NAG C Library

Essential Introduction

This document is essential reading for any prospective user of the Library

Contents

1 The NAG C Library and its Documentation

1.1 Structure of the Library

The NAG C Library is a collection of C functions for the solution of numerical and statistical problems. It is divided into chapters, each containing a set of functions devoted to a particular branch of numerical or statistical analysis. Generally each chapter has a three-character name and a title, for example

c06 – Fourier Transforms

but exceptionally Chapters f and s have one-character names. The chapters and their names are based on the ACM modified SHARE classification index (see Section 7).

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 f06) 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 3.3 for further details.

1.2 Structure of the Manual

The **NAG C Library Manual** is the principal documentation for the NAG C Library and 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 and they appear in alphanumeric order. Library functions are sometimes termed routines in the text, particularly where confusion with the mathematical term function could arise.

Most chapters contain a Chapter Introduction and a set of documents, one for each user-callable function in the chapter. A function document has the same long name and short name as the function which it describes. The documents are ordered alphanumerically using the short function name. The manual is available in both printed and online form.

1.3 Marks of the Library

New Marks of the NAG C Library will be released as new functions are added to the Library and improvements or corrections are made to existing functions. At each Mark the documentation of the Library will be updated; it is important that the Mark of the documentation used is consistent with the Mark of the Library software you are using.

The Library may be updated between Marks to an intermediate maintenance level so that corrections can be incorporated. Maintenance levels are indicated by a letter following the Mark number, e.g., 6A, 6B, and so on.

1.4 Implementations of the Library

The NAG C Library is available for use on different computer systems. For each distinct system, an **implementation** of the Library is prepared by NAG. An implementation is usually specific to a range of machines (e.g., Sun SPARCstations); it may also be specific to a particular operating system, C compiler, or compiler option. The implementation distributed to sites includes a tested compiled library.

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 provided in a separate **Users' Note** for each implementation.

1.5 Library Identification

The Mark and implementation of the NAG C Library you are using may be found by calling the function nag_implementation_details (a00aac) from a C program.

1.6 C Language Standards

All functions in the NAG C Library conform fully to ANSI C, and functions introduced at Mark 6 take advantage of the `const` keyword to allow for safer code which can be more easily optimised. Although existing 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 Documentation

The Manual is designed to serve the following purposes:

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

(ii) to advise on the choice of the most suitable function or functions to solve a particular problem;

(iii) to advise on the choice of the most suitable function or functions to solve a particular problem;

(iv) to give all the information needed to call a function correctly from a C program and to assess the results.

The Chapter Introductions contain general advice on a suitable choice of function. When you have chosen a function, you must consult the relevant document. Each function document is essentially self-contained (it may contain references to related documents). It includes a description of the method, detailed specifications of each parameter, explanations of each error exit, remarks on accuracy, and an example program to illustrate the use of the function.

2.1 Structure of Function Documents

At Mark 6, a new typesetting system was used to generate the documentation. New and revised function documents differ in appearance from previous marks, although their structure remains the same. Additional information is now provided in the Parameter section.

Each function document is subdivided into the following sections.

Purpose

A concise description of the purpose of the function.

Specification

An ANSI C declaration of the function together with a list of the include files required to enable use of the function.

Description

A description of the operation(s) carried out by the function, together with background information.

Parameters

Each parameter is listed. The parameter is now numbered, its type is given explicitly and, if relevant, the minimum array size. The input/output properties of the parameter are given together with its purpose. Any constraints on the parameter value are also stated. If a function is called with an invalid value of a parameter, the function will usually take an error exit, producing an error message on the C standard error stream `stderr` if error message printing is requested (see Section 3.6).

Error Indicators and Warnings

This section lists all possible error exits with the appropriate NAG error code. Where necessary, further advice is given to help the user correct the error.

Further Comments

Notes on the accuracy of the function, timing information and other comments.

References

List of references which give background information relevant to the function.

See Also

Cross references to other similar or complementary NAG C Library functions.

Example

Each function has a simple example program, if appropriate, showing its use; input data (if required) and results are also listed. These programs and their data should be available online at your installation, and users are encouraged to use them as a basis for their own calling programs. Please note that the example results from different implementations of the Library may in some cases differ slightly from each other and from those printed in the Manual.

In Chapters e04 and g13 the more complex functions have function documents with additional sections; these describe the optional function arguments and printing facilities and may also give additional information on the operation of the algorithm. These documents have two example programs, the first of which appears after the section describing the mandatory parameters, and the second at the end of the document.

Additionally some of the functions in Chapters g02 and g13 which do not take optional arguments also have two example programs, illustrating the more complex use of these functions.

3 Using the Library

3.1 General Advice

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

(a) formulating the problem and choosing appropriate NAG C function(s),

(b) producing a C program to call the required NAG C function(s) correctly,

(c) assessing the significance of the results.

Advice on points (a) and (c) is given in the Chapter Introductions and function documents; point (b) is discussed below.

3.2 Programming Advice

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 the user has 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 2.1 above) and one of these example calling programs is repeated below.

Example program to call NAG C Library function nag_real_eigensystem (f02agc).

```
/* nag_real_eigensystem(f02agc) Example Program
 *
 * Copyright 1989 Numerical Algorithms Group.
 *
 * Mark 1, 1990
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagf02.h>

#define NMAX 4
#define TDA NMAX
```

```
#define TDV NMAX
#define COMPLEX(A) A.re, A.im

main()
{
  Integer  i, j, n;
  double a[NMAX][TDA];
  Complex r[NMAX], v[NMAX][TDV];
  Integer iter[NMAX];

  Vprintf("f02agc Example Program Results\n");
  Vscanf("%*[^\n]");  /* Skip heading in data file */
  Vscanf("%ld", &n);

  if (n<1 || n>NMAX)
    {
      Vfprintf(stderr, "n is out of range: n = %5ld\n", n);
      exit(EXIT_FAILURE);
    }
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      Vscanf("%lf", &a[i][j]);
  f02agc(n, (double *)a, (Integer)TDA, r, (Complex *)v,
         (Integer)TDV, iter, NAGERR_DEFAULT);
  Vprintf("Eigenvalues\n");
  for (i=0; i<n; i++)
    Vprintf("(%7.3f, %7.3f) \n", COMPLEX(r[i]));
  Vprintf("\nEigenvectors\n");
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      Vprintf("(%7.3f, %7.3f) %s", COMPLEX(v[i][j]),
              (j%4==3 || j==n-1)? "\n" : " ");
  exit(EXIT_SUCCESS);
}
```

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; the user should refer to the above example calling program while studying the description of these features. Users are also recommended to pay particular attention to the specification of the function parameters, array sizes and array indices.

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 the Appendix to this Introduction. 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.

The user may also need to include the header file `nag_stdlib.h` in the calling program; see part (a) of the Appendix to this Introduction.

3.2.1.1 NAG data types

Integer

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

Boolean

This data type is used for all parameters that take a true or false value. It is defined to be the shortest practical integer type, usually **char**. If `TRUE` and `FALSE` have not previously been defined, then the following definitions are used:

```
#define TRUE 1
#define FALSE 0
```

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 the user's 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;
   comm.p = (Pointer)&s;  /* assign address of user defined structure to comm.p
*/
  s.xend = 10.0;
  s.k = 4;
  s.h = (s.xend-x) /(double)(s.k+1);
  d02ejc(neq, fcn, NULLFN, &x, y, s.xend, tol, Nag_Relative,
         out, NULLDFN, &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--;
}
```

Enumeration Types

A number of enumerated types are defined in `<nag_types.h>` for use in calls to various C Library functions. Users must use these enumerated types in their calling programs.

Other structure types

A number of structures have been defined to facilitate calls to NAG functions. These are described in the relevant function documents.

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` 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 the example program for nag_1d_quad_gen (d01ajc).

3.2.1.3 Arrays

One-dimensional Arrays

One-dimensional arrays are passed to NAG functions as pointers to the first element of the array, e.g., arrays **r** and **iter** in the example program at the start of Section 3.2. The size of the array must be at least as large as that required by the problem; in the example call to nag_real_eigensystem (f02agc) **n** is the size of the problem, so the array **iter** must have **n** or more elements. Constraints upon array sizes are given in the individual function documents.

Two-dimensional Arrays

A two-dimensional array is passed to a NAG C Library function as a pointer to the start of a contiguous block of storage (i.e., a single vector) which should contain the elements of the array in the usual C order (i.e., in row order). Because the C language ensures that two-dimensional arrays are held in memory with data stored contiguously, such an array can, like one-dimensional arrays, be passed as a pointer to the first element as long as an appropriate cast is applied. For example, a two-dimensional array of type `double`, **d_array**, defined in the calling program must be cast to `(double *)d_array` in the call to the NAG C function. See arrays **a** and **v** in the example program at the start of Section 3.2.

Dynamically allocated 'two-dimensional' arrays should always be allocated as one-dimensional arrays, e.g., a dynamically allocated version of the array **a** in the example program, having dimensions **n** and **tda** determined at run time, should be allocated as a single block of storage of length `n*tda` thus:

```
a = NAG_ALLOC(n*tda, double);
```

To aid in the correct access of a given element of the array in the calling program, the definition of a macro such as

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

is recommended. This allows the use of an expression such as `A(i,j)` to access an element of the notional two-dimensional array **a**.

The example call to nag_real_eigensystem (f02agc), suitably modified to use dynamically allocated arrays, is shown below:

```
/* nag_real_eigensystem(f02agc) Example Program
 *
 * Copyright 1989 Numerical Algorithms Group.
 *
 * Mark 1, 1990.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagf02.h>

#define A(I,J) a[(I)*tda + (J)]
#define V(I,J) v[(I)*tda + (J)].re, v[(I)*tda + (J)].im
#define R(I)   r[(I)].re, r[(I)].im

main()
{
  Integer  i, j, n;
  double *a = 0;
  Complex *r = 0, *v = 0;
  Integer *iter = 0;
  Integer tda, tdv;

  Vprintf("f02agc Example Program Results\n");
  /* Skip heading in data file */
  Vscanf("%*[^\n]");
  Vscanf("%ld", &n);
  tda = n;
  tdv = n;

  a = NAG_ALLOC(n*tda, double);
  r = NAG_ALLOC(n, Complex);
  v = NAG_ALLOC(n*tdv, Complex);
  iter = NAG_ALLOC(n, Integer);

  if ( !a || !r || !v || !iter )
    {
      Vfprintf(stderr, "Memory allocation failed\n");
      exit(EXIT_FAILURE);
    }

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      Vscanf("%lf", &A(i,j));
  nag_real_eigensystem(n, a, tda, r, v, tdv, iter, NAGERR_DEFAULT);
  Vprintf("Eigenvalues\n");
  for (i=0; i<n; i++)
    Vprintf("(%7.3f, %7.3f) \n", R(i));
  Vprintf("\nEigenvectors\n");
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      Vprintf("(%7.3f, %7.3f) %s", V(i,j),
              (j%4==3 || j==n-1)? "\n" : " ");
```

```
  NAG_FREE(a);
  NAG_FREE(r);
  NAG_FREE(v);
  NAG_FREE(iter);
  exit(EXIT_SUCCESS);
}
```

Further parameters to nag_real_eigensystem (f02agc) define the size of the problem (in the example this is **n**) and the size of the second dimension of the array (**tda** and **tdv** in the example). The user must ensure that the second dimensions are at least as large as **n** (the size of the problem). The particular size constraints will be given in the function document. The second dimension information is essential if the function called is to access the correct array elements. The user must also ensure that the first dimension of the array is sufficiently large for the problem (i.e., as large as **n** in this example) even though the first dimension is not passed; normally the problem size in this respect will be represented by the required number of rows in the matrix.

3.2.1.4 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`. Note that there are six separate header files for functions in Chapter f: `nagf01.h`, `nagf02.h`, `nagf03.h`, `nagf04.h`, `nagf06.h` and `nagf11.h`. See also the Appendix to this document.

3.3 Use of NAG Long Names

The long names defined in the header file `nag_names.h` are #defines. Users 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`, users 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, 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 Optimization (e04) and Operations Research (h02) areas read input from external files.

3.5 Auxiliary Functions

In addition to the documented functions available to the user, the NAG C Library contains a much larger number of auxiliary functions. Users do not normally need to concern themselves with these functions, as they will automatically be called as required by the user-callable function selected by the user. The function declarations of these auxiliary functions can be found in the relevant chapter header files together with the user-callable function declarations.

3.6 NAG Error Handling and the fail Parameter

All functions that have error exits have a parameter that allows the user control over the printing of error messages when an error is detected. There is a further option which allows users either to continue running their 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.

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** parameter 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.

3.6.2 Use of the fail parameter

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

```
typedef struct {
  int code;
  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 3.6.3).

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

```
static NagError fail;
```

The address of the parameter is then passed to the NAG C function being called. The use of **static** in the declaration is recommended as all members of the structure must be initialised before passing the parameter to the called function, even though a member may not actually be required by the user. As an alternative to the use of the storage specifier **static** the NAG-defined macro SET_FAIL or INIT_FAIL may be used. SET_FAIL initialises **fail** and sets the **fail.print** member to TRUE.

(a) Use of the fail parameter with the print member set to TRUE

If the user requires 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** parameter must be declared with all members initialised and the **print** member set to TRUE. Use of the NAG-defined macro SET_FAIL with the statement SET_FAIL(fail); performs the appropriate assignments. Alternatively the initialisation could be done by declaring the **fail** parameter with **static** (see the example declaration above) and then setting **fail.print** to 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. The user must ensure that the calling program tests the **code** member of the **fail** parameter on return from the NAG C function; the user 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 routine document for nag_real_svd (f02wec)).

(b) Use of the fail parameter with the print member set to FALSE

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

This method is suitable for users who wish to produce their 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** parameter (see the description of 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** parameter. It should be set to `TRUE` if the NAG error message is to be printed, otherwise `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 `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);
    }
}
```

3.7 Approach to Underflow and Overflow Conditions

The NAG C Library algorithms are generally written to avoid underflow and overflow by checking that the input parameters are in the appropriate range and taking other prudent measures when evaluating certain expressions. However, not all expressions are checked as this would cause excessive run-time overhead.

When underflow occurs, it is generally assumed to have no side effects. Most run-time environments have some means of trapping or warning of floating point underflow. It is not recommended that such facilities be enabled whilst using the NAG C Library.

Overflow can occur particularly when a problem is ill-posed. Such conditions are generally regarded as terminal and are expected to be trapped as such by the run-time environments.

4 Optional Function Parameters

Some of the more sophisticated functions in the NAG C Library use algorithms which require a large number of parameters to be set before use. To avoid having an excessively long argument list and yet allow the user control over these parameters, an 'option setting' facility is provided for these functions.

The least used of the parameters have been combined into a structure; users need only assign values to those members of the structure that govern the algorithm parameter they are interested in. Any members not set by the user will cause the selection of a suitable default value for that parameter.

This mechanism of option setting provides a simple interface to the function, which allows it to be used quickly and easily but still allows control of the more complex features of the algorithm when this is required.

Use of the C mechanism of variable-length argument lists for option setting has been avoided, as variable-length argument lists are inherently unsafe with regard to the identification of optional argument type and do not provide a simple interface. The mechanism provided by the NAG C Library fulfils the criteria of being safe and easy to use.

4.1 Use of Optional Parameters

The general method of 'option setting' is to initialise an *options structure* with a call to a utility function, assign appropriate values to selected structure members and then pass the address of the structure to the required function.

The principal example of the use of the 'option setting' facility within the library is in the optimization functions of Chapter e04. These functions use a structure of type **Nag_E04_Opt** to hold the optional parameters. The structure contains members for the output of results as well as the input of parameter values. A typical example of its use is:

```
/* Initialise the options structure */
e04xxc(&options);

/* Assign selected values */
options.optim_tol = sqrt(X02AJC); /* Adjust final solution accuracy. */
options.linesearch_tol = 0.7;     /* Adjust linesearch accuracy. */
options.print_level = Nag_Soln;   /* e04dgc will print the final result. */

/* Pass the structure address to the optimization function */
e04dgc(n, objfun, x, &objf, g, &options, NAGCOMM_NULL, &fail);
```

For the call to nag_opt_conj_grad (e04dgc) there are more than ten optional input parameters in addition to the ones selected in the example code above. Those members which are not selected will cause suitable default values to be used for the relevant parameter.

Most functions return some results within the options structure; these values supply extra information about the solution which the user may wish to consult but are less important than the values returned via the other function arguments.

4.2 Memory Management

Pointers within the options structure are often used to hold arrays of values, usually for output but occasionally for input as well. Memory is allocated to these pointers by the NAG functions and a memory deallocation function is provided for each options structure.

The optimization chapter permits the allocation of memory to these pointers by the user; however, this facility is rarely needed and in general it is recommended that the user rely solely upon the NAG functions for the management of memory.

An example of using a function where pointers in the options structure are used to output results is:

```
/* Assign data to function parameters. */
n = 20;
  .
  .
  .
for (i = 0; i < n; ++i) Vscanf("%lf", &x[i]);

/* Initialise the options structure */
e04xxc(&options);

/* Assign selected values to options structure. */

/* Adjust number of iterations allowed in feasibility phase. */
options.fmax_iter = 20;

options.ftol = 10*sqrt(X02AJC); /* Adjust feasibility tolerance. */

/* e04nfc will print intermediate and final results in full. */
options.print_level = Nag_Soln_Iter_Full;


/* Pass the structure address to the optimization function */
e04nfc(n, nclin, (double *)a, tda, bl, bu, cvec, (double *)h, tdh,
       NULLFN, x, &objf, &options, NAGCOMM_NULL, &fail);

 /* On return from e04nfc the following results can be obtained via
* the pointers in the options structure.
*
* options.ax      holds the final values of the constraints.
* options.lambda  the Lagrange multipliers.
* options.state   the status of the constraints.
*/

 /* When no longer required free the memory allocated by e04nfc
* and set pointers to NULL.
*/
e04xzc(&options, "all", &fail2);
```

If memory is allocated to pointers within the options structure by the user then the user must also manage the deallocation of this memory and set the pointer to NULL after deallocation if a further call to a NAG function will occur using the same options structure. The NAG memory deallocation functions will not free memory that has not been assigned by a NAG function.

It is sometimes necessary to make a further call to a function after returning from the first call, for example the iteration limit might be reached while the function is still making progress towards the optimum. In this situation some functions have a ‘warm start’ facility that allows the function to make use of the information returned in a previous call, rather than re-initialising all values which is the case with the default ‘cold start’. An example of a repeated call with a ‘warm start’ is:

```
/* Initialise the options structure */
e04xxc(&options);

/* Assign selected values to options structure. */

/* Adjust number of iterations allowed in feasibility phase. */
```

```
options.fmax_iter = 20;

options.ftol = 10*sqrt(X02AJC); /* Adjust feasibility tolerance. */

/* e04nfc will print intermediate and final results in full. */
options.print_level = Nag_Soln_Iter_Full;


/* Pass the structure address to the optimization function */
e04nfc(n, nclin, (double *)a, tda, bl, bu, cvec, (double *)h, tdh,
       NULLFN, x, &objf, &options, NAGCOMM_NULL, &fail);

if (fail.code == NW_TOO_MANY_ITER)
{
    /* Specify a warm start. */
    options.start = Nag_Warm;

    /* Call the optimization function again. */
    e04nfc(n, nclin, (double *)a, tda, bl, bu, cvec, (double *)h, tdh,
           NULLFN, x, &objf, &options, NAGCOMM_NULL, &fail);
}


/* Free the memory allocated by e04nfc and set pointers to NULL. */
e04xzc(&options, "all", &fail2);
```

Note that the options structure used in the first call must be the same as that supplied in the second call. Information for the 'warm start' is communicated between the two calls via members of the options structure; in the example above the pointer **options.state** supplies the status of the constraints at the end of the first call to the second call of nag_opt_qp (e04nfc). The memory deallocation function should not be called between successive calls of the optimization function, even for a 'cold start', as the optimization function will manage any required deallocation and reallocation of memory.

4.3 Reading Option Values from a File

In Chapter e04, option values can be read from a file and assigned to the option structure using the utility function nag_opt_read (e04xyc). This facility allows option values to be changed between different runs of a program without recompilation. The values read from the file are checked before assignment to ensure that they are in the correct range for the option specified; an error message is generated for any value that is out of range and the value is not assigned. An example of its use is:

```
/* Initialise the options structure */
e04xxc(&options);

/* Assign a value to the options structure. */

/* Adjust maximum number of iterations allowed. */
options.max_iter = 100;

/* Read other option values from a file, in this case stdin */
fail.print = TRUE;
print = TRUE;
e04xyc("e04fcc", "stdin", &options, print, "stdout", &fail);

if (fail.code == NE_NOERROR)
  /* Pass the structure address to the optimization function */
  e04fcc(m, n, lsqfun, x, &fsumsq, fvec, (double *)fjac, tdj,
         &options, &comm, &fail);

/* Free the memory allocated by e04fcc and set pointers to NULL. */
```

```
e04xzc(&options, "all", &fail2);
```

An example of an option file which could be supplied to the above program is:

```
Options file for e04fcc

begin e04fcc

  linesearch_tol = 0.8 /* Adjust linesearch tolerance. */

  /* e04fcc will print intermediate and final results in full. */
  print_level = Nag_Soln_Iter_Full

end
```

The option values must be set between the keyword sequences `begin <function name>` and `end`, where function name is the name of the optimization function for which the options are being set, in this case nag_opt_lsq_no_deriv (e04fcc). Within the set of option values C style comments are allowed whereas outside the option set any text is permitted. If the parameter **print** is `TRUE` in the call to nag_opt_read (e04xyc) then messages confirming that an option has been read and assigned are output to a file, in the example above, `stdout`.

4.4 Results Printout

Some NAG C Library functions print the results of the function call to a file, `stdout` unless specified otherwise by the user. This printout can be controlled by function parameters. In the case of the file reading function nag_opt_read (e04xyc), printout is switched on or off by means of the Boolean argument **print**; however, in Minimizing or Maximizing a Function (Chapter e04) and Time Series Analysis (Chapter g13) functions, the options structure member **options.print_level** is used to control the level of printout. The default value of **options.print_level** in Chapter e04 is the enum value **Nag_Soln_Iter**, which specifies that a single line of intermediate results be output after each iteration and the details of the final result be output prior to return. The available range of other enum values depends upon the function being used, but they usually allow more detailed results to be printed out as well as less detailed results (including none).

4.4.1 User-defined print function

A print function may also be defined and supplied by the user for all Chapter e04 routines except for nag_opt_nlp_sparse (e04ugc), allowing users to output intermediate results in their own style. The print function defined needs to be assigned to the pointer to function **options.print_fun**. Any function assigned to this member will be called in preference to the NAG default print function. Calls to this user-defined printing function are also controlled by means of **options.print_level**.

5 Summary for New Users

If you are unfamiliar with the NAG C Library and are thinking of using a function from it, please follow these instructions:

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

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

(c) or search through the **Keywords in Context Index**, **GAMS Index** or via an online search facility;

(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 NAG C 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 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.

6 Support from NAG

(a) Contact with NAG

Queries concerning this library should be directed initially to your local Advisory Service. If you have difficulty in making contact locally, you can write to NAG directly. Users subscribing to the Support Service are encouraged to contact one of the NAG Response Centres.

(b) NAG Response Centres

The NAG Response Centres are available for general enquiries from all users and also for technical queries from sites with an annually licensed product or Support Service.

The Response Centres are open during office hours, but contact is possible by fax, email and telephone (answering machine) at all times.

When contacting a Response Centre please quote your NAG user reference and NAG product code.

(c) 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 and technical reports. NAG Web sites can be accessed at

`www.nag.co.uk` or
`www.nag.com` or
`www.naggmbh.de` or
`www.nag-j.co.jp`

7 References

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

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

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

Appendix: NAG Header Files

(a) Header files intended for inclusion by the user 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>`	`<nage01.h>`	`<nage02.h>`
`<nage04.h>`	`<nagf01.h>`	`<nagf02.h>`
`<nagf03.h>`	`<nagf04.h>`	`<nagf06.h>`
`<nagf11.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>`

`<nag_stdlib.h>` defines `EXIT_SUCCESS` and `EXIT_FAILURE` and the memory allocation macro `NAG_ALLOC` and `NAG_FREE`. This header file must be included by the user if the NAG definitions of `EXIT_SUCCESS`, `EXIT_FAILURE`, `NAG_ALLOC` and `NAG_FREE`, as used in the example programs, are required.

(b) The following three header files are included by `nag.h` (the user does 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.