NAG C Library

Essential Introduction

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

Contents

1 The Library and its Documentation

1.1 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.,

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

c06ebc

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 (except the Linear Algebra Support Functions in Chapter f16) also has a more meaningful and longer name, for example

nag_fft_hermitian

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 2.3 for further details.

1.2 Structure of the Documentation

The NAG C 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., Contents – d01;

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

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. Exceptionally, some chapters (Chapters x01 and x02) do not have individual function documents; instead, these functions are described together in the Chapter Introduction. It should be noted that all the LAPACK computational functions from Release 3 are included in the NAG C Library and can be called by their LAPACK name, although not all of these functions are currently documented in Chapters f07 and f08.

Currently NAG provides documentation in the form of PDF files accompanied by an HTML index. Introductory material is provided in both PDF and HTML but the Keyword Index and GAMS Classification Index are provided in HTML only. It is anticipated that future releases will provide XHTML files (taking advantage of technology that is currently being developed, e.g., MathML) for the full manual with fully linked PDF files for all but the Keyword Index and GAMS Classification Index. The most up-to-date version of the documentation is accessible via the NAG web site (http://www.nag.com) (see Section 4).

1.3 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 Sun Solaris 32-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 (e.g., the PCs running Windows); it may also be specific to a particular operating system, C compiler, 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.

1.4 Precision of the Library

The NAG C Library is developed in **double precision** only.

1.5 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** 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 Library function nag_implementation_details (a00aac).

This function has no arguments; it simply outputs text to the standard output.

1.6 C Language Standards

All functions in the NAG C Library conform fully to ANSI C, and functions introduced since Mark 6 take advantage of the `const` keyword to allow for safer code which can be more easily optimised. Although pre-Mark 6 functions have not yet been modified, it is our intention to extend the use of `const` and other features of the C99 standard in future releases.

2 Using the Library

2.1 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 2 is concerned with (b).

2.2 Programming Advice

The Library and its documentation are designed on the assumption that you know how to write a calling program in C.

When a suitable NAG function has been selected, the function must be called from the C Library via a suitable user-written C 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 3.2).

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 refer to the above example calling program while studying the description of these features. You are also recommended to pay particular attention to the specification of the function arguments, array sizes and array indices.

2.2.1 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 2.2.1.6. 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 normally located in the standard directory for C include files. The exact location is installation dependent; 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. File `nag.h` also contains the definitions for the input/output and string handling functions `Vscanf`, `Vprintf`, `Vfprintf`, `Vsprintf`, `Vstrcpy` which are the C functions `scanf`, `printf`, etc., cast to void.

You may also need to include the header file `nag_stdlib.h` in the calling program; see Section 2.2.1.6.

2.2.1.1 NAG data types

Integer

This data type is used for almost all integer arguments to NAG C Library functions. It is normally defined to be `long`.

Nag_Boolean

This is an enumeration type defined as

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

This replaces the existing Boolean type. In order that you can continue to use your existing program, a preprocessor define CL07_COMPATABILITY has been introduced. This feature can be used from the command line by specifying

```
-D CL07_COMPATABILITY
```

Complex

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

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

Pointer

This data type represents a generic pointer and is defined as `void *`.

Nag_User

This data type is a structure containing a generic pointer 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. A brief example of use (taken from Chapter d02) is given below:

```
struct user
{ double xend, h;
  Integer k;
};

main()
{
  Integer neq = 3;
  double x = 0.0, tol = 0.0001, y[3] = {1.0, 0.0, 0.0};
  Nag_User comm;
  struct user s;
  /* assign address of user defined structure to comm.p */
  comm.p = (Pointer)&s;
  s.xend = 10.0;
   s.k = 4;
   s.h = (s.xend-x) /(double)(s.k+1);
   d02ejc(neq, fcn, 0, &x, y, s.xend, tol, Nag_Relative,
   out, 0, &comm, NAGERR_DEFAULT);
 }
```

```
static void out(Integer neq, double *xsol, double y[], Nag_User *comm)
{
  Integer j;
  struct user *s = (struct user *)comm->p;

  Vprintf("%8.2f", *xsol);
  for (j=0; j<3; ++j)
    Vprintf("%13.5f", y[j]);
  Vprintf ("\n");
  *xsol = s->xend - (double)s->k * s->h;
  s->k--;
}
```

Nag_Comm

This is also a communication structure with a generic pointer with usage similar to Nag_User. In addition it has double* and Integer* pointers to allow double and Integer arrays to be communicated with no casting.

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 enumerated types are defined in <nag_types.h> for use in calls to various NAG C Library functions. You must use these enumerated types in your calling programs.

Other structure types

A number of structures have been defined to facilitate calls to NAG functions. The components of a structure that are relevant to a call to a 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. For futher details please contact NAG.

2.2.1.2 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 2.6).

The macros NAG_ALLOC 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_FREE frees memory allocated by NAG_ALLOC; its single argument is the pointer which specifies the memory to be deallocated. The statement

```
NAG_FREE(p);
```

deallocates memory pointed to by p.

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 its use, see Section 9.1 of the document for nag_1d_quad_inf_wt_trig (d01asc).

2.2.1.3 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 77, arrange two-dimensional arrays by column ('column-major' ordering). Commencing at Mark 7, those functions in the 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 their 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 which supports column-major ordering.

2.2.1.4 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 a 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. The C pre-processor allows a succinct notation for computing this explicit address by using a macro definition:

```
#define A(I,J) A[I*dim2+J]
```
[1]

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));
```
[2]

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 [1] is already defined.

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 [2] 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]
```
[3]

Note the difference in definition between [1] and [3] above.

In order to simplify the documentation, we refer to array elements using the macro definitions above. Further, we note that in the pre-processor directive [1] and [3], 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.

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 pre-processor macro `NAG_ROW_MAJOR`.

Example 1

```
/* Example Program, with memory allocated, based on:
 *
 * nag_dorgqr (f08afc) Example Program.
 *
 * Copyright 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 */

#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;

  /* Initialise 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)) )
    {
      Vprintf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

#ifdef NAG_ROW_MAJOR
  Vprintf("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)
      {
        Vprintf("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)
      {
        Vprintf("Error from f08afc.\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Print the leading N columns of Q only */
    Vsprintf(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)
      {
        Vprintf("Error from x04cac.\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
#else
    Vprintf("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)
      {
        Vprintf("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)
      {
        Vprintf("Error from f08afc.\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Print the leading N columns of Q only */
    Vsprintf(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)
      {
        Vprintf("Error from x04cac.\n%s\n", fail.message);
        exit_status = 1;
        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 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 */

#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[MMAX][NMAX], 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 */
  Vprintf("Using row major storage, declared memory\n");
  f08aec(Nag_RowMajor, m, n, &a_row[0][0], pda, tau, &fail);
```

```
  if (fail.code != NE_NOERROR)
    {
      Vprintf("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)
    {
      Vprintf("Error from f08afc.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* Print the leading N columns of Q only */
  Vsprintf(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)
    {
      Vprintf("Error from x04cac.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
#else
  Vprintf("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)
    {
      Vprintf("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)
    {
      Vprintf("Error from f08afc.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  /* Print the leading N columns of Q only */
  Vsprintf(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)
    {
      Vprintf("Error from x04cac.\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
#endif
 END:
  return exit_status;
}
```

2.2.1.5 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.

2.2.1.6 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_fft_complex (c06ecc) 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>  <nagd01.h>
<nagd02.h>  <nagd03.h>  <nagd06.h>  <nage01.h>  <nage02.h>  <nage04.h>
<nagf01.h>  <nagf02.h>  <nagf03.h>  <nagf04.h>  <nagf06.h>  <nagf07.h>
<nagf08.h>  <nagf11.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>
```

`<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.

2.3 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.

2.4 Input/Output

NAG C Library functions output all error and warning messages to the C standard error stream `stderr`. Chapters e04, 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 the Minimizing or Maximizing a Function (e04) and Operations Research (Chapter h) areas read input from external files.

2.5 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. The function declarations of these

auxiliary functions can be found in the relevant chapter header files together with the user-callable function declarations.

2.6 NAG Error Handling and the fail Argument

All functions that have error exits have a 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 f07 and 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.

2.6.1 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 `abort` (in some implementations the program may be stopped with `exit` rather than `abort`). This method of use is illustrated in the above example program for nag_real_eigensystem (f02agc). `NAGERR_DEFAULT` is defined in `<nag.h>` as `(NagError *)0`.

2.6.2 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 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 2.6.3).

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. All 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 `SET_FAIL` macro is also available. This sets **fail.print** 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** (see the example declaration above) 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_real_svd (f02wec) 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_real_svd (f02wec) the code could be altered to allow the program to continue if the specific error code of **NE_QR_NOT_CONV** occurs, as in such a case useful partial results are returned (see the function document for nag_real_svd (f02wec)).

(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).

2.6.3 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 abort, exit or longjmp.

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)
    {
```

```
        Vprintf("\nError or warning from %s.\n", name);
        Vprintf("%s\n", string);
      }
  }
```

2.7 Thread Safety

The NAG C Library is thread safe by design; all communication between functions is via argument lists, and, with the exception of Chapter g05 (Random Number Generators), no use of read and write static storage is made.

In Chapter g05, for efficiency, functions in the range g05c–g05h use global variables internally for communication. Simultaneous access to these global variables by multiple threads has been guarded against using thread mutexes. For UNIX machines, these mutexes come from the POSIX standard thread library (ANSI/IEEE POSIX (1995)), also known as pthreads. For PC implementations of the NAG C Library, the mutexes are those implemented by the Microsoft thread package.

The NAG C Library is therefore thread safe with respect to the thread package against which it was built, either pthreads or Microsoft threads. Some older UNIX architectures did not have implementations of pthreads available, and implementations of the NAG C Library on such architectures could not therefore be made thread safe – though of course on such machines it would not be possible to write a POSIX multi-threaded program anyway. Current implementations of the NAG C Library that are NOT thread safe are:

CLSG506DA

Although the NAG C Library is thread safe, sometimes 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. Please see documentation for nag_1d_quad_gen (d01ajc) and nag_1d_quad_gen_1 (d01sjc) for examples.

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

Note that in some implementations, the Library is linked with one or more vendor libraries to provide, for example, efficient BLAS routines. NAG cannot guarantee that any such vendor library is thread safe.

2.8 Calling the Library from Other Languages

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

3 Using the Documentation

3.1 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 'Mark 8 News' provides details of new functions added, details of documents scheduled for withdrawal and details of documents withdrawn at this mark.

The document entitled 'Library Contents' (a structured list of functions in the Library, by chapter) may help you to find the chapter, and possibly the function, which you need to solve your problem.

The document entitled 'Withdrawn Routines' provides full details of all functions withdrawn from the NAG C Library and the document entitled 'Advice on Replacement Calls for Withdrawn/Superseded Routines' provides details of the minimum change necessary to allow you to take full advantage of new functionality provided by replacement functions.

The online documentation provides you with a fully linked HTML Keyword Index (a keyword index to functions) and GAMS Classification Index (a list of NAG functions classified according to the GAMS scheme).

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.

3.2 Structure of Function Documents

All function documents have the same structure, with the exception of Chapter f16, consisting of nine numbered sections:

1. **Purpose**
2. **Specification**
3. **Description**
4. **References**
5. **Arguments** (see Section 3.3 below)
6. **Error Indicators and Warnings**
7. **Accuracy**
8. **Further Comments**
9. **Example** (see Section 3.4 below)

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

10. **Algorithmic Details**
11. **Optional Arguments**
12. **Description of Monitoring Information**

The Chapter f16 function documents currently consist of:

1. **Purpose**
2. **Specification**
3. **Arguments** (see Section 3.3 below)
4. **Error Indicators and Warnings**

3.3 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.

3.3.1 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 on or before entry to the function; the function may assign values to them.

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

Communication Structure: array arguments which are used to communicate data from one subfunction call to another.

External Procedure: a subfunction or 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.

3.3.2 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*: $\mathbf{n} > 0$.

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

The phrase '*Suggested Values*:' 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.

3.4 Example Programs and Results

The **example program** in Section 9 of each function document illustrates a simple call of the function. The programs are designed so that they can fairly easily be 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. They may also be obtained directly from the NAG Web site.

Note that the results from running the example programs may not be identical in all implementations, and may not agree exactly with the results in the Manual and which were obtained from a double precision implementation (with approximately 16 digits of precision).

The Users' Note for your implementation will mention any special changes which need to be made to the example programs, and any significant differences in the results.

3.5 Summary for New Users

If you are unfamiliar with this library and are thinking of using a function from it, please follow these instructions:

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

(b) consult the Library Contents list to select an appropriate chapter or function;

(c) or search through the **Keyword Index** or **GAMS Classification Index**;

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

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

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

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

(h) obtain an online copy of the example program 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 the Chapter Introduction;

read the function document;

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

4 Support from NAG

NAG Response Centres

The NAG Response Centres are available for general enquiries from all users and also for technical queries from users with support.

The Response Centres are open during office hours, but contact is possible by fax, email and telephone (answering machine) at all times. Please see the Users' Note or the NAG web sites for contact details.

When contacting one of the NAG Response Centres, it helps to deal with you query quickly if you can quote your NAG user reference 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, down-loadable software, case studies, industry articles and technical reports. The NAG web site can be accessed via:

```
http://www.nag.co.uk
http://www.nag.com
http://www.nag-j.co.jp
```

5 Background to NAG

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

6 References

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

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

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

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