NAG C Library

ライブラリとドキュメントの使い方

目次

1 ライブラリの識別

定期的にライブラリの新たな **Mark**（バージョン）がリリースされます．具体的には，新たな関数の追加，既存の関数の修正，改良，さらに改良版導入に伴う関数の削除といった内容が伴います．ただし，Mark 26.1 のようなポイント・リリースには関数の削除はありません．

ユーザーはライブラリの**どの実装，どの Mark 及びリビジョン**を使用しているかを知っていなくてはなりません．これらの情報を確認するには，ライブラリ関数 nag_implementation_details（a00aac）を呼び出すプログラムを実行します．

この関数は引数を取らず，単に文字列を標準出力に出力します．また，ライブラリの詳細はライブラリ関数 nag_implementation_separated_details（a00adc）でも確認することができます．

2 ライブラリ関数の探し方

NAG ライブラリの利用経験を問わず，以下の利用要領が推奨されます．

(a) この **How to Use the NAG Library and its Documentation**（ライブラリとドキュメントの使い方）を読む．
(b) **Keyword and GAMS Search** を用いて，適切な Chapter（チャプター）や Function（関数）を選択する．
(c) 関連する **Chapter Introduction**（チャプターイントロダクション）を読む．
(d) 関数を選択し **Function Document**（関数ドキュメント）を読む．関数がニーズに合致しなかった場合にはステップ (b) に戻る．
(e) ご利用のライブラリ製品の**ユーザーノート**（Users' Note）を読む．
（ユーザーノートには，ご利用のライブラリのリンク方法が記載されています．）
(f) 利用方法についてのローカルなドキュメント（サイトで用意されているもの等）があれば，それを読む．
(g) 該当関数の Example プログラム（セクション 4.4 参照）を使用してみる．

この段階でユーザープログラムへのライブラリ関数のコーディングは終わり，コンパイル，実行を試みる段階にきているはずです．もちろん問題が発生したり，結果に確信が持てないような場合には，再度関連するドキュメントを参照する必要があります．

ライブラリの利用経験に応じて (a) から (g) までのステップのいくつかはスキップできますが，内容変更の可能性のある以下のドキュメントは常に最新の状態に保つことが望まれます．

- **How to Use the NAG Library and its Documentation**
- **Chapter Introduction**（チャプターイントロダクション）
- **Function Document**（関数ドキュメント）
- 実装固有の**ユーザーノート**（Users' Note）

3 ライブラリの使い方

3.1 ライブラリの構成

NAG C ライブラリは，数値計算や統計解析の分野の問題を解くための様々な **Function**（関数）の集合体です．

ライブラリは **Chapter**（チャプター）に区分されており，その各々は数値計算や統計解析の個別の分野に対応しています．各チャプターには 3 文字からなる名称とタイトルが付けられています．

（例） d01 - Quadrature（数値積分）

チャプター h と s は例外で，1 文字の名称からなります．これらのチャプターの構成と名称は ACM 修正版 SHARE 分類インデックス（ACM (1960–1976) 参照）に基づいています．

ドキュメント化された関数には全て 2 つの名前が付けられています．一つは SHARE インデックス分類に基づくもので，チャプター／サブチャプターの名称を表す文字で始まる 6 文字から構成されます．

（例） c06pcc

このタイプの関数名はすべて小文字であり，2 番目と 3 番目の文字は数字，末尾の文字は c です．この関数名をショートネームと呼びます．これに対し，それぞれの関数はより意味のある長い名称（ロングネームと呼びます）も持っています．

（例） nag_sum_fft_complex_1d

関数を呼び出す場合，ショートネームの代りにロングネームを用いても構いません．詳細はセクション 3.4 をご参照ください．

3.1.1 実験的な関数

ライブラリ関数には，Experimental（実験的）というカテゴリに分類される関数があります．関数が実験的である場合は，該当の関数ドキュメントの一番上にその旨の Note（注意書き）があります．

次の場合に，関数は実験的として分類されます．

(a) 関数の引数および／または機能が，ライブラリの Mark 間で変わる可能性がある場合．
これらの関数は，そのような変更を最小限に抑えるように設計されています．

(b) 関数の複雑さのため，または関数スイートの一部分であるため，包括的なテストが難しい場合．
これらの関数は，通常の関数と同じテストおよびレビュープロセスを経ていますが，使用するときには注意が必要です．

実験的な関数は，後に実験的でなくなった時点で通常の関数と同様に引数の仕様が固定します．

3.2 一般的なアドバイス

ライブラリ関数は与えられたデータとは無関係に常に有意義な結果を返すことを**保証するものではありません**．次の点に関する**注意**や**配慮**が必要です．

(a) 問題の定式化
(b) ライブラリ関数を使用したプログラミング
(c) 結果の評価

(b) と (c) については本ドキュメントの残りの部分で説明します.

3.3 プログラミングに関するアドバイス

ライブラリとそのドキュメントはユーザーが関数を呼び出すプログラムを C 言語で書くことを前提に作成されています.（その他の言語からの利用に関してはセクション 3.8 をご参照ください.）

適切なライブラリ関数が選択できたら（セクション 4.1 参照），その関数はユーザーによって書かれたプログラム（呼び出し元プログラム）を介してライブラリから呼び出されることになります．本ドキュメントでは，ユーザーがそのようなプログラムを書く上で十分な C 言語の知識をお持ちであることを仮定しています．個々のライブラリ関数のドキュメントには，呼び出し元プログラムの Example が含まれています（セクション 4.4 参照）.

呼び出し元プログラムの開発においては，呼ばれる側であるライブラリ関数に特有の諸事項に加え，（ライブラリ関数を利用する全てのプログラムに共通の）環境に関する数多くの事項を守る必要があります．これらの事項については以下で説明します．また，Example プログラムを参照するようにしてください.

3.3.1 NAG C 環境

ライブラリの利用環境は多くのインクルードファイルで定義されます．一覧についてはセクション 3.3.1.6 に掲載されています．ヘッダーファイルの中で最も重要なものが `nag.h` で，それはライブラリ関数を利用するすべてのプログラムにインクルードされなくてはなりません．また，`nag.h` は他のすべての NAG のヘッダーファイルに先行する必要があります.

これらのインクルードファイルはインストーラーによって <product_folder>/include フォルダーに置かれます．実際の格納場所は**ユーザーノート（Users' Note）**や他のローカルなドキュメントをご参照ください.

ファイル `nag.h` は，ライブラリで使用されるデータ型とエラーコードを規定すると共に，Example プログラムで用いられる多くのマクロも定義しています.

メモリ管理が必要な場合には，呼び出し元プログラムにはヘッダーファイル `nag_stdlib.h` もインクルードする必要があります．セクション 3.3.1.2 をご参照ください.

3.3.1.1 NAG データ型

Integer

このデータ型はライブラリ関数のほとんど全ての整数型引数で使用されています．これは通常 `long` として定義されます．正確な型はユーザーノート（Users' Note）をご参照ください.

Nag_Boolean

これは次のように定義される列挙型のデータです.

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

Complex

このデータ型は複素数を使用するために定義された構造体です．

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

Pointer

このデータ型は一般的なポインタ `void *` を表します．

Nag_Comm

このデータ型は呼び出し元プログラムとユーザー定義関数（ライブラリ関数の引数としてユーザーが定義する関数）との間の交信に使用される多くのフィールド（汎用ポインタ，Integer 型ポインタ，double 型ポインタなど）を持つ構造体です．このデータ型を用いることによって，そのような交信に対してグローバル変数を使用する必要がなくなります．また，Integer 型および double 型のポインタを用いることによって，Integer 型および double 型の配列の交信をキャストなしで行うことができます．具体的な用法については，nag_opt_simplex_easy（e04cbc）の関数ドキュメントのセクション 10 をご参照ください．

Nag_User

これも Nag_Comm と同様の用法のポインタを持つ交信用構造体です．

Nag_FileID

このデータ型は入出力動作を伴うある種のライブラリ関数によって用いられる整数型データです．ファイル名をこのファイル ID に対応付けるにはチャプター x04 の関数を使用する必要があります．

列挙型

種々のライブラリ関数を呼び出す際に使用される多くの列挙型データが `nag_types.h`（`nag.h` にインクルードされている）に定義されています．これらは C/C++ 呼び出し元プログラムで使用する必要があります．C/C++ 以外の言語からライブラリ関数が呼ばれる場合に対して，列挙型のメンバを整数値にマッピングする関数が提供されています（nag_enum_name_to_value（x04nac）参照）．

その他の構造体

ライブラリ関数の呼び出しをサポートするために多数の構造体が定義されています．ご利用のライブラリ関数の呼び出しに関連した構造体の要素は対応する関数ドキュメントに記述されています．NAG によって定義された構造体の完全な仕様はヘッダーファイル `nag_types.h` に含まれています．

3.3.1.2 メモリ管理

メモリはライブラリ関数の内部でしばしば動的にアロケートされます．その際，すべてのメモリアロケートは成功だったか失敗だったかがチェックされます．万一失敗した場合には，ライブラリ関数はエラーメッセージ **NE_ALLOC_FAIL** と共にリターンもしくは終了します（ライブラリのエラー処理の詳細についてはセクション 3.7 をご参照ください）．

マクロ NAG_ALLOC, NAG_REALLOC, NAG_FREE は，ライブラリに対して適切なメモリ管理関数を選択できるように用意されています．NAG_ALLOC は 2 つの引数を取ります．1 番目の引数はアロケートされるべき要素の数を，2 番目の引数は要素の型を指定します．ステートメント

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

は double 型の要素 n 個分のメモリを double 型のポインタ p にアロケートします．

NAG_REALLOC は 3 つの引数を取ります．1 番目の引数はメモリが拡張されるべきポインタの名前を，2 番目の引数は要素の数を，3 番目の引数は要素の型を指定します．ステートメント

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

は double 型の要素 n 個分のメモリを double 型のポインタ p にアロケートします．

NAG_FREE は NAG_ALLOC または NAG_REALLOC によってアロケートされたメモリを解放します．唯一の引数は解放すべきメモリへのポインタです．ステートメント

```
NAG_FREE(p);
```

は p によってポイントされているメモリを解放し，p に **NULL** を設定します．

これらのマクロはヘッダーファイル nag_stdlib.h に定義されています．呼び出し元プログラムでこれらのマクロを使用する場合には，このヘッダーファイルをインクルードする必要があります．ライブラリ関数によってアロケートされたメモリを解放するには，NAG_FREE を使用してください．NAG_ALLOC を使ってアロケートされたメモリは，必ず NAG_FREE によって解放されなくてはなりません．具体的な用法については，nag_1d_cheb_fit_constr（e02agc）の関数ドキュメントのセクション 10.1 をご参照ください．NAG_ALLOC, NAG_REALLOC, NAG_FREE の使用は強く推奨されます．

3.3.1.3 Nag_Order 引数

2 次元データのメモリ上での配置はプログラミング言語によって異なったものとなります．C 言語の場合，2 次元配列は行ごとに並べられた 1 塊のメモリとして扱われます（行優先順（‘row-major’ ordering））．これに対し，Fortran を含む他の言語の場合には，2 次元配列は列ごとに並べられます（列優先順（‘column-major’ ordering））．ライブラリ関数で 2 次元配列を扱うものは，適切な場合に，order と称する追加の引数をサポートしています．これはデータが行ごとに並べられているか（order を Nag_RowMajor に設定），列ごとに並べられているか（order を Nag_ColMajor に設定）を指定できるようにするものです．この機能は，列優先順に基づく言語（Visual Basic 7 以降など）からライブラリを利用する場合に有用です．

3.3.1.4 配列参照

C においては，次の様式の表記により 2 次元変数の宣言が可能です．

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

この変数が関数の引数だった場合，それはコンパイラにより実質的に，スタック上に dim1*dim2 のメモリをアロケートされた double 型のポインタ *a として扱われます．この配列の要素，例えば，a[3][5] のアドレスは C がデータを行優先順に格納するため *(a+3*dim2+5) と計算されます．

別の策として，型が double * のポインタに対して（ヒープ上の）メモリを明示的にアロケートすることもできます．

```
a=(double *)malloc((size_t)(dim1*dim2*sizeof(double));                    [1]
```

この場合，次のようなマクロ定義を使うと，このアドレス計算を簡単な表現で行わすことができます．

```
#define A(I,J) a[I*dim2+J]                                                [2]
```

この配列の ij 要素はポインタ表記では `*(a+i*dim2+j)` のように，配列表記では `a[i*dim2+j]` のように，またマクロ [2] が定義されている場合には `A(I,J)` のようにインデックスされます．

しばしば，行列 A の部分行列（例えば，行が k から l で列が m から n の部分行列）を参照したい場合があります．便宜上，$A_{ij} = A(i-1, j-1)\,, i = k, \ldots, l\,, j = m, \ldots, n$（`[2]` と `[4]` 参照）で与えられる部分行列を配列 `a` に対して `A(k:l,m:n)` と表記します．すなわち，行優先順に対しては

```
A(k:l,m:n) = {a[p], p = (i-1)*pda+j-1, i=k,...,l and j=m,...n}            [3]
```

となります．

配列変数を

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

のように定義し，かつデータが列優先順に格納されるのであれば，ij でインデックスされる要素は実質的に転置されたことになります．すなわち，行優先順の場合の要素 `a[i][j]` は，列優先順の場合の要素 `a[j][i]` に対応します．

また，上記 `[1]` のように malloc でメモリを確保する場合には，ij でインデックスされる要素はポインタ表記では `*(a+j*dim1+i)` のように，配列表記では `a[j*dim1+i]` のように，またマクロが

```
#define A(I,J) A[J*dim1+I]                                                [4]
```

のように定義されている場合には `A(I,J)` のようにインデックスできます．`[2]` と `[4]` の定義の違いに注意してください．

ドキュメントの簡素化のため，配列要素の参照には上記のマクロ定義を用いることにします．また，プリプロセッサに対する指令 [2], [4] において決め手となる次元は，行優先順の場合は `dim2`，列優先順の場合は `dim1` である点に注意してください．`dim2`，あるいは `dim1` をメモリ上での並べ方に応じて主次元（principal dimension）と呼ぶことにします．主次元は order 引数に応じて行，あるいは列を横断する際の歩幅と考えることができます．

配列の格納形式を選択できる（**order** 引数を持つ）関数は '**pda**' 引数を持ちます．配列の格納形式が行優先順か列優先順か（**order** の設定）によって，**pda** は異なる値を持ちます．配列の格納形式が行優先順の関数は '**tda**'（trailing dimension を意味する）引数を持ちます．配列の格納形式が列優先順の関数は '**lda**'（leading dimension を意味する）引数を持ちます．Mark 24 から，ライブラリの新しい関数は 2 次元データを列優先順で格納します．また，'stride separating row elements' というフレーズは列優先順の格納を意味し，'stride separating column elements' というフレーズは行優先順の格納を意味します．

関数を呼び出す際に相互に排他的な格納方式でデータを渡す必要がある場合，例えば，関数 A には行優先順でデータを渡す必要があり一方で関数 B には列優先でデータを渡す必要がある場合（しかも，どちらの関数も 'order' 引数を持たない場合），チャプター f16 で提供される関数が役に立ちます．例えば，nag_dge_copy（f16qfc）を用いれば，行列の転置をコピーすることによって行優先順を列優先順に変えることができます．その他，チャプター f16 の関数を用いて様々なコピー処理（行列の三角部分をコピーする等）を行うことができます．

以上の概念を 2 つの例を用いて説明します．一つ目はメモリをアロケートする例，二つ目はメモリを宣言する例です．いずれの場合も行優先か列優先かはプリプロセッサマクロ NAG_ROW_MAJOR を使って明確に宣言されています．メモリのアロケートには明示的に malloc を呼び出す代わりに NAG_ALLOC（nag_stdlib.h で定義されているマクロ）が使われています．このマクロはメモリアロケート用のライブラリ内部関数に対応付けられています．また，メモリの解放には NAG_FREE が使われていることに注意してください．

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;

  /* Initialize 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;
}
```

3.3.1.5 内部データ構造

効率面の理由から，ライブラリは可能な限り Basic Linear Algebra Subprograms（BLAS）を利用する形で設計されています．BLAS は多くのコンピューターメーカによって特定のハードウェア用にチューンされた基本的な関数群です．BLAS は Fortran の仕様に則っており列優先順の格納様式を使用しているため，ライブラリでも新たな関数についてはなるべくこの様式を内部的に使うようにしています．従って，ユーザーが用意した 2 次元配列はライブラリ関数の内部で（入力または出力の際に）適宜に並び変えられる場合があります．故に，列優先順の格納様式を使用した方がわずかですが効率的には有利です．しかし，非常に大きなデータセットの場合を除けばその差は無視できるレベルです．

3.3.1.6 チャプターヘッダーファイル

チャプターヘッダーファイルにはライブラリ関数に対する ANSI 関数プロトタイプ付きの関数の宣言が含まれています．ユーザープログラムから呼び出されるそれぞれのライブラリ関数ごとに適切なチャプターヘッダーファイルがインクルードされなくてはなりません．例えば，関数 nag_sum_fft_complex_1d（c06pcc）を呼び出すには次のようにしてチャプターヘッダーファイル `nagc06.h` をインクルードする必要があります．

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

ネーミングに関しては，`nag` の後に関数名の先頭 3 文字（小文字）を続け，ファイル拡張子として `.h` を付け

るという慣行が用いられています．ただし，チャプター s の関数についてはヘッダーファイル nags.h が使用されます．

(a) インクルード対象のヘッダーファイル

nag.h はライブラリを使用する際の基本環境を定義します．このヘッダーファイルはライブラリを利用する個々のプログラムに対しインクルードされなくてはならず，またインクルードされる他のすべてのヘッダーファイルに先行する必要があります．これに続けて次のチャプターヘッダーファイルを一つ又は複数インクルードします．

```
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  nagx06.h  nagx07.h
```

nag_stdlib.h はメモリアロケーションマクロ NAG_ALLOC と NAG_FREE を定義します．上記用例で示されているように，NAG_ALLOC と NAG_FREE の定義が必要な場合には，このヘッダーファイルをインクルードする必要があります．

(b) nag.h によってインクルードされるヘッダーファイル

次の 3 つのヘッダーファイルは nag.h をインクルードすると自動的にインクルードされるので，ユーザー側で特にインクルードする必要はありません．

nag_types.h は，ライブラリで使用されるデータ型を定義します．

nag_errlist.h は，ライブラリで使用されるエラーコードとメッセージを定義します．

nag_names.h は，ライブラリ関数のロングネームをショートネームにマッピングします．

3.3.2 ダイレクト／リバースコミュニケーション関数

ユーザー提供の関数を必要とするライブラリ関数は，ダイレクトコミュニケーションとリバースコミュニケーションのどちらかに分類されます．

ダイレクトコミュニケーション関数では，ユーザー提供の関数を実引数として渡します．ユーザー提供の関数は，該当の関数ドキュメントに記載される仕様に従って書かなくてはなりません．大抵の場合，ダイレクトコミュニケーションの方が簡単かつ便利です．しかし時に，この用法がネックになる場合があります．

(i) 決められた関数の仕様では関数と呼び出し元プログラムの間で必要な情報をやり取りできない場合
(ii) 他のコンピューター言語からダイレクトコミュニケーション関数を呼び出す際に，そのコンピューター言語がライブラリと完全に互換性のある形で関数引数をサポートしていない場合

これらの制限はリバースコミュニケーション関数を用いることで取り除かれます．リバースコミュニケーション関数は，1 回の呼び出しで解を得るのではなく，解法プロセスを 1 ステップだけ実行し適切なフラグ（irevcm）をセットして呼び出し元プログラムに戻ります．プロセスが終了したかどうか，もしくは新しい情報が必要かどうかは irevcm の値で判断します．新しい情報が必要な場合，リバースコミュニケーション関数を再度呼び出す前に必要な情報を計算しなければなりません．要するに，解法プロセスの反復ループをユーザー側で行うことになります．一般的に，リバースコミュニケーション関数はダイレクトコミュニケーション関数よりも使用が複雑ですが，関数の評価に対してより大きな柔軟性を持ちます．

3.4 ロングネームの使用

ライブラリ関数のロングネームは，ヘッダーファイル `nag_names.h` に `#define` を用いて定義されています（ショートネームと対応付けられています）．ヘッダーファイル `nag_names.h` は `nag.h` を介してインクルードされるため，呼び出し元プログラム側でインクルードする必要はありません．

3.5 入出力

ライブラリ関数は全てのエラーメッセージと警告メッセージを C 標準エラーストリーム `stderr` に出力します．チャプター e04, e05, g02, g13 のライブラリ関数は，結果を C 標準出力ストリーム `stdout`，またはユーザーが指定したファイルに出力させることもできます．関数の最小化／最大化（チャプター e04）やオペレーションズリサーチ（チャプター h）の多くのライブラリ関数は外部ファイルからの入力を伴います．

3.6 補助関数

ドキュメント化された関数の他に，ライブラリにはより多くの補助関数が含まれています．これらはユーザーが利用するライブラリ関数から必要に応じて自動的に呼び出されるため，通常は気にかける必要はありません．

3.7 エラー処理と fail 引数

エラー出力を持つ全てのライブラリ関数は，エラーメッセージの出力を制御するための引数を用意しています．また，エラーが検出された際にライブラリ関数からリターンして呼び出し元のプログラムを続行させるか，または（ライブラリ関数内での `exit` あるいは `abort` により）プログラムを強制終了するかを選択するためのオプションも用意されています．これらエラー処理機能の使用法については以下で説明します．

実装によっては，ライブラリは LAPACK 関数を含むベンダーライブラリとリンクされており，チャプター f07, f08 関数が対応するベンダー LAPACK 関数に対するラッパーとして機能することがあります．この場合，チャプター f07, f08 関数を介して渡される **fail** 引数は，エラーメッセージの出力に関して完全な形でのコントロールを提供できません．また，エラーが検出された時に制御を呼び出し元プログラムに戻すかどうかを選択することもできなくなります．

3.7.1 NAGERR_DEFAULT の使用

エラー処理機能の利用で最も簡単な方法は，ライブラリ関数の **fail** 引数のところに NAGERR_DEFAULT と書く方法です．エラーが検出された場合には適切なエラーメッセージが stderr に出力され，プログラムは exit により強制終了します．

3.7.2 fail 引数の使用

エラー処理機能の使用法についての残り 2 つの説明は，呼び出し元プログラムでの **fail** 引数の定義を伴います．**fail** 引数は，nag_types.h に定義されている NagError 型の構造体であり，次のようなメンバを持ちます．

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

この中で記号 NAG_ERROR_BUF_LEN は通常 512 と定義されます．

この構造体にはライブラリ関数から返されるエラーコードとエラーメッセージが収納されます．エラーコードとエラーメッセージは nag_errlist.h に定義されています．この構造体の個々のメンバについては以下で説明します（セクション 3.7.3 参照）．

エラー引数 **fail** の呼び出し元プログラムでの宣言は次のように行います．

```
NagError fail;
```

引数のアドレスがライブラリ関数に渡されます．構造体のメンバはライブラリ関数に引数が渡される前にすべて初期化されていなくてはなりません．この初期化には NAG によって定義されたマクロ INIT_FAIL の使用を推奨します．

INIT_FAIL マクロは **fail** メンバに以下の値を設定します．

- **fail.code** = NE_NOERROR
- **fail.print** = Nag_FALSE
- **fail.errnum** = 0
- **fail.handler** = 0

SET_FAIL マクロも利用できます．このマクロは **fail.print** を Nag_TRUE に設定する以外は INIT_FAIL と同じ仕様で **fail** メンバの設定を行います．

(a) print メンバに Nag_TRUE を設定した形での fail 引数の使用

エラーが検出された時にエラーメッセージは出力するが制御は呼び出し元プログラムに戻ってきて欲しい場合，**fail** 引数を全メンバの初期化と共に宣言した上で **print** メンバを Nag_TRUE に設定する必要があります．NAG 定義のマクロ SET_FAIL を使って（SET_FAIL(fail); というステートメントで）同じことが行えま

す．**fail** 引数を **static** を付けて宣言する形で初期化を行い，次に **fail.print** を Nag_TRUE に設定するというアプローチも取れます．

エラーが起こらなかった場合，呼ばれた関数からのリターン時 **fail.code** にはエラーコード NE_NOERROR が設定されます．しかし，エラーが検出された場合，制御が呼び出し元プログラムに戻される前に適切なエラーメッセージが `stderr` に出力されます．また，**fail.code** には該当するエラーコードが設定されます．ライブラリ関数からのリターン時には **fail** 引数の **code** メンバを必ずチェックするようにしてください．その上で呼び出し元プログラムを終了するか継続するかを選択すべきです．例として nag_real_nonlin_eqns_easy（c05qbc）の Example プログラムをご参照ください．エラーが生じたにしても得られた結果になにがしかの意味があるなら処理を継続する方が得策です．nag_real_nonlin_eqns_easy（c05qbc）の場合，エラーコードが **NE_TOO_MANY_FEVALS**，**NE_TOO_SMALL**，**NE_NO_IMPROVEMENT** の時には処理を継続する方が良いでしょう．なぜならば，有用な部分結果を得ることができるからです（nag_real_nonlin_eqns_easy（c05qbc）の関数ドキュメントをご参照ください）．

(b) print メンバに Nag_FALSE を設定した形での fail 引数の使用

エラーが検出された時にエラーメッセージが自動的に出力されるのを望まない場合，**fail** 引数を全メンバの初期化と共に宣言した上で **print** メンバを Nag_FALSE に設定する必要があります．**fail** の宣言において **static** を使用した場合には **print** メンバに Nag_FALSE が設定されます．`INIT_FAIL(fail)` を使用した場合も同様です．

この方法は独自のエラーメッセージを出力したい場合に有効です．ライブラリのエラーメッセージに代わる独自のエラーメッセージは，呼び出し元プログラムに直接コードとして埋め込んでおく方法の他に，**fail** 引数の **handler** メンバに代入されたユーザー作成のエラー処理関数を介して出力させる方法もあります（`handler` メンバについては下記をご参照ください）．

3.7.3 `NagError` 構造体

`NagError` 構造体のメンバについて説明します．

`code`

正常終了の場合，`code` にはエラーコード NE_NOERROR が設定されます．エラー，または警告条件が検出された場合，`code` には該当するエラーコード，または警告コードが設定されます．エラーコードの先頭には NE_ が，警告コードの先頭には NW_ が付けられています．

`print`

`print` は **fail** 引数を用いてライブラリ関数を呼び出す場合には事前に値が設定されていなくてはなりません．エラーメッセージを出力する場合には Nag_TRUE を，そうでない場合には Nag_FALSE を設定します．設定された値はライブラリ関数によって変更されることはありません．

`message`

正常終了の場合，配列 `message` には文字列 `"NE_NOERROR:\n    No error"` が設定されます．エラーが検出された場合，それが出力されるかどうかに関係なく，`message` にはエラーメッセージの文字列が設定されます．

errnum

正常終了の場合，errnum は変更されません．ある種のエラーや警告条件が存在する場合，errnum にはエラーに関する追加情報を示す値が設定されます．例えば，ベクトルに不正があった場合，errnum はベクトルのどの成分が不正であったかを示すことになります．errnum によってどのような情報が返されるかは該当する関数ドキュメントをご参照ください．

handler

エラーが検出された後に呼び出し元の関数に制御を戻したい場合は，handler の値を 0 に設定してください．そうでない場合はユーザーが用意したエラー処理関数をポイントしてください．エラー処理関数の ANSI C による宣言文の例を次に示します．

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

string にはエラーメッセージが，code にはエラーコードが，name にはエラーが検出されたライブラリ関数のショートネームが設定されます．print が Nag_TRUE の場合，エラーメッセージはユーザーが用意したエラー処理関数の呼び出しに先立ち出力されます．ユーザーのエラー処理関数からライブラリに制御を戻した場合，ライブラリのエラー処理関数は呼び出し元プログラムに制御を戻します．または，ユーザーのエラー処理関数で exit を行うこともできます．

デフォルトの stderr にではなく stdout にエラーメッセージを出力する場合の用例を次に示します．

```
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);
    }
}
```

3.7.4 NAG エラーメッセージの構造

2 つ例を上げて，ライブラリのエラーメッセージのフォーマットを考察します．
関数ドキュメントの説明：

NE_INT
On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} > 1$.

NE_BAD_PARAM
On entry, argument $\langle value \rangle$ had an illegal value.

これらに関して NAG 関数がエラーを検出した場合，エラーメッセージを次のようなフォーマットで表示します．（エラーメッセージが表示されるのは，デフォルトのエラーハンドラ NAGERR_DEFAULT が使われているか， もしくは **fail.print** = Nag_TRUE が設定されている場合です．）
エラーメッセージの表示：

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

または，

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

つまり，関数ドキュメントのエラーメッセージの説明に出てくる表記 ⟨*value*⟩ はプレースホルダーです．エラーメッセージが実際に表示される時に，具体的な情報（変数値や引数名など）に置き換わります．

3.7.5 ライセンス管理

ご使用の実装がライセンス管理されている場合，その運用の詳細情報はユーザーのローカルサイトに存在するはずです．サイト管理者にお尋ねください．お使いのマシン上で正規のライセンスが利用できるかどうかを確認するためには，ライブラリ関数 nag_licence_query（a00acc）の Example プログラムを実行してください．

ライブラリからライセンス管理関数を呼び出した際，万一正しいライセンスが見つからなかった場合には，関数はエラー状態 NE_NO_LICENCE を出力してリターンもしくは終了します．その場合，Unix ベースのシステムでは，環境変数 NAG_KUSARI_FILE に適切なライセンスファイルまでのフルパスが正しく指定されているかどうかをチェックしてください．また，ライセンスファイルの内容が適切かどうか（例えば，異なる製品のライセンスを使用してないかどうか等）をチェックしてください．すべての設定が正しいと思われる場合は，日本 NAG にお問い合わせください．

3.7.6 予期しないエラー

万一予期しないエラーが起こった場合でも，関数はエラー状態 NE_INTERNAL_ERROR を出力してリターンもしくは終了します．

3.8 多言語からのライブラリの呼び出し

一般的に，データ型の間に適切なマッピングが存在すれば他言語（C++, C#, Java, Visual Basic, Python など）からも NAG C ライブラリを呼び出すことができます．
NAG C ライブラリ関連情報（https://www.nag.com/numeric/cl/classocinfo.asp）をご参照ください．

3.9 算術の考察と結果の再現性

NAG C ライブラリ関数から得られる結果は，問題の解決に使用されたアルゴリズムだけではなく，ライブラリをビルドする際に使用されたコンピューターやコンパイラの実行時ライブラリ，また実行に使われるマシンの算術特性にも依存します．

歴史的に，異なる種類のコンピューターハードウェアは異なる種類の算術システムを持つ傾向がありました．浮動小数点数の格納に，あるマシンでは基数 16 を使い，あるマシンでは基数 2 を使いました（基数が 8 や 10 といったマシンもありました）．そのような違いはライブラリプロバイダーにとって頭痛の種でした．ある算

術システムで上手く動作したコードが別の算術システムでは同じように動作しないかもしれないからです．ライブラリコードをポータブルにするためには，たくさんの注意を払わなければなりませんでした．

加えて，マシンの算術がフローやエラーを起こすことがあり，掛け算や割り算などの基本的な演算が（特に非常に大きい数や非常に小さい数に対して）時おり間違った結果を与えることがありました．

浮動小数点数の算術に関する最初の IEEE 標準（ANSI/IEEE (1985)）が 1980 年代に導入された後に，この状況は大きく改善されました．現在，主なハードウェア（NAG C ライブラリが動作するほとんどのハードウェア）は IEEE スタイルの基数 2 の算術を使っています．これによりポータブルコードの製造がより簡単になりました．しかし，IEEE 標準には自由裁量の部分があるため，まだ問題は残されています．例えば，算術に 80-bit 内部レジスタ（元々は 1980 年代に Intel 8087 コプロセッサで導入された）を使うハードウェアは，特にコンパイラが算術の部分式を保持する最適化コードを生成する場合には，64-bit レジスタを使うハードウェアとは微妙に異なる動きをします．

コンピューターの算術は（IEEE 標準の算術がそうであるように）一般的に有限精度です．そのため，NAG C ライブラリ関数で実装されている数値計算法のほとんどの解は（単に丸め誤差の蓄積により）厳密解の近似ということになります．

従って，二つの異なるマシンで同じデータを用いて NAG C ライブラリ関数のプログラムを実行すると，コンパイラやハードウェア，また実行時ライブラリなどの違いによって結果が異なります．大抵この違いは小さく，例えば，二つの異なるマシンで良条件の連立一次方程式を解いた場合に，二つの計算結果が最後の数ビットだけ異なるといった具合です．しかし，時には小さな違いが重要視される場合があります．例えば，条件判断がその小さな違いに依存している場合などです．最適化問題の関数は常に同じ局所的最適解に収束するとは限りません（常に同じ局所的最適解が得られる場合は，その旨が関数ドキュメントに注記されています）．また，たとえ同じ局所的最適解に収束するとしても，反復回数は異なるかもしれません．

最新のハードウェアと最適化コンパイラは算術演算に更なる問題を喚起します．その一例が**ストリーミング SIMD 拡張（SSE）**命令の利用にあります．

SSE 命令は浮動小数点数算術演算の低レベルの並列化を可能にします．例えば，128-bit SSE レジスタは二つの 64-bit 倍精度の数値（または四つの 32-bit 単精度の数値）を同時に保持し，同時に演算することができます．大量のデータを扱っている場合，これは大きな時間の節約になります．

しかし，SSE 命令の効率的な使用はメモリ上のデータの並び方に依存します．メモリ間のデータの移動を行う SSE 命令は，データが 16 バイト境界に並んでいることを必要とします．もし NAG 関数が使用しているデータ（例えば，数値の配列へのポインタ）がたまたま上手く整列していない場合，それらの SSE 命令は使用できません．これに対して，最適化コンパイラは二つの命令ストリーム（データが上手く整列している場合とそうでない場合）を上手く生成します．

例えば，二つの n 次元ベクトル x と y の内積を計算する場合を考えてみましょう．ベクトルの内積は，二つのベクトルの相対する成分の積を取り，個々の積を足し合わせることで得られます．良い最適化コンパイラでコンパイルされた関数は，二つ（または四つ）の数値を一度にロードします．そして，二つ（または四つ）の数値同士の積を一度に行って最終結果に加算します．

しかし，データがメモリ上に上手く整列していない状況では（これは良くあることなのですが），一度に一つの数値同士の演算しかできません（従って，最終結果を得るのにより長い時間がかかります）．最適化コンパ

イラはこの状況に対応するコードパスも生成します．そして，実行時にコードは速いパスを取れるかどうかをチェックして適切に動作します．

問題は，加算の順序が変わると最終結果が変わるということです．これはコンピューターの算術が有限精度であるところから来る丸め誤差に由来します．以下の内積の代わりに，

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

次のような内積が行われるかもしれません．

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

どちらの方法でも同じ結果が得られるように思えますが（ただし算術は有限精度であるため，どちらの結果も近似解でしかありません），二つの結果は微妙に異なります．もしこの小さな違いが呼び出し元のプログラムで異なる分岐をもたらすならば，呼び出し元のプログラムの動作に大きな違いを生じさせることになります．

更に，同じマシンで同じデータ用いて同じプログラムを連続で 2 回実行した場合でも，結果は異なる可能性があります．これは，プログラムがロードされた時に，たまたまデータが特定の境界に整列する場合もあれば，そうでない場合もあるからです．

より新しいハードウェアにおいては，AVX 命令が 256-bit と 512-bit のレジスタを使用することにより，一度により多くの数値を演算することができます．AVX 命令に対して，データはメモリ上に 32 バイト幅で並んでいる必要があります．

NAG C ライブラリ関数によって使用されるデータは，NAG C ライブラリの内部でメモリに割当てられます．NAG C ライブラリ関数は上述した計算結果の違いを最小にするために，自前のメモリ割当て関数を使うなどして，データが常に上手く整列するように配慮します．しかし，その配慮は部分的にコンパイラのサポートに依存しているため，常に有効とは限りません．

当然のことながら，関数に渡される前のデータのメモリ割当てをライブラリ関数はコントロールできません．もし容認できないような非決定的な計算結果に遭遇し，それがメモリ上のデータの並び方に依るものではないかと疑われる場合は，データがメモリ上で上手く整列するように配慮することが望まれます．しかし，これをどのように行うかはご利用のシステムに依存した話となります．

マルチスレッド NAG C ライブラリやマルチスレッドベンダーライブラリの並列性は，計算結果の非再現性のまた別の要因となります．関数によっては，異なるコア数で実行した時に（または，同じコア数で同じ計算を繰り返した時でさえ）異なる結果を得ることがあるかもしれません．結果の再現性が重要な場合は，並列化されていないシリアル NAG C ライブラリを利用するのが最良です．

3.9.1 ビット単位の再現性（Bit-wise Reproducibility（BWR））

固定長浮動小数点数（例えば，32-bit 単精度や 64-bit 倍精度など）の数学演算では，結合法則が常に満たされるとは限りません．つまり，コンピューターの計算では $a+(b+c)$ と $(a+b)+c$ の結果が異なる場合があるということです．例えば，IEEE 754 32-bit 浮動小数点数では，$2^{24}+(1-2^{24}) = 1$ となる一方で $(2^{24}+1)-2^{24} = 0$ となります．これは，IEEE 754 32-bit 浮動小数点数の仮数が 23 ビットであるため，$2^{24}+1 = 2^{24}$ となるからです．BWR という用語は，コンピュータープログラム（例えば，一連のソースプログラム）が以下のような異なるコンピューター環境においてビット単位で正確に同じ答えを生成する場合を指します．

1. 異なるオペレーティングシステム（例えば，Windows 対 Linux など）
2. 異なる CPU アーキテクチャ（例えば，Intel 対 AMD または Intel Sandy Bridge 対 Intel Ivy Bridge など）
3. 異なるコンパイラバージョン
4. 異なるスレッド数

ユーザーはしばしば BWR を望みますが，しかし，これを達成することは極めて困難です．一般的に，次の条件を満たさなければなりません．

(a) 命令を常に正確に同じ順番で実行すること．
(b) 他のプロセッサでは利用できないかもしれない高度な CPU 機能（例えば，SSE3，SSE4，AVX）を使わないこと．
(c) スレッド数を常に固定すること．

条件 (a) は，最適化なしで（または，非常に制限した最適化で）コンパイルを行うことを意味します．何故なら，一般的に，コンパイラのバージョンに依って最適化の方法が異なるからです．条件 (b) は，一般的に，最も普及している基本的な SSE 命令だけを利用し，新しいプロセッサの強化された SIMD 命令は利用しない，ということを意味します．

要するに，幅広いコンピューター環境で BWR を達成するためには，パフォーマンスを犠牲にする必要があるということです．

3.9.1.1 ベンダーライブラリと条件付きビット単位の再現性（Conditional BWR（CBWR））

NAG C ライブラリの実装には，ベンダーライブラリ（特に，ベンダーライブラリの線形代数関数）を利用するものがあります．ベンダーライブラリから得られる計算結果については，NAG C ライブラリは BWR を直接制御することができません．

CBWR を導入した一部のベンダーライブラリでは，呼び出し元のコードが一定の条件を満たしていれば，環境変数を設定することによって BWR が有効になります．しかしながら，多くの NAG 関数は，ベンダーライブラリの CBWR が要求する条件を満たしていません．従って，ベンダーライブラリを利用するタイプの NAG C ライブラリ実装では，異なる CPU アーキテクチャに渡って BWR を保証することはできません．

3.10 マルチスレッド

3.10.1 スレッドセーフ

マルチスレッドアプリケーションでは，チーム内の各スレッドは，同じメモリアドレス空間を共有しながら独立に命令を処理します．マルチスレッドアプリケーションが正しく動作するためには，アプリケーションから呼び出される関数がスレッドセーフでなければなりません．つまり，それらの関数に含まれるグローバル変数が異なるスレッドから同時にアクセスされないようになっていなければなりません．これは，OpenMP にみられるような適切な同期によって保証されます．

スレッドセーフと明記されている関数は，複数のスレッドから安全に呼び出すことができます．また，スレッドセーフでない関数でも，その関数自体がマルチスレッド化されている場合があります．セクション 3.10.2 で説明されているように，関数内でスレッドのチームを生成して，ワークロードを共有することができます．

NAG C ライブラリは設計上スレッドセーフです．関数はグローバル変数を使用せず，関数間の交信はすべて引数リストを介して行われるため，プログラム内の複数のスレッドから同時に安全に呼び出すことができます．

3.10.1.1 関数引数を持つ関数

一部のライブラリ関数では，引数にユーザーが関数を提供する必要があります．多くの場合，ユーザー提供関数の引数リストには，グローバル変数を使わずにユーザー提供関数に情報を渡すための配列引数（comm）が含まれています．

引数リストを介して与えられる以上の情報をユーザー提供関数に与える必要がある場合は，目的のライブラリ関数と同等のリバースコミュニケーション関数があるかどうか，該当のチャプターイントロダクションを確認することをお勧めします．リバースコミュニケーションは，ユーザー提供関数の引数リストが十分な柔軟性を持つことができない場合に特化した設計になっており，グローバル変数を介してデータを提供する必要もありません．リバースコミュニケーションが利用できない場合は，通常，ユーザー提供関数と呼び出し元プログラムの両方からアクセス可能なグローバル変数を使用します．ただし，異なるスレッドによるグローバル変数の同時アクセスを回避できる場合にのみ（OpenMP によって threadprivate にされている場合，もしくは適切な同期を使用している場合にのみ），この方法はスレッドセーフです．

ユーザー提供関数のスレッドの安全性は，NAG C ライブラリのマルチスレッド版でもまた問題になります．NAG C ライブラリのマルチスレッド版の多くの関数は，ユーザー提供関数の呼び出しを内部的に並列化します．この問題はグローバル変数だけでなく comm 配列の使用方法にも影響します．このような場合，スレッドの安全性を確保するために同期が必要な場合があります．チャプター x06 には OpenMP 並列領域から呼び出されているかどうかを判断する関数が提供されおり，ユーザー提供関数内で使用することができます．

3.10.1.2 入出力

マルチスレッドアプリケーションで NAG C ライブラリを使用する場合，ライブラリのエラー処理機能を使用する場合には出力を抑止する（**fail.print** = Nag_FALSE と設定する）ことを推奨します．

3.10.1.3 実装依存の問題

非常にまれなケースですが，スレッドの安全性を保証することができない実装もあります．また，実装によっては，例えば，BLAS 関数を高速化するなどの理由により，ベンダーライブラリとリンクされていることがあります．それらのベンダーライブラリがスレッドセーフであることを NAG は保証できません．実装固有の情報については，該当のユーザーノートをご参照ください．

3.10.2 並列性

3.10.2.1 イントロダクション

通常の並列化されていない（シリアル）NAG C ライブラリの計算時間は，使用されるプロセッサの逐次処理性能に大きく左右されます．一方で，並列化された（マルチスレッド）NAG C ライブラリは，計算タスクを複数のコアに分配し並列に処理するため，プロセッサの逐次処理性能を超えて計算を行うことができます．

従来，コンピューターシステムは少数のシングルコアプロセッサで構成されていました．そして，プロセッサの処理性能の向上はクロック周波数の増加によって行われました．しかし，このクロック周波数の増加が限界に達し，処理性能を向上させる別の方法が求められました．そこで登場したのがマルチコアプロセッサです．そして，それは今や至るところで使われています．マルチコアプロセッサは，単一のコアではなく，複数のコア（各コアは基本的には CPU と小さなキャシュから成る）で構成されています．従って，現今のハードウェアリソースを最大限に利用するためには，並列性を活用することが必須となっています．

並列化の有効性は，並列プログラムが同等の逐次プログラムに比べてどの程度速いかで評価できます．これは並列高速化（parallel speedup）と呼ばれます．逐次プログラムが並列化されたときの高速化は，同じ計算問題に対して，逐次プログラムの計算時間を並列プログラムの計算時間で割ることによって定義されます．もし，並列プログラムが n コアを使用した場合に，この高速化の値が n ならば（つまり，並列プログラムの計算時間が元の逐次プログラムの $\frac{1}{n}$ ならば），理想的な高速化が得られたことになります．使用するコア数の増加に対して，並列プログラムの高速化がこの理想的な値に近いならば，その並列プログラムはスケーラビリティ（scalability）が良いと言います．

以下の 2 つの要因のため，並列プログラムのスケーラビリティは理想的な値よりも小さくなります．

(a) 並列化に伴うオーバーヘッド
(b) プログラムの並列化不可能な逐次部分

オーバーヘッドは並列化に必要なセットアップだけでなく通信と同期も含みます．このようなオーバーヘッドは，コンパイラとオペレーティングシステムのライブラリの効率，そして基礎となるハードウェアに依存して異なります．プログラムの並列化不可能な逐次部分の影響は，アムダールの法則（Amdahl's law）によって理論的に（つまり，オーバーヘッドがゼロである理想的なシステムを仮定して）説明されます．アムダールの法則は，逐次部分を持つ並列プログラムの高速化に上限を設けます．r をプログラムの並列部分の割合，$s = 1 - r$ を逐次部分の割合とすると，n コアを使用した場合の高速化 S_n は次の条件を満たします．

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

例えば，4 分の 1 が逐次部分である並列プログラムの高速化は高々 4 となります．なぜなら，$n \to \infty$，$S_n \leq 4$ だからです．

並列化は共有メモリマシンと分散メモリマシンの二種類のシステムで利用でき，それぞれ異なるプログラミング技術を必要とします．分散メモリマシンは，ネットワークで繋がった複数のコンポーネントで構成されており，個々のコンポーネントがプロセッサとメモリ領域を持ちます．これらのコンポーネント間の通信と同期は明示的です．共有メモリマシンは，複数の（もしくは一つの）マルチコアプロセッサを持ち，これらが同じメモリ領域にアクセスします．そして，この共有メモリが通信と同期に使われます．マルチスレッド NAG C ライブラリは OpenMP を用いた共有メモリ並列化を利用します（セクション 3.10.2.2 参照）．

OpenMP を用いた並列プログラムは，実行時に必要に応じて，単一のプロセスから複数のスレッドを生成します（fork）．（なお，共有メモリ並列化を利用するプログラムのことをマルチスレッドプログラムと呼びます．）スレッドは一つのマスタースレッドといくつかのスレーブスレッドで構成されるチームを形成します．これらのスレッドは，互いに独立して並列にプログラム命令を実行することができます．並列処理が一旦完了すると，スレーブスレッドはマスタースレッドに制御を戻し，次の並列処理領域まで非アクティブとなります

（join）．スレッドは同じメモリアドレス空間（例えば，親プロセスのメモリアドレス空間）を共有します．そして，この共有メモリが通信と同期に使われます．OpenMP はアクセス制御のメカニズムを提供します．各スレッドは共有変数にアクセスできるだけでなく，自分だけがアクセス可能な他の変数のプライベートコピーを持つことができます．チーム内のスレッドは並列領域内に独自の並列領域を作成できます．この次のレベルの並列処理では，新しいチームを作るスレッドがそのチームのマスタースレッドになります．これをネスト並列処理（nested parallelism）と呼びます．

シリアルプログラムとの違いとしてマルチスレッドプログラムで理解しておかなければならないことは，同一の結果は保証されないし，また期待すべきでもないということです．並列プログラムでは多くの場合，同一の結果を得ることは不可能です．なぜなら，使用するスレッド数が異なると浮動小数点演算の順番が異なり（しかし，どれも正しい），結果として丸め誤差の累積が変わるからです．結果の再現性についての更なる議論はセクション 3.9 をご参照ください．

3.10.2.2 NAG C ライブラリはどのように並列化されているか?

マルチスレッド NAG C ライブラリは OpenMP によるマルチスレッディングを利用しているという点でシリアル NAG C ライブラリとは異なります．OpenMP は多くの異なるハードウェアプラットフォームの多くの異なるコンパイラで利用可能な共有メモリプログラミングについての移植性のある仕様です．

マルチスレッド NAG C ライブラリの全ての関数が並列化されているわけではないので注意してください．各関数の並列性とパフォーマンスについての詳細は，各関数ドキュメントのセクション 8 をご参照ください．

NAG C ライブラリ関数の呼び出しが並列化の恩恵を受ける状況には以下の 2 つがあります：

1. 呼び出した NAG C ライブラリ関数が OpenMP を用いて並列化された NAG 固有の関数である．もしくは，呼び出した NAG C ライブラリ関数が OpenMP を用いて並列化された別の NAG 固有の関数を内部的に呼び出している．これは，NAG C ライブラリのマルチスレッド版にのみ適用されます．
2. 呼び出した NAG C ライブラリ関数が BLAS または LAPACK 関数を呼び出している．ご利用の NAG C ライブラリ製品で使用が推奨されているベンダーライブラリは（NAG C ライブラリ自体が並列化されているかどうかに関わらず）並列化されています．更なる情報は，ご利用の製品のユーザーノートをご参照ください．

NAG C ライブラリ関数の完全なリストならびに並列化の状況はセクション 3.10.3 をご参照ください．

ライブラリの非効率な使用を避けるために，ライブラリの中で OpenMP がどのように利用されているかを知ることは有益です．

並列化された NAG 固有の関数は，実行中に OpenMP を用いて複数の並列処理領域でスレッドチームを生成します．スレッドチームは並列領域の開始時に分岐（fork）し，並列領域の終了時に合流（join）します．fork と join は両方とも呼び出された関数の内部で起こります．ただし，ユーザー提供関数の中に OpenMP 指示文があれば，そこでスレッドチームが利用されるという状況はあり得ます．並列領域内に含まれていない指示文を親無し指示文（orphaned directive）と呼びます．（詳細は関数ドキュメントのセクション 8 をご参照ください．）NAG 関数内の OpenMP 指示構文は NAG コード内で生成されたスレッドチームによって実行されます（つまり，ライブラリ自身には親無し指示文はありません）．本ドキュメントでは，ユーザーノートで推奨されているコンパイラの使用，特に単一の OpenMP ランタイムライブラリの使用を想定しています．従って，す

べての OpenMP 環境変数はユーザーコードと NAG 関数に適用されます．しかし，それらをオーバーライドする仕組みを持っているベンダーライブラリには適用されないかもしれません．NAG C ライブラリでは，プログラム全体のスレッドを制御するための関数をチャプター x06 に提供しています．これは，NAG C ライブラリによって呼び出されるベンダーライブラリ固有のスレッドにも適用されます．ユーザープログラムの並列領域から NAG 関数を呼び出すときには注意が必要です．ネスト並列処理（nested parallelism）が有効になっている場合（デフォルトでは無効になっています），NAG 関数は各スレッドからの呼び出しに対してスレッドチームを fork および join します．これによりシステムリソースの競合が起き，パフォーマンスが著しく低下します．競合によるパフォーマンスの低下は，要求されたスレッド数がマシンの物理コア数を超える場合や，ハードウェアリソースが他のプロセス（共有システムにおける他のユーザープロセス）の実行でビジーな場合にも起こります．このような理由から，マシンで利用可能な物理コア数を考慮して，リソースの競合を最小にするスレッド数を選択する必要があります．スレッド数の設定についてのアドバイスは，ご利用の製品のユーザーノートをご参照ください．

他のスレッドメカニズムからマルチスレッド NAG 関数を呼び出す場合は，そのスレッドメカニズムが，ご利用のマルチスレッド NAG C ライブラリをビルドする際に使用された OpenMP コンパイラランタイムと互換性があるかどうかが問題となります．更なるアドバイスは，ご利用の製品のユーザーノートをご参照ください．

NAG 関数の多くは並列化の対象になっていませんが，これらの並列化されていない NAG 関数でも，並列化された NAG 関数，および／または，並列化されたベンダー関数（例えば，BLAS と LAPACK）を内部的に呼び出して利用している関数は間接的に並列化の恩恵を受けます．従って，ライブラリ全体に渡り多くの箇所で並列処理は行われます．並列化によるパフォーマンスの向上は，呼び出す関数の種類，問題サイズと引数，システム設計そして OS 構成に依って変わってきます．もし，同様のデータサイズと引数で関数を頻繁に呼び出すなら，最適なパフォーマンスを得るために，色々なスレッド数を試してみることは価値があります．

一般的な指針として，以下の分野の主な関数は共有メモリ並列化の恩恵を受けます：

- Dense and Sparse Linear Algebra（密・スパース線形代数）
- FFTs（高速フーリエ変換）
- Random Number Generators（擬似乱数生成）
- Quadrature（数値積分）
- Partial Differential Equations（偏微分方程式）
- Interpolation（補間）
- Curve and Surface Fitting（曲線・曲面のあてはめ）
- Correlation and Regression Analysis（相関・回帰分析）
- Multivariate Methods（多変量解析）
- Time Series Analysis（時系列解析）
- Financial Option Pricing（オプションプライシング）
- Global Optimization（大域的最適化）
- Wavelets（ウェーブレット変換）

3.10.3 並列化関数

NAG C ライブラリのマルチスレッド版では多くの関数が OpenMP を用いて並列化されています．マルチスレッド版の製品コードの形式はシリアル版の ‘CL_____’ に対して ‘CS_____’ となります．詳細については，各関数ドキュメントのセクション 8 をご参照ください．NAG によって並列化された関数のリストは Multithreaded Routines ドキュメントをご参照ください．BLAS または LAPACK 関数を内部的に呼び出す関数のリストも同じドキュメントに含まれています．NAG C ライブラリが利用するベンダーライブラリに含まれる BLAS および LAPACK 関数は，NAG C ライブラリがシリアル版かマルチスレッド版かに関わらず，ベンダーライブラリ内で並列化されています．詳細についてはベンダーライブラリのドキュメントをご参照ください．NAG 製品固有の情報については，各製品のユーザーノートをご参照ください．

4 ドキュメントの使い方

4.1 マニュアルの使用

NAG C Library Manual, Mark 26（ライブラリマニュアル）は，NAG C ライブラリに対して次の役割を果たすべく作成されています．

- 数値計算と統計解析の種々の分野に関する背景情報を提供する．
- 特定の問題の解決のためにどのライブラリ関数を使用すべきかに関する助言を与える．
- C プログラムからライブラリ関数を呼び出し，その結果を評価する上で必要になるすべての情報を提供する．

マニュアルの先頭部（Introduction）には背景情報や追加情報に関する一般的な紹介資料が含まれています．

“NAG C Library News, Mark 26”というドキュメントは，新たに追加された関数，削除予定の関数，当 Mark で削除された関数の詳細を記述しています．また，当 Mark のユーザーに影響を与える内部的な変更についても説明があります．

“Advice on Replacement Calls for Withdrawn/Superseded Functions”（削除済み関数に代わる推奨関数）というドキュメントにはユーザープログラムに対して必要となる変更についてのアドバイスが記述されています．

オンラインドキュメントでは Keyword and GAMS Search を用いて，キーワードでライブラリを検索することができます．検索には，各ページの右上にある Keyword Search ボックスが利用できます．

該当しそうなチャプターや関数が見つかったら，まず **Chapter Introduction**（チャプターイントロダクション）をお読みください．該当する数値計算分野に関する背景情報，および関数の選択に関する推奨案（インデックス，テーブル，決定木を含む）が記述されています．

関数の選択が終わったら，今度は **Function Document**（関数ドキュメント）の参照が必要です．個々の関数ドキュメントは本質的に自己完結型の資料です（関連ドキュメントへの参照は含まれますが）．それには手法の説明，各引数の詳細仕様，個々のエラー復帰に関する説明，精度に関するコメント，および（ほとんどの場合）関数の用法を示す Example プログラムが含まれています．場合によっては，Example プログラムの実行結果を示すプロットが付随しています．

4.2 ドキュメントの構成

ライブラリマニュアルはライブラリを使用する際に基本となるドキュメントです．それはライブラリと同一のチャプター構成を取っています．すなわち，複数のライブラリ関数から成る個々のチャプターにはマニュアルの同名のチャプターが対応しています．これらのチャプターはアルファベット順に並べられています．また，マニュアルの先頭部（Introduction）にはライブラリの使用に際して基本となるドキュメントが配置されています．

それぞれのチャプターは次のドキュメントから構成されます．

- **Chapter Contents** - 例えば，d01 - Quadrature
- **Chapter Introduction** - 例えば，d01 Chapter Introduction
- **Function Documents** - チャプターに含まれる関数ごとの仕様書

関数ドキュメントの名称は関数名（ショートネーム）と同一です．チャプター内において，関数ドキュメントはショートネームのアルファベット順に並んでいます．ライブラリには LAPACK, Release 3 で提供される（数値計算の）関数がすべて含まれており，NAG が提供するインターフェース関数によって呼び出すことができます．これらの関数名は，小文字である事と先頭に ‘nag_’ が付いている事を除き，LAPACK の命名規則に従っています．

すべての関数ドキュメントは 10 のセクションからなる同一の構成を取っています．

1. **Purpose**（目的）
2. **Specification**（仕様）
3. **Description**（説明）
4. **References**（参考文献）
5. **Arguments**（引数）（セクション 4.3 参照）
6. **Error Indicators and Warnings**（エラーと警告）
7. **Accuracy**（精度）
8. **Parallelism and Performance**（並列性とパフォーマンス）
9. **Further Comments**（コメント）
10. **Example**（使用例）（セクション 4.4 参照）

一部のドキュメント（主に，チャプター e04, e05, h）においてはさらに 3 つのセクションが加わります．

11. **Algorithmic Details**（アルゴリズム詳細）
12. **Optional Parameters**（オプショナルパラメーター）
13. **Description of Monitoring Information**（監視情報）

上記のセクション 11. と 13. は無い場合もあります．その場合 **Optional Parameters**（オプショナルパラメーター）がセクション 11. として登場します．

4.3 引数の仕様

各関数ドキュメントのセクション 5 には，引数の仕様が引数リストに現れる順に記述されています．

4.3.1 引数の分類

引数は次のように分類されます.

Input（入力）: 関数を実行する際に，これらの引数に対し値を代入しておく必要があります．これらの値は関数から復帰した際も変化しません.

Output（出力）: 関数を実行する際に，これらの引数に対し値を代入しておく必要はありません．関数の中で値が設定されます.

Input/Output（入出力）: 関数を実行する際に，これらの引数に対し値を代入しておく必要があります．また，関数の中で値が変更されることがあります.

Communication Structure and Arrays（コミュニケーション構造体および配列）: 一つの関数から別のものへデータを受け渡すために使用される引数です.

External Function（外部関数）: ユーザーによって提供されなくてはならない関数（例えば，被積分関数の値を評価したり，中間結果を出力したりするためのもの）．通常それは呼び出し元のプログラムの一部として提供されなくてはなりません．関数ドキュメントにはこの外部手続きの引数リストと各引数の仕様の詳細が含まれることになります．その引数はライブラリ関数の引数と同様の形で分類されますが，ユーザーはそれを呼び出すのではなく自ら書く立場にあるため分類の意味は異なってきます.

Input（入力）: 呼び出し時に値が設定されます.
Output（出力）: 手続きから復帰する際に値を設定してください.
Input/Output（入出力）: 呼び出し時に値が設定されます．手続きから復帰する際に必要に応じて値を設定してください.

4.3.2 制約条件と推奨値

入力引数の仕様にある“*Constraint*”，または“*Constraints*”（制約条件）という語は，その引数に対する適正な値の範囲を規定します.

（例） *Constraint*: $\mathrm{n} > 0$

不正な引数値を用いて呼び出された場合（例えば，$\mathrm{n} = 0$），関数は通常エラー復帰します.

場合によっては，新しい Mark（バージョン）で既存の関数が拡張され，引数の制約が緩くなることがあります．しかし，これによる既存のコードへの影響はありませんし，新しいコードでは拡張された機能を利用することができます.

“*Suggested value*”（推奨値）という語は，ユーザーがどのような値にすべきかわからないケースを想定して，入力引数に対する妥当な設定値を示唆します（例えば，精度や最大反復回数など）．個別の問題に対しそれらの値が適切ではないと判断される場合には異なる値を設定してください.

4.4 Example プログラムと結果

関数ドキュメントのセクション 10 に記述されている **Example プログラム**は，関数の呼び出し方を示す具体例です．修正が容易な形で作られているので，ユーザープログラムを開発する際のテンプレートとしてもご利用いただけます.

これらの Example プログラムはライブラリの実装ごとに機械可読（machine-readable）な形で配布されます．必要な修正は実施済みです．多くのサイトでは，この形式でプログラムをユーザーに公開しています．製品で提供される nagc_example スクリプトを用いて，これらの Example プログラムを簡単に利用することができます．実装固有の修正を加えていない一般形式の Example プログラムは，NAG のウェブサイトから直接ダウンロードすることができます．ご利用の実装に対するユーザーノート（Users' Note）には，一般形式の Example プログラムに対して必要となる変更についての記載があります．

これらの Example プログラムは，プラットフォーム間のポータビリティのために，`NAG_CALL` の様なプロセッサ識別子を含んでいるかもしれません．そのような識別子はヘッダーファイル `nag.h` または `nag_types.h` を通して（C プリプロセッサ `#define` を用いて）適切な実装固有のトークンに置き換えられます．

Example プログラムの実行結果はすべての実装で同一になるとは限りません．また，マニュアルに記載されている結果とも一致しないことがあります．

多くの関数ドキュメントでは Example プログラムの実行結果に対してプロット図が提供されています．場合によっては，解のプロット図をより典型的なものとするために，より大きな実行結果を生成するような（若干の）変更が Example プログラムに施されています．

4.5 オンラインドキュメント

ライブラリマニュアルは次の形式で閲覧することができます．

- **HTML** - HTML, SVG, MathML によって閲覧可能なマニュアル（各ドキュメントの PDF 版へのリンクを含む）
- **PDF** - PDF のしおり（または HTML 目次ファイル）によって閲覧可能な PDF マニュアル
- **PDF（単一ファイル）** - 複数の PDF マニュアルを 1 つにまとめた単一の PDF マニュアル
- **Windows HTML ヘルプ（単一ファイル）** - Windows HTML ヘルプ形式のマニュアル

単一のファイルからなる形式は関数ごとに一つファイルを使う形式よりもコンパクトです．例えば，マニュアル全体でテキスト検索を行うことができます．しかし，いくつかの関数のドキュメントしか見ないのであれば大きなサイズのファイルは不便かもしれません．

以下のセクションでは，ドキュメントを表示するために必要なソフトウェアを入手する方法を説明し，ブラウザを使う場合と使わない場合でドキュメントを閲覧する方法を説明します．

4.5.1 HTML 形式

4.5.1.1 HTML ファイルの表示

これらのファイルは特定のブラウザに固有の機能は使用せず W3C 勧告（HTML，MathML，SVG，CSS）に準拠しています．

これらの言語をサポートするにはブラウザのアップデートや追加フォントのインストールが必要な場合があります．

4.5.1.2 Firefox（その他 Mozilla ベースのブラウザ）

Firefox 4 以降のバージョンの Firefox ではデフォルトで HTML ファイルに MathML が表示されます．

STIX や他の OpenType 数学フォントがご利用のシステムに含まれていない場合は，これらのフォントをインストールすることによって数式の描画が改善されます．インストーラーの詳細は Firefox MathML フォントページをご参照ください．
http://www.mozilla.org/projects/mathml/fonts/

4.5.1.3 その他のブラウザ

Firefox を使っていない場合，ページの javascript は MathML を有効にするために MathJax javascript ライブラリ（http://www.mathjax.org）を読み込みます．デフォルトでは MathJax コンテンツ配信ネットワークを使用してウェブからロードされます．インターネットに接続せずにドキュメントを参照する必要がある場合は，前のセクションで説明したように Firefox を使用するか，もしくは http://docs.mathjax.org/en/latest/installation.html からダウンロードした MathJax のローカルコピーをご利用のローカルファイルサーバーまたはローカルファイルシステムに解凍してください．そして，../styles/nagmathml.js ファイルの `cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.0` という行を，先ほどのローカルコピーを参照するように編集してください．

4.5.1.4 HTML ファイルの閲覧

メインの目次ファイル（html/frontmatter/manconts.html）が提供されており，個々のチャプターコンテンツにリンクしています．ブラウザを使ってこの目次ファイルから各ドキュメントを閲覧します．

各ドキュメントは色々な要素（引数，セクション，チャプターコンテンツなど）へのハイパーリンクを含んでいます．各要素に対して使用される文字色を以下の表に示します．

文字色	要素
black	NAG 型
green	付録，チャプターイントロダクション，決定木，一般的なイントロダクション，セクション
grey	廃止されたドキュメント
pale blue	方程式，図形，リスト内の項目，注釈，書誌参照，表，URL，逐語的な項目，ウェブサイト
navy blue	IFAIL 値
red	引数名
pink	メンバ
purple	オプショナルパラメーター
royal blue	HTML 目次，Example プロット，関数ドキュメント，目次からの Example へのリンク

4.5.1.5 HTML ファイルの印刷

HTML ファイルをブラウザから印刷することは可能ですが，ブラウザからの印刷のサポート，特に数式の印刷のサポートは，ご利用のブラウザ，プラットフォームおよびプリンタードライバーのバージョンによって大きく異なります．

5 NAG C ライブラリの設計と開発

NAG C ライブラリの設計と開発に関する種々の見解，NAG の技術的方針と組織に関する情報については Ford (1982), Ford *et al*. (1979), Ford and Pool (1984), Hague *et al*. (1982) をご参照ください.

6 NAG C ライブラリの標準準拠

NAG C ライブラリは多くの国際標準に準拠しています．ISO/IEC (1990), Kernighan and Ritchie (1988), ANSI/IEEE POSIX (1995), Basic Linear Algebra Subprograms Technical (BLAST) Forum (2001) をご参照ください.

7 参考文献

The BLAS Technical Forum Standard

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

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

ANSI/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

---

日本語版 26.2.0