure. It reads:

```
USE nag_lib_support
CALL nag_lib_ident
```

This is intended to remind you that the `USE` statement is *essential*; it names the module in which the procedure is to be found. You must always include the correct `USE` statement in any program unit which calls a NAG *f*90 procedure. `USE` statements must precede all other statements in a program unit, apart from the initial `PROGRAM`, `SUBROUTINE`, `FUNCTION` or `MODULE` statement.

Here is a complete program to call the procedure `nag_lib_ident`:

```
PROGRAM tutorial_ex1a
  USE nag_lib_support, ONLY : nag_lib_ident
  IMPLICIT NONE
  CALL nag_lib_ident
END PROGRAM tutorial_ex1a
```

If you try it, you should see something like this appear on the standard output unit (normally the screen):

```
*** Start of NAG Fortran 90 Library implementation details ***

Implementation title: Generalised Base Version
  Product Code: FNBAS04D9
    Release: 4
    Precision: double (KIND= 2)

*** End of NAG Fortran 90 Library implementation details ***
```

You will probably not see exactly the same text, but that is as it should be: the precise details depend on the system on which your program is running. It is enough for now to have verified that you have correctly called a NAG *f*90 procedure.

### 1.4 Remarks on Programming Style and Conventions

The program could be stripped down to its bare bones thus:

```
USE nag_lib_support
CALL nag_lib_ident
END
```

However, we have adopted a few simple elements of programming style for all the example programs in this Tutorial, in order to show how Fortran 90 can assist what we believe to be good programming practice, as the following remarks explain.

- The `PROGRAM` statement is not essential, but is included in order to identify the program by name. It is a useful practice to include the name of each program unit in its `END` statement also — it provides extra signposts if you have a large program containing several program units.
- The `ONLY` qualifier in the `USE` statement has the effect of documenting which entities are intended to be accessed from the module — again a useful practice which is helpful in a more complex program unit, especially if it accesses more than one module.
- The `IMPLICIT NONE` statement ensures that all variables must have their types explicitly declared. It is unnecessary here because the program does not use any variables, but in less trivial programs it is a useful protection against unexpected consequences of the implicit typing rules of Fortran 90 (the same as in Fortran 77) — for example, a supposed integer `p`, which would be taken as real unless explicitly declared to be integer.
- Fortran 90 statement keywords (like `CALL`) appear in upper case, as do the names of intrinsic procedures; other names (of variables, procedures and so on) appear in lower case. This convention (along with other details of layout) has been enforced with the aid of the NAGWare `f90` tool `nag-polish90`.

These elements of style will be adopted in all the example programs in this Tutorial. Similar stylistic practices are adopted in the example programs in the module documents.

## 1.5 USE Statements and Compile-time Errors

The most important thing to note about the program `tutorial_ex1a` is the `USE` statement. The `USE` statement accesses the module `nag_lib_support`, which contains information about the interfaces to the procedures in it. The compiler can use this information to check calls to NAG *f*90 procedures at compile-time. This is a valuable aid: a great many programming mistakes in procedure-calls, such as an incorrect number of arguments, or arguments of the wrong type, which in Fortran 77 often resulted in obscure or unpredictable behaviour at run-time, can now be detected by the compiler.

For example, suppose you mistakenly thought that the procedure had an argument (say, the unit number on which the output was to appear), and wrote the program like this:

```
PROGRAM tutorial_ex1c !erroneous: mismatched argument
  USE nag_lib_support
  IMPLICIT NONE
  CALL nag_lib_ident(6)
END PROGRAM tutorial_ex1c
```

Then from the NAGWare *f*90 compiler you would get the error message:

```
Error: No specific match for reference to generic NAG_LIB_IDENT at line 4
[f90 error termination]
```

Essentially the compiler is complaining about a mismatch between the actual arguments in the calling program and the dummy arguments in the procedure in the Library. (The occurrence of the word ‘generic’ in this message may be a little puzzling or surprising; the reason is that all NAG *f*90 procedures are accessed through a ‘generic interface’.)

Another common mistake is to omit the `USE` statement, so that the program becomes:

```
PROGRAM tutorial_ex1d !erroneous: no USE statement
  IMPLICIT NONE
  CALL nag_lib_ident
END PROGRAM tutorial_ex1d
```

This program is compiled without error, but it cannot be linked correctly to the Library. On a typical UNIX system you would get an error message like this:

```
Error: Undefined:
nag_lib_ident_
```

(The reason for this is that there is no external reference `nag_lib_ident_` in the compiled Library; the module `nag_lib_support` must be accessed by the compiler in order to map the name `nag_lib_ident` onto the correct external reference.)

## 2 Example 2: Evaluating a Special Function

### 2.1 Arguments of Real Type

For our next example, we take another simple task, but one that does involve some numerical computation, namely to evaluate the function  $\operatorname{arcsinh} x$  or  $\operatorname{arccosh} x$  for given values of  $x$ . We consider  $\operatorname{arcsinh} x$  first.

As ‘ $\operatorname{arcsinh}$ ’ is a special function, you can see from the List of Contents that Chapter 3 is the one that deals with the special functions. You will then find that ‘ $\operatorname{arcsinh}$ ’ is a procedure within the module `nag_inv_hyp_fun` (3.1).

The list of contents on the front page of the module document shows that there is a procedure `nag_arcsinh` for evaluating  $\operatorname{arcsinh} x$ . The specification for that procedure states that it is a *function* with a single argument. The **Usage** section reads:

```
USE nag_inv_hyp_fun
```

```
[value =] nag_arcsinh(x)
```

and states that ‘the function result is a scalar, of type real(kind=wp)’. The **Arguments** section gives the specification of the argument  $x$ :

$x$  — real(kind=wp), intent(in)

*Input:* the argument  $x$  of the function.

Before going any further, we need to explain the type specification ‘real(kind=wp)’. For the time being, most readers can take it as equivalent to ‘DOUBLE PRECISION’. If you are running on a Cray, or perhaps on one or two other systems, you can take it as equivalent to REAL (that is, single precision). It depends on the *precision* of the implementation of the Library which you are using.

The information displayed by `nag_lib_ident` states the precision or precisions in which the Library is implemented. On many systems, only one precision is likely to be available — normally that which corresponds to 64-bit floating-point data — but the design of the Library allows for two or more precisions to be available in a single compiled library.

If you have used the NAG Fortran 77 Library, you may remember the use of the *bold italicised term ‘real’* in the documentation to denote either DOUBLE PRECISION or REAL according to the precision of the implementation of the Library. In NAG f90 documentation, ‘real(kind=wp)’ serves the same purpose.

Interpreting ‘real(kind=wp)’ as ‘DOUBLE PRECISION’, you could compile and run the following program to evaluate  $\operatorname{arcsinh} x$ :

```
PROGRAM tutorial_ex2a
  USE nag_inv_hyp_fun, ONLY : nag_arcsinh
  IMPLICIT NONE
  DOUBLE PRECISION :: x, y
  DO
    WRITE (*,*) 'Enter x'
    READ (*,*,end=10) x
    y = nag_arcsinh(x)
    WRITE (*,*) 'arcsinh(x) = ', y
  END DO
10  CONTINUE
END PROGRAM tutorial_ex2a
```

If you are interpreting ‘real(kind=wp)’ as ‘REAL’, you need only to replace DOUBLE PRECISION by REAL in the declaration of  $x$  and  $y$ . Note that there is no type declaration for `nag_arcsinh` in the main program: that information is obtained from the module `nag_inv_hyp_fun`.

If you use a precision which does not match the precision of the Library, you should get an error message at compile time — for example, from the NAGWare f90 compiler:

Error: No specific match for reference to generic NAG\_ARCSINH at line ...

The program asks for a value of  $x$  to be entered, and displays the value of  $\operatorname{arcsinh} x$ ; it repeats this in a loop which it exits only when end-of-file is detected by the READ statement. If you run the program and enter some values for  $x$ , you should see something like this on your screen:

```
Enter x
2.0
arcsinh(x) =    1.4436354751788103
Enter x
-5.0
arcsinh(x) =   -2.3124383412727525
Enter x
```

The computed values which you obtain for  $\operatorname{arcsinh} x$  may not be identical to those printed here because the program prints them out to their full precision; they may therefore be affected by differences in numerical behaviour between one system and another.

## 2.2 Precision and Kind Values

Now we will explain ‘real(kind=wp)’ properly.

Fortran 90 has inherited the traditional Fortran 77 concepts of single and double precision for real data, but has introduced a more powerful and flexible concept of a *parameterized* real data type. The parameter is called a *kind value*, and each kind value corresponds to a different ‘kind’ of real data. The traditional single precision real is one kind (the default), and the traditional double precision is another kind. The language allows for the possibility of other kinds (which might correspond to triple or quadruple precision, say), but compilers are not obliged to support them.

This feature of the language makes it much easier in principle to convert a program from one ‘kind’ of real data type to another: or in traditional language, from single to double precision, or conversely.

The actual kind values are not specified by the language and may vary from one compiler to another: for example, one compiler may use 1 for single precision and 2 for double; another compiler may use 4 for single (4-byte words) and 8 for double (8-byte words).

The type definition ‘real(kind=wp)’ that is used in NAG *f90* documentation conforms to the syntax of the language. The kind value is given symbolically as *wp* (standing for *working precision* or, if you prefer, *whatever precision*); *wp* is set in a different typeface to remind you that the actual kind value may vary from one implementation of the Library to another.

It is a sensible practice not to ‘hard-wire’ numeric kind values into your program, otherwise you may find you have to make a lot of changes to your program, either when converting from one precision to another, or when switching between different compilers or machines. Suitable programming practices are recommended in the next section.

All that has been said about the precision of real data applies also to the precision of complex data; complex data will be introduced in Section 4.1.

## 2.3 Portable Programming for Real Data

In order to make our programs easy to port from one system, compiler or precision to another, we will code everything to do with real data in terms of a kind value given by a named constant *wp*. (Of course you can choose any name you like for your kind value, but it is advisable not to make it much longer because it will appear very often.) Thus:

- all declarations of real variables or arrays will use the form `REAL(wp)`, which is a shortened form of `REAL(KIND=wp)`;
- all real constants will have the suffix `_wp` appended to them, which is how the kind value of a real constant is defined — for example, `1.0_wp`;
- the intrinsic function `REAL` will always be called with the optional argument `kind` present — for example, `REAL(i,kind=wp)`

In addition, the value of *wp* must be defined, preferably in one place only, so that it can easily be changed.

For the remaining programs in this Tutorial, we will assume that *wp* is defined in a separate module `tutorial_kind`. Here is one way to code the module:

```
MODULE tutorial_kind ! double precision version for NAGWare f90 compiler
  INTEGER, PARAMETER :: wp = 2
END MODULE tutorial_kind
```

Alternatively, you can write `wp = KIND(0.0D0)`, which sets *wp* to the kind value of the double precision constant `0.0D0`; that is, it sets *wp* to the kind value for double precision, be it 2, 8 or whatever.

Yet again, you can write `wp = SELECTED_REAL_KIND(10)`, which sets *wp* to the kind value which gives the equivalent of at least 10 decimal digits of precision; in practice this means double precision on most modern machines, but single precision on a Cray and one or two others.

Each program in this Tutorial will include a `USE` statement to access the value of *wp* from the module `tutorial_kind`:

```
USE tutorial_kind, ONLY : wp
```

The module `tutorial_kind` will not be reproduced at the start of each program. Fortran 90 compiling systems should make it easy for programs to access pre-compiled modules. Therefore the recommended procedure for using the module `tutorial_kind` is to set the value of `wp` in this module to the correct value for your implementation of the Library, to compile the module on its own, and then to make sure that the compiler can access this module when compiling any other programs from this Tutorial.

Here is program `tutorial_ex2a` rewritten in this style.

```
PROGRAM tutorial_ex2b
  USE nag_inv_hyp_fun, ONLY : nag_arcsinh
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp) :: x, y
  DO
    WRITE (*,*) 'Enter x'
    READ (*,*,end=10) x
    y = nag_arcsinh(x)
    WRITE (*,*) 'arcsinh(x) = ', y
  END DO
10  CONTINUE
END PROGRAM tutorial_ex2b
```

## 2.4 Arguments of `intent(in)` and Passing Constants as Arguments

The specification of the argument `x` of `nag_arcsinh` begins:

`x` — `real(kind=wp), intent(in)`

We have explained the type specification '`real(kind=wp)`'; now we consider the *intent* specification '`intent(in)`'. It means that the argument does not get redefined during the execution of the procedure (nor does it become undefined); in other words, its status on return from the procedure is exactly the same as it was immediately before the procedure call.

All arguments to NAG *f90* procedures (except for a few which are pointers) have a specified intent; it may be '`intent(in)`', '`intent(inout)`' or '`intent(out)`'. Here we discuss '`intent(in)`'.

If an argument has `intent(in)`, it is legitimate to supply a *constant* as the actual argument, but not if the argument has `intent(out)` or `intent(inout)`.

Note that a constant must have the correct type and kind value to match the dummy argument; it is easy to make a mistake. For example, suppose you want to compute `arcsinh 2`:

`nag_arcsinh(2)` would be *wrong*, because the supplied argument is an integer, instead of real;

`nag_arcsinh(2.0)` would be *wrong* unless you were using a single precision implementation of the Library, because 2.0 has the default single precision real type; for a double precision implementation, you must supply a double precision constant;

`nag_arcsinh(2.0_wp)` is the *correct* way to program the expression in the portable style of Section 2.3.

In the first two cases, the error would be detected at compile time. The NAGWare *f90* compiler would give the same message as we have seen before:

```
Error: No specific match for reference to generic NAG_ARCSINH at line ...
```

It is also legitimate to supply an *expression* as the actual argument if the dummy argument has `intent(in)`, but not if it has `intent(inout)` or `intent(out)`. So, for example, you can write `nag_arcsinh(SINH(x))` and see how close the result is to `x`.

Arguments of `intent(inout)` are discussed in Section 5.2.

## 2.5 A First Look at Run-time Errors

The procedure `nag_arcsinh` is unusual in that it has no error exits: any real value of  $x$  that can be represented internally in the computer can be supplied as an argument, and the procedure will return a good approximation to the exact result.

But suppose we want a program to compute  $\operatorname{arccosh} x$ , rather than  $\operatorname{arcsinh} x$ . By similar steps, we might arrive at the following very similar program:

```

PROGRAM tutorial_ex2c
  USE nag_inv_hyp_fun, ONLY : nag_arccosh
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp) :: x, y
  DO
    WRITE (*,*) 'Enter x'
    READ (*,*,end=10) x
    y = nag_arccosh(x)
    WRITE (*,*) 'arccosh(x) = ', y
  END DO
10  CONTINUE
END PROGRAM tutorial_ex2c

```

If you run this program, entering the same values of  $x$  as in Section 2.1, you should see something like this:

```

Enter x
2
arccosh(x) =    1.3169578969248166
Enter x
-5
***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_arccosh                      Level = 3   Code = 301
An input argument has an invalid value.
x = -5.0000000000000000E+000
x must be >= 1.0.
***** Execution halted *****

```

The function  $\operatorname{arccosh} x$  is not defined when  $x < 1$ , so  $-5$  is an invalid value of its argument. In the documentation of `nag_arccosh`, the specification of  $x$  contains the text:

*Constraints:*  $x \geq 1.0$ .

Furthermore, the **Error Codes** section lists a ‘**Fatal error**’ with code 301 and description ‘An input argument has an invalid value’. This is indeed the case. The input value of  $x$  violated the stated constraint; the procedure detected this, output an informative error message, and halted the program.

This is a cue to start explaining how run-time errors are handled by the Library.

They are classified into three levels of increasing severity.

**Level 1 (Warning):** a warning that, although the computation has been completed, the results may not be completely satisfactory;

**Level 2 (Failure):** a numerical failure during computation (for example, failure of an iterative algorithm to converge);

**Level 3 (Fatal):** a fatal error which prevents the procedure from attempting any computation (for example, invalid arguments, or failure to allocate enough memory).

You may have noticed in the documentation of `nag_arccosh` that it has an *optional argument error*. If an argument is optional, you do not need to supply a corresponding actual argument in the procedure call; the documentation specifies what happens if the argument is not supplied. The specification of **error** is standard text which appears in the documentation of almost every NAG *f*l90 procedure. It reads:



**error** — type(nag\_error), intent(inout), optional

The NAG *f*l90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied ...

We shall leave it until Sections 5.4 and 7.2 to explain what happens if you do supply the argument **error**. Until then it will be omitted, as it was in the reference to `nag_arccosh` in the program `tutorial_ex2c`. The procedure then behaves according to the following rules:

- if it detects an error of level 1 (warning), it writes an error message to the standard output unit and returns control to your calling program;
- if it detects an error of level 2 or 3 (failure in computation or fatal error), it writes an error message to the standard error-message unit and halts execution of the program.

The error in `nag_arccosh` is classified as a fatal error (as was stated in the documentation), so the procedure did indeed output an error message and halt the program.

If you wish to allow your program to cope with arbitrary values of  $x$ , you are recommended to insert a test before `nag_arccosh` is called. This is simpler than modifying the error-handling mechanism so that the program does not halt after a fatal error. For example:

```

READ (*,*,end=10) x
IF (x<1.0_wp) THEN
  WRITE (*,*) 'arccosh(x) is undefined'
ELSE
  y = nag_arccosh(x)
  WRITE (*,*) 'arccosh(x) = ', y
END IF

```

### 3 Example 3: Summary Statistics of Univariate Data

#### 3.1 A First Look at Array Arguments

The problem to be tackled in this example is to compute basic descriptive statistics (mean, standard deviation, skewness and so on) for a sample of observations of a single variable. The appropriate NAG *f*l90 procedure is called `nag_summary_stats_1v` and it is in the module `nag_basic_stats` (22.1).

The procedure has only one *mandatory* (that is, non-optional) argument **x**, and several optional arguments. The specification of **x** reads:

**x**( $m$ ) — real(kind=wp), intent(in)

*Input:* the data values,  $x_i$ ,  $i = 1, \dots, m$ .

The notation **x**( $m$ ) is a convention used in the argument specifications in NAG *f*l90 documentation to denote that **x** is an *array* which must have exactly  $m$  elements. Here  $m$  is the number of values of  $x$  in the sample.

All array arguments to NAG *f*l90 procedures are *assumed-shape* arrays. This technical term means that the procedure does not require a separate argument **m**, say, to specify how many elements in **x** contain the data to be analysed; but it does require that the actual array supplied in the procedure call must have *exactly*  $m$  elements: the procedure then determines the value of  $m$  from the ‘assumed shape’ of **x**. There is a statement to this effect at the start of the **Arguments** section of the documentation for the procedure:

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$m > 1$  — the number of data points.

You might think that the requirement to pass an array argument with exactly the right number of elements would be unduly restrictive. If you knew in advance that you wanted to deal with several samples of data, all of the same size (100, say), then you could declare an array of this size, and pass the name of the array as the first argument to `nag_summary_stats_1v`:

```
REAL (wp) :: x(100)
. . .
CALL nag_summary_stats_1v(x,...)
```

However, usually we want programs to be more flexible and to handle samples of data of any size. We will show two ways in which this can be done, using a very simple call of `nag_summary_stats_1v`, which just computes the mean of the sample, namely:

```
CALL nag_summary_stats_1v(x,mean=xbar)
```

(We will explain ‘`mean=xbar`’ in Section 3.4.)

We will assume that the data is read from a file, in which the first record contains the value of  $m$  (the number of  $x$ -values in the sample), and the remaining records contain the  $x$ -values themselves. To run the programs we will use this data file:

```
24
193 215 112 161 92 140 38 33 279 249 67 61
473 339 60 130 20 50 257 284 447 52 150 220
```

## 3.2 Passing Array Sections as Arguments

Here is the first program:

```
PROGRAM tutorial_ex3a
  USE nag_basic_stats, ONLY : nag_summary_stats_1v
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER, PARAMETER :: m_max = 100
  INTEGER :: m
  REAL (wp) :: xbar
  REAL (wp) :: x(m_max)
  OPEN (4,file='tutorial_ex3a.dat')
  READ (4,*) m                                ! read number of x-values
  IF (m>m_max) THEN                          ! check it
    WRITE (*,*) 'too many x-values'
  ELSE
    READ (4,*) x(1:m)                        ! read x-values
    CALL nag_summary_stats_1v(x(1:m),mean=xbar) ! compute mean
    WRITE (*,'(1x,a,f10.2)') 'mean =', xbar    ! print mean
  END IF
END PROGRAM tutorial_ex3a
```

This program declares the array `x` with a fixed size which is hopefully larger than any expected sample. It then uses a *section* of this array, `x(1:m)`, to store the  $x$ -values that are read in, and passes the same section `x(1:m)` as the actual argument to `nag_summary_stats_1v`.

## 3.3 Using Allocatable Arrays

Here is the second program:

```
PROGRAM tutorial_ex3b
  USE nag_basic_stats, ONLY : nag_summary_stats_1v
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: m
  REAL (wp), ALLOCATABLE :: x(:)
  REAL (wp) :: xbar
  OPEN (4,file='tutorial_ex3a.dat')
  READ (4,*) m                                ! read number of x-values
```

```

      ALLOCATE (x(m))                ! allocate storage for x-values
      READ (4,*) x                   ! read x-values
      CALL nag_summary_stats_1v(x,mean=xbar) ! compute mean
      WRITE (*,'(1x,a,f10.2)') 'mean =', xbar ! print mean
END PROGRAM tutorial_ex3b

```

This program declares `x` to be an *allocatable* array. In the declaration its shape is given as '`x(:)`', which means that it is a rank-1 array (that is, one-dimensional), but its size is to be determined later; that happens when the `ALLOCATE` statement is executed. The `ALLOCATE` statement allocates exactly `m` elements to `x`, so that when `x` is passed as an argument to `nag_summary_stats_1v`, it has the required shape.

Both these programs give the same results when executed using the example data file, namely:

```
mean =      171.75
```

The second style, using an allocatable array, is used in many of the example programs in the module documents, and will be used in other programs in this Tutorial, but there are advantages and disadvantages with either method, and this is not the place to argue between them. The point is that the Library can be used just as easily with both.

If you use allocatable arrays, you may need to *deallocate* them when they are no longer needed. In particular, it is illegal to allocate an array that has already been allocated without deallocating it first. For example, suppose you wish to extend the program `tutorial_ex3b` so that it would read any number of sets of data of different sizes from the same file (until end-of-file is reached): then you could replace the executable statements with the following:

```

      DO
        READ (4,*,end=10) m          ! read number of x-values
        ALLOCATE (x(m))              ! allocate storage for x-values
        READ (4,*) x                 ! read x-values
        CALL nag_summary_stats_1v(x,mean=xbar) ! compute mean
        WRITE (*,'(1x,a,f10.2)') 'mean =', xbar ! print mean
        DEALLOCATE (x)              ! deallocate storage
      END DO
10    CONTINUE

```

If you did not include the `DEALLOCATE` statement, you would get a run-time error with a message like the following:

```

ALLOCATABLE array X has already been ALLOCATED
Program terminated by fatal error

```

### 3.4 Optional Arguments and Argument Keywords

Now we turn to the *optional arguments* of `nag_summary_stats_1v`. The procedure may seem a little unusual in that *all* its output arguments are optional: there are 11 of them, and a user might reasonably want to compute any combination of them. The only mandatory argument is the input array `x`, so that a procedure call using mandatory arguments only, namely:

```
CALL nag_summary_stats_1v(x)
```

would do nothing at all. In fact the documentation states: 'at least one output argument must be present in every call', and a call with no output arguments present results in a fatal error, with the error message:

```

***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_summary_stats_1v          Level = 3   Code = 305
Invalid absence of an optional argument.
At least one optional output argument must be present.
***** Execution halted *****

```

The procedure `nag_summary_stats_1v` has two optional input arguments and 11 optional output arguments. We consider the output arguments first. Here is the specification of one of them:

**mean** — real(kind=wp), intent(out), optional

*Output:* the mean.

We have already used the following **CALL** statement to compute the mean:

```
CALL nag_summary_stats_1v(x,mean=xbar)
```

This illustrates an alternative means of matching dummy arguments with actual arguments, namely the use of *argument keywords*. In this example, **mean** is the dummy argument name, and **xbar** is the name of the actual argument (a variable in the calling program).

Here are some more examples of possible calls:

```
REAL (wp) :: range, xbar, xsd, xvar
. . .
CALL nag_summary_stats_1v(x,mean=xbar,std_dev=xsd)
CALL nag_summary_stats_1v(x,range=range,mean=xbar,variance=xvar)
```

The specification **range=range** may seem odd at first sight, but it is simply a consequence of choosing the same name for the actual argument as for the dummy argument: if a name is good for one, it is quite likely to be good for the other. You may like it better if you put the argument keywords into upper case:

```
CALL nag_summary_stats_1v(x,RANGE=range,MEAN=xbar,VARIANCE=xvar)
```

Fortran 90 allows actual arguments to be matched with dummy arguments (whether optional or not) either by position (the traditional Fortran 77 mechanism) or by keyword (as just illustrated). But after one keyword has appeared in the list of actual arguments, all subsequent actual arguments must be matched by keyword. These are the rules of the Fortran 90 language.

However, *when you call a NAG fl90 procedure*, we require you to conform to a more rigid convention, namely that *all optional arguments must be supplied by keyword*. A reminder is stated at the head of the **Optional Arguments** section in each procedure specification. The order in which optional arguments are listed in this section is not necessarily the order in which they occur in the argument list in the software. Additional optional arguments may be added at future releases of the Library.

In the argument lists of NAG fl90 procedures, all mandatory (that is, non-optional) arguments appear *before* any optional arguments, so they can always be matched by position. You can use keywords for the mandatory arguments if you wish, but in the programs in this Tutorial, as in the example programs in the module documents, we shall stick to the convention that keywords will be used for optional arguments only. This will help to remind you which arguments are optional and which are not.

Finally, we consider one of the optional input arguments of **nag\_summary\_stats\_1v**. Its specification reads:

**freq(m)** — integer, intent(in), optional

*Input:* the frequencies  $f_i$  associated with the data values  $x_i$ , for  $i = 1, \dots, m$ .

*Default:* **freq** = 1.

*Constraints:* **freq** > 0.

When an input argument is optional, the documentation usually states a default value which is assumed if the argument is not present. (Occasionally, a procedure takes some default action which is more complicated than simply assuming a default value for the argument.) In this case, the default is that the frequencies are assumed to be 1.

Note that the notation **freq(m)** specifies that **freq** must have  $m$  elements, the same as **x**. If it does not, you will get a fatal error message like this:

```
***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_summary_stats_1v          Level = 3   Code = 303
Array arguments have inconsistent shapes.
SIZE(freq) =          10  SIZE(x) =          24
SIZE(freq) must equal SIZE(x).
***** Execution halted *****
```

The following program is a more elaborate version of the program `tutorial_ex3b`, which expects frequencies to be read from the data file as well, and prints mean, standard deviation, skewness and kurtosis:

```
PROGRAM tutorial_ex3c
  USE nag_basic_stats, ONLY : nag_summary_stats_1v
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: m
  REAL (wp), ALLOCATABLE :: x(:)
  INTEGER, ALLOCATABLE :: freq(:)
  REAL (wp) :: xbar, std_dev, skewness, kurtosis
  OPEN (4,file='tutorial_ex3c.dat')
  READ (4,*) m                                ! read number of x-values
  ALLOCATE (x(m),freq(m))                    ! allocate storage
  READ (4,*) x                                ! read x-values
  READ (4,*) freq                             ! read frequencies
  CALL nag_summary_stats_1v(x,freq=freq,mean=xbar,std_dev=std_dev, &
    skewness=skewness,kurtosis=kurtosis)
  WRITE (*,'(1x,a,f10.2)') 'mean              =', xbar
  WRITE (*,'(1x,a,f10.2)') 'standard deviation =', std_dev
  WRITE (*,'(1x,a,f10.2)') 'skewness          =', skewness
  WRITE (*,'(1x,a,f10.2)') 'kurtosis          =', kurtosis
END PROGRAM tutorial_ex3c
```

With this data file:

```
24
193 215 112 161 92 140 38 33 279 249 67 61
473 339 60 130 20 50 257 284 447 52 150 220
  1  2  1  1  3  1  2  1  2  3  2  1
  1  1  1  2  1  1  3  2  1  1  2  3
```

the results were:

```
mean              =    177.46
standard deviation =    111.74
skewness          =     0.62
kurtosis          =     0.02
```

## 4 Example 4: Eigenvalues and Eigenvectors

### 4.1 Generic Procedures for Different Data Types

Suppose we wish to compute the eigenvalues of a general real matrix. The NAG *f190* procedure for this is `nag_nsym_eig_all` in the module `nag_nsym_eig` (6.2).

The **Description** for this procedure begins:

`nag_nsym_eig_all` is a *generic* procedure which computes all the eigenvalues, and optionally all the left or right eigenvectors, of a *real or complex* general matrix  $A$  of order  $n$ .

Note the words which have been italicized in this extract (but not in the document itself). Whether you wish to compute the eigenvalues of a real matrix or of a complex matrix, you call the same procedure `nag_nsym_eig_all`, or at least that is the way it appears and the way it is documented. (Strictly speaking, what the Library provides is a *generic interface* to two specific procedures, one for real matrices and one for complex matrices.)

In the documentation of the procedure, the specification for the argument `a` reads:

$\mathbf{a}(n,n)$  — real(kind=wp) / complex(kind=wp), intent(inout)

*Input:* the general matrix  $A$ .

*Output:* overwritten by intermediate results.

The type specification 'real(kind=wp) / complex(kind=wp)' indicates that **a** may be of either type; the compiler will determine from the type of the actual argument which of the specific procedures (for real or complex matrices) should be called.

Since the eigenvalues and eigenvectors of a real matrix are in general complex, the arguments **lambda**, **vr** and **vl** of this procedure are always of type `complex(kind=wp)` whatever the type of **a**.

Note that in Fortran 90, the intrinsic type `COMPLEX` can be parameterized with a kind value in just the same way as the `REAL` type, and all standard-conforming compilers must support both a double precision and a single precision complex type.

The first program of this section computes the eigenvalues of a real matrix:

```
PROGRAM tutorial_ex4a
  USE nag_nsym_eig, ONLY : nag_nsym_eig_all
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  REAL (wp), ALLOCATABLE :: a(:, :)
  COMPLEX (wp), ALLOCATABLE :: lambda(:)
  OPEN (4, file='tutorial_ex4a.dat')
  READ (4, *) n
  ALLOCATE (a(n, n), lambda(n))
  READ (4, *) (a(i, 1:n), i=1, n)
  CALL nag_nsym_eig_all(a, lambda)
  WRITE (*, *) 'Eigenvalues'
  WRITE (*, '(1x, "(,f8.4, ", ", f8.4, ")")') lambda
END PROGRAM tutorial_ex4a
```

If we supply the following data file `tutorial_ex4a.dat`

```
4
3.5  4.5 -1.4 -1.7
0.9  0.7 -5.4  3.5
-4.4 -3.3 -0.3  1.7
2.5 -3.2 -1.3  1.1
```

we get the results:

```
Eigenvalues
(  7.9948,  0.0000)
(-0.9941,  4.0079)
(-0.9941, -4.0079)
(-1.0066,  0.0000)
```

Now suppose that we wish to compute the eigenvalues of a complex matrix. All that needs to be changed in the program is the type declaration for the array **a**. Here is the program:

```
PROGRAM tutorial_ex4b
  USE nag_nsym_eig, ONLY : nag_nsym_eig_all
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  COMPLEX (wp), ALLOCATABLE :: a(:, :)
  COMPLEX (wp), ALLOCATABLE :: lambda(:)
  OPEN (4, file='tutorial_ex4b.dat')
  READ (4, *) n
  ALLOCATE (a(n, n), lambda(n))
  READ (4, *) (a(i, 1:n), i=1, n)
  CALL nag_nsym_eig_all(a, lambda)
  WRITE (*, *) 'Eigenvalues'
  WRITE (*, '(1x, "(,f8.4, ", ", f8.4, ")")') lambda
END PROGRAM tutorial_ex4b
```

Here is the data file `tutorial_ex4b.dat`:

```
4
(-3.97,-5.04) (-4.11, 3.70) (-0.34, 1.01) ( 1.29,-0.86)
( 0.34,-1.50) ( 1.52,-0.43) ( 1.88,-5.38) ( 3.36, 0.65)
( 3.31,-3.85) ( 2.50, 3.45) ( 0.88,-1.08) ( 0.64,-1.48)
(-1.10, 0.82) ( 1.81,-1.59) ( 3.25, 1.33) ( 1.57,-3.44)
```

And here are the results:

```
Eigenvalues
( -6.0004, -6.9998)
( -5.0000,  2.0060)
(  7.9982, -0.9964)
(  3.0023, -3.9998)
```

All the procedures in Chapters 5 and 6 (the linear algebra chapters) of NAG *f90* are generic for real and complex data. So also are other procedures, such as those for printing matrices, which are used in the next section.

## 4.2 Reading and Writing Two-dimensional Arrays

The procedure `nag_nsym_eig_all` has optional output arguments `vr` and `vl` (which, like `a`, are two-dimensional arrays) for returning the right and/or the left eigenvectors. (Right eigenvectors are the usual eigenvectors which satisfy  $Ax = \lambda x$ ; left eigenvectors are the usual eigenvectors of  $A^H$ , the conjugate transpose of  $A$ .)

Before giving an example of the computation of eigenvectors, we show how the procedure `nag_write_gen_mat` in the module `nag_write_mat` (1.3) can be used for printing a two-dimensional array. This also is a generic procedure that allows the array to be printed to be of integer, real or complex type.

The procedure `nag_write_gen_mat` can be called with just one argument, the array `a` to be printed, but it has several optional arguments for format-control and annotation. The following program illustrates the use of some of these options — for a title, and for integer labels on each row and column:

```
PROGRAM tutorial_ex4c
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  REAL (wp), ALLOCATABLE :: a(:, :)
  OPEN (4, file='tutorial_ex4a.dat')
  READ (4, *) n
  ALLOCATE (a(n,n))
  READ (4, *) (a(i,1:n), i=1,n)
  CALL nag_write_gen_mat(a, int_row_labels=.TRUE., int_col_labels=.TRUE., &
    title='matrix A')
END PROGRAM tutorial_ex4c
```

Using the same data file `tutorial_ex4a.dat` as before, we obtain the output:

```
matrix A
      1      2      3      4
1    3.5000  4.5000 -1.4000 -1.7000
2    0.9000  0.7000 -5.4000  3.5000
3   -4.4000 -3.3000 -0.3000  1.7000
4    2.5000 -3.2000 -1.3000  1.1000
```

Note the `READ` statement which was used to read in the array `a`:

```
READ (4, *) (a(i,1:n), i=1,n)
```

This reads data into the array *row by row*, as it appears in the data file. Beware of using the simpler statement:

```
READ (4, *) a
```

This would read data into the array *column by column*, so the first record of the data file (the first row of the matrix) would be read into the first column of the array, and so on: the effect would be to transpose the array.

Finally, we show how the program `tutorial_ex4a` can be extended to compute the right eigenvectors as well as the eigenvalues:

```
PROGRAM tutorial_ex4d
  USE nag_nsym_eig, ONLY : nag_nsym_eig_all
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  REAL (wp), ALLOCATABLE :: a(:, :)
  COMPLEX (wp), ALLOCATABLE :: lambda(:), v(:, :)
  OPEN (4, file='tutorial_ex4a.dat')
  READ (4, *) n
  ALLOCATE (a(n,n), lambda(n), v(n,n))
  READ (4, *) (a(i, 1:n), i=1, n)
  CALL nag_nsym_eig_all(a, lambda, vr=v)
  WRITE (*, *) 'Eigenvalues'
  WRITE (*, '(1x,i5,"  ",f8.4," ",f8.4,"")') (i, lambda(i), i=1, n)
  WRITE (*, *)
  CALL nag_write_gen_mat(v, title='Eigenvectors', int_col_labels=.TRUE.)
END PROGRAM tutorial_ex4d
```

Using the data file `tutorial_ex4a.dat` once more, we obtain the results:

#### Eigenvalues

```
1 ( 7.9948, 0.0000)
2 (-0.9941, 4.0079)
3 (-0.9941, -4.0079)
4 (-1.0066, 0.0000)
```

#### Eigenvectors

```

          1          2          3
( 0.6551, 0.0000) ( -0.1933, 0.2546) ( -0.1933, -0.2546)
( 0.5236, 0.0000) ( 0.2519, -0.5224) ( 0.2519, 0.5224)
( -0.5362, 0.0000) ( 0.0972, -0.3084) ( 0.0972, 0.3084)
( 0.0956, 0.0000) ( 0.6760, 0.0000) ( 0.6760, 0.0000)

          4
( 0.1253, 0.0000)
( 0.3320, 0.0000)
( 0.5938, 0.0000)
( 0.7221, 0.0000)
```

The eigenvectors are stored in the columns of the array `v`, which are numbered by the procedure `nag_write_gen_mat`.

## 5 Example 5: Solving Systems of Linear Equations

### 5.1 More on Genericity

Chapter 5 of NAG *f90* is concerned with solving systems of linear equations, for which the conventional mathematical notation is

$$Ax = b. \tag{1}$$

The chapter contains five modules:

`nag_gen_lin_sys` (5.1) for systems with a general matrix  $A$ ;

`nag_sym_lin_sys` (5.2) for systems with a symmetric matrix  $A$ ;



`nag_tri_lin_sys` (5.3) for systems with a triangular matrix  $A$ .

`nag_gen_bnd_lin_sys` (5.4) for systems with a general banded matrix  $A$ ;

`nag_sym_bnd_lin_sys` (5.5) for systems with a symmetric banded matrix  $A$ ;

In this Tutorial we will illustrate the use of the procedure `nag_gen_lin_sol` from the module `nag_gen_lin_sys`, but if you have a system with a symmetric, triangular or banded matrix, it is preferable to use a procedure from one of the other modules, which will be more efficient and possibly more reliable.

The procedure `nag_gen_lin_sol` is a generic procedure which can handle either real or complex systems, like the procedure `nag_nsym_eig_all` which was discussed in Section 4. It is also generic in another respect.

Users often wish to solve several systems with the same matrix  $A$ , but with different right-hand sides  $b_i$  and different solutions  $x_i$ :

$$Ax_i = b_i \quad \text{for } i = 1, \dots, r,$$

which can be rewritten in matrix notation

$$AX = B, \tag{2}$$

the columns of  $B$  being the right-hand side vectors  $b_i$ , and the columns of  $X$  being the solution vectors  $x_i$ .

For solving systems of the form (1), it is convenient to store the right-hand side vector  $b$  in a one-dimensional array (an array of rank 1 in Fortran 90 terminology); for solving systems of the form (2), it is convenient to store the matrix  $B$  in a two-dimensional array (an array of rank 2). The procedure `nag_gen_lin_sol` allows for both these possibilities through its generic interface.

Note that if a dummy argument is an assumed-shape array, the actual argument which matches it must have the same rank: an array of rank 1 — declared, for example, as `x(n)` — is *not* equivalent to an array of rank 2 with one of its dimensions equal to 1 — declared as `x(1,n)` or `x(n,1)`. However, NAG *f90* avoids any inconvenience which this rule of the language might cause, by providing a generic interface and thus allowing either a rank-1 or a rank-2 array to be supplied as the actual argument.

For procedures with a considerable degree of genericity, the **Usage** section of the procedure documentation contains a section on **Interfaces**, which for `nag_gen_lin_sol` reads as follows.

Distinct interfaces are provided for each of the four combinations of the following cases:

Real / complex data

**Real data:**  $\mathbf{a}$  and  $\mathbf{b}$  are of type `real(kind=wp)`.

**Complex data:**  $\mathbf{a}$  and  $\mathbf{b}$  are of type `complex(kind=wp)`.

One / many right-hand sides

**One r.h.s.:**  $\mathbf{b}$  is a rank-1 array, and the optional arguments `bwd_err` and `fwd_err` are scalars.

**Many r.h.s.:**  $\mathbf{b}$  is a rank-2 array, and the optional arguments `bwd_err` and `fwd_err` are rank-1 arrays.

Thus the generic interface for `nag_gen_lin_sol` provides access to *four* specific procedures, all covered by a single procedure specification.

The specification of the argument  $\mathbf{b}$  takes the form:

**b(n) / b(n,r)** — real(kind=wp) / complex(kind=wp), intent(inout)

*Input:* the right-hand side vector *b* or matrix *B*.

*Output:* overwritten on exit by the solution vector *x* or matrix *X*.

*Constraints:* **b** must be of the same type as **a**.

*Note:* if optional error bounds are requested then the solution returned is that computed by iterative refinement.

Note that the stated constraint forbids arbitrary combinations of arguments: you cannot call the procedure with real **a** and complex **b**. If you try to, you will get an error message at compile time; from the NAGWare f90 compiler it will be:

Error: No specific match for reference to generic NAG\_GEN\_LIN\_SOL at line ...

Here is a simple program to solve a real system of linear equations with a single right-hand side:

```
PROGRAM tutorial_ex5a
  USE nag_gen_lin_sys, ONLY : nag_gen_lin_sol
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  REAL (wp), ALLOCATABLE :: a(:, :), b(:)
  OPEN (4,file='tutorial_ex5a.dat')
  READ (4,*) n
  ALLOCATE (a(n,n),b(n))
  READ (4,*) (a(i,1:n),i=1,n)
  READ (4,*) b
  CALL nag_gen_lin_sol(a,b)
  WRITE (*,*) 'Solution'
  WRITE (*,'(1x,f8.4)') b
END PROGRAM tutorial_ex5a
```

Here is the data file `tutorial_ex5a.dat`:

```
4
1.80  2.88  2.05 -0.89
5.25 -2.95 -0.95 -3.80
1.58 -2.69 -2.90 -1.04
-1.11 -0.66 -0.59  0.80
9.52
24.35
0.77
-6.22
```

And here are the results:

```
Solution
 1.0000
-1.0000
 3.0000
-5.0000
```

## 5.2 Arguments of intent(inout)

Arguments **a** and **b** of `nag_gen_lin_sol` have the intent specification 'inout': they are intended to pass data into the procedure, and to pass results back to the calling program.

The actual argument which matches a dummy argument of intent(inout) must be *definable*; it may not be a constant or an expression.

In Fortran 77, passing a constant as an actual argument to a subroutine which redefines it has been a notorious cause of puzzling errors. The `INTENT` attribute in Fortran 90 provides protection against this (although some compilers may not check rigorously that arguments are used in conformity with their intents).

In NAG *f*90, arguments of `intent(inout)` are either arrays or structures, which to some extent reduces the risk of supplying a constant or expression. But Fortran 90 allows array constants or array expressions.

Suppose, for example, that you wish to solve the system of equations

$$|A|x = b,$$

where  $|A|$  denotes the matrix with elements  $|a_{ij}|$ . You might be tempted to replace the call to `nag_gen_lin_sol` by

```
CALL nag_gen_lin_sol(ABS(a),b)
```

The NAGWare compiler detects this as an error at compile time, giving the message:

```
Error: No specific match for reference to generic NAG_GEN_LIN_SOL at line 12
```

because the actual argument `ABS(a)` cannot be matched with the dummy argument `a` that has `intent(inout)`.

Instead, you must assign `ABS(a)` to another array of the same shape, and then pass this as the argument:

```
aa = ABS(a)
CALL nag_gen_lin_sol(aa,b)
```

### 5.3 Sensitivity of Numerical Results

Although the main purpose of this Tutorial is to teach readers about aspects of the Fortran 90 language, a few remarks will be made here about the sensitivity of numerical results.

Here is a slightly extended version of the program `tutorial_ex5a`

```
PROGRAM tutorial_ex5b
  USE nag_gen_lin_sys, ONLY : nag_gen_lin_sol
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, n
  REAL (wp) :: e, rcond
  REAL (wp), ALLOCATABLE :: a(:,,:), b(:)
  OPEN (4,file='tutorial_ex5b.dat')
  READ (4,*) n
  ALLOCATE (a(n,n),b(n))
  READ (4,*) (a(i,1:n),i=1,n)
  READ (4,*) b
  CALL nag_gen_lin_sol(a,b,rcond=rcond,fwd_err=e)
  WRITE (*,*) 'Solution'
  WRITE (*,'(1x,f8.4)') b
  WRITE (*,'(1x,a,1p,e10.1)') 'Relative error =', e
  IF (rcond==0) THEN
    WRITE (*,*) 'A is singular'
  ELSE
    WRITE (*,'(1x,a,1p,e10.1)') 'Estimate of condition number =', &
      1/rcond
  END IF
END PROGRAM tutorial_ex5b
```

Here is the data file `tutorial_ex5b.dat` which differs in only one digit from the data file `tutorial_ex5a.dat`: the element  $a_{4,3}$  of  $A$  has been changed from  $-0.59$  to  $-0.58$ .

```
4
1.80  2.88  2.05 -0.89
5.25 -2.95 -0.95 -3.80
1.58 -2.69 -2.90 -1.04
-1.11 -0.66 -0.58  0.80
9.52
24.35
0.77
-6.22
```

And here are the results:

```
Solution
  0.8576
 -0.9494
  2.9562
 -5.2250
Relative error =   4.6E-14
Estimate of condition number =   1.4E+02
```

The solution is noticeably different from that obtained using the data-file `tutorial_ex5a`. Why?

The program `tutorial_ex5b` prints a bound on the relative error in the computed solution, as well as the solution itself. The printed value is of the order of  $10^{-14}$ , so we can expect an accuracy of around 14 digits in the solution. (If the components of the solution differ widely in magnitude, this figure applies to the largest component of the solution.) So in our example the computed solution is highly accurate — as a solution of the given system of equations.

But the system of equations is *not* the same as in the previous example!

The results include an estimate of the *condition number* of the matrix  $A$ . This is a measure of how sensitive the solution of the system of equations is to small perturbations in the data. A condition number of the order of 100 is not unusually sensitive, but it means that perturbations in the data ( $A$  and  $b$ ) may be amplified by a factor of as much as 100 in the solution  $x$ .

In our examples, the difference between the two data files involves a perturbation in  $A$  of the order of 1 part in 1,000 (relative to the largest element of  $A$ ). The difference between the two solutions is roughly 1 part in 25, so the perturbation has certainly been amplified, if not by the maximum amount indicated by the condition number.

The condition number also indicates how sensitive a computed solution is to rounding errors made during the computation. Rounding errors in solving systems of linear equations have an effect equivalent to small perturbations in the original data; the size of the perturbations is a modest multiple of the machine precision, which is the value returned by the Fortran 90 intrinsic function `EPSILON`. But some matrices are so *ill conditioned* — that is, they have such a large condition number — that the effect of rounding errors may swamp any accuracy in the computed solution. The procedure `nag_gen_lin_sol` detects such extremely ill conditioned matrices and issues a warning, as illustrated in the next example.

Error bounds and estimates of condition numbers are provided throughout the **Linear Equations** chapter of the Library; for a fuller explanation, see the Introduction to that chapter.

The next example program solves a sequence of systems of equations, of increasing order  $n$ , where the coefficient matrix  $A$  is a *Hilbert matrix* defined by  $a_{ij} = 1/(i+j-1)$ . Hilbert matrices are an often-used example for demonstrating the effects of ill conditioning: as  $n$  increases,  $A$  rapidly becomes more and more ill conditioned.

The following program uses a right-hand side vector  $b = (1, 1, \dots, 1)^T$ , but does not bother to print the solution. For each value of  $n$  it prints the condition number and the error bound on the computed solution.

```
PROGRAM tutorial_ex5c
  USE nag_gen_lin_sys, ONLY : nag_gen_lin_sol
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER, PARAMETER :: n_max = 12
  INTEGER :: i, j, n
  REAL (wp) :: rcond, err
  REAL (wp) :: a(n_max,n_max), b(n_max)
  WRITE (*,*) '  n    condition    forward'
  WRITE (*,*) '      number      error'
  DO n = 1, n_max
    DO j = 1, n
      DO i = 1, n
        a(i,j) = 1.0_wp/REAL(i+j-1,kind=wp)
      END DO
    END DO
    CALL nag_gen_lin_sol(a, b, rcond, err, n, wp)
    WRITE (*,*) n, rcond, err
  END DO
```

```

      END DO
      b(1:n) = 1.0_wp
      CALL nag_gen_lin_sol(a(1:n,1:n),b(1:n),rcond=rcond,fwd_err=err)
      WRITE (*,'(1x,i3,1p,3e12.1)') n, 1.0_wp/rcond, err
    END DO
  END PROGRAM tutorial_ex5c

```

It gives the following results on a system with double precision IEEE arithmetic:

n	condition number	forward error
1	1.0E+00	8.9E-16
2	2.7E+01	9.3E-15
3	7.5E+02	2.0E-13
4	2.8E+04	6.6E-12
5	9.4E+05	2.0E-10
6	2.9E+07	6.0E-09
7	9.9E+08	2.1E-07
8	3.4E+10	6.9E-06
9	1.1E+12	2.2E-04
10	3.5E+13	7.5E-03
11	1.2E+15	2.6E-01

```

***** Warning reported by NAG Fortran 90 Library *****
Procedure nag_gen_lin_sol                      Level = 1   Code = 101
The matrix of the coefficients is nearly singular.
Reciprocal of the condition number = 2.632742811818276E-017 .
The solution may have no accuracy at all.
Examine the estimate of the forward error that may be returned in fwd_err.
***** Execution continued *****
12      3.8E+16      7.8E+00

```

When  $n = 12$ , the matrix is so ill conditioned that the procedure has issued a warning. On systems with different arithmetic properties, the warning may first occur at a different value of  $n$ ; you may need to increase the value of `n_max` in the program.

If you examine the value of the forward error, you will see that for  $n = 12$  it is greater than 1; in other words, the size of the error is larger than the size of the solution, and so the solution has no accuracy at all. (When  $n = 11$ , the forward error is roughly 0.1, which indicates that only the first decimal digit can be expected to have any accuracy.)

## 5.4 Handling Warning Exits from the Library

The appearance of a warning message from the Library in the middle of the results is intended to alert users to possible problems. But, once having been alerted, you may wish to build some response to this condition into your program; you may wish to test whether a warning has been raised, and take action accordingly.

To do this, you must supply the optional error-handling argument `error`, so that you can test it on exit from the procedure. Here is a modified version of program `tutorial_ex5c`, which traps the warning, suppresses the message from the procedure, and takes alternative action (for the purposes of this simple example, it just prints a different message).

```

PROGRAM tutorial_ex5d
  USE nag_gen_lin_sys, ONLY : nag_gen_lin_sol, nag_error, nag_set_error
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER, PARAMETER :: n_max = 12
  INTEGER :: i, j, n
  REAL (wp) :: rcond, err
  TYPE (nag_error) :: error
  REAL (wp) :: a(n_max,n_max), b(n_max)
  WRITE (*,*) '  n    condition    forward'
  WRITE (*,*) '      number      error'
  DO n = 1, n_max
    DO j = 1, n

```

```

      DO i = 1, n
        a(i,j) = 1.0_wp/REAL(i+j-1,kind=wp)
      END DO
    END DO
    b(1:n) = 1.0_wp
    CALL nag_set_error(error,print_level=2)
    CALL nag_gen_lin_sol(a(1:n,1:n),b(1:n),rcond=rcond,fwd_err=err, &
      error=error)
    SELECT CASE (error%level)
    CASE DEFAULT
      WRITE (*,'(1x,i3,1p,3e12.1)') n, 1.0_wp/rcond, err
    CASE (1)
      WRITE (*,'(1x,i3,a)') n, ' singular to working precision'
    CASE (2:)
      WRITE (*,'(1x,a)') ' Other errors'
    END SELECT
  END DO
END PROGRAM tutorial_ex5d

```

Now the line in the results for  $n = 12$  reads:

```
12 singular to working precision
```

Note the following points.

1. The error-handling argument **error** is a *structure* — that is, an object of a *derived type*.
2. Its type **nag\_error** is defined by the Library, and must be accessed by a **USE** statement: it is accessible through the same module as the library procedure, in this case through the module **nag\_gen\_lin\_sys** (alternatively it can be accessed from the module **nag\_error\_handling**).
3. You must declare a structure **error** of this type in the calling program.
4. You *must initialize* the structure by a call to the procedure **nag\_set\_error** before you pass it to the procedure **nag\_gen\_lin\_sol**; if you do not, you will get an error message:

```

***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_gen_lin_sol           Level = 3   Code = 399
The optional argument error has been supplied but has not been initialized.
***** Execution halted *****

```

5. **nag\_set\_error** must be accessed through a **USE** statement in the same way as the type **nag\_error**.
6. In this example, **nag\_set\_error** is called with the optional argument **print\_level=2**, which means that an error message is printed by the Library only if the level of the error is at least 2 (warnings are at level 1).
7. On exit from **nag\_gen\_lin\_sol**, the component **error%code** contains the error code, as documented in the **Error Codes** section of the procedure documentation, or 0 if the procedure has returned without any error condition ('successful exit').

These points should suffice to explain the program **tutorial\_ex5d**.

To handle warnings in other ways, different arguments must be supplied in the call to **nag\_set\_error**:

To test for a warning *without* suppressing the error message:

```
CALL nag_set_error(error)
```

To *halt* the program after a warning has occurred:

```
CALL nag_set_error(error, halt_level=1)
```

The values of `halt_level` and `print_level` which are supplied as optional arguments to `nag_set_error` are stored as components of the structure `error`; if they are not present, the default values are `halt_level = 2` and `print_level = 1`.

Each error detected by a NAG *f*90 procedure has a designated level, stated in the documentation, and equal to the first digit of its error code. If an error is detected, the procedure prints an error message if the error has a level  $\geq$  `print_level`, and then halts the program if it has a level  $\geq$  `halt_level`.

On return from the procedure, the error code and the level are stored in the components `code` and `level` of the structure `error`, and may be tested, as shown in program `tutorial_ex5d`.

## 6 Example 6: Finding a Solution of a Single Nonlinear Equation

### 6.1 A First Look at Procedure Arguments

The module `nag_nlin_eqn` (10.2) contains a single procedure `nag_nlin_eqn_sol`, which finds a solution of an equation of the following form:

$$f(x) = 0 \text{ where } x \in [a, b].$$

The function  $f(x)$  may be any continuous function which is defined on the interval  $[a, b]$  and which has opposite signs at the end-points  $a$  and  $b$ ; this guarantees that the function has at least one zero in the interval.

In order to use `nag_nlin_eqn_sol` to find a solution of such an equation, the function  $f(x)$  must be defined by a *function subprogram*, which you (the user) must supply. This function is passed as an argument to `nag_nlin_eqn_sol`.

For our example, we will solve the equation

$$e^x = 3x^2 \text{ where } x \in [0, 3],$$

which must be expressed as:

$$f(x) \equiv e^x - 3x^2 = 0. \tag{3}$$

The documentation for `nag_nlin_eqn_sol` specifies the function argument `f` as follows:

**f** — function

**f** must evaluate the function  $f$  at the point  $x$ .

```
function f(x)

real(kind=wp), intent(in) :: x
    Input: the point at which the function must be evaluated.

real(kind=wp) :: f
    Result: the value of the function at the point x.
```

The specification of **f** is very similar to the specification of the library procedure `nag_nlin_eqn_sol`, but it serves a different purpose: you have to *call* `nag_nlin_eqn_sol`, but you have to *write* the function **f**. That is not difficult in this case. Here is a first attempt:

```
FUNCTION f(x)
  f = EXP(x) - 3*x**2
END FUNCTION f
```

Now here is a main program which calls `nag_nlin_eqn_sol` to solve equation (3):

```
PROGRAM tutorial_ex6a ! not recommended: uses EXTERNAL
  USE nag_nlin_eqn, ONLY : nag_nlin_eqn_sol
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp) :: f, x
  EXTERNAL f
  CALL nag_nlin_eqn_sol(f,x,0.0_wp,3.0_wp)
  WRITE (*,'(1x,a,f12.7)') 'solution at x =', x
END PROGRAM tutorial_ex6a
```

This program is not recommended as a model to copy; it does not make full use of the features of Fortran 90. It uses the Fortran 77 `EXTERNAL` statement which tells the compiler that `f` is an external subprogram, but nothing more.

If we compile the main program and the function `f`, link them together and run the program, then the behaviour of the program is likely to vary from one system to another, but on one particular system it gave the result:

```
solution at x = 3.0000000
```

This is wrong!  $e^3 \neq 3 \times 3^2$ ! The reason is that although the main program has been coded using a symbolic kind value `wp` in our usual style (resulting in the use of double precision), the function `f` has been coded using the default (single precision) real type. The NAGWare compiler would have detected an error at compile time if the main program and the function `f` were presented in the same file, giving the error message

```
Error: Wrong data type for reference to function F from TUTORIAL_EX7A
```

But it does not detect an error if they are presented in separate files, and indeed it cannot, because as individual program units both the main program and the function `f` are free from error; the error arises from linking them together.

The compiler needs more information about the characteristics of the function `f`, when it is compiling the main program, than is provided by the `EXTERNAL` statement.

One mechanism provided by Fortran 90 for providing this information is an *interface block*, but a much more satisfactory approach is to embed the function in a module.

## 6.2 Embedding User-supplied Procedures in Modules

The following program is a restructured version of `tutorial_ex6a`, with the function `f` embedded in a module:

```
MODULE tutorial_ex6b_mod
CONTAINS

  FUNCTION f(x)
    f = EXP(x) - 3*x**2
  END FUNCTION f

END MODULE tutorial_ex6b_mod

PROGRAM tutorial_ex6b
  USE tutorial_ex6b_mod, ONLY : f
  USE nag_nlin_eqn, ONLY : nag_nlin_eqn_sol
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp) :: x
  CALL nag_nlin_eqn_sol(f,x,0.0_wp,3.0_wp)
  WRITE (*,'(1x,a,f12.7)') 'solution at x =', x
END PROGRAM tutorial_ex6b
```

Another `USE` statement has been included in the main program in order to access the definition of `f` from the module; the type definition and the `EXTERNAL` declaration for `f` have been deleted from the



main program. (In technical terms, the function `f` has been converted from an *external procedure* into a *module procedure*.)

The program `tutorial_ex6b` will still not be correct for use with a double precision implementation of the Library, but now the error can be detected at compile-time. The NAGWare f90 compiler gives the familiar message:

Error: No specific match for reference to generic NAG\_NLIN\_EQN\_SOL at line 16

Now we will code the function `f` correctly in the same style as the main program:

```
MODULE tutorial_ex6c_mod
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
CONTAINS

  REAL (wp) FUNCTION f(x)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x
    f = EXP(x) - 3.0_wp*x**2
  END FUNCTION f

END MODULE tutorial_ex6c_mod

PROGRAM tutorial_ex6c
  USE tutorial_ex6c_mod, ONLY : f, wp
  USE nag_nlin_eqn, ONLY : nag_nlin_eqn_sol
  IMPLICIT NONE
  REAL (wp) :: x
  CALL nag_nlin_eqn_sol(f,x,0.0_wp,3.0_wp)
  WRITE (*,'(1x,a,f12.7)') 'solution at x =', x
END PROGRAM tutorial_ex6c
```

Note that `wp` must be defined in the module, because it is needed in the type declarations in the function `f`; the main program can access this value from the module, instead of defining it independently; this ensures that the same value is used in both program units.

Now the program gives the correct result:

```
solution at x =    0.9100076
```

### 6.3 Using Modules to Share Data

There is another advantage from embedding the user-supplied function `f` in a module: it allows data to be shared between the main program and the function, without using `COMMON` (the only means available in Fortran 77).

Consider the slightly more general problem of finding a solution to

$$e^x = \alpha x^2 \text{ where } x \in [a, b],$$

allowing arbitrary values of  $\alpha$ ,  $a$  and  $b$  to be input at run-time.

The value of  $\alpha$  is read by the main program, but needs to be communicated to the function `f`. The following program shows a shared variable `alpha` declared in the module; `alpha` is accessible to the function `f` which is contained in the module, and it is also accessible to the main program through the `USE` statement.

```
MODULE tutorial_ex6d_mod
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp) :: alpha
CONTAINS

  REAL (wp) FUNCTION f(x)
    IMPLICIT NONE
```

```

      REAL (wp), INTENT (IN) :: x
      f = EXP(x) - alpha*x**2
END FUNCTION f

END MODULE tutorial_ex6d_mod

PROGRAM tutorial_ex6d
  USE tutorial_ex6d_mod, ONLY : f, wp, alpha
  USE nag_nlin_eqn, ONLY : nag_nlin_eqn_sol
  IMPLICIT NONE
  REAL (wp) :: a, b, x
  DO
    WRITE (*,*) 'enter alpha, a, b'
    READ (*,*,end=10) alpha, a, b
    CALL nag_nlin_eqn_sol(f,x,a,b)
    WRITE (*, '(1x,4(a,f12.7))') 'alpha =', alpha, ' a =', a, ' b =', b,&
      ' x =', x
  END DO
10  CONTINUE
END PROGRAM tutorial_ex6d

```

Here are some results obtained from it:

```

enter alpha, a, b
2.5 0.0 2.0
alpha = 2.5000000 a = 0.0000000 b = 2.0000000 x = 1.0916243
enter alpha, a, b
2.5 2.0 4.0
alpha = 2.5000000 a = 2.0000000 b = 4.0000000 x = 3.3104727
enter alpha, a, b
2.5 0.0 4.0
***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_nlin_eqn_sol Level = 3 Code = 301
An input argument has an invalid value.
f(a)*f(b) = 1.459815003314424E+001
a = 0.0000E+00, f(a) = 0.1000E+01
b = 0.4000E+01, f(b) = 0.1460E+02
Bounds a and b must be chosen such that f(a)*f(b) <= 0.
***** Execution halted *****

```

The reason for the fatal error is that the function  $f(x) = e^x - 2.5x^2$  has *two* zeros between 0 and 4, as the preceding lines have shown;  $f(0)$  and  $f(4)$  have the same sign, and this violates the constraints on the arguments  $a$  and  $b$ .

## 7 Example 7: One-dimensional Quadrature

### 7.1 An Array-valued User-supplied Function

The problem in this section is to compute an approximation to the integral

$$\int_0^1 \frac{dx}{\sqrt{|x^2 + 2x - 2|}}. \quad (4)$$

The module `nag_quad_1d` (11.1) handles numerical integration of a function of one variable, over a finite interval. It contains three procedures for integrands of special forms  $f(x) = w(x)g(x)$ , where  $w(x)$  is a weight function of a specified class. However, the integral (4) does not fit any of these forms, so we will use the procedure `nag_quad_1d_gen` which is designed to handle general integrands.

This procedure has a function argument `f` which defines the integrand. It is documented as follows:

**f** — function

**f** must return the values of the integrand  $f$  at a set of points.

```
function f(x)

real(kind=wp), intent(in) :: x(:)
    Input: the points at which the integrand f must be evaluated.

real(kind=wp) :: f(SIZE(x))
    Result: f(i) must contain the value of f at x(i), for i = 1, 2, ..., SIZE(x).
```

There is an important difference between this function and the function argument of `nag_nlin_eqn_sol` which was described in Section 6.1. The argument **f** of `nag_quad_1d_gen` is *array valued*; its result is an array, of the same length as **x**, as is stated by the specification:

```
real(kind=wp) :: f(SIZE(x))
```

Note that the function result **f** cannot be an assumed shape array, because it is not a dummy argument; its shape must either be constant, or be defined in terms of properties of its arguments, as here.

The reason for requiring **f** to be array valued is that a quadrature algorithm often needs to evaluate its integrand at a large number of points, and it is sometimes much more efficient if several function values can be computed in a single call to the user-supplied function. The number of function values required in each call to **f** is determined by the library procedure `nag_quad_1d_gen`; it can be determined within the code for **f** as the value `SIZE(x)`.

Here is a program to evaluate the integral (4) using default values for the optional arguments which specify the required accuracy. The procedure **f** is embedded in a module, as was recommended in Section 6.2.

```
MODULE tutorial_ex7a_mod
  USE tutorial_kind, ONLY : wp
CONTAINS

  FUNCTION f(x)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x(:)
    REAL (wp) :: f(SIZE(x))
    f = 1.0_wp/SQRT(ABS(x**2+2.0_wp*x-2.0_wp))
  END FUNCTION f

END MODULE tutorial_ex7a_mod

PROGRAM tutorial_ex7a
  USE tutorial_ex7a_mod, ONLY : f, wp
  USE nag_quad_1d, ONLY : nag_quad_1d_gen
  IMPLICIT NONE
  REAL (wp) :: result, abs_err
  INTEGER :: nf
  CALL nag_quad_1d_gen(f,0.0_wp,1.0_wp,result,abs_err=abs_err, &
    num_fun_eval=nf)
  WRITE (*, '(1x,a,f12.8)') 'result =', result, 'abs_err =', abs_err
  WRITE (*, '(1x,a,i12)') 'number of function evaluations =', nf
END PROGRAM tutorial_ex7a
```

This gives the results:

```
result = 1.50446645
```

```
abs_err = 0.00008186
number of function evaluations = 903
```

The code for the function `f` does not look at first sight as if it is array valued. The statement

```
f = 1.0_wp/SQRT(ABS(x**2+2.0_wp*x-2.0_wp))
```

is identical to the statement that you would write for a scalar-valued function with a scalar argument. But on the right-hand side of the assignment, `x` is an array, so the arithmetic expressions are array valued, and the intrinsic functions `SQRT` and `ABS` are *elemental*, which means that they return an array-valued result if their arguments are arrays. And finally `f` on the left-hand side of the assignment is an array, so the statement as a whole is an array assignment. It is equivalent to:

```
DO i = 1, SIZE(x)
  f(i) = 1.0_wp/SQRT(ABS(x(i)**2+2.0_wp*x(i)-2.0_wp))
END DO
```

A DO-loop would be essential if the expression for  $f(x)$  involved other functions that are not array valued.

## 7.2 Handling Failure Exits from the Library

Now suppose that we wish to investigate the effect of specifying different values for the requested accuracy. Since the value of the integral is close to 1, absolute and relative accuracy have roughly the same meaning. We will set the relative accuracy `rel_acc` to zero, which means that only absolute accuracy is operative.

The following program requests a progressively more accurate result, setting `abs_acc` to  $10^{-3}$ ,  $10^{-4}$ , ...:

```
MODULE tutorial_ex7b_mod
  USE tutorial_kind, ONLY : wp
CONTAINS

  FUNCTION f(x)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x(:)
    REAL (wp) :: f(SIZE(x))
    f = 1.0_wp/SQRT(ABS(x**2+2.0_wp*x-2.0_wp))
  END FUNCTION f

END MODULE tutorial_ex7b_mod

PROGRAM tutorial_ex7b
  USE tutorial_ex7b_mod, ONLY : f, wp
  USE nag_quad_1d, ONLY : nag_quad_1d_gen
  IMPLICIT NONE
  INTEGER :: i, nf
  REAL (wp) :: result, abs_err, abs_acc
  WRITE (6,*) ' requested      result      estimated      number of'
  WRITE (6,*) ' accuracy              accuracy evaluations'
  DO i = 3, 10
    abs_acc = 10.0_wp**(-i)
    CALL nag_quad_1d_gen(f,0.0_wp,1.0_wp,result,abs_acc=abs_acc, &
      rel_acc=0.0_wp,abs_err=abs_err,num_fun_eval=nf)
    WRITE (*,'(1x,3f12.8,i12)') abs_acc, result, abs_err, nf
  END DO
END PROGRAM tutorial_ex7b
```

The results are:

requested accuracy	result	estimated accuracy	number of evaluations
0.00100000	1.50439172	0.00075447	819
0.00010000	1.50446645	0.00008186	903
0.00001000	1.50460652	0.00000114	1197
0.00000100	1.50462235	0.00000005	1743
0.00000010	1.50462235	0.00000005	1827

\*\*\*\*\* Failure reported by NAG Fortran 90 Library \*\*\*\*\*

```

Procedure nag_quad_1d_gen                      Level = 2   Code = 203
Failure due to bad local behaviour:
Extremely bad local behaviour of the integrand causes a very
strong subdivision around one (or more) points of the interval.
***** Execution halted *****

```

When the requested accuracy is  $10^{-8}$ , the procedure takes an error exit, reporting error code 203, which is a computational *failure*: the procedure cannot satisfy the requested accuracy. The error message refers to ‘bad local behaviour’, and the more extensive advice in the procedure documentation says: ‘look at the integrand’. The integrand which we have chosen is in fact quite simple, and it would be easy to plot it, or to analyse it mathematically, and reveal that it has a singularity within the interval of integration.

However, even when that is not possible, `nag_quad_1d_gen` returns information in its optional argument `subint_info` which can help to diagnose the cause of the difficulty. But we cannot examine the information returned in `subint_info` unless we enable the program to continue execution on return from `nag_quad_1d_gen` after the computational failure.

To do this, we must supply the optional argument `error` and initialise it using `nag_set_error`, as was described in Section 5.4. Only this time we must set `halt_level` to 3, so that the procedure will only halt the program if the error is at level 3 (a fatal error).

The required call to `nag_set_error` is:

```
CALL nag_set_error(error, halt_level=3)
```

or if we want to suppress the error message at the same time,

```
CALL nag_set_error(error, halt_level=3, print_level=3)
```

Take care: there are several different error exits at level 2, and the calling program must be prepared to deal with *all* of them. However, in our example we do not in fact need to distinguish between them. All we want to do is to print the information returned in `subint_info`.

### 7.3 An Argument Which is an Array Pointer

The specification of `subint_info` begins:

**subint\_info(:, :)** — real(kind=wp), pointer, optional

*Output:* details of the computation which may be examined in the event of a failure . . .

It is a *pointer*, and the actual argument in the calling program must therefore be a pointer also.

The reason for specifying it as a pointer is that the calling program cannot know in advance what shape of array will be required: it depends on details of the computation, specifically on the number of subintervals into which the algorithm adaptively subdivides the interval of integration. The procedure allocates storage to the pointer internally, and uses this storage to return the requested information. The calling program can deallocate this storage when it is no longer needed, using a `DEALLOCATE` statement.

In this context, an array pointer is used simply so that the procedure can allocate the required amount of storage; the Fortran 90 language does not allow an allocatable array to appear as a dummy argument, so a pointer array must be used instead.

On entry to the procedure the actual argument should not be associated; that is, it should not ‘point to anything’. If it is associated, that association will be broken by the procedure.

In most respects the pointer array can be used just like an ordinary array. Here is a program:

```

MODULE tutorial_ex7c_mod
  USE tutorial_kind, ONLY : wp
  CONTAINS

  FUNCTION f(x)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x(:)

```

```

      REAL (wp) :: f(SIZE(x))
      f = 1.0_wp/SQRT(ABS(x**2+2.0_wp*x-2.0_wp))
END FUNCTION f

END MODULE tutorial_ex7c_mod

PROGRAM tutorial_ex7c
  USE tutorial_ex7c_mod, ONLY : f, wp
  USE nag_quad_1d, ONLY : nag_quad_1d_gen, nag_error, nag_set_error
  IMPLICIT NONE
  INTEGER :: i, j, nf
  REAL (wp) :: result, abs_err, abs_acc
  TYPE (nag_error) :: error
  REAL (wp), POINTER :: info(:, :)
  WRITE (6,*) ' requested      result      estimated   number of'
  WRITE (6,*) ' accuracy          accuracy evaluations'
  DO i = 3, 10
    abs_acc = 10.0_wp**(-i)
    CALL nag_set_error(error, halt_level=3)
    CALL nag_quad_1d_gen(f, 0.0_wp, 1.0_wp, result, abs_acc=abs_acc, &
      rel_acc=0.0_wp, abs_err=abs_err, num_fun_eval=nf, subint_info=info, &
      error=error)
    IF (error%code==0) THEN
      WRITE (*, '(1x,3f12.8,i12)') abs_acc, result, abs_err, nf
    ELSE
      WRITE (*,*) 'integration failed - subint_info returns:'
      WRITE (*, '(1x,3f12.8)') (info(j,1:3), j=1, SIZE(info,1))
    END IF
    DEALLOCATE (info)
  END DO
END PROGRAM tutorial_ex7c

```

Note the use of the expression `SIZE(info,1)` to determine the first dimension of the array that has been allocated by the procedure.

The results are too long to reproduce here, but they clearly show that the procedure has located a difficulty in the region of  $x = 0.73205\dots$ , where the integrand has a singularity.

The `DEALLOCATE` statement inside the main loop deserves some explanation. Each call to `nag_quad_1d_gen` creates a new *target* for the pointer `info` — that is, it allocates new storage to hold the data to which the optional argument `info` points. The program would give just the same results if the `DEALLOCATE` statement were omitted; but only the target created in the most recent call could be accessed though the pointer `info`; the targets created in previous calls would be inaccessible and the storage allocated to them could not be recovered. This situation is known as a ‘memory leak’. If large amounts of memory ‘leak away’ in this manner, the program could fail for lack of memory. That would not be a realistic danger for the trivial program `tutorial_ex7c`; the `DEALLOCATE` statement is included as an illustration of safe programming practice.

## 8 Example 8: Interpolation

### 8.1 Using a Structure to Communicate Between Procedures

The example problem here is to interpolate the following set of data:

$x_i$	$f_i$
0.0	0.95
0.1	0.90
0.3	0.10
0.4	0.05
0.5	0.05
0.7	0.20
1.0	1.00

To be more precise, we want to be able to calculate a ‘reasonable’ value of  $f(x)$  for any value of  $x$  in the interval  $[0, 1]$  — that is, within the range of the given values  $x_i$ .

NAG *f190* contains two procedures for interpolating a curve through a set of data:

`nag_pch_monot_interp` in module `nag_pch_interp` (8.1) computes a ‘monotonicity preserving’ interpolant which is represented mathematically as a ‘piecewise cubic Hermite’ polynomial (hence ‘pch’);

`nag_spline_1d_interp` in module `nag_spline_1d` (8.2) computes a cubic spline interpolant.

The Introduction to Chapter 8 (**Curve and Surface Fitting**) offers advice on the respective merits of these two methods for interpolation. Here we choose the first, which ‘preserves monotonicity in the data’. This means that when the data are strictly decreasing, as they are in our example over the interval  $[0, 0.4]$ , the interpolated curve is likewise strictly decreasing; and when the data are strictly increasing, as they are over  $[0.5, 1]$ , so also is the interpolated curve. This criterion prevents unwanted fluctuations in the curve.

To solve the stated problem, you will need to call *two* NAG *f190* procedures:

first, `nag_pch_monot_interp` computes a representation of the interpolant (that is, the interpolating curve);

second, `nag_pch_eval` evaluates the interpolant at a specified value or values of  $x$ .

The two parts of the computation — computing the interpolant and evaluating it — are decoupled. This decoupling is a typical feature of software for data fitting, and is used throughout Chapter 8.

In our example, the representation of the interpolant must be communicated from one procedure to another. For this purpose a *structure* is used — that is, an object of a *derived type* which is defined by the Library.

The name of the derived type is given in the documentation as `nag_pch_comm_wp`; we will explain the suffix `_wp` shortly. Each derived type defined by the Library is documented in a separate section of a module document, following the documentation of the procedures. For the type `nag_pch_comm_wp`, you will find that the **Type Definition** section reads

```
type nag_pch_comm_wp
  private
  .
  .
  .
end type nag_pch_comm_wp
```

and the **Components** section states: ‘In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.’ For normal purposes, you *do not need to know* how the interpolant is represented in the structure; you only need to declare a structure of the type `nag_pch_comm_wp` and pass it as an argument to the procedures, as will be illustrated shortly. (For those who want to perform some specialized operation that requires access to the details of the interpolant, a procedure `nag_pch_extract` is provided.)

Here is a program which reads in data from a file, calls `nag_pch_monot_interp` to construct an interpolant, and then calls `nag_pch_eval` to evaluate the interpolant at specified values of  $x$ .

```
PROGRAM tutorial_ex8a ! valid for double precision only
  USE nag_pch_interp, ONLY : nag_pch_monot_interp, nag_pch_eval
  USE nag_pch_interp, ONLY : nag_pch_comm_dp
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, m
  REAL (wp) :: x, f
  REAL (wp), ALLOCATABLE :: xdata(:), fdata(:)
```

```

      TYPE (nag_pch_comm_dp) :: interp
      OPEN (4,file='tutorial_ex8a.dat')
      READ (4,*) m
      ALLOCATE (xdata(m),fdata(m))
      READ (4,*) (xdata(i),fdata(i),i=1,m)
      CALL nag_pch_monot_interp(xdata,fdata,interp)
      DO
        WRITE (*,*) 'Enter x-value for interpolation'
        READ (*,*,end=10) x
        CALL nag_pch_eval(interp,x,f)
        WRITE (*,'(1x,2(a,f8.4))') 'At x =', x, ' f(x) =', f
      END DO
10    CONTINUE
      END PROGRAM tutorial_ex8a

```

Suppose that the data are read from the following file:

```

7
0.0  0.95
0.1  0.90
0.3  0.10
0.4  0.05
0.5  0.05
0.7  0.20
1.0  1.00

```

then here is an example of what might appear on the screen:

```

Enter x-value for interpolation
0.2
At x =  0.2000 f(x) =  0.5000
Enter x-value for interpolation
0.45
At x =  0.4500 f(x) =  0.0500
Enter x-value for interpolation
0.6
At x =  0.6000 f(x) =  0.0968
Enter x-value for interpolation
0.8
At x =  0.8000 f(x) =  0.3727
Enter x-value for interpolation
0.9
At x =  0.9000 f(x) =  0.6480
Enter x-value for interpolation

```

## 8.2 Derived Types and Precision

Note that this program, as it stands, is valid only for double precision implementations of the Library. This relates to the use of the derived type `nag_pch_comm_wp`. The type has real components which may in practice be either double precision or single precision. It would help if we could write something like this:

```

      TYPE (nag_pch_comm(KIND=wp)) :: interp ! not valid Fortran 90

```

but the Fortran 90 language does not allow derived types to be parameterized with a kind value in the same way as the intrinsic type `REAL`. (Proposals are being considered for overcoming this limitation in a future revision of the standard, but for the time being we have to live with it.)

Therefore, NAG *f*90 has to define the type with a precision-dependent name:

`nag_pch_comm_dp` for double precision;

`nag_pch_comm_sp` for single precision.

When the documentation uses the name `nag_pch_comm_wp`, the suffix `_wp` must be interpreted as either `_dp` or `_sp`. If you expect to run your programs always in double precision, then you can simply translate `_wp` to `_dp` in your mind, and write your program like `tutorial_ex8a` above.



To convert that program to single precision, you would need to change two statements:

```
USE nag_pch_interp, ONLY : nag_pch_comm_sp
...
TYPE(nag_pch_comm_sp) :: interp
```

If you do expect that you will need to convert your program from one precision to another, the following coding practice means that you will need to make only one change:

```
USE nag_pch_interp, ONLY : nag_pch_comm_wp => nag_pch_comm_dp
...
TYPE(nag_pch_comm_wp) :: interp
```

Here we are using the *renaming* facility in the `USE` statement to define a *local name* `nag_pch_comm_wp` which is mapped onto the name defined in the module. To convert the program to single precision, we need only change the `USE` statement to:

```
USE nag_pch_interp, ONLY : nag_pch_comm_wp => nag_pch_comm_sp
```

However, this still means that there is a precision dependency in your program. It is preferable in general, especially if you have a large program with several program units, to remove all precision dependencies into a separate module, just as the definition of the kind value `wp` has been removed into the module `tutorial_kind` in all the examples in this Tutorial since Section 2.3.

Therefore we will use the following module:

```
MODULE tutorial_types ! double precision version
  USE nag_pch_interp, ONLY : nag_pch_comm_wp => nag_pch_comm_dp
  USE nag_nlin_lsq, ONLY : nag_nlin_lsq_cntrl_wp => nag_nlin_lsq_cntrl_dp
END MODULE tutorial_types
```

which should be pre-compiled and made accessible to the compiler in the same manner as the module `tutorial_kind`. (The type `nag_nlin_lsq_cntrl_dp` will be needed in Section 9.2.) The program `tutorial_ex8a` can then be re-written as follows:

```
PROGRAM tutorial_ex8b
  USE nag_pch_interp, ONLY : nag_pch_monot_interp, nag_pch_eval
  USE tutorial_types, ONLY : nag_pch_comm_wp
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, m
  REAL (wp) :: x, f
  REAL (wp), ALLOCATABLE :: xdata(:), fdata(:)
  TYPE (nag_pch_comm_wp) :: interp
  OPEN (4,file='tutorial_ex8a.dat')
  READ (4,*) m
  ALLOCATE (xdata(m),fdata(m))
  READ (4,*) (xdata(i),fdata(i),i=1,m)
  CALL nag_pch_monot_interp(xdata,fdata,interp)
  DO
    WRITE (*,*) 'Enter x-value for interpolation'
    READ (*,*,end=10) x
    CALL nag_pch_eval(interp,x,f)
    WRITE (*,'(1x,2(a,f8.4))') 'At x =', x, ' f(x) =', f
  END DO
10  CONTINUE
END PROGRAM tutorial_ex8b
```

Frequently one wants to interpolate a curve at a large number of points, in order to plot it for example. The procedure `nag_pch_eval` is another example of a procedure which is generic with respect to the *rank* of its arguments. Its arguments `u` and `h` may be either scalars or one-dimensional arrays; they must have the same rank, and if arrays, the same shape.

Here is a modified version of `tutorial_ex8b` which evaluates the interpolant at equally spaced intervals between the limits of the data.

```

PROGRAM tutorial_ex8c
  USE nag_pch_interp, ONLY : nag_pch_monot_interp, nag_pch_eval
  USE tutorial_types, ONLY : nag_pch_comm_wp
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  INTEGER :: i, m, n
  REAL (wp), ALLOCATABLE :: xdata(:), fdata(:), x(:), f(:)
  TYPE (nag_pch_comm_wp) :: interp
  OPEN (4,file='tutorial_ex8a.dat')
  READ (4,*) m
  ALLOCATE (xdata(m),fdata(m))
  READ (4,*) (xdata(i),fdata(i),i=1,m)
  CALL nag_pch_monot_interp(xdata,fdata,interp)
  WRITE (*,*) 'Enter number of evaluation points'
  READ (*,*) n
  ALLOCATE (x(n),f(n))
  DO i = 1, n
    x(i) = ((n-i)*xdata(1)+(i-1)*xdata(m))/(n-1)
  END DO
  CALL nag_pch_eval(interp,x,f)
  WRITE (*,*) '      x      f(x)'
  WRITE (*,'(1x,2f8.4)') (x(i),f(i),i=1,n)
END PROGRAM tutorial_ex8c

```

Here are some typical results:

```

Enter number of evaluation points
11
      x      f(x)
0.0000  0.9500
0.1000  0.9000
0.2000  0.5000
0.3000  0.1000
0.4000  0.0500
0.5000  0.0500
0.6000  0.0968
0.7000  0.2000
0.8000  0.3727
0.9000  0.6480
1.0000  1.0000

```

## 9 Example 9: Nonlinear Least-squares

### 9.1 A Procedure Argument with an Optional Argument

This example concerns a set of data, which represents observations or measurements of a variable  $y$ , for given values of variables  $u$ ,  $v$  and  $w$ :

$u_i$	$v_i$	$w_i$	$y_i$
1.0	15.0	1.0	0.14
2.0	14.0	2.0	0.18
3.0	13.0	3.0	0.22
4.0	12.0	4.0	0.25
5.0	11.0	5.0	0.29
6.0	10.0	6.0	0.32
7.0	9.0	7.0	0.35
8.0	8.0	8.0	0.39
9.0	7.0	7.0	0.37
10.0	6.0	6.0	0.58
11.0	5.0	5.0	0.73
12.0	4.0	4.0	0.96
13.0	3.0	3.0	1.34
14.0	2.0	2.0	2.10
15.0	1.0	1.0	4.39

We assume that the data can be modelled by an expression of the form

$$y = \alpha + \frac{u}{\beta v + \gamma w} \quad (5)$$

with model parameters  $\alpha$ ,  $\beta$  and  $\gamma$ . The problem is to determine values of those parameters so that the model best fits the data (in a least-squares sense). The model is *nonlinear* in the parameters  $\alpha$ ,  $\beta$  and  $\gamma$ , so the problem is described as a *nonlinear least-squares* problem (or as a *nonlinear regression* problem).

More precisely, the problem is to find values of  $\alpha$ ,  $\beta$  and  $\gamma$  which *minimize* the sum of the squares of the residuals, that is, the differences between the observed values  $y_i$  and the predictions of the model (5):

$$\sum_{i=1}^m \left[ y_i - \left( \alpha + \frac{u_i}{\beta v_i + \gamma w_i} \right) \right]^2.$$

Such problems can be solved by the NAG *f90* procedure `nag_nlin_lsq_sol` in the module `nag_nlin_lsq` (9.2). This procedure is described as being applicable to problems of the form

$$\text{Minimize } F(x) = \sum_{i=1}^m (f_i(x))^2$$

where  $x = (x_1, x_2, \dots, x_n)^T$  and  $m \geq n$ . We need to match this notation with that which we have used for our problem, as follows:

$$f_i(x) = y_i - \left( x_1 + \frac{u_i}{x_2 v_i + x_3 w_i} \right) \quad (6)$$

where  $x_1 = \alpha$ ,  $x_2 = \beta$  and  $x_3 = \gamma$ , so  $m = 15$  (the number of observations), and  $n = 3$  (the number of unknowns). The functions  $f_i$  are often referred to as the ‘residuals’.

The description of `nag_nlin_lsq_sol` says: ‘You must supply a procedure to calculate the values of the  $f_i(x)$  and, optionally, their first derivatives  $\partial f_i / \partial x_j$  at any point  $x$ .’

Like most algorithms for nonlinear minimization, the algorithm used by `nag_nlin_lsq_sol` is usually more reliable, robust and efficient if partial derivatives of the function can be evaluated at any point, as well as the function itself. If you do not supply code to evaluate partial derivatives, `nag_nlin_lsq_sol` estimates them by finite differences, but this is likely to be less accurate, and requires many more evaluations of the functions  $f_i(x)$  themselves.

For complicated functions, writing code to evaluate partial derivatives may be difficult, and itself prone to error. The difficulties can often be alleviated by use of a symbolic algebra package, or by the technique of automatic differentiation (which we hope to provide in a future release of NAG *f90*). For our simple

functions (6), the partial derivatives are:

$$\frac{\partial f_i}{\partial x_1} = -1$$

$$\frac{\partial f_i}{\partial x_2} = \frac{u_i v_i}{(x_2 v_i + x_3 w_i)^2}$$

$$\frac{\partial f_i}{\partial x_3} = \frac{u_i w_i}{(x_2 v_i + x_3 w_i)^2}$$

and they can be coded quite easily.

To reduce the risk of error, `nag_nlin_lsqr_sol` performs some preliminary checks that the code for the partial derivatives is consistent with the code for the function values.

Before you can write the code, you need to read the specification of the procedure `lsqr_fun` which you must supply (some of the text has been omitted here for simplicity):

**lsqr\_fun** — subroutine

...

Its specification is:

```
subroutine lsqr_fun(x, finish, f_vec, f_jac)

real(kind=wp), intent(in) :: x(:)
    Shape: x has shape (n).
    Input: the point x at which the values of f_i and (optionally) ∂f_i/∂x_j are required, for
           i = 1, 2, ..., m; j = 1, 2, ..., n.

logical, intent(inout) :: finish
    Input: finish will always be .false. on entry.
    Output: if you wish to terminate the call to nag_nlin_lsqr_sol then finish should be set
           to .true.. ...

real(kind=wp), intent(out) :: f_vec(:)
    Shape: f_vec has shape (m).
    Output: unless finish is reset to .true., f_vec(i) must contain the value of f_i at the
           point x, for i = 1, 2, ..., m.

real(kind=wp), intent(out), optional :: f_jac(:, :)
    Shape: f_jac has shape (m, n).
    Output: if present, f_jac(i, j) must contain the value of the first derivative ∂f_i/∂x_j at
           the point x, for i = 1, 2, ..., m; j = 1, 2, ..., n.
    Note: ...
```

The matrix of first partial derivatives is known as the *Jacobian* matrix; hence the argument name `f_jac`. `nag_nlin_lsqr_sol` has an optional argument `deriv` which specifies whether partial derivatives are provided in `lsqr_fun`. The two options are:

**Partial derivatives supplied:** omit the optional argument `deriv` or set it to its default value `.true.`; include code in `lsqr_fun` to return values of the partial derivatives in `f_jac` if present.

**Partial derivatives not supplied:** set the optional argument `deriv` to `.false.`; the argument `f_jac` will never be present in calls to `lsqr_fun`, and hence no code to evaluate derivatives need be included.

Note however that you *must always include* `f_jac` in the argument-list of `lsq_fun`: it is optional in calls to `lsq_fun`; it is *not* optional in the argument-list.

The first option is the default. Here is code for a suitable subroutine `lsq_fun`, embedded in a module as was recommended in Section 6.2:

```
MODULE tutorial_ex9x_mod
  USE tutorial_kind, ONLY : wp
  REAL (wp), ALLOCATABLE :: u(:), v(:), w(:), y(:)
CONTAINS

  SUBROUTINE lsq_fun(x,finish,f_vec,f_jac)
    REAL (wp), INTENT (IN) :: x(:)
    LOGICAL, INTENT (INOUT) :: finish
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (OUT), OPTIONAL :: f_jac(:, :)
    f_vec = y - (x(1) + u/(x(2)*v+x(3)*w))
    IF (PRESENT(f_jac)) THEN
      f_jac(:,1) = -1.0_wp
      f_jac(:,2) = (u*v)/(x(2)*v+x(3)*w)**2
      f_jac(:,3) = (u*w)/(x(2)*v+x(3)*w)**2
    END IF
  END SUBROUTINE lsq_fun

END MODULE tutorial_ex9x_mod
```

Note these points:

1. The subroutine statement and the declarations of the arguments are essentially identical to those in the specification.
2. The array arguments of `lsq_fun` are all specified as assumed-shape arrays; a separate line in the specification (with the subheading *Shape:*) states the dimensions of the arrays which will be passed to `lsq_fun` in terms of the dimensions of the problem ( $m$  and  $n$ ). Even though you know that  $\mathbf{x}$  will always have dimension (3) (as long as you try to fit the same model), you must declare it as an assumed-shape array `x(:)`, and not as an explicit shape array `x(3)`, because then your procedure `lsq_fun` would not match the characteristics of `lsq_fun` as defined in the library.
3. The value of  $m$  (which might well vary with different data-files) could be obtained in `lsq_fun` as `SIZE(f_vec)`. However, for our problem the code can be written entirely in array syntax, without explicitly using  $m$ .
4. The code as written is likely to involve repeated evaluation of the array subexpression `x(2)*v+x(3)*w` (unless the compiler is very clever). The complete program which follows contains a revised version of `lsq_fun` in which this subexpression is evaluated once and stored in a temporary array `denom`.
5. There is a risk of division by zero if any element of this array subexpression is zero. The revised version of `lsq_fun` includes a test for zero elements; if one is found, `lsq_fun` sets the argument `finish` to `.true.`, in order to force a clean error exit from `nag_nlin_lsq_sol`.

Here now is a complete program to solve the problem:

```
MODULE tutorial_ex9a_mod
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp), ALLOCATABLE :: u(:), v(:), w(:), y(:)
CONTAINS

  SUBROUTINE lsq_fun(x,finish,f_vec,f_jac)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x(:)
    LOGICAL, INTENT (INOUT) :: finish
    REAL (wp), INTENT (OUT) :: f_vec(:)
```

```

REAL (wp), INTENT (OUT), OPTIONAL :: f_jac(:, :)
REAL (wp) :: denom(SIZE(f_vec))
denom = x(2)*v+x(3)*w
IF (ANY(denom==0)) THEN
  WRITE (*,*) 'lsq_fun encountered a zero denominator'
  finish = .TRUE.
ELSE
  f_vec = y - (x(1) + u/denom)
  IF (PRESENT(f_jac)) THEN
    f_jac(:,1) = -1.0_wp
    f_jac(:,2) = (u*v)/denom**2
    f_jac(:,3) = (u*w)/denom**2
  END IF
END IF
END SUBROUTINE lsq_fun

END MODULE tutorial_ex9a_mod

PROGRAM nag_tutorial_ex9a
  USE tutorial_ex9a_mod, ONLY : lsq_fun, u, v, w, y, wp
  USE nag_nlin_lsq, ONLY : nag_nlin_lsq_sol
  IMPLICIT NONE
  REAL (wp) :: f_sum_sq
  INTEGER :: i, m
  REAL (wp) :: x(3)
  REAL (wp), ALLOCATABLE :: f_vec(:)
  OPEN (4,file='tutorial_ex9a.dat')
  READ (4,*) m
  ALLOCATE (f_vec(m),u(m),v(m),w(m),y(m))
  READ (4,*) (u(i),v(i),w(i),y(i),i=1,m)
  READ (4,*) x
  CALL nag_nlin_lsq_sol(lsq_fun,x,f_sum_sq,f_vec)
END PROGRAM nag_tutorial_ex9a

```

The program does not include any code to print the results because `nag_nlin_lsq_sol` (like other optimization procedures in NAG *f90*) prints its own results; this is the default option, but it can be suppressed (see Section 9.2).

The data may be supplied in the following file, which also includes an initial guess for `x` (an estimate of the solution) in the last record:

```

15
1.0 15.0 1.0 0.14
2.0 14.0 2.0 0.18
3.0 13.0 3.0 0.22
4.0 12.0 4.0 0.25
5.0 11.0 5.0 0.29
6.0 10.0 6.0 0.32
7.0 9.0 7.0 0.35
8.0 8.0 8.0 0.39
9.0 7.0 7.0 0.37
10.0 6.0 6.0 0.58
11.0 5.0 5.0 0.73
12.0 4.0 4.0 0.96
13.0 3.0 3.0 1.34
14.0 2.0 2.0 2.10
15.0 1.0 1.0 4.39
0.0 1.0 2.0

```

and then the results will look like this:

```
Parameters
-----

number of residuals (m)      15      number of variables (n)      3

list.....                  .true.   print_level.....            10
lin_deriv.....              .true.   linesearch_tol.....        9.00E-01
step_max.....              1.00E+05  optim_tol.....             1.49E-08

deriv.....                  .true.   verify.....                 .true.
max_iter.....                50      unit.....                   6
```

```
Verification of the Jacobian matrix
-----
```

The Jacobian matrix seems to be ok.

```
Intermediate Results
-----
```

Itn	Step	Nfun	Objective	Norm g	Grade
0		1	4.710087E-01	3.7E+00	3
1	1.0E+00	2	1.776776E-02	6.0E-01	3
2	1.0E+00	3	8.218126E-03	1.1E-02	3
3	1.0E+00	4	8.214877E-03	3.8E-06	2
4	1.0E+00	5	8.214877E-03	5.4E-10	2

```
Final Result
-----
```

x	g	Singular values
8.24106E-02	3.7E-12	4.1E+00
1.13304E+00	4.9E-10	1.6E+00
2.34370E+00	-2.4E-10	6.1E-02

```
exit from nag_nlin_lsqr after      4 iterations.
```

```
final objective value =      0.8214877E-02
```

```
final residual norm =      9.1E-02
```

```
final gradient norm =      5.4E-10
```

The results include not only the final solution, but also a list of the values of various parameters (the default values in this set of results), a report on successful verification of the computed partial derivatives, and intermediate results which monitor the progress of the iterative algorithm: to interpret the results, see the procedure documentation. Section 9.2 shows how to set options for modifying the output produced by the procedure. Details of the results, such as the final gradient norm or the number of iterations, may vary if you run the program on your own system; you may even find that the procedure returns very similar numerical results, but with a warning:

```
***** Warning reported by NAG Fortran 90 Library *****
Procedure nag_nlin_lsqr_sol                      Level = 1  Code = 102
Current point cannot be improved upon.
***** Execution continued *****
```

This indicates that the stringent criteria used by `nag_nlin_lsqr_sol` for accepting a successful solution could not quite be satisfied, although it is likely that the solution will be satisfactory for practical purposes; see the procedure documentation for further explanation.

The next program shows how the problem can be solved *without* computing partial derivatives in `lsq_fun`. Note that even though `f_jac` is never used, it must still appear in the argument-list for `lsq_fun`.

```

MODULE tutorial_ex9b_mod
  USE tutorial_kind, ONLY : wp
  IMPLICIT NONE
  REAL (wp), ALLOCATABLE :: u(:), v(:), w(:), y(:)
CONTAINS

  SUBROUTINE lsq_fun(x,finish,f_vec,f_jac)
    IMPLICIT NONE
    REAL (wp), INTENT (IN) :: x(:)
    LOGICAL, INTENT (INOUT) :: finish
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (OUT), OPTIONAL :: f_jac(:, :)
    REAL (wp) :: denom(SIZE(f_vec))
    denom = x(2)*v+x(3)*w
    IF (ANY(denom==0)) THEN
      WRITE (*,*) 'lsq_fun encountered a zero denominator'
      finish = .TRUE.
    ELSE
      f_vec = y - (x(1) + u/denom)
    END IF
  END SUBROUTINE lsq_fun

END MODULE tutorial_ex9b_mod

PROGRAM nag_tutorial_ex9b
  USE tutorial_ex9b_mod, ONLY : lsq_fun, u, v, w, y, wp
  USE nag_nlin_lsq, ONLY : nag_nlin_lsq_sol
  IMPLICIT NONE
  REAL (wp) :: f_sum_sq
  INTEGER :: i, m
  REAL (wp) :: x(3)
  REAL (wp), ALLOCATABLE :: f_vec(:)
  OPEN (4,file='tutorial_ex9a.dat')
  READ (4,*) m
  ALLOCATE (f_vec(m),u(m),v(m),w(m),y(m))
  READ (4,*) (u(i),v(i),w(i),y(i),i=1,m)
  READ (4,*) x
  CALL nag_nlin_lsq_sol(lsq_fun,x,f_sum_sq,f_vec,deriv=.FALSE.)
END PROGRAM nag_tutorial_ex9b

```

With the same data file as before, the results will look like this:

#### Parameters

-----

number of residuals (m)	15	number of variables (n)	3
list.....	.true.	print_level.....	10
lin_deriv.....	.true.	linesearch_tol.....	5.00E-01
step_max.....	1.00E+05	optim_tol.....	1.49E-08
deriv.....	.false.	verify.....	.true.
max_iter.....	50	unit.....	6

#### Intermediate Results

-----

Itn	Step	Nfun	Objective	Norm g	Grade
0		4	4.710087E-01	3.7E+00	3
1	1.0E+00	8	1.776776E-02	6.0E-01	3



2	1.0E+00	12	8.218126E-03	1.1E-02	3
3	1.0E+00	21	8.214877E-03	3.7E-06	2
4	1.0E+00	27	8.214877E-03	9.5E-10	2

Final Result  
-----

x	g	Singular values
8.24106E-02	7.7E-10	4.1E+00
1.13304E+00	3.1E-10	1.6E+00
2.34370E+00	4.6E-10	6.1E-02

exit from nag\_nlin\_lsqr after 4 iterations.

final objective value = 0.8214877E-02

final residual norm = 9.1E-02

final gradient norm = 9.5E-10

The results are very similar to those obtained by `tutorial_ex9a`, but the column headed `Nfun` indicates that many more calls to `lsqr_fun` were required, outweighing the fact that each call to `lsqr_fun` does less work than in `tutorial_ex9a`.

## 9.2 Using a Structure for Option Setting

Algorithms and procedures for nonlinear optimization are usually quite elaborate, and have many adjustable parameters for controlling their behaviour. Each of the optimization procedures in NAG *f90* has an optional argument `control`, which is a *structure* in which the adjustable parameters are packaged together for convenience.

For `nag_nlin_lsqr_sol`, the type of the structure is `nag_nlin_lsqr_cntrl_wp` (see Section 8.2 for an explanation of the suffix `_wp`). Its components are *public*, and are documented in the specification of the derived type.

A utility procedure `nag_nlin_lsqr_cntrl_init` is provided to initialize the components of the structure to default values: it *must* be called before the structure is passed to `nag_nlin_lsqr_sol`, otherwise a fatal error will occur:

```
***** Fatal error reported by NAG Fortran 90 Library *****
Procedure nag_nlin_lsqr_sol           Level = 3   Code = 301
An input argument has an invalid value.
control has not been initialized by a call to nag_nlin_lsqr_cntrl_init.
***** Execution halted *****
```

After calling `nag_nlin_lsqr_cntrl_init`, you may set any of the components to different values by simple assignments.

Program `tutorial_ex9c` is a modified version of `tutorial_ex9a`; it uses the same module `tutorial_ex9a_mod`. All output from the routine is suppressed (`control%list = .FALSE.` and `control%print_level = 0`); and the requested accuracy is relaxed (`control%optim_tol = 0.0001_wp`). The program contains its own code to print results, including the residuals at the final solution.

```
PROGRAM nag_tutorial_ex9c
  USE tutorial_ex9a_mod, ONLY : lsqr_fun, u, v, w, y, wp
  USE tutorial_types, ONLY : nag_nlin_lsqr_cntrl_wp
  USE nag_nlin_lsqr, ONLY : nag_nlin_lsqr_sol, nag_nlin_lsqr_cntrl_init
  IMPLICIT NONE
  REAL (wp) :: f_sum_sq
  INTEGER :: i, m
  REAL (wp) :: x(3)
  REAL (wp), ALLOCATABLE :: f_vec(:)
  TYPE (nag_nlin_lsqr_cntrl_wp) :: control
  OPEN (4,file='tutorial_ex9a.dat')
```

```

      READ (4,*) m
      ALLOCATE (f_vec(m),u(m),v(m),w(m),y(m))
      READ (4,*) (u(i),v(i),w(i),y(i),i=1,m)
      READ (4,*) x
      CALL nag_nlin_lsq_cntrl_init(control)
      control%list = .FALSE.
      control%print_level = 0
      control%optim_tol = 0.0001_wp
      CALL nag_nlin_lsq_sol(lsq_fun,x,f_sum_sq,f_vec,control=control)
      WRITE (*,'(1x,a,3f8.2)') 'x =', x
      WRITE (*,*)
      WRITE (*,*) ' i          y(i)      residual'
      WRITE (*,'(1x,i3,2f10.2)') (i,y(i),f_vec(i),i=1,m)
END PROGRAM nag_tutorial_ex9c

```

With the same data file as before, the results will look like this:

```

x =      0.08      1.13      2.34

 i      y(i)      residual
 1      0.14      0.01
 2      0.18      0.00
 3      0.22      0.00
 4      0.25     -0.01
 5      0.29      0.00
 6      0.32      0.00
 7      0.35      0.00
 8      0.39      0.02
 9      0.37     -0.08
10      0.58      0.02
11      0.73      0.01
12      0.96      0.01
13      1.34      0.01
14      2.10      0.00
15      4.39     -0.01

```

The results show that with the computed values of  $x_1$ ,  $x_2$  and  $x_3$  (that is,  $\alpha$ ,  $\beta$  and  $\gamma$ ) the model (5) fits the data fairly well, except at the 9th observation.

## 10 Concluding Remarks

The aim of this Tutorial has been to help you to become familiar with NAG *f90*, and to develop confidence in using it.

If there are topics which are not yet covered, but which you would find helpful, please let us know, so that we can consider them for a future revision.