

Essential Introduction to the NAG Fortran 90 Library

This document is a summary of the essential features of the design of the NAG Fortran 90 Library. It assumes that you are familiar with Fortran 90. Sections marked with an asterisk () can be omitted at first reading unless specifically required. If you want a more informal and expansive introduction to the Library, or are unfamiliar with Fortran 90, please turn to the document **Tutorial for the NAG Fortran 90 Library**.*

Contents

1	Structure of the Library and its Documentation	0.1.2
2	Naming Scheme	0.1.3
3	Accessing the Library	0.1.3
4	Design and Documentation of Procedures	0.1.4
	4.1 Optional Arguments	0.1.4
	4.2 Assumed-shape Arrays	0.1.5
	4.3 Derived Types	0.1.5
	4.4 Generic Interfaces	0.1.6
	4.5 Key Arguments (*).....	0.1.6
	4.6 Procedure Arguments	0.1.6
5	Precision	0.1.7
	5.1 Precision of Procedures	0.1.7
	5.2 Precision of Derived Types	0.1.8
6	Memory Allocation	0.1.8
7	Error Handling	0.1.9
	7.1 Classification of Errors	0.1.9
	7.2 Default Error Handling.....	0.1.10
8	Example Programs	0.1.10
9	Library Details	0.1.11
	9.1 Releases of the Library	0.1.11
	9.2 Implementations of the Library.....	0.1.11
	9.3 The Procedure <code>nag_lib_ident</code>	0.1.12
10	Non-default Error Handling (*)	0.1.12
	10.1 The Argument <code>error</code>	0.1.12
	10.2 Examples	0.1.13
11	Portability and Precision (*)	0.1.14
12	Relationship to the NAG Fortran 77 Library (*)	0.1.15
13	Support from NAG (*)	0.1.15
14	User Feedback (*)	0.1.16
A	List of Abbreviations (*)	0.1.17
B	Contact Addresses (*)	0.1.19

1 Structure of the Library and its Documentation

The NAG Fortran 90 Library — known as NAG *fl90* — is divided into *chapters*, each covering one major area of numerical or statistical computing. For a complete list of chapters, see the **List of Contents** document.

Each chapter contains a number of *modules*, and each module typically contains a group of closely related *procedures* (subroutines or functions). Some modules also contain definitions of *derived types* or *named constants* which are required for use with the procedures in the module.

For example, Chapter 8 on **Curve and Surface Fitting** contains a module `nag_spline_1d`, which handles curve fitting problems using splines. This module contains seven procedures for fitting or evaluating a spline and so on; it also defines a derived type which is used to represent a spline. The chapter on **Curve and Surface Fitting** also contains three other modules: `nag_pch_interp` for piecewise cubic Hermite interpolation, `nag_spline_2d` for surface fitting with splines, and `nag_scatter_interp` for curve and surface fitting of scattered data.

The documentation has the same chapter structure as the Library. There is

- a **Chapter Introduction** document for each chapter, and
- a **module document** for each module.

The module document is the basic unit of documentation. Each module document has an index number of the form *c.m*, where *c* is the chapter number and *m* is the module number within the chapter. The index number defines the order in which the module documents appear in the Manual.

Thus, for example, the module documents in Chapter 8 are numbered as follows:

- 8.1 – Module `nag_pch_interp`
- 8.2 – Module `nag_spline_1d`
- 8.3 – Module `nag_spline_2d`
- 8.4 – Module `nag_scatter_interp`

Each module document contains the following sections and subsections:

- Introduction (optional)
- Procedure Specifications
 - 1. Description
 - 2. Usage
 - 3. Arguments
 - 4. Error Codes
 - 5. Examples of Usage
 - 6. Further Comments (optional)
- Derived Type Specifications (optional)
 - 1. Description
 - 2. Type Definition
 - 3. Components
- Examples
- Additional Examples (optional)
- Mathematical Background (optional)
- References (optional)

A Keywords in Context (KWIC) index and an index based upon GAMS classification are provided in the printed manual to help you find a module or procedure to meet your needs.

2 Naming Scheme

All modules, procedures, derived types and named constants provided by the Library have names beginning with the prefix ‘`nag_`’.

All names are constructed from components separated by underscores. Many of the components are abbreviations which come from a list of abbreviations given in Appendix A. These abbreviations are also used in the names of dummy arguments and components of derived types.

The names have been designed, as far as possible, to be reasonably compact, intelligible, memorable and indicative of the essential function of the named entity. But obviously these are conflicting criteria about which individuals may have widely varying views.

Note that it is always possible to rename an entity from the Library when you access it in a `USE` statement, as described in Section 3.

3 Accessing the Library

Suppose that you have found a module which deals with your problem.

Any of your program units which refer to procedures (or other entities) from that module *must* contain a `USE` statement for that module.

For example, if you wish to fit a spline curve, you will need to use the module `nag_spline_1d` (8.2). If you wish to call the procedure `nag_spline_1d_auto_fit` from that module, your program unit must contain the statements:

```
USE nag_spline_1d
. . .
CALL nag_spline_1d_auto_fit( . . . )
```

The required `USE` statement is given in the **Usage** section of each procedure specification.

The `USE` statement gives access to an explicit interface for each documented procedure in the module. Therefore mismatches between actual and dummy arguments can be detected at compile time.

The `USE` statement also gives access to all entities in the Library that are required for the use of the procedures in the module, for example, definitions of derived types and elements of the infrastructure of the Library, such as those concerned with error handling (see Section 7) or deallocation of memory (see Section 6).

If within a single program unit you call procedures from more than one module, you will need to include more than one `USE` statement.

Access to the Library is carefully controlled by use of `PRIVATE` and `PUBLIC` statements in the library modules, so that only the documented entities are accessible.

Note that it is good practice to document which entities you access through a `USE` statement, by naming them in an `ONLY` clause, for example:

```
USE nag_spline_1d, ONLY : nag_spline_1d_auto_fit
```

You may also rename a procedure (or other entity) in the `USE` statement if you wish:

```
USE nag_spline_1d, ONLY : fit => nag_spline_1d_auto_fit
. . .
CALL fit( . . . )
```

4 Design and Documentation of Procedures

Several features of the Fortran 90 language have been employed in NAG *fl90* to increase the power and ease of use of library procedures, in particular:

- optional arguments
- assumed-shape arrays
- derived types
- generic interfaces

Sections 4.1 to 4.5 describe how these features are used and documented.

Section 4.6 focuses on the design and documentation of *procedure arguments*, that is, arguments which are themselves procedures (subroutines or functions), to be supplied by users.

4.1 Optional Arguments

Most procedures in the Library have several optional arguments. In particular, the argument `error`, which controls error handling, is always optional (see Section 7).

As a general rule, mandatory arguments are used to supply the essential data to a procedure and to receive the essential results. Optional arguments are used, for example:

- to define additional aspects of the problem (for example, constraints)
- to supply non-default values for arguments which control the execution of the algorithm (for example, requested accuracy, or maximum number of iterations)
- to request additional results to be computed by the algorithm (for example, eigenvectors in addition to eigenvalues, or an estimate of the accuracy actually achieved)
- to request information about the performance of the algorithm (for example, number of iterations required)

If an optional input argument is not supplied, then the procedure uses a default value (which of course is documented), or in some cases may take some more general default action.

In the procedure specifications, there are separate subsections for **Mandatory Arguments** and **Optional Arguments**.

It is a convention of NAG *fl90* that optional arguments *must* be passed by *keyword*, for example:

```
CALL nag_quad_1d_gen( f, a, b, result, abs_acc=zero, rel_acc=0.00001_wp )
```

and *not*

```
CALL nag_quad_1d_gen( f, a, b, result, zero, 0.00001_wp )
```

The reason for this convention is that the order in which the optional arguments are listed in the documentation is *not* necessarily the same as their positional order in the argument list. Additional optional arguments may be introduced at future releases. (Calls in which optional arguments are passed by position may work correctly, but *NAG does not support this mode of use* and does not guarantee that such calls will continue to work indefinitely.)

The fact that an argument is optional does not necessarily mean that you have complete freedom to supply it or not. Some library procedures impose constraints on the combinations of optional arguments that are allowed to be present. These constraints are stated in the documentation, and checked at run-time by the library procedures.

See also Section 4.6 on optional arguments of (user-supplied) procedure arguments.

4.2 Assumed-shape Arrays

All array arguments of library procedures are *assumed-shape* arrays, apart from a few which are array pointers. An actual argument which corresponds to an assumed-shape array can be a whole array or an array section; or if it is an input argument, it can be an array expression (which includes an array constructor or the result of an array-valued function). In all cases the actual argument must have the *exact* shape required by the problem; in other words it must have the correct extent in each dimension.

For example, the procedure `nag_gen_lin_sol` (5.1) solves a (square) system of linear equations $Ax = b$ of order n . The supplied rank-2 array `a` which holds the matrix A must have exactly n rows and n columns, and the rank-1 array `b` which holds the right-hand side must have exactly n elements. If necessary, the array-section notation can be used to achieve this, for example:

```
CALL nag_gen_lin_sol( a(1:n,1:n), b(1:n) )
```

Library procedures normally determine the dimensions of the problem from the shapes of the supplied arrays (thus reducing the number of arguments): in the above example, n is determined from the first (or leading) dimension of `a`. Library procedures check that the shapes of array arguments are consistent with one another: for example, `nag_gen_lin_sol` checks that `a` is square, and that the number of elements in `b` is the same as the number of rows or columns in `a`.

Array arguments are specified in the documentation in the following style:

`x(n)` — real(kind=*wp*), intent(...)

`y(m,n)` — real(kind=*wp*), intent(...)

(See Section 5 for the documentation of real types.) Here `x` must be a rank-1 array which must have exactly n elements, and `y` must be a rank-2 array with exactly m rows and n columns; m and n will have already been defined in the description of the problem that the procedure solves.

4.3 Derived Types

Some library procedures have arguments which are *structures*, that is, objects of a *derived type* which is defined by the Library. The definition of the derived type is accessible from the same module as the procedure.

One use of structures is to communicate data from one library procedure to another, or between repeated calls of the same procedure. In such cases, the components of the type are usually *private*, so that the data is protected from accidental corruption. Additional service procedures are provided to access some of these components if that is likely to be useful.

Another use of structures is to package together items of data which serve a similar function: for example, several parameters which control an optimization algorithm are grouped together in an argument `control`; in these cases, the components are public.

If a derived type contains real or complex components, then its name is precision dependent (see Section 5.2).

The specification of a derived type is given in a separate section of the module document for the module(s) in which it is used, with the exception of the derived type `nag_error`, which is used throughout the Library, and is described in Section 10.1 and in the module document `nag_error_handling` (1.2).

Many derived types defined by the Library have a special component which allows a library procedure to check whether a structure of this type has been initialized.

See also Section 6 concerning the allocation of memory to components of types defined by the Library.

4.4 Generic Interfaces

Each procedure in the Library is made accessible through a *generic interface*. The documented procedure name is the generic name. With few exceptions, each generic interface provides access to more than one specific version of the procedure. These specific versions may differ according to the following properties of their arguments.

precision: see Section 5 for more details. This is the commonest type of genericity.

data type: for example, an argument may be real or complex.

rank: for example, an argument may be a scalar or an array.

Each procedure specification describes the generic interface as if it were a single procedure with generic capabilities. For example, a single generic procedure is provided to solve real or complex systems of linear equations with one or several right-hand sides.

The alternatives are documented in the following style:

$$\mathbf{b}(n) / \mathbf{b}(n, r) \text{ — real(kind=wp) / complex(kind=wp), ...}$$

This means that the argument \mathbf{b} may be either a rank-1 array of shape (n) or a rank-2 array of shape (n, r) ; its type may be either `real(kind=wp)` or `complex(kind=wp)`. (See Section 4.2 for the documentation of array arguments and Section 5 for the documentation of real and complex types.)

In addition (except in the simplest cases), the properties of the different interfaces are summarised in an extra **Interfaces** subsection of the **Usage** section of the procedure specification.

You do not normally need to know the specific procedure names; you may however see them in loader maps, traceback or debug information.

4.5 Key Arguments (*)

This section describes a convention of NAG *f90* which is used only in some procedures for linear algebra and Fourier transforms.

These procedures have arguments, referred to as ‘keys’, whose sole function is to distinguish between different specific versions of a generic procedure, in cases where the other mandatory arguments are insufficient to do this.

For example, most of the procedures for linear algebra in the Library are generic procedures for real or complex data, and the different versions are distinguished by the type of one or more of the mandatory arguments. However, in the procedure `nag_bidiag_svd` (6.3), it is only the optional arguments that can differ in type between real and complex. Therefore this procedure has a mandatory ‘key’ argument, and real and complex problems are distinguished thus:

```
CALL nag_bidiag_svd( nag_key_real, uplo, d, e, vt=vt ) ! vt is real
```

```
CALL nag_bidiag_svd( nag_key_cplx, uplo, d, e, vt=vt ) ! vt is complex
```

Here `nag_key_real` and `nag_key_cplx` are named constants defined by the Library, which are accessible from the same module as the procedure `nag_bidiag_svd`; you must supply one of them as the first actual argument to the procedure.

All key arguments in the Library have the dummy argument name `nag_key`, and the values supplied by the Library all have names beginning with the prefix ‘`nag_key_`’. As a user, all you need to do is to supply the correct name. (Formally, key arguments are named constants, each of a different derived type, defined by the Library.)

4.6 Procedure Arguments

Some NAG *f90* procedures require you to supply a Fortran 90 procedure (subroutine or function) as an argument. It is referred to as a *procedure argument*. It may be optional.

A procedure argument is documented in a similar style to a library procedure, with its specification enclosed in a box. However, because you must *write* the procedure, rather than *call* it, there are some important differences. Here is a simple example:

f — function

f must return the value of the integrand f at a given point.

```
function f(x)

real(kind=wp), intent(in) :: x(:)

    Shape: x has shape (n).

    Input: the co-ordinates of the point at which the integrand  $f$  must be evaluated.

real(kind=wp) :: f

    Result: f must contain the value of  $f$  at the point with co-ordinates  $x(i)$ , for  $i = 1, 2, \dots, n$ .
```

This example illustrates the following general points.

1. The specification gives a precise Fortran 90 declaration of the attributes of each argument, which can be copied exactly into your code (except for the symbol `wp` which is explained in Section 5).
2. Array arguments are specified as assumed-shape arrays.

The dimensions of the actual arguments (passed by the library procedure) are usually specified under the subheading *Shape*, in terms of the dimensions of the problem. In the above example, n is a problem-dependent value (the number of dimensions of the integrand): within the code of your procedure `f`, n may have a known fixed value, or its value may be accessible from elsewhere in your calling program; alternatively it can be determined as the value of `SIZE(x)`.

However, in some procedure arguments, the dimensions of assumed-shape arrays are determined within the library procedure by the details of the algorithm, not by the problem: in such cases the actual dimensions of the supplied arrays can be determined (if needed) within the code of the procedure argument *only* by using the intrinsic function `SIZE`.

3. If an argument has the attribute `intent(in)`, its value must not be changed by the user-supplied procedure.
4. The result returned by a function argument is described in the same style as the arguments, with the special subheading *Result*.

Some procedure arguments have *optional arguments*.

Optional arguments allow more flexibility in the ways in which a procedure argument can be called by a library procedure: for example, the presence of an optional argument can be used to indicate that the procedure argument is required to compute and return additional information.

The code of the procedure argument *must test* for the presence of each optional argument, using the Fortran 90 intrinsic function `PRESENT`, and must take appropriate action depending on the result of the test.

5 Precision

5.1 Precision of Procedures

On your machine, NAG *fl90* may contain versions of procedures in *double precision* or *single precision*, or possibly both. In the double precision version of a library procedure all `real` or `complex` arguments are in double precision; in a single precision version, they are all in single precision. On some systems, there may be versions in other precisions as well (e.g., quadruple).

Each procedure has a *generic* name (see Section 4.4) which covers all available precision versions. The compiler will determine which precision-specific version of the procedure to call, according to the precision of the real or complex arguments that you have used. You may even call different specific versions of the same procedure within one program unit.

If there is no version available corresponding to the precision of your arguments, the compiler will report an error. *All* real and complex arguments in a call to a library procedure must be in the same precision; if they are in mixed precision, the compiler will report an error.

You need to know which precisions are available in the Library on your machine. This information can be obtained by calling the NAG *f90* procedure `nag_lib_ident` in the module `nag_lib_support` (see Section 9.3).

In the procedure specifications, the type of a real or complex argument is given as

`real(kind=wp)` or `complex(kind=wp)`.

Here *wp* denotes a kind value, which needs to be interpreted in a way that fits the precision of your program and the style in which you have coded it. (The symbol *wp* stands for ‘working precision’; the different typeface is a reminder that its interpretation may vary.)

Suppose that on your machine the Library contains procedures in both double and single precision:

for double precision, ‘`real(kind=wp)`’ is equivalent to ‘`real(kind=kind(1.0D0))`’ or ‘double precision’;

for single precision, ‘`real(kind=wp)`’ is equivalent to ‘`real(kind=kind(1.0))`’ or ‘real’.

However, there are advantages in using a named constant (such as `wp`) to define the kind values for all real or complex data in your program; for further guidance, see Section 11 on **Portability and Precision**.

5.2 Precision of Derived Types

Derived types cannot be parameterized with a kind value; this is a limitation of the Fortran 90 language. Derived types defined by the Library with real or complex components must have distinct names for each available precision. The following convention is used.

- In double precision, the type name has the suffix `_dp`; in single precision, it has the suffix `_sp`; for example, `nag_seed_dp` and `nag_seed_sp`. (For other precisions, if available, see the Users’ Note for your implementation.)
- The name used in the documentation has the suffix `_wp`, for example `nag_seed_wp`; note that the suffix is in a different typeface to remind you that its interpretation may vary.

6 Memory Allocation

Many library procedures allocate memory for internal workspace; such workspace is always deallocated before exit from the procedure.

Some library procedures have dummy arguments which are *array pointers*. Array pointers are used when the precise amount of memory required can only be determined by the procedure in the course of computation. Fortran 90 does not allow allocatable arrays as dummy arguments, so array pointers are used instead.

When you call the procedure, you must supply a pointer of the correct type and rank. In order to avoid problems with potential memory leaks, the pointer *should not* be associated as the association will be lost. The procedure allocates the required amount of memory to the pointer, and on return, you may use the pointer to refer to the results.

If and when the results are no longer needed, you can deallocate the memory in your calling program, using a `DEALLOCATE` statement (this will keep memory usage to a minimum). (It will be deallocated automatically at the end of your program.) If you pass the same pointer repeatedly to several calls of a library procedure, you should deallocate it between calls; otherwise, the previously allocated memory will become inaccessible (a ‘memory leak’).

For example, you may wish to compute the eigenvectors corresponding to the eigenvalues in a given interval; how many such eigenvalues there are may not be known in advance, so you are asked to supply array pointers to receive the eigenvalues and eigenvectors. The following fragment of code illustrates the pattern:

```
REAL (KIND=wp), POINTER :: lambda(:), z(:, :)
. . .
CALL nag_sym_eig_sel( uplo, a, lambda, z=z, . . . )
. . .
DEALLOCATE (lambda, z)
```

A similar situation occurs when an argument is a structure with components which are array pointers. Again, memory is allocated by the procedure, and it is your responsibility to deallocate it if and when it is no longer needed. A generic procedure `nag_deallocate` (1.1) is provided for this purpose: it has one mandatory argument, namely the structure (a scalar). The specification of each derived type states whether it has components which may need to be deallocated using `nag_deallocate`.

In the following fragment of code, `spline` is a structure which represents a spline curve. Memory is allocated to the structure by the call to `nag_spline_1d_lsq_fit`; the structure is passed to `nag_spline_1d_eval`, and the memory is deallocated by a call to `nag_deallocate`:

```
TYPE (nag_spline_1d_dp) :: spline
. . .
CALL nag_spline_1d_lsq_fit( x, f, knots, spline )

CALL nag_spline_1d_eval( spline, x, s )

CALL nag_deallocate( spline )
```

7 Error Handling

7.1 Classification of Errors

NAG *f*90 procedures may detect and report various kinds of error, failure or warning conditions. 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).

Each error is given a numeric code. Warnings (level 1) have codes in the range 100–199; failures (level 2) have codes in the range 200–299; fatal errors (level 3) have codes in the range 300–399.

Standard error codes are used for the common types of fatal errors.

301: an argument has an invalid value

302: an array argument has an invalid shape

303: array arguments have inconsistent shapes

304: an optional argument is present when it is not allowed (for example, because certain combinations of optional arguments are forbidden)

305: an optional argument is absent when it must be present (for example, because certain pairs of optional arguments must be present together)

You do not need to remember the meanings of these codes: they are repeated, where relevant, in the specifications of the procedures. The error messages which are output by the procedures give full details of the particular arguments which have caused the errors.

To control the way in which any of these errors are handled, all NAG *f190* procedures have an *optional* argument **error** (except for a few which cannot give rise to any error condition).

7.2 Default Error Handling

If the optional argument **error** is *omitted*, then the library procedure takes the following *default* action.

- If no error is detected, it returns control to your calling program.
- 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 output unit and halts execution of the program.

The default action may not always be suitable: for example, you may wish to take corrective action after a computational failure, or you may wish to halt execution after a warning. For details, see Section 10 on **Non-default Error Handling**.

8 Example Programs

Each module document contains one or more complete example programs, together with data (if required) and typical results, illustrating a simple use of the procedures in the module (often in combination).

These example programs are distributed to sites as source code, and are designed so that they can easily be modified to solve similar simple problems.

For procedures with a high degree of genericity (see Section 4.4), the example programs published in the documentation may not illustrate all the different specific versions of the procedure. However, example programs covering each specific version are provided with the software, and are listed in the **Additional Examples** section of each module document.

The published example programs may illustrate the use of only some of the optional arguments of a procedure. The **Examples of Usage** section in each procedure specification presents additional fragments of code, if necessary, to supplement the complete example programs in the module document.

The following programming conventions are used in the example programs.

- Fortran 90 statement keywords and intrinsic function names appear in upper case; all other names are in lower case.
- The published example programs are all in double precision. To convert them to single precision, you need only change the definition of the named constant **wp** and the names of any precision-dependent derived types defined by the Library; see below for details.
- An integer named constant **wp** is used as the kind value for all real and complex entities. Type declarations for real and complex data appear simply as **REAL (wp)** and **COMPLEX (wp)**, rather than the equivalents **REAL (KIND=wp)** and **COMPLEX (KIND=wp)**.
- The constant **wp** is defined to be the kind value for double precision, in the following statement:

```
INTEGER, PARAMETER :: wp = KIND(1.0D0)
```

To change the precision to single, simply change 1.0D0 to 1.0 (or 1.0E0) in this statement.

- All **USE** statements have an **ONLY** qualifier, whose purpose is to document which entities are accessed from the module.
- If the program uses a precision-dependent derived type defined by the Library (see Section 5.2), the type is renamed when it is accessed in a **USE** statement, so that the suffix is changed from **_dp** to **_wp**. For example:

```
USE nag_rand_contin, ONLY :: nag_seed_wp => nag_seed_dp
```

All other references to this type use the local name `nag_seed_wp`. If you are converting the program to single precision, you must change the name `nag_seed_dp` in the `USE` statement to `nag_seed_sp`:

```
USE nag_rand_contin, ONLY :: nag_seed_wp => nag_seed_sp
```

- Allocatable arrays are used in many programs for passing as array arguments to NAG *f90* procedures. Integers which specify the size of the problem are read from a data file, and the arrays are then allocated with the required shape.
- All allocatable arrays and array pointers are explicitly deallocated at the end of the program, using a `DEALLOCATE` statement; all structures that have had memory allocated to them by NAG *f90* procedures are deallocated by a call to `nag_deallocate`. See Section 6. Of course, it is not necessary to deallocate storage immediately before the end of a main program, but these statements are included in case the code is copied into part of a much larger program where it may be important to avoid memory leaks.
- In calls to NAG *f90* procedures, arguments are called by keyword if and only if they are optional. Note that it is a convention of NAG *f90* that all optional arguments *must* be called by keyword (see Section 4.1).
- If the library procedure has an argument which is a user-supplied procedure, the actual procedure is always contained in a user-supplied module. This ensures that an explicit interface for the user-supplied procedure is automatically available, and in some example programs allows global data to be shared between the user's procedure and the main program (without using `COMMON`).
- Named constants `nag_std_in` and `nag_std_out` (which are obtained from the module `nag_examples_io`) are used as the unit numbers for input and output, respectively.
- A one-line heading appears (for purposes of identification) as the first record of each data file; it is skipped by a `READ` statement with no input list. Similarly a one-line heading is output as the first record of each results file.

9 Library Details

9.1 Releases of the Library

Periodically, the Library and its documentation will be updated to a new *release*; new modules and procedures will be added, and corrections or improvements may be made to existing code and documentation.

The code may also be updated between releases to an intermediate maintenance level, in order to incorporate corrections. Maintenance levels are indicated by a letter following the release number, for example 1A, 1B.

9.2 Implementations of the Library

Distinct *implementations* of the Library are provided for different systems. Each implementation is distributed to sites as a tested, compiled library. An implementation is usually specific to a range of machines (for example, Sun 4); it may also be specific to a particular operating system, Fortran 90 compiler, or compiler option.

The documentation supports all implementations of the Library; a small amount of implementation-dependent information is provided in a separate **Users' Note** for each implementation.

The Users' Note states in particular which precisions are supported (see Section 5).

9.3 The Procedure `nag_lib_ident`

To find out which release, which implementation, and which precision(s) of the Library are available at your site, you can run a program which calls the procedure `nag_lib_ident` in the module `nag_lib_support` (1.1). This is a subroutine with no arguments, which writes the information onto the standard output unit.

The following program is all that is needed to call `nag_lib_ident`:

```
USE nag_lib_support
CALL nag_lib_ident
END
```

and here is an example of the output (which will of course vary from one implementation of the Library to another):

```
*** 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 ***
```

10 Non-default Error Handling (*)

The information given here should be sufficient for handling errors in almost all situations.

10.1 The Argument `error`

All library procedures (except for a few which cannot give rise to any error condition) have an optional argument `error`. If the default error handling action described in Section 7 is not suitable, you must supply the argument `error`. It is a *structure* of a derived type `nag_error`, defined by the Library. A complete specification of this derived type is given in the document for the module `nag_error_handling` (1.2), but you do not normally need to know more than is given here.

The argument `error` is an `intent(inout)` argument which serves two purposes.

1. It allows you to specify what action a library procedure should take should it detect an error.
2. It reports the state of the library procedure (either success or an error-condition), and returns the text of any error message, to your calling program.

If you supply the argument `error`

- you *must initialize* it before calling the procedure. To initialize it, you must call the procedure `nag_set_error`. For a complete specification of this procedure, see the module document `nag_error_handling` (1.2). The following is all you normally need to know. The procedure has one mandatory output argument, a scalar of type `nag_error` (the structure to be initialized), and three optional input arguments:

halt_level — integer, `intent(in)`, optional

Input: specifies that the program is to be halted if an error of level \geq `halt_level` is detected.

Default: `halt_level = 2`.

print_level — integer, `intent(in)`, optional

Input: specifies that an error message is to be printed if an error of level \geq `print_level` is detected.

Default: `print_level = 1`.

unit — integer, intent(in), optional

Input: specifies the unit on which any error message is to be printed.

Default: `unit = nag_std_out` (the default output unit in your implementation of the Library).

- you *must test* one of the integer components `code` or `level` of the structure `error` on return to your calling program; if you do not, your program may continue computing with invalid or undefined results. The components `code` and `level` are set by the procedure as follows:
 - = **0**: the procedure has exited successfully;
 - ≠ **0**: the procedure has detected an error: the value of `code` indicates the nature of the error, as stated in the specification of the procedure, and the value of `level` indicates its classification.

If you do not call `nag_set_error` to initialize the structure, this will almost certainly be reported by the procedure, or by the system, but there is a small possibility that your program may handle errors in an unpredictable way.

You do not normally need to include any additional `USE` statements in your calling program in order to use the derived type `nag_error` and the procedure `nag_set_error`; they are always accessible through any `USE` statement which gives access to a library procedure with the optional argument `error`. (They are also accessible through the module `nag_error_handling`.)

10.2 Examples

The following examples use the procedure `nag_gen_lin_sol` (5.1), which may exit with the following error codes:

301, 302, 303: fatal errors, due to invalid arguments;

201: a failure, because exact singularity has been detected;

101: a warning that approximate singularity (extreme ill conditioning) has been detected.

All examples (except the first) assume that a structure `error` has been declared of type `nag_error`, using statements such as the following:

```
USE nag_gen_lin_sys, ONLY : nag_gen_lin_sol, nag_error, nag_set_error
```

```
TYPE (nag_error) :: error
```

- Default action (see Section 7.2): print an error message after a warning, failure or fatal error; halt program after failure or fatal error.

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

- To take special action after a warning, without changing the default conditions for halting or printing an error message; note that you *must* still call `nag_set_error` to initialize the structure.

```
CALL nag_set_error( error )
```

```
CALL nag_gen_lin_sol( a, b, error=error )
```

```
IF (error%level==1) THEN
! code to handle the warning
END IF
```

- To halt execution (with an error message) after a warning (as well as after failures and fatal errors):

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

```
CALL nag_gen_lin_sol( a, b, error=error )
```

- To continue execution after a failure or warning, without printing an error message, and to take corrective action (different action is assumed for different error codes):

```
CALL nag_set_error( error, halt_level=3, print_level=3 )
CALL nag_gen_lin_sol( a, b, error=error )
SELECT CASE (error%code)
CASE (0)
! code to handle successful exit
CASE (101)
! code to handle error code 101
CASE (201)
! code to handle error code 201
END SELECT
```

- Default action, except that any error message is to be printed on unit 20:

```
CALL nag_set_error( error, unit=20 )
CALL nag_gen_lin_sol( a, b, error=error )
```

The document for the module `nag_error_handling` (1.2) gives complete example programs illustrating various aspects of error handling, and examples of the text of error messages from the Library.

11 Portability and Precision (*)

This section offers some advice in case you wish to port a program which calls NAG *f*90 procedures from one system to another — assuming that the Library is available on each system — or to run your program in more than one precision on the same system. The example programs in the module documents illustrate the style of programming which is described here.

Calls to NAG *f*90 procedures do *not* need to be modified, because the generic names will be matched with the required specific version, according to the kind values of the real or complex arguments.

The Fortran 90 standard does not prescribe what kind values should be used for ‘double precision’ or ‘single precision’ (for example, it could be 2 and 1, 8 and 4, or 64 and 32). In other words, ‘double precision’ on one machine may offer roughly the same precision as ‘single precision’ on another. Therefore it is good programming practice always to use a named integer constant as the kind value for all your real and complex data; this is true whether or not you call NAG *f*90 procedures. If this constant is defined in a module, this is the only statement that may need to be changed should you need to use a different kind value.

```
MODULE my_precision ! precision-dependent
  INTEGER, PARAMETER :: wp = KIND(1.0D0) ! kind value for double precision
END MODULE my_precision

PROGRAM portable
  USE my_precision, ONLY : wp
  USE nag_gamma_fun, ONLY : nag_gamma
  REAL (wp) :: x, y
  . . .
  y = nag_gamma( x )
  . . .
END PROGRAM portable
```

If you can define your required precision in absolute terms, you can even make the module `my_precision` portable, by writing, say:

```
MODULE my_precision ! gives at least 10 digits of precision
  INTEGER, PARAMETER :: wp = SELECTED_REAL_KIND(10)
END MODULE my_precision
```

There is one complication that arises if your program calls NAG *f*90 procedures which require the use of precision-dependent derived types, that is, types defined by the Library with real or complex components. Derived types cannot be parameterized with a kind value in the same way as the intrinsic real and complex types, as was mentioned in Section 5.2; the Library provides distinct types for each

available precision, and you may need, for example, to use the double precision type `nag_seed_dp` on one system, but the single precision type `nag_seed_sp` on another.

A convenient practice is to use a neutral name (say, `nag_seed_wp`) throughout your program, and to define this name in a separate module in only one place.

```

MODULE my_precision
  INTEGER, PARAMETER :: wp = KIND(1.0D0) ! kind value for double precision
  USE nag_rand_util, ONLY : nag_seed_wp => nag_seed_dp ! double precision type
END MODULE my_precision

PROGRAM portable
  USE my_precision , ONLY : wp, nag_seed_wp
  USE nag_rand_contin, ONLY : nag_rand_uniform
  REAL (wp) :: x
  TYPE (nag_seed_wp) :: seed1
  . . .
  x = nag_rand_uniform( seed1 )
  . . .
END PROGRAM PORTABLE

```

12 Relationship to the NAG Fortran 77 Library (*)

NAG *f*l90 is a distinct library from the NAG Fortran 77 Library, although it uses essentially the same algorithms. You may call routines from both libraries within the same program unit — provided of course that implementations of both libraries are available for your system and compiler. Implementations of the Fortran 77 Library are already available for use with a Fortran 90 compiler on several systems, enhanced by the provision of modules of interface blocks which enable calls to Fortran 77 library routines to be checked at compile time.

Thus the Fortran 77 Library may be used as a source of algorithms that are not yet provided in NAG *f*l90. The two libraries are available together as a single product with the name NAG FL90plus.

The document **Conversion from the NAG Fortran 77 Library** gives advice on replacing calls to NAG Fortran 77 library routines with calls to NAG *f*l90 procedures (if an equivalent procedure exists).

13 Support from NAG (*)

(a) Contact with NAG

Queries concerning this document or the implementation generally should be directed initially to your local Advisory Service. If you have difficulty in making contact locally, you can contact NAG directly at one of the addresses given in Appendix B.

(b) NAG Response Centres

The NAG Response Centres are available for general enquiries from all users and also for technical queries from sites with an annually licensed product or support service. The Response Centres are open during office hours, but contact is possible by fax, e-mail and phone (answer machine) at all times. When contacting a Response Centre please quote your NAG site reference and NAG product code.

(c) NAG Websites

The NAG websites are an information service providing items of interest to users and prospective users of NAG products and services. The information is reviewed and updated regularly and includes implementation availability, descriptions of products, downloadable software, product documentation and technical reports. The NAG websites can be accessed at

<http://www.nag.co.uk/>

or

<http://www.nag.com/> (in North America)

or

<http://www.nag-j.co.jp/> (in Japan)

(d) NAG Electronic Newsletter

If you would like to be kept up to date with news from NAG you may want to register to receive our electronic newsletter, which will alert you to special offers, announcements about new products or product/service enhancements, case studies and NAG's event diary. To register visit the NAG Ltd website or contact us at nagnews@nag.co.uk.

14 User Feedback (*)

Many factors influence the way NAG's products and services evolve and your ideas are invaluable in helping us to ensure that we meet your needs. If you would like to contribute to this process we would be delighted to receive your comments via feedback@nag.co.uk. Alternatively feel free to contact the appropriate NAG Response Centre who will be happy either to record your comments or to send you a printed copy of the survey.

A List of Abbreviations (*)

1d	one-dimensional	err	error
1v	one-variable	eval	evaluation
2d	two-dimensional	exp	exponential
3d	three-dimensional	gs	Gaussian
abs	absolute	fac	factorize, factorization
acc	accuracy	feas	feasibility
ampl	amplitude	fft	FFT (fast Fourier transform)
acf	autocorrelation function	form	format
alg	algorithm	freq	frequency
arb	arbitrary	fun	function
auto	automatic	fwd	forward
bidia	bidagonal	helm	Helmholtz
binom	binomial	herm	Hermitian
bivar	bivariate	hess	Hessian
bnd	band, banded	hilb	Hilbert
brk	break	hyp	hyperbolic
bwd	backward	hypergeo	hypergeometric
canon	canonical	gen	general
cg	conjugate gradient	ilu	incomplete LU factorization
cheb	Chebyshev	incompl	incomplete
chisq	chi-square	indx	index
chol	Cholesky	inf	infinite
cmplx	complex	infeas	infeasible, infeasibility
coeff	coefficients	init	initialize
coh	coherence	int	integer
col	column	intg	integrand, integral
comm	communication	interp	interpolation
comp	component(s)	intvl	interval
con	constraint(s)	inv	inverse
cond	condition	invar	invariant
conj	conjugate	ip	integer programming
contin	continuous	iter	iteration
conv	convolution	ivp	IVP (initial value problem)
coo	coordinate sparse form	jac	jacobian
coord	coordinate system	len	length
correc	correction	lev	level
correl	correlation	lib	library
cos	cosine	lim	limit
cov	covariance	lin	linear
cntrl	control(s)	loc	local
csc	compressed sparse column form	log	logarithm, logarithmic
csr	compressed sparse row form	lsq	least-squares
decomp	decompose	mat	matrix
deg	degree	max	maximum
deriv	derivative	md	multidimensional
det	determinant	min	minimum
dev	deviation	mintg	multiple integrands
diag	diagonal	miss	missing
diff	differences	mom	moment
dist	distribution	monit	monitor, monitoring
dupl	duplicate	monot	monotonic
eig	eigenproblem	mult	multi-, multiple
ell	elliptic	msg	message
eqn	equation	mv	multivariate

neg	negative	svd	SVD (singular value decomposition)
nlin	nonlinear	sys	system
nlp	nonlinear programming	sym	symmetric
nsym	nonsymmetric	tap	tapered
num	number	term	termination
obj	objective	thresh	threshold
ode	ODE (ordinary differential equation)	tol	tolerance
optim	optimality	trans	transpose
orth	orthogonal	tri	triangular
pacf	partial autocorrelation function	tridiag	tridiagonal
parab	parabolic	trig	trigonometric
part	partial	ts	time series
pch	piecewise cubic Hermite	tsa	time series analysis
pde	PDE (partial differential equation)	util	utility
pdf	PDF (probability distribution function)	uv	univariate
perm	permutation	val	value(s)
polynom	polynomial	var	variable(s); variance; variate; varying
pos	positive (as in positive definite)	vec	vector(s)
prec	preconditioned, preconditioning	wt	weight
prin	principal		
prob	probability; problem		
prod	product		
pt	point		
ptr	pointer		
qp	quadratic programming		
qtr	quarter		
quad	quadrature		
rand	random		
rcond	reciprocal condition number		
rec	record		
rect	rectangular		
reduc	reduction		
ref	reference		
reg	regression		
rel	relative		
resid	residual		
rf	response function		
rhs	right hand side		
rk	Runge–Kutta		
rms	root mean square; residual mean square		
scat	scattered		
sel	selected		
sig	significance		
sin	sine		
sing	singular		
sol	solution		
sqrd	squared		
sqrt	square root		
ssor	symmetric successive over relaxation		
stats	statistics		
std	standard		
struct	structure		
subint	subintervals		
subrgn	subregion		
sup	super		

B Contact Addresses (*)

NAG Ltd

Wilkinson House
Jordan Hill Road
OXFORD OX2 8DR
United Kingdom

Tel: +44 (0)1865 511245
Fax: +44 (0)1865 310139

NAG Ltd Response Centre
email: infodesk@nag.co.uk

Tel: +44 (0)1865 311744
Fax: +44 (0)1865 311755

NAG Inc

1400 Opus Place, Suite 2000
Downers Grove
IL 60515-5702
USA

Tel: +1 630 971 2337
Fax: +1 630 971 2706

NAG Inc Response Centre
email: infodesk@nag.com

Tel: +1 630 971 2345
Fax: +1 630 971 2346

NAG GmbH

Schleissheimerstrasse 5
85748 Garching
Deutschland
email: info@naggmbh.de

Tel: +49 (0)89 320 7395
Fax: +49 (0)89 320 7396

Nihon NAG KK

Yaesu Nagaoka Building No. 6
1-9-8 Minato
Chuo-ku
Tokyo
Japan
email: help@nag-j.co.jp

Tel: +81 (0)3 5542 6311
Fax: +81 (0)3 5542 6312