NAG Library, Mark 31.1, Multithreaded
NSL6I311BL - Licence Managed
Linux, 64-bit, Intel Classic C/C++ or Intel Classic Fortran

ユーザーノート

目次

1 はじめに

このドキュメントは、タイトルに記載されている NAG ライブラリ実装のすべてのユーザーにとって必読の資料です。NAG Mark 31.1 ライブラリマニュアル（以下、ライブラリマニュアルと呼びます）に記載されている情報を補完する実装固有の詳細情報を提供しています。ライブラリマニュアルで「お使いの実装のユーザーノート」という記述がある場合は、このノートを参照してください。

さらに、NAG は任意のライブラリルーチンを呼び出す前に、ライブラリマニュアル（セクション 5 参照）から以下の参考資料を読むことをお勧めします：

(a) NAG ライブラリの使用方法
(b) 章の概要
(c) ルーチンドキュメント

2 追加情報

以下の URL をご確認ください：

https://support.nag.com/doc/inun/ns31/l6i1bl/supplementary.html

この実装の適用性や使用方法に関する新しい情報の詳細が記載されています。

3 一般情報

この NAG ライブラリの実装では、Intel® Math Kernel Library for Linux（MKL）というサードパーティのベンダー性能ライブラリを使用して、Basic Linear Algebra Subprograms（BLAS）と Linear Algebra PACKage（LAPACK）ルーチン（セクション 4 に記載されているルーチンを除く）を提供する静的ライブラリと共有ライブラリが用意されています。また、これらのルーチンの NAG 参照版を使用した自己完結型の静的ライブラリと共有ライブラリも提供しています（自己完結型ライブラリと呼びます）。この実装は MKL のバージョン 2021.0.4 でテストされており、このバージョンは本製品の一部として提供されています。MKL の詳細については、Intel のウェブサイト（https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html）をご覧ください。最高のパフォーマンスを得るには、提供されている MKL ベンダーライブラリに基づく NAG ライブラリのバリアント（`libnag_mkl.a` または `libnag_mkl.so`）を使用することをお勧めします。自己完結型の NAG ライブラリ（`libnag_nag.a` または `libnag_nag.so`）よりも優先して使用してください。

この実装には、32 ビット整数（`lp64` で示される）と 64 ビット整数（`ilp64` で示される）の両方で使用するためのライブラリ（および関連ファイル）が含まれています。

NAG AD ライブラリはこの実装には含まれていません。

NAG ライブラリは、使用されたメモリがライブラリ自体によって、または C ルーチンの場合はユーザーが NAG_FREE()を呼び出すことで回収できるように注意深く設計されています。ただし、ライブラリ自体がコンパイラのランタイムやその他のライブラリに依存しており、これらが時々メモリリークを起こす可能性があります。NAG ライブラリにリンクされたプログラムにメモリトレースツールを使用すると、これが報告される場合があります。リークするメモリの量はアプリケーションによって異なりますが、過剰になることはなく、NAG ライブラリへの呼び出しが増えても無制限に増加することはありません。

マルチスレッドアプリケーション内で NAG ライブラリを使用する場合は、以下のドキュメントを参照してください：

- CL インターフェースのマルチスレッド処理
- FL インターフェースのマルチスレッド処理

（適切な方）詳細情報については。

スレッド化されたアプリケーションで提供されている Intel MKL ライブラリを使用する際の詳細情報は、https://software.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/managing-performance-and-memory/improving-performance-with-threading.html で入手できます。

この実装で提供されているライブラリは OpenMP でコンパイルされています。ただし、異なるコンパイラの OpenMP ランタイムライブラリは互換性がない場合があるため、インストールノートのセクション 2.2 に記載されているコンパイラと対応する OpenMP ランタイムを使用している場合にのみ、自身の OpenMP コード（セクション 4 に記載されているルーチンのユーザー提供関数で必要な OpenMP 文を含む）と組み合わせてこの実装を使用することをお勧めします。

システムのデフォルトのスレッドスタックサイズは、マルチスレッドアプリケーション内ですべての NAG ライブラリルーチンを実行するのに十分でない場合があることに注意してください。OpenMP 環境変数 `OMP_STACKSIZE` を使用してこのスタックサイズを増やすことができます。

Intel は MKL に条件付きビット単位再現性（BWR）オプションを導入しました。ユーザーのコードが特定の条件を満たしていれば（https://software.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/obtaining-numerically-reproducible-results/reproducibility-conditions.html を参照）、`MKL_CBWR` 環境変数を設定することで BWR を強制できます。詳細については、MKL のドキュメントを参照してください。ただし、多くの NAG ルーチンはこれらの条件を満たしていないことに注意してください。つまり、MKL 上に構築された特定の NAG ライブラリに対して、`MKL_CBWR` を設定することで異なる CPU アーキテクチャ間ですべての NAG ルーチンの BWR を保証することは不可能かもしれません。ビット単位再現性に関する一般的な情報については、NAG ライブラリの使用方法のセクション 8.1 を参照してください。

3.1 ライブラリへのアクセス

このセクションでは、ライブラリがディレクトリ `[*install_dir*]` にインストールされていることを前提としています。デフォルトでは `[*install_dir*]`（インストールノート (in.html) 参照）は `$HOME/NAG/nsl6i311bl` です。ただし、インストールを行った担当者によって変更されている可能性があります。その場合は、その担当者に確認してください。環境変数 `LD_LIBRARY_PATH` は、リンク時および実行時に `[*install_dir*]` 以下の適切なライブラリの場所を指すように正しく設定する必要があることに注意してください。これを行う方法については、以下を参照してください。

NAG ライブラリは、NAG C ライブラリと NAG Fortran ライブラリ（Fortran ライブラリインターフェースへの C ラッパーを含む）の両方のユーザーのための統合された代替品です。この実装に含まれるさまざまなライブラリの呼び出しを支援するために、スクリプト `nagvars.sh` と `nagvars.csh` が含まれており、NAG ルーチンを呼び出すアプリケーションのコンパイルとリンクを支援する NAG 固有の環境変数を設定します。また、標準の環境変数 `PATH` と `LD_LIBRARY_PATH` を修正して、コンパイル時、リンク時、実行時に NAG 実行可能プログラムとライブラリが見つかるようにします。

`nagvars` スクリプトは次のように使用するように設計されています：

```
. [*install_dir*]/scripts/nagvars.sh [-help] [-unset] [-quiet] [-ifx] [-gnu] \
  {int32,int64} {vendor,nag} {static,dynamic}
```

または

```
source [*install_dir*]/scripts/nagvars.csh [-help] [-unset] [-quiet] [-ifx] [-gnu] \
       {int32,int64} {vendor,nag} {static,dynamic}
```

ここで：

- `nagvars.sh` は Bourne、Bash または同等のシェルで使用します（注意：Ubuntu などの Linux の一部の Debian 派生ディストリビューションで利用可能な Dash ではありません）。`nagvars.csh` は Csh、Tcsh または同等のシェルで使用します。
- `{int32,int64}` は、NAG ルーチン内の整数引数および変数のデフォルトサイズを指定します。
- `{vendor,nag}` は、BLAS および LAPACK に提供されている MKL ライブラリに依存する NAG ライブラリのセットを使用するか（オプション `vendor`）、自己完結型の NAG ライブラリを使用するか（オプション `nag`）を指定します。最高のパフォーマンスを得るには、オプション `vendor` をお勧めします。
- `{static,dynamic}` は、NAG ライブラリの静的バージョンまたは動的（共有）バージョンにリンクするかを指定します。

デフォルト値は使用されないため、3 つのオプションすべてを設定する必要があります。指定する順序は重要ではありません。オプションの `nagvars` スクリプト引数は次のとおりです：

- `-help` スクリプトに関する情報を表示します。
- `-unset` NAG 固有の環境変数をクリアし、標準の環境変数 `PATH` および `LD_LIBRARY_PATH` から **この NAG 実装のみ** のマテリアルへのすべての参照を削除しようとします。
- `-quiet` stdout への出力を抑制します。
- `-ifx` このセクションの最後で説明します。
- `-gnu` このセクションの最後で説明します。

したがって、Bash で環境を設定するために `nagvars` スクリプトを使用する例は次のとおりです：

```
source [*install_dir*]/scripts/nagvars.sh int64 vendor dynamic
```

設定される NAG 固有の環境変数は次のとおりです：

- `NAGLIB_CC` - この NAG ライブラリの作成に使用された C コンパイラ。
- `NAGLIB_CXX` - この NAG ライブラリの作成に使用された C++コンパイラ。
- `NAGLIB_F77` - この NAG ライブラリの作成に使用された Fortran コンパイラ。
- `NAGLIB_CFLAGS` - 必要または推奨される C コンパイラフラグ。
- `NAGLIB_CXXFLAGS` - 必要または推奨される C++コンパイラフラグ。
- `NAGLIB_FFLAGS` - 必要または推奨される Fortran コンパイラフラグ。
- `NAGLIB_INCLUDE` - NAG C ヘッダおよび/または Fortran モジュールファイルへのインクルードパス。
- `NAGLIB_CLINK` - C コンパイラを使用して通常の NAG（およびオプションでベンダーBLAS および LAPACK）ルーチンにリンクするために必要なライブラリ。
- `NAGLIB_CXXLINK` - C++コンパイラを使用して通常の NAG（およびオプションでベンダーBLAS および LAPACK）ルーチンにリンクするために必要なライブラリ。
- `NAGLIB_FLINK` - Fortran コンパイラを使用して通常の NAG（およびオプションでベンダーBLAS および LAPACK）ルーチンにリンクするために必要なライブラリ。

NAG ライブラリおよび提供されている Intel MKL ライブラリ（必要に応じて）を使用するには、次のようにリンクできます：

- C プログラムの場合：
  `${NAGLIB_CC} ${NAGLIB_CFLAGS} ${NAGLIB_INCLUDE} program.c ${NAGLIB_CLINK}`
- または C++プログラムの場合：
  `${NAGLIB_CXX} ${NAGLIB_CXXFLAGS} ${NAGLIB_INCLUDE} program.cpp ${NAGLIB_CXXLINK}`
- または Fortran プログラムの場合：
  `${NAGLIB_F77} ${NAGLIB_FFLAGS} ${NAGLIB_INCLUDE} program.f90 ${NAGLIB_FLINK}`

新しいバージョンのコンパイラを使用している場合など、コンパイラのランタイムライブラリなどを指すように `LD_LIBRARY_PATH` を設定する必要がある場合があることに注意してください。

異なるコンパイラ、または実際には異なるバージョンの Intel コンパイラを使用している場合は、`[*install_dir*]/rtl/lib/intel64` で提供されている Intel コンパイラのランタイムライブラリに対してリンクする必要がある場合があります。これは、`LD_LIBRARY_PATH` に `[*install_dir*]/rtl/lib/intel64` を追加することで容易になる場合があります。

Intel icx または ifx コンパイラから呼び出すには、icc または ifort の代わりにそれぞれ icx または ifx を単に置き換えます（ただし、ifx での静的リンクに関する以下の注意を参照してください）。これを行う最も簡単な方法は、`nagvars.sh` および/または `nagvars.csh` スクリプトの `-ifx` オプションを使用することです。このオプションを使用すると、NAG 固有の環境変数が変更され、Intel ifx、icx、または icpx コンパイラ（適切なもの）が使用されます。

Intel Fortran コンパイラ（ifort または ifx、バージョン 2022.7 以降）は、Linux システムにおいて多重定義によるリンクエラーを引き起こします。これは、バージョン 2022.7 におけるこれらのコンパイラのデフォルトのリンク動作の変更に起因します。このバージョンより前では、コンパイラは Fortran コンパイラのランタイムライブラリを自動的にリンクしませんでした。

最近の NAG ライブラリは 2022.7 より少し前のバージョンの ifort を使用してコンパイル・ビルドされたため、リンク時にコンパイラのランタイムライブラリのセットを明示的にリストする必要をなくすために、一部の Intel ランタイムライブラリがライブラリの静的バリアントに静的に組み込まれていました。2022.6 以降のバージョンの Intel Fortran コンパイラは、必要なランタイムライブラリを明示的にリストすることなく自動的にリンクします。そのため、これらのランタイムを同様に含む NAG 静的ライブラリにリンクすると、多重定義が発生します。

Linux 上でこれらの最近の静的バージョンのライブラリを ifort または ifx でリンクする場合の回避策は、「-intel-shared」フラグを追加することです。このフラグは C コンパイラを使用してリンクする場合には必要ありません。あるいは、代わりに NAG ライブラリの共有バリアントのいずれかを使用することもできます。共有ライブラリとのリンクは、パフォーマンスに大きな損失がなく、実行可能ファイルのサイズがはるかに小さくなるため、推奨されるリンク形式です。

Linux 上の NAG ライブラリの Mark 31.2 リリースから、Intel ifx コンパイラを使用した静的リンクでは、「-intel-shared」リンクフラグが文書化され使用されるようになり、このフラグは nagvars および nag_example スクリプトに含まれるようになります。C または C++コンパイラでリンクするユーザーの便宜のため、Intel Fortran コンパイラのランタイムは引き続き静的ライブラリに静的に含まれます。

このバージョンの NAG ライブラリは、GNU gcc および g++コンパイラから呼び出すことができます。少なくとも、当社のウェブサイトの追加情報ページに記載されているバージョンの gcc を使用します。これを行う最も簡単な方法は、`nagvars.sh` および/または `nagvars.csh` スクリプトの `-gnu` オプションを使用するこ

とです。このオプションを使用すると、NAG 固有の環境変数が変更され、GNU gcc および g++（適切なもの）が使用されます。

GNU gfortran はサポートされていないため、Fortran 関連の NAG 環境変数は引き続き Intel ifort コンパイラを参照することに注意してください。また、MKL とのリンクでは、NAG ライブラリに存在する OpenMP ルーチンとの一貫性を保つために、引き続き Intel コンパイラの OpenMP ランタイムが使用されることにも注意してください。

3.1.1 使用するスレッド数の設定

NAG ライブラリのこの実装と MKL は、OpenMP を使用してライブラリルーチンの一部でスレッド処理を実装しています。実行時に使用されるスレッド数は、環境変数 `OMP_NUM_THREADS` を適切な値に設定することで制御できます。

C シェルでは、次のように入力します：

```
setenv OMP_NUM_THREADS N
```

Bourne シェルでは、次のように入力します：

```
OMP_NUM_THREADS=N
export OMP_NUM_THREADS
```

ここで `N` は必要なスレッド数です。環境変数 `OMP_NUM_THREADS` は、必要に応じてプログラムの各実行の間で再設定できます。プログラムの実行中にプログラムの異なる部分で使用するスレッド数を変更したい場合は、NAG ライブラリの第 X06 章にこのプロセスを支援するためのルーチンが用意されています。

NAG ライブラリと MKL のいくつかのルーチンには複数レベルの OpenMP 並列性が存在する可能性があり、また、独自のアプリケーションの OpenMP 並列領域内からこれらのマルチスレッドルーチンを呼び出す場合もあります。デフォルトでは、OpenMP のネストされた並列性は無効になっているため、最も外側の並列領域のみが実際にアクティブになり、上記の例では `N` 個のスレッドを使用します。内部レベルはアクティブにならず、1 つのスレッドで実行されます。OpenMP のネストされた並列性が有効になっているかどうかを確認し、有効/無効を選択するには、OpenMP 環境変数 `OMP_NESTED` を照会および設定するか、第 X06 章の適切なルーチンを使用します。OpenMP のネストされた並列性が有効になっている場合、上記の例では各並列領域で上位レベルの各スレッドに対して `N` 個のスレッドを作成するため、OpenMP 並列性が 2 レベルある場合は合計 `N*N` 個のスレッドとなります。ネストされた並列性をより詳細に制御するために、環境変数 `OMP_NUM_THREADS` をカンマ区切りのリストとして設定し、各レベルで希望するスレッド数を指定できます。

C シェルでは、次のように入力します：

```
setenv OMP_NUM_THREADS N,P
```

Bourne シェルでは、次のように入力します：

```
OMP_NUM_THREADS=N,P
export OMP_NUM_THREADS
```

これにより、最初のレベルの並列性で `N` 個のスレッドが作成され、内部レベルの並列性に遭遇したときに各外部レベルスレッドに対して `P` 個のスレッドが作成されます。

注意：環境変数 `OMP_NUM_THREADS` が設定されていない場合、デフォルト値はコンパイラやベンダーライブラリによって異なり、通常は 1 か、システムで利用可能な最大コア数に等しくなります。後者は、システムを他のユーザーと共有している場合や、独自のアプリケーション内で高レベルの並列性を実行している場

合に問題になる可能性があります。したがって、常に OMP_NUM_THREADS を希望する値に明示的に設定することをお勧めします。

一般的に、使用することをお勧めする最大スレッド数は、共有メモリシステム上の物理コア数です。ただし、ほとんどの Intel プロセッサはハイパースレッディングと呼ばれる機能をサポートしており、これにより各物理コアが同時に最大 2 つのスレッドをサポートし、オペレーティングシステムには 2 つの論理コアとして認識されます。この機能を利用することが有益な場合もありますが、この選択は特定のアルゴリズムと問題サイズに依存します。パフォーマンスが重要なアプリケーションについては、追加の論理コアを利用する場合としない場合でベンチマークを行い、最適な選択を決定することをお勧めします。これは通常、OMP_NUM_THREADS を介して使用するスレッド数を適切に選択するだけで達成できます。ハイパースレッディングを完全に無効にするには、通常、システムの BIOS で起動時に希望の選択を設定する必要があります。

提供されている Intel MKL ライブラリには、MKL 内のスレッド処理をより詳細に制御するための追加の環境変数が含まれています。これらについては、https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2023-0/onemkl-specific-env-vars-for-openmp-thread-ctrl.html で説明されています。多くの NAG ルーチンは MKL 内のルーチンを呼び出すため、MKL 環境変数は NAG ライブラリの動作にも間接的に影響を与える可能性があります。MKL 環境変数のデフォルト設定はほとんどの目的に適しているため、これらの変数を明示的に設定しないことをお勧めします。代わりに、ユーザーは第 X06 章のルーチンを使用することをお勧めします。これらは、呼び出しプログラム、NAG ルーチン、MKL の OpenMP に等しく適用されます。さらなるアドバイスが必要な場合は、NAG にお問い合わせください。

3.2 Fortran インターフェースブロック

NAG ライブラリインターフェースブロックは、ユーザーが呼び出し可能な各 NAG ライブラリ Fortran ルーチンのタイプと引数を定義します。これらは、Fortran プログラムから NAG ライブラリを呼び出すために必須ではありませんが、その使用を強く推奨します。また、提供されている例題を使用する場合は不可欠です。

その目的は、Fortran コンパイラが NAG ライブラリルーチンが正しく呼び出されているかどうかをチェックできるようにすることです。インターフェースブロックを使用することで、コンパイラは以下のことをチェックできます：

(a) サブルーチンがサブルーチンとして呼び出されていること
(b) 関数が正しい型で宣言されていること
(c) 正しい数の引数が渡されていること
(d) すべての引数が型と構造において一致していること

NAG ライブラリインターフェースブロックファイルは、ライブラリの章ごとに整理されています。これらは以下の名前の 1 つのモジュールにまとめられています：

```
nag_library
```

モジュールは、Intel Classic Fortran コンパイラで使用するためにコンパイル済みの形式（.mod ファイル）で提供されています。これらは、各コンパイラ呼び出しで -Ipathname オプションを指定することでアクセスできます。ここで pathname （[*install_dir*]/lp64/nag_interface_blocks または [*install_dir*]/ilp64/nag_interface_blocks）は、必要な整数サイズのコンパイル済みインターフェースブロックを含むディレクトリのパスです。

.mod モジュールファイルは、インストールノートのセクション 2.2 に示されている Fortran コンパイラでコンパイルされました。このようなモジュールファイルはコンパイラに依存するため、これらのモジュールと互換性のないコンパイラを使用して NAG の Example プログラムを使用したり、独自のプログラムでインターフェースブロックを使用したりする場合は、まず独自のコンパイラバージョンでインターフェースブロックを再コンパイルする必要があります。再コンパイルされたインターフェースブロックのセットは、[*install_dir*] ディレクトリから提供されているスクリプトコマンドを使用して、別のディレクトリ（例：nag_interface_blocks_alt）に作成できます：

```
[*install_dir*]/scripts/nag_recompile_mods {int32,int64} nag_interface_blocks_alt
```

このスクリプトは、PATH 環境の Intel Classic Fortran コンパイラのバージョンを使用します。別のバージョンを指定するには、[*install_dir*]/scripts/nag_recompile_mods を実行する前に、そのバージョンの Intel Classic Fortran コンパイラ環境スクリプトを実行するのが最も安全です。

新しくコンパイルされたモジュールのセットをデフォルトセットにするには、まず、[*install_dir*]/scripts/nag_recompile_mods への個別の呼び出しで {int32,int64} の両方のオプションを使用して、両方の整数サイズの新しいモジュールセットを生成することをお勧めします。次に、次のコマンドを使用して両方のデフォルトセットを変更できます（[*install_dir*] は実際のディレクトリパスに置き換えてください）：

```
  mv [*install_dir*]/lp64/nag_interface_blocks [*install_dir*]/lp64/nag_interface_blocks_or
iginal
  mv [*install_dir*]/lp64/nag_interface_blocks_alt [*install_dir*]/lp64/nag_interface_block
s
  mv [*install_dir*]/ilp64/nag_interface_blocks [*install_dir*]/ilp64/nag_interface_blocks_
original
  mv [*install_dir*]/ilp64/nag_interface_blocks_alt [*install_dir*]/ilp64/nag_interface_blo
cks
```

これで、新しくコンパイルされたモジュールファイルを通常の方法で使用できるはずです。

3.3 Example プログラム

配布されている Example 結果は、インストールノートのセクション 2.2 で説明されているソフトウェアを使用して Mark 31.1 で生成されました。これらの Example 結果は、Example プログラムが若干異なる環境（例えば、異なる C または Fortran コンパイラ、異なるコンパイラランタイムライブラリ、または異なる BLAS または LAPACK ルーチンのセット）で実行された場合、厳密に再現できない場合があります。このような違いに最も敏感な結果は以下の通りです：固有ベクトル（多くの場合-1、時には複素数のスカラー倍で異なる場合があります）、反復回数と関数評価回数、および残差やマシン精度と同じオーダーの他の「小さな」量です。

配布されている Example 結果は、32 ビット整数の静的ライブラリ libnag_mkl.a （つまり MKL BLAS と LAPACK ルーチンを使用）で得られたものです。NAG BLAS または LAPACK を使用して例題を実行すると、若干異なる結果が得られる場合があります。

例題資料は、必要に応じてライブラリマニュアルで公開されているものから適応されており、これにより、プログラムはこの実装でそれ以上の変更なしで実行するのに適しています。配布されている Example プログラムは、可能な限りライブラリマニュアルのバージョンよりも優先して使用する必要があります。Example プログラムには、[*install_dir*]/scripts ディレクトリにある nag_example スクリプトを使用すると最も簡単にアクセスできます。

このスクリプトは、Example プログラム（およびそのデータとオプションファイル、もしあれば）のコピーを提供し、プログラムをコンパイルし、適切なライブラリとリンクします。最後に、実行可能プログラムが実行され（必要に応じてデータ、オプション、結果ファイルを指定する適切な引数を使用）、結果がファイルとコマンドウィンドウに送られます。デフォルトでは、nag_example は 32 ビット整数と、自己完結型 libnag_nag.a ライブラリへの静的リンクを使用するように選択します。これらの選択は、それぞれオプションのスイッチ -int64、-shared、-vendor で変更できます。-quiet オプションを指定すると、コメントと上記の各段階で実行されるコマンドの両方の表示を最小限に抑えることができます。 利用可能なオプションのリストを表示するには、次を実行します：

```
nag_example -help
```

nag_example スクリプトは、セクション 3.1 で説明した nagvars スクリプトの使用方法を示していますが、呼び出し元のシェルの環境は変更しないことに注意してください。

対象となる Example プログラムと使用する OpenMP スレッドの数は、コマンドの引数で指定します。例えば、NAG C ルーチンの場合：

```
nag_example e04ucc 4
```

これにより、Example プログラムとそのデータおよびオプションファイル（e04ucce.c、e04ucce.d、e04ucce.opt）が現在のフォルダにコピーされ、プログラムがコンパイルおよびリンクされ、4 つの OpenMP スレッドを使用して実行され、Example プログラムの結果が e04ucce.r ファイルに出力されます。

同様に、NAG Fortran ルーチンの場合：

```
nag_example e04nrf 4
```

これにより、Example プログラムとそのデータおよびオプションファイル（e04nrfe.f90、e04nrfe.d、e04nrfe.opt）が現在のフォルダにコピーされ、プログラムがコンパイルおよびリンクされ、4 つの OpenMP スレッドを使用して実行され、Example プログラムの結果が e04nrfe.r ファイルに出力されます。

3.4 メンテナンスレベル

ライブラリのメンテナンスレベルは、a00aaf または a00aac を呼び出す例題をコンパイルおよび実行するか、nag_example スクリプトを引数 a00aaf または a00aac で呼び出すことで確認できます。セクション 3.3 を参照してください。この例題は、タイトルと製品コード、使用されているコンパイラと精度、マークとメンテナンスレベルを含む実装の詳細を出力します。

3.5 C データ型

この実装では、32 ビット整数用と 64 ビット整数用の両方のライブラリが含まれています。

32 ビット整数ライブラリ（ディレクトリ [*install_dir*]/lp64/lib にあります）の場合、NAG C タイプ Integer と Pointer は次のように定義されています：

NAG タイプ	C タイプ	サイズ（バイト）
Integer	int	4
Pointer	void *	8

64 ビット整数ライブラリ（ディレクトリ [*install_dir*]/ilp64/lib にあります）の場合、NAG C タイプ Integer と Pointer は次のように定義されています：

NAG タイプ	C タイプ	サイズ（バイト）
Integer	long	8
Pointer	void *	8

sizeof(Integer) と sizeof(Pointer) の値は、a00aac Example プログラムでも提供されています。他の NAG データ型に関する情報は、ライブラリマニュアル（セクション 5 参照）の NAG CL インターフェース概要のセクション 3.1.1 で入手できます。

3.6 Fortran データ型と太字斜体語の解釈

この NAG ライブラリの実装には、32 ビット整数用（ディレクトリ [*install_dir*]/lp64/lib にあります）と 64 ビット整数用（ディレクトリ [*install_dir*]/ilp64/lib にあります）の両方のライブラリが含まれています。

NAG ライブラリとドキュメントは、浮動小数点変数にパラメータ化された型を使用しています。したがって、すべての NAG ライブラリルーチンのドキュメントには、以下の型が表示されます：

```
REAL(KIND=nag_wp)
```

ここで、nag_wp は Fortran KIND パラメータです。nag_wp の値は実装によって異なり、その値は nag_library モジュールの使用によって取得できます。我々は nag_wp 型を NAG ライブラリの「作業精度」型と呼びます。なぜなら、ライブラリで使用される多くの浮動小数点引数と内部変数がこの型だからです。

さらに、少数のルーチンは以下の型を使用します：

```
REAL(KIND=nag_rp)
```

ここで、nag_rp は「精度低下」型を表します。現在ライブラリでは使用されていない別の型は：

```
REAL(KIND=nag_hp)
```

で、「高精度」型または「追加精度」型を表します。

これらの型の正しい使用については、ライブラリと一緒に配布されているほとんどの Example プログラムを参照してください。

この実装では、これらの型は以下の意味を持ちます：

```
REAL (kind=nag_rp)      は REAL （つまり単精度）を意味します
REAL (kind=nag_wp)      は DOUBLE PRECISION を意味します
COMPLEX (kind=nag_rp)   は COMPLEX （つまり単精度複素数）を意味します
COMPLEX (kind=nag_wp)   は 倍精度複素数（例えば COMPLEX*16）を意味します
```

さらに、マニュアルの FL インターフェースセクションでは、いくつかの用語を区別するために**太字斜体**を使用する規則を採用しています。詳細については、NAG FL インターフェース概要のセクション 2.5 を参照してください。

3.7 C/C++から NAG Fortran ルーチンを呼び出す

注意深く行えば、NAG ライブラリの Fortran ルーチンを C、C++、または互換性のある環境から使用することができます。Fortran ルーチンをこの方法で使用することは、C 言語ルーチンの同等物が利用できないレ

ガシーFortran ルーチンにアクセスする場合や、他の言語からの使用がより便利な可能性のある、基本的な C データ型のみを使用したより低レベルの C インターフェースを持つ場合に好ましい場合があります。

ユーザーが Fortran と C の型のマッピングを行うのを支援するため、C 視点からの Fortran インターフェースの説明（C ヘッダインターフェース）が各 Fortran ルーチンドキュメントに含まれています。C/C++ヘッダファイル（32 ビット整数用の `[*install_dir*]/lp64/include/nag.h` および 64 ビット整数用の `[*install_dir*]/ilp64/include/nag.h`）も提供されています。NAG Fortran ルーチンをこの方法で使用したいユーザーは、アプリケーションで適切なヘッダファイルを `#include` することをお勧めします。

NAG ライブラリの Fortran ルーチンを C および C++から呼び出す方法についてのアドバイスを提供するドキュメント `alt_c_interfaces.html` も利用可能です。（NAG ライブラリの以前のマークでは、このドキュメントは `techdoc.html` と呼ばれていました。）

3.8 LAPACK、BLAS 等の C 宣言

NAG C/C++ヘッダファイルには、NAG ライブラリに含まれる LAPACK、BLAS、BLAS 技術フォーラム（BLAST）ルーチンの宣言が含まれています。ユーザーは、提供されている Intel MKL など、他のライブラリに関連する C include ファイルからこれらの定義を取得することを好む場合があります。このような状況で、異なる C ヘッダ宣言間の衝突を避けるために、これらのルーチンの NAG 宣言は、セクション 3.1 で説明されている C または C++コンパイル文に以下のコンパイルフラグを追加することで無効にすることができます：

```
-DNAG_OMIT_LAPACK_DECLARATION -DNAG_OMIT_BLAS_DECLARATION -DNAG_OMIT_BLAST_DECLARATION
```

代替 NAG F01、F06、F07、F08 ルーチン名の宣言は残ります。

4 ルーチン固有の情報

この実装の 1 つ以上のルーチンに適用される追加情報は、以下に章ごとにリストされています。

(a) OpenMP 並列領域内でユーザー関数を呼び出すルーチン

この実装では、以下のルーチンは、NAG ルーチン内の OpenMP 並列領域からユーザー関数を呼び出します。

C ルーチン：

```
e05ucc  e05usc  f01elc  f01emc  f01flc  f01fmc  f01jbc  f01jcc
f01kbc  f01kcc
```

Fortran ルーチン：

```
d03raf  d03rbf  e05saf  e05sbf  e05ucf  e05usf  f01elf  f01emf
f01flf  f01fmf  f01jbf  f01jcf  f01kbf  f01kcf
```

したがって、インストールノートのセクション 2.2 にリストされている NAG ライブラリ実装の構築に使用されたものとは異なるコンパイラを使用していない限り、ユーザー関数で孤立した OpenMP ディレクティブを使用できます。また、ユーザーワークスペース配列 `IUSER`、`RUSER`、`CPUSER` をスレッドセーフな方法で使用することを確認する必要があります。これは、ユーザー関数に読み取り専用データを提供するためにのみこれらを使用することで最も良く達成されます。

(b) C06

この実装では、以下の NAG C ルーチンで、可能な限り提供されている MKL ライブラリから Intel 離散フーリエ変換インターフェース（DFTI）ルーチンへの呼び出しが行われます：

```
c06pac  c06pcc  c06pfc  c06pjc  c06pkc  c06ppc  c06pqc  c06prc
c06psc  c06puc  c06pvc  c06pwc  c06pxc  c06pyc  c06pzc  c06rac
c06rbc  c06rcc  c06rdc
```

そして以下の NAG Fortran ルーチンで：

```
c06paf  c06pcf  c06pff  c06pjf  c06pkf  c06ppf  c06pqf  c06prf
c06psf  c06puf  c06pvf  c06pwf  c06pxf  c06pyf  c06pzf  c06raf
c06rbf  c06rcf  c06rdf
```

Intel DFTI ルーチンは内部で独自のワークスペースを割り当てるため、上記の NAG Fortran ルーチンに渡されるワークスペース配列 WORK のサイズを、それぞれのライブラリドキュメントで指定されているものから変更する必要はありません。

(c) F06、F07、F08、F16

F06、F07、F08、F16 の章では、BLAS および LAPACK 派生ルーチンの代替ルーチン名が利用可能です。代替ルーチン名の詳細については、関連する章の概要を参照してください。最適なパフォーマンスを得るためには、アプリケーションは NAG スタイルの名前ではなく、BLAS/LAPACK 名でルーチンを参照する必要があることに注意してください。

多くの LAPACK ルーチンには、呼び出し側がルーチンに提供するワークスペースの量を決定するためにルーチンを照会できる「ワークスペース照会」メカニズムがあります。MKL ライブラリの LAPACK ルーチンは、同等の NAG 参照バージョンのこれらのルーチンとは異なる量のワークスペースを必要とする場合があることに注意してください。ワークスペース照会メカニズムを使用する際は注意が必要です。

この実装では、自己完結型でない NAG ライブラリの BLAS および LAPACK ルーチンへの呼び出しは、以下のルーチンを除いて MKL への呼び出しによって実装されています：

```
blas_damax_val  blas_damin_val  blas_daxpby     blas_ddot       blas_dmax_val
blas_dmin_val   blas_dsum       blas_dwaxpby    blas_zamax_val  blas_zamin_val
blas_zaxpby     blas_zsum       blas_zwaxpby
dbdsvdx dgesvdx dgesvj  dsbgvd  zgejsv  zgesvdx zgesvj  zhbgvd
```

(d) S07 - S21

これらの章の関数の動作は、実装固有の値に依存する場合があります。

一般的な詳細はライブラリマニュアルに記載されていますが、この実装で使用される具体的な値は以下の通りです：

```
s07aa[fc] (nag[f]_specfun_tan)
    F_1 = 1.0e+13
    F_2 = 1.0e-14

s10aa[fc] (nag[f]_specfun_tanh)
    E_1 = 1.8715e+1
s10ab[fc] (nag[f]_specfun_sinh)
    E_1 = 7.080e+2
```

```
s10ac[fc] (nag[f]_specfun_cosh)
    E_1 = 7.080e+2

s13aa[fc] (nag[f]_specfun_integral_exp)
    x_hi = 7.083e+2
s13ac[fc] (nag[f]_specfun_integral_cos)
    x_hi = 1.0e+16
s13ad[fc] (nag[f]_specfun_integral_sin)
    x_hi = 1.0e+17

s14aa[fc] (nag[f]_specfun_gamma)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.70e+2
    ifail = 2 (NE_REAL_ARG_LT) if x < -1.70e+2
    ifail = 3 (NE_REAL_ARG_TOO_SMALL) if abs(x) < 2.23e-308
s14ab[fc] (nag[f]_specfun_gamma_log_real)
    ifail = 2 (NE_REAL_ARG_GT) if x > x_big = 2.55e+305

s15ad[fc] (nag[f]_specfun_erfc_real)
    x_hi = 2.65e+1
s15ae[fc] (nag[f]_specfun_erf_real)
    x_hi = 2.65e+1
s15ag[fc] (nag[f]_specfun_erfcx_real)
    ifail = 1 (NW_HI) if x >= 2.53e+307
    ifail = 2 (NW_REAL) if 4.74e+7 <= x < 2.53e+307
    ifail = 3 (NW_NEG) if x < -2.66e+1

s17ac[fc] (nag[f]_specfun_bessel_y0_real)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.0e+16
s17ad[fc] (nag[f]_specfun_bessel_y1_real)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.0e+16
    ifail = 3 (NE_REAL_ARG_TOO_SMALL) if 0 < x <= 2.23e-308
s17ae[fc] (nag[f]_specfun_bessel_j0_real)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) > 1.0e+16
s17af[fc] (nag[f]_specfun_bessel_j1_real)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) > 1.0e+16
s17ag[fc] (nag[f]_specfun_airy_ai_real)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.038e+2
    ifail = 2 (NE_REAL_ARG_LT) if x < -5.7e+10
s17ah[fc] (nag[f]_specfun_airy_bi_real)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.041e+2
    ifail = 2 (NE_REAL_ARG_LT) if x < -5.7e+10
s17aj[fc] (nag[f]_specfun_airy_ai_deriv)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.041e+2
    ifail = 2 (NE_REAL_ARG_LT) if x < -1.9e+9
s17ak[fc] (nag[f]_specfun_airy_bi_deriv)
    ifail = 1 (NE_REAL_ARG_GT) if x > 1.041e+2
    ifail = 2 (NE_REAL_ARG_LT) if x < -1.9e+9
s17dc[fc] (nag[f]_specfun_bessel_y_complex)
    ifail = 2 (NE_OVERFLOW_LIKELY) if abs(z) < 3.92223e-305
    ifail = 4 (NW_SOME_PRECISION_LOSS) if abs(z) or fnu+n-1 > 3.27679e+4
    ifail = 5 (NE_TOTAL_PRECISION_LOSS) if abs(z) or fnu+n-1 > 1.07374e+9
s17de[fc] (nag[f]_specfun_bessel_j_complex)
    ifail = 2 (NE_OVERFLOW_LIKELY) if AIMAG(z) > 7.00921e+2
    ifail = 3 (NW_SOME_PRECISION_LOSS) if abs(z) or fnu+n-1 > 3.27679e+4
    ifail = 4 (NE_TOTAL_PRECISION_LOSS) if abs(z) or fnu+n-1 > 1.07374e+9
```

```
s17dg[fc] (nag[f]_specfun_airy_ai_complex)
    ifail = 3 (NW_SOME_PRECISION_LOSS) if abs(z) > 1.02399e+3
    ifail = 4 (NE_TOTAL_PRECISION_LOSS) if abs(z) > 1.04857e+6
s17dh[fc] (nag[f]_specfun_airy_bi_complex)
    ifail = 3 (NW_SOME_PRECISION_LOSS) if abs(z) > 1.02399e+3
    ifail = 4 (NE_TOTAL_PRECISION_LOSS) if abs(z) > 1.04857e+6
s17dl[fc] (nag[f]_specfun_hankel_complex)
    ifail = 2 (NE_OVERFLOW_LIKELY) if abs(z) < 3.92223e-305
    ifail = 4 (NW_SOME_PRECISION_LOSS) if abs(z) or fnu+n-1 > 3.27679e+4
    ifail = 5 (NE_TOTAL_PRECISION_LOSS) if abs(z) or fnu+n-1 > 1.07374e+9

s18ad[fc] (nag[f]_specfun_bessel_k1_real)
    ifail = 2 (NE_REAL_ARG_TOO_SMALL) if 0 < x <= 2.23e-308
s18ae[fc] (nag[f]_specfun_bessel_i0_real)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) > 7.116e+2
s18af[fc] (nag[f]_specfun_bessel_i1_real)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) > 7.116e+2
s18dc[fc] (nag[f]_specfun_bessel_k_complex)
    ifail = 2 (NE_OVERFLOW_LIKELY) if abs(z) < 3.92223e-305
    ifail = 4 (NW_SOME_PRECISION_LOSS) if abs(z) or fnu+n-1 > 3.27679e+4
    ifail = 5 (NE_TOTAL_PRECISION_LOSS) if abs(z) or fnu+n-1 > 1.07374e+9
s18de[fc] (nag[f]_specfun_bessel_i_complex)
    ifail = 2 (NE_OVERFLOW_LIKELY) if REAL(z) > 7.00921e+2
    ifail = 3 (NW_SOME_PRECISION_LOSS) if abs(z) or fnu+n-1 > 3.27679e+4
    ifail = 4 (NE_TOTAL_PRECISION_LOSS) if abs(z) or fnu+n-1 > 1.07374e+9

s19aa[fc] (nag[f]_specfun_kelvin_ber)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) >= 5.04818e+1
s19ab[fc] (nag[f]_specfun_kelvin_bei)
    ifail = 1 (NE_REAL_ARG_GT) if abs(x) >= 5.04818e+1
s19ac[fc] (nag[f]_specfun_kelvin_ker)
    ifail = 1 (NE_REAL_ARG_GT) if x > 9.9726e+2
s19ad[fc] (nag[f]_specfun_kelvin_kei)
    ifail = 1 (NE_REAL_ARG_GT) if x > 9.9726e+2

s21bc[fc] (nag[f]_specfun_ellipint_symm_2)
    ifail = 3 (NE_REAL_ARG_LT) if an argument < 1.583e-205
    ifail = 4 (NE_REAL_ARG_GE) if an argument >= 3.765e+202
s21bd[fc] (nag[f]_specfun_ellipint_symm_3)
    ifail = 3 (NE_REAL_ARG_LT) if an argument < 2.813e-103
    ifail = 4 (NE_REAL_ARG_GT) if an argument >= 1.407e+102
```

(e) X01

数学定数の値は以下の通りです：

```
x01aa[fc] (nag[f]_math_pi)
     = 3.1415926535897932
x01ab[fc] (nag[f]_math_euler)
     = 0.5772156649015328
```

(f) X02

マシン定数の値は以下の通りです：

モデルの基本パラメータ

```
x02bh[fc] (nag[f]_machine_model_base)
     = 2
x02bj[fc] (nag[f]_machine_model_digits)
     = 53
x02bk[fc] (nag[f]_machine_model_minexp)
     = -1021
x02bl[fc] (nag[f]_machine_model_maxexp)
     = 1024
```

浮動小数点演算の派生パラメータ

```
x02aj[fc] (nag[f]_machine_precision)
     = 1.11022302462516e-16
x02ak[fc] (nag[f]_machine_real_smallest)
     = 2.22507385850721e-308
x02al[fc] (nag[f]_machine_real_largest)
     = 1.79769313486231e+308
x02am[fc] (nag[f]_machine_real_safe)
     = 2.22507385850721e-308
x02an[fc] (nag[f]_machine_complex_safe)
     = 2.22507385850721e-308
```

コンピューティング環境の他の側面のパラメータ

```
x02ah[fc] (nag[f]_machine_sinarg_max)
     = 1.42724769270596e+45
x02bb[fc] (nag[f]_machine_integer_max)
     =          2147483647 (32 ビット整数ライブラリの場合)<br>     = 9223372036854775807 (64 ビット整数ライブラリの場合)
x02be[fc] (nag[f]_machine_decimal_digits)
     = 15
```

(g) X04

Fortran ルーチン： 明示的な出力を生成できるルーチンのエラーおよびアドバイザリメッセージのデフォルト出力ユニットは、どちらも Fortran ユニット 6 です。

(h) X06

X06 章のルーチンは、このライブラリ実装で MKL スレッド処理の動作も変更します。

5 ドキュメント

ライブラリマニュアルは、NAG ウェブサイトの NAG Library Manual, Mark 31.1 でアクセスできます。

ライブラリマニュアルは、HTML と MathML を使用した完全にリンクされたバージョンのマニュアルである HTML5 で提供されています。これらのドキュメントは、ウェブブラウザを使用してアクセスできます。

ドキュメントの表示とナビゲーションに関するアドバイスは、Guide to the NAG Library Documentation で見つけることができます。

さらに、以下が提供されています：

- `in.html` – インストールノート
- `un.html` – ユーザーノート（このドキュメント）
- `alt_c_interfaces.html` – C および C++から NAG ライブラリの Fortran ルーチンを呼び出す方法に関するアドバイス

6 サポート

製品のご利用に関してご質問等がございましたら、電子メールにて「日本 NAG ヘルプデスク」までお問い合わせください。その際、ご利用の製品の製品コード（NSL6I311BL）並びに、お客様の User ID をご明記いただきますようお願い致します。

ご返答は平日 9：30～12:00、13:00～17:30 に行わせていただきます。

日本 NAG ヘルプデスク
Email: naghelp@nag-j.co.jp

7 コンタクト情報

日本ニューメリカルアルゴリズムズグループ株式会社（日本 NAG）
〒104-0032
東京都中央区八丁堀 4-9-9 八丁堀フロンティアビル 2F
Email: sales@nag-j.co.jp
Tel: 03-5542-6311
Fax: 03-5542-6312

NAG のウェブサイトでは製品およびサービスに関する情報を定期的に更新しています。

https://www.nag-j.co.jp/ （日本）
https://nag.com/ （英国本社）