ユーザズ・マニュアル
EigenExa
Version 2.3c

EigenExa 開発グループ
大規模並列数値計算技術研究チーム
理化学研究所
計算科学研究機構

2015 年 6 月 24 日

目 次

第1章　はじめに

1.1　EigenExa とその開発経緯

EigenExa は高性能固有値ソルバである．EigenExa の歴史は, 2002 年に世界一位を記録した地球シミュレータ上で開発された EigenES(正式名ではなくコードネーム)[1] まで溯る．EigenES の成果は SC2006 でのゴードンベル賞にノミネートされ, その後大規模な PC クラスタ上での固有値ソルバーとして継続されている．EigenExa の直前のライブラリである EigenK[3, 4] の開発が 2008 年頃に開始され, 京 [5, 6] の運用開始後の 2013 年 8 月に名称を EigenExa に変更し一般公開が開始された．EigenExa は将来登場するであろうポストペタスケール計算機システム (所謂「エクサ」または「エクストリーム」システム) でスケーラブルに動作する固有値ライブラリ実現を目標に開発が継続されている．現リリース（バージョン 2.3c) では, 標準固有値問題と一般化固有値問題のいずれに対しても全ての固有対 (「固有値」と「対応する固有ベクトル」の組) を計算するという最もシンプルな機能を提供する．文献 [2, 3, 4] で報告されているように, EigenK がそうであるのと同様 EigenExa もまた古典的なアルゴリズムと先進的なアルゴリズムの両者を採用して対角化に要する計算時間を削減している．

EigenExa は, MPI, OpenMP, 高性能 BLAS, 更に SIMD ベクトル化 Fortran90 コンパイラ技術など様々な並列プログラミング言語とライブラリを用いて開発されている．次に挙げる項目が複数同時に機能して, 高性能計算を実現することが期待される．

1. MPI による分散メモリ型のノード間並列性
2. OpenMP による共有メモリ型並列計算機もしくはマルチコアプロセッサでの並列性
3. ベンダにより高度に最適化された BLAS を用いた高い並列性
4. ベンダ提供の高性能コンパイラを用いた SIMD もしくは疎粒度並列性

EigenExa には Fortran90 のよい特徴も積極的に取り込まれている．EigenExa の API は Fortran77 による実装ライブラリよりも柔軟であり, モジュールインターフェイスや省略可能引数によりユーザーフレンドリーなインターフェイスが提供される．データ分散は 2 次元サイクリック分割に限定されるが, プロセッサマップはほぼ任意形状が指定でき, ScaLAPACK が提供するデータ再分散関数を利用すれば既存の数値計算ライブラリとの親和性・整合性も保証されている．更に, 実行性能を左右するブロックパラメータを利用者側から指定できる (省略も可能) など, 性能向上のためのインターフェイスも提供している．

ライブラリ単体の並列性能の観点から見てみると, EigenK の通信オーバヘッドを削減することで高い性能向上を達しており, 多くの場合に EigenExa は EigenK や ScaLAPACK などの最高水準の数値計算ライブラリより高性能であることが確認されている [4].

現在, EigenExa は「京」コンピュータをはじめとして, その商用機である Fujitsu PRIMEHPC FX10, Intel x86 系プロセッサを搭載する各種クラスタシステム, IBM Blue/Gene Q システム, NEC のベクトルコンピュータ SX シリーズなど, 多くの HPC プラットフォームで動作する. また, EigenExa に関して学会等で報告している [7, 8, 9, 10, 11] ので, 必要に応じて参照されたい.

本ドキュメントは EigenExa version 2.3c のユーザズ・マニュアルである. 導入開始から実際の使用までの内容を記している. 特に, 導入, コンパイル, クイックチュートリアル, API リスト, EigenK との互換性の注意が選択されている. EigenExa チームの全ての開発者は, 本ドキュメントが多くの利用者に対して並列シミュレーションを効率よく走らせるための手助けになることを期待している.

1.2 利用許諾/Copyright

EigenExa は 2 条項 BSD ライセンス (The BSD 2-Clause License) に基づき利用を許諾する (ライブラリ内の LICENCE.txt に記載).

LICENCE.txt

```
Copyright (C) 2012- 2014 RIKEN.
Copyright (C) 2011- 2012 Toshiyuki Imamura
 Graduate School of Informatics and Engineering,
 The University of Electro-Communications.
Copyright (C) 2011- 2014 Japan Atomic Energy Agency.

-----------------------------------------------------------------------
 Copyright notice is from here
-----------------------------------------------------------------------
  Redistribution  and  use  in  source and binary forms, with or without
  modification,  are  permitted  provided  that the following conditions
  are met:

  * Redistributions  of  source  code  must  retain  the above copyright
    notice,  this  list  of  conditions  and  the  following  disclaimer.
  * Redistributions  in  binary  form must reproduce the above copyright
    notice,  this list of conditions and the following disclaimer in the
    documentation  and/or other materials provided with the distribution.

  THIS  SOFTWARE  IS  PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  ``AS IS''  AND  ANY  EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A  PARTICULAR  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL,  EXEMPLARY,  OR  CONSEQUENTIAL  DAMAGES  (INCLUDING,  BUT NOT
  LIMITED  TO,  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA,  OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY  OF  LIABILITY,  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF  THIS  SOFTWARE,  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
-----------------------------------------------------------------------
```

第2章　利用の前に

2.1　EigenExa のインストールのために必要なソフトウェア

EigenExa ライブラリをコンパイルするためには幾つかのソフトウェアパッケージを準備しなければならない．BLAS, LAPACK, ScaLAPACK 更に MPI は EigenExa のコンパイルの前にシステムにインストールされていなくてはならない．現在のところ, EigenExa は以下に示すライブラリでコンパイルできることが確認されている．

BLAS	Intel MKL, GotoBLAS, OpenBLAS, ATLAS
	Fujitsu SSL II, IBM ESSL, NEC MathKeisan
LAPACK	Version 3.4.0 以降
ScaLAPACK	Version 1.8.0 以降
MPI	MPICH2 version 1.5 以降, MPICH version 3.0.2 以降
	OpenMPI version 1.6.4 以降，MPI/SX

2.2　EigenExa の入手方法

EigenExa に関する全ての情報は次の URL から入手可能である．:

```
http://www.aics.riken.jp/labs/lpnctrt/EigenExa.html
```

tarball の配布も上記サイトからされる．EigenExa に関するその他の情報も提供されていく予定である．

2.3　コンパイルとインストール手順

EigenExa ライブラリのコンパイルには幾つかの手順が必要である．次のインストレーション手引きに従ってほしい．

解凍と展開　まず, tarball をワーキングディレクトリ上で展開する．そして, ディレクトリ EigenExa-2.3c に移動する．

```
% tar zxvf EigenExa-2.3c.tgz
% cd EigenExa-2.3c
```

環境設定 次に, Makefile と make_inc.xxx を使用者の環境に合わせて編集する ここで, xxx には

1. BX900
2. Intel もしくは Intel.shared
3. K_FX10 もしくは K_FX10.shared
4. gcc
5. BlueGeneQ
6. SX

の中から使用するコンパイラに応じた文字列が入る. なお, ".shared" が付いたものはシェアード・ライブラリを作成するときに指定する.

メイク 3 番目に, make を実行する. その結果スタティック・ライブラリ libEigenExa.a もしくはシェアードライブラリ libEigenExa.so が作成される.

```
% make
```

インストール 最後に, ライブラリ自身である libEigenExa.a (シェアードライブラリの場合は libEigenExa.so) と eigen_libs.mod と eigen_blacs.mod をインストールディレクトリにコピーし終了である. 例えば /usr/local/lib にインストールする場合は次のようにする.

```
% cp libEigenExa.a eigen_libs.mod eigen_blacs.mod  /usr/local/lib/
```

一般化固有値計算ドライバルーチン インストールしたバージョン番号と同じ一般化固有値ドライバルーチン KMATH_EIGEN_GEV を

http://www.aics.riken.jp/labs/lpnctrt/KMATH_EIGEN_GEV.html

からダウンロードし, make すると一般化固有値用のドライバモジュール KMATH_EIGEN_GEV.o が作成される. プログラムリンク時に本オブジェクトをリンクすると一般化固有値計算が可能となる. なお, 本機能は現時点では EigenExa を固有値計算エンジンとして使用する上位の独立したモジュールである. 本ドキュメントでは EigenExa に関連する機能として説明する.

第3章　クイックチュートリアル

ワーキングディレクトリに移り, 'make benchmark' を実行すると標準ベンチマークコードが得られる. ソースコード中の 'main2.F' と 'Makefile' はコード作成に役立つはずである.

main2.f の核部分を取り出したものが次のようになる.

main2.f

```
use MPI
use eigen_libs
...
call MPI_Init_thread( MPI_THREAD_MULTIPLE, i, ierr )
call eigen_init( )

N=10000; mtype=0

call eigen_get_matdims( N, nm, ny )
allocate ( A(nm,ny), Z(nm,ny), w(N) )
call mat_set( N, a, nm, mtype )
call eigen_sx( N, N, a, nm, w, z, nm, m_forward=32, m_backward=128)
deallocate ( A, Z, w )
...
call eigen_free( )
call MPI_Finalize( ierr )
end
```

上のコードは骨格部分を示したのみであり実際の動作はしないが, 「初期化」→「配列確保」→「固有値計算」→「終了手続き」の流れを示すには十分なものである.

上例では初期化関数 eigen_init() を引数省略型呼び出しで実行している. eigen_init() には固有値計算を実施するグループをコミュニケータとして comm=XXX の形で指定できる. 複数のグループで同時に固有値計算を並列実行したいときには MPI_Comm_split() 等で作成されたコミュニケータを渡すことで並列計算可能となる. ただし, 現実装では eigen_init() の内部でコミュニケータ MPI_COMM_WORLD に対する集団操作を行うため, eigen_init() は MPI_COMM_WORLD に属する全プロセスに同時に呼び出されなくてはならないという制約がある. そこで, 個々のプロセスごとに異なるコミュニケータを指定できるので, 固有値計算に参加しないプロセスは MPI_COMM_NULL を eigen_init() に指定して, 固有値計算ドライバ eigen_sx() 自身の呼び出しをスキップさせることができる. つまり, eigen_sx() の処理以外に eigen_sx() を含む様々な演算を同時実行することができる.

MPI_Comm_split と MPI_COMM_NULL

```
if ( my_rank < 10 ) then
   comm = MPI_COMM_SELF
else
   comm = MPI_COMM_NULL
endif
call eigen_init( comm )
call eigen_sx( .... )
```

EigenExa では eigen_init() で指定されたコミュニケータに属するプロセスを 2 次元プロセスグリッドに配置して使用する．できるだけ通信量が削減できるように正方に近い形のプロセスグリッドを採用するよう設計されている．また, EigenExa は利用者の便宜を高めるという観点から, MPI で採用されている 2 次元カーテシアンを comm に指定することができるように開発されている．原理的にはカーテシアンの形状が 2 次元であれば任意のプロセス配置に対して EigenExa を呼び出して計算することができるので, 上述の複数種類のコミュニケータとの組み合わせにより複雑な並列処理を実行することができる．なお, カーテシアンは基本的にプロセスグリッドが Row-major になるため, order='C' 指定と矛盾するときはカーテシアンを優先して扱うことになっている．なお, EigenExa は歴史的経緯からデフォルトのプロセスグリッドは Column-Major を採用している．

eigen_sx() の呼び出しの直前に呼び出している mat_set() において行列データの生成を行っている．行列データは指定された 2 次元プロセスグリッド上に 2 次元サイクリック分割のスタイルで分散されており, ローカル配列として各プロセスに格納されている．各プロセスは行列の一部のデータのみを格納するため, 行列要素のアクセスをする場合にはグローバルインデックスとローカルインデックス間の変換ルールが必要である．

次のプログラムは mat_set() からの抜粋であり, グローバルカウンタループ構成の Frank 行列生成プログラムをローカルカウンタループに変換した両者を対比として示している．

matset(before)

```
! Global loop program to compute a Frank matrix
do i = 1, n
   do j = 1, n
      a(j, i) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

↓↓↓↓↓↓

matset(after)

```
! Translated local loop program to compute a Frank matrix
use MPI
use eigen_libs
call eigen_get_procs( nnod, x_nnod, y_nnod )
call eigen_get_id   ( inod, x_inod, y_inod )

j_2 = eigen_loop_start( 1, x_nnod, x_inod )
j_3 = eigen_loop_end  ( n, x_nnod, x_inod )
i_2 = eigen_loop_start( 1, y_nnod, y_inod )
i_3 = eigen_loop_end  ( n, y_nnod, y_inod )
do i_1 = i_2, i_3
   i = eigen_translate_l2g( i_1, y_nnod, y_inod )
   do j_1 = j_2, j_3
      j = eigen_translate_l2g( j_1, x_nnod, x_inod )
      a(j_1, i_1) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

ループ範囲の変換は `eigen_loop_start()` もしくは `eigen_loop_end()` を使用する. 第二, 第三引数は分散方向を示すコミュニケータから派生されるプロセス数とプロセス ID を指定する. 本ドキュメントでは常に「行」→”x”, 「列」→”y”の対応になっている (参加プロセス全体のコミュニケータの場合は, ”x”や”y”の部分を無文字にしている). ここで, 重大な注意として「EigenExa ではプロセス ID を 1 から始まる整数で管理している」. そのため, 問合せ関数 `eigen_get_id( )` によって取得したプロセス ID は MPI のランクと 1 だけずれている. MPI のランクが必要な場合はプロセス ID から 1 減じる必要がある.

上記プログラムではローカルなループカウンタ値から対応するグローバルカウンタ値に変換して使用する. その変換には `eigen_translate_l2g()` を使用する. 第二, 第三引数は `eigen_loop_start()` などと同様に指定するとよい. 逆に, グローバルカウンタ値をローカルカウンタ値に変換するには `eigen_translate_g2l()` を使用する. ただし, `eigen_translate_g2l()` はグローバルカウンタ値をループカウンタとして見たときに, オーナープロセス (そのグローバルカウンタ値に対応するローカルカウンタ値をループ内に含むプロセス) となるプロセス上で対応するローカルカウンタ値を返すこととする. 指定したグローバルループカウンタのオーナープロセスを知るには `eigen_owner_node()` を使用する. 特定の行ベクトルや列ベクトルの値を参照したりブロードキャストする際に利用するとよい.

更に, ScaLAPACK と連携した上級者向けの計算を進めたいときは, 補助関数 `eigen_get_blacs_context()` により EigenExa で使用するプロセスグリッドコンテキストを取得すればよい. `mat_set()` 関数の `mtype=2` の部分を参照するとよい (次例は, 行列 AS の転置を行列 A に格納する `PDTRAN()` 呼び出しの核部分を取り出したものである).

pdtran

```
! Cooperation with ScaLAPACK
NPROW = x_nnod; NPCOL = y_nnod

ICTXT = eigen_get_blacs_context( )
CALL DESCINIT( DESCA, n, n, 1, 1, 0, 0, ICTXT, nm, INFO )

! A ← AS^T
CALL PDTRAN( n, n, 1D0, as, 1, 1, DESCA, 1D0, a, 1, 1, DESCA )
```

コンパイル時には mpif90 を使用するとともに, `eigen_libs` などモヂュールへのアクセスが必要となるためパスの設定をしておく必要がある（多くの場合は`-I` オプションである）. また, EigenExa ライブラリをリンクするには, MPI, OpenMP, ScaLAPACK(バージョンが 1.8 以前の場合は BLACS も) などを同時にリンクする必要がある. Intel コンパイラベースの MPI の場合は以下のようにする (ScaLAPACK や BLAS まわりライブラリ名は環境によって異なる).

```
% mpif90 -c a.f -openmp -I/usr/local/include -I/usr/local/lib
% mpif90 -o exe a.o -openmp -L/usr/local/lib -lEigenExa -lscalapack \
  -llapack -lblas
```

第4章　API

本節では‘eigen_libs.mod’中でpublic属性を与えられた関数をリストアップする．始めの三ルーチンはメインドライバーであり, その他はユーティリティ関数である．Optional属性のついた引数 (斜体で記述している) の場合は省略もできるし, Fortranのフォーマット形式であるTERM=variable *or* constant valueでも指定することができる．

4.1　eigen_init

EigenExaの機能を初期化する．プロセスグリッドマッピングを引数‘comm’や‘order’を通じて指定することができる．本手続きは集団的である (現バージョンでは内部でMPI_COMM_WORLDに対する集団操作を行う故に, 全プロセスが本手続きを呼ばなくてはない), なお, commは各プロセスグループごとに異なる値を指定でき, 異なるプロセスグループが同時にドライバ関数 (eigen_sx()もしくはeigen_s()) を呼び出した際には, ドライバ関数単位で並列動作する．commがMPI_COMM_NULLの場合, ハンドラeigen_sx()やeigen_s()が呼ばれた際には内部で何も行わず直ちにリターンする．インターコミュニケータは使用できない．

```
subroutine eigen_init( comm, order )
```

1. `integer, optional, intent(IN) :: comm = MPI_COMM_WORLD`
   基盤となるコミュニケータ.
   commが2次元カーテシアンの場合はプロセスグリッドマッピングが有効になる．
   注意：省略時はMPI_COMM_WORLD
2. `character*(*), optional, intent(IN) :: order = 'C'`
   Rowもしくは Column
   注意：省略時は‘C’として扱われる．グリッドメジャーがカーテシンアンcommの指定と矛盾するときは‘R’が採用される．

4.2　eigen_free

EigenExaの機能を終了する．

```
subsroutine eigen_free( flag )
```

1. `integer, optional, intent(IN) :: flag = 0`
   タイマープリンタのフラグ
   この引数は開発用途のためのものであり通常は指定しない．省略時は0である．

4.3 eigen_get_blacs_context

EigenExa で規定されるプロセスグリッド情報に対応する ScaLAPACK(BLACS) のコンテキストを返す.

```
integer function eigen_get_blacs_context( )
```

4.4 eigen_sx

EigenExa の主たるドライバルーチンである. 五重対角行列への変換を経て固有対を計算する. 本ドライバは集団操作であり, 呼び出しを行うプロセスグループに属する全てのプロセスが呼び出しに参加しなくてはならない.

```
subroutine eigen_sx( n, nvec, a, lda, w, z, ldz, \
                     m_forward, m_backward, mode )
```

1. `integer, intent(IN) :: n`
   行列・ベクトルの次元
2. `integer, intent(IN) :: nvec`
   計算する固有ベクトルの本数
   現在このオプションはサポートされていない.
   `eigen_sx()` は全固有ベクトルを計算する.
3. `real(8), intent(INOUT) :: a(lda,*)`
   対角化される対称行列
   サブルーチン終了時には配列の内容は破壊される.
   ただし, `a(1, 1)` には FLOPS カウントが格納される.
4. `integer, intent(IN) :: lda`
   配列 `a` の整合寸法（リーディングディメンジョン）
5. `real(8), intent(OUT) :: w(n)`
   昇順の固有値
6. `real(8), intent(OUT) :: z(ldz,*)`
   行列 `a` の直交固有ベクトル
7. `integer, intent(IN) :: ldz`
   配列 `z` の整合寸法（リーディングディメンジョン）
8. `integer, optional, intent(IN) :: m_forward = 48`
   ハウスホルダー変換のブロック係数 (偶数でなければいけない) 省略時は 48.
9. `integer, optional, intent(IN) :: m_backward = 128`
   ハウスホルダー逆変換のブロック係数. 省略時は 128.
10. `character, optional, intent(IN) :: mode = 'A'`
    'A' : 全ての固有値と対応する固有ベクトル (default)
    'N' : 固有値のみ
    'X' : モード A に加えて固有値の精度改善を行う

4.5 eigen_s

EigenExa のドライバルーチンである.

```
subroutine eigen_s( n, nvec, a, lda, w, z, ldz, \
                    m_forward, m_backward, mode )
```

1. `integer, intent(IN) :: n`
   行列・ベクトルの次元
2. `integer, intent(IN) :: nvec`
   計算する固有ベクトルの本数
   現在このオプションはサポートされていない.
   `eigen_s()` は全固有ベクトルを計算する.
3. `real(8), intent(INOUT) :: a(lda,*)`
   対角化される対称行列
   サブルーチン終了時には配列の内容は破壊される.
   ただし，`a(1, 1)` には FLOPS カウントが格納される.
4. `integer, intent(IN) :: lda`
   配列 `a` の整合寸法（リーディングディメンジョン）
5. `real(8), intent(OUT) :: w(n)`
   昇順の固有値
6. `real(8), intent(OUT) :: z(ldz,*)`
   行列 `a` の直交固有ベクトル
7. `integer, intent(IN) :: ldz`
   配列 `z` の整合寸法（リーディングディメンジョン）
8. `integer, optional, intent(IN) :: m_forward = 48`
   ハウスホルダー変換のブロック係数
9. `integer, optional, intent(IN) :: m_backward = 128`
   ハウスホルダー逆変換のブロック係数
10. `character, optional, intent(IN) :: mode = 'A'`
    'A' : 全ての固有値と対応する固有ベクトル (default)
    'N' : 固有値のみ
    'X' : モード A に加えて固有値の精度改善を行う

4.6 eigen_get_version

EigenExa のバージョン情報を返す.

```
subroutine eigen_get_version( version, data, vcode )
```

1. `integer, intent(OUT) :: version`
   3 桁のバージョン番号.
   各桁は上位から major version，minor version，patch level を表す.
2. `character, intent(OUT) :: date`
   リリース日.
3. `character, intent(OUT) :: vcode`
   各バージョンに対応するコードネーム.

4.7 eigen_show_version

EigenExa のバージョン情報を標準出力する.

```
subroutine eigen_show_version()
```

4.8 eigen_get_matdims

EigenExa で推奨する配列サイズを返す．ユーザーは本関数で取得した配列寸法 (nx,ny) もしくはそれ以上の数値を使用してローカルな配列を動的に確保することが望ましい．行列全体は (CYCLIC,CYCLIC) 分割されている.

```
subroutine eigen_get_matdims( n, nx, ny )
```

1. `integer, intent(IN) :: n`
   行列の次元
2. `integer, intent(OUT) :: nx`
   配列 `a` ならびに `z` の整合寸法（リーディングディメンジョン）の下限値
3. `integer, intent(OUT) :: ny`
   配列 `a` ならびに `z` の第二インデックスの下限値

4.9 eigen_memory_internal

本関数は EigenExa が呼び出されている間に内部で動的に確保されるメモリサイズを返す．利用者は本関数の返却値を知り, メモリ不足に陥らないようにすべきである．バージョン 2.3c より 8 バイト整数（`integer(8)`）を返り値とする仕様変更がなされた．返り値が -1（負値）のときは行列サイズが大きすぎて EigenExa 内部で使用する整数値がオーバフローするおそれがあることを警告しており，注意が必要である.

```
integer(8) function eigen_memory_internal( n, lda, ldz, m1, m0 )
```

1. `integer, intent(IN) :: n`
   行列の次元
2. `integer, intent(IN) :: lda`
   配列 a の整合寸法（リーディングディメンジョン)
3. `integer, intent(IN) :: ldz`
   配列 z の整合寸法（リーディングディメンジョン)
4. `integer, intent(IN) :: m1`
   ハウスホルダー変換のブロック係数 (偶数でなければいけない)
5. `integer, intent(IN) :: m0`
   ハウスホルダー逆変換のブロック係数

4.10 eigen_get_comm

`eigen_init()` によって生成された MPI コミュニケータを返す.

```
subroutine eigen_get_comm( comm, x_comm, y_comm )
```

1. `integer, intent(OUT) :: comm`
   基盤となるコミュニケータ
2. `integer, intent(OUT) :: x_comm`
   行コミュニケータ. 行 id が一致する全プロセス所属する.
3. `integer, intent(OUT) :: y_comm`
   列コミュニケータ. 列 id が一致する全プロセスが所属する.

4.11 eigen_get_procs

`eigen_init()` によって生成されたコミュニケータに関するプロセス数情報を返す.

```
subroutine eigen_get_procs( procs, x_procs, y_procs )
```

1. `integer, intent(OUT) :: procs`
   comm 中のプロセス数
2. `integer, intent(OUT) :: x_procs`
   x_comm 中のプロセス数
3. `integer, intent(OUT) :: y_procs`
   y_comm 中のプロセス数

4.12 eigen_get_id

`eigen_init()` によって生成されたコミュニケータに関するプロセス ID 情報を返す. ここでプロセス ID は MPI ランクとは異なり 1 から開始する整数値で, MPI ランク=プロセス ID − 1 の関係にある.

```
subroutine eigen_get_id( id, x_id, y_id )
```

1. `integer, intent(OUT) :: id`
   comm で定義されたプロセス ID
2. `integer, intent(OUT) :: x_id`
   x_comm で定義されたプロセス ID
3. `integer, intent(OUT) :: y_id`
   y_comm で定義されたプロセス ID

4.13 eigen_loop_start

指定されたグローバルループ開始値に対応するローカルなループ構造におけるループ開始値を返す.

```
integer function eigen_loop_start( istart, nnod, inod )
```

1. `integer, intent(IN) :: istart`
   グローバルループ開始値
2. `integer, intent(IN) :: nnod`
   プロセス数
3. `integer, intent(IN) :: inod`
   プロセス ID

4.14 eigen_loop_end

指定されたグローバルループ終端値に対応するローカルなループ構造におけるループ終端値を返す.

```
integer function eigen_loop_end( iend, nnod, inod )
```

1. `integer, intent(IN) :: istart`
   グローバルループ終端値
2. `integer, intent(IN) :: nnod`
   プロセス数
3. `integer, intent(IN) :: inod`
   プロセス ID

4.15 eigen_translate_l2g

```
integer function eigen_translate_l2g( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   ローカルカウンタ
2. `integer, intent(IN) :: nnod`
   プロセス数
3. `integer, intent(IN):: inod`
   プロセス ID

4.16 eigen_translate_g2l

```
integer function eigen_translate_g2l( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   グローバルカウンタ
2. `integer, intent(IN) :: nnod`
   プロセス数
3. `integer, intent(IN) :: inod`
   プロセス ID

4.17 eigen_owner_node

指定されたグローバルループカウンタ値に対応するオーナープロセスの ID を返す.

```
integer function eigen_owner_node( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   グローバルループカウンタ
2. `integer, intent(IN) :: nnod`
   プロセス数
3. `integer, intent(IN) :: inod`
   プロセス ID

4.18 KMATH_EIGEN_GEV

EigenExa を固有値計算エンジンとして利用する一般化固有計算ドライバルーチンである. 本ドライバの内部で五重対角行列への変換を経て固有対を計算する `eigen_sx` を呼び出す. 本ドライバは `eigen_sx` と同様の制約を受ける. 本ドライバールーチンを使用する際は EigenExa 本体と共に `KMATH_EIGEN_GEV.o` をリンクしなくてはいけない.

`subroutine kmath_eigen_gev( n, a, lda, b, ldb, w, z, ldz )`

1. `integer, intent(IN) :: n`
   行列・ベクトルの次元
2. `real(8), intent(INOUT) :: a(lda,*)`
   計算対象のペンシル $(A - \lambda B)$ の行列 A
   サブルーチン終了時には配列の内容は破壊される
3. `integer, intent(IN) :: lda`
   配列 `a` の整合寸法（リーディングディメンジョン）
4. `real(8), intent(INOUT) :: b(ldb,*)`
   計算対象のペンシル $(A - \lambda B)$ の行列 B
   サブルーチン終了時には標準固有値問題への変換行列が格納される.
5. `integer, intent(IN) :: ldb`
   配列 `b` の整合寸法（リーディングディメンジョン）
6. `real(8), intent(OUT) :: w(n)`
   昇順の固有値
7. `real(8), intent(OUT) :: z(ldz,*)`
   一般化固有値問題 の B 直交固有ベクトル
8. `integer, intent(IN) :: ldz`
   配列 `z` の整合寸法（リーディングディメンジョン）

第5章　その他の注意事項

5.1　互換性の注意

EigenExa は EigenK の後継で多くの機能を継承している．しかしながら両者の間には完全な互換性を保証していない．それは内部実装の詳細が異なることに起因しており, 主に関数や変数の命名則, コモン領域の管理方法の違いによるものである．また, 同様の理由により EigenExa と EigenK を同時にリンクすることを勧めない.

5.2　他の言語との結合

Fortran90 以外からの EigenExa の呼び出し方法は利用者の環境に大きく依存する．コンパイラマニュアルを引き,「言語結合 (language bindings)」や「複数プログラミング言語とのリンク方法」を参照してほしい．なお, Python 言語からの呼び出しを可能とする”Python binding of EigenExa” プロジェクト [12] も存在するので, そちらも参考にしてほしい.

5.3　エラー発生時の振舞い

EigenExa は初期化時に適切な条件のもとで実行されているかのチェックを行うが, 実行時にはエラー検出は行っていない．リンクした BLAS や LAPACK など下位のライブラリがエラーを吐いてライブラリ強制終了される場合がある．バグ発見の情報はライブラリの品質向上には欠かすことができないものである．バグ発見時には, ライブラリ公開 HP に書かれている開発者のメイルアドレスまで一報してほしい.

5.4　バージョン 1.x におけるシェアード・ライブラリの扱い

旧バージョン 1.x ではシェアード・ライブラリをサポートしない．バージョン 1.x 開発当時, シェアード・ライブラリ使用時に関数名の解決が衝突なく完全にできることを保証できなかったからである（gcc のあるバージョンでは実行時に関数名の解決ができず異常終了する場合があった）．バージョン 1.x をシェアード・ライブラリとして利用する場合はあくまでも利用者の責任の元実行して欲しい．なお, EigenExa のバージョン 2.x からは, 理化学研究所計算科学研究機構高度利用化チームの前田俊行チームリーダらの技術協力を受けシェアード・ライブラリ化が実現している．ライブラリビルドの説明にもあったように選択する make.inc を適切に選び, 実行時には適切な環境変数 (`LD_LIBRARY_PATH` など) の設定を**忘れずに**行えばよい.

付 録A　アルゴリズムの概要

A.1　はじめに

本章では，EigenExa で採用されている固有値計算アルゴリズムの概要を述べる．EigenExa では，実対称密行列の全ての固有値と固有ベクトルを計算する場合を想定しており，これを実現する二つのドライバルーチン（eigen_s と eigen_sx）を提供する．本章では，主にこの二つのルーチンの違いに主眼を置いて，両者で採用されているアルゴリズムを概観する．なお，密行列向け固有値計算アルゴリズムの一般的な内容は [13, 14, 15, 16, 17, 18, 19, 20] などを参照されたい．

A.2　様々なアプローチと関連プロジェクト

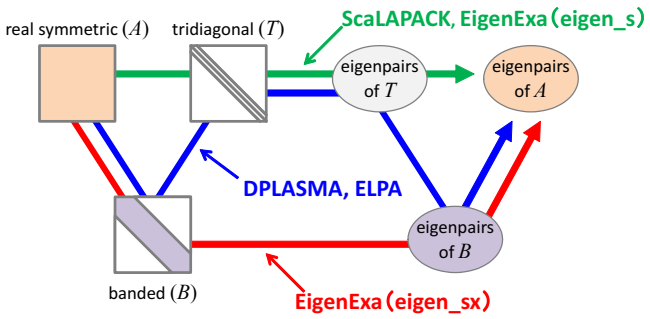


図 A.1: 実対称密行列向け固有値計算の様々なアプローチ.

最初に，実対称密行列向け固有値計算の手順については簡単に紹介する．一般的な行列計算の教科書に記載されているのは，入力行列の三重対角化に基づくアプローチ（図 A.1 の緑色のパス）であり，ScaLAPACK[21]（や LAPACK）で採用されている．しかし，このアプローチでは，最初のステップ（三重対角化）がメモリバンド幅に律速され，近年の計算機システム上では高い性能が期待できないことが問題となっている．

この問題を踏まえて，ELPA[22] や DPLASMA[23] と呼ばれるプロジェクトでは，帯行列を経由する二段階の三重対角化に基づくアプローチ（図 A.1 の青色のパス）が採用されている．この二段階の三重対角化における主要コストは一段階目の帯行列化の部分であり，その部分の要求 byte/flop 値が直接三重対角化する場合よりも小さくなるため，実効性能の向上が期待できる．ただし，固有ベクトルの逆変換も二段階必要（こちらはコストが単純に二倍）になり，その一段階目（T から B

への逆変換）の高性能実装が困難となっている．そのため，求める固有ベクトルの本数が多くなると逆変換のコストが膨大となってしまう問題を抱えている．

この状況を踏まえて，EigeExa では従来の（一段階の）三重対角化に基づくアプローチ（図 A.1 の緑色のパス）を採用したルーチン（eigen_s）と，帯行列の固有値・固有ベクトルを直接計算するアプローチ（図 A.1 の赤色のパス）を採用したルーチン（eigen_sx）の二つを開発・提供している．これら二つのアプローチについては，次節以降でもう少し詳しく述べる．

A.3　eigen_s

EigenExa で提供されるドライバルーチンの一つである eigen_s は，上述のように ScaLAPACK 等で採用されている従来の（一段階の）三重対角化に基づくアプローチを採用している．具体的には，以下の三つのステップにより，固有値問題 $A\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i$ $(i = 1, \ldots, N)$ を解く．

1. ハウスホルダー変換による入力行列の三重対角化：$Q^\top AQ \to T$
2. 分割統治法による三重対角行列の固有値・固有ベクトルの計算：$T\boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i$
3. 固有ベクトルの逆変換：$Q\boldsymbol{y}_i \to \boldsymbol{x}_i$

最初のステップでは，ハウスホルダー変換を両側から

$$H_{N-2}^\top \cdots H_1^\top A H_1 \cdots H_{N-2} \to T, \quad H_i = I - \boldsymbol{u}_i \beta_i \boldsymbol{u}_i^\top \tag{A.1}$$

と作用させて，入力行列を一列（一行）ごとに三重対角行列に変形していく（図 A.2(a)）．なお，β はスカラーであるが，後述の内容との対応を明確にするために，式中の位置を配慮している．詳細は割愛するが，一つのハウスホルダー変換による変形の計算は，

$$(I - \boldsymbol{u}\beta\boldsymbol{u}^\top)^\top A(I - \boldsymbol{u}\beta\boldsymbol{u}^\top) = A - \boldsymbol{u}\boldsymbol{v}^\top - \boldsymbol{v}\boldsymbol{u}^\top, \quad \boldsymbol{v} = (\boldsymbol{w} - \frac{1}{2}\boldsymbol{u}\beta^\top(\boldsymbol{w}^\top\boldsymbol{u}))\beta, \quad \boldsymbol{w} = A\boldsymbol{u} \tag{A.2}$$

として A の対称性を利用して行われることが一般的となっている．また，Dongarra の手法を用いることで，複数個のハウスホルダー変換による A の更新を以下のように行列積の形で一度に行うことが可能となる．

$$(I - \boldsymbol{u}_K\beta_K{\boldsymbol{u}_K}^\top)^\top \cdots (I - \boldsymbol{u}_1\beta_1{\boldsymbol{u}_1}^\top)^\top A(I - \boldsymbol{u}_1\beta_1{\boldsymbol{u}_1}^\top) \cdots (I - \boldsymbol{u}_K\beta_K{\boldsymbol{u}_K}^\top) = A - UV^\top - VU^\top. \tag{A.3}$$

しかし，行列ベクトル積はそのまま残り（行列 V の計算のため），この部分がメモリバンド幅律速となり，高性能化の際に大きなボトルネックとなる．

二番目のステップでは，Cuppen により提案された分割統治法 [24] により，三重対角行列の固有値と固有ベクトルを計算する．この手法は，図 A.2(b) に示すように三重対角行列をブロック対角行列とランク 1 の摂動に分解し，ブロック対角行列の各ブロックの固有値分解結果を利用して元の（ランク 1 の摂動が加わった）行列の固有値分解を効率的に計算する，という原理に基づいている．

三番目のステップでは，最初のステップで得られたハウスホルダー変換を逆の順番に作用させて固有ベクトルの逆変換を行う．実際には，複数個のハウスホルダー変換が，

$$H_1 \cdots H_K = (I - \boldsymbol{u}_1\beta_1\boldsymbol{u}_1^\top) \cdots (I - \boldsymbol{u}_K\beta_K\boldsymbol{u}_K^\top) \to I - USU^\top, \quad U = [\boldsymbol{u}_1 \cdots \boldsymbol{u}_K] \tag{A.4}$$

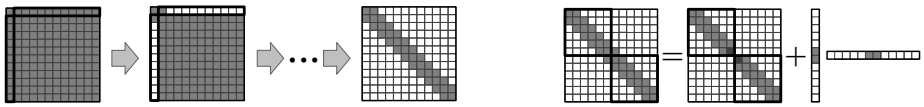
と少ないコスト（S の計算のみ）で行列を使った表現に合成できる（compact-WY 表現）ことを用いて，

$$H_1 \cdots H_{N-2} Y = (I - U_1 S_1 U_1^\top) \cdots (I - U_M S_M U_M^\top) Y \to X \tag{A.5}$$

として行列積（Level-3 BLAS）で計算するため，高性能が期待できる．なお，ここで

$$Y = [\boldsymbol{y}_1 \cdots \boldsymbol{y}_N], \quad X = [\boldsymbol{x}_1 \cdots \boldsymbol{x}_N] \tag{A.6}$$

である．



(a) 三重対角化　　(b) 分割統治法

図 A.2: eigen_s の固有値計算の概略図.

eigen_s では，一番目と三番目のステップは独自の実装を採用しており，適切なスレッド並列化等を施すことで高性能化を図っている．一方，二番目のステップについては，ScaLAPACK のコードをベースとした実装となっている．

A.4 eigen_sx

EigenExa で提供されるもう一つのドライバルーチン eigen_sx は帯行列の固有値・固有ベクトルを直接計算するアプローチを採用している．現状，後述の理由により帯行列として五重対角行列を採用している．具体的には，以下の三つのステップにより固有値問題を解く．

1. ブロック版のハウスホルダー変換による入力行列の五重対角化：$\tilde{Q}^\top A \tilde{Q} \to B$
2. 分割統治法による五重対角行列の固有値・固有ベクトルの計算：$B\boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i$
3. 固有ベクトルの逆変換：$\tilde{Q}\boldsymbol{y}_i \to \boldsymbol{x}_i$

最初のステップでは，ブロック版のハウスホルダー変換を両側から

$$\tilde{H}_{N/2-1}^\top \cdots \tilde{H}_1^\top A \tilde{H}_1 \cdots \tilde{H}_{N/2-1} \to P, \quad \tilde{H}_i = I - \tilde{\boldsymbol{u}}_i \tilde{\beta}_i \tilde{\boldsymbol{u}}_i^\top \tag{A.7}$$

と作用させて，入力行列を二列（二行）ごとに三重対角行列に変形していく（図 A.3(a)）．ただし，

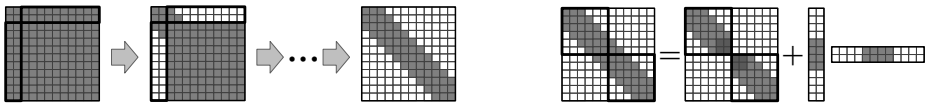
$$\tilde{\boldsymbol{u}}_i = [\boldsymbol{u}_1^{(i)}\ \boldsymbol{u}_1^{(i)}], \quad \tilde{\beta}_i = \begin{pmatrix} \beta_{11}^{(i)} & \beta_{12}^{(i)} \\ \beta_{21}^{(i)} & \beta_{22}^{(i)} \end{pmatrix} \tag{A.8}$$

である．あとは，式 (A.2) と全く同じ形式で計算を行うことができ，Dongarra の手法も同様に適用が可能となっている．その結果として，最終的にボトルネックになるのは，$A\tilde{\boldsymbol{u}}$ の部分となる．

二番目のステップでは，図 A.3(b) に示すように五重対角行列をブロック対角行列とランク 2 の摂動に分解し，三重対角行列に対する分割統治法と同様の原理を繰り返す（ランク 2 の摂動を二つ

のランク 1 の摂動として処理する）ことにより，五重対角行列の固有値・固有ベクトルを計算する [25].

最後のステップでは，最初のステップで用いたブロック版ハウスホルダー変換を逆順で作用させることで，固有ベクトルの逆変換を計算する．ブロック版のハウスホルダー変換に関しても，式 (A.4) と同様の形で行列を使った表現に合成することが可能であるため，同様の計算手順により行列積が利用可能で高性能が期待できる．



(a) 五重対角化　　(b) 分割統治法

図 A.3: eigen_sx の固有値計算の概略図.

eigen_sx でも，一番目と三番目のステップは独自の実装を採用しており，適切なスレッド並列化等を施すことで高性能化を図っている．一方，二番目のステップについては，ScaLAPACK のコードをベースとして，五重対角行列向けに拡張した実装を行っている．

A.5　eigen_s と eigen_sx の違い

先の二つの節の説明から，eigen_s と eigen_sx は同じような三つのステップで構成されており，特に固有ベクトルの逆変換のステップに関しては，ほぼ同様の計算内容となり両者の間に大きな差はない．本節では，この二つのステップにおける両者の差を述べる．

三重（五重）対角化

このステップにおける違いは，eigen_s では一本のベクトルを単位として処理をするのに対して，eigen_sx では二本のベクトルを単位として処理をする部分である．具体的には，

$$\boldsymbol{w} = A\boldsymbol{u} \text{ (in eigen\_s)} \quad \rightarrow \quad [\boldsymbol{w}_1\ \boldsymbol{w}_2] = A[\boldsymbol{u}_1\ \boldsymbol{u}_2] \text{ (in eigen\_sx)} \tag{A.9}$$

といった部分である。まず，ステップ全体としての演算量に関しては，(少なくとも最高次の項に関しては）両者は同程度となる．これは，一回当たり演算量は eigen_s の方が少ない（約半分）が，処理の回数は二列（二行）ずる進める eigen_sx の方が少ない（約半分）からである．同様に，分散並列計算時に通信するデータの総量に関しても，両者は同程度となる．この理由も演算量の場合と同じで，一回当たりのデータ量と回数の関係にある．

両者の間で生じる一つ目の差は，浮動小数点演算（特に行列ベクトル積）の実効性能である．$A\boldsymbol{u}$ を計算する場合に対して $A[\boldsymbol{u}_1\ \boldsymbol{u}_2]$ を計算する場合，行列 A のデータの再利用性が向上する．つまり，要求 byte/flop 値が減少することになる．これにより，メモリバンド幅に律速される影響が小さく（理論的には約半分）なり，演算の実効性能の向上が期待できる．

二つ目の差は，通信回数の違いから生じる，通信のレイテンシの差である．eigen_sx は一回の通信データ量が多いが，通信回数は eigen_s よりも少なく（約半分）なる．近年，通信のレイテンシ

が大きな問題となっているため，この通信回数が少ない（Communication-Avoidance）という特徴は，特に並列数が多くなった場合に非常に大きな差となる．

このように，eigen_sx は eigen_s に比べて，演算性能と通信のレイテンシの面で有利であり，並列数に対して問題サイズが十分に大きい（演算時間が支配的な）場合は前者の効果が大きく，逆に並列数が多い（通信時間が支配的な）場合は後者の効果が大きくなることが予想される．

分割統治法

分割統治法では，eigen_s が毎回ランク 1 の摂動を処理するのに対して，eigen_sx ではランク 2 の摂動を処理する必要があるため，その分の演算量と通信コストが両者の差として生じる．さらに，五重対角行列に対する分割統治法では，ランク 2 の摂動を二つのランク 1 の摂動として処理するが，二回目のランク 1 の摂動を処理する場合に，行列の構造を利用することが出来なくなり（一回目はブロック対角という構造を活用して演算量を削減できる），その結果として，演算量が三倍程度となってしまう．

ただし，分割統治法では，デフレーションと呼ばれる技法により，演算量を大幅に削減することが可能となる場合がある．デフレーションの発生頻度は問題依存であり，三重対角行列を経由した場合と五重対角行列を経由した場合では，同じ入力行列でも，デフレーションの起こり方に差が生じる．そのため，両者のコストの差を理論的に見積もることは容易ではない．

A.6 おわりに

本章では，EigenExa で提供されている二つのルーチン eigen_s と eigen_sx のアルゴリズムの概要とその違いを説明した．一般に経由する帯行列の帯幅を大きくすると，最初の変換するステップでは有利になり，二番目の分割統治法のステップでは不利になるというトレードオフがあり，現状では，五重対角行列が妥当であるとの判断から，eigen_sx では五重対角化を採用している．今後のシステムの性能や分割統治法の実装の改善次第（現状の ScaLAPACK ベースのコードの性能は十分とは言い難い）で，より大きな帯行列（例えば七重対角行列）が有望となる可能性もある．逆に，従来の三重対角化（eigen_s）を使う場合に最善となるケースも状況次第では在り得ると思われる．ぜひ，本章の内容を把握した上で，ユーザが適切なルーチンを選択してもらえると幸いである．

謝辞

EigenExa チーム全員の真摯な協力と理化学研究所計算科学研究機構各位の支援に対して感謝する．また，多くの利用者からバグ情報や品質向上のためのフィードバックを頂いている．関係した多くのメンバーの努力なしには EigenExa を公開することは叶わなかったであろう．最後に，EigenExa 開発には幾つかの外部資金の援助を受けている．それらは以下の通りである．併せてここに謝意を表したい．

- 科学技術振興機構　戦略的創造研究推進事業 CREST 「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」(平成 23 年度〜27 年度)
- 文部科学省　科学研究費補助（科研費）: 基盤研究 (B) 課題番号 21300013 (平成 24 年度), 基盤研究 (A) 課題番号 23240005 (平成 23 年度〜25 年度), 基盤研究 (B) 課題番号 15H02709 (平成 27 年度〜29 年度)
- HPCI 利用研究課題, 「京」一般利用: 課題番号 hp140069 (平成 26 年度), hp120170(平成 24 〜25 年度)
- 「京」調整高度化枠利用: 課題番号 ra000005 (平成 25 年度〜)

また, EigenExa 以前の EigenK の時代には以下の外部資金からの支援を受け開発していた．ここに合わせて謝意を示す．

- 科学技術振興機構　戦略的創造研究推進事業 CREST 「マルチスケール・マルチフィジックス現象の統合シミュレーション」(平成 18 年度〜24 年度)

参考文献

[1] S. Yamada, T. Imamura, T. Kano and M. Machida, “High-Performance Computing for Exact Numerical Approaches to Quantum Many-Body Problems on the Earth Simulator”, Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC ’06), November 2006. Tampa USA.
http://doi.acm.org/10.1145/1188455.1188504

[2] 今村俊幸, “T2K スパコンにおける固有値ソルバの開発”, 東京大学スーパーコンピューティングニュース, Vol.11, No.6, pp.12-32 (2009).
http://www.cc.u-tokyo.ac.jp/support/press/news/VOL11/No6/200911imamura.pdf

[3] T. Imamura, S. Yamada and M. Machida, “Development of a High Performance Eigensolver on the Peta-Scale Next Generation Supercomputer System”, Progress in Nuclear Science and Technology, the Atomic Energy Society of Japan, Vol. 2, pp.643–650 (2011) .

[4] T. Imamura, S. Yamada and M. Machida, “Eigen-K: high performance eigenvalue solver for symmetric matrices developed for K computer ”, 7th International Workshop on. Parallel Matrix Algorithms and Applications (PMAA2012), June 2012, London UK.

[5] 「京」について — 理化学研究所 計算科学研究機構（AICS）,
http://www.aics.riken.jp/jp/k/

[6] スーパーコンピュータ「京」— Fujitsu, Japan,
http://www.fujitsu.com/jp/about/businesspolicy/tech/k/

[7] T. Imamura and Y. Yamamoto, “CREST: Dense Eigen-Engine Groups”, International Workshop on Eigenvalue Problems: Algorithms; Software and Applications, in Petascale Computing (EPASA 2014), Tsukuba, March 7–9, 2014 (poster).
http://www.aics.riken.jp/labs/lpnctrt/EPASA2014_dense_poster_ImamuraT_only.pdf

[8] T. Imamura, “The EigenExa Library – High Performance & Scalable Direct Eigensolver for Large-Scale Computational Science”, HPC in Asia, Leipzig, Germany, June 22–26, 2014.

[9] T. Imamura, Y. Hirota, T. Fukaya, S. Yamada and M. Machida, “EigenExa: high performance dense eigensolver, present and future”, 8th International Workshop on Parallel Matrix Algorithms and Applications (PMAA14), Lugano, Switzerland, July 2–4, 2014.

[10] 深谷猛, 今村俊幸, “FX10 4800 ノードを用いた密行列向け固有値ソルバ EigenExa の性能評価 “, 東京大学スーパーコンピューティングニュース, Vol.16 No.3, pp.20-27 (2014). http://www.cc.u-tokyo.ac.jp/support/press/news/VOL16/No3/09_User201405-1.pdf

[11] T. Fukaya and T. Imamura, “Performance evaluation of the EigenExa eigensolver on the Oakleaf-FX supercomputing system”, Annual Meeting on Advanced Computing System and Infrastructure (ACSI) 2015, Tsukuba, January 26–28, 2015.

[12] Python binding of EigenExa, HPC Usability Research Team, RIKEN AICS, http://www.hpcu.aics.riken.jp/

[13] 杉原正顯, 室田一雄, “線形計算の数理”, 岩波書店 (2009).

[14] B. Parlett, “The Symmetric Eigenvalue Problem”, SIAM (1987).

[15] J. Demmel, “Applied Numerical Linear Algebra”, SIAM (1997).

[16] L. Trefethen and D. Bau, III, “Numerical Liner Algebra”, SIAM (1997).

[17] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, “Numerical Recipes: The Art of Scientific Computing”, 3rd ed., Cambridge University Press (2007).

[18] G. Golub and C. Van Loan, “Matrix Computations”, 4th ed., Johns Hopkins University Press (2012).

[19] 山本有作, “密行列固有値解法の最近の発展 (I) : Multiple Relatively Robust Representations アルゴリズム”, 日本応用数理学会論文誌, Vol.15, No.2, pp.181–208 (2005).

[20] 山本有作, “密行列固有値解法の最近の発展 (II) : マルチシフト QR 法”, 日本応用数理学会論文誌, Vol.16, No.4, pp.507–534 (2006).

[21] ScaLAPACK, http://www.netlib.org/scalapack/

[22] ELPA, http://elpa.rzg.mpg.de/

[23] DPLASMA, http://icl.cs.utk.edu/dplasma/

[24] J. Cuppen, “A divide and conquer method for the symmetric tridiagonal eigenproblem”, Numer. Math, Vol.36, pp.177–195 (1981).

[25] P. Arbenz, “Divide and conquer algorithms for the bandsymmetric eigenvalue problem”, Parallel Computing, Vol.18, No.10, pp.1105–1128 (1992).