NAG Library

Essential Introduction

Note: *this document is essential reading for any prospective user of the Library.*

Contents

1 Summary for All Users

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

(a) read the whole of this **Essential Introduction**;

(b) select an appropriate chapter or function by searching through the **Keyword and GAMS Search**;

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

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

(e) read the **Users' Note** for your implementation;

(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 4.4) for the particular function of interest and experiment with it.

You should now be in a position to include a call to the function 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 (h) can be omitted, but it is always essential to:

be familiar with this **Essential Introduction**;

be familiar with the **Chapter Introduction**;

read the **function document**;

be aware of the **Users' Note** for your implementation.

2 The Library and its Documentation

2.1 Structure of the Library

The NAG C Library is a comprehensive collection of **functions** 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 functions have two names. One is based on the SHARE index classification and consists of a six-character name which begins with the characters of the chapter/subchapter name, for example

c06pcc

The letters of this type of function name are always lower case, the second and third characters being digits and the last letter being c. This function name is referred to as the short name. Each function also has a more meaningful and longer name, for example

nag_sum_fft_complex_1d

which we refer to as the long name. The long name may be used as an alternative to the short name when calling the function. See Section 3.3 for further details.

2.2 Structure of the Documentation

The NAG Library Manual is the principal documentation for the NAG C Library. It has the same chapter structure as the Library: each chapter of functions 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., Chapter d01;

Chapter Introduction, e.g., the d01 Chapter Introduction;

Function Documents, one for each documented function in the chapter.

A function document has the same short name as the function which it describes. Within each chapter, function documents occur in alphanumeric order of short names. It should be noted that all the computational functions from LAPACK, Release 3 are included in the NAG C Library and can be called by the NAG provided C Library interfaces to LAPACK. The NAG C Library names follow the naming convention of LAPACK except that the names are in lower case and 'nag_' is prepended.

Documentation is provided 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.

Advice on viewing and navigating the formats available can be found in the document Online Documentation.

The most up-to-date version of the documentation is accessible via the NAG web site (see Section 5).

2.3 Implementations of the Library

The Library is available on many different computer systems. For each distinct system, an **implementation** of the Library is prepared by NAG, e.g., the Windows 64 bit implementation. The implementation is distributed to sites as one or more tested compiled libraries.

An implementation is usually specific to a range of machines/operating systems (e.g., PCs running specified variants of the Microsoft Windows operating system/Linux); it may also be specific to a particular C compiler (gcc/Intel C), or compiler option (such as threaded mode).

Essentially the same facilities are provided in all implementations of the Library, but, because of differences in arithmetic behaviour and in the compilation system, functions cannot be expected to give identical results on different systems, especially for sensitive numerical problems.

The documentation supports all implementations of the Library, with the help of a few simple conventions, and a small amount of implementation-dependent information, which is published in a separate **Users' Note** for each implementation.

2.4 Library Identification

Periodically a new **Mark** of the NAG C Library is released: new functions are added, corrections and/or improvements are made to existing functions; and occasionally functions are withdrawn if they have been superseded by improved functions.

You must know **which implementation** and **which mark and revision** of the Library you are using or intend to use. To find out which implementation, precision and mark of the Library is available at your site, you can run a program which calls the NAG C Library function nag_implementation_details (a00aac).

This function has no arguments; it simply outputs text to the standard output, if available. Details can also be found by a call to nag_implementation_separated_details (a00adc).

Note: for the NAG C Library the release prior to Mark 23 was Mark 9. The content of the NAG C Library, Mark 24 is generally equivalent to that of other NAG Libraries at this mark.

2.5 C Language Standards

All functions in the NAG C Library conform fully to ANSI C (C90).

3 Using the Library

3.1 General Advice

A NAG C Library function **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 functions;

(c) assessing the significance of the results.

The remainder of Section 3 is concerned with (b) and (c).

3.2 Programming Advice

The Library and its documentation are designed with the assumption that you will write a calling program in C (although it may be called from other languages – see Section 3.9).

When a suitable NAG function has been selected, the function must be called from the C Library via a suitable user-written program, the calling program. This manual assumes that you have sufficient knowledge of the C programming language to be able to write such a program. Each C Library function document contains an example of a suitable calling program (see Section 4.4).

When writing a calling program, a number of environmental features common to all such NAG programs must be observed in addition to specific features which are relevant to the particular NAG function being called. These features are discussed below; you should also refer to the example program.

3.2.1 The NAG C environment

The environment for the NAG C Library is defined in a number of include files; a list is given in Section 3.2.1.6. The most important of the header files is `nag.h`, which must be included in any program that calls a NAG C Library function and must precede any other NAG header file.

These include files are placed in <product_folder>/include folder by the installer. For its exact location please see the **Users' Note** or other local documentation.

The file `nag.h` defines data types and error codes used in the NAG C Library together with a number of macros used in example programs.

You will also need to include the header file `nag_stdlib.h` in the calling program, if any memory management is required; see Section 3.2.1.2.

3.2.1.1 NAG data types

Integer
This data type is used for integer arguments to NAG C Library functions. It is normally defined to be `long`. Please refer to the Users' Note for its exact type.

Nag_Boolean
This is an enumeration type defined as

```
typedef enum{Nag_FALSE=0; Nag_TRUE=1} Nag_Boolean;
```

This replaces the previously used Boolean type. In order that you can continue to use your existing program, a preprocessor define CL07_COMPATIBILITY has been introduced. This feature can be used from the command line by specifying

```
-DCL07_COMPATIBILITY
```

Complex
This data type is a structure defined for use with complex numbers:

```
typedef struct {double re, im;} Complex;
```

Pointer
This data type represents the generic pointer `void *`.

Nag_Comm
This data type is a structure with a number of fields among which are a generic pointer, an Integer pointer and a double pointer which may be used for communicating information between a user-defined function and your calling program, where the user-defined function is supplied as an argument to the NAG function. This avoids the necessity of using global variables for such communication. The use of Integer and double pointers allows double and Integer arrays to be communicated with no casting. For an example of use, please see the NAG C Library Section 10 in nag_opt_simplex_easy (e04cbc).

Nag_User
This is also a communication structure with a generic pointer with usage similar to Nag_Comm.

Nag_FileID
This data type is an integer type used by certain NAG C Library functions which deal with input or output files. Functions in Chapter x04 must be used to associate a file name with the file ID.

Enumeration Types
A number of other enumeration types are defined in `nag_types.h` (included by nag.h) for use in calls to various NAG C Library functions. You should use these enumeration types in your C/C++ calling programs. If the NAG C Library is being called from languages other than C/C++ we have provided a function to map the enumeration members to integer values (see nag_enum_name_to_value (x04nac)).

Other structure types
A number of structures have been defined to facilitate calls to NAG functions. Please note that only those components of a structure that are relevant to a call to a specific NAG function are described in the corresponding function document. A complete specification of a NAG defined structure can be found in the NAG C Library header file `nag_types.h` included in the distribution materials.

3.2.1.2 Memory management in the Library

Memory is frequently dynamically allocated within NAG C Library functions. All requests for memory are checked for success or failure. In the unlikely event of failure occurring the Library function returns or terminates with the error state **NE_ALLOC_FAIL** (details of error handling in the Library are given in Section 3.6).

The macros `NAG_ALLOC`, `NAG_REALLOC` and `NAG_FREE` are defined to select suitable memory management functions for the NAG C Library. `NAG_ALLOC` has two arguments; the first specifies the number of elements to be allocated while the second specifies the type of element. The statement

```
p = NAG_ALLOC(n, double);
```

allocates `n` elements of memory of type `double` to `p`, a pointer to `double`.

`NAG_REALLOC` has three arguments; the first specifies the name of the pointer whose memory is to be extended, the second specifies the number of elements, the third specifies the type of element.

The statement

```
p = NAG_REALLOC(p, n, double);
```

allocates `n` elements of memory of type `double` to `p`, a pointer to `double`.

`NAG_FREE` frees memory allocated by `NAG_ALLOC` or `NAG_REALLOC`; its single argument is the pointer which specifies the memory to be deallocated. The statement

```
NAG_FREE(p);
```

deallocates memory pointed to by `p` and sets its value to **NULL**.

These macros are defined in the header file `nag_stdlib.h` which must be included if these macros are used in the calling program. `NAG_FREE` must be used to free memory allocated and returned from a NAG function. If memory is allocated using `NAG_ALLOC` for whatever reason, it must be freed using `NAG_FREE`. For an illustration of their use, see Section 10.1 in nag_1d_cheb_fit_constr (e02agc). The use of `NAG_ALLOC`, `NAG_REALLOC` and `NAG_FREE` is strongly recommended.

3.2.1.3 The Nag_Order argument

Different programming languages lay out two-dimensional data in memory in different ways. The C language treats a two-dimensional array as a single block of memory arranged in rows, the so called 'row-major' ordering. Some other languages, notably Fortran, arrange two-dimensional arrays by column ('column-major' ordering). Commencing at Mark 7, those functions in the NAG C Library that deal with two-dimensional arrays and where it is sensible to do so, have an extra argument, called order, which allows you to specify that your data is arranged in rows (by setting order to Nag_RowMajor) or in columns (by setting order to Nag_ColMajor). This is particularly useful if the NAG C Library is being called from a language which uses the column-major ordering or if you wish to call a function from a language, such as Visual Basic 7 onwards, which supports column-major ordering.

3.2.1.4 Array references

In C it is possible to declare a two-dimensional variable using notation of the form:

```
double a[dim1][dim2];
```

When this variable is an argument to a function, it is effectively treated by the compiler as a pointer, `*a`, of type `double` with an allocated memory of `dim1*dim2` on the stack. The address of an element of this array, say `a[3][5]` is then an explicit address computed to be `*(a+3*dim2+5)`, since C stores data in row-major order.

Alternatively it is possible for you to allocate memory explicitly (on the heap) to a pointer of type `double *`, using the form:

```
a=(double *)malloc((size_t)(dim1*dim2*sizeof(double));
```

In this case the C preprocessor allows a succinct notation for computing this explicit address by using a macro definition:

```
#define A(I,J) a[I*dim2+J]
```

The element of this array if indexed ij is then indexed using the pointer notation `*(a+i*dim2+j)` or by using the array notation `a[i*dim2+j]`; or by using `A(I,J)` assuming the macro [2] is already defined.

We often wish to refer to the storage of elements representing a submatrix of the matrix A, for example, the submatrix comprising rows k to l and columns m to n. For convenience we use the notation `A(k:l,m:n)` to refer to those elements of `a` storing the given submatrix with elements $A_{ij} = \mathrm{A}(i-1, j-1)$ (see [2] and [4]), for $i = k, \ldots, l$ and $j = m, \ldots, n$. That is,

```
A(k:l,m:n) = {a[p], p = (i-1)*pda+j-1, i=k,...,l and j=m,...,n},
```

for row major ordering.

If the data is to be stored using column-major ordering and we have declared an array variable as

```
double a[dim1][dim2];
```

then the element indexed ij is effectively transposed. That is, the element `a[i][j]` under row-major ordering is the element `a[j][i]` under column-major ordering.

As another alternative you may choose to malloc the required memory as in [1] above. In this case, the element indexed ij is using the pointer notation `*(a+j*dim1+i)`; or by using the array notation `a[j*dim1+i]`; or by using `A(I,J)` if the macro is defined as

```
#define A(I,J) A[J*dim1+I]
```

Note the difference in definition between [2] and [4] above.

In order to simplify the documentation, we refer to array elements using the macro definitions above. Further, we note that in the preprocessor directive [2] and [4], the critical dimension is either `dim2` if the row-major ordering is used or `dim1` if column-major ordering is used. We designate either `dim2` or `dim1` as the principal dimension depending on the storage ordering scheme. The principal dimension can be thought of as a stride which must be taken to traverse either a row or a column depending on the order argument.

Typically in the NAG C Library we use the convention '**pda**' if the function has the **order** argument and **pda** can have different values depending on whether the array is stored in row major or in column major order. If the specification of an array is such that its elements must be stored in row major order we use the term '**tda**' (meaning trailing dimension). For arrays whose elements must be stored in column major order we use the term '**lda**' (meaning leading dimension). As of Mark 24, new functions in the library generally store two-dimensional data in the column order. We are standardizing on the phrase 'stride separating row elements' to mean column order storage and the phrase 'stride separating column elements' to mean row order storage.

In order to facilitate calling functions in which data has to be stored in a mutually exclusive manner, such as for example, function A requires data to be in row order and while function B requires data to be stored in column order and the 'order' parameter has not been provided, then functions provided in the f16 chapter will have to be used. For example, nag_dge_copy (f16qfc) can be used to change from row order to column order by performing a transposed copy. Functions are available in this chapter for performing a variety of copying tasks such as triangular copy, etc..

We illustrate these concepts using two examples. In the first example, memory is allocated while in the second example memory is declared. In both examples, row or column modes are demarcated using the preprocessor macro `NAG_ROW_MAJOR`. Memory is allocated using `NAG_ALLOC`, which is a macro defined in `nag_stdlib.h`, in preference to explicit malloc calls. This macro maps to calls to internal Library functions to allocate memory. Also note the use of `NAG_FREE` to free the memory.

Example 1

```
/* Example Program, with memory allocated, based on:
 *
 * nag_dorgqr (f08afc) Example Program.
 *
 * Copyright Numerical Algorithms Group.
 *
 */

#include <stdio.h>
#include <string.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf08.h>
#include <nagx04.h>

int main(void)
{
  /* Scalars */
  Integer  i, j, m, n, pda_row, pda_column, tau_len;
  Integer  exit_status=0;
  NagError fail;

  /* Arrays */
  char   *title=0;
  double *a_row=0, *a_column=0, *tau=0;
  char matrx_data [] = {
    " -0.57  -1.28  -0.39   0.25 "
    " -1.93   1.08  -0.31  -2.14 "
    "  2.30   0.24   0.40  -0.35 "
    " -1.93   0.64  -0.66   0.08 "
    "  0.15   0.30   0.15  -2.13 "
    " -0.02   1.03  -1.43   0.50 "
  }, *matrix_data_ptr = matrx_data;

  /* Initialize strtok */
  matrix_data_ptr = strtok(matrix_data_ptr, " \t\n");
```

```
#define A_COLUMN(I,J) a_column[(J-1)*pda_column + I - 1]
#define A_ROW(I,J) a_row[(I-1)*pda_row + J - 1]

  INIT_FAIL(fail);

  m = 6;
  n = 4;;
  pda_column = m;
  pda_row = n;
  tau_len = MIN(m, n);

  /* Allocate memory */
  if ( !(title = NAG_ALLOC(31, char)) ||
       !(a_row = NAG_ALLOC(m * n, double)) ||
       !(a_column = NAG_ALLOC(m * n, double)) ||
       !(tau = NAG_ALLOC(tau_len, double)) )
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

#ifdef NAG_ROW_MAJOR
  printf("Using row major storage, allocated memory\n");
  /* Read A from data above */
  for (i = 1; i <= m; ++i)
    {
      for (j = 1; j<= n; j++)
        {
          sscanf(matrix_data_ptr, "%lf", &A_ROW(i,j));
          matrix_data_ptr = strtok(0, " \t\n");
        }
    }

  /* Compute the QR factorization of A */
  f08aec(Nag_RowMajor, m, n, a_row, pda_row, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08aec.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* Form the leading N columns of Q explicitly */
  f08afc(Nag_RowMajor, m, n, n, a_row, pda_row, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08afc.\n%s\n", fail.message);
      exit_status = 2;
      goto END;
    }
  /* Print the leading N columns of Q only */
  sprintf(title, "The leading %2ld columns of Q\n", n);
  x04cac(Nag_RowMajor, Nag_GeneralMatrix, Nag_NonUnitDiag, m, n,
 a_row, pda_row, title, 0, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from x04cac.\n%s\n", fail.message);
      exit_status = 3;
      goto END;
    }
#else
  printf("Using column major storage, allocated memory\n");
  /* Read A from data above */
  for (i = 1; i <= m; ++i)
    {
      for (j = 1; j<= n; j++)
        {
          sscanf(matrix_data_ptr, "%lf", &A_COLUMN(i,j));
          matrix_data_ptr = strtok(0, " \t\n");
```

```
          }
      }

    f08aec(Nag_ColMajor, m, n, a_column, pda_column, tau, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from f08aec.\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Form the leading N columns of Q explicitly */
    f08afc(Nag_ColMajor, m, n, n, a_column, pda_column, tau, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from f08afc.\n%s\n", fail.message);
        exit_status = 2;
        goto END;
      }
    /* Print the leading N columns of Q only */
    sprintf(title, "The leading %2ld columns of Q\n", n);
    x04cac(Nag_ColMajor, Nag_GeneralMatrix, Nag_NonUnitDiag, m, n,
           a_column, pda_column, title, 0, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from x04cac.\n%s\n", fail.message);
        exit_status = 3;
        goto END;
      }
#endif
 END:
    if (title) NAG_FREE(title);
    if (a_row) NAG_FREE(a_row);
    if (a_column) NAG_FREE(a_column);
    if (tau) NAG_FREE(tau);

    return exit_status;
}
```

Example 2

```
/* Example Program, with memory declared, based on:
 *
 * nag_dorgqr (f08afc)) Example Program.
 *
 * Copyright Numerical Algorithms Group.
 *
 */

#include <stdio.h>
#include <string.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf08.h>
#include <nagx04.h>

#define MMAX 10
#define NMAX 8

int main(void)
{
  /* Scalars */
  Integer  i, j, m, n, pda, tau_len;
  Integer  exit_status=0;
  NagError fail;

  /* Arrays */
  char   title[30];
  double a_row[MMAX][NMAX], a_column[NMAX][MMAX], tau[NMAX];
  char matrx_data [] = {
    " -0.57  -1.28  -0.39   0.25 "
    " -1.93   1.08  -0.31  -2.14 "
```

```
    "  2.30   0.24   0.40  -0.35 "
    " -1.93   0.64  -0.66   0.08 "
    "  0.15   0.30   0.15  -2.13 "
    " -0.02   1.03  -1.43   0.50 "
  }, *matrix_data_ptr = matrx_data;

  /* Initialise strtok */
  matrix_data_ptr = strtok(matrix_data_ptr, " \t\n");


  INIT_FAIL(fail);

  m = 6;
  n = 4;;
  pda = NMAX;
  tau_len = MIN(m, n);

  /* Read A from data above */
#ifdef NAG_ROW_MAJOR
  for (i = 0; i < m; ++i)
    {
      for (j = 0; j< n; j++)
        {
          sscanf(matrix_data_ptr, "%lf", &a_row[i][j]);
          matrix_data_ptr = strtok(0, " \t\n");
        }
    }

  /* Compute the QR factorization of A */
  printf("Using row major storage, declared memory\n");
  f08aec(Nag_RowMajor, m, n, &a_row[0][0], pda, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08aec.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* Form the leading N columns of Q explicitly */
  f08afc(Nag_RowMajor, m, n, n, &a_row[0][0], pda, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08afc.\n%s\n", fail.message);
      exit_status = 2;
      goto END;
    }
  /* Print the leading N columns of Q only */
  sprintf(title, "The leading %2ld columns of Q\n", n);
  x04cac(Nag_RowMajor, Nag_GeneralMatrix, Nag_NonUnitDiag, m, n,
 &a_row[0][0], pda, title, 0, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from x04cac.\n%s\n", fail.message);
      exit_status = 3;
      goto END;
    }
#else
  printf("Using column major storage, declared memory\n");
  for (i = 0; i < m; ++i)
    for (j = 0; j< n; j++)
      {
/* Note column data is transposed */
sscanf(matrix_data_ptr, "%lf", &a_column[j][i]);
matrix_data_ptr = strtok(0, " \t\n");
      }

  f08aec(Nag_ColMajor, m, n, &a_column[0][0], pda, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08aec.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
```

```
    }
  /* Form the leading N columns of Q explicitly */
  f08afc(Nag_ColMajor, m, n, n, &a_column[0][0], pda, tau, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from f08afc.\n%s\n", fail.message);
      exit_status = 2;
      goto END;
    }
  /* Print the leading N columns of Q only */
  sprintf(title, "The leading %2ld columns of Q\n", n);
  x04cac(Nag_ColMajor, Nag_GeneralMatrix, Nag_NonUnitDiag,  m, n,
 &a_column[0][0], pda, title, 0, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from x04cac.\n%s\n", fail.message);
      exit_status = 3;
      goto END;
    }
#endif
 END:
  return exit_status;
}
```

3.2.1.5 Internal data structures in the NAG C Library

For efficiency, and wherever possible, the NAG C Library is designed to make use of the Basic Linear Algebra Subprograms (BLAS), a suite of low-level functions tuned by many computer manufacturers for their particular hardware. Since the BLAS are specified in Fortran, and therefore use the column-major storage order, the NAG C Library also uses this scheme, internally, for new functions wherever it is practical. Thus any two-dimensional arrays you have provided may be re-ordered on entry and/or on exit from the NAG functions, as appropriate. It is therefore slightly more efficient to use the column-major ordering; however, except for very large data sets, the effect is negligible in practice.

3.2.1.6 Chapter header files

Chapter header files contain the function declarations for the NAG C Library with ANSI function prototyping. The appropriate chapter header file must be included for each NAG function called by your program. For example, to call the function nag_sum_fft_complex_1d (c06pcc) the chapter header file `nagc06.h` must be included as

```
#include <nagc06.h>
```

The naming convention is to prefix the first three characters of the function name in lower case by `nag` and use `.h` as the postfix as in normal C practice, except that all functions in Chapter s use the header file `nags.h`.

(a) Header files intended for your inclusion within calling programs to the NAG C Library

`nag.h` defines the basic environment for use of the NAG C Library. This header file must be included in each calling program to the NAG C Library and must precede all other header files that are included. This must be followed by one or more of the following chapter header files.

```
naga00.h  naga02.h  nagc02.h  nagc05.h  nagc06.h  nagc09.h
nagd01.h  nagd02.h  nagd03.h  nagd04.h  nagd05.h  nagd06.h
nage01.h  nage02.h  nage04.h  nage05.h  nagf01.h  nagf02.h
nagf03.h  nagf04.h  nagf06.h  nagf07.h  nagf08.h  nagf11.h
nagf12.h  nagf16.h  nagg01.h  nagg02.h  nagg03.h  nagg04.h
nagg05.h  nagg07.h  nagg08.h  nagg10.h  nagg11.h  nagg12.h
nagg13.h  nagh02.h  nagh03.h  nagm01.h  nags.h    nagx01.h
nagx02.h  nagx04.h  nagx07.h
```

`nag_stdlib.h` defines the memory allocation macro `NAG_ALLOC` and `NAG_FREE`. You must include this header file if the NAG definitions of `NAG_ALLOC` and `NAG_FREE`, as used in the example programs, are required.

(b) The following three header files are included by `nag.h` (you do not need to supply a specific statement to include them)

`nag_types.h` defines the NAG types used in the Library.

`nag_errlist.h` defines the NAG error codes and messages used in the Library.

`nag_names.h` maps the NAG long names to short names.

3.3 Use of NAG Long Names

The long names defined in the header file `nag_names.h` are #defines. You should note that the short function names given in upper case in this file are also #defines and therefore their corresponding long names will not require a terminating pair of brackets. These declarations are to be found in `nagx01.h` and `nagx02.h`. As the header file `nag_names.h` is already included via `nag.h`, you need not include `nag_names.h` in their calling programs.

3.4 Input/Output

NAG C Library functions output all error and warning messages to the C standard error stream `stderr`. Chapters e04, e05, g02 and g13 will optionally output results to the C standard output stream `stdout` or to an alternative user-specified file. A number of functions in Minimizing or Maximizing a Function (Chapter e04) and Operations Research (Chapter h) read input from external files.

3.5 Auxiliary Functions

In addition to the documented functions, the NAG C Library contains a much larger number of auxiliary functions. You do not normally need to concern yourself with these functions, as they will automatically be called as required by the user-callable function you have selected.

3.6 NAG Error Handling and the fail Argument

All functions that have error exits have an argument that allows you control over the printing of error messages when an error is detected. There is a further option which allows you to either continue running your program, having returned from the NAG function, or to stop with either an exit statement or an abort within the NAG function. The different ways of using these error handling facilities are described below.

Note that in some implementations, the Library is linked with the vendor library containing LAPACK functions and the Chapters f07 and f08 function interfaces, where appropriate, act as wrappers to the corresponding vendor LAPACK functions. In this case, the **fail** argument passed through the Chapter f07 and Chapter f08 interfaces does not have full control over the printing of error messages; nor does it determine whether or not control is returned to the calling program when an error is detected.

3.6.1 Use of `NAGERR_DEFAULT`

The simplest method of using the error handling facility is to put `NAGERR_DEFAULT` in place of the **fail** argument in calls to the NAG C functions. If an error is detected the appropriate NAG error message is output on `stderr` and the program is stopped by the use of `exit`.

3.6.2 Use of the fail argument

The two remaining ways of using the NAG error handling facility both involve defining the **fail** argument in the calling program. The **fail** argument is of type `NagError` which is a structure defined in `nag_types.h` as:

```
typedef struct {
  int code;
```

```
    Nag_Boolean print;
    char message[NAG_ERROR_BUF_LEN];
    Integer errnum;
    void (*handler)(char*,int,char*);
} NagError;
```

where the symbol NAG_ERROR_BUF_LEN is normally defined to be 512.

This structure will contain the NAG error code and message on return from a call to a NAG C Library function. The NAG error codes and some associated NAG error messages are defined in nag_errlist.h. A detailed description of the individual members of this structure is given below (see Section 3.6.3).

The NAG error argument **fail** is declared in the calling program as:

```
NagError fail;
```

The address of the argument is then passed to the NAG C function being called. Relevant members of the structure must be initialized before passing the argument to the called function, even though you may not actually require all members. It is recommended that the NAG defined macro INIT_FAIL be used for this purpose.

The INIT_FAIL macro sets:

fail.**code** = NE_NOERROR

fail.**print** = Nag_FALSE

fail.**errnum** = 0

fail.**handler** = 0

The SET_FAIL macro is also available. It sets the contents of **fail** in the same way as INIT_FAIL except that **fail**.**print** is set to Nag_TRUE.

(a) Use of the fail argument with the print member set to Nag_TRUE

If you require that the NAG error message be printed when an error is found, but that the called function should return control to the calling program, then the **fail** argument must be declared with all members initialized and the **print** member set to Nag_TRUE. Use of the NAG-defined macro SET_FAIL with the statement SET_FAIL(fail); performs the appropriate assignments. Alternatively the initialization could be done by declaring the **fail** argument with **static** and then setting **fail.print** to Nag_TRUE.

If no error occurs, **fail.code** will contain the error code NE_NOERROR on return from the called function. However, if an error is found, the appropriate NAG error message will be output on stderr before returning control to the calling program; **fail.code** will contain the relevant NAG error code. You must ensure that the calling program tests the **code** member of the **fail** argument on return from the NAG C function; you may then choose whether to exit the calling program or continue. See the example program for nag_zero_nonlin_eqns_easy (c05qbc) for such a case. The option of continuing may be advantageous if the results being returned are of some value even when an error has been detected. In the case of nag_zero_nonlin_eqns_easy (c05qbc) the code could be altered to allow the program to continue if the specific error codes **NE_TOO_MANY_FEVALS**, **NE_TOO_SMALL** and **NE_NO_IMPROVEMENT** occur, as in such a case useful partial results are returned (see the function document for nag_zero_nonlin_eqns_easy (c05qbc)).

(b) Use of the fail argument with the print member set to Nag_FALSE

If you do not wish the NAG error messages to be printed automatically when an error is found then the **fail** argument must be declared with all members initialized and the **print** member set to Nag_FALSE. Use of **static** in the declaration of **fail** will automatically leave the **print** member as Nag_FALSE as will the use of INIT_FAIL(fail).

This method is suitable for those of you who wish to produce your own error messages rather than use the NAG C Library versions. Alternative error messages may be coded directly into the calling program or be produced via a user-written error-handling function which is assigned to the **handler** member of the **fail** argument (see the description of the handler member below).

3.6.3 The `NagError` structure

The individual members of the `NagError` structure are described in full below.

`code`

On successful exit, `code` contains the NAG error code NE_NOERROR; if an error or warning has been detected, then `code` contains the specific error or warning code. Error codes are prefixed with NE_ whereas warning codes have the prefix NW_.

`print`

`print` must be set before calling any NAG C Library function with a **fail** argument. It should be set to Nag_TRUE if the NAG error message is to be printed, otherwise Nag_FALSE. It is not changed by the NAG C Library function.

`message`

On successful exit the array `message` contains the character string `"NE_NOERROR:\n No error"`. If an error has been detected, then `message` contains the error message text, whether or not this is printed.

`errnum`

On successful exit, `errnum` is unchanged. For certain error or warning exits `errnum` will contain a value specifying additional information concerning the error. For example if a vector is supplied incorrectly, then `errnum` may specify which component of the vector is wrong. Cases where `errnum` returns information are described in the relevant function documents.

`handler`

`handler` must be set to `0` if control is to be returned to the calling function after an error has been detected. Otherwise it must point to a user-supplied error-handling function. An example of the ANSI C declaration of a user-supplied error function (here called `errhan`) is:

```
void errhan(const char *string, int code, const char *name)
```

where `string` contains the NAG error message on input, `code` is the NAG error code and `name` is the short name of the NAG C Library function which detected the error. If `print` (see above) is Nag_TRUE, then the NAG error message is printed before the user-supplied error handler is called. If the user-supplied error handler returns control, then the NAG error handler will return control to the calling program; otherwise the user-supplied error handler may exit.

An elementary example of where this feature might be used is if it is preferred to print error messages on `stdout` rather than the default `stderr`. In this case `errhan` could be defined as:

```
void errhan(const char *string, int code, const char *name)
{
  if (code != NE_NOERROR)
    {
      printf("\nError or warning from %s.\n", name);
      printf("%s\n", string);
    }
}
```

3.6.4 Structure of the NAG error messages

For illustrative purposes, let us consider two examples of the format of the NAG C Library error messages, in the NAG C Library documentation:

NE_INT

On entry, N = ⟨*value*⟩.
Constraint: N > 1.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

If the NAG function in question detects an error and error messages are being displayed, either by using the default error handler `NAGERR_DEFAULT` or by setting **fail**.**print** = Nag_TRUE, then text of the following form would be displayed:

```
NE_INT:
  On entry, n = 1
  Constraint: n > 1.
```

or

```
NE_BAD_PARAM:
  On entry, argument order had an illegal value.
```

i.e., 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.

3.7 Thread Safety

The NAG C Library is thread safe by design; all communication between functions is via argument lists.

Although the NAG C Library is thread safe, care must be taken to use it in a thread safe way. In particular, this is the case with Chapter c05 (Roots of One or More Transcendental Equations), and Chapter d01 (Quadrature). In these chapters, the solution of a problem usually involves a user-supplied function being passed to the NAG Library function, the user-supplied C function being called to evaluate the mathematical function which is the object of the problem. It is sometimes the case that you would like the evaluation function to communicate with the main program, to pass relevant information involved in the function evaluation. The simplest way to do this is to use global variables for communication, but these global variables render the resulting code unsuitable for use by multiple threads simultaneously.

Using pthread or Microsoft thread package functionality, it is perfectly possible for you to avoid this problem by writing code which passes thread-specific data between main program and evaluation function. As an alternative to this method, however, where relevant we have provided alternative versions of the NAG Library functions. These alternative functions have an extra argument which can be used for safe communication of additional data if it is required.

Finally we recommend that when using the C Library error mechanism, the output is switched off (by setting $fail.print$ = Nag_FALSE).

Note 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.

3.8 Parallelism

3.8.1 Introduction

The time taken to execute a function from the NAG Fortran 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 functions 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 had until recently 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 maximise 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 parallelised 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 can depend on efficiency of implementation and use of Application Programming Interfaces (APIs), and can vary depending on 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 $n \to \infty$, $S_n \leq 4$.

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 the OpenMP API as described in Section 3.8.2.

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.) Once the parallel work has been completed the threads return control to the parent process and become inactive (or "join") until the next region of parallel 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. For shared variables, thread safety is an issue. A program is deemed to be "thread safe" if it can be executed using two or more threads without compromising results. Thread safe programs should return equally valid results no matter how many threads are used in the parallel regions. However, that is not to say that identical results can be guaranteed, or 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 3.9.

3.8.2 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 the OpenMP API (version 3.0), 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 functions are parallelised; users should check Section 8 of the function documents to find details about parallelism and performance of functions of interest.

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

1. The function being called is a NAG-specific function that has been threaded using OpenMP, or that internally calls another NAG-specific function that is threaded.
2. The function being called calls through to the vendor library (e.g., Intel MKL, AMD ACML, IBM ESSL, Oracle Sunperf, etc.). This happens if the function is not specific to the NAG library, and the vendor library offers superior parallel performance and equivalent numerical properties. For example, most BLAS and LAPACK functions fall into this category. 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 documentation for the vendor library for further information.

A complete list of all the functions in the NAG Library, and their threaded status is given in the Multithreaded Functions document.

It is informative for users 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 function 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 function call (although there are situations in which the teams of threads may be made available to orphaned directives in user code via user-supplied subprograms, see Section 8 of the function documents for further information). Furthermore, OpenMP constructs within NAG functions bind to teams of threads created within the NAG code (i.e., there are no orphaned directives). For threaded NAG-specific functions all thread management is performed by the OpenMP run-time and NAG does not provide any extra threading controls or options. Thus all OpenMP environment variables and function settings apply equally to calls to these NAG functions and to users' own parallel regions. In particular, users should take care when calling these NAG functions from within their own parallel regions, since if nested parallelism is enabled (it is disabled by default) the NAG function 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 that which the hardware is capable of supporting, or if some hardware resources are busy executing other processes (which may belong to other users in a shared system). For these reasons users should be aware of the maximum number of threads supported in hardware and the workload of their machine, and use this information in selecting a number of threads which minimises contention on resources. Please read the Users' Note for advice about setting the number of threads to use, or contact NAG for advice.

If you are calling multithreaded NAG functions 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 library since, although many functions have not been the focus of parallel development by NAG, they may benefit by calling functions in the library that have, and/or by calling parallel vendor functions (e.g., BLAS, LAPACK). Thus, the performance improvement due to multithreading, if any, will vary depending upon which function is called, problem sizes and other parameters, system design and operating system configuration. If you frequently call a function 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 functions 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

3.9 Calling the Library from Other Languages

In general the NAG C Library can be called from other computer languages (such as C++, C#, Java, Visual Basic and Python) provided that appropriate mappings exist between the NAG C Library data types and their data types (see http://www.nag.co.uk/numeric/CL/classocinfo.asp).

3.10 Arithmetic Considerations and Reproducibility of Results

The results obtained when calling a NAG Library function 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 functions can only return an approximation to the true solution, simply due to accumulation of rounding errors.

It should therefore be clear that running a program which calls a NAG Library function 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 function that searches for a mininum of an optimization problem may converge to a different local minimum, but in general, so long as the function’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 functions.

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 function 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 function 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 function, 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 the user of the program depends on always getting identical results for the same data.

On even newer hardware, **AVX** instructions use 256-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 functions 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 functions where available – but that is not always possible since it partly depends on the support of the compiler.

Of course, no Library function has control over memory you have allocated before being passed to the function. 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.

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 functions 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 functions 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 consult NAG for advice.

4 Using the Documentation

4.1 Using the Manual

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

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

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

to give all the information needed to call a NAG Library function correctly from a C 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 'NAG C Library News, Mark 24' provides details of new functions added, details of functions scheduled for withdrawal and details of functions withdrawn at this mark.

The document entitled 'Functions Withdrawn or Scheduled for Withdrawal' provides full details of all functions withdrawn from the NAG C Library and the document entitled 'Advice on Replacement Calls for Withdrawn/Superseded Functions' provides advice on how to modify your program.

The online documentation includes a Keyword and GAMS Search which provides you with a form to search the Library for keywords and function names.

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

When you have chosen a function, you must consult the **function document**. Each function 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 function.

4.2 Structure of Function Documents

All function documents have the same structure consisting of nine numbered sections:

1. **Purpose**
2. **Specification**
3. **Description**
4. **References**
5. **Arguments** (see Section 4.3 below)
6. **Error Indicators and Warnings**
7. **Accuracy**
8. **Parallelism and Performance**
9. **Further Comments**
10. **Example** (see Section 4.4 below)

In some documents (notably Chapters e04, e05 and h) there are a further three sections:

11. **Algorithmic Details**
12. **Optional Arguments**

13. Description of Monitoring Information

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

4.3 Specification of Arguments

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

4.3.1 Classification of arguments

Arguments are classified as follows.

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

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

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

Communication Structure and Arrays: arguments which are used to communicate data from one function call to another.

External Function: a function 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 function, 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.
>
> *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.

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 function is called with an invalid value for the argument (e.g.,$n = 0$), the function will usually take an error exit.

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.

4.4 Example Programs and Results

The **example program** in Section 9 of most function documents illustrates a simple call of the function. 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. These programs can also be obtained by using the nagc_example scripts, provided with the product. 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 from the generic form.

These example programs contain a number of preprocessor identifiers such as `NAG_CALL` and `NAG_IFMT` to enable cross platform portability. These identifiers are subsequently replaced by appropriate implementation specific tokens via the header files `nag.h` or `nag_types.h`, using the C preprocessor #defines.

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 function 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.

5 Support from NAG

NAG Technical Support Service

The NAG Technical Support Service is available for general enquiries from all users and also for technical queries from sites that subscribe to the support service.

The service is availabel during office hours, but contact is possible by email and telephone (answering machine) at all times. Please see the Users' Note or the NAG web site for contact details.

When contacting the NAG Technical Support Service, it helps us to deal with your query quickly if you can quote your NAG customer reference number and NAG product code.

NAG Web Site

The NAG web site is an information service providing items of interest to users and prospective users of NAG products and services. The information is regularly updated and reviewed, and includes implementation availability, descriptions of products, downloadable software, case studies, industry articles and technical reports. The NAG web site can be accessed via:

NAG UK at http://www.nag.co.uk

NAG North America at http://www.nag.com

NAG Japan at http://www.nag-j.co.jp

NAG Taiwan at http://www.nag-gc.com

6 Background to NAG

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).

7 References

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

Al–Mohy A H and Higham N J (2011) Computing the action of the matrix exponential, with an application to exponential integrators *SIAM J. Sci. Statist. Comput.* **33(2)** 488-511

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/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/IEC (1990) Information technology – programming language C *Current C Language Standard* ISO/IEC 9899:1990

Kernighan B W and Ritchie D M (1988) *The C Programming Language* (2nd Edition) Prentice–Hall

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