IPSJ Transactions on Mathematical Modeling and Its Applications Vol.6 No.2 25–35 (Aug. 2013)

Regular Paper

On Implementation and Evaluation of Inverse Iteration Algorithm with Compact WY Orthogonalization

Hiroyuki Ishigami[1,a)] Kinji Kimura[1] Yoshimasa Nakamura[1]

Received: July 26, 2012, Revised: November 1, 2012,
Accepted: January 23, 2013

Abstract: In this paper, we introduce an inverse iteration algorithm that can be used to compute all the eigenvectors of a real symmetric tri-diagonal matrix on parallel processors. To overcome the sequential bottleneck created by modified Gram-Schmidt orthogonalization in classical inverse iteration, we propose the use of the compact WY representation in the reorthogonalization process, based on the Householder transformation. This change results in drastically reduced synchronization cost during parallel processing.

Keywords: parallel processing, inverse iteration, reorthogonalization, compact WY representation, Householder transformation

1. Introduction

The eigenvalue decomposition of a symmetric matrix (i.e., decomposition into a product of matrices consisting of eigenvectors and eigenvalues) is one of the most important operations in linear algebra. It is used in vibrational analysis, image processing, data searches, etc. In general cases, the eigenvalue decomposition of a real symmetric matrix A is reduced to that of a real symmetric tri-diagonal matrix T. This reduction is called *tri-diagonalization*. The eigenvectors of A are obtained by transforming eigenvectors of T after solving the eigenvalue decomposition of T. This transformation is called *inverse transformation*.

With the introduction of computers equipped with multicore processors, there has been a sharp increase in the demand for an eigenvalue decomposition algorithm that can be effectively parallelized. Several parallelization techniques for tri-diagonalization and inverse transformation have been proposed for shared memory parallel computers and for distributed memory parallel systems [6], [8].

The inverse iteration algorithm computes eigenvectors independently associated with mutually distinct eigenvalues. When eigenvalues are very close to one another, we must reorthogonalize the eigenvectors, typically using *modified Gram-Schmidt* (*MGS*) algorithm (hereafter referred to *classical inverse iteration*). Unfortunately, the MGS is sequential and does not map effectively to parallel processors.

The Householder transformation [12] offers another method to orthogonalize vectors (hereafter referred to as *Householder orthogonalization*). Like MGS, the Householder orthogonalization is sequential, and therefore unsuited to parallel processing. Unlike MGS, however, the Householder orthogonalization can be considered stable, since the orthogonality of the resulting vectors does not depend on the condition number of the matrix [13].

In 1989, R. Schreiber et al. proposed a stable and parallelizable Householder orthogonalization in terms of the compact WY representation [11] (hereafter referred to as *compact WY orthogonalization*). Yamamoto et al. [13] reformulated this algorithm for incremental orthogonalization, and showed that it could theoretically achieve very accurate orthogonality and high scalability through parallel computation [13]. The incremental orthogonalization is implemented on several numerical computation libraries, such as LAPACK (Linear Algebra PACKage) [9], which is implemented using BLAS (Basic Linear Algebra Subroutines). The compact WY orthogonalization algorithm can be implemented by using BLAS, directly.

In Ref. [7], the authors implemented the compact WY orthogonalization for reorthogonalization of inverse iteration in computing eigenvectors of a symmetric tri-diagonal matrix. They also showed that, in parallel processing, this inverse iteration algorithm is faster than the classical one.

In this paper, we introduce two new implementations of compact WY orthogonalization. The first is based on BLAS, and is focused on reformulating the mathematical structure of the algorithm in order to reduce the overall computational cost. The second focuses on the inverse iteration algorithm for a real symmetric tri-diagonal matrix. We then evaluate the performance of the second implementation through numerical experiments.

2. Classical Inverse Iteration and Its Limitation

2.1 Classical Inverse Iteration

We first consider the problem of computing eigenvectors of a real symmetric tri-diagonal matrix $T \in \mathbb{R}^{n\times n}$. Let $\lambda_j \in \mathbb{R}$ be eigenvalues of T such that $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ and let $\boldsymbol{v}_j \in \mathbb{R}^n$ be the eigenvector associated with λ_j. When $\tilde{\lambda}_j$, an approximate

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[a)] hishigami@amp.i.kyoto-u.ac.jp

value of λ_j, and a starting vector $\boldsymbol{v}_j^{(0)}$ are given, we can compute the eigenvectors of T using the following equation iteratively:

$$\left(T - \tilde{\lambda}_j I\right) \boldsymbol{v}_j^{(k)} = \boldsymbol{v}_j^{(k-1)}, \tag{1}$$

where I is the n-dimensional identity matrix. If the eigenvalues of T are mutually well-separated, $\boldsymbol{v}_j^{(k)}$, the solution of Eq. (1), generically converges to the eigenvector associated with λ_j as k goes to ∞. The above iteration method is *inverse iteration*. Its computational cost is $O(mn)$ when we compute m eigenvectors. In the implementation, we must normalize the vectors $\boldsymbol{v}_j^{(k)}$ to avoid overflow.

When some of the eigenvalues are close to one another or there are clusters of eigenvalues of T, we must reorthogonalize all the eigenvectors associated with such eigenvalues. In the classical inverse iteration, we use the MGS algorithm to accomplish this, with a computational cost of $O(m^2 n)$. Thus, when we compute eigenvectors of a matrix with many clustered eigenvalues, the computational cost increases significantly. Note that the classical inverse iteration has also been implemented using the Peters-Wilkinson method [10]. In this method, when the distance between the close eigenvalues is less than $10^{-3}\|T\|$, we regard them as members of the same cluster of eigenvalues, and we orthogonalize all of the eigenvectors associated with the clustered eigenvalues. The classical inverse iteration algorithm is given by Algorithm 1, where j_1 denotes the index of the minimum eigenvalue of some cluster. This algorithm is implemented as DSTEIN, a LAPACK [9] code for computing eigenvectors of a real symmetric tri-diagonal matrix.

2.2 Limitation of Classical Inverse Iteration

Inverse iteration is an important method for computing eigenvectors, because it allows us to compute eigenvectors independently. When there are many clusters in the distribution of eigenvalues, the inverse iteration can be parallelized by assigning each cluster to a core.

Let us consider the Peters-Wilkinson method in the classical inverse iteration. When the dimension of T is greater than 1,000, most of the eigenvalues are regarded as being in the same cluster [3], in which case we need to parallelize the inverse iteration not with respect to the cluster but with respect to the computation loop described in lines 2 through 16 in Algorithm 1. We can consider this parallelization in several ways, but this computation loop is sequential. Thus, parallelization with respect to BLAS operations should be adequate for this loop.

The computation loop includes the inverse iteration in Eq. (1) and the orthogonalization of the eigenvectors and the latter represents a bottleneck in the classical inverse iteration with respect to the computational cost. Since the MGS algorithm is mainly based on BLAS level-1 operations, when we compute all the eigenvectors in parallel processing, the number of synchronizations performs $O(m^2)$ time. Thus, it is unsuitable for parallel processing.

Algorithm 1 Classical inverse iteration

```
1:  for $j = 1$ to $n$ do
2:      Generate $\boldsymbol{v}_j^{(0)}$ from random numbers.
3:      $k = 0$.
4:      repeat
5:          $k \leftarrow k + 1$.
6:          Normalize $\boldsymbol{v}_j^{(k-1)}$.
7:          Solve $\left(T - \tilde{\lambda}_j I\right)\boldsymbol{v}_j^{(k)} = \boldsymbol{v}_j^{(k-1)}$ (Eq. (1)).
8:          if $|\tilde{\lambda}_j - \tilde{\lambda}_{j-1}| \le 10^{-3}\|T\|$, then
9:              for $i = j_1$ to $j - 1$ do
10:                 $\boldsymbol{v}_j^{(k)} \leftarrow \boldsymbol{v}_j^{(k)} - \langle \boldsymbol{v}_j^{(k)}, \boldsymbol{v}_i \rangle \boldsymbol{v}_i$
11:             end for
12:         else
13:             $j_1 = j$.
14:         end if
15:     until some condition is met.
16:     Normalize $\boldsymbol{v}_j^{(k)}$ to $\boldsymbol{v}_j$.
17: end for
```

3. Other Orthogonalization Algorithms

In this section, we present alternatives to the MGS orthogonalization algorithm. Consider the incremental orthogonalization of $\boldsymbol{v}_j \in \mathbb{R}^n$ to $\boldsymbol{q}_j \in \mathbb{R}^n$ ($j = 1, \ldots, m$, $m \le n$). The incremental orthogonalization occurs in the reorthogonalization of the inverse iteration and is defined as follows: $\boldsymbol{v}_j$ ($2 \le j \le m$) is not given in advance but is computed from $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{j-1}$.

Let us then define a vector $\boldsymbol{0}_i$ as the i-dimensional zero vector and matrices $V, Q \in \mathbb{R}^{n\times m}$ as $V = [\boldsymbol{v}_1 \cdots \boldsymbol{v}_m]$, $Q = [\boldsymbol{q}_1 \cdots \boldsymbol{q}_m]$.

3.1 Householder Orthogonalization

If vectors $\boldsymbol{u}_j, \boldsymbol{w}_j \in \mathbb{R}^n$ ($j = 1, \ldots, m$) satisfy $\|\boldsymbol{u}_j\|_2 = \|\boldsymbol{w}_j\|_2$, there exist orthogonal matrices H_j called the Householder matrices, satisfying $H_j H_j^\top = H_j^\top H_j = I$, $H_j \boldsymbol{u}_j = \boldsymbol{w}_j$ defined by $H_j = I - t_j \boldsymbol{y}_j \boldsymbol{y}_j^\top$, $\boldsymbol{y}_j = \boldsymbol{u}_j - \boldsymbol{w}_j$, $t_j = 2/\|\boldsymbol{y}_j\|_2^2$. The transformation from $\boldsymbol{u}_j$ to $\boldsymbol{v}_j$ by H_j is called the Householder transformation, and is used for orthogonalization, as shown in Algorithm 2. The vector $\boldsymbol{y}_j$ is the vector in which the elements from 1 to $(j-1)$ are the same as the elements of $\boldsymbol{u}_j$ and the elements from $(j+1)$ to n are zeros. The vectors $\boldsymbol{u}_j$ and $\boldsymbol{w}_j$ are defined as follows:

$$\begin{aligned} \boldsymbol{u}_j &= \begin{bmatrix} u_{1,j} & \cdots & u_{j-1,j} & u_{j,j} & u_{j+1,j} & \cdots & u_{n,j} \end{bmatrix}^\top \\ &= H_{j-1} H_{j-2} \cdots H_2 H_1 \boldsymbol{v}_j, \\ \boldsymbol{w}_j &= \begin{bmatrix} u_{1,j} & \cdots & u_{j-1,j} & c_j & \boldsymbol{0}_{n-j}^\top \end{bmatrix}^\top, \end{aligned}$$

where $u_{i,j}$ ($i = 1, \ldots, n$) is the i-th element of $\boldsymbol{u}_j$ and

$$c_j = -\operatorname{sgn}(u_{j,j}) \sqrt{\sum_{i=j}^{n} u_{i,j}^2}.$$

Hence, $\boldsymbol{y}_j$ is computed as

Algorithm 2 Householder orthogonalization

```
1:  for $j = 1$ to $m$ do
2:      $\boldsymbol{u}_j \leftarrow \left(I - t_1 \boldsymbol{y}_1 \boldsymbol{y}_1^\top\right)\boldsymbol{v}_j$
3:      for $i = 2$ to $j - 1$ do
4:          $\boldsymbol{u}_j \leftarrow \left(I - t_i \boldsymbol{y}_i \boldsymbol{y}_i^\top\right)\boldsymbol{u}_j$
5:      end for
6:      Compute $\boldsymbol{y}_j$ and $t_j$ by using $\boldsymbol{u}_j$
7:      $\boldsymbol{q}_j \leftarrow \left(I - t_j \boldsymbol{y}_j \boldsymbol{y}_j^\top\right)\boldsymbol{e}_j$
8:      for $i = j - 1$ to 1 do
9:          $\boldsymbol{q}_j \leftarrow \left(I - t_i \boldsymbol{y}_i \boldsymbol{y}_i^\top\right)\boldsymbol{q}_j$
10:     end for
11: end for
```

$$\boldsymbol{y}_j = \begin{bmatrix} \boldsymbol{0}_{j-1}^\top & u_{j,j} - c_j & u_{j+1,j} & \cdots & u_{n,j} \end{bmatrix}^\top. \tag{2}$$

The vector $\boldsymbol{e}_j$ in Algorithm 2 is the j-th vector of an n-dimensional identity matrix.

The orthogonality of the vectors $\boldsymbol{q}_j$ generated by the Householder orthogonalization does not depend on the condition number of V, making it more stable than that of MGS. On the other hand, like MGS, it is a sequential algorithm, mainly based on BLAS level-1 operations, with a computational cost of about double than that of MGS. Thus, it too is an ineffective for parallel processing.

3.2 Compact WY Orthogonalization

Yamamoto and Hirota [13] suggested that the Householder orthogonalization can be computed using BLAS level-2 operations in terms of *compact WY representation* proposed by Schreiber and van Loan [11]. They also theoretically demonstrated that this algorithm could achieve high orthogonality and high scalability across parallel processing [13].

Let us consider the Householder orthogonalization in Algorithm 2 with the addition of compact WY representation. First, we define $Y_1 = [\boldsymbol{y}_1] \in \mathbb{R}^{n\times 1}$ and $T_1 = [t_1] \in \mathbb{R}^{1\times 1}$. We then define matrices $Y_j \in \mathbb{R}^{n\times j}$ and upper triangular matrices $T_j \in \mathbb{R}^{j\times j}$, recursively, as follows:

$$Y_j = \begin{bmatrix} Y_{j-1} & \boldsymbol{y}_j \end{bmatrix},\ T_j = \begin{bmatrix} T_{j-1} & -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j \\ \boldsymbol{0}_{j-1}^\top & t_j \end{bmatrix}. \tag{3}$$

In this case, the following equation holds

$$H_1 H_2 \cdots H_j = I - Y_j T_j Y_j^\top. \tag{4}$$

As shown in Eq. (4), we can rewrite the product of the Householder matrices $H_1 H_2 \cdots H_j$ in a simple block matrix form. Here $I - Y_j T_j Y_j^\top$ is the compact WY representation of the product $H_1 H_2 \cdots H_j$ of the Householder matrices. Algorithm 3 shows the compact WY orthogonalization algorithm.

3.3 Implementation of Compact WY Orthogonalization

In this subsection, we present an implementation of the compact WY orthogonalization algorithm using BLAS operations. We also discuss the mathematical structure of this algorithm and present a new implementation of the compact WY orthogonalization that reduces computational cost.

3.3.1 Ordinary Implementation of Compact WY Orthogonalization Using BLAS

Consider the implementation of the compact WY orthogonalization in lines 5 to 8 of Algorithm 3. To use BLAS operations here, we need to reformulate line 5 as follows:

$$\begin{aligned} \boldsymbol{u}_j &= \left(I - Y_{j-1} T_{j-1}^\top Y_{j-1}^\top\right) \boldsymbol{v}_j \\ &= \boldsymbol{v}_j - Y_{j-1} T_{j-1}^\top Y_{j-1}^\top \boldsymbol{v}_j. \end{aligned}$$

We can now implement this formula by using BLAS as follows:

$$\begin{cases} \boldsymbol{u}_j \leftarrow \boldsymbol{v}_j & \text{(DCOPY)} \\ \boldsymbol{v}'_{j-1} \leftarrow Y_{j-1}^\top \boldsymbol{u}_j + 0 \cdot \boldsymbol{v}'_{j-1} & \text{(DGEMV)} \\ \boldsymbol{v}'_{j-1} \leftarrow T_{j-1}^\top \boldsymbol{v}'_{j-1} & \text{(DTRMV)} \\ \boldsymbol{u}_j \leftarrow (-1) \cdot Y_{j-1} \boldsymbol{v}'_{j-1} + \boldsymbol{u}_j & \text{(DGEMV)} \end{cases},$$

where $\boldsymbol{v}'_{j-1} \in \mathbb{R}^{j-1}$. We set the initial memory address of $\boldsymbol{v}'_{j-1}$ to correspond to that of $\boldsymbol{v}_j$. DCOPY denotes the copying operation of a vector $\boldsymbol{x}$ to a vector $\boldsymbol{y}$: $\boldsymbol{y} \leftarrow \boldsymbol{x}$. DGEMV denotes the matrix-vector operation $\boldsymbol{y} \leftarrow \alpha A \boldsymbol{x} + \beta \boldsymbol{y}$, where A is a general rectangular matrix. DTRMV denotes the matrix-vector product $\boldsymbol{x} \leftarrow T\boldsymbol{x}$, where T is a triangular matrix.

Next, on line 6, we compute $\boldsymbol{y}_j$ in Eq. (2) and t_j. These computations are mostly performed using BLAS level-1 operations, so their cost is relatively low. We implement the computation of $\boldsymbol{y}_j$ and t_j as follows:

$$\begin{cases} y_{i,j} \leftarrow 0,\ (i = 1, \ldots, j-1) & \\ y_{i,j} \leftarrow u_{i,j},\ (i = j, \ldots, n) & \text{(DCOPY)} \\ y_{j,j} \leftarrow u_{j,j} - c_j,\ c_j = -\operatorname{sgn}(u_{j,j}) \sqrt{\sum_{i=j}^n u_{i,j}^2} & \text{(DNRM2)} \\ t_j \leftarrow 2/\|\boldsymbol{y}_j\|_2^2 & \text{(DNRM2)} \end{cases},$$

where $y_{i,j}$ $(i = 1, \ldots, n)$ is the i-th column element of $\boldsymbol{y}_j$, and DNRM2 denotes the computation of the 2-norm of a vector.

On line 7, updating Y_j and t_j is easily accomplished. Let $\hat{\boldsymbol{t}}_j \in \mathbb{R}^{j-1}$ be $\hat{\boldsymbol{t}}_j = -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j$. Note that $\hat{\boldsymbol{t}}_j$ is implemented by using BLAS as follows:

$$\begin{cases} \hat{\boldsymbol{t}}_j \leftarrow (-t_j) Y_{j-1}^\top \boldsymbol{y}_j + 0 \cdot \hat{\boldsymbol{t}}_j & \text{(DGEMV)} \\ \hat{\boldsymbol{t}}_j \leftarrow T_{j-1} \hat{\boldsymbol{t}}_j & \text{(DTRMV)} \end{cases}.$$

Finally, on line 8, we can reformulate as follows:

$$\begin{aligned} \boldsymbol{q}_j &= \left(I - Y_j T_j Y_j^\top\right) \boldsymbol{e}_j \\ &= \boldsymbol{e}_j - Y_j T_j Y_j^\top \boldsymbol{e}_j, \end{aligned}$$

where the matrix-vector product $Y_j^\top \boldsymbol{e}_j$ can be simplified by

$$Y_j^\top \boldsymbol{e}_j = \begin{bmatrix} y_{j,1} \\ \vdots \\ y_{j,j} \end{bmatrix}.$$

This computation can be performed only by copying the j-th column of Y_j to some vector. Thus, using BLAS, we implement the formula of line 8 as

$$\begin{cases} \boldsymbol{q}_j \leftarrow \boldsymbol{e}_j & \text{(DCOPY)} \\ \boldsymbol{v}'_j \leftarrow \begin{bmatrix} y_{j,1} & \cdots & y_{j,j} \end{bmatrix}^\top & \text{(DCOPY)} \\ \boldsymbol{v}'_j \leftarrow T_j^\top \boldsymbol{v}'_j & \text{(DTRMV)} \\ \boldsymbol{q}_j \leftarrow (-1) \cdot Y_j \boldsymbol{v}'_j + \boldsymbol{q}_j & \text{(DGEMV)} \end{cases},$$

Algorithm 3 compact WY orthogonalization algorithm

```
1: Compute $\boldsymbol{y}_1$ and $t_1$ by using $\boldsymbol{u}_1 = \boldsymbol{v}_1$
2: $Y_1 = [\boldsymbol{y}_1]$, $T_1 = [t_1]$
3: $\boldsymbol{q}_1 \leftarrow \left(I - Y_1 T_1 Y_1^\top\right) \boldsymbol{e}_j$
4: for $j = 2$ to $m$ do
5:     $\boldsymbol{u}_j \leftarrow \left(I - Y_{j-1} T_{j-1}^\top Y_{j-1}^\top\right) \boldsymbol{v}_j$
6:     Compute $\boldsymbol{y}_j$ and $t_j$ by using $\boldsymbol{u}_j$
7:     $Y_j = \begin{bmatrix} Y_{j-1} & \boldsymbol{y}_j \end{bmatrix}$,  $T_j = \begin{bmatrix} T_{j-1} & -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j \\ \boldsymbol{0}_{j-1}^\top & t_j \end{bmatrix}$.
8:     $\boldsymbol{q}_j \leftarrow \left(I - Y_j T_j Y_j^\top\right) \boldsymbol{e}_j$
9: end for
```

where $\boldsymbol{v}'_j \in \mathbb{R}^j$, $\boldsymbol{q}_j \in \mathbb{R}^n$. We set the initial memory address of $\boldsymbol{v}'_j$ and $\boldsymbol{q}_j$ to correspond to that of $\boldsymbol{u}_j$ and $\boldsymbol{v}_j$, respectively.

The computational cost of the above compact WY orthogonalization algorithm is almost $4m^2n + m^3$. In the worst case, i.e., when $m = n$, the computational cost is $5n^3$. Worse yet, for this implementation, we have to use almost $mn + m^2$ memory, since Y_m uses mn and T_m uses m^2.

3.3.2 New Implementation of Compact WY Orthogonalization Using BLAS

Having detailed the ordinary implementation of the compact WY orthogonalization algorithm, we now discuss the mathematical structure of this algorithm and present a new implementation of compact WY orthogonalization with lower computational cost.

We discuss the formula in line 5 before moving on to that in line 6 of Algorithm 3. Since

$$c_j = -\operatorname{sgn}\left(u_{j,j}\right)\sqrt{\sum_{i=j}^{n} u_{i,j}^2},$$

we have

$$\begin{aligned}\|\boldsymbol{y}_j\|_2^2 &= \left(u_{j,j} - c_j\right)^2 + \sum_{i=j+1}^{n} u_{i,j}^2\\ &= \sum_{i=j}^{n} u_{i,j}^2 - 2u_{j,j}c_j + c_j^2\\ &= 2(c_j^2 - u_{j,j}c_j).\end{aligned}$$

Hence, we have

$$t_j = \frac{2}{\|\boldsymbol{y}_j\|_2^2} = \frac{1}{c_j^2 - u_{j,j}c_j}.$$

From this fact and the definition of $\boldsymbol{y}_j$ and c_j, we observe that we do not need to compute the 1-th to the $(j-1)$-th elements of $\boldsymbol{u}_j$ in actual. Restricting our computation to the j-th to the n-th elements of $\boldsymbol{u}_j$, the formula on line 5 is reduced to

$$\hat{\boldsymbol{u}}_j = \hat{\boldsymbol{v}}_j - \hat{Y}_{j-1}T_{j-1}^{\top}Y_{j-1}^{\top}\boldsymbol{v}_j,$$

where $\hat{\boldsymbol{u}}_j \in \mathbb{R}^{n-(j-1)}$ is $\hat{\boldsymbol{u}}_j = \begin{bmatrix} u_{j,j} & \cdots & u_{n,j}\end{bmatrix}^{\top}$ and $\hat{\boldsymbol{v}}_j \in \mathbb{R}^{n-(j-1)}$ is $\hat{\boldsymbol{v}}_j = \begin{bmatrix} v_{j,j} & \cdots & v_{n,j}\end{bmatrix}^{\top}$.

Now, considering the structure of $\boldsymbol{y}_j$, we can represent $\boldsymbol{y}_j$ $(j = 2, \ldots, m)$ from Eq. (2) as the block vector of the form

$$\boldsymbol{y}_j = \begin{bmatrix}\boldsymbol{0}_{j-1}\\ \hat{\boldsymbol{y}}_j\end{bmatrix},$$

where $\hat{\boldsymbol{y}}_j \in \mathbb{R}^{n-(j-1)}$ is the vector of nonzero elements of $\boldsymbol{y}_j$. From this fact, Y_j can be represented as the following block matrix:

$$Y_j = \begin{bmatrix}L_j\\ \hat{Y}_j\end{bmatrix},$$

where $L_j \in \mathbb{R}^{j\times j}$ is a lower triangular matrix and $\hat{Y}_j \in \mathbb{R}^{(n-j)\times j}$ is generally a dense rectangular matrix. Let us also consider $\boldsymbol{v}_j$ as the block vector of the form

$$\boldsymbol{v}_j = \begin{bmatrix}\check{\boldsymbol{v}}_j\\ \hat{\boldsymbol{v}}_j\end{bmatrix},$$

where $\check{\boldsymbol{v}}_j \in \mathbb{R}^{j-1}$ is $\check{\boldsymbol{v}}_j = \begin{bmatrix} v_{1,j} & \cdots & v_{j-1,j}\end{bmatrix}^{\top}$. Using these block forms of $\boldsymbol{v}_j$ and Y_j, we can reduce the computational cost of the matrix-vector product $Y_{j-1}^{\top}\boldsymbol{v}_j$ through

$$Y_{j-1}^{\top}\boldsymbol{v}_j = \begin{bmatrix}L_{j-1}\\ \hat{Y}_{j-1}\end{bmatrix}^{\top}\begin{bmatrix}\check{\boldsymbol{v}}_j\\ \hat{\boldsymbol{v}}_j\end{bmatrix} = L_{j-1}^{\top}\check{\boldsymbol{v}}_j + \hat{Y}_{j-1}^{\top}\hat{\boldsymbol{v}}_j.$$

Hence, the formula of $\hat{\boldsymbol{u}}_j$ can be simplified as follows:

$$\hat{\boldsymbol{u}}_j = \hat{\boldsymbol{v}}_j - \hat{Y}_{j-1}T_{j-1}^{\top}\left(L_{j-1}^{\top}\check{\boldsymbol{v}}_j + \hat{Y}_{j-1}^{\top}\hat{\boldsymbol{v}}_j\right).$$

This formula can then be implemented using BLAS as follows:

$$\begin{cases}\hat{\boldsymbol{u}}_j \leftarrow \hat{\boldsymbol{v}}_j & \text{(DCOPY)}\\ \check{\boldsymbol{v}}_j \leftarrow L_{j-1}^{\top}\check{\boldsymbol{v}}_j & \text{(DTRMV)}\\ \check{\boldsymbol{v}}_j \leftarrow \hat{Y}_{j-1}^{\top}\hat{\boldsymbol{v}}_j + \check{\boldsymbol{v}}_j & \text{(DGEMV)}\\ \check{\boldsymbol{v}}_j \leftarrow T_{j-1}^{\top}\check{\boldsymbol{v}}_j & \text{(DTRMV)}\\ \hat{\boldsymbol{u}}_j \leftarrow (-1)\cdot\hat{Y}_{j-1}\check{\boldsymbol{v}}_j + \hat{\boldsymbol{u}}_j & \text{(DGEMV)}\end{cases}.$$

Given the above, we can now implement the computation on line 6 using BLAS as follows:

$$\begin{cases}y_{i,j} \leftarrow u_{i,j},\ (i = j, \ldots, n) & \text{(DCOPY)}\\ y_{j,j} \leftarrow u_{j,j} - c_j,\ c_j = -\operatorname{sgn}(u_{j,j})\sqrt{\sum_{i=j}^{n} u_{i,j}^2} & \text{(DNRM2)}\\ t_j \leftarrow 1/\left(c_j^2 - u_{j,j}c_j\right)\end{cases}.$$

On line 7, we can further reduce the computational cost of $\hat{\boldsymbol{t}}_j$ through

$$\begin{aligned}\hat{\boldsymbol{t}}_j &= -t_jT_{j-1}Y_{j-1}^{\top}\boldsymbol{y}_j\\ &= -t_jT_{j-1}\begin{bmatrix}L_{j-1}\\ \hat{Y}_{j-1}\end{bmatrix}^{\top}\begin{bmatrix}\boldsymbol{0}_{j-1}\\ \hat{\boldsymbol{y}}_j\end{bmatrix}\\ &= -t_jT_{j-1}\left(L_{j-1}^{\top}\boldsymbol{0}_{j-1} + \hat{Y}_{j-1}^{\top}\hat{\boldsymbol{y}}_j\right)\\ &= -t_jT_{j-1}\hat{Y}_{j-1}^{\top}\hat{\boldsymbol{y}}_j.\end{aligned}$$

This formula can be implemented using BLAS as follows:

$$\begin{cases}\hat{\boldsymbol{t}}_j \leftarrow (-t_j)\hat{Y}_{j-1}^{\top}\hat{\boldsymbol{y}}_j + 0\cdot\hat{\boldsymbol{t}}_j & \text{(DGEMV)}\\ \hat{\boldsymbol{t}}_j \leftarrow T_{j-1}\hat{\boldsymbol{t}}_j & \text{(DTRMV)}\end{cases}.$$

Finally, on line 8, even if the sign of the orthogonal vector $\boldsymbol{q}_j$ is reversed, the orthogonality with respect to other vectors is not changed. Thus, we can reformulate $\boldsymbol{q}_j$ as $\boldsymbol{q}_j = \left(Y_jT_jY_j^{\top} - I\right)\boldsymbol{e}_j$. Furthermore, let us consider $\boldsymbol{q}_j$ as the following block vector:

$$\boldsymbol{q}_j = \begin{bmatrix}\check{\boldsymbol{q}}_j\\ \hat{\boldsymbol{q}}_j\end{bmatrix},$$

where $\check{\boldsymbol{q}}_j \in \mathbb{R}^j$, $\hat{\boldsymbol{q}}_j \in \mathbb{R}^{n-j}$. These can be reformulated as follows:

$$\begin{bmatrix}\check{\boldsymbol{q}}_j\\ \hat{\boldsymbol{q}}_j\end{bmatrix} = \begin{bmatrix}L_jT_jY_j^{\top}\boldsymbol{e}_j\\ \hat{Y}_jT_jY_j^{\top}\boldsymbol{e}_j\end{bmatrix} - \begin{bmatrix}\check{\boldsymbol{e}}_j\\ \boldsymbol{0}_{n-j}\end{bmatrix},$$

where $\check{\boldsymbol{e}}_j$ is the j-th vector of the j-dimensional identity matrix. This formula can be implemented by using BLAS as follows:

$$\begin{cases}\boldsymbol{x}_j \leftarrow \begin{bmatrix}y_{j,1} & \cdots & y_{j,j}\end{bmatrix}^{\top} & \text{(DCOPY)}\\ \boldsymbol{x}_j \leftarrow T_j^{\top}\boldsymbol{x}_j & \text{(DTRMV)}\\ \check{\boldsymbol{q}}_j \leftarrow \boldsymbol{x}_j & \text{(DCOPY)}\\ \check{\boldsymbol{q}}_j \leftarrow L_j\check{\boldsymbol{q}}_j & \text{(DTRMV)}\\ \hat{\boldsymbol{q}}_j \leftarrow \hat{Y}_j\boldsymbol{x}_j + 0\cdot\hat{\boldsymbol{q}}_j & \text{(DGEMV)}\\ q_{j,j} \leftarrow q_{j,j} - 1\end{cases},$$

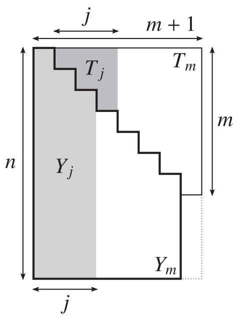


Fig. 1 Assignment model for Y_j and T_j.

Table 1 Comparison of the orthogonalization methods [1], [13].

orthogonalization	*Computation*	*Synchronization*	*Orthogonality*
MGS	$2m^2n$	$O(m^2)$	$O(\varepsilon\kappa(V))$
Householder	$4m^2n$	$O(m^2)$	$O(\varepsilon)$
compact WY	$4m^2n+m^3$	$O(m)$	$O(\varepsilon)$
new compact WY	$4m^2n-m^3$	$O(m)$	$O(\varepsilon)$

where $\boldsymbol{x}_j \in \mathbb{R}^j$.

When the above implementation is adopted, the highest order of computational cost for the compact WY algorithm is reduced from $4m^2n+m^3$ to $4m^2n-m^3$. In the worst case, i.e., when $m=n$, the computational cost of the new implementation of the compact WY algorithm is no more than $3n^3$. In addition, our implementation does not have to refer any zero elements of Y_j and T_j, so if Y_j and T_j are assigned as in **Fig. 1**, memory use can be reduced to almost $n(m+1)$.

3.4 Comparison of Orthogonalization Algorithms

The compact WY orthogonalization has a stable orthogonality arising from the Householder transformations, and its numerical computation is mainly performed by BLAS level-2 operations. As a result, it is better suited to parallel processing than MGS. **Table 1** displays the differences in performance among the orthogonalization algorithms mentioned above. *Computation* denotes the order of the computational cost, *Synchronization* indicates the order of the number of synchronizations, *Orthogonality* indicates the norm $\|Q^\top Q - I\|$, ε denotes the machine epsilon, and $\kappa(V)$ denotes the condition number of V.

4. Inverse Iteration Algorithm with Compact WY Orthogonalization

The authors have also proposed an alternative inverse iteration algorithm in Ref. [7]. This algorithm is based on the classical inverse iteration algorithm implemented in DSTEIN, with compact WY orthogonalization substituted for MGS orthogonalization, as given in Algorithm 4 (hereafter referred to as *compact WY inverse iteration*). We identify two codes of this algorithm, DSTEIN-cWY and DSTEIN-ncWY, corresponding to the two implementations of compact WY orthogonalization. DSTEIN-cWY is based on the processing described in Section 3.3.1, and has already been shown to be faster for parallel processing than classical inverse iteration [7]. DSTEIN-ncWY is based on the processing described in Section 3.3.2.

There are a couple of differences between the classical inverse iteration and the compact WY inverse iteration. For one, in the classical inverse iteration algorithm, we need not know the index

Algorithm 4 compact WY inverse iteration

```
1:  for $j = 1$ to $n$ do
2:      Generate $\boldsymbol{v}_j^{(0)}$ from random numbers.
3:      $k = 0$
4:      repeat
5:          $k \leftarrow k + 1$.
6:          Normalize $\boldsymbol{v}_j^{(k-1)}$.
7:          Solve $\left(T - \tilde{\lambda}_j I\right)\boldsymbol{v}_j^{(k)} = \boldsymbol{v}_j^{(k-1)}$.
8:          if $|\tilde{\lambda}_j - \tilde{\lambda}_{j-1}| \le 10^{-3}\|T\|$, then
9:              $j_c \leftarrow j - j_1$.
10:             if $j_c = 1$ and $k = 1$, then
11:                 Compute $Y_1 = [\boldsymbol{y}_1]$ and $T_1 = [t_1]$ by using $\boldsymbol{v}_{j_1}$.
12:             end if
13:             $\boldsymbol{u}_{j_c+1} = \left(I - Y_{j_c} T_{j_c}^\top Y_{j_c}^\top\right)\boldsymbol{v}_j^{(k)}$.
14:             Compute $\boldsymbol{y}_{j_c+1}$ and $t_{j_c+1}$ by using $\boldsymbol{u}_{j_c+1}$.
15:             $Y_{j_c+1} = \begin{bmatrix} Y_{j_c} & \boldsymbol{y}_{j_c+1} \end{bmatrix}$, $T_{j_c+1} = \begin{bmatrix} T_{j_c} & -t_{j_c+1} T_{j_c} Y_{j_c}^\top \boldsymbol{y}_{j_c+1} \\ \boldsymbol{0}_{j_c}^\top & t_{j_c+1} \end{bmatrix}$.
16:             $\boldsymbol{v}_j^{(k)} \leftarrow \left(I - Y_{j_c+1} T_{j_c+1} Y_{j_c+1}^\top\right)\boldsymbol{e}_{j_c+1}$.
17:         else
18:             $j_1 \leftarrow j$.
19:         end if
20:     until Some condition is met.
21:     Normalize $\boldsymbol{v}_j^{(k)}$ to $\boldsymbol{v}_j$.
22: end for
```

k that denotes the k-th eigenvalue of the cluster in computing the eigenvector associated with it, but we must know the index for the compact WY orthogonalization when we compute and update T_k and Y_k. To address this, we introduce a variable j_c on line 9. The introduction of j_c enables us to execute the intended code. Further, in the classical inverse iteration algorithm, we need not know the first eigenvalue λ_{j_1} of the cluster, but we must compute $\boldsymbol{y}_1$ and t_1 in the compact WY inverse iteration algorithm. Hence, at the starting point of the computation of the eigenvector associated with the second eigenvalue λ_{j_1+1}, we compute $T_1 = [t_1]$, $Y_1 = [\boldsymbol{y}_1]$ using $\boldsymbol{v}_{j_1}$. Because $\boldsymbol{v}_{j_1}$ is a normalized vector equal to $(I - Y_1 T_1 Y_1^\top)\boldsymbol{e}_1$, we need not compute $\boldsymbol{v}_{j_1}$ again.

5. Numerical Experiments

We now describe some numerical experiments involving DSTEIN, DSTEIN-cWY, and DSTEIN-ncWY on parallel processors, and compare the computation time and orthogonality of computing eigenvectors using each of these codes. Here DSTEIN of LAPACK is based on the classical inverse iteration, and DSTEIN-cWY and DSTEIN-ncWY make use of the compact WY inverse iteration presented in the previous section.

5.1 Details of Numerical Experiments

Our goal is to report the computations of all the eigenvectors associated with eigenvalues of given matrices using DSTEIN, DSTEIN-cWY, and DSTEIN-ncWY on parallel processors, and compare their elapsed time and orthogonality. **Table 2** shows the specifications of all three experimental environments. Note that Environments 2 and 3 are run the same computer but use different compilers and BLAS software. We begin by finding the approximate eigenvalues using LAPACK's code DSTEBZ, which can compute eigenvalues using the bisection method. We then record the elapsed time for DSTEIN, DSTEIN-cWY, and

Table 2 The specification of Environments 1, 2 and 3.

	Environment 1	Environment 2	Environment 3
CPU	AMD Opteron 2.0 GHz 32 cores (8 cores × 4)	Intel Xeon 2.93 GHz 8 cores (4 cores × 2)	Intel Xeon 2.93 GHz 8 cores (4 cores × 2)
RAM	256 GB	32 GB	32 GB
Compiler	gfortran-4.4.5	gfortran-4.4.5	Intel Fortran Compiler 13.0.1
LAPACK	LAPACK-3.3.0	LAPACK-3.3.0	Intel Math Kernel Library
BLAS	GotoBLAS2-1.13	GotoBLAS2-1.13	11.0 update 1

DSTEIN-ncWY using SYSTEM_CLOCK (an internal function of Fortran) and compute the orthogonality criterion of eigenvectors as $\|VV^\top - I\|_\infty$ where $V = \begin{bmatrix}\boldsymbol{v}_1 & \cdots & \boldsymbol{v}_n\end{bmatrix}$ and $\boldsymbol{v}_j$ ($j = 1, \cdots, n$) is an eigenvector computed by each code.

In our experiments, we used two computers equipped with multicore CPUs. As mentioned in Section 2.2, we parallelize the inverse iteration with respect to BLAS operations. For this parallelization, we employed GotoBLAS2 [5] in Environments 1 and 2 and the Intel Math Kernel Library in Environment 3.

For our target matrices, we used symmetric tri-diagonal matrices of three types. Type-1 is a tri-diagonal random matrix, the elements of which were set to random numbers in the range $[0, 1)$. Assuming that the dimension of a random matrix is sufficiently large, we divide the eigenvalues of a tri-diagonal random matrix into clusters using Peters-Wilkinson method [10]. These clusters were typically included 10–100 isolated eigenvalues, the major portion of which composed the largest cluster. The tri-diagonal matrix of Type-2 is defined as follows:

$$T = \begin{bmatrix} 1 & 1 & & & \\ 1 & 1 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & 1 \end{bmatrix}. \tag{5}$$

All the eigenvalues of a Type-2 matrix with large dimensions were included in a single cluster as determined with the Peters-Wilkinson method except in the case where $n = 1{,}050$. Type-3 is a glued-Wilkinson matrix $W_g^\dagger$. $W_g^\dagger$ consists of the block matrix $W_{21}^\dagger \in \mathbb{R}^{21\times 21}$, and the scalar parameter $\delta \in \mathbb{R}$ and is defined as follows:

$$W_g^\dagger = \left[\begin{array}{c|c|c|c} W_{21}^\dagger & \delta & & \\ \hline \delta & W_{21}^\dagger & \delta & \\ \hline & \delta & \ddots \quad \ddots & \\ & & \ddots \quad \ddots & \delta \\ \hline & & \delta & W_{21}^\dagger \end{array}\right], \tag{6}$$

where $W_{21}^\dagger$ is defined by

$$W_{21}^\dagger = \begin{bmatrix} 10 & 1 & & & & \\ 1 & 9 & 1 & & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & 0 & \ddots & \\ & & & \ddots & \ddots & 1 \\ & & & & 1 & 10 \end{bmatrix}, \tag{7}$$

where δ satisfies $0 < \delta < 1$ and is also the semi-diagonal element of $W_g^\dagger$. Since $W_g^\dagger$ is real symmetric tri-diagonal and its semi-diagonal elements are nonzero, all the eigenvalues of $W_g^\dagger$ are real and are divided into 14 clusters of closed eigenvalues. The size of seven of these clusters is $n/21$ and the size of the remaining clusters is $2n/21$. When δ is small, the distance between the minimum and maximum eigenvalues in any cluster is small. In our experiments, we set $\delta = 10^{-4}$. Computing eigenvalues and eigenvectors of the Type-3 (glued-Wilkinson) matrix is one of the benchmark problems of eigenvalue decomposition, and was also used to evaluate the performance of matrix eigenvalue algorithms [2], [4].

5.2 Results and Discussion

5.2.1 Computation Time

Tables 3, **4**, and **5** show the dimension n and the computation time for Environments 1, 2, and 3, respectively. In the tables, n is the dimension of the target matrices and t, t_{cWY}, and t_{ncWY} are the computation times for DSTEIN, DSTEIN-cWY, and DSTEIN-ncWY, respectively. We also introduce a barometer t/t_{ncWY} of the reduction effect using the code DSTEIN-ncWY, which depends on n. **Figures 2**, **3**, and **4** illustrate the results from Tables 3, 4, and 5, respectively.

From Figs. 2 and 3, we can see that both DSTEIN-ncWY and DSTEIN-cWY were faster than DSTEIN for all cases except those for Type-1 matrix at $n = 1{,}050$ in Environments 1 and 2. On the other hand, from Fig. 4, we can see that DSTEIN-ncWY was faster than DSTEIN for all matrix types and DSTEIN-cWY is faster than DSTEIN for Type-1 and Type-3 matrices as n becomes larger in Environment 3.

In addition, comparing Tables 4 and 5, DSTEIN in Environment 3 is faster than in Environment 2. Therefore, when computing eigenvectors using DSTEIN, we can get better performance from the processors equipped in Environments 2 and 3 using Intel Math Kernel library than by GotoBLAS2. We also see that the change from MGS to compact WY orthogonalization in the DSTEIN code for parallel processing results in a significant reduction of computation time. Compared with DSTEIN-ncWY and DSTEIN-cWY, DSTEIN-ncWY is faster than DSTEIN-cWY as n becomes larger for all the test matrices. In particular, from

Table 3 Dimension n of each matrix and the computation time in Environment 1. t, t_{cWY} and t_{ncWY} denotes the computation time for DSTEIN, DSTEIN-cWY and new DSTEIN-cWY, respectively and the unit of them are second. A barometer t/t_{ncWY} is the reduction effect by using the code DSTEIN-ncWY.

(a) Type-1 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.39	0.41	0.41	0.94
2,100	1.76	1.64	1.60	1.10
3,150	5.30	3.77	3.77	1.41
4,200	17.4	7.61	7.85	2.22
5,250	53.6	13.8	13.7	3.90
6,300	157	25.6	25.1	6.22
7,350	996	137	115	8.64
8,400	2,436	381	307	7.93
9,450	4,004	554	449	8.93
10,500	13,231	1,953	1,291	10.25

(b) Type-2 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	1.73	0.46	0.45	3.85
2,100	154	14.2	7.04	21.93
3,150	448	53.2	28.1	15.94
4,200	989	171	94.6	10.45
5,250	1,897	267	167	11.34
6,300	3,281	494	311	10.56
7,350	5,192	739	476	10.92
8,400	7,749	1,268	795	9.74
9,450	10,986	1,596	1,029	10.68
10,500	14,867	2,165	1,389	10.70

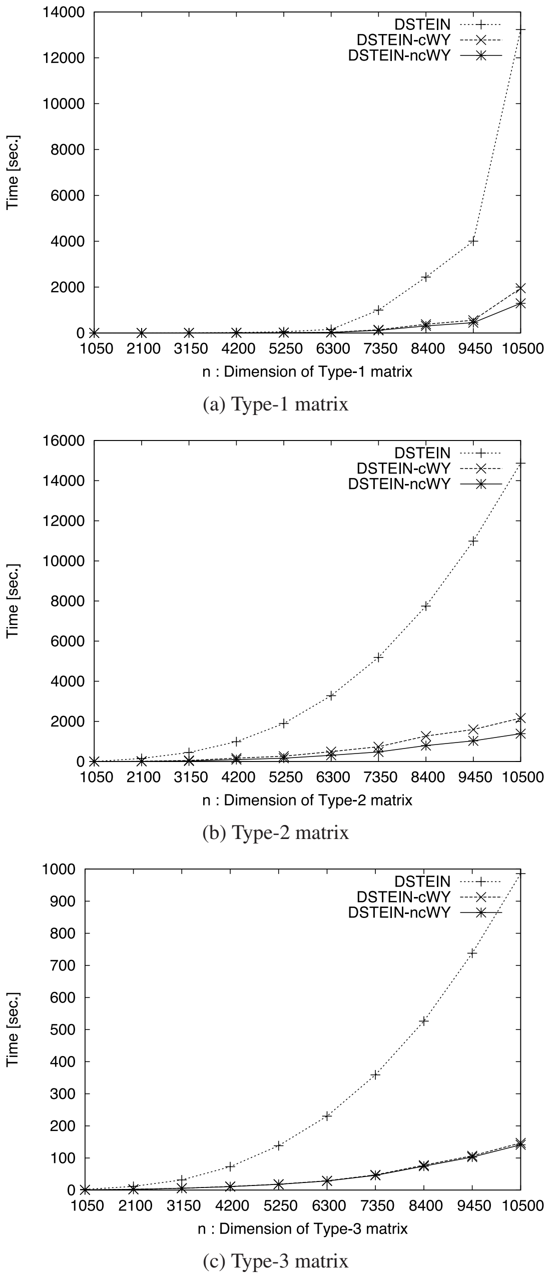
(c) Type-3 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	2.26	0.64	0.62	3.66
2,100	11.5	2.50	2.49	4.62
3,150	31.8	5.85	5.82	5.47
4,200	72.9	11.2	10.9	6.71
5,250	138	18.2	18.1	7.66
6,300	230	29.2	28.4	8.10
7,350	359	47.4	45.9	7.82
8,400	526	77.8	74.5	7.06
9,450	738	107	103	7.18
10,500	986	147	141	6.99

Tables 3 and 4, when the size of the eigenvalue cluster equals the dimension size n, as in Type-2 matrices, DSTEIN-ncWY is about 1.6 times faster than DSTEIN-cWY (using GotoBLAS2). These results are in accordance with the ratio of the computational cost of the compact WY orthogonalization to that of the new algorithm. Thus, the reduction of the computational cost of the compact WY orthogonalization effectively accelerates the re-orthogonalization sub-process of the inverse iteration algorithm.

We now introduce a barometer t/t_{ncWY} for the reduction effect using the code DSTEIN-ncWY which depends on n, the dimension of the target matrix. In Environment 1, the maximum value of $\alpha = t/t_{\mathrm{ncWY}}$ is $\alpha = 10.25$ for $n = 10{,}500$ of Type-1, $\alpha = 21.93$ for $n = 2{,}100$ of Type-2, and $\alpha = 8.10$ for $n = 6{,}300$ of Type-3. In Environment 2, $\alpha = 2.51$ for $n = 5{,}250$ of Type-1, $\alpha = 4.69$ for $n = 2{,}100$ of Type-2, and $\alpha = 3.83$ for $n = 3{,}150$ of Type-3. In Environment 3, $\alpha = 1.51$ for $n = 10{,}500$ of Type-1, $\alpha = 1.51$ for $n = 2{,}100$ of Type-2, and $\alpha = 1.58$ for $n = 4{,}200$ of Type-3. Given these conditions, even if the dimension of the target matrices is larger than that of these examples, we cannot expect that the computation time can be further reduced using DSTEIN-ncWY.

We can see that DSTEIN-cWY and DSTEIN-ncWY are faster than DSTEIN for any dimension n of the target matrix in Environments 1 and 2. As mentioned earlier, according to the theoretical background in Section 3.3, this result shows that the compact WY orthogonalization is an effective algorithm for parallel processing.



(a) Type-1 matrix
(b) Type-2 matrix
(c) Type-3 matrix

Fig. 2 Dimension n of each matrix and the computation time by DSTEIN, DSTEIN-cWY and DSTEIN-ncWY in Environment 1.

The cause of this is related to the time required for floating-point arithmetic and for synchronization in parallel computing. Floating-point computation time increases with the dimension n of the target matrices. By comparison, the synchronization cost does not change significantly even if n becomes larger. Here, in parallel processing, DSTEIN, which contains MGS (for which the number of synchronizations is large), creates a significant bottleneck for the synchronization cost when n is small. This bottleneck gradually lessons as n increases. However, DSTEIN-cWY and DSTEIN-ncWY have a smaller bottleneck for the synchronization cost, because compact WY orthogonalization requires less synchronization, and floating-point computation time becomes greater than that of DSTEIN. This reduction effect can

Table 4 Dimension n of each matrix and the computation time in Environment 2. t, t_{cWY} and t_{ncWY} denotes the computation time for DSTEIN, DSTEIN-cWY and new DSTEIN-cWY, respectively and the unit of them are second. A barometer t/t_{ncWY} is the reduction effect by using the code DSTEIN-ncWY.

(a) Type-1 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.16	0.18	0.18	0.91
2,100	0.75	0.74	0.73	1.02
3,150	2.13	1.72	1.70	1.25
4,200	6.41	3.39	3.42	1.87
5,250	19.2	7.94	7.66	2.51
6,300	58.3	26.0	24.7	2.36
7,350	372	213	179	2.08
8,400	889	539	430	2.06
9,450	1,416	857	703	2.01
10,500	4,357	3,011	1,933	2.25

(b) Type-2 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.52	0.19	0.20	2.67
2,100	57.4	24.9	12.2	4.69
3,150	171	95.8	55.3	3.10
4,200	375	216	136	2.75
5,250	688	440	266	2.58
6,300	1,143	745	462	2.48
7,350	1,774	1,141	723	2.45
8,400	2,570	1,764	1,067	2.41
9,450	3,586	2,416	1,519	2.36
10,500	4,884	3,400	2,070	2.36

(c) Type-3 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.68	0.27	0.27	2.54
2,100	3.58	1.10	1.10	3.27
3,150	10.4	2.87	2.72	3.83
4,200	24.5	6.90	6.59	3.72
5,250	50.1	18.6	16.9	2.97
6,300	86.8	38.4	35.7	2.43
7,350	137	67.1	63.4	2.16
8,400	203	112	103	1.97
9,450	289	157	149	1.94
10,500	393	225	209	1.88

be seen in Tables 3, 4, and 5.

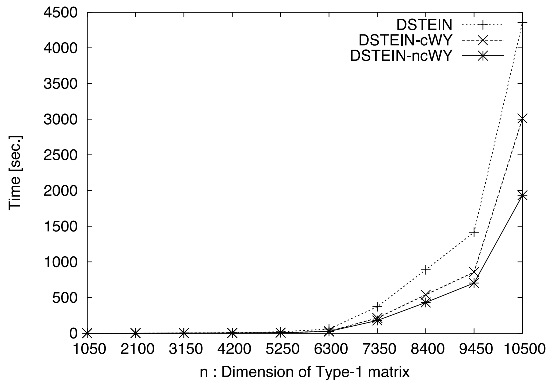
In some cases when n is small, DSTEIN-cWY is faster than DSTEIN-ncWY. This can result from bottlenecks in synchronization. Since DSTEIN-ncWY includes more BLAS operations than DSTEIN-cWY, the computation time of DSTEIN-ncWY is more delayed by synchronization in such cases.

5.2.2 Orthogonality

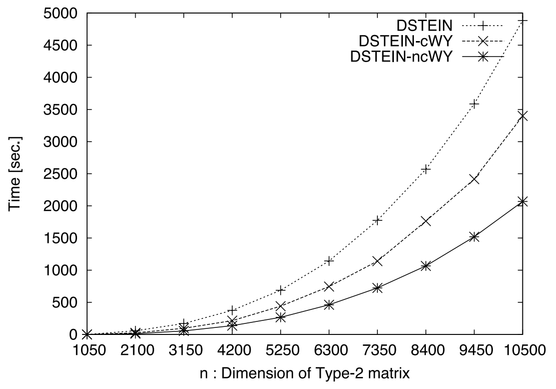
Figure 5 graphs the dimension n and the orthogonality criterion $\|VV^{\top} - I\|_{\infty}$ in Environment 1.

From our experiments, the orthogonality of eigenvectors computed by DSTEIN-ncWY was a little bit better than those computed by DSTEIN-cWY. However, contrary to theoretical analysis, the orthogonality of the eigenvectors calculated by DSTEIN was the best.

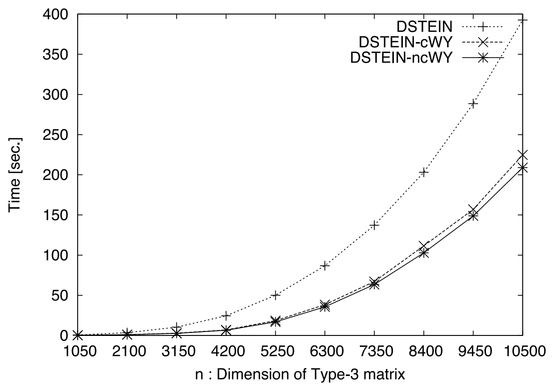
Since the reorthogonalization in the inverse iteration algorithm is performed on each inverse iteration, the condition number of vectors to be orthogonalized may change on each inverse iteration and reorthogonalization. On the other hand, the computational cost of the compact WY orthogonalization is greater than that of the MGS algorithm, as can been seen in Section 3.3, and the computation error generally increases as the computational cost becomes larger. These phenomena occur because the orthogonality computed by DSTEIN-cWY and DSTEIN-ncWY is affected



(a) Type-1 matrix



(b) Type-2 matrix



(c) Type-3 matrix

Fig. 3 Dimension n of each matrix and the computation time by DSTEIN, DSTEIN-cWY and DSTEIN-ncWY in Environment 2.

by computation errors.

6. Expectation Model of Computation Time

In this section, we present expectation models for eigenvector computation time using DSTEIN-ncWY on each computer shown in Table 2.

From the experiment results in the previous section, we can see that the majority eigenvalues of large-dimensional Type-1 matrices end up in the largest single cluster, and all eigenvalues of large-dimensional Type-2 matrices are included in a single cluster. Given this fact, we can expect that computation times on Type-2 matrices will be slowest.

For estimation modeling, we extract many sample data points from the computation time in Environments 1, 2, and 3 and perform multiple regression analysis using the least square method (*Regression* in Data Analysis tools of Microsoft Excel 2010). Let

Table 5 Dimension n of each matrix and the computation time in Environment 3. t, t_{cWY} and t_{ncWY} denotes the computation time for DSTEIN, DSTEIN-cWY and new DSTEIN-cWY, respectively and the unit of them are second. A barometer t/t_{ncWY} is the reduction effect by using the code DSTEIN-ncWY.

(a) Type-1 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.13	0.16	0.16	0.83
2,100	0.57	0.78	0.78	0.74
3,150	1.54	1.84	1.86	0.83
4,200	4.31	3.92	3.95	1.09
5,250	12.3	8.80	8.77	1.40
6,300	36.1	25.6	24.8	1.45
7,350	227	204	174	1.30
8,400	546	493	411	1.33
9,450	882	796	661	1.33
10,500	2,818	2,745	1,861	1.51

(b) Type-2 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.17	0.17	0.18	0.97
2,100	21.8	24.2	14.4	1.51
3,150	82.4	91.1	57.9	1.42
4,200	196	223	145	1.35
5,250	383	407	272	1.41
6,300	661	704	474	1.39
7,350	1,051	1,112	740	1.42
8,400	1,568	1,662	1,101	1.42
9,450	2,237	2,324	1,537	1.46
10,500	3,081	3,162	2,095	1.47

(c) Type-3 matrix

	t	t_{cWY}	t_{ncWY}	t/t_{ncWY}
1,050	0.21	0.25	0.25	0.84
2,100	1.45	1.32	1.33	1.09
3,150	5.67	3.85	3.89	1.46
4,200	14.0	9.05	8.90	1.58
5,250	28.8	18.9	18.3	1.57
6,300	51.9	36.4	34.9	1.49
7,350	84.3	61.8	59.2	1.42
8,400	127	97.9	94.5	1.35
9,450	182	142	137	1.33
10,500	250	195	188	1.33

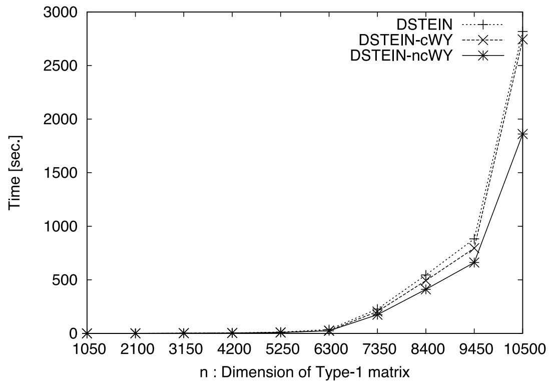
$m \times 10^3$ be the number of required eigenvectors and $n \times 10^3$ be a dimension of a Type-2 matrix, and let the computation time depend on m and n. For sample data, the combination of m and n is used so as not to reduce the computation time through cache use: $3 \leq m \leq n \leq 11$ in Environment 1, $2 \leq m \leq n \leq 10$ in Environments 2 and 3.

By this approach, we construct our model from the function $\bar{t}_i = a_i m^3 + b_i m^2 n + c_i mn + d_i$ $(i = 1, 2, 3)$, since the total computational cost of the compact WY orthogonalization is $4m^2n - m^3$ and that of the inverse iteration is on the order of mn. Here, $\bar{t}_1$, $\bar{t}_2$, and $\bar{t}_3$ correspond to the expectation models in Environments 1, 2, and 3, respectively.

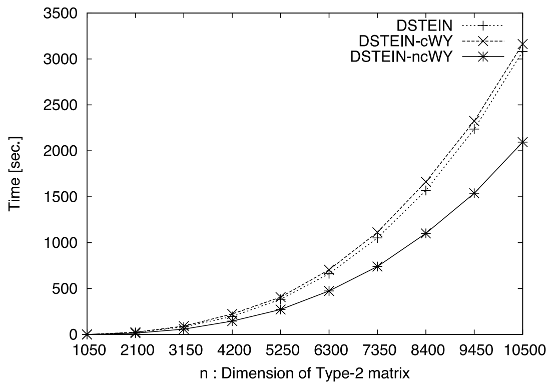
From the above analysis, we get the following function for our expectation models:

$$(a_1, b_1, c_1, d_1) = (-0.183, 1.36, 0.714, -10.2),$$
$$(a_2, b_2, c_2, d_2) = (-0.351, 1.81, 0.680, -6.03),$$
$$(a_3, b_3, c_3, d_3) = (-0.211, 1.94, 1.86, -15.5).$$

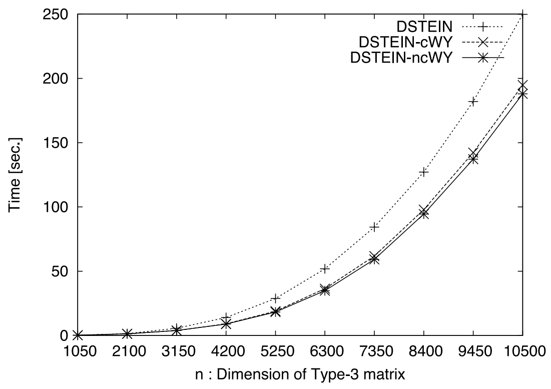
Figures 6, **7**, and **8** show the expectation model and sample data in Environments 1, 2, and 3, respectively. Here, b_1, b_2, and b_3 are coefficients of m^2n, on which the models greatly depend. In models given above, we can see $b_1 < b_2 < b_3$, which corresponds to the fact that Environment 1 has a more parallel configuration than Environments 2 and 3. In addition, as a result of the theoretical analysis of computational cost, the sign of coefficients a_1, a_2, and a_3 is minus.



(a) Type-1 matrix



(b) Type-2 matrix



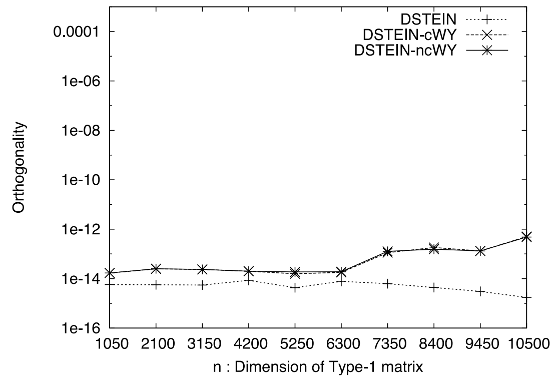
(c) Type-3 matrix

Fig. 4 Dimension n of each matrix and the computation time by DSTEIN, DSTEIN-cWY and DSTEIN-ncWY in Environment 3.

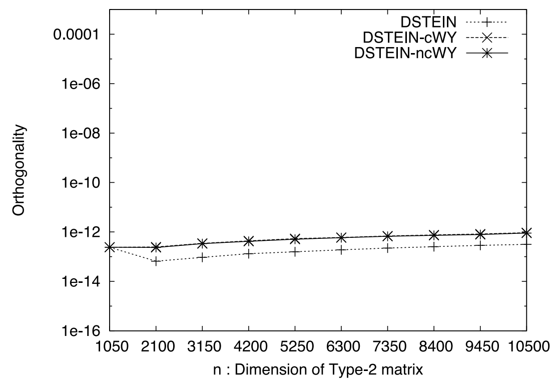
7. Conclusions

In this study, we presented a new inverse iteration algorithm for computing all the eigenvectors of a real symmetric tri-diagonal matrix. The new algorithm is equipped with our new implementation of the compact WY orthogonalization algorithm.

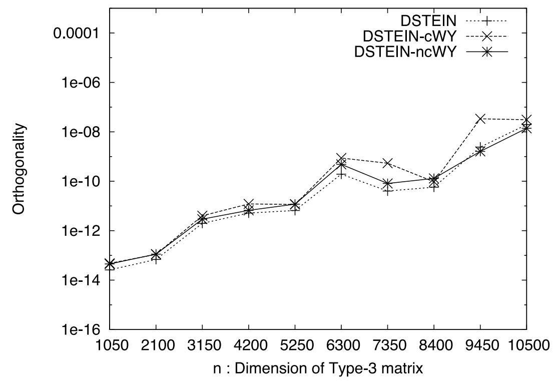
We tested the performance of two types of inverse iteration algorithm by computing eigenvectors for certain real symmetric tri-diagonal matrices with several thousand dimensions. The results of these experiments showed that the compact WY inverse iteration is more efficient than classical inverse iteration owing to parallel processing. As the number of cores of the CPU increases, so



(a) Type-1 matrix



(b) Type-2 matrix



(c) Type-3 matrix

Fig. 5 Dimension n of each matrix and the orthogonality $\|VV^\top - I\|_\infty$ by DSTEIN, DSTEIN-cWY and DSTEIN-ncWY in Environment 1.

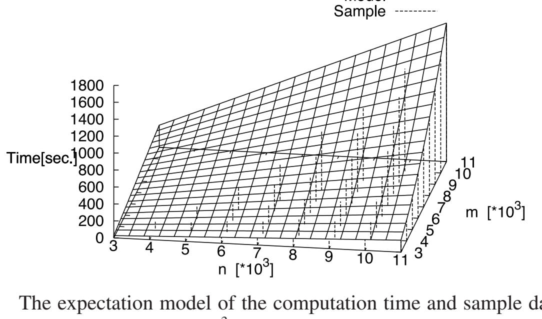


Fig. 6 The expectation model of the computation time and sample data in Environment 1. $m \times 10^3$ is a number of required eigenvectors and $n \times 10^3$ is a dimension of Type-2 matrices.

does parallelization efficiency.

We expect that our new inverse iteration algorithm can be applied to other types of matrix eigenvector problems, such as eigenvectors of a real symmetric band matrix, or singular vectors

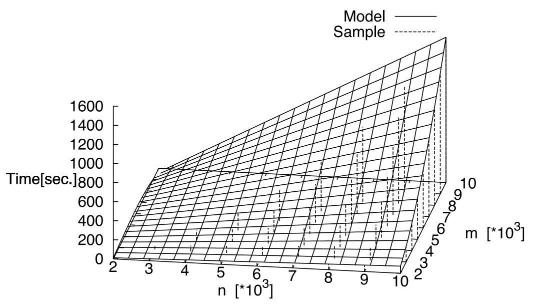


Fig. 7 The expectation model of the computation time and sample data in Environment 2. $m \times 10^3$ is a number of required eigenvectors and $n \times 10^3$ is a dimension of Type-2 matrices.

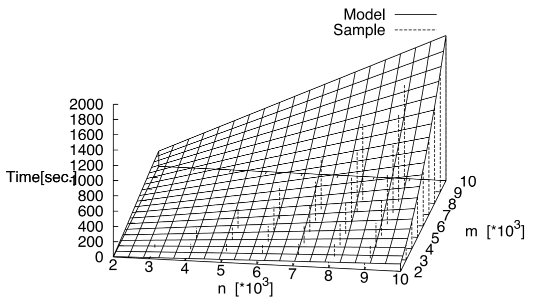


Fig. 8 The expectation model of the computation time and sample data in Environment 3. $m \times 10^3$ is a number of required eigenvectors and $n \times 10^3$ is a dimension of Type-2 matrices.

of a bidiagonal matrix.

While we have shown that the proposed algorithm is effective on shared memory computers, we do not expect it to be effective on distributed memory parallel systems. This is because this algorithm is parallelized with respect to BLAS operations, which do not map efficiently to distributed systems. If we implement the compact WY orthogonalization algorithm to the reorthogonalization process of the block inverse iteration algorithm, the resulting algorithm could be effective in distributed environments, as it would be based on BLAS level-3 operations. This application will be considered in a future work.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number 24360038. The authors would like to express our gratitude to reviewers of this paper for their insightful comments and thank Professor Yusaku Yamamoto of Kobe University for providing several helpful suggestions.

References

[1] Demmel, J., Grigori, L., Hoemmen, M.F. and Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations, *SIAM J. Sci. Comput.*, Vol.34, No.1, pp.A206–A239 (2012).

[2] Demmel, J.W., Marques, O.A., Parlett, B.N. and Vömel, C.: Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers, *SIAM J. Sci. Comput.*, Vol.30, No.3, pp.1508–1526 (2008).

[3] Dhillon, I.S.: A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem, PhD Thesis, EECS Department, University of California, Berkeley (1997).

[4] Dhillon, I.S., Parlett, B.N. and Vömel, C.: Glued matrices and the MRRR algorithm, *SIAM J. Sci. Comput.*, Vol.27, No.2, pp.496–510 (2005).

[5] GotoBLAS2: (online), available from ⟨http://www.tacc.utexas.edu/tacc-projects/gotoblas2/⟩.

[6] Imamura, T., Yamada, S. and Machida, M.: Development of a high performance eigensolver on the peta-scale next generation supercomputer system, *Prog. Nuclear Science and Technology*, Vol.2, pp.643–650 (2011).

[7] Ishigami, H., Kimura, K. and Nakamura, Y.: Implementation and performance evaluation of new inverse iteration algorithm with

Householder transformation in terms of the compact WY representation, *Proc. 2011 International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA2011*), Vol.II, pp.775–780 (2011).

[8] Katagiri, T. and Itoh, S.: A massively parallel dense symmetric eigensolver with communication splitting multicasting algorithm, *High Performance Computing for Computational Science VECPAR 2010*, Vol.6449, pp.139–150 (2011).

[9] LAPACK: (online), available from ⟨http://www.netlib.org/lapack/⟩.

[10] Peters, G. and Wilkinson, J.: *The calculation of specified eigenvectors by inverse iteration*, Handbook for Automatic Computation, pp.418–439, Springer-Verlag, Berlin (1971).

[11] Schreiber, R. and van Loan, C.: A storage-efficient WY representation for products of Householder transformations, *SIAM J. Sci. Stat. Comput.*, Vol.10, No.1, pp.53–57 (1989).

[12] Walker, H.: Implementation of the GMRES method using Householder transformations, *SIAM J. Sci. Stat. Comput.*, Vol.9, No.1, pp.152–163 (1988).

[13] Yamamoto, Y. and Hirota, Y.: A parallel algorithm for incremental orthogonalization based on the compact WY representation, *JSIAM Letters*, Vol.3, pp.89–92 (2011).

Hiroyuki Ishigami received his B.E. and M.I. degrees from Kyoto University in 2011 and 2013. Since 2013, he has been a doctoral course student at Kyoto University and a research fellow (DC1) of JSPS. He is a student member of IPSJ and JSIAM.

Kinji Kimura received his Ph.D. degree from Kobe University in 2004. He became a PRESTO, COE, and CREST researcher in 2004 and 2005. He became an assistant professor at Kyoto University in 2006, an assistant professor at Niigata University in 2007, a lecturer at Kyoto University in 2008, and has been a program-specific associate professor at Kyoto University since 2009. He is an IPSJ member.

Yoshimasa Nakamura has been a professor of Graduate School of Informatics, Kyoto University from 2001. His research interests include integrable dynamical systems which originally appear in classical mechanics. But integrable systems have a rich mathematical structure. His recent subject is to design new numerical algorithms such as the mdLVs and I-SVD for singular value decomposition by using discrete-time integrable systems. He is a member of JSIAM, SIAM, MSJ and AMS.