NAG Library

How to Use the NAG Library and its Documentation

Contents

1 Library Identification . . . 3

2 How to Find a NAG Library Routine . . . 3

3 How to Use the NAG Library . . . 4

3.1 Structure of the Library . . . 4
3.1.1 Experimental Routines . . . 4
3.1.2 Long Names for Library Routines . . . 5
3.2 General Advice . . . 5
3.3 Programming Advice . . . 5
3.3.1 Alternative Routine Names . . . 6
3.3.2 The NAG Fortran Environment . . . 6
3.3.3 Direct and Reverse Communication Routines . . . 7
3.4 Error Handling and the Argument ifail . . . 7
3.4.1 Errors, Failure and Warning Conditions . . . 7
3.4.2 The ifail Argument . . . 7
3.4.3 Hard Fail Option . . . 8
3.4.4 Soft Fail Option . . . 8
3.4.5 Structure of the NAG Error Messages . . . 9
3.4.6 Legacy Error Handling . . . 9
3.5 Input/output in the Library . . . 9
3.6 Auxiliary Routines as External Procedure Arguments . . . 9
3.7 Dynamic Memory Allocation . . . 10
3.8 License Management . . . 10
3.9 Unexpected Errors . . . 10
3.10 Calling the Library from Other Languages . . . 10
3.11 Arithmetic Considerations and Reproducibility of Results . . . 10
3.11.1 Bit-wise Reproducibility (BWR) . . . 12
3.11.1.1 Vendor Math Libraries and Conditional Bitwise Reproducibility (CBWR) 13
3.12 Multithreading . . . 13
3.12.1 Thread Safety . . . 13
3.12.1.1 Routines with Routine Arguments . . . 13
3.12.1.2 Input/Output . . . 14
3.12.1.3 Implementation Issues . . . 15
3.12.2 Parallelism . . . 15
3.12.2.1 Introduction . . . 15
3.12.2.2 How is Parallelism Used in the NAG Library? . . . 16
3.12.3 Multithreaded Routines . . . 17

4 How to Use NAG Documentation . . . 18

4.1 Using the Manual . . . 18
4.2 Structure of the Documentation . . . 18

4.3 Specification of Arguments . . . 19
4.3.1 Classification of Arguments (Intents) . . . 19
4.3.2 Constraints and suggested values . . . 20
4.3.3 Array Arguments . . . 20
4.4 Implementation-dependent Information . . . 21
4.5 Example Programs and Results . . . 21
4.6 Online Documentation . . . 22
4.6.1 HTML Format . . . 22
4.6.1.1 Viewing HTML5 Files . . . 22
4.6.1.2 Firefox (and other Mozilla based browsers) . . . 22
4.6.1.3 Other Browsers . . . 22
4.6.1.4 Navigating HTML5 Files . . . 23
4.6.1.5 Printing HTML5 Files . . . 23
4.6.1.6 Windows HTML Help . . . 23
4.6.2 PDF Format . . . 23
4.6.2.1 Viewing and Printing PDF Files . . . 23
4.6.2.2 Navigating the PDF Files . . . 23

5 Background to the Problems . . . 24

6 NAG Library Standards . . . 24

7 References . . . 24

1 Library Identification

Periodically a new **Mark** of the NAG Library is released: new routines are added, corrections and/or improvements are made to existing routines; and, occasionally, routines are withdrawn if they have been superseded by improved routines. A point release such as 26.1, only differs from a new Mark (e.g., 27) in that no routines are withdrawn.

You must know **which implementation**, **which precision** and **which mark and revision** of the Library you are using or intend to use. To find out these Library details, you can run a program which calls the NAG Library routine **a00aaf**.

The program could be:

```
Use nag_library, Only: a00aaf
Call a00aaf
End
```

Alternatively, the example program for **a00aaf** can be run using the `nag_example` scripts supplied with your implementation (see the Users' Note for details).

An example of the output is:

```
*** Start of NAG Library implementation details ***
Implementation title: Linux, 64-bit, NAG Fortran (32-bit integers)
   Precision: FORTRAN double precision
Product Code: FLL6A2619L
Mark: 26.1 (self-contained)

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

2 How to Find a NAG Library Routine

All users both familiar or unfamiliar with this Library who are thinking of using a routine from it, are asked to please follow these instructions:

(a) read **How to Use the NAG Library and its Documentation** as it provides valuable background information;

(b) select an appropriate chapter or routine by using the online **Keyword and GAMS Search**;

(c) read the relevant **Chapter Introduction**;

(d) choose a routine, and read the **routine document**. If the routine does not after all meet your needs, return to step (b);

(e) read the **Users' Note** for your implementation (this contains instructions on how to compile and run a program);

(f) consult local documentation, which should be provided by your local support staff, about access to the Library on your computing system;

(g) obtain a copy of the example program (see Section 4.5) for the particular routine of interest and experiment with it.

You should now be in a position to include a call to the routine in a program, and to attempt to compile and run it. You may of course need to refer back to the relevant documentation in the case of difficulties, for advice on assessment of results, and so on.

As you become familiar with the Library, some of steps (a) to (g) can be omitted, but it is useful to keep up to date with the following documents as they are subject to change:

How to Use the NAG Library and its Documentation;

the **Chapter Introduction**;

the **routine document**;

the **Users' Note** for your implementation.

3 How to Use the NAG Library

3.1 Structure of the Library

The NAG Library is a comprehensive collection of **routines** for the solution of numerical and statistical problems.

The Library is divided into **chapters**, each devoted to a branch of numerical analysis or statistics. Each chapter has a three-character name and a title, e.g.,

Chapter D01 – Quadrature

Exceptionally, Chapters H and S have one-character names. The chapters and their names are based on the ACM modified SHARE classification index (see ACM (1960–1976)).

All documented routines in the Library have six-character names, beginning with the characters of the chapter name, e.g.,

c06pcf

Note that the second and third characters are **digits**, not letters; e.g., 0 is the digit zero, not the letter O. The last letter of each routine name almost always appears as 'f' in the documentation. Chapters D03 and E04 have some routines whose last letter is 'a' rather than 'f'. An 'a' version is always paired with an 'f' routine, the 'a' version being safe to use in a multithreaded environment, but otherwise having identical functionality to the 'f' version.

Chapter F06 (Linear Algebra Support Routines) contains all the Basic Linear Algebra Subprograms, BLAS (Dongarra *et al.* (1988) and Dongarra *et al.* (1990)), with NAG-style names as well as the actual BLAS names, e.g., **f06paf (dgemv)**. The name in brackets is the equivalent double precision BLAS name. Chapter F16 contains some of the routines specified in the BLAS Technical Form (The BLAS Technical Forum Standard (2001) and Blackford *et al.* (2002)) and also some additional routines for integer valued vectors that are not in the standard. Some of the routines in Chapter F16 have both NAG-style names and BLAS names. Chapter F07 (Linear Equations (LAPACK)) and Chapter F08 (Least Squares and Eigenvalue Problems (LAPACK)) contain routines derived from the LAPACK project (Anderson *et al.* (1999)); also, Chapter F01 (Matrix Operations, Including Inversion) contains storage conversion routines derived from the LAPACK project. Like the BLAS, these routines have NAG-style names as well as LAPACK names, e.g., **f07adf (dgetrf)**. Details regarding these alternate names can be found in the relevant Chapter Introductions.

In order to take full advantage of machine-specific versions of BLAS and LAPACK routines provided by some computer hardware vendors, you are encouraged to use the BLAS and LAPACK names (e.g., **DGEMV** and **DGETRF**) rather than the corresponding NAG-style names (e.g., **f06paf (dgemv)** and **f07adf (dgetrf)**) wherever possible in your programs.

3.1.1 Experimental Routines

Some routines in the Library may be classified as experimental. These routines will be flagged by a note at the top of the documentation.

Routines may be classified as experimental if:

(a) The interface and / or functionality of the routine may change between Marks of the Library. The routine will have been designed in such a way as to minimize the number of such changes.

(b) The complexity of the routine, or suite of routines it is part of, are such that comprehensive testing is not practical. Routines classified as experimental have gone through the same testing and review processes as a normal Library routine, however it is recommended that additional care is taken when using the routine.

A routine classified as experimental may be reclassified as no longer being experimental, at which point the interface will become fixed as per a normal Library routine.

3.1.2 Long Names for Library Routines

Each documented routine has, in addition to its short six-character name, a long name beginning with the root **nagf_** and consisting of an underscore separated list of words. The long-name naming scheme has been chosen so that the long names group like routines together and group routines within a suite together.

The long name for each routine in a chapter is listed in the respective Chapter Contents page. The second word in the long name is fixed for each chapter, e.g., routines in Chapter D01 (Quadrature) all have long names that begin **nagf_quad_**.

Each chapter has a unique second word in its set of long names with the exception of Chapters F07 and F08 which share the same second word (**lapack**).

Note that the long names of BLAS and LAPACK routines, such as **nagf_blas_dgemm**, will not take advantage of machine-specific versions of BLAS and LAPACK. As mentioned in Section 3.1 you are recommended to use the plain BLAS or LAPACK name (in this case **DGEMM**) for performance reasons.

For those chapters that have both 'a' and 'f' versions of a routine, the long name for the f version is the same as that of the a version, but with an additional last word (**old** – signifying that the f version predates the a version).

It should be noted that the long names are implemented by the use of aliasing in the NAG Library interface block modules, and so long names are only accessible when calling the NAG Library from a Fortran program that Uses nag_library.mod.

Please refer to Section 3.3.1 for advice on supplying alternative routine names and, possibly, simplified routine interfaces.

3.2 General Advice

A NAG Library routine **cannot** be guaranteed to return meaningful results irrespective of the data supplied to it. Care and thought **must** be exercised in:

(a) formulating the problem;

(b) programming the use of Library routines;

(c) assessing the significance of the results.

The Foreword to the Manual provides some further discussion of points (a) and (c); the remainder of this document is concerned with (b) and (c).

3.3 Programming Advice

The Library and its documentation are designed with the assumption that you will write a calling program in Fortran (although it may be called from other languages – see Section 3.10).

When programming a call to a routine, read the routine document carefully, especially the description of the **arguments**. This states clearly which arguments must have values assigned to them on entry to the routine, and which return useful values on exit. See Section 4.3 for further guidance.

The most common types of programming error in using the Library are:

incorrect arguments in a call to a Library routine;

calling the Library from a single precision program.

The Use of the `nag_library` MODULE will help detect or prevent some of these errors. For example, when using this, incorrect parameter types will be caught at compile time and using `KIND=nag_wp` in the type of real and complex variables will maintain consistency with the Library.

Therefore, if a call to a Library routine results in an unexpected error message from the system (or possibly from within the Library), **check** the following:

Have some actual array arguments been passed as different dummy arguments (i.e., an array appears more than once in the argument list with different INTENTs)?

Have all array arguments been dimensioned correctly?

Avoid the use of NAG-type names for your own program units or COMMON blocks: in general, do not use names which contain a three-character NAG chapter name embedded in them; they may clash with the names of an auxiliary routine or COMMON block used by the NAG Library.

3.3.1 Alternative Routine Names

If the Library is called from a Fortran program then it is possible to use alternative names for user-callable routines. This can be done via the 'Use nag_library' statement at the start of the (sub)program in which the Library routine is called. For example, you wish to use the name 'BesselJ0' instead of the Library name **s17aef**. In this case the line

```
Use nag_library, Only: s17aef
```

would be replaced by

```
Use nag_library, Only: BesselJ0 => s17aef
```

The (sub)program would then use the name 'BesselJ0' in place of **s17aef** and call it with the identical interface.

If Library routines are called from other environments then many such environments offer ways of 'aliasing' a routine name by a preferred alternative name.

For many of the Library routines with more complex interfaces it is likely that only a subset of the functionality is required and that some parameter values will always remain unchanged or will not be referenced. In such cases it may be preferable to write your own wrapper to the Library routine with a much simpler interface and with a preferred alternative name. For example, if you wish to integrate a system of stiff ordinary differential equations without root finding or intermediate output, you could create the simple interface wrapper to the more complicated **d02ejf** interface.

```
Subroutine BDFsolve(xend,y)
Use nag_library, Only: nag_wp, d02ejf, d02ejw, d02ejx, d02ejy
Real(kind=nag_wp) :: xend, y(:)
Real(kind=nag_wp) :: tol, xstart
Integer           :: ifail, iw, n
Character         :: relabs
Real(kind=nag_wp), Allocatable :: w(:)

n = Size(y)
tol = 1.0e-3_nag_wp
relabs = 'M'
iw = (12+n)*n + 50
Allocate(w(iw))
ifail = 0
xstart = 0.0_nag_wp
Call d02ejf(xstart,xend,n,y,fcn,d02ejy,tol,relabs,d02ejx, &
            d02ejw,w,iw,ifail)
Return
End Subroutine BDFsolve
```

The above example of a user-defined wrapper would be compiled and linked with a main program that would include the simple call:

```
Call BDFsolve(xend,y)
```

3.3.2 The NAG Fortran Environment

The environment for the NAG Library is defined by the `nag_library` MODULE. Certain routines require you to Use this to access named constants (e.g., nag_wp). It is recommended that you also Use the MODULE to enable checking of INTERFACEs in the Library.

The exact location of `nag_library.mod` is installation dependent; please see the Users' Note for your implementation.

3.3.3 Direct and Reverse Communication Routines

Routines in the Library that require a user-supplied function may be classified as either direct communication or reverse communication.

Direct communication routines require a user-supplied subroutine to be provided as an actual argument to the NAG Library routine. You must write this subroutine using a very rigid interface as specified in the relevant routine document. For the majority of applications this is the simplest and most convenient usage. Sometimes however this approach can be restrictive:

(i) when the required format of the subroutine does not allow useful information to be passed conveniently to and from your calling program;

(ii) when the direct communication routine is being called from another computer language which does not fully support procedure arguments in a way that is compatible with the Library.

These restrictions can be removed by using a reverse communication routine. Instead of obtaining the solution in one call, reverse communication routines perform one step of the solution process before returning to the calling program with an appropriate flag (irevcm) set. The value of irevcm determines whether the process has finished or whether fresh information is required. In the latter case the required information must be calculated before re-entering the reverse communication routine. Thus you have the responsibility for providing an iterative loop. Although reverse communication routines will typically be more complicated to use than direct communication equivalents they do provide greater flexibility for the evaluation of the function.

3.4 Error Handling and the Argument ifail

3.4.1 Errors, Failure and Warning Conditions

The error, failure or warning conditions considered here are those that can be detected by explicit coding in a Library routine. Such conditions must be anticipated by the author of the routine. They should not be confused with run-time errors detected by the compilation system, e.g., detection of overflow or failure to assign an initial value to a variable.

In the rest of this document we use the word ‘error’ to cover all types of error, failure or warning conditions detected by the routine. They fall roughly into three classes.

(i) On entry to the routine the value of an argument is out of range. This means that it is not useful, or perhaps even meaningful, to begin computation.

(ii) During computation the routine decides that it cannot yield the desired results, and indicates a failure condition. For example, a matrix inversion routine will indicate a failure condition if it considers that the matrix is singular and so cannot be inverted.

(iii) Although the routine completes the computation and returns results, it cannot guarantee that the results are completely reliable; it therefore returns a warning. For example, an optimization routine may return a warning if it cannot guarantee that it has found a local minimum.

All three classes of errors are handled in the same way by the Library.

Each error which can be detected by a Library routine is associated with a number. Some numbers such as those associated with a failure in dynamic memory allocation (see Section 3.7) or detecting a valid licence (Section 3.8) are the same for all Library routines and may not be listed in individual routine documents. Recently added routines have standardized on using the same number for unexpected error exits (Section 3.9). All other numbers, with explanations of the errors, are listed in Section 6 (Error Indicators and Warnings) in the routine document. Unless the document specifically states to the contrary, you should not assume that the routine necessarily tests for the occurrence of the errors in their order of error number, i.e., the detection of an error does not imply that other errors have or have not been detected.

3.4.2 The ifail Argument

Most of the NAG Library routines which can be called directly by you have an argument called **ifail**. This argument is concerned with the NAG Library error trapping mechanism (and, for some routines, with controlling the output of error messages and advisory messages).

ifail has **two** purposes:

(i) to allow you to specify what action the Library routine should take if an error is detected;

(ii) to inform you of the outcome of the call of the routine.

For purpose (i), you **must** assign a value to **ifail** before the call to the Library routine. Since **ifail** is reset by the routine for purpose (ii), the argument must be the name of a variable, **not** a literal or constant.

The value assigned to **ifail** before entry should be either 0 (**hard fail** option), or 1 or -1 (**soft fail** option). If after completing its computation the routine has not detected an error, **ifail** is reset to 0 to indicate a **successful call.** Control returns to the calling program in the normal way. If the routine does detect an error, its action depends on whether the hard or soft fail option was chosen. If **ifail** is set to any value other than -1, 0 or 1 before calling the Library routine, a default of **ifail** $= 1$ is assumed.

3.4.3 Hard Fail Option

If you set **ifail** to 0 before calling the Library routine, execution of the program will terminate if the routine detects an error. Before the program is stopped, this error message is output:

```
** ABNORMAL EXIT from NAG Library routine XXXXXX: ifail = n
** NAG hard failure - execution terminated
```

where `XXXXXX` is the routine name, and `n` is the number associated with the detected error. An explanation of error number `n` is given in Section 6 of the routine document `XXXXXX`.

In addition, most routines output explanatory error messages immediately before the standard termination message shown above.

The hard fail option should be selected if you are in any doubt about continuing the execution of the program after an unsuccessful call to a NAG Library routine. For environments where it might be inappropriate to halt program execution when an error is detected it is recommended that the hard fail option is **not** used.

3.4.4 Soft Fail Option

To select this option, you must set **ifail** to 1 or -1 before calling the Library routine. Note that **ifail** $= 1$ is assumed when **ifail** is set to an invalid value before calling the Library routine.

If the routine detects an error, **ifail** is reset to the associated error number; further computation within the routine is suspended and control returns to the calling program.

If you set **ifail** to 1, then no error message is output (**silent exit**). If the output of error messages is undesirable, then silent exit is recommended.

If you set **ifail** to -1 (**noisy exit**), then before control is returned to the calling program, the following error message is output:

```
** ABNORMAL EXIT from NAG Library routine XXXXXX: ifail = n
** NAG soft failure - control returned
```

In addition, most routines output explanatory error messages immediately before the above standard message.

It is most important to test the value of ifail on exit if the soft fail option is selected. A nonzero exit value of **ifail** implies that the call was not successful so it is imperative that your program be coded to take appropriate action. That action may simply be to print **ifail** with an explanatory caption and then terminate the program. Many of the example programs in Section 9 of the routine documents have ifail-exit tests of this form. In the more ambitious case, where you wish your program to continue, it is essential that the program can branch to a point at which it is **sensible** to resume computation.

The soft fail option puts the onus on you to handle any errors detected by the Library routine. With the proviso that you are able to implement it **properly**, it is clearly more flexible than the hard fail option since it allows computation to continue in the case of errors. In particular there are at least two cases where its flexibility is useful:

(i) where additional information about the error or the progress of computation is returned via some of the other arguments;

(ii) in some routines, 'partial' success can be achieved, e.g., a probable solution found but not all conditions fully satisfied, so the routine returns a warning. On the basis of the advice in Section 6 and elsewhere in the routine document, you may decide that this partially successful call is adequate for certain purposes.

3.4.5 Structure of the NAG Error Messages

The notation ⟨*value*⟩ appearing in the documented error message is a place holder that will be populated by the value of a variable, argument name or some other piece of information when that error message is displayed.

3.4.6 Legacy Error Handling

A few routines (introduced mainly at Marks 7 and 8) use **ifail** in a nonstandard way to control the output of error messages, and also of advisory messages (see Chapter X04). In those routines **ifail** is regarded as an integer of the form $100c + 10b + a$, where a and b are either 0 or 1 and have the following significance:

$a = 0$: hard failure	$a = 1$: soft failure
$b = 0$: silent exit	$b = 1$: noisy exit

Details are given in the documents of the relevant routines; for those routines this alternative use of **ifail** remains valid.

3.5 Input/output in the Library

Most NAG Library routines perform no output to an external file, except possibly to output an error message. All error messages are written to a logical **error message** unit. This unit number (which is set by default to 6 in most implementations) can be changed by calling the Library routine **x04aaf**.

Some NAG Library routines may optionally output their final results, or intermediate results to monitor the course of computation. In general, output other than error messages is written to a logical **advisory message** unit. This unit number (which is also set by default to 6 in most implementations) can be changed by calling the Library routine **x04abf**. Although it is logically distinct from the error message unit, in practice the two unit numbers may be the same. Suites of routines with an option setting facility usually allow this unit number to be specified directly as an option.

All output from the Library is appropriately formatted.

There are only a few Library routines which perform input from an external file. These examples occur in Chapters E04, E05 and H. The unit number of the external file is an argument to the routine, and all input is formatted.

You must ensure that the relevant Fortran unit numbers are associated with the desired external files, either by an OPEN statement in your calling program, or by operating system commands.

3.6 Auxiliary Routines as External Procedure Arguments

In addition to those Library routines which are documented and are intended to be called by you directly, the Library also contains many auxiliary routines.

In general, you need not be concerned with them at all, although you may be made aware of their existence if, for example, you examine a memory map of an executable program which calls NAG routines. The only exception is that when calling some NAG Library routines you may be required or allowed to supply the name of an auxiliary routine from the NAG Library as an external procedure argument. The routine documents give the necessary details. In such cases, you only need to supply the name of the routine; you **never** need to know details of its argument list.

NAG auxiliary routines have names which are similar to the name of the documented routine(s) to which they are related, but with last letter 'Z', 'Y', and so on, e.g.,

g13afz is an auxiliary routine called by **g13aff**.

A few chapters contain auxiliary routines whose names are obtained by adding 50 to the second and third characters of the chapter name. For instance, Chapter E04 has an auxiliary routine with the name e54nfu which is normally used as the actual argument for the **qphess** argument of **e04nfa**; the corresponding name to be used with **e04nff** is e04nfu.

3.7 Dynamic Memory Allocation

Some NAG Library routines perform dynamic memory allocation to simplify their interfaces. Where possible, the amount of memory allocated by a routine will be given in the routine document (usually as a function of routine arguments). All memory allocated by NAG routines is deallocated before exit.

In the case where a routine detects a failure to dynamically allocate sufficient memory, the routine will set an error condition, by setting **ifail** $= -999$, and exit with an appropriate error message.

3.8 License Management

If your implementation is license managed then your local site will have details on how the license management is implemented; please contact your site installer for details. To determine whether a valid license is available on your machine run the example program for **a00acf**.

Should a valid license not be found when calling license managed routines from the Library then the routine will set an error condition, by setting **ifail** $= -399$, and exit with an appropriate error message. On Unix based systems, the appropriate environment variables should then be checked (e.g., NAG_KUSARI_FILE) to make sure this points to the licence file containing a valid licence, and the licence file should be checked for any obvious errors (e.g., the licence refers to a different implementation). If everything appears to be correct then please contact NAG (see Support from NAG for details).

3.9 Unexpected Errors

Internal calls to Library routines are checked for error exits even when these exits are not to be expected. Should an unexpected error exit occur the routine will set an error condition by setting **ifail** and exit with an appropriate error message. Historically, the number returned in **ifail** was particular to that routine and differing numbers could be used for this purpose. However, recently added routines have standardized by setting **ifail** $= -99$ for unexpected error detection.

3.10 Calling the Library from Other Languages

In general the NAG Library can be called from other computer languages (such as C and Visual Basic) provided that appropriate mappings exist between their data types).

NAG has produced C Header Files which comprise of a set of header files, indicating the match between C and Fortran data types for various compilers, documentation and examples. The documentation, examples and C Header Files are available from the NAG Web sites (see Support from NAG).

The Dynamic Link Library (DLL) implementation can be called in a straightforward manner from a number of languages and environments, e.g., Visual Basic, Visual Basic for Applications (Excel), Fortran, C and C++. Guidance on this is provided in the Users' Note for the NAG Library DLLs. Further details can be found on the NAG Web sites.

3.11 Arithmetic Considerations and Reproducibility of Results

The results obtained when calling a NAG Library routine depend not only on the algorithm used to solve the problem, but also on the compiler used to build the library, compiler run-time libraries and also the arithmetic properties of the machine on which the code is run.

Historically, different kinds of computer hardware tended to have different kinds of arithmetic. Some machines would store floating-point numbers using a base 16 significand and exponent system, others would use base 2, and some even used base 8 or 10. Such differences caused major headaches for

software library providers because code that worked well on one arithmetic system might not behave in exactly the same way on another. This meant that great care had to be taken to make the library code **portable**.

In addition, it was not unheard of for machine arithmetic to have flaws or errors where basic operations such as multiplication or division could sometimes give incorrect results, especially on numbers that were in some way ‘extreme’, such as being very large or small.

After the first of the IEEE standards for floating-point arithmetic (ANSI/IEEE (1985)) was introduced in the 1980s, the situation improved greatly. Nowadays most significant hardware, and certainly most hardware that NAG libraries run on, will use IEEE-style base 2 arithmetic. This makes production of portable code easier, but there are still problems, partly due to the latitude allowed by the IEEE standards. For example, hardware which uses extra-precise 80-bit internal registers for arithmetic, as originally introduced in the Intel 8087 coprocessor in the 1980s, behaves slightly differently from hardware that uses 64-bit registers, particularly if a compiler generates optimized code which holds arithmetic subexpressions in the extra-precise registers.

Since, for performance reasons, computer arithmetic is generally finite precision (as is certainly the case for IEEE standard arithmetic) most of the numerical methods implemented by NAG Library routines can only return an approximation to the true solution, simply due to the accumulation of rounding errors.

It should therefore be clear that running a program which calls a NAG Library routine with the same data on two different machines can give different results, due to compiler, hardware and run-time library considerations. Usually these differences are small – it may be that a result computed on one machine differs only in the last few significant bits from the same result computed on another machine – for example, when solving a well-conditioned set of linear equations on two different machines. Occasionally small differences may be magnified, for example if a conditional test depends on an imprecise result. A routine that searches for a mininum of an optimization problem may converge to a different local minimum, but in general, so long as the routine's documentation doesn't claim that the **same** local minimum will always be obtained, this should be acceptable. Even if an algorithm converges to the same local minimum, arithmetic differences may mean that a different number of iterations is taken to get there.

Modern hardware and optimizing compilers have introduced further scope for arithmetic quirks. An example is in the use of **Streaming SIMD Extension (SSE)** instructions. These low-level machine instructions allow hardware to operate on more than one number in parallel, if your compiler is smart enough to generate and use them correctly, or if you hand-code your own assembly language routines.

SSE instructions enable low-level parallelism of floating-point arithmetic operations. For example, a 128-bit SSE register can hold two 64-bit double precision (or four 32-bit single precision) numbers at the same time, and operate on them all simultaneously. This can lead to big time savings when working on large amounts of data.

But this may come at a price. Efficient use of SSE instructions can sometimes depend on exactly how the memory used to store data is aligned. Some SSE instructions for moving data to and from memory need memory to be aligned on a 16-byte boundary. If it happens that the memory (for example, a pointer to an array of numbers) that a NAG routine uses is **not** aligned nicely, then it may not be possible to use those SSE instructions. An optimizing compiler might well generate two instruction streams, one for when it detects that memory is aligned and one for when it is not.

An example should serve to make things clearer. Suppose we wish to compute the inner product of two vectors, **x** and **y**, each of length **n**. The inner product (or dot product) of two vectors is computed by multiplying together corresponding elements of the two vectors, and summing the individual products to get the result. A routine compiled by a good optimizing compiler would load numbers two or four at a time, multiply them together two or four at a time, and accumulate the results into the final result.

But if the memory is not nicely aligned – and it may well not be – the compiler needs to generate a different code path to deal with the situation. Here the result will take longer to get because the products must be computed and accumulated one at a time. At run-time, the code checks whether it can take the fast path or not, and works appropriately.

The problem is that by altering the order of the accumulations, we are quite possibly changing the final result, simply due to rounding differences when working with finite precision computer arithmetic.

Instead of getting the inner product

$$s = x_1 \times y_1 + x_2 \times y_2 + x_3 \times y_3 + \cdots + x_n \times y_n$$

we may get

$$s = (x_1 \times y_1 + x_3 \times y_3) + (x_2 \times y_2 + x_4 \times y_4) + \cdots .$$

It is likely that the result will be just as accurate either way – neither result will be precise due to finite arithmetic – but they may differ by a tiny amount. And if that tiny difference leads to a different decision being made by the code that called the inner product routine, the difference may be magnified.

Furthermore, it is possible that the same program running with bitwise identical data on the same machine may give different results when run twice in a row simply because, when the program is loaded, by chance some piece of memory may or may not be aligned on a particular boundary. Such non-deterministic results can be frustrating if you depend on always getting identical results for the same data.

On even newer hardware, **AVX** instructions use 256-bit and 512-bit registers, and can therefore operate on more numbers at a time. For AVX instructions, memory may need to be 32-byte aligned.

Some memory used by NAG Library routines is allocated inside the NAG Library. In order to minimize differences due to effects like that described above, we can try to make sure the memory is always aligned nicely – for example, by use of more controllable memory allocation routines where available – but that is not always possible since it partly depends on the support of the compiler.

Of course, no Library routine has control over memory you have allocated before being passed to the routine. If you do observe non-deterministic results which you suspect are due to memory considerations, and you are unable to accept this variation, then you are advised to make sure that any memory you allocate is aligned nicely; unfortunately, precisely how you do this is dependent on your system, but you may be able to get advice through NAG's usual support channels (see Support from NAG).

Parallelism, coming from a multithreaded implementation of the NAG Library and/or a multithreaded vendor library is another potential source of non-determinism in numerical results. Some routines may give different results when run on different numbers of cores, or even different results when a calculation is repeated on the same number of cores. Where reproducibility of results is vital, a purely serial NAG Library, without parallelism in either NAG routines or calls to parallel vendor library routines will generally be available in an appropriate implementation, and may be the best choice. You are advised to contact NAG (see Support from NAG) for advice.

3.11.1 Bit-wise Reproducibility (BWR)

Mathematical operations on fixed-length floating point numbers (e.g., 32-bit floats or 64-bit doubles) are not associative. This means that a computer may produce different results for $a + (b + c)$ and $(a + b) + c$. For example, an IEEE 754 32-bit floating point number has a mantissa of 23 bits. Therefore in this number format $2^{24} + 1 = 2^{24}$, which means that for instance $\left(2^{24} + 1\right) - 2^{24} = 0$ while $2^{24} + \left(1 - 2^{24}\right) = 1$. BWR is a term which refers to the case in which a given computer program (e.g., a set of source codes) produces bit-for-bit the exact same answer in different computing environments such as

1. Different operating systems (e.g., answers produced on Windows vs answers produced on Linux).
2. Different CPU architectures (e.g., Intel vs AMD or Intel Sandy Bridge vs Intel Ivy Bridge etc.).
3. Different compiler versions.
4. Different numbers of threads.

Users often desire BWR however it is extremely difficult to achieve. Typically you should ensure that:

(a) Instructions are always executed in exactly the same order.

(b) No advanced CPU features are used which may not be available on other processors (e.g., SSE3, SSE4, AVX).

(c) A fixed number of threads is always used.

Often condition (a) is equivalent to compiling with no (or very limited) compiler optimizations, since newer versions of compilers typically improve their code optimization algorithms, which means one version of a compiler may optimize a set of operations one way while the next version may optimize it a different way. Condition (b) typically means that only basic SSE instructions are allowed, such as are supported across the widest range of processors and the enhanced SIMD instructions present in newer processors are not exploited.

The result is that to achieve BWR across a wide range of computing environments one often has to sacrifice a lot of performance.

3.11.1.1 Vendor Math Libraries and Conditional Bitwise Reproducibility (CBWR)

An implementation of the NAG Library that is not self-contained will make calls to an appropriate vendor library containing, in particular, high performance linear algebra routines. The NAG Library has no direct control over BWR with respect to results obtained from calls to the vendor library. However, for at least one such vendor library, CBWR has been introduced such that if an environment variable is set and a set of conditions adhered to in the code calling the vendor library then BWR can be forced. Where CBWR is available for a vendor library used by an implementation of the NAG Library, details will be given in the Users' Note for that implementation.

It should be noted that many NAG routines do not adhere to the conditions set out by vendor library CBWR and so it may not be possible to ensure BWR for all NAG Library routines across different CPU architectures for implementations that are not self-contained.

3.12 Multithreading

3.12.1 Thread Safety

In multithreaded applications, each thread in a team processes instructions independently while sharing the same memory address space. For these applications to operate correctly any routines called from them must be **thread safe**. That is, any global variables they contain are guaranteed not to be accessed simultaneously by different threads, as this can compromise results. This can be ensured through appropriate synchronization, such as that found in OpenMP.

When a routine is described as thread safe we are considering its behaviour when it is *called* by multiple threads. It is worth noting that a thread unsafe routine can still, itself, be multithreaded. A team of threads can be created inside the routine to share the workload as described in Section 3.12.2.

Most routines in the NAG Fortran Library are thread safe, however there are some routines that are not thread safe as they use unsynchronised global variables (such as module variables, common blocks or variables with the SAVE attribute). These routines should not be called by multiple threads in a user program. Please consult Section 8 of each routine document for further information. A table available in the HTML documentation lists which routines are threadunsafe.

In the NAG Fortran Library there are some pairs of routines which share the same five character root name, for example, the routines **e04ucf/e04uca**. Each routine in the pair has exactly the same functionality, except that one of them has additional parameters in order to make it thread safe. The thread safe routine has a different last character in the name in place of the usual character (typically ‘a’ instead of ‘f’). Such pairs are documented in a single routine document and are listed in the individual Chapter Contents.

3.12.1.1 Routines with Routine Arguments

Some Library routines require you to supply a routine and to pass the name of the routine as an actual argument in the call to the Library routine. For many of these Library routines, the supplied routine interface includes array arguments (called **iuser** and **ruser**) specifically for you to pass information to the supplied routine without the need for global variables.

In the Fortran Library if the interfaces of a pair of thread safe (ending ‘a’) and non-thread safe (ending ‘f’) routines contain a user-supplied routine argument then the ‘a’ routine will contain the additional array arguments **iuser** and **ruser** (possibly plus others for internal use). In some cases the ‘a’ routine may need to be initialized by a separate initialization routine; this requirement will be clearly documented.

From Mark 26.1 routines with routine arguments will also contain the argument cpuser which is of Type (c_ptr) (from the iso_c_binding module). This allows more complicated data structures to be passed easily to the user-supplied routine in cases where **iuser** and **ruser** would be inconvenient. The following code fragment shows how this can be used.

```
      Module mymodule
          Use iso_c_binding, Only: c_f_pointer, c_ptr
          Private
          Public                              :: myfun
         Type, Public                         :: mydata
           Integer                            :: nx
           Real (Kind=nag_wp), Allocatable :: x(:)
         End Type mydata
       Contains
         Subroutine myfun(...,iuser,ruser,cpuser)
           Type (c_ptr), Intent (In)          :: cpuser
           Type (mydata), Pointer             :: md
           Call c_f_pointer(cpuser,md)
                ... Use md%x and md%nx ...
         End Subroutine myfun
      End Module mymodule
      ...
      Program myprog
         Use mymodule, Only: myfun,mydata
         Use iso_c_binding, Only: c_loc, c_ptr
         Type (c_ptr)                        :: cpuser
         Type (mydata), Target               :: md
...
         md%nx = 1000
         Allocate (md%x(md%nx))
         cpuser = c_loc(md)
...
         call nagroutine(...,myfun,cpuser,iuser,ruser,ifail)
...
      End Program
```

This mechanism is used, for example, in Section 10 in **e04stf**.

If you need to provide your supplied routine with more information than can be given via the interface argument list, then you are advised to check, in the relevant Chapter Introduction, whether the Library routine you intend to call has an equivalent reverse communication interface. These have been designed specifically for problems where user-supplied routine interfaces are not flexible enough for a given problem, and their use should eliminate the need to provide data through global variables. Where reverse communication interfaces are not available, it is usual to use global variables containing the required data that is accessible from both the supplied routine and from the calling program. It is thread safe to do this only if any global data referenced is made threadprivate by OpenMP or is updated using appropriate synchronisation, thus avoiding the possibility of simultaneous modification by different threads.

Thread safety of user-supplied routines is also an issue with a number of routines in multi-threaded implementations of the NAG Library, which may internally parallelize around the calls to the user-supplied routines. This issue affects not just global variables but also how the **iuser** and **ruser** arrays may be used. In these cases, synchronisation may be needed to ensure thread safety. Chapter X06 provides routines which can be used in your supplied routine to determine whether it is being called from within an OpenMP parallel region. If you are in doubt over the thread safety of your program you are advised to contact NAG for assistance.

3.12.1.2 Input/Output

The Library contains routines for setting the current error and advisory message unit numbers (**x04aaf** and **x04abf**). These routines use the SAVE statement to retain the values of the current unit numbers between calls. It is therefore not advisable for different threads of a multithreaded program to set the message unit numbers to different values. A consequence of this is that error or advisory messages output simultaneously may become garbled, and in any event there is no indication of which thread produces which message. You are therefore advised always to select the ‘soft failure’ mechanism without any error message (**ifail** $= +1$, see Section 3.4) on entry to each NAG Library routine called

from a multithreaded application; it is then essential that the value of **ifail** be tested on return to the application.

3.12.1.3 Implementation Issues

In very rare cases we are unable to guarantee the thread safety of a particular specific implementation. Note also that in some implementations, the Library is linked with one or more vendor libraries to provide, for example, efficient BLAS functions. NAG cannot guarantee that any such vendor library is thread safe. Please consult the Users' Note for your implementation for any additional implementation-specific information.

3.12.2 Parallelism

3.12.2.1 Introduction

The time taken to execute a routine from the NAG Library has traditionally depended, to a large degree, on the serial performance capabilities of the processor being used. In an effort to go beyond the performance limitations of a single core processor, multithreaded implementations of the NAG Library are available. These implementations divide the computational workload of some routines between multiple cores and executes these tasks in parallel. Traditionally, such systems consisted of a small number of processors each with a single core. Improvements in the performance capabilities of these processors happened in line with increases in clock frequencies. However, this increase reached a limit which meant that processor designers had to find another way in which to improve performance; this led to the development of **multicore** processors, which are now ubiquitous. Instead of consisting of a single compute core, multicore processors consist of two or more, which typically comprise at least a Central Processing Unit and a small cache. Thus making effective use of parallelism, wherever possible, has become imperative in order to maximize the performance potential of modern hardware resources, and the multithreaded implementations.

The effectiveness of parallelism can be measured by how much faster a parallel program is compared to an equivalent serial program. This is called the parallel **speedup**. If a serial program has been parallelized then the speedup of the parallel implementation of the program is defined by dividing the time taken by the original serial program on a given problem by the time taken by the parallel program using n cores to compute the same problem. Ideal speedup is obtained when this value is n (i.e., when the parallel program takes $\frac{1}{n}$th the time of the original serial program). If speedup of the parallel program is close to ideal for increasing values of n then we say the program has good **scalability**.

The scalability of a parallel program may be less than the ideal value because of two factors:

(a) the overheads introduced as part of the parallel implementation, and

(b) inherently serial parts of the program.

Overheads include communication and synchronisation as well as any extra setup required to allow parallelism. Such overheads depend on the efficiency of the compiler and operating system libraries and the underlying hardware. The impact on performance of inherently serial fractions of a program is explained theoretically (i.e., assuming an idealised system in which overheads are zero) by **Amdahl's law**. Amdahl's law places an upper bound on the speedup of a parallel program with a given inherently serial fraction. If r is the parallelizable fraction of a program and $s = 1 - r$ is the inherently serial fraction then the speedup using n sub-tasks, S_n, satisfies the following:

$$S_n \leq \frac{1}{\left(s + \frac{r}{n}\right)}$$

Thus, for example, this says that a program with a serial fraction of one quarter can only ever achieve a speedup of 4 since as $n \rightarrow \infty$, $S_n \leq 4$.

Parallelism may be utilised on two classes of systems: shared memory and distributed memory machines, which require different programming techniques. Distributed memory machines are composed of processors located in multiple components which each have their own memory space and are connected by a network. Communication and synchronisation between these components is explicit. Shared memory machines have multiple processors (or a single multicore processor) which can all access the same memory space, and this shared memory is used for communication and

synchronisation. The NAG Library makes use of shared memory parallelism using OpenMP as described in Section 3.12.2.2.

Parallel programs which use OpenMP create (or "fork") a number of **threads** from a single process when required at run-time. (Programs which make use of shared memory parallelism are also called **multithreaded** programs.) The threads form a **team** comprising of a single **master** thread and a number of **slave** threads. These threads are capable of executing program instructions independently of one another in parallel. Once the parallel work has been completed the slave threads return control to the master thread and become inactive (or "join") until the next **parallel region** of work. The threads share the same memory address space, i.e., that of the parent process, and this shared memory is used for communication and synchronisation. OpenMP provides some mechanisms for access control so that, as well as allowing all threads to access shared variables, it is possible for each thread to have private copies of other variables that only it can access. Threads in a team can create their own parallel regions within the current parallel region. At this next **level** of parallelism, the thread creating the new team becomes the master thread of that team. We call this **nested** parallelism.

Something to be aware of for multithreaded programs, compared to serial ones, is that identical results cannot be guaranteed, nor should be expected. Identical results are often impossible in a parallel program since using different numbers of threads may cause floating-point arithmetic to be evaluated in a different (but equally valid) order, thus changing the accumulation of rounding errors. For a more in-depth discussion of reproducibility of results see Section 3.11.

3.12.2.2 How is Parallelism Used in the NAG Library?

The multithreaded implementations differ from the serial implementations of the NAG Library in that it makes use of multithreading through use of OpenMP, which is a portable specification for shared memory programming that is available in many different compilers on a wide range of different hardware platforms (see OpenMP).

Note that not all routines are parallelized; you should check Section 8 of the routine documents to find details about parallelism and performance of routines of interest.

There are two situations in which a call to a routine in the NAG Library makes use of multithreading:

1. The routine being called is a NAG-specific routine that has been threaded using OpenMP, or that internally calls another NAG-specific routine that is threaded. This applies to multithreaded implementations of the NAG Library only.
2. The routine being called calls through to BLAS or LAPACK routines. The vendor library recommended for use with your implementation of the NAG Library (whether the NAG Library is threaded or not) may be threaded. Please consult the Users' Note for further information.

A complete list of all the routines in the NAG Library, and their threaded status is given in Section 3.12.3.

It is useful to understand how OpenMP is used within the Library in order to avoid the potential pitfalls which lead to making inefficient use of the Library.

A call to a threaded NAG-specific routine may, depending on input and at one or more points during execution, use OpenMP to create a team of threads for a parallel region of work. The team of threads will fork at the start of the parallel region before joining at the end of the parallel region. Both the fork and the join will happen internally within the routine call. However, there are situations in which the teams of threads may be made available to OpenMP directives in your code via user-supplied subprograms, we refer to directives not contained within a parallel region as **orphaned** directives. (See Section 8 of the routine documents for further information.) Furthermore, OpenMP constructs within NAG routines are executed by teams of threads created within the NAG code, that is, there are no orphaned directives in the Library itself. Throughout this documentation we assume the use of the recommended compiler as given in the Users' Note, and in particular the use of a single OpenMP run-time library. Thus all OpenMP environment variables will apply to your own code and to NAG routines. However, they may not be respected by vendor libraries that have a mechanism for overriding them. NAG provides routines in Chapter X06 to control threads for your **whole** program, including any specific to a vendor library being called by NAG. You should take care when calling these NAG routines from within your own parallel regions, since if nested parallelism is enabled (it is disabled by

default) the NAG routine will fork-and-join a team of threads for each calling thread, which may lead to contention on system resources and very poor performance. Poor performance due to contention can also occur if the number of threads requested exceeds the number of physical cores in your machine, or if some hardware resources are busy executing other processes (which may belong to other users in a shared system). For these reasons you should be aware of the number of physical cores available to your program on your machine, and use this information in selecting a number of threads which minimizes contention on resources. Please read the Users' Note for advice about setting the number of threads to use, or contact NAG (see Support from NAG) for advice.

If you are calling multithreaded NAG routines from within another threading mechanism you need to be aware of whether or not this threading mechanism is compatible with the OpenMP compiler runtime used to build the multithreaded implementation of the NAG Library on your platform(s) of choice. The Users' Note document for each of the implementations in question will include some guidance on this, and you should contact NAG for further advice if required.

Parallelism is used in many places throughout the NAG Library since, although many routines have not been the focus of parallel development by NAG, they may benefit by calling routines that have, and/or by calling parallel vendor routines (e.g., BLAS, LAPACK). Thus, the performance improvement due to multithreading, if any, will vary depending upon which routine is called, problem sizes and other parameters, system design and operating system configuration. If you frequently call a routine with similar data sizes and other parameters, it may be worthwhile to experiment with different numbers of threads to determine the choice that gives optimal performance. Please contact NAG for further advice if required.

As a general guide, many key routines in the following areas are known to benefit from shared memory parallelism:

Dense and Sparse Linear Algebra

FFTs

Random Number Generators

Quadrature

Partial Differential Equations

Interpolation

Curve and Surface Fitting

Correlation and Regression Analysis

Multivariate Methods

Time Series Analysis

Financial Option Pricing

Global Optimization

Wavelets

3.12.3 Multithreaded Routines

Many routines are threaded using OpenMP in multithreaded implementations of the NAG Library. These implementations are denoted by having a product code of the form 'FS_____', rather than 'FL_____' for serial NAG Library implementations. Please consult Section 8 of each routine document for further information. A table available in the HTML documentation lists which routines have been threaded by NAG. The list also includes routines which internally call BLAS or LAPACK routines, which may be threaded within the vendor library used by both serial and multithreaded NAG Library implementations. You are advised to consult the documentation for the vendor library for further information. Please consult the Users' Note for your implementation for any additional implementation-specific information.

4 How to Use NAG Documentation

4.1 Using the Manual

The Manual is designed to serve the following functions for the NAG Library:

to give background information about different areas of numerical and statistical computation;

to advise on the choice of the most suitable NAG Library routine or routines to solve a particular problem;

to give all the information needed to call a NAG Library routine correctly from a Fortran program, and to assess the results.

At the beginning of the Manual are some general introductory documents which provide some background and additional information.

The document entitled 'Mark 26 NAG Fortran Library News' provides details of new routines added, details of routines scheduled for withdrawal and details of routines withdrawn at this mark. This document also provides details of internal changes affecting the user at this Mark.

The document entitled 'Advice on Replacement Calls for Withdrawn/Superseded Routines' provides advice on how to modify your program.

The online documentation includes a Keyword and GAMS Search which is available as a keyword search box at the top of every page and, additionally, a separate page containing a form and search guidelines.

Having found a likely chapter or routine, you should read the corresponding **Chapter Introduction,** which gives background information about that area of numerical computation, and recommendations on the choice of a routine, including indexes, tables and decision trees.

When you have chosen a routine, you must consult the **routine document**. Each routine document is essentially self-contained (it may, however, contain references to related documents). It includes a description of the method, detailed specifications of each argument, explanations of each error exit, remarks on accuracy, and (in most cases) an example program to illustrate the use of the routine. In some cases a plot accompanies an example program to illustrate the results from running the example program (possibly amended from the original to output more data points).

4.2 Structure of the Documentation

The NAG Library Manual is the principal documentation for the NAG Library. It has the same chapter structure as the Library: each chapter of routines in the Library has a corresponding chapter (of the same name) in the Manual. The chapters occur in alphanumeric order. General introductory documents appear at the beginning of the Manual.

Each chapter consists of the following documents:

Chapter Contents, e.g., D01 (quad) Chapter Contents;

Chapter Introduction, e.g., D01 (quad) Chapter Introduction;

Routine Documents, one for each documented routine in the chapter.

A routine document has the same name as the routine which it describes. Within each chapter, routine documents occur in alphanumeric order of short names. For those chapters that have both 'a' and 'f' versions of a routine, the routine descriptions are combined into one routine document.

All routine documents have the same structure consisting of ten numbered sections:

1. **Purpose**
2. **Specification**
3. **Description**
4. **References**
5. **Arguments** (see Section 4.3)

6. Error Indicators and Warnings

7. Accuracy

8. Parallelism and Performance

9. Further Comments

10. Example (see Section 4.5)

In some documents (notably Chapters E04, E05 and H) there are a further three sections:

11. Algorithmic Details

12. Optional Parameters

13. Description of Monitoring Information

The sections numbered 11. and 13. above are optional; thus, the section titled **Optional Parameters** may appear as (the possibly final) Section 11.

4.3 Specification of Arguments

The specification of routines in routine documents (see Section 2 in **d03pdf/d03pda**) includes the C Header interface which can be used to call a NAG Library routine from C or C++; the name used for this interface is the Fortran Interface short name appended by an underscore (e.g., d03faf_ corresponds to d03faf). For routines with 'a' and 'f' versions, the same approach also applies to the 'a' version (e.g., d03pda_ corresponds to d03pda).

Section 5 of each routine document contains the specification of the arguments, in the order of their appearance in the argument list.

4.3.1 Classification of Arguments (Intents)

Arguments are classified as follows.

Input: you must assign values to these arguments on or before entry to the routine, and these values are unchanged on exit from the routine.

Output: you need not assign values to these arguments before entry to the routine; the routine may assign values to them.

Input/Output: you must assign values to these arguments before entry to the routine, and the routine may then change these values.

Workspace: array arguments which are used as workspace by the routine. You must supply arrays of the correct type and dimension. In general, you need not be concerned with their contents.

Communication Array: arguments which are used to communicate data from one routine call to another.

External Procedure: a routine which must be supplied (e.g., to evaluate an integrand or to print intermediate output). Usually it must be supplied as part of your calling program, in which case its specification includes full details of its argument list and specifications of its arguments (all enclosed in a box). Its arguments are classified in the same way as those of the Library routine, but because you must write the procedure rather than call it, the significance of the classification is different.

> *Input*: values may be supplied on entry, which your procedure **must not** change.
>
> *Output*: you may or must assign values to these arguments before exit from your procedure.
>
> *Input/Output*: values may be supplied on entry, and you may or must assign values to them before exit from your procedure.

Occasionally, as mentioned in Section 3.6, the procedure can be supplied from the NAG Library, and then you only need to know its name.

User Workspace: array arguments which are passed by the Library routine to an external procedure argument. They are not used by the routine, but you may use them to pass information between your calling program and the external procedure.

Dummy: a simple variable which is not used by the routine. A variable or constant of the correct type must be supplied, but its value need not be set. (A dummy argument is usually an argument which was required by an earlier version of the routine and is retained in the argument list for compatibility.)

4.3.2 Constraints and suggested values

The word '*Constraint*:' or '*Constraints*:' in the specification of an *Input* argument introduces a statement of the range of valid values for that argument, e.g.,

Constraint: **n** > 0.

If the routine is called with an invalid value for the argument (e.g., **n** $= 0$), the routine will usually take an error exit, returning a nonzero value of **ifail** (see Section 3.4).

Constraints on arguments of type Character only list upper case alphabetic characters, e.g.,

Constraint: **check** = 'n'.

In practice, all routines with Character arguments will permit the use of lower case characters.

Occasionally, an enhancement of an existing routine at a given Mark may weaken some constraints on some arguments, this will not change the behaviour of existing code that calls the routine, but will allow new code to take advantage of enhanced functionality.

The phrase '*Suggested value*:' introduces a suggestion for a reasonable initial setting for an *Input* argument (e.g., accuracy or maximum number of iterations) in case you are unsure what value to use; you should be prepared to use a different setting if the suggested value turns out to be unsuitable for your problem.

4.3.3 Array Arguments

Most array arguments have dimensions which depend on the size of the problem. In Fortran terminology they have 'adjustable dimensions': the dimensions occurring in their declarations are integer variables which are also arguments of the Library routine.

For example, a Library routine might have the specification:

```
Subroutine <name> (m, n, a, b, ldb)
Integer            m, n, a(n), b(ldb,n), ldb
```

For a **one-dimensional** array argument, such as a in this example, the specification would begin

a(**n**) – Integer array

You must ensure that the dimension of the array, as declared in your calling (sub)program, is at least as large as the value you supply for **n**. It may be larger, but the routine uses only the first **n** elements.

For a **two-dimensional** array argument, such as **b** in the example, the specification might be

b(**ldb**,**n**) – Integer array

On entry: the m by n matrix B.

and the argument **ldb** might be described as follows:

ldb – Integer

On entry: the first dimension of the array **b** as declared in the (sub)program from which <name> is called.

Constraint: **ldb** $\geq$ **m**.

You **must** supply the **first** dimension of the array b, as declared in your calling (sub)program, through the argument **ldb**, even though the number of rows actually used by the routine is determined by the argument **m**. You must ensure that the first dimension of the array is at least as large as the value you supply for **m**. The extra argument **ldb** is needed to allow the routine to act on subarrays of a larger two-dimensional array, e.g., factorizing a diagonal submatrix of a larger matrix.

You must also ensure that the **second** dimension of the array, as declared in your calling (sub)program, is at least as large as the value you supply for **n**. It may be larger, but the routine uses only the first n columns.

A program to call the hypothetical routine used as an example in this section might include the statements:

```
Integer aa(100), bb(100,50)
ldb = 100
.
.
.
m = 80
n = 20
Call <name>(m,n,aa,bb,ldb)
```

or

```
Integer Allocatable :: aa(:), bb(:,:)
Integer             :: m, n, ldb
.
.
.
Read(5,*) m, n
ldb = m
Allocate (aa(m),bb(ldb,n))
Call <name>(m,n,aa,bb,ldb)
```

Many NAG routines contain array arguments declared with the 'assumed size' array dimension, and would be given as

```
Integer        a(*), b(ldb,*)
```

However, the original declaration of an array in your calling program must always have dimensions, greater than or equal to the minimum value documented. The advantage of using allocatable arrays is that they can be dynamically allocated to be of a correct size not known at compile time.

Consult an expert or a textbook on Fortran if you have difficulty in calling NAG routines with array arguments.

4.4 Implementation-dependent Information

In order to support all implementations of the Library, the Manual has adopted a convention of using ***bold italics*** to distinguish terms which have different interpretations in different implementations.

One bold italicised term is ***machine precision***, which denotes the relative precision to which real floating-point numbers are stored in the computer, e.g., in an implementation with approximately 16 decimal digits of precision, ***machine precision*** has a value of approximately 10^{-16}.

The precise value of ***machine precision*** is given by the routine **x02ajf**. Other routines in Chapter X02 return the values of other implementation-dependent constants, such as the overflow threshold, or the largest representable integer. Refer to the X02 Chapter Introduction for more details.

The bold italicised term ***block size*** is used only in Chapters F07 and F08. It denotes the block size used by block algorithms in these chapters. You only need to be aware of its value when it affects the amount of workspace to be supplied – see the arguments **work** and **lwork** of the relevant routine documents and the appropriate Chapter Introduction.

For each implementation of the Library, a separate **Users' Note** is published. This is a short document, revised at each mark. At most installations it is available in machine-readable form. It gives any necessary additional information which applies specifically to that implementation, in particular:

the values returned by Chapter X02 routines;

the default unit numbers for output (see Section 3.5);

the meanings of the precision arguments nag_rp (***reduced precision***), nag_wp (***basic precision***) and nag_hp (***additional precision***).

4.5 Example Programs and Results

The **example program** in Section 10 of most routine documents illustrates a simple call of the routine. The programs are designed so that they can be fairly easily modified, and so serve as the basis for a simple program to solve your problem.

For each implementation of the Library, NAG distributes the example programs in machine-readable form, with all necessary modifications already applied. Many sites make the programs accessible to you in this form. Generic forms of the programs, without implementation-specific modifications, may be

obtained directly from the NAG web site. The Users' Note for your implementation will mention any special changes which need to be made to the example programs.

Note that the results obtained from running the example programs may not be identical in all implementations and may not agree exactly with the results in the Manual.

For many routine documents, a plot of the example program results is also provided. In some cases the example program has been modified slightly to produce larger sets of results to give a more representative plot of the solution profile produced.

4.6 Online Documentation

The complete NAG Library Manual, Mark 26 can be viewed online in the following formats:

HTML, a fully linked version of the manual using HTML, SVG and MathML (recommended for browsing) and providing links to the PDF version of each document (recommended for printing);

PDF, a full PDF manual browsed using the PDF bookmarks, or via HTML index files;

Single file PDF, the manual as a single PDF file;

Windows HTML help, Windows HTML help version as a single file.

The two single file formats are more compact than the formats that use one file per routine and, for example, allow text searches across the entire manual, but of course the larger files may not be so convenient if you only need to view the documentation for a few routines.

The following sections describe how to obtain the software required to view the documentation and advises you how best to navigate the files with or without a browser.

4.6.1 HTML Format

4.6.1.1 Viewing HTML5 Files

These files do not use any proprietary browser specific features, and conform to relevant W3C Recommendations (HTML, MathML, SVG, CSS).

Support for these languages may require that your browser be updated and/or the installation of additional fonts. This information is restricted to the more widely used browsers. If you require information for additional browsers please contact NAG.

4.6.1.2 Firefox (and other Mozilla based browsers)

Versions of Firefox from Firefox 4 onwards should display MathML in HTML files by default.

Rendering of the mathematics is improved if you install the STIX or other OpenType math fonts if they are not already included on your system (as is the case with OS X and some Linux distributions). Full details of the installers available for these fonts on all the major platforms are included in the Firefox MathML fonts page:
http://www.mozilla.org/projects/mathml/fonts/

4.6.1.3 Other Browsers

If Firefox is not being used, then the javascript on the page loads the MathJax javascript library (http://www.mathjax.org) to enable MathML rendering. By default this is loaded from the web using the MathJax Content Distribution Network. If you require the documentation to work without an Internet connection then you may either use Firefox as described in the previous section or you may download a local copy of MathJax (http://docs.mathjax.org/en/latest/installation.html) which needs to be unpacked on to your local fileserver or file system, and then edit the file ../styles/nagmathml.js changing the line `http://cdn.mathjax.org/mathjax/latest/` to refer to your local installation.

4.6.1.4 Navigating HTML5 Files

A main index file has been provided (html/frontmatter/manconts.html) which links to individual Chapter Contents documents, which in turn link to a complete set of HTML files. Use your browser to navigate from this main index file. For each routine document in HTML format you are provided with a link to its equivalent PDF file, this file has been provided primarily for printing purposes.

Each library document contains a number of hyperlinks to particular elements, e.g., arguments, sections, chapter contents, etc. The following key identifies the colour used for each element:

CSS colour	**CSS name**
black	NAG type
green	appendix, chap, chapter introduction, decision tree, general introduction, section
grey	withdrawn document
pale blue	equation, figure, item in a list, note, bibliographic reference, table, url, verbatim item, website
navy blue	ifail value
red	parameter name
pink	member
purple	optional parameter
royal blue	html table of contents, example plot, routine document, link to a routine example from a table of contents

4.6.1.5 Printing HTML5 Files

It is possible to print your HTML5 files from the browser, however support for printing from browsers, especially support for printing mathematics, varies considerably between versions of browsers and platforms and printer drivers in use.

4.6.1.6 Windows HTML Help

The Windows HTML Help version of the manual is essentially a compressed version of the HTML5 help, customised for the Windows HTML Help viewer (with a bundled copy of MathJax). This format can be very convenient as it is a small compressed single file version allowing full text search over the entire library. You may find this useful if you have a Microsoft Windows desktop, even if you have the NAG Library installed on a different platform.

4.6.2 PDF Format

4.6.2.1 Viewing and Printing PDF Files

If you do not already have a copy of Adobe Acrobat Reader, a free copy can be downloaded from http://www.adobe.com/reader. Please check this site for availability of a reader for your platform. While we recommend the use of Acrobat Reader, there are alternative PDF viewers available which can also be used, such as xpdf or ghostview.

If Acrobat is not running as a plug-in then the bookmark links will not work correctly if you are browsing the PDF files via http rather than the local filesystem. You are advised to reinstall Adobe Acrobat which should rectify the problem.

4.6.2.2 Navigating the PDF Files

The manual is supplied as a set of individual PDF files, one for each routine document, chapter introduction, etc.. Each PDF file contains bookmarks that can be used to navigate between the files. Alternatively, and often more conveniently, HTML tables of contents are supplied which allow you to navigate to the desired file using a browser, and then using Acrobat as a browser plugin to read or print the document.

Alternatively the single file version of the PDF may be used. In this case the bookmarks will provide links to every routine in the Library, and text search may be used to search the entire Library contents.

5 NAG Library Design and Development

Various aspects of the design and development of the NAG Library, and NAG's technical policies and organization are given in Ford (1982), Ford *et al.* (1979), Ford and Pool (1984), and Hague *et al.* (1982).

6 NAG Library Standards

NAG Library development adheres to a number of international standards, see ISO Fortran 95 (1997), ANSI (1966), ANSI (1978), ANSI/IEEE POSIX (1995), Basic Linear Algebra Subprograms Technical (BLAST) Forum (2001).

7 References

ACM (1960–1976) Collected algorithms from ACM index by subject to algorithms

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide* (3rd Edition) SIAM, Philadelphia http://www.netlib.org/lapack/lug

ANSI (1966) USA standard Fortran *Publication X3.9* American National Standards Institute

ANSI (1978) American National Standard Fortran *Publication X3.9* American National Standards Institute

ANSI/IEEE (1985) IEEE standard for binary floating-point arithmetic *Std 754-1985* IEEE, New York

ANSI/IEEE POSIX (1995) *POSIX Standard Thread Library* ANSI/IEEE POSIX 1003.1c:1995

Basic Linear Algebra Subprograms Technical (BLAST) Forum (2001) *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard* University of Tennessee, Knoxville, Tennessee http://www.netlib.org/blas/blast-forum/blas-report.pdf

Blackford L S, Demmel J, Dongarra J J, Duff I S, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K and Whaley R C (2002) An updated set of *Basic Linear Algebra Subprograms (BLAS) ACM Trans. Math. Software* **28** 135–151

Dongarra J J, Du Croz J J, Duff I S and Hammarling S (1990) A set of Level 3 basic linear algebra subprograms *ACM Trans. Math. Software* **16** 1–28

Dongarra J J, Du Croz J J, Hammarling S and Hanson R J (1988) An extended set of FORTRAN basic linear algebra subprograms *ACM Trans. Math. Software* **14** 1–32

Ford B (1982) Transportable numerical software *Lecture Notes in Computer Science* **142** 128–140 Springer–Verlag

Ford B, Bentley J, Du Croz J J and Hague S J (1979) The NAG Library 'machine' *Softw. Pract. Exper.* **9(1)** 65–72

Ford B and Pool J C T (1984) The evolving NAG Library service *Sources and Development of Mathematical Software* (ed W Cowell) 375–397 Prentice–Hall

Hague S J, Nugent S M and Ford B (1982) Computer-based documentation for the NAG Library *Lecture Notes in Computer Science* **142** 91–127 Springer–Verlag

ISO Fortran 95 (1997) ISO Fortran 95 programming language (ISO/IEC 1539–1:1997)

OpenMP *The OpenMP Specification for Parallel Programming* http://www.openmp.org

The BLAS Technical Forum Standard (2001) http://www.netlib.org/blas/blast-forum