universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

"On Implementation and Evaluation of a Divide & Conquer Eigensolver for the Real Bandsymmetric Eigenproblem and Comparison to Current Methods"

verfasst von / submitted by

Kastor Maximilian Felsner, B.Sc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna, 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 940
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Scientific Computing
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, M.Sc.

Contents

Abstract

As the architecture of modern computers continues to evolve, numerical software has to adapt. To make use of the ever increasing computational power parallelism needs to be exploited efficiently. For the specific case of a symmetric eigenproblem it was found that a new approach for the tridiagonal reduction is needed. This new approach first reduces the full problem S to banded form B and later tridiagonalises B.

Arbenz [4] generalised the idea behind *Cuppen's divide and conquer algorithm* and proposed a new method which directly solves the bandsymmetric eigenproblem B. This creates the possibility of omitting the computationally expensive second reduction stage when solving a full symmetric eigenproblem S. Unfortunately, Arbenz [4] reported numerical instabilities which lead to insufficiently accurate eigenvectors.

This thesis evaluates the approach of computing the eigendecomposition of S by using a full-to-band reduction followed by a bandsymmetric eigensolver. In a first step the algorithm described in [4] is implemented and the reported results are reproduced. Later the numerical inaccuracies of the algorithm are analysed and an improved version is proposed. The final algorithm is then compared to state-of-the-art eigensolvers which include both bandsymmetric ones as well as those available in `LAPACK` which use a full-to-tridiagonal reduction.

It is found that the algorithm has a promising runtime complexity, however further improvements are needed to guarantee orthogonal eigenvectors for problems with tightly clustered eigenvalues.

Zusammenfassung

Die fortlaufende Weiterentwicklung moderner Computerarchitekturen fordert eine stetige Anpassung numerischer Software. Um die Rechenleistung heutiger Systeme auszunutzen müssen Algorithmen ein hohes Maß an Parallelität aufweisen. Für den spezifischen Fall eines symmetrischen Eigenwertproblems wurde festgestellt, dass eine neue Methode für die Tridiagonalisierung der ursprünglichen dicht besetzten Matrix S erforderlich ist, um parallele Systeme voll auszulasten. Dieser neue Ansatz bringt S in einem ersten Schritt durch eine Ähnlichkeitstransformation auf bandsymmetrische Form B und tridiagonalisiert B anschließend.

Arbenz [4] verallgemeinerte die Idee hinter *Cuppens divide and conquer* Algorithmus und entwickelte so eine neue Methode, welche das bandsymmetrische Eigenwertproblem B direkt löst. Dies ermöglicht es, die zeitintensive Tridiagonalisierung der Bandmatrix B zu vermeiden. Bedauerlicherweise berichtete Arbenz [4] von numerischen Instabilitäten, die in ungenauen Eigenvektoren resultierten.

Die vorliegende Arbeit evaluiert den Ansatz, die Spektralzerlegung einer dicht besetzten Matrix S durch Reduktion zu einer Bandmatrix B gefolgt von der Berechnung der Eigenwertzerlegung von B zu lösen. Hierfür werden in einem ersten Schritt der in [4] beschriebene Algorithmus implementiert und die Laufzeit optimiert. Später werden die numerischen Ungenauigkeiten des Algorithmus analysiert und eine verbesserte Version vorgestellt. Der finale Algorithmus wird abschließend mit modernen Methoden verglichen. Dieser Vergleich umfasst sowohl bandsymmetrische Verfahren, als auch die in `LAPACK` verfügbaren, die eine herkömmliche Tridiagonalisierung durchführen.

Es zeigt sich, dass der Algorithmus eine vielversprechende Laufzeit-Komplexität aufweist, jedoch weitere Verbesserungen vonnöten sind, um orthogonale Eigenvektoren auch für Probleme mit eng geclusterten Eigenwerten garantieren zu können.

1. Introduction

1.1. Problem Statement (Motivation and Goal)

This thesis is concerned with the real symmetric eigenproblem $Sx = \lambda x$, with $S \in \mathbb{R}^{n\times n}$ being symmetric, $\lambda \in \mathbb{R}$ an eigenvalue and $x \in \mathbb{R}^n$ the corresponding eigenvector. More precisely with the computation of the spectral decomposition $S = Q\Lambda Q^\top$ for large n, where $\Lambda \in \mathbb{R}^{n\times n}$ is a diagonal matrix containing the eigenvalues of S and $Q \in \mathbb{R}^{n\times n}$ is orthonormal.

The traditional approach of computing all eigenpairs of S consists of three steps: First S is reduced to tridiagonal form $T = V^\top SV$.[1] In a second step the tridiagonal eigenproblem is solved, yielding the eigendecomposition of the tridiagonal problem $T = \bar{Q}\Lambda\bar{Q}^\top$. Finally the spectral decomposition of the original problem presents itself as $S = (V\bar{Q})\Lambda(V\bar{Q})^\top$ and hence a matrix-matrix multiplication gives the matrix $Q = V\bar{Q}$ whose i-th column q_i is the eigenvector corresponding to the i-th eigenvalue $\lambda_i \in \sigma(S)$.

It turns out that this approach cannot utilise the full potential of modern architectures, like hybrid GPU-CPU systems or multi-core architectures. Haidar, Ltaief and Dongarra [34] showed that the two stage tridiagonalisation process developed by Bischof, Sun and Lang [9] is more efficient on such parallel machines. The new approach first reduces the dense matrix S to banded form

$$B = \begin{pmatrix} b_{00} & \cdots & b_{0r} & & & & \\ \vdots & \ddots & & \ddots & & & \\ b_{0r} & & \ddots & & \ddots & & \\ & \ddots & & \ddots & & b_{n-r-1,n-1} & \\ & & \ddots & & \ddots & \vdots & \\ & & & b_{n-r-1,n-1} & \cdots & b_{n-1,n-1} & \end{pmatrix} \in \mathbb{R}^{n\times n}$$

and later tridiagonalises B, see fig. 1.1. The first stage ($S \mapsto V_1 B V_1^\top$) is compute bound

[1] Notation: The matrices V and Q are both orthonormal. V is used to denote a change of basis due to a reduction process, whereas Q is used for the spectral decomposition of a matrix.

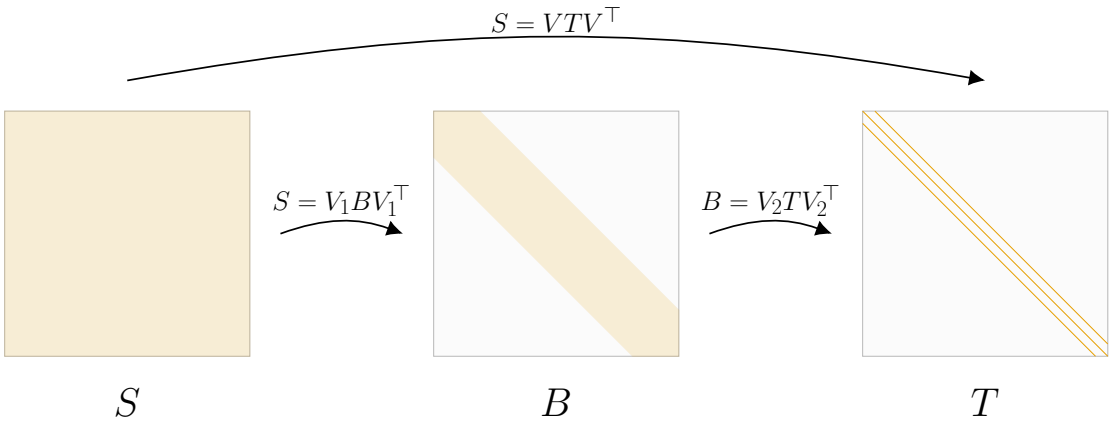


Figure 1.1.: The traditional one step reduction $S \mapsto VTV^\top$ versus the two stage tridiagonalisation $S \mapsto V_1BV_1^\top \mapsto V_1V_2TV_2^\top V_1^\top$. S is a dense symmetric matrix, B is bandsymmetric and T a symmetric tridiagonal matrix. The transformation matrices V and V_i are orthonormal. By using a bandsymmetric eigensolver instead of a tridiagonal one both the band-to-tridiagonal reduction and the correspond back-transformation are no long needed.

and can thus be run efficiently in parallel, however the second stage ($B \mapsto V_2TV_2^\top$) is memory/communication bound.[2]

Arbenz [4] proposed three divide and conquer approaches for the banded symmetric eigenproblem. As these algorithms operate directly on the banded structure they could yield a significant speedup by avoiding the memory bound second stage of the tridiagonal reduction process. Its important to note that not only the band-to-tridiagonal reduction can be omitted but also the first back-transformation. The algorithms in [4] are generalisations of *Cuppen's divide and conquer algorithm* [17], using a rank-r instead of a rank-*one* tear to subdivide the problem. One of these methods is the so called *divide and conquer by modification* algorithm which is well studied, see [26], [28], [27], [33], however, the runtime complexity is highly sensitive to the bandwidth. Arbenz [4] implemented a second approach, called *divide and conquer by extension*, with a promising runtime complexity. Unfortunately, numerical instabilities which lead to inaccurate eigenvectors have been reported in [4].

This thesis will re-evaluate the *divide and conquer approach by extension* algorithm in comparison to modern eigensolvers. Note that the most efficient algorithms available in `LAPACK` have been developed after 1992, and thus the comparison in [4] is outdated. Chapter 5 will focus on creating an efficient implementation of the *divide and conquer by*

[2] The routine implemented in `PLASMA` has a memory bound second stage [34]. Ballard, Demmel and Knight [6] reported better data reuse properties of the second stage for matrices with small bandwidths.

extension method, later denoted as `dsbkdc`. The algorithm is expected to perform well on parallel machines, however, such an implementation is out of the scope of this thesis. Chapter 6 is concerned with analysing the numerical instabilities that occur during the computation of the eigenvectors. In section 6.4 an improved version using extended precision is proposed.

Finally the state-of-the-art eigensolvers available in `LAPACK` which use a full-to-tridiagonal reduction ($S \mapsto VTV^\top$, see fig. 1.1) are compared to methods which only use a full-to-band reduction process ($S \mapsto V_1 B V_1^\top$). The second approach uses a bandsymmetric eigensolver to compute the spectral decomposition of B, whereas `LAPACK` solves the tridiagonal problem T. The comparison will includes the newly implemented *divide and conquer by extension* algorithm (`dsbkdc`). Test data for the numerical evaluation is adapted from Demmel, Marques, Parlett *et al.* [19] which provides for a fair comparison.

1.2. Document Structure

The thesis starts by introducing the real symmetric eigenproblem in chapter 2. The most important mathematical properties are covered in section 2.1, e.g. the condition of the eigenproblem. After a short discussion of the method of *inverse iteration* for a dense matrix S in section 2.2.1, section 2.3 deals with the traditional tridiagonal reduction. Both *Householder transformations* and *Givens rotations* are presented and their respective runtime complexities given. Section 2.3.2 provides a short overview for the two stage tridiagonal reduction, which is essential to exploit parallelism. The rest of the chapter is concerned with the computation of the spectral decomposition of a tridiagonal matrix T. The main tridiagonal eigensolvers available in `LAPACK` are presented in section 2.4. A comparison of these methods in section 2.4.4 concludes the chapter.

Chapter 3 focuses on the bandsymmetric eigenvalue problem which is a subproblem of the symmetric eigenproblem. In total three algorithms to compute the eigendecomposition of a bandsymmetric matrix B are discussed. Section 3.1 starts with the method of *inverse iteration* for banded matrices. Afterwards the divide and conquer approaches proposed by Arbenz [4] are discussed. Section 3.2.2 gives an in depth introduction to the *divide and conquer by extension* method. This algorithm is later implemented in chapter 5 and chapter 6. For the *divide and conquer by modification* method a short overview of the most important literature is given in section 3.2.3.

For the implementation of `dsbkdc` in chapter 5 and chapter 6 and for the numerically evaluates in chapter 7 a fair testing setup is needed. The main characteristics of the system are listed in chapter 4. Both the used hardware as well as the testing program

and compiler settings are covered. Section 4.2 explains the process of generating test data.

Chapter 5 covers the implementation and performance optimisation of `dsbkdc`. After the presentation of the baseline implementation in section 5.1 each section discusses a separate optimisation. The improvements are illustrating by plotting the runtime, e.g. by showing the speedup compared to the baseline implementation. Section 5.2 implements the superlinear root-finding method *zeroin* to speed up *bisection*. The largest speedup is achieved in section 5.3 by blocking the back-transformation of the eigenvectors. This allows for the use of level 3 `BLAS` and hence reduces the impact of the n^3 term in the runtime complexity. The observed complexity (up to $n = 10200$) is below $\mathcal{O}(n^3)$. The storage scheme is changed in section 5.4 to a packed format. Instead of storing B as a dense matrix only the block tridiagonal part is saved. Section 5.5 further improves the root-finding method. The chapter is concluded with the introduction of recursion in section 5.6, which preserves the complexity for small bandwidths.

The numerical instabilities are analysed in chapter 6 and certain improvements are proposed. Most insight is gained from section 6.2 which correlates the numerical instabilities when calculating the eigenvectors to insufficiently accurate eigenvalues. The problem is similar to the inaccuracies occurring in early implementations of *Cuppen's divide and conquer algorithm*, see section 2.4.2.4. Section 6.4 takes advantage of this observation by using extended precision to determine the eigenvalues to higher precision if needed. The approach of using *inverse iteration* to solve numerical inaccuracies, as taken by Arbenz [4], is found to be not competitive, see section 6.3. The rest of the chapter covers hard test cases with tightly clustered eigenvalues for which `dsbkdc` fails to guarantee orthogonal eigenvectors.

Chapter 7 compares bandsymmetric eigensolvers to the traditional approach of using a tridiagonal reduction. The test data are banded matrices, however, the results of this evaluation can be generalised to the dense symmetric eigenproblem, as the two-stage reduction scheme yields a banded matrix as interim result, see Haidar, Ltaief and Dongarra [34]. The final implementations[3] of the *divide and conquer by extension* method are part of this comparison.

Chapter 8 summarises the main results. Appendix A lists additional measurements.

[3] Three version of `dsbkdc` are evaluated. Two using only double-precision one with and one without recursion. The third implementation uses extended precision to improve the accuracy.

1.3. Related Work

The related work can roughly be subdivided into three groups, covering:

(a) tridiagonal eigensolver.

(b) the tridiagonal reduction.

(c) solver for the bandsymmetric eigenproblem.

The symmetric eigenproblem is discussed in most textbooks covering numerical linear algebra. For this thesis [36], [30], [20] and [48] were consulted. They also discuss the foundation of group (a) and group (b).

1.3.1. Tridiagonal Eigensolver

Group (a) is concerned with the computation of the spectral decomposition of a tridiagonal matrix. Jessup and Ipsen [38] discusses the traditional *inverse iteration* giving both stopping and re-orthogonalisation criteria that are used in section 3.1.

Cuppen [17] proposed the *divide and conquer method* based on the work of Golub [29] and Bunch, Nielsen and Sorensen [15] for the rank-*one* modification. The numerical stability and scalability of the proposed algorithm were analysed by Sorensen and Tang [47] and Dongarra and Sorensen [23] respectively. Later Gu and Eisenstat [31] [32] proposed an elegant way of computing the eigenvectors to high precision. Parallel implementations of *Cuppen's divide and conquer algorithm* are discussed in [50], [12], [51] and [43].

The second important tridiagonal eigensolver is the *multiple relatively robust representation method* proposed by Dhillon and Parlett [21], [8], [22]. Marques, Vömel, Demmel *et al.* [40] [19] provide an in depth comparison of the most important currently available tridiagonal eigensolvers in `LAPACK`.

1.3.2. Tridiagonal Reduction

Group (b) deals with the tridiagonal reduction. The traditional one step tridiagonalisation is covered by the textbooks mentioned above. The bulge chasing procedure for the band-to-tridiagonal reduction using *Givens rotations* is described in Schwarz [46]. Davis and Rajamanickam [18] [45] presented the library `PIRO_BAND` that optimises this routine by grouping successive plane rotations and apply them in a blocked fashion.

The rest of group (b) is primarily concerned with the two-stage tridiagonalisation process, which was introduced by Bischof, Lang and Sun [10] [11]. Ltaief, Luszczek and

Dongarra [39] adopted this two stage approach with a new implementation using a tile layout and an improved second stage. This method is available in `PLASMA` and was refined by Haidar, Ltaief and Dongarra [34]. Haidar, Tomov, Dongarra *et al.* [35] showed that such an two-stage reduction process is essential to fully exploit parallelism on modern computer architectures. A communication avoiding algorithm for the band-to-tridiagonal reduction was presented in Ballard, Demmel and Knight [6]. They reported a high level of data reuse for small bandwidths. Unfortunately an implementation is not yet available.

1.3.3. Bandsymmetric Eigensolver

Finally group (c) discusses methods that compute the spectral decomposition of a banded matrix without a tridiagonal reduction. Beattie and Fox [7] reviewed different low rank modifications of a banded matrix with known eigendecomposition and presented a spectrum slicing method. Similar was developed by Arbenz, Gander and Golub [5] [3] who also outlined a method to compute the eigenvectors for the modified eigenvalue problem. Arbenz [3] applied this theory to bandsymmetric Teoplitz matrices. Arbenz [4] summarised the theory of low rank modification for the bandsymmetric eigenproblem and proposed three divide and conquer algorithms. They furthermore provided experimental results for the *divide and conquer by extension* approach.

Gansterer, Ward and Muller [28] refined the second algorithm proposed in [4] and presented an implementation for block tridiagonal matrices with rank-one off-diagonal blocks. Later Gansterer, Ward, Muller *et al.* [27] generalised the algorithm to arbitrary off-diagonal blocks. Their implementation uses r rank-one update instead of one rank-r as done by [4]. This approach can be seen as a direct generalisation of *Cuppen's divide and conquer* algorithm. For small bandwidth the runtime complexity is promising, however, as r increases tridiagonalisation based algorithms become superior [41]. Haidar, Ltaief and Dongarra [33] parallelised this method using a tile algorithm.

2. The Symmetric Eigenproblem

This chapter will give an overview of the real symmetric eigenproblem $Sx = \lambda x$, with $S \in \mathbb{R}^{n \times n}$ being symmetric, $\lambda \in \mathbb{R}$ an eigenvalue and $x \in \mathbb{R}^n$ the corresponding eigenvector. Section 2.1 starts by summarising the main mathematical properties. In section 2.3 the state-of-the-art approach for solving symmetric eigenproblems, as applied in numerical libraries such as `LAPACK` [2] or `MAGMA` [1], is presented. Section 2.4 concludes the chapter with a discussion of tridiagonal eigensolvers. Most notably *Cuppens divide and conquer algorithm* is introduced in section 2.4.2, which is a predecessor of the bandsymmetric counterpart proposed by Arbenz [4], see section 3.2.

The theory of symmetric eigenproblems is well studied and part of most textbooks covering numerical linear algebra. The content of this chapter is based on standard literature (see [36], [30], [20], [48]) with the exception of section 2.3.2, as this is still an active area of research. The theorems included are each adopted from one of these books. Additional citations point to corresponding proofs.

2.1. Properties

The eigenvalues of a matrix A are given by the zeros of the characteristic polynomial. The *Frobenius companion matrix*

$$F = \begin{pmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & -a_{n-1} \end{pmatrix} \tag{2.1}$$

has the characteristic polynomial

$$p(\lambda) = \lambda^n + a_{n-1}\lambda^{n-1} + a_{n-2}\lambda^{n-2} + \cdots a_1\lambda + a_0. \tag{2.2}$$

It is thus clear that the eigenvalue problem is hard. The real *symmetric* eigenproblem on the other hand has many useful mathematical properties which make it easier. First and foremost all eigenvalues are real. To see this consider $\lambda \in \sigma(S)$, then

$$\lambda = x^*Sx = x^*S^*x = \overline{x^*Sx} = \overline{\lambda}$$

holds, with $x \in \mathbb{C}^n$ being the corresponding eigenvector, and thus $\lambda \in \mathbb{R}$.

Furthermore the algebraic and geometric multiplicities of all eigenvalues match which yields the following theorem (see [30, p. 440]).

Theorem 2.1.1 (Spectral decomposition)**.** *If $S \in \mathbb{R}^{n\times n}$ is symmetric, then there exists a real orthogonal $Q \in \mathbb{R}^{n\times n}$ such that*

$$Q^\top SQ = \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n). \tag{2.3}$$

Moreover, for $i = 1, 2, \ldots, n$, $Sq_i = \lambda_i q_i$, with q_i being the i-th column of Q.

As already mentioned, this thesis is mainly concerned with computing the complete spectral decomposition $Q^\top SQ$ which is represented in the algorithms discussed. If only specific eigenpairs are of interest other algorithms perform better.

2.1.1. Condition of the Eigenproblem

To determine the condition of the eigenproblem both the sensitivity of an eigenvalue as well as of the corresponding eigenvector are of interest. The next corollary is a special case of the theorem from Bauer-Frike (see [36, p. 214]) and shows that a small perturbation in the original matrix only slightly changes the eigenvalues of the perturbed matrix.

Corollary 2.1.1.1. *If S and $S + E$ are $n \times n$ symmetric matrices, then*

$$|\lambda_i(S + E) - \lambda_i(S)| \leq ||E||_2 \tag{2.4}$$

for $i = 1, 2, \ldots, n$.

The prerequisite that S is symmetric is essential as a perturbation of order ε in a non-symmetric matrix A can introduce a perturbation of order $\varepsilon^{1/n}$ in the eigenvalues of A [48, p. 38]. For any backward stable algorithm[1] the error can thus be bound by

[1] A backwards stable algorithm $\tilde{f}$ computes $\tilde{f}(x)$ such that $\exists \Delta x : \tilde{f}(x) = f(x+\Delta x)$ with $|(\Delta x)|/|x| \leq C_R\epsilon$ where C_R is a small constant and f the exact algorithm in the absence of numerical errors [36, p. 20].

$|\lambda_i - \hat{\lambda}_i| \leq ||E||_2 = \mathcal{O}(\epsilon)||S||_2$ where $\hat{\lambda}_i$ are the computed and λ_i the exact eigenvalues of S.[2]

In the next theorem (compare with [36, p. 215]) the sensitivity of the corresponding eigenvectors is analysed.

Theorem 2.1.2. *Let $S \in \mathbb{R}^{n\times n}$ be a symmetric matrix, $\lambda \in \mathbb{R}$ and $x \in \mathbb{R}^n$ a vector with*

$$||Sx - \lambda x||_2 = \varepsilon||x||_2, \qquad \varepsilon > 0, \tag{2.5}$$

then $\exists\hat{\lambda} \in \sigma(S)$ such that

$$|\lambda - \hat{\lambda}| = \min_{\mu\in\sigma(S)} |\lambda - \mu| \leq \varepsilon \tag{2.6}$$

holds. If $\hat{x}$ is the eigenvector corresponding to $\hat{\lambda}$ and γ the distance to the next eigenvalue $\mu \neq \hat{\lambda},\ \mu \in \sigma(S)$, then

$$|\sin\theta| \leq \varepsilon/\gamma, \qquad \gamma = \min_{\substack{\mu\in\sigma(S)\\ \mu\neq\hat{\lambda}}} |\lambda - \mu| \tag{2.7}$$

follows, with θ being the angle between x and $\hat{x}$.

One can see that the computation of the eigenvectors is more sensitive to perturbations in the original matrix than the computation of the corresponding eigenvalues. The closer two eigenvalues are the harder the computation of an orthogonal and invariant eigenspace becomes. Hence the distribution of the eigenvalue of a matrix has a large impact on the hardness of the problem. One can think of $1/\gamma$, as defined in eq. (2.7), as a condition number for the sensitivity of the corresponding eigenvector. Theorem 2.1.2 can be generalised to invariant subspaces of S, see [30, p. 445].

Furthermore theorem 2.1.2 shows that a small absolute residual $||Sx - \lambda x||_2$ is not sufficient to get orthogonal eigenvectors.

2.2. Direct Methods

In this section a first method to compute an eigenpair will be introduced. It is one of the simplest algorithms, however, it is still useful if only selected eigenpairs are required. Other direct methods which lead to an eigendecomposition are e.g. the *QR algorithm* or the *Jacobi method*. Instead of a direct approach modern numerical libraries like `LAPACK` or

[2] ϵ denotes the *machine epsilon* or *unit roundoff*. For double precision the unit roundoff is $\epsilon = 2^{-52} \approx 2.2 \cdot 10^{-16}$ [42, p. 23].

MAGMA use a tridiagonalisation process in combination with a highly efficient tridiagonal eigensolver. This newer approach will be discussed in section 2.3 and section 2.4.

2.2.1. Inverse Iteration

Algorithm 1 describes a method to approximate the largest eigenvalue of a given matrix S, by applying a randomly initialised start vector $q^{(0)}$ multiple times to S. The main properties of this method are summarised in the following theorem (compare with [36, p. 219]).

Theorem 2.2.1. *Let $S \in \mathbb{R}^{n\times n}$ be symmetric with eigenvalues $|\lambda_1| < |\lambda_2| \leq \cdots \leq |\lambda_n|$. Let y be the eigenvector corresponding to λ_1 and $y^\top q^{(0)} \neq 0$ with $q^{(0)} \in \mathbb{R}^n$, $||q^{(0)}|| = 1$. If $\tilde{q}^{(k)} = Sq^{(k-1)}$ and $q^{(k)} = \tilde{q}^{(k)}/||\tilde{q}^{(k-1)}||_2$, then*

$$\limsup_{k\to\infty} \left| ||\tilde{q}^{(k)}|| - |\lambda_1| \right|^{1/k} \leq |\lambda_2/\lambda 1|, \tag{2.8}$$

and

$$\limsup_{k\to\infty} \left|\left| q^{(k)} - \operatorname{sign}(\lambda_1^k) v \right|\right|^{1/k} \leq |\lambda_2/\lambda 1|, \tag{2.9}$$

with $v = \pm y$.

This shows that the sequence $\{\lambda^{(k)}\}$ converges to the spectral radius of S and $\{q^{(k)}\}$ to the corresponding eigenvector. The rate of convergence is linear with $|\lambda_2/\lambda_1|$.

Algorithm 1 Power Method

1: random initialise $q^{(0)}$.
2: **for** $k \leftarrow 1, 2, \ldots$ **do**
3: $\quad \tilde{q}^{(k)} = Sq^{(k-1)}$
4: $\quad q^{(k)} = \tilde{q}^{(k)}/||\tilde{q}^{(k)}||_2$
5: $\quad \lambda^{(k)} = \left(q^{(k-1)}\right)^\top \tilde{q}^{(k)}$
6: **end for**

Note that a random start vector $q^{(0)}$, as used in algorithm 1, may not fulfil the prerequisite $[q^{(0)}]^\top y \neq 0$ of theorem 2.2.1. However it turns out that in practice a component in direction of y will get introduced to $q^{(k)}$ due to rounding error [36, p. 222].

To approximate a different eigenvalue the iteration matrix S needs to be transformed. By substituting S with $(S - \mu\lambda)^{-1}$ the algorithm from above will converge to $\lambda = \min_i |\mu - \lambda_i|$ which is the closest eigenvalue of S to μ. This method is called *inverse iteration*. If the eigenvalue is updated as well, see algorithm 2, it is call *Rayleigh quotient iteration* which has a local cubic rate of convergence [36, p. 224].

Algorithm 2 Rayleigh quotient iteration

1: random initialise $q^{(0)}$.
2: **for** $k \leftarrow 1, 2, \ldots$ **do**
3: $\quad \tilde{q}^{(k)} = (S - \lambda^{(k-1)} I)^{-1} q^{(k-1)}$
4: $\quad q^{(k)} = \tilde{q}^{(k)} / ||\tilde{q}^{(k)}||_2$
5: $\quad \lambda^{(k)} = \left(q^{(k-1)}\right)^\top \tilde{q}^{(k)}$
6: **end for**

In case an eigenvalue λ of S is known, algorithm 2 can be used to calculate the corresponding eigenvector. In this case line 5 is not needed and $\lambda^{(k)}$ is given by λ. Such an *inverse iteration* converges quickly, e.g. `LAPACK` uses 2-7 iteration in `dstein`. The computational effort for computing the LU-decomposition of $(S - \lambda^{(k)} I)$ is $2n^3/3$ flops [30, p. 119]. For the computation of one eigenvector via the inverse iteration the floating point operation count is thus $2n^3/3 + c\mathcal{O}(n^2)$, where c is the iteration count.

2.3. Reduction to Tridiagonal Form

For a symmetric matrix S it is possible to find an orthogonal V such that

$$V^\top S V = T,$$

with T tridiagonal, that is $t_{ij} = 0$ for $|i - j| > 1$. Most algorithms which solve the eigenproblem are more efficient if S is first reduced to tridiagonal from. For instance the *inverse iteration* algorithm needs $\mathcal{O}(n)$ flops (see section 2.4.1) to compute an eigenvector of T but $\mathcal{O}(n^3)$ when applied of S. Furthermore the most efficient algorithms for the tridiagonal problem cannot be applied to a full matrix, e.g. *Cuppen's divide and conquer algorithm* or the newer *multiple relatively robust representations method*. Once the spectral decomposition of T is computed a back-transformation step yields the eigendecomposition of the original problem. In short, to efficiently compute the spectral decomposition $S = QDQ^\top$ the following three steps are performed:

1. Reduction to tridiagonal form:

$$S = VTV^\top.$$

2. Computation of the eigendecomposition of the tridiagonal problem:

$$T = \bar{Q} D \bar{Q}^\top.$$

3. Backtransformation of the eigenvectors:

$$Q = V\bar{Q}.$$

This approach is used in most numerical libraries such as `LAPACK`, `MKL`, `PLASMA` or `MAGMA`. Section 2.3.1 and section 2.3.2 discuss different approaches in finding an orthonormal transformation matrix V.

2.3.1. One Step Tridiagonalisation

This section introduces two methods of finding an orthonormal matrix V which reduces the symmetric matrix S to tridiagonal form. First *Householder reflectors* which reflect a given vector x along a hyperplain are discussed. For banded matrices, however, one can take advantage of the sparse structure. In these cases mostly *Givens rotations* which rotate a vector x in a two dimensional plain are used to perform the tridiagonal reduction.

2.3.1.1. Householder Transformations.

A *Householder reflector* $P \in \mathbb{R}^{n\times n}$ is defines by

$$P = I - \beta vv^\top, \qquad \beta\frac{2}{vv^\top}, \quad v \neq 0. \tag{2.10}$$

Such a *Householder transformation* is a symmetric and orthogonal rank-1 modification of the identity that reflect a vector x along the hyperplain $\text{span}\{v\}^\perp$. By choosing

$$v = x - ||x||_2 e_1 \tag{2.11}$$

one can introduce zeros in x, such that x is a positive multiple of e_1:

$$Px = \left(I - \beta vv^\top\right) x = ||x||_2 e_1. \tag{2.12}$$

As stated in [30, p. 235] care must be taken to avoid cancellation in the calculation of v, e.g. if x is close to a multiple of e_1. If v is computed correctly it can be shown that an update using the generated $\hat{P}$ is equivalent to an exact update P of a lightly perturbed matrix:

$$\text{fl}(\hat{P}A) = P(A + E), \quad ||E||_2 = \mathcal{O}(\epsilon||A||_2). \tag{2.13}$$

Here $\text{fl}(\cdot)$ indicates the floating point operation.

The computation of the *Householder vector* takes $3n$ flops. The application of P to a matrix A is basically a matrix-vector product

$$PA = (I - \beta v v^\top)A = A - (\beta v)(v^\top A) \tag{2.14}$$

and takes $4n^2$ flops.

Householder transformations can be used to tridiagonalise a symmetric matrix S. Algorithm 3 (see [30, p. 459]) produces a tridiagonal matrix T and stores the *Householder reflectors* in the lower triangular (without the first subdiagonal) part of the matrix. This is done by successively applying *Householer reflectors* to ever smaller submatrices $S(k+1:n, k+1:n)$.[3] The first step produces zeros in the elements s_{31} to s_{n1}. The second step annihilates the elements s_{42} to s_{n2}, and so on. The function `house` in line 2 constructs a *Householder reflector* as discussed above.

Algorithm 3 Tridiagonalisation

1: **for** $k \leftarrow 1 : (n-1)$ **do**
2: $\quad [v, \beta] = \text{house}(S(k+1:n, k))$
3: $\quad p = \beta S(k+1:n, k+1:n)v$
4: $\quad \omega = p - (\beta p^\top v/2)v$
5: $\quad S(k+1, k) = ||S(k+1:n, k)||_2$
6: $\quad S(k, k+1) = S(k+1, k)$
7: $\quad S(k+1:n, k+1:n) = S(k+1:n, k+1:n) - v\omega^\top - \omega v^\top$
8: **end for**

If the symmetry of S is exploited algorithm 3 requires $4n^3/3$ flops [30, p. 459]. In case the transformation matrix V is needed an additional $4n^3/3$ flops are required for its computation. The algorithm is implemented in `LAPACK` in `L/dsytrd`, which is used in `L/dsyevd` to solve a dense symmetric eigenproblem, see fig. 2.1 (right).[4]

2.3.1.2. Givens Rotations.

In cases where S has a sparse structure it can be advantageous to introduce zeros more selectively. An example of such matrices are symmetric band matrices. These are matrices B with $b_{ij} = 0$ for $|j - i| > r$ where r is called the semi-bandwidth of B, see chapter 3.

[3] The Syntax is to be understood as follows. $1 : n$ denotes the set $\{1, 2, \ldots, n\}$. If I is a set of row-indices and J is a set of column-indices, then $S(I, J)$, denotes the submatrix which arises by selecting the rows with index $i \in I$ and the columns with index $j \in J$.

[4] `B/`, `L/` and `P/` indicate a `BLAS`/`LAPACK`/`PLASMA` routine respectively.

A *Givens rotation* is defined by an orthonormal matrix

$$G(i,k,\theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}, \tag{2.15}$$

with $c = \cos(\theta),\ s = \sin(\theta)$. The matrix-vector product $G(i,k,\theta)x$ rotates a vector clockwise by θ. This can be used to introduce a single zero in the k-th element of x by setting

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}}, \quad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}}. \tag{2.16}$$

If the computation is done right it takes 5 flops and one square root to computed c and s. An update of $S \in \mathbb{R}^{n\times n}$ involves only two rows

$$S([i,k],:) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} S([i,k],:), \tag{2.17}$$

and takes $6n$ flops [30, p. 241].

Schwarz [46] presented a method to reduce a banded matrix to tridiagonal form using *Givens rotations*. In each step one element[5] inside the band is annihilated which produces two nonzero elements outside the band. These are zeroed out in a *bulge chasing* operation which restores the banded form. It takes roughly n/p plain rotations to introduce a zero using this bulge chasing method. To tridiagonalise a banded matrix with semi-bandwidth r it takes

$$\sum_{i=1}^{n}(n-i)\frac{r-1}{r} \approx \frac{n^2}{2}\frac{r-1}{r} \tag{2.18}$$

rotations, each taking $\mathcal{O}(r)$ flops [48, p. 189]. This method is fast, with a complexity of $\mathcal{O}(n^2 r)$, if the rotations are not accumulated. However, it is important to note that $\mathcal{O}(n^2)$ rotations need to be accumulated in case the transformation matrix V is needed. This yields a flop count of approximately $n^2\frac{r-1}{r}(6n+12r)$ [4, p. 18], making *Householder reductions* more favorable in case V is required. This algorithm is implemented in `LAPACK`

[5] More precisely one pair of elements, since B is symmetric.

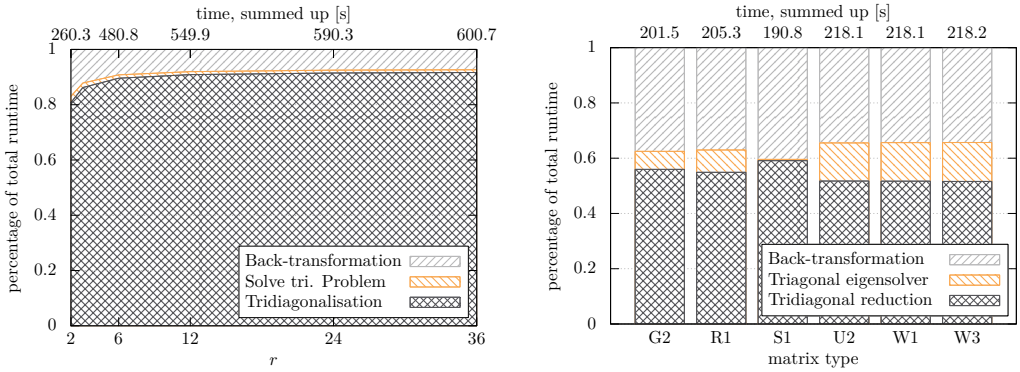


Figure 2.1.: Computing the eigendecomposition of a bandsymmetric matrix. Percentage of the total runtime used by each step. On the left: The routine `L/dsbevd`, which uses the band-to-tridiagonal reduction routine `L/dsbtrd` (using *Givens rotations*), for varying r and matrix type `G2`. On the right: The runtime of `L/dsyevd`, using the full-to-tridiagonal reduction routine `L/dsytrd` (using *Householder reductions*), for different matrix types, $n = 10200$. See chapter 4 for an overview of the testing setup as well as a description the matrix types.

in `L/dsbtrd`, which is used in `L/dsbevd` to solve a dense symmetric eigenproblem, see fig. 2.1 (left).

As for the *Householder transformation* it can be shown that the transformation has good numerical properties. Let $\hat{G}(i,k,\theta)$ be the computed update, whereas $G(i,k,\theta)$ is the exact update in the absence of numerical errors. If the update is computed in an optimal manner, the computed update is the exact update of a nearby matrix [30, p. 241]:

$$fl\left(\hat{G}(i,k,\theta)A\right) = G(i,k,\theta)(A+E), \quad ||E||_2 \approx \epsilon||A||_2, \tag{2.19}$$

2.3.1.3. Numerical Stability

Householder and *Givens* transformations are both orthogonal to working precision

$$||\hat{V}^\top\hat{V} - I_n|| = \mathcal{O}(\epsilon), \tag{2.20}$$

where $\hat{V}$ is the generated *Householder reflector* or *Givens rotation*. A sequence of such transformations has nice error bounds in that the update is an exact orthogonal update of a matrix near S. That is: if $S_1, \ldots, S_p$ are the consecutive updated matrices via

$$S_i = \text{fl}(\hat{V}_i S_{i-1} \hat{V}_i^\top), \qquad i = 1, 2, \ldots, p, \tag{2.21}$$

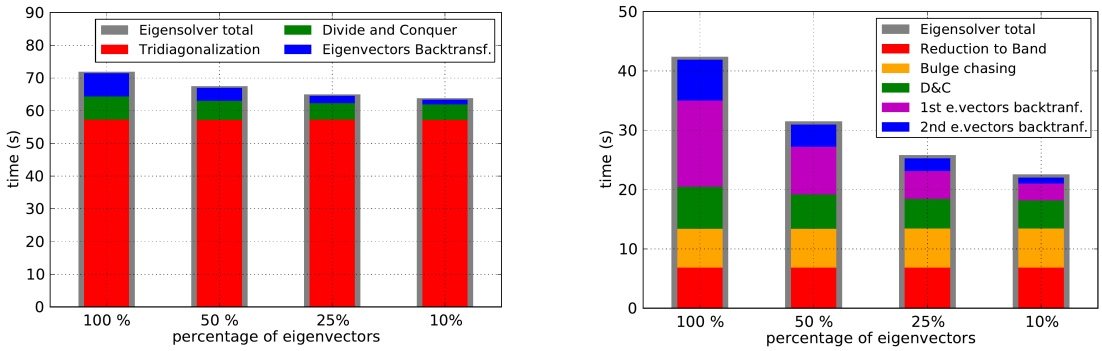


Figure 2.2.: Graphics taken from [35]. Run on a hybrid system (eight-core Intel Xeon E5-2670 and a Nvidia K20c GPU). Double-precision with $n = 14000$. Left: Traditional tridiagonalisation. Right: Two stage reduction.

then

$$S_p = (V_p \cdots V_1)(S + E)(V_p \cdots V_1)^\top \tag{2.22}$$

where $||E||_2 \leq c \cdot \epsilon ||S||_2$, with E being symmetric and c a small constant depending on n and p, see [30, p. 243].

2.3.2. Two Stage Tridiagonal Reduction

To transform a dense symmetric matrix S to triadonal form mostly *Householder reductions* are used. E.g. `LAPACK` uses such an approach in the tridiagonalisation routine `L/dsytrd`. The complexity of this reduction is, as discussed in section 2.3.1.1, $8n^3/3$. We will see in section 2.4.4 that the solution of the tridiagonal eigenproblem only takes approximately $\mathcal{O}(n^{2.3})$ with mostly level 3 `BLAS` operations. The overall computation is thus dominated by the reduction process followed by the back-transformation of the eigenvalues with a complexity of $\mathcal{O}(n^3)$. The cost for the different steps when solving a banded eigenproblem via `L/dsyevd`[6] or `L/dsbevd` are shown in fig. 2.1. In case S is banded the algorithm proposed by Schwarz [46] may lead to a better performance. However, the complexity of accumulating the rotations is high, yielding an overall flop count of $n^2 \frac{r-1}{r}(6n + 12r)$ for the tridiagonalisation of a banded matrix with semi-bandwidth r. In numerical experiments `L/dsytrd` always outperformed `L/dsbtrd` - even for a semi-bandwidth of $r = 2$. This can be seen in fig. 2.1: `Ldsbtrd` has a runtime of 260.3 seconds for $r = 2$, whereas it takes `L/dsyevd` only 201.5 seconds to compute the spectral decomposition for a matrix of type `G2`.

[6] `L/dsyevd` uses `L/dsytrd` for the tridiagonal reduction. Thus it does not take advantage of the sparse banded structure of S. See section 7.1 for further details.

In case only eigenvalues are calculated `L/dsbtrd` takes approximately $12rn^2$ flops making it an order of magnitude faster than *Householder transformations*. This is taken advantage of for the reference implementation of an *inverse iteration* algorithm for bandsymmetric matrices, see section 3.1.

Bischof, Sun and Lang [9] developed a toolbox called *Successive Band Reductions* (`SBR`) that uses a two step tridiagonalisation method that first reduces a dense symmetric matrix to banded form and tridiagonalises the banded matrix afterwards, see fig. 1.1. This approach improved the data locality and enabled the use of level 3 `BLAS` operations. They also discussed a multistep reduction process which includes an intermediate step where the bandwidth of a banded matrix is reduced before bringing it to tridiagonal form [10]. In [11] they concluded that this approach outperforms the traditional one step process if the transformation matrix V is not accumulated. However, if the update V is needed the new approach greatly increases the flop count. In case V is calculated their algorithm outperformed `L/dsbtrd`, but not `L/dsytrd` [10].

Davis and Rajamanickam [18] presented the library `PIRO_BAND` [45] that tries to group successive plane rotations (*Givens rotations*) and apply them in a blocked fashion when reducing a banded matrix to tridiagonal form. They reported a speedup of are least 2 in comparison to `L/dsbtrd` independent whether V is accumulated [44]. However, this could not be reproduced on our system. In fact, for our test calculations, `L/dsbtrd` turned out to be fast if V is calculated as well.

For modern multicore architectures the processor-memory speed gap is still increasing. The two step tridiagonalisation methods developed by Bischof, Sun and Lang [9] provides a way to maximise the use of level 3 `BLAS` operations. This results in a compute bound first stage that reduced a matrix to banded form and a memory bound second stage which yields the tridiagonal system [35]. Ltaief, Luszczek and Dongarra [39] adopted this two stage approach with a new implementation using a tile layout[7] and an improved second stage. This method is available in `PLASMA` and was refined by Haidar, Ltaief and Dongarra [34]. They reported very favourable results for multicore architectures with a speedup of 12 compared to `SBR`.

A communication avoiding algorithm is presented by Ballard, Demmel and Knight [6]. They reported a 6 fold speedup over the tridiagonalisation routine present in `PLASMA` tested on an ten-core processor. Further they showed that for small bandwidth their algorithms can attain as much data reuse as `dgemm`. However no results for accumulating V are presented.

Haidar, Tomov, Dongarra *et al.* [35] implemented a similar two stage approach as

[7] Tile algorithms use a task based parallelisation approach instead of the traditional fork-join model [39].

discussed in [34] but for a hybrid CPU-GPU architecture. The implementation is available via the `MAGMA` library. The main results are show in fig. 2.2. The left figure shows the cost for solving a symmetric eigenproblem using the traditional approach. On the right the runtime using the two stage reduction method is display. For computing the spectral decomposition the two stage approach yields an approximate speedup of 1.6.

2.4. Symmetric Tridiagonal Eigensolvers

This section is concerned with the real symmetric tridiagonal eigenproblem

$$Tx_i = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & b_{n-1} & a_n \end{pmatrix} x_i = \lambda_i x_i. \tag{2.23}$$

The goal is once again to compute all eigenpairs and thus the spectral decomposition $T = Q\Lambda Q^\top$. Such tridiagonal systems can be obtained via orthogonal transformation. In the following paragraphs various algorithms are discusses which are later compared in section 2.4.4. The focus lies on the divide and conquer method proposed by Cuppen, as it is a predecessor of the divide and conquer algorithms for bandsymmetric systems, see chapter 3. All eigensolvers from this section are available as `LAPACK` routines.

In most cases it can be assumed that $b_i \neq 0 \; \forall i$, since otherwise the problem degenerates into two smaller ones.

2.4.1. Bisection and Inverse Iteration

Using the *inverse iteration* algorithm discussed in section 2.2.1 with a tridiagonal matrix T instead of a dense matrix S reduces the computational cost drastically to $\mathcal{O}(n)$ flops per eigenvector. Hence $\mathcal{O}(nk)$ flops are needed to computed k eigenvectors. This makes *inverse iteration* the fastest algorithm to compute some or all eigenvectors if the corresponding eigenvalues are known. Unfortunately, *inverse iteration* cannot guarantee orthogonality between the computed eigenvectors. If the eigenvalues are clustered an additional $\mathcal{O}(nk^2)$ flops are required for re-orthogonalisation, e.g. via *modified Gram-Schmidt*, see algorithm 6. To keep the additional work minimal re-orthogonalisation is only done for groups of clustered eigenvalues. However orthogonality can still not be

guaranteed, see section 3.1.1. This method is implemented in `L/dstein`. In section 3.1 a reference implementation of the *inverse iteration* for bandsymmetric matrices will be presented.

The remaining of this section covers the computation the eigenvalues. *Sylvester's law of inertia*[8] provides a way to use bisection to determine all eigenvalues in a given interval. The corresponding `LAPACK` routine is `L/dstebz`.

Definition 2.4.1 (Inertia)**.** The inertia of a quadratic matrix $A \in \mathbb{R}^{n\times n}$ is a triples (π, ν, ζ) of integers. π denotes the number of positive, ν the number of negative eigenvalues of A and $\zeta = \text{kern}(A)$.

Theorem 2.4.1 (Sylvester's Law of Inertia)**.** *Let $S \in \mathbb{R}^{n\times n}$ be a symmetric matrix and $X \in \mathbb{R}^{n\times n}$ a regular matrix. Then*

$$\text{interia}(S) = \text{interia}(X^\top SX) \tag{2.24}$$

and the matrices S and $X^\top SX$ are said to be congruent.

Algorithm 4 Bisection

```
1: procedure BISECTION(T, k)
2:     while |b − a| > ϵ|b| do
3:         m = (a + b)/2
4:         if n_−(T, m) < k then
5:             a = m
6:         else
7:             b = m
8:         end if
9:     end while
10: end procedure
```

Let $n_-(T, \lambda)$ denote ν of inertia$(T - \lambda)$ and thus the number of eigenvalues of T smaller λ. Then one can use bisection to compute all or specific eigenvalues, e.g. all in a given interval. Algorithm 4 calculates the k-th largest eigenvalue of T using n_-. The stopping criterion in line 2 is chosen to be the minimal interval in the used precision, where ϵ is the machine precision.

The evaluation of $n_-(T, \lambda)$ only takes $4n$ flops if the LDL-decomposition is used [20,

[8] A proof can be found in [30, p. 448].

p. 231]. To see this, note that if

$$T - \lambda = \begin{pmatrix} a_1 - \lambda & b_1 & & & \\ b_1 & a_2 - \lambda & b_2 & & \\ & b_2 & a_3 - \lambda & \ddots & \\ & & \ddots & \ddots & b_{r-1} \\ & & & b_{r-1} & a_r - \lambda \end{pmatrix} = LDL^\top \tag{2.25}$$

then $T - \lambda$ and D are congruent and D is diagonal.

Bisection has a linear rate of convergence. As soon as an interval containing exactly one eigenvalue and thus one zero, it is advantageous to use a superlinear zero-finder, e.g. by applying *Brent's zeroin* on the determinant of $T - \lambda$. The following lemma[9] shows that $\det(T - \lambda)$ can be evaluated in $\mathcal{O}(n)$ flops.

Lemma 2.4.2. *Let T_r denote the $r \times r$ submatrix $T(1 : r, 1 : r)$ of T. Then*

$$p_r(\lambda) = \det(T_r - \lambda I) \tag{2.26}$$

$$= (a_r - \lambda)p_{r-1}(\lambda) - b_{r-1}^2 p_{r-2}(\lambda) \tag{2.27}$$

2.4.2. Cuppen's Divide and Conquer Algorithm

This algorithm was introduced in 1981 by Cuppen [17]. In each step it uses a rank-*one* modification to partition the eigenproblem of size n into two small problems of size $n/2$. The method of solving such rank one modifications was first fully described by Bunch, Nielsen and Sorensen [15] in 1978, which is mainly bases on Golub [29]. A parallel implementation for shared memory architectures was presented by Dongarra and Sorensen [23] in 1987. Unfortunately the algorithm may not lead to orthogonal eigenvectors which lead to the use of extended precision [47]. In 1994 Gu and Eisenstat [31] presented a new method that guarantees accurate eigenvalues without the use of extended precision. The resulting algorithm is in every aspect more favourable than *QR* or *Bisection with Inverse Iteration* [32]. The corresponding `LAPACK` routine is `L/dstevd`. Tisseur and Dongarra [50] presented a scaleable implementation for distributed memory architectures for the `ScaLAPACK` [12] library. For hybrid GPU-accelerated multicore systems an efficient implementation is discusses in Vömel, Tomov and Dongarra [51]. An implementation for the `PLASMA` library that exploits parallelism on a multicore architecture using a task-flow model was presented by Pichon, Haidar, Faverge *et al.* [43].

[9] This can be seen by applying *Laplace expansion* twice [30, p. 467].

2.4.2.1. Rank-1 Update

The following theorem[10] provides a way to compute the spectral decomposition of the rank-1 modification

$$D + \rho z z^\top$$

of a diagonal matrix D.

Theorem 2.4.3. *Let $D \in \mathbb{R}^{n\times n}$ be a diagonal matrix $D = \operatorname{diag}(d_1, d_2, \ldots, d_n)$ with elements $d_i \neq d_j$. Assume $\rho \neq 0$ and $z \in \mathbb{R}^n$ with no zero component, then*

(a) The eigenvalue of $D + \rho z z^\top$ are the n zeros of

$$f(\lambda) = 1 + \rho z^\top (D - \lambda I)^{-1} z = 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda}. \tag{2.28}$$

(b) The eigenvalues of $D + \rho z z^\top$ are interlaced by d_i:

$$\text{if } \rho > 0, \quad \lambda_1 > d_1 > \lambda_2 > d_2 > \cdots > \lambda_n > d_n, \tag{2.29}$$

$$\text{if } \rho < 0, \quad d_1 > \lambda_1 > d_2 > \lambda_2 > \cdots > d_n > \lambda_n. \tag{2.30}$$

(c) The eigenvectors $x_i \in \mathbb{R}^n$ of $D + \rho z z^\top$ are given by

$$x_i = (D - \lambda_i I)^{-1} z. \tag{2.31}$$

The property described in eq. (2.29) allows the efficient computation of the eigenvalues, since each interval (d_i, d_{i+1}) contains exactly one zero. For an efficient methods to solve the so called *secular equation* see [20, p. 221].

2.4.2.2. Divide and Conquer Algorithm

Any divide and conquer method consists for three parts: The decomposition of the problem giving two subproblems of lower dimension, the solution of the simpler subproblems, and the combination of their solutions to solve the original problem.

The algorithm described in [17] splits a tridiagonal matrix into two subproblems of lower dimension plus a rank one modification. Theorem 2.4.3 is used to find the spectral decomposition of T given those of $T_i,\ i = 1, 2$. Given a symmetric tridiagonal matrix T

[10] First fully described in [15]. Compare with [30, p. 470] and [36, p. 248].

the subproblems T_i are defined as follows:

$$T = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & b_{n-1} & a_n \end{pmatrix} = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho v v^\top \tag{2.32}$$

with

$$T_1 = \begin{pmatrix} a_1 & b_1 & & \\ b_1 & a_2 & \ddots & \\ & \ddots & \ddots & b_{r-1} \\ & & b_{r-1} & c_r \end{pmatrix}, \quad T_2 = \begin{pmatrix} c_{r+1} & b_{r+1} & & \\ b_{r+1} & a_{r+2} & \ddots & \\ & \ddots & \ddots & b_{n-1} \\ & & b_{n-1} & a_n \end{pmatrix}, \tag{2.33}$$

$c_r = a_r - b_r, \quad c_{r+1} = a_{r+1} - b_r, \quad \rho = b_r, \quad v = \begin{pmatrix} e_m \\ e_1 \end{pmatrix}.$

In a next step the subproblems T_i are solve, yielding

$$T_1 = Q_1 \Lambda_1 Q_1^\top, \qquad T_2 = Q_2 \Lambda_2 Q_2^\top.$$

This can be done by applying a standard algorithm, as for example the `QR` method, or in a recursive manner. `LAPACK`'s subroutine `L/dstedc` uses the `QR` algorithm for subproblems smaller $n < 25$.[11]

Finally the solution of the original problem presents itself as a rank-one update of the the spectral decompositions of the subproblems.

$$T = \left[\begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho v v^\top \right] = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left[\begin{pmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{pmatrix} + \rho z z^\top \right] \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^\top \tag{2.34}$$

with

$$z = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^\top v = \begin{pmatrix} \pm Q_1^\top e_m \\ Q_2^\top e_1 \end{pmatrix}. \tag{2.35}$$

By setting $D = \Lambda_1 \oplus \Lambda_2$ the prerequisites of theorem 2.4.3 are satisfied and thus the eigenvalues of T can be computed by solving eq. (2.28) for $D + \rho z z^\top$. The corresponding

[11] This is true for `LAPACK` version 3.7.0. The threshold is set in the subroutine `L/ilaenv`.

eigenvectors are given by $q_i = Qx_i$, where x_i is computed as in eq. (2.31). Note that the algorithm can be generalised to deal with cases in which the prerequisites $z_i \neq 0$ and $\Lambda_{ii} \neq \Lambda_{jj}$ are not met. This process is called *deflation* and can be also applied if $z_i \approx 0$, e.g. by setting $z_i = 0$. For deflated eigenvalues the corresponding eigenvector has a simple form and the matrix-vector multiplication is not required. This results in a large performance gain and for most problems in a lower asymptotic complexity [17].

2.4.2.3. Operation Count

The computation of an eigenvalue takes a small number of iterations[12], each iteration taking $\mathcal{O}(n)$ flops. An additional $\mathcal{O}(n)$ flops are needed for the computation of the corresponding eigenvector x_i of $D + \rho zz^\top$. To finish the computation of the eigenvectors

$$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^\top X, \tag{2.36}$$

where X contains the eigenvectors x_i of $D + \rho zz^\top$ is needed, which takes $c \cdot n^3$ flops, $c < 1$. Using an recursive approach the number of floating point operations can be written as

$$t(n) = 2t(n/2) + cn^3 + \mathcal{O}(n^2). \tag{2.37}$$

Disregarding the $\mathcal{O}(n^2)$ term this yields an overall complexity of $t(n) \approx c4n^3/3$. The process of *deflation* simplifies eq. (2.36) resulting in a $c \ll 1$ [20, p. 221].

2.4.2.4. Stable computation of the Eigenvectors

The computation of the eigenvectors via eq. (2.31) turns out to be numerical unstable if two eigenvalues λ_i and λ_{i+1} are close. The problem has its origin in the cancellations occurring in $(D - \lambda_i I)^{-1}z$ and $(D - \lambda_{i+1} I)^{-1}z$. That is because one element of D is in between λ_i and λ_{i-1}. In these cases the orthogonality of the eigenvalues is not guaranteed. The following theorem allows for an elegant fix, see [20, pp. 225, 226].

Theorem 2.4.4 (Löwner)**.** *Let $D \in \mathbb{R}^{n\times n} = \operatorname{diag}(d_1, \ldots, d_n)$ be a diagonal matrix with elements $d_i \neq d_j$. Assume λ_i satisfy the interlacing property*

$$d_n < \lambda_n < \cdots < d_{i+1} < \lambda_{i+1} < d_i < \lambda_i < \cdots < d_1 < \lambda_1. \tag{2.38}$$

[12]The `LAPACK` routine `L/slaed4` takes on average two to three iterations per eigenvalue [20, p. 223].

Then λ_i are the exact eigenvalues of $D + \hat{u}\hat{u}^\top$, where the vector $\hat{u}$ has element

$$|\hat{u}_i| = \left[\frac{\prod_{j=1}^{n}}{\prod_{j=1, j\neq 1}^{n} (d_j - d_i)} \right]^{1/2}. \tag{2.39}$$

To get an accurate eigenvector x_i first calculate $\hat{u}$ as described in theorem 2.4.4, such that the already found eigenvalue λ_i of $D + \rho zz^\top$ is the *exact* eigenvalue of $D + \hat{u}\hat{u}^\top$. Afterwards eq. (2.31) is used to calculate the eigenvector corresponding to the eigenvalue λ_i of $D + \hat{u}\hat{u}^\top$. The resulting eigenvector is also an accurate eigenvector of $D + \rho zz^\top$.

2.4.3. Two Further Algorithms

The `QR` algorithm presented in 1961 by Francis [25] is a well studied classical algorithm to computed the spectral decomposition of a matrix. The `LAPACK` driver routine is called `L/dstev`. It is still the fastest method available in `LAPACK` to find all eigenvalues, taking only $\mathcal{O}(n^2)$ flops. For the computation of all eigenpairs it takes on average little over $6n^3$ flops, making it slower than *Cuppens divide and conquer algorithm* (`DC`) [20, p. 213]. However, for small matrices ($n < 25$) it is still the fastest routine [20, p. 211] and is thus used in `DC` for small subproblems.

A newer algorithm that has promising parallel properties is the *Multiple Relatively Robust Representation* (`MR`) algorithm Dhillon and Parlett [21], [8], [22]. It is implemented in the `LAPACK` routine `L/dstemr`. The algorithm requires $\mathcal{O}(kn)$ operation to compute k eigenvectors and has thus a lower complexity as `DC`. `MR` uses a sophisticated variant of inverse iteration and can guarantee orthogonal eigenvectors without using a re-orthogonalisation process as e.g. the *modified Gram-Schmidt* algorithm.

2.4.4. Comparison of Tridiagonal Eigensolvers

Marques, Vömel, Demmel *et al.* [40] developed a testing infrastructure for tridiagonal eigensolvers. They describe the generation of test matrices with different eigenvalue distributions. These correspond to differently conditioned problems as the hardness of a symmetric eigenproblem depends on the distribution of the eigenvalue, see theorem 2.1.2. They used this test infrastructure to compare the stage-of-the-art tridiagonal eigensolvers available in `LAPACK`. In the following a short summary of [19] is given.

Four algorithms are discussed, namely (here the `LAPACK` routine is stated first):

- `L/dstevx`: Bisection and Inverse Iteration (`BI`)
- `L/dstevd`: Divide and Conquer (`DC`)

Table 2.1.: Comparison of LAPACKS tridiagonal eigensolvers.

	workspace(double)	workspace(int)	complexity
L/dstevx	$8n$	$5n$	$\mathcal{O}(n^3)$
L/dstevd	$n^2 + 4n + 1$	$5n + 3$	$4n^3/3$
L/dstev	$2n - 2$	0	$6n^3$
L/dstevr	$18n$	$10n$	$\mathcal{O}(n^2)$

- `L/dstev`: QR iteration (`QR`)
- `L/dstevr`: Multiple Relatively Robust Representations (`MR`)

The workspace requirements as well as the theoretical complexity are summarised in table 2.1. Note that the complexity is problem dependent. The `DC` algorithms has a worst case complexity of $\mathcal{O}(n^3)$ flops, in practice however, it appears as $\mathcal{O}(n^{2.3})$ due to deflation [43]. The `BI` method has a complexity of $\mathcal{O}(n^2)$ for matrices with well separated eigenvalues. However, since it uses the *modified Gram-Schmidt* procedure to re-orthogonalise clusters of eigenvalues it may degenerate into an $\mathcal{O}(n^3)$ algorithm. Demmel, Marques, Parlett *et al.* [19] have shown that `BI` and `QR` are much slower than either `DC` or `MR` on synthetic as well as on practical matrices. They noted further that `DC` indeed has a higher flop count than `MR`, however, since it can makes use of level 3 `BLAS` operations it may still be faster. Note that for symmetric matrices little runtime difference between `MR` and `DC` can be observed, because tridiagonalisation dominates the runtime (see section 2.3.2).

As already discussed in section 2.2.1 `BI` can fail to compute orthogonal eigenvectors for problems with cluster eigenvalues. On the other hand `QR`, `DC` and `MR` guarantee an orthogonality error and a residual less than

$$c \max_{i \neq j} \frac{|q_i^\top q_j|}{\epsilon n ||S||_2} \quad \text{and} \quad c \max_i \frac{||Sq_i - \lambda_i q_i||_2}{\epsilon n ||S||_2}, \tag{2.40}$$

respectively, where c is a small constant [19]. `MR` has lower accuracy guarantees than `DC` and even fails to computed the correct eigendecomposition for some of our test matrices.[13] Hence `DC` is the algorithm of choice if the higher accuracy matters. The only drawback of `DC` is the large workspace requirement that originates from the use of level 3 `BLAS` operations.

[13] More precisely `MR` fails to compute the spectral decomposition for some matrices of type `S2`, e.g. for `S2_10200_3_20062`.

The DC algorithm has been successfully parallelised on many architectures [23], [50], [51], [43]. Pichon, Haidar, Faverge *et al.* [43] showed that for shared memory systems their implementation is competitive with the fastest MR implementation.

3. The Bandsymmetric Eigenproblem

In this chapter we will consider the real bandsymmetric eigenproblem $Bx = \lambda x$. As in chapter 2 the goal is to compute all n eigenvalues and corresponding -vectors, hence the eigendecomposition

$$B = Q\Lambda Q^\top. \tag{3.1}$$

Here Λ is a diagonal matrix containing the eigenvalues in increasing order and Q is a full matrix containing the corresponding eigenvectors. This is possible since the algebraic and geometric multiplicity coincide, see section 2.1. A symmetric band matrix B has the form

$$B = \begin{pmatrix} b_{00} & \cdots & b_{0r} & & & & \\ \vdots & \ddots & & \ddots & & & \\ b_{0r} & & \ddots & & \ddots & & \\ & \ddots & & \ddots & & b_{n-r-1,n-1} & \\ & & \ddots & & \ddots & \vdots & \\ & & & b_{n-r-1,n-1} & \cdots & b_{n-1,n-1} & \end{pmatrix} \in \mathbb{R}^{n\times n},$$

with $b_{ij} = 0$ for $|i-j| > r$, where r is call the semi-bandwidth[1] of B. In the following the semi-bandwidth r is always greater one. The easier case of $r = 1$ gives the tridiagonal eigenproblem discussed in section 2.4.

Bandsymmetric matrices may arise from real world problems, but more importantly also as an interim result when reducing a dense matrix to tridiagonal form. That is, modern numerical libraries use a two-stage tridiagonalisation approach to utilise parallelism, see section 2.3.2. The first stage is compute-bound and yields a banded matrix, the second stage reduces the banded matrix to tridiagonal form and is memory/communication-bound [34]. This process is depicted in fig. 1.1. Since the reduction is carried out in two steps also two back-transformation are needed. Note that the bulge chasing procedure (second stage) consists of $O(n^2)$ rotations, making the back-transformation costly, see

[1] The bandwidth of B is thus $2r + 1$.

fig. 2.2. The algorithms discussed in this section allow to skip the second reduction stage and thus also the first back-transformation. In return they are far more expensive then the tridiagonal analogue. This trade-off is numerically evaluated for the sequential case in chapter 7. A parallel comparison is out of the scope of this thesis.

In this chapter two approaches to take advantage of the banded structure of B are introduced. Section 3.1 describe `dsbein`, an implementation of *inverse iteration* for banded matrices. This algorithm is not particular suited to compute the complete spectral decomposition, but will give an idea of the potential of avoiding tridiagonalisation. Section 3.2 constitutes the main focus of this chapter and reviews the three divide and conquer algorithms proposed by Arbenz [4]. In particular section 3.2.2 will lay the theoretical background needed for the following chapters.

3.1. An Inverse Iteration Based Approach

In section 2.3 different methods for the tridiagonal reduction of a symmetric matrix have been discussed. In recent years many new algorithms have been developed for the band-to-tridiagonal reduction, see section 2.3.2. If the transformation matrix Q is not accumulated many of those implementation are highly efficient and produce notable speedups in comparison to `L/dsbtrd`, as e.g. `PIRO_BAND` [44]. In this section an approach is discussed that takes advantage of these efficient reduction routines and combines them with *inverse iteration* for banded matrices. To compute the spectral decomposition of a banded matrix B the following three steps are performed:

1. Reduce B to tridiagonal form T without accumulating Q.
2. Compute eigenvalues of T.
3. Find corresponding eigenvectors using *inverse iteration* applied to B.

The implementation discussed below allows to estimate the potential of such an approach. Further it illustrates the benefits of working directly on the band structure. The performance is evaluated in section 7.3.1.

For the tridiagonal analogue of this routine see section 2.4.1. There a *bisection* based approach using the inertia of $T - \lambda I$ is used to locate selected eigenvalues. In this section a combination of the tridiagonal reduction using *Givens rotation* (via `L/dsbtrd`) and the tridiagonal `QR` method (via `L/dsbevd`) is used instead. Note that since the transformation matrix Q is not accumulated `L/dsbtrd` is extremely fast, taking approximately $12rn^2$

flops.[2] The tridiagonal `QR` algorithm has a complexity of $\mathcal{O}(n^2)$ and is the algorithm of choice if all eigenvalues are calculated.

The final step computes the corresponding eigenvalues using *inverse iteration*, see algorithm 5. This implementation is adopted from `L/dstein`, which is the tridiagonal *inverse iteration* method in `LAPACK`. In lines 2 to 4 some thresholds are set, see [38] for a justification of these values. For each eigenvalue the LU-decompostion of $B - \lambda_i I$ is calculated in line 8. Note that extra workspace (in algorithm 5 denoted as `LU`) is needed to not corrupt the original matrix B. The computation of the LU-decomposition takes $\mathcal{O}(nr^2)$ flops. To approximate the eigenvector the algorithm solves (line 11)

$$x^{(k)} = (B - \lambda_i I)^{-1} x^{(k-1)} = LU \backslash x^{(k-1)}, \tag{3.2}$$

up to 7 times, each iteration taking $\mathcal{O}(nr)$ flops.

Unfortunately *inverse iteration* does not guarantee orthogonal eigenvectors, if the eigenvalues are clustered. Algorithm 5 uses the *modified Gram-Schmidt* method (see section 3.1.1) to re-orthogonalise eigenvectors inside a cluster $\mathcal{C}$ of eigenvalues. A cluster $\mathcal{C}$ is defined as a group of eigenvalues such that

$$\forall \lambda_i \in \mathcal{C} \quad \exists \lambda_j \in \mathcal{C} \backslash \{\lambda_i\} : \quad |\lambda_i - \lambda_j| < ortol \tag{3.3}$$

holds, where *ortho* is defined in line 2. Hence inside a cluster of size $c = |\mathcal{C}|$ each iteration takes $\mathcal{O}(nc + nr)$.

The overall complexity of this approach is $\mathcal{O}(n^2 r^2)$ in case of unclustered eigenvalues, but $O(n^3)$ for systems with clustered eigenvalues.

3.1.1. Modified Gram-Schmidt Algorithm

Algorithm 6 (see [30, p. 255]) describes the *modified Gram-Schmidt* algorithm for $A \in \mathbb{R}^{m \times n}$ and requires $2n^2 m$ flops. The orthogonality guarantees of this algorithm are

$$\hat{Q}^\top Q = I + E_{MGS}, \qquad ||E_{MGS}||_2 \approx \epsilon \frac{\lambda_n}{\lambda_0}, \ \lambda_i \in \sigma(A) \tag{3.4}$$

whereas

$$\hat{Q}^\top Q = I + E_H, \qquad ||E_H||_2 \approx \epsilon \tag{3.5}$$

holds if Q is computed using *Householder transformations*, see [30, p. 254]. Here $\hat{Q}$ denotes the output of the respective algorithm, whereas Q is the exact result in the

[2] New algorithms like `PIRO_BAND` should be even faster.

Algorithm 5 Inverse Iteration

procedure DSBEIN(B, LU)
  $ortol \leftarrow 10^{-3}\lambda_n$
  $gpind \leftarrow 0$
  $dtpcrt \leftarrow \sqrt{n/10}$
  randomly initialise Q : $q_{ij} \in (0,1)$ ▷ *random start vectors*
  for $i \leftarrow 1, 2, \ldots n$ **do**
    copy $B - \lambda_i I$ to LU
    compute LU-decomposition ▷ *using* `L/dgbtrf`
    for $j \leftarrow 1, 2, \ldots 7$ **do**
      $q_i \leftarrow q_i / ||q_i||_2$
      solve $LU \backslash q_i$ ▷ *using* `L/dgbtrs`
      if dist$(\lambda_i - \lambda_{i-1}) > ortol$ **then**
        $gpind = i$
      else
        for $k \leftarrow gpind, gpind + 1, \ldots, i$ **do** ▷ *MGS*
          $q_i = -(q_i^\top q_k) \cdot q_k$
        end for
      end if
      if $\max_j |(q_i)_j| > dtpcrt$ AND $j < 5$ **then**
        $j \leftarrow 5$ ▷ *if criterion is met, do another 2 iterations*
      end if
    end for
    $q_i \leftarrow q_i / ||q_i||_2$
  end for
end procedure

Algorithm 6 Modified Gram-Schmidt

procedure MGS(A)
  for $i \leftarrow 1 : n$ **do**
    $A(1:n, i) = A(1:n, i) / ||A(1:n, i)||$
    for $j \leftarrow (i+1) : n$ **do**
      $A(1:n, j) = -\left[A(1:n, j)^\top A(1:n, i)\right] A(1:n, i)$
    end for
  end for
end procedure

absence of floating point errors. Hence the `modified Gram-Schmidt` algorithm does not guarantee fully (to working precision) orthogonal eigenvectors.

To re-orthogonalise a cluster of eigenvectors `dsbein` (using algorithm 6) requires $O(c^2 n)$ flops, where c is the size of the cluster [20, p. 231].

3.1.2. Packed Storage

Throughout this thesis the algorithms are discusses as if stored as dense matrices. For banded matrices, however, only approximately $(2r+1)n$ elements are non-zero. If B is symmetric less than $(r+1)n$ unique elements exist. It would be a waste of memory to store B in full storage if $r \ll n$. One approach which is used in `LAPACK` and thus in the algorithm presented in this section (`dsbein`), stores each (sub)diagonal as a row in an array `AB` which has dimension $(r+1) \times n$. Hence only the upper or lower triangle of the bandsymmetric matrix B is stored. Thereby the j-th column of B is stored in the j-th column of `AB`, resulting in the following formula for storing the upper part of B (see [48, p. 187], [2, p. 142])

$$b_{ij} \longmapsto AB[r+1+i-j, j], \qquad i \leq j. \tag{3.6}$$

The array `LU` in algorithm 5 has a similar storage scheme, but needs to store the complete band of size $(2r+1)n$, see [2, p. 140].

A different approach of reducing the storage requirements for a banded matrix is taken for `dsbkdc`, see section 5.4.

3.2. Divide and Conquer Approaches

Beattie and Fox [7] adapted the Weinstein-Aronszajn theory of intermediate problems to restricted rank modifications of Hermitian matrices. They reviewed three low rank modification of a matrix, namely the *extension*, *restriction* and direct *modification* of a matrix. Furthermore they presented a spectrum slicing method using a *Weinstein matrix* defined for each problem. Similar was developed by Arbenz, Gander and Golub [5] [3] who also outlined a method to computed the eigenvectors for the modified eigenvalue problem. Arbenz [3] applied the theory for *restricted eigenvalue problems* to banded symmetric Teoplitz matrices and investigated difference root-finding methods for the *Weinstein determinant*. Arbenz [4] later proposed the three divide and conquer approaches for the bandsymmetric eigenvalue problem based on the theory of low rank modifications. These algorithms can be seen as a generalisations of Cuppen's tridiagonal divide and conquer algorithm.

Section 3.2.1 provides an overview of the divide and conquer approach and describes the different tearing strategies which yield the rank-r modifications. In section 3.2.2 the main results from [4] are summarised and the *divide and conquer by extension* algorithm, as proposed by [4], is discussed.

Gansterer, Ward and Muller [28] presented a *divide and conquer by modification* algorithm for block tridiagonal matrices with rank-one off-diagonal blocks.[3] Later Gansterer, Ward, Muller *et al.* [27] generalised the algorithm to arbitrary off-diagonal blocks. The resulting algorithm will in the following be denoted as `dsb1dc` and is analysed in section 7.3.2. Haidar, Ltaief and Dongarra [33] parallelised this method using a tile algorithm. Section 3.2.3 summarises the existing results of the *divide and conquer by modification* approach.

3.2.1. Overview

Arbenz [4] described three divide and conquer approaches. They each consist of the following steps:

Divide Part. Partition the matrix B in such a way that two subproblems of similar form emerge, e.g. two band matrices (B_1 and B_2) of lower dimension.

Conquer Part. Solve subproblems B_i yielding the spectral decompositions

$$B_i = Q_i \Lambda_i Q_i^\top, \qquad i = 1, 2,$$

where $\Lambda_i = \operatorname{diag}(\lambda_0^{(i)}, \ldots, \lambda_{b_i-1}^{(i)})$ are the increasingly ordered eigenvalues of the subprobem B_i. Q_i holds the corresponding eigenvectors. This can be done via the same algorithm or using a standard tridiagonalisation based eigensolver.

Assemble Part. Solve the original eigenproblem $Bx = \lambda x$ using the solution of the subproblems. This is the most complex and time consuming part.

The three methods presented by Arbenz [4] differ by their respective strategies to partition and re-assemble the matrices. The subdivision schemes are:

1. Divide and conquer *by modification*:

$$B = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} + D, \quad B_i \in \mathbb{R}^{b_i \times b_i}, \qquad b_1 + b_2 = n \tag{3.7}$$

[3] Note that a bandmatrix can be re-interpreted as a block tridiagonal matrix, see section 5.4.

2. Divide and conquer *by restriction*:

$$B_0 = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix}, \quad B_i \in \mathbb{R}^{b_i \times b_i}, \qquad b_1 + b_2 = n + r \tag{3.8}$$

3. Divide and conquer *by extension*:

$$B = \begin{pmatrix} B_1 & V_1 & 0 \\ V_1^\top & \Delta & V_2^\top \\ 0 & V_2 & B_2 \end{pmatrix}, \qquad B_i \in \mathbb{R}^{b_i \times b_i},\ V_i \in \mathbb{R}^{b_i \times r},\ \Delta \in \mathbb{R}^{r \times r} \tag{3.9}$$

with $b_1 + b_2 + r = n$.

Note that the dimension of the subproblems differs between these schemes. Both the *modified* and *extended* eigenvalue problem are discussed in the following subsections.

The subproblem B_i for the *divide and conquer by restriction* approach are defined in such a way that they both contain the $r \times r$ block

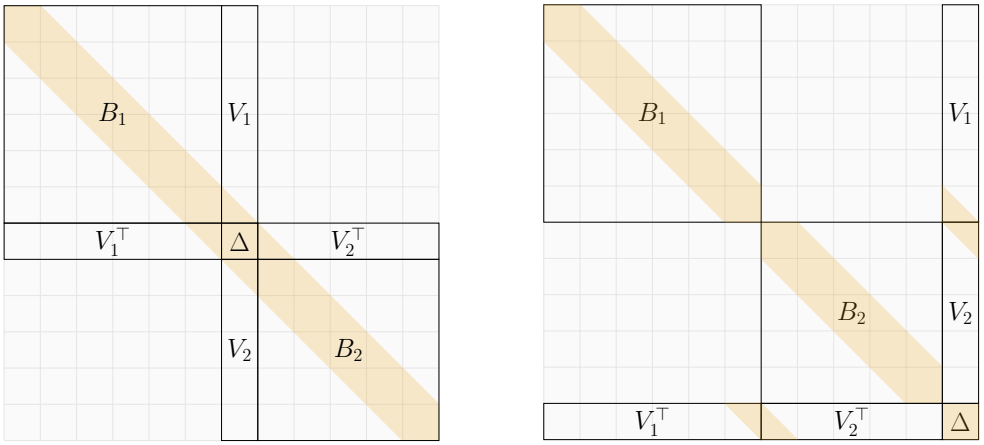
$$\frac{1}{2} \begin{pmatrix} b_{b_1-r,b_1-r} & \cdots & b_{b_1-r,b_1-1} \\ \vdots & & \vdots \\ b_{b_1-1,b_1-r} & \cdots & b_{b_1-1,b_1-1} \end{pmatrix} \tag{3.10}$$

thus the original problem gets "stretched out" until it tears [4]. Beattie and Fox [7] noted that the main difference between a rank-r modification by *restriction* and *extension* is the point of view, hence that they are tightly connected. This relation is also highlighted in [4]. The *divide and conquer by extension* algorithm has the advantage that the subproblems have lower dimension than for the *restricted* problem. To the best of our knowledge no implementation of the low rank modification *by restriction* approach exists.

3.2.2. Low Rank Modifications By Extension - DSBKDC

This section summarises the algorithm proposed in [4], later denoted as `dsbkdc`. *Divide and conquer by extension* uses the partitioning scheme eq. (3.9) where one $r \times r$ block (denoted as Δ) from the main block diagonal and the four neighbouring submatrices V_i are cut out, leaving two band matrices, see fig. 3.1 (left). Note that

$$V_1 = \begin{pmatrix} 0 \\ \hat{V}_1 \end{pmatrix} \quad \text{and} \quad V_2 = \begin{pmatrix} \hat{V}_2 \\ 0 \end{pmatrix} \tag{3.11}$$



(a) B (b) $\bar{B}$

Figure 3.1.: Partitioning for the divide and conquer by extension methods.

with $\hat{V}_1, \hat{V}_2 \in \mathbb{R}^{r\times r}$ being lower and upper triangular respectively. B_i are once again band matrices.

After partitioning the initial problem B the two subproblems B_1 and B_2 need to be solved. This can be done recursively or by using a standard tridiagonalisation based eigensolver. Clearly a recursive approach is favourable, however, only up to the point that a standard method is by faster. See section 5.5 for a discussion of this issue.

Finally the spectral decomposition of B is constructed from the solutions of the subproblems

$$B_i = Q_i \Lambda_i Q_i^\top, \qquad i = 1, 2. \tag{3.12}$$

To located the eigenvalues $\lambda_i \in \sigma(B)$ the similarity of $B - \lambda$ to $B_0 \oplus W(\lambda)$ is used, see theorem 3.2.1. This allows the use of *bisection* using the inertia of $W(\lambda)$ to locate all eigenvalues of B. The corresponding eigenvectors are calculated afterwards using the transformation matrix G as defined in eq. (3.19). To ease the notation the spectral decomposition of

$$\bar{B} = P^\top B P = \begin{pmatrix} B_0 & V \\ V^\top & \Delta \end{pmatrix}, \qquad B_0 = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix}, \quad V = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \tag{3.13}$$

with

$$P = \begin{pmatrix} I_{b_1} & 0 & 0 \\ 0 & 0 & I_r \\ 0 & I_{b_2} & 0 \end{pmatrix}, \tag{3.14}$$

is discussed instead of that of B. As a permutation is simple a change of basis the eigenvalue are the same. If $\bar{x}_i$ is an eigenvector of $\bar{B}$ then $P\bar{x}$ yields the eigenvector for B. Note that this done implicitly in the implementation, see section 5.1. The resulting partition is illustrated in fig. 3.1.

The following lemma (compare with [4]) lays the groundwork for the computation of the spectral decomposition of a rank-k modification.

Lemma 3.2.1. *Let $\lambda \notin \sigma(B_0)$, where $B_0 \coloneqq B_1 \oplus B_2$ and $V = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}$. Then the matrices*

$$\bar{B} - \lambda = \begin{pmatrix} B_0 - \lambda & V \\ V^t & \Delta - \lambda \end{pmatrix} \quad \textit{and} \quad \begin{pmatrix} B_0 - \lambda & 0 \\ 0 & W(\lambda) \end{pmatrix} \tag{3.15}$$

with

$$W(\lambda) = \Delta - \lambda - V^t(B_0 - \lambda)^{-1}V, \tag{3.16}$$

are congruent.

Let λ be an eigenvalue of B_0 of multiplicity μ and let $Q \in \mathbb{R}^{n\times\mu}$ be a matrix with orthonormal columns spanning the eigenspace $\mathcal{N}(B_0 - \lambda)$ corresponding to λ. Then the matrices

$$\begin{pmatrix} B_0 - \lambda & V & 0 \\ V^t & \Delta - \lambda & 0 \\ 0 & 0 & 0_\mu \end{pmatrix} \quad \textit{and} \quad \begin{pmatrix} B_0 - \lambda & 0 \\ 0 & W_e(\lambda) \end{pmatrix} \tag{3.17}$$

with

$$W_e(\lambda) = \begin{pmatrix} \Delta - \lambda - V^t(B_0 - \lambda)^+V & V^tQ \\ Q^tV & 0_\mu \end{pmatrix}, \tag{3.18}$$

are congruent. $Q \in \mathbb{R}^{n\times\mu}$ are the eigenvalues of B_0 spanning $\mathcal{N}(B_0 - \lambda)$.

Proof. To see that the matrices in eq. (3.15) are congruent consider the regular matrix

$$G = \begin{pmatrix} I & -(B_0 - \lambda)^{-1}V \\ 0 & I \end{pmatrix}, \tag{3.19}$$

then it holds that

$$G^\top \begin{pmatrix} B_0 - \lambda & V \\ V^t & \Delta - \lambda \end{pmatrix} G = \begin{pmatrix} B_0 - \lambda & 0 \\ 0 & W(\lambda) \end{pmatrix}. \tag{3.20}$$

and thus *Sylvester's law of interia* (see theorem 2.4.1) shows that the matrices in eq. (3.15) have the same inertia. Analogously the transformation matrix

$$G_e = \begin{pmatrix} I - QQ^t & -(B_0 - \lambda)^+ V & Q \\ 0 & I & 0 \\ Q^t & 0 & 0 \end{pmatrix} \tag{3.21}$$

where $(B_0 - \lambda)^+$ denotes the pseudo inverse of $(B_0 - \lambda)$, show that the matrices in eq. (3.17) are congruent. □

3.2.2.1. Computing the Eigenvalues of B

As the matrices in eq. (3.15) are congruent they have the same inertia and thus for any given λ they have same number of eigenvalues $< \lambda$, in the following denoted as $N(\lambda)$. Let further $N_0(\lambda)$ denotes the number of eigenvalues of B_0 less than λ, then

$$N(\lambda) = \begin{cases} N_0(\lambda) + \nu(W(\lambda)), & \text{if } \lambda \notin \sigma(B_0) \\ N_0(\lambda) + \nu(W_e(\lambda)), & \text{if } \lambda \in \sigma(B_0), \end{cases} \tag{3.22}$$

holds, where $\nu(W)$ denotes the negative inertia of the matrix W. This allows to apply a spectrum slicing method to locate the eigenvalues of B, e.g. by applying *bisection* on $N(\cdot)$.

The following theorem (see [36, p. 205]) gives a method to calculate lower ($\lambda_{-\infty}$) and upper (λ_∞) bounds for the spectrum of B. Together with eq. (3.22) it yields the interlacing property

$$\lambda_{j-r}^{(0)} \leq \lambda_j \leq \lambda_j^{(0)}, \quad 0 \leq j < n, \tag{3.23}$$

with $\lambda_j^{(0)} = \lambda_{-\infty}$ for $j < 0$ and $\lambda_j^{(0)} = \lambda_\infty$ for $j \geq n - r$. This shows that two neighbouring eigenvalues of B_0 have at most r eigenvalues of B in between.

Theorem 3.2.2 (Gerschgorin Circle Theorem)**.** *Let $A \in \mathbb{R}^{n \times n}$ and $\lambda \in \sigma(A)$, then*

$$\lambda \in \bigcup_{i=1}^{n} \left\{ \zeta : |\zeta - a_{ii}| \leq \sum\nolimits_{j \neq i}^{j=1^n} |a_{ij}| \right\}. \tag{3.24}$$

Equation (3.22) already provided a spectrum slicing method, however, once an eigenvalue of B is isolated in an interval (a, b) more efficient root-finding algorithms can be

applied. From eq. (3.19) it is clear that for any $\lambda \notin \sigma(B_0)$

$$\det W(\lambda) = \frac{\det(\bar{B} - \lambda)}{\det(B_0 - \lambda)}, \quad \lambda \notin \sigma(B_0) \tag{3.25}$$

hold, as $\det G = 1$. Hence, $\det W(\lambda)$ has poles at the eigenvalues of B_0 and is zeros for $\lambda \in \sigma(B) \backslash \sigma(B_0)$, that are the eigenvalue of B that are not eigenvalues of B_0. This allows to use a superlinear zerofinder on the *Weinstein determinant* to speed up the root-finding method.

Finally note that if $\det W_e(\lambda) = 0$ then λ is an eigenvalue of B. The process of locating an eigenvalue of B is described in section 3.2.2.3.

3.2.2.2. Computing Eigenvectors of B

The computation of the eigenvector y corresponding to λ is simple, but once again two cases need to be considered. First let $\lambda \notin \sigma(B_0)$ be an eigenvalue of B. Then the transformation matrix G maps the nullspace of $(B_0 - \lambda) \oplus W(\lambda)$ onto the nullspace of $\bar{B} - \lambda$, thus

$$y \coloneqq \begin{pmatrix} y1 \\ y2 \end{pmatrix} \coloneqq G \begin{pmatrix} 0 \\ x_2 \end{pmatrix} = \begin{pmatrix} -(B_0 - \lambda)^{-1} V x_2 \\ x_2 \end{pmatrix}, \tag{3.26}$$

with

$$x_2 \in \mathcal{N}(W(\lambda)). \tag{3.27}$$

For the second case let $\lambda \in \sigma(B_0)$ be an eigenvalue of B. Then G_e (see eq. (3.21)) maps $(B_0 - \lambda) \oplus W_e(\lambda)$ onto the nullspace of $\bar{B} - \lambda$, thus

$$y \coloneqq \begin{pmatrix} y1 \\ y2 \end{pmatrix} \coloneqq G_e \begin{pmatrix} 0 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -(B_0 - \lambda)^{+} V x_2 + Q x_3 \\ x_2 \end{pmatrix}, \tag{3.28}$$

with

$$\begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \in \mathcal{N}(W_e(\lambda)). \tag{3.29}$$

3.2.2.3. The Assemble Algorithm

Algorithm 7 (compare with [4]) describes one merging operation, that is one rank-r modification. The algorithm start by merging the eigenvalues of the subproblems B_i in line 2 which yields the spectrum of B_0 in line 3. Next bounds for spectrum of B are

calculated in line 4, e.g. using Gershgorin circles. Lines 5 to 10 iterate over all eigenvalues $\lambda_j^{(0)}$ of B_0 and calculate both the corresponding *extended Weinstein matrix* and its inertia. Note that the inertia is later needed to compute M in line 12. If $W(\lambda_j^{(0)})$ is singular, then $\lambda_j^{(0)} \in \sigma(B)$ and the corresponding eigenvector can be computed using eq. (3.28), see line 8.

Algorithm 7 assemble

1: **procedure** ASSEMBLE($\Delta, V, Q_1, \Lambda_1, Q_2, \Lambda_2$)
2: Merge and sort $\lambda_j^{(1)} \in \Lambda_1$ and $\lambda_j^{(2)} \in \Lambda_2$ into

$$\lambda_0^{(0)} \leq \lambda_1^{(0)} \leq \cdots \leq \lambda_{b_1+b_2-1}^{(0)}.$$

3: Set $m = |\sigma(B_0)|$. The spectrum of B_0 is $\{\lambda_j^{(0)}\},\ j = 0, \ldots, m-1$ with

$$\hat{\lambda}_0^{(0)} < \hat{\lambda}_1^{(0)} < \cdots < \hat{\lambda}_{m-1}^{(0)}.$$

4: Determine $\lambda_{-\infty}$ and λ_∞ such that ▷ *Using theorem 3.2.2*

$$\lambda_{-\infty} \leq \lambda_j \leq \lambda_\infty \qquad \forall j = 0, \ldots, n-1.$$

5: **for** $j \leftarrow 0, m-1$ **do** ▷ *iterates over all* $\lambda_j^{(0)} \in \sigma(B_0)$
6: Compute $W_e(\hat{\lambda}_j^{(0)})$ and its inertia (π_j, ν_j, ζ_j)
7: **if** $W_e(\hat{\lambda}_j^{(0)})$ singular **then** ▷ *thus* $\hat{\lambda}_j^{(0)} \in \sigma(B)$.
8: (i) Determine corresponding eigenvectors, according to eq. (3.28).
9: **end if**
10: **end for**
11: **for** $j \leftarrow 0, m$ **do** ▷ *iterates over all intervals* $(\hat{\lambda}_{j-1}^{(0)}, \hat{\lambda}_j^{(0)})$
12: $M = N(\lambda_i^{(0)}) - [N(\lambda_{i-1}) + \zeta(W_e(\lambda_{i-1}^{(0)}))]$ ▷ *using eq.* (3.22)
13: **if** $M > 0$ **then**
14: (i) Isolate eigenvalue in the interval. ▷ *bisection*
15: (ii) Determine the isolated eigenvalue. ▷ *superlinear zerofinder*
16: (iii) Compute corresponding eigenvectors, according to eq. (3.26).
17: **end if**
18: **end for**
19: **end procedure**

In lines 11 to 18 the algorithm iterates over all intervals $(\hat{\lambda}_{j-1}^{(0)}, \hat{\lambda}_j^{(0)})$. For each interval first the number M of eigenvalues $\lambda \in \sigma(B)$ located inside the interval is calculated in line 12. If $M > 1$ the algorithm applies *bisection* using the inertia of $W(\lambda)$ to find M intervals each containing one eigenvalue, e.g. line 14. Once the eigenvalues are isolated

the superlinear root-finding method, called *zeroin* (see section 5.2), is used to locate the eigenvalues (line 15). The corresponding eigenvectors are finally calculated according to eq. (3.26) in line 16.

3.2.2.4. Complexity of DSBKDC

It is left to asses the operation count of the *divide and conquer by extension* algorithm. For the first rank-r modification, that is the initial subdivision of the problem $B \in \mathbb{R}^{n\times n}$ with subproblems of size $b_1 \approx b2 = n/2$, the main operations are:

- Computation of the spectral decomposition of the subproblems. Using `L/dsyevd` this amounts to approximately $8(\frac{n}{2})^3/3$ flops for the tridiagonalisation, $2(\frac{n}{2})^3$ for the back-transformation and a negligible cost for the computation of the eigen-decomposition of the tridiagonal eigenproblem, see section 2.3.1.1, section 2.4.2.3 and fig. 2.1. In total not more than $6(\frac{n}{2})^3$ flops are needed.

- Locating the eigenvalues. For each eigenvalue c *Weinstein matrices* and their inertia need to be computed, where c is the average number of iterations needed to located an eigenvalue. The computation of $W(\cdot)$ takes $\mathcal{O}(nr^2)$ flops and the calculation of its inertia $O(r^3)$. Hence for $r \ll n$ and n eigenvalues this amounts to approximately cn^2r^2 flops.

- Calculation of the eigenvectors. For one eigenvector the most expensive operation is the back-transformation, e.g. the multiplication with $Q = Q_1 \oplus Q_2$ in eq. (3.26). Hence for n eigenvectors this amounts to n^3.

Arbenz [4] analysed the complexity for a recursion depth of q. He concluded that the n^3 term quickly tends to $4/3$ as $p = 2^q$ get large. This runtime complexity is extremely promising especially since the n^3 stems from two `dgemm` call per rank-r update, and is thus well optimised. However, in our experimental evaluation `dsbkdc` has a runtime complexity below $\mathcal{O}(n^3)$ up to $n = 10200$.[4] This is due the expensive calculations of the *Weinstein matrices* that dominate the runtime.

3.2.3. Low Rank Modifications By Modification - DSB1DC

The *divide and conquer by modification* approach partitions B as defined in eq. (3.7). Arbenz [4] described an approach of calculating the eigenpairs of B using r rank-one modification instead of one rank-r update. Gansterer, Ward, Muller *et al.* [27] refined this

[4] Larger matrices were not tested.

approach and provided a sequential implementation, which is in the following denoted as `dsb1dc`. Note that these rank-one modification are well studied, as they are also used in *Cuppen's divide and conquer algorithm*. This has two advantages: Firstly, it allows for the use of theorem 2.4.4 and hence for a stable computation of the eigenvalues. Secondly, *deflation* techniques can be applied, which have shown to greatly reduce the runtime-complexity of the tridiagonal analogue. The partition eq. (3.7) can be rewritten to

$$B = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} + Z\Sigma Z^\top = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} + \sigma_1 z_1 z_1^\top + \sigma_2 z_2 z_2^\top + \cdots + \sigma_r z_r z_r^\top, \qquad (3.30)$$

see [33]. However, that approach of using r rank-one modifications has the disadvantage of needing several consecutive eigenvector transformation that quickly dominate the runtime as r increases. In the worst case, that is when no *deflation* occurs, the algorithm proposed by Gansterer, Ward, Muller *et al.* [27] requires $r-1$ full matrix-matrix multiplications. The back-transformation of the eigenvectors for one rank-r update is thus approximately $2r-1$ times as expensive as for the `dsbkdc` algorithm. The total complexity of `dsb1dc` is dominated by the back-transformation of the eigenvectors in the last merging operation. Assuming no deflation this last update and thus the algorithm has complexity $2(r-1)n^3$.

A parallel version of `dsb1dc` it was implemented by Haidar, Ltaief and Dongarra [33].

3.3. Comparison of the Approaches

In this section the runtime complexity of a tridiagonal eigensolver, as e.g. available in `LAPACK`, is compared to the divide and conquer methods presented in this chapter. A tridiagonalisation via *Householder reductions* takes $8n^3/3$ flops, see section 2.3.1.1, and the back-transformation at most $2n^3$ flops. The costs for the tridiagonal eigensolver are negligible in comparison (see section 2.4.2.3 and fig. 2.1), yielding an overall complexity of not more than $6n^3$ for modern tridiagonal based eigensolvers.

The algorithm `dsb1dc` has a worst case complexity of $2(r-1)n^3$, however, it has shown to be much faster for matrices with clustered eigenvalues [41]. The complexity suggests, that the algorithm `dsb1dc` is promising for small bandwidths, however, large amounts of *deflation* need to occur to keep it competitive for medium and larger bandwidths. Note that the approach of of using r rank-one update has the advantage the a numerical stable method to compute the eigenvectors is given by theorem 2.4.4.

On the other hand `dsbkdc` uses one rank-r update. This yields a much lower n^3 term in the runtime complexity, e.g. $4/3n^3$ for a recursive approach [4]. The runtime complexity

is hence independent from r. The dependency on r is given with $\mathcal{O}(n^2r^2)$ for the n^2 term. Since the n^2 term (which stems from the computation of the *Weinstein matrices*) takes up the majority of the actual runtime for matrices up to $n = 10200$, increasing r has nevertheless a large impact on the runtime.

The disadvantage of `dsbkdc` is that it is yet unclear how to avoid the numerical instabilities that occur during the computation of the eigenvectors. Especially the orthogonality of the resulting eigenvectors suffers for test matrices with tightly clustered eigenvalues.

4. Infrastructure for Numerical Experiments

This thesis evaluates the *divide and conquer by extension* approach using numerical experiments. In chapter 5 the baseline implementation of `dsbkdc` as well as performance optimisations are covered. The accuracy is analysed in chapter 6 and improvements are implemented. The final versions[1] of the algorithm are compared to modern tridiagonalisation based eigensolvers, see chapter 7. All of the above steps require well chosen test problems of variable hardness. In this chapter the testing setup as well as the test matrices used throughout this thesis are introduced.

4.1. Platform & Software

All algorithms are run single threaded on one CPU core with fixed frequency. It was tried to minimise CPU-migrations and context-switches by shielding one CPU-Core using the `cset-shield` command, however, this made no difference. To compensate for remaining fluctuations in runtime and ensure consistent measurements all computations are done 4 times and the fastest measured time is logged. A parallel implementation is out of the scope of this thesis as the numerical instabilities of the algorithm, as reported by [4], need to be addressed first. Parallelism is only used to check the results, e.g. via multithreaded `BLAS` operations when computing the residual or when computing the spectral decomposition via the `PLASMA` library to compare the computed eigenvalues to those computed by `P/dsyedc`.

Calculations are performed in double-precision (except for the extended precision variant of `dsbkdc`, see section 6.4). As pointed out by Moldaschl and Gansterer [41] subnormal numbers may have a significant impact on tridiagonalisation-base algorithms. This issue is discussed in more detail in section 4.1.2. To disable subnormal number the denormals-are-zero (DAZ) and flush-to-zero (FTZ) flags are enabled via the `_mm_setcsr` command provided by `gcc`.

[1] One implementation using only double-precision and a slightly modified version using extended precision to improve the accuracy.

Table 4.1.: Platform

CPU	Intel i7-4770K Haswell (3.5GHz)
Memory	16 GB
OS	Ubuntu
Compiler	gcc
Timer	`omp_get_wtime()`
BLAS	Openblas r0.2.20
LAPACK	3.7.0 (OpenBLAS)
PLASMA	2.8.0

Table 4.1 summarises the most important characteristics of the system used during development and testing.

4.1.1. Extended Precision

In section 6.4 the quadruple-precision datatype (`__float128`) is used to locate the zero of a function to higher precision. It has an exponent width of 15 bits and a mantissa of 112 bits. The `GCC` library `libquadmath` [49] needs to be included to use this datatype.

4.1.2. A Note on Subnormal Numbers

Subnormal numbers are numbers which are smaller than the smallest normalised number, that is $2^{-1022} \approx 2.2 \cdot 10^{-308}$ in double precision [42]. By considering only normalised numbers the next larger number is $2^{-1022} + 2^{-1074}$, whereas the next smaller number is zero. That is because the exponent cannot get smaller. Subnormal numbers close this gap. The IEEE standard specifies that a number with an all-zero exponent bitstring E and an nonzero fraction bitstring $b_1 b_2 ... b_n$ should be treated as $0.b_1 b_2 ... b_n \cdot 2^{-1022}$ [42]. Including subnormal numbers the next smaller number of 2^{-1022} is $\sum_{i=1023}^{1074} 2^{-i} \approx 2^{-1022} - 2^{-1074}$ in double precision.

Unfortunately there is rarely hardware support for these special numbers making calculations with them extremely time consuming. As it turns out a large quantity of subnormal numbers occur during the tridiagonal reduction process of a dense matrix. However, this only happens if Q is accumulated and not for all matrices to the same extent. The impact of software emulated operations on the runtime can be seen in table 4.2. The most notably increase in runtime can be made out in the `dgemm` operation for the back-transformation of the computed eigenvectors, followed by the tridiagonal

Table 4.2.: Runtime in seconds of `L/dsyevd` with subnormal numbers enabled/disabled, for the test matrix `G1_10200_24_16228`, that is a matrix with dimension $n = 10200$ and semi-bandwidth $r = 24$.

	subnormal numbers enabled	subnormal numbers disabled
total runtime	386.2	202.7
tridiagonalisation step	210.3	114.1
tridiagonal eigensolver	13.1	13.1
back-transformation step	162.8	75.6
orthogonality error	$1.07137 \cdot 10^{-14}$	$1.07137 \cdot 10^{-14}$
residual	$2.71708 \cdot 10^{-15}$	$2.71708 \cdot 10^{-15}$

reduction step. In contrast there seems to be no difference in either the runtime for the tridiagonal eigensolver nor does the quality of computed spectral decomposition change. Similar slowdowns were reported by Moldaschl and Gansterer [41].

In the testing program subnormal numbers are disabled which also affects called libraries.

4.1.3. The Testing Program

The build automation tool `make` is used to generate the testing program. The source code of the main program as well as any helper function – e.g. functions to manipulate matrix-files or functions to debug test data – are compiled without compiler optimisations. Algorithms that need to be compiled (e.g. `dsb1dc`, `dsbkdc` and `dsbein`) are each compiled with `-O3` and `-march=native`.

The most important parameters of the executable are: `matrixDir`, `nr`, `logFile` and `alg`. The program proceeds as follows: First it initialises some global variables, e.g. setting the number of threads of `OpenBLAS` to one. Afterwards it processes the matrix-files located in `matrixDir` one by one. For each matrix the program first allocates the workspace needed by the algorithm being tested (here denoted as `alg`) and computes the spectral decomposition. The computation is done `nr` times logging only the fastest measured time. The program continues by computing the residual, orthogonality error and optionally $\max_i |\hat{\lambda}_i - \lambda_i|$ where $\hat{\lambda}_i$ are the eigenvalues calculated with `P/dsyedc` and stores the results in `outputDir/logFile`. Possible values for `alg` are listed in section 7.1.

4.2. Test Data

To implement and evaluate the eigensolver described in section 3.2 well chosen test-matrices are needed. For an initial implementation (see chapter 5) banded matrices with elements randomly drawn from $(0, 1)$ are used. To analyse the accuracy in chapter 6 more sophisticated test data are needed. Therefore the test matrices defined in [40] are adapted.

The test data can be categorised into two types of synthetic test matrices which are listed below.

1. Matrices with random entries:

   (R1) Random entries drawn from a uniform distribution on $(0, 1)$, as used in [4].

   (R2) Random entries drawn from a uniform distribution on $(0, 1)$. The resulting matrix $\hat{B}$ is scaled by $1/||B||_2$, yielding

   $$B = \frac{\hat{B}}{\hat{\lambda}_{n-1}} \tag{4.1}$$

   with $\hat{\lambda}_i \in \sigma(\hat{B})$ being the increasingly ordered eigenvalues of $\hat{B}$.

2. Matrices with fixed eigenvalue distributions, as defined in Marques, Vömel, Demmel *et al.* [40]:

   - Strongly clustered eigenvalues:

     (S1) $\lambda_0 = 1,\ \lambda_i = \varepsilon,\ i = 1, 2, \ldots, n-1$

     (S2) $\lambda_0 = \varepsilon,\ \lambda_i = 1,\ i = 1, 2, \ldots, n-1$

   - Weakly clustered eigenvalues:

     (W1) $\lambda_0 = 1,\ \lambda_i = i\varepsilon,\ i = 1, 2, \ldots, n-1$

     (W2) $\lambda_0 = \varepsilon,\ \lambda_i = 1 + i\sqrt{\varepsilon}, i = 1, 2, \ldots, n-2,\ \lambda_{n-1} = 2$

     (W3) $\lambda_i = 1 + 100i\varepsilon,\ i = 0, 1, \ldots, n-1$

   - Geometric distributions:

     (G1) $\lambda_i = \varepsilon^{\frac{i}{n-1}}, i = 0, 1, \ldots, n-1$

     (G2) $\lambda_i = \varepsilon^{x_i}$, where x_i is drawn from a uniform distribution on $(\varepsilon, 1)$

   - Uniform distributions:

     (U1) $\lambda_i = 1 - \left(\frac{i}{n-1}\right)(1 - \varepsilon),\ i = 0, 1, \ldots, n-1$

(U2) $\lambda_i = x_i$, where x_i is drawn from a uniform distribution on $(\varepsilon, 1)$

The parameter ε is set to $\sqrt{\epsilon_{double}}$ where ϵ_{double} is the double precision machine epsilon. For the analysis for eigenvalues with multiplicities $\mu > 1$ in section 6.5 matrices with $\varepsilon = \epsilon_{double}$ will be used, however, this will be indicated in the particular sections. Before generating the banded matrices from the fixed eigenvalue distributions the eigenvalues λ_i are randomly negated.

The generation of the test matrices is done by `L/dlagsy`. This `LAPACK` routine pre- and post-multiplies the diagonal matrix $\Lambda = \text{diag}(\lambda_0, \ldots, \lambda_{n-1})$ with a random orthogonal matrix U yielding a dense symmetric matrix $S = U\Lambda U^\top$. In a final step S is reduced to banded form B using *Householder transformations*.

The generated matrices are saved as plain text files in the *Matrix Market Exchange Format* for sparse matrices [13]. The first line (starting with %%) contains a sequence of keywords. Following lines marked with % are comments. The first non-comment line contains three integers which give the number of rows, columns and nonzero elements in the matrix. Each line thereafter defines one element. The line starts with two integers defining the coordinates (i, j) followed by the floating point value of that element. The naming scheme of a matrix file is `TYPE_N_R_SEED` where `N` and `R` specify the dimension and the semi-bandwidth respectively. `TYPE` denotes the matrix type as defined above, e.g. `R2` and `SEED` is the seed value used in the generation process.

On advantage of using such an easy format is that the generated matrices can easily be imported into `GNU Octave` to double check the correctness of the generation process as well as that of the testing program. The eigenvalue distribution of two generated matrices is displayed in fig. 4.1. In the upper half of each plot a histogram shows the distribution of the eigenvalues of B. Below the specific eigenvalues are plotted using points $(\hat{\lambda}_i, x)$, where $x \in (0, 1)$ is a random number that provides the vertical spread and $\hat{\lambda}_i$ is the computed eigenvalue using `GNU Octave`. Note that $\hat{\lambda}_i$ does generally not equal λ_i. The spectral radii of `R1` are larger than those of any other matrix type defined above. They can be estimated using theorem 3.2.2 with $||B||_2 \approx r + 1$ (more precisely: $\lim_{n\to\infty} ||B||_2 = r + 1$) where r is the semi-bandwidth, compare fig. 4.1 (left). All other types have a fixed spectral radius of $||B||_2 \in \{1, 2\}$. As a consequence the test problems get harder for increasing n as the gap between eigenvalues decreases, see theorem 2.1.2. This is most notable for the method of *inverse iteration*, see chapter 7.

The program used to generate the test data is written `C++11`. One can specify the semi-bandwidth `r`, the dimension `n`, the distribution of the eigenvalues `type` and the random-seed `seed`. If `seed` is set to zero a random seed is used.

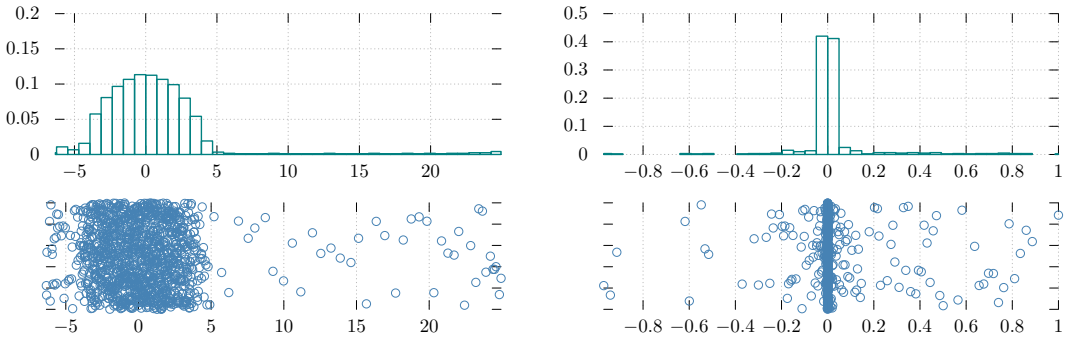


Figure 4.1.: Distribution of eigenvalues of `R1_1200_24_31866` (left) and `G1_600_3_23709` (right) as calculated by `GNU Octave`. The blue points (λ, x), $x \in (0, 1)$ random, each represent an eigenvalue λ. Above the corresponding histogram is plotted.

4.3. Evaluation Metrics

For a test matrix B the computed spectral decomposition is given by $Q\Lambda Q^\top$, with $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ containing the computed eigenvalues and $Q = (q_1, q_2, \ldots, q_n)$ the corresponding eigenvectors. To evaluate the quality of the result the residual

$$\mathfrak{R}(B) = \mathfrak{R}(B, \Lambda, Q) = \max_i \frac{||Bq_i - \lambda_i q_i||_2}{||B||_2} \tag{4.2}$$

and the orthogonality error

$$\mathfrak{O}(B) = \mathfrak{O}(Q) = \max_{i \neq j} |q_i^\top q_j| \tag{4.3}$$

are computed. These metrics are similar to those used in Demmel, Marques, Parlett *et al.* [19], however, they are not scaled by ϵ nor by the dimension n. The metrics are chosen to be independent of n and ϵ to make it easier to estimate the quality of inaccurate results.

5. Towards an Efficient Implementation of DSBKDC

Chapter 5 and chapter 6 give an overview of the process of implementation starting by an initial implementation in section 5.1, which yields the baseline algorithm `dsbkdcV0`.[1] This chapter is mainly concerned with runtime improvements, whereas chapter 6 will focus on improving the accuracy of the computed spectral decomposition. As a first optimisation step section 5.2 will introduce the superlinear root-finder *zeroin*, as already mentioned in section 3.2.2. The largest speedup is achieved by blocking the eigenvector transformation which is implemented in `dsbkdcV2`, see section 5.3. In section 5.4 the storage scheme is changed to a more efficient one and thus reduces the space requirements. Section 5.5 further improves the runtime by applying the root-finding method to slightly modified functions. Section 5.6 concludes the chapter by discussing the potential of a recursive approach.

In this chapter the algorithm is analysed using test data of type `R2` (see section 4.2), as this matrix type was used during development. The integration of further test matrices is done in chapter 6. Throughout the chapter different plot are used to illustrate the effect each improvement has on the runtime. For the complete picture of the final algorithm, both regarding runtime and accuracy see section 7.3.3.

5.1. Version 0 - A Baseline Implementation

The algorithm is implemented in `C++11`, however, most of the code is plain C. Where possible `BLAS` operations (using the c-interface from `OpenBLAS`) are used to achieve good performance. Furthermore `LAPACK` (the fortran-interface from `OpenBLAS`) is used to solve smaller problems. More precisely, the eigendecompostion of the subproblems B_1 and B_2 as well as those of the $r \times r$ *Weinstein matrices* are solved using `L/dsyevd`. As described

[1] In the following the different version are denoted by `dsbkdcVx`, where the appended `Vx` stands for *Version x*. In chapter 6 the versions are denoted by appending `Ax` (*Accuracy x*) to highlight the focus on accuracy improvements.

in section 3.2 the algorithm is recursive and thus the subproblems could also be solved using `dsbkdc`, see section 5.6 for a discussion thereof.

To start off fig. 5.1 presents an overview over all functions and their interplay used in this implementation. The testing program calls `dsbkdcV0` which has a `LAPACK` like interface. The parameters are listed in table 5.1.[2] Note that the input matrix B is left unchanged. This could be optimised to store the eigenvectors in B instead, however in section 5.4 the storage scheme will be changed to a packed one where this optimisation is no longer possible. The workspace requirements are $\mathcal{O}(n^2)$, which is mainly needed to store the spectral decomposition of the subproblems. The size of `work` in multiple of floating point numbers is

$$\frac{\texttt{sizeof(work)}}{\texttt{sizeof(double)}} = \underbrace{b_1^2 + b_2^2}_{\text{eigenvectors of subproblems}} + \underbrace{2b_1 + 2b_2 + 2n}_{\text{eigenvalues of subproblems}} \tag{5.1}$$

$$+ \underbrace{2b_1 r + 2b_2 r}_{U_1 \text{ and } U_2} + \underbrace{(r+1)(r+2)}_{\textit{Weinstein matrix}} \tag{5.2}$$

$$+ \underbrace{2b_2^2 + 6b_2 + 1}_{\text{workspace for } \texttt{LAPACK}} . \tag{5.3}$$

The workspace needed by `LAPACK` is bound by the requirements of `L/dsyevd` for the larger subproblem, namely B_2. Similarly the size of `iwork` in multiple of integers is

$$\frac{\texttt{sizeof(iwork)}}{\texttt{sizeof(int)}} = \underbrace{2n}_{\text{indices of } \lambda_i^{(0)} \text{ in } \Lambda_1 \text{ or } \Lambda_2} + \underbrace{2(b_1 + b_2)}_{\text{inertia of } \lambda_i^{(0)} \in B_0} \tag{5.4}$$

$$+ \underbrace{5b_2 + 3}_{\text{integer workspace for } \texttt{LAPACK}} \tag{5.5}$$

5.1.1. The Main Procedure

See algorithm 8 for an overview of `dsbkdcV0`. First the size of the subproblems B_1 and B_2 are defined and corresponding pointers to the sub-matrices are set. As can be seen in line 2, B_0 is split such that the both subproblems are as small as possible:

$$b_1 = \left\lfloor \frac{n-r}{2} \right\rfloor, \qquad b_2 = \begin{cases} b_1 & \text{if } (n-r)/2 \text{ is even,} \\ b_1 + 1 & \text{if } (n-r)/2 \text{ is odd.} \end{cases} \tag{5.6}$$

[2] An additional two parameters are used for debugging and are not listed.

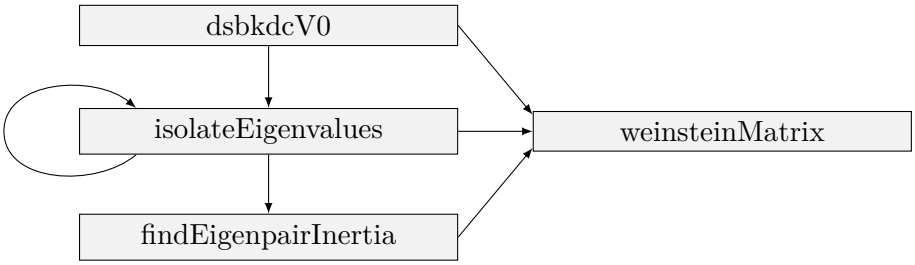


Figure 5.1.: Call graph for `dsbkdcV0`. Each rectangle represents a c-function. An arrow indicates a dependency, e.g. `dsbkdcV0` has calls to both `weinsteinMatrix` and `isolateEigenvalues`.

Table 5.1.: Parameters of `dsbkdcV0`.

type & name		size	purpose
`int n`	in	1	dimension of B
`double *A`	in	$n \times n$	band matrix in full storage
`int lda`	in	1	leading dimension of B
`int r`	in	1	semi-bandwidth of B
`double *Q`	out	$n \times n$	on exit: contains eigenvectors
`double *Lambda`	out	n	on exit: contains eigenvalues
`double *work`	out	$\mathcal{O}(n^2)$	double precision workspace
`int *iwork`	out	$\mathcal{O}(n)$	integer workspace

Algorithm 8 dsbkdcV0

1: **procedure** DSBKDCV0(n, B, r, Q, Λ, work, iwork)
2: $b_1 = (n - r)/2$
3: $b_2 = (n - r)/2 + (n - r) \bmod 2$
4: Compute the spectral decomposition of the subproblems B_i via `L/dsyevd`:

$$B_i = Q_i \Lambda_i Q_i^\top \quad i = 1, 2.$$

5: Build U_1 and U_2 using `dgemm`.
6: Merge and sort $\lambda_j^{(1)} \in \Lambda_1$ and $\lambda_j^{(2)} \in \Lambda_2$ to get $\sigma(B_0)$.
7: Determine bounds $\lambda_{-\infty}$ and λ_∞ for $\sigma(B)$ using Gerschgorin circles.
8: **for** $j \leftarrow 0, m - 1$ **do**
9: Compute the *Weinstein matrix* $W_e(\hat{\lambda}_j^{(0)})$ and its inertia (π_j, ν_j, ζ_j).
10: **if** $W_e(\hat{\lambda}_j^{(0)})$ singular **then**
11: Determine corresponding eigenvectors.
12: **end if**
13: **end for**
14: **for** $j \leftarrow 0, m$ **do**
15: Call `isolateEigenvalues`.
16: **end for**
17: **end procedure**

Next the subproblems are solved using `L/dsyevd` which uses *Cuppen's dived and conquer algorithm*, see section 2.4.2. This driver-routine reduces a full matrix to tridiagonal form and thus cannot take advantage of the sparse structure of the subproblems B_i. However, as discussed in section 2.3 (e.g. fig. 2.1), reducing a full matrix to tridiagonal form using `L/dsytrd` is faster than using `L/dsbtrd` which preserves the banded structure throughout the computation. Of course this is only true if the transformation matrices are accumulated. In this algorithm the transformation matrices Q_i are needed for the back-transformation of the eigenvalues, making `L/dsyevd` the best suited algorithm available in `LAPACK`.

In line 5 the matrices $U_1 = Q_1^\top V_1$ and $U_2 = Q_2^\top V_2$ are computed, which are needed for the computation of a *Weinstein matrix*

$$W(\lambda) = \Delta - \lambda - V^t (B_0 - \lambda)^{-1} V \tag{5.7}$$

$$= \Delta - \lambda - U^t ((\Lambda_1 - \lambda I_{b1}) \oplus (\Lambda_2 - \lambda I_{b2}))^{-1} U, \qquad U = \begin{pmatrix} Q_1^\top V_1 \\ Q_2^\top V_2 \end{pmatrix}. \tag{5.8}$$

U_1 and U_2 have dimension $b_1 \times r$ and $b_2 \times r$ respectively. They are calculated via two

`dgemm` calls. Note that V_i can be treated as a $r \times r$ matrix, as the rest of V_i is zero, see fig. 3.1. More precisely, only the last r columns of $Q_1^\top$ and first r columns of $Q_2^\top$ are used. Building U takes little time, however, a significant part of the whole runtime is spent on computing *Weinstein matrices*. Therefore an optimised routine to compute $W(\cdot)$ is essential. This routine is called `weinsteinMatrix`, see section 5.1.5.

In line 6 the eigenvalues of B_i are merged to get the spectrum of B_0. It may happen that $\lambda_i^{(g)} = \lambda_j^{(h)},\ g, h \in \{1, 2\}$ in which case the corresponding eigenvalue $\lambda_i^{(0)}$ of B_0 has multiplicity $\mu > 1$.[3] For such an eigenvalue the dimension of the *extended Weinstein matrix* changes to $(r + \mu) \times (r + \mu)$, see theorem 3.2.1. This could also be applied to very close eigenvalues of B_0. More precisely if two eigenvalues $\lambda_i^{(h)} \in \sigma(B_h)$ and $\lambda_j^{(g)} \in \sigma(B_g)$ are close, e.g.

$$|\lambda_i^{(h)} - \lambda_j^{(g)}| < 5 \cdot |\min(\lambda_i^{(h)}, \lambda_j^{(g)})| \cdot \epsilon_{double},$$

they could be interpreted as one eigenvalue of B_0 with multiplicity $\mu > 1$. For now this it not relevant since the eigenvalues of the subproblems are always well separated for matrices of type `R2`.[4] Note that this is not the case for highly clustered test data. Section 6.5 will discuss these cases.

As a next step (line 7) bounds for the spectrum of B are calculated. These bounds are denoted as $\lambda_{-\infty}, \lambda_\infty$ and calculated using theorem 3.2.2:

$$\lambda_{-\infty} = \min_i b_{ii} - \sum_{j \neq i} |b_{ij}|, \tag{5.9}$$

$$\lambda_\infty = \min_i b_{ii} + \sum_{j \neq i} |b_{ij}|. \tag{5.10}$$

They are chosen such that

$$\forall \lambda \in \sigma(B): \quad \lambda \in (\lambda_{-\infty}, \lambda_\infty) \tag{5.11}$$

holds. By setting $\lambda_j^{(0)} = \lambda_{-\infty}$ for $j < 0$ and $\lambda_j^{(0)} = \lambda_\infty$ for $j \geq n - r$ one gets the interlacing properties

$$\lambda_{j-r}^{(0)} \leq \lambda_j \leq \lambda_j^{(0)}, \qquad 0 \leq j < n, \tag{5.12}$$

see eq. (3.23). Hence the points

$$\lambda_{-\infty} < \lambda_0^{(0)} < \cdots < \lambda_{m-1}^{(0)} < \lambda_\infty \tag{5.13}$$

[3] Here $\lambda_i^{(h)}$ denoted the eigenvalues of the subproblem $\sigma(B_h)$.

[4] For all tested matrices of type `R2` the eigenvalues of the subproblems are separated by at least 10^3 floating point numbers.

have at most r eigenvalues of B in between each neighbouring pair. In lines 8 to 13 the algorithm iterates over these points and calculates the inertia[5] of the corresponding *extended Weinstein matrix* $W_e(\lambda_i^{(0)})$ using `L/dsyev`. In case $W_e(\lambda_i^{(0)})$ is singular the eigenvector(s) need to be calculated. The computation of the eigenvectors is covered in section 5.1.4. Let

$$\hat{\omega}_i \coloneqq \min_j |\omega_j| \in \sigma(W_e(\lambda_i^{(0)})) \tag{5.14}$$

denote the minimal absolute eigenvalues of the *extended Weinstein matrix* corresponding to λ_i. Then $W_e(\lambda_i^{(0)})$ is considered to be singular if

$$\hat{\omega}_i < \epsilon, \tag{5.15}$$

where ϵ denotes the double precision machine epsilon, see fig. 5.2 (1.). If multiple eigenvalues of the *(extended) Weinstein matrix* are singular to working precision the eigenvalue $\lambda_i^{(0)}$ has multiplicity $\mu > 1$. The eigenvectors are in this case calculated using the corresponding eigenvectors of $W_e(\lambda_i^{(0)})$, see eq. (3.28).

To calculate the remaining eigenvalues of B the algorithm iterates over all neighbouring pairs of eq. (5.13), see line 14. For each interval $(\lambda_{i-1}^{(0)}, \lambda_i^{(0)})$ `isolateEigenvalues` is called with

$$\begin{aligned} M &= N(\lambda_i^{(0)}) - [N(\lambda_{i-1}) + \zeta(W_e(\lambda_{i-1}^{(0)}))] && (5.16)\\ &= \nu(W_e(\lambda_i)) + N_0(\lambda_i^{(0)}) - [\nu(W_e(\lambda_{i-1})^{(0)}) + N_0(\lambda_{i-1}^{(0)}) + \zeta(W_e(\lambda_{i-1}^{(0)}))] && (5.17)\\ &= \nu(W_e(\lambda_i^{(0)})) + i - [\nu(W_e(\lambda_{i-1}^{(0)})) + i - 1 + \zeta(W_e(\lambda_{i-1}^{(0)}))], && (5.18) \end{aligned}$$

to locate the eigenvalue in that interval, see fig. 5.2 (2.). M denotes the number of eigenvalues in that interval. $N(\lambda)$ and $N_0(\lambda)$ are the number of eigenvalues that are $< \lambda$ of B and B_0 respectively. For the first and last interval

$$M = \begin{cases} \nu(W_e(\lambda_0^{(0)})) & \text{for } (\lambda_{-\infty}, \lambda_0^{(0)}),\\ n - [\nu(W_e(\lambda_i^{(0)})) + i - 1] & \text{for } (\lambda_m^{(0)}, \lambda_\infty). \end{cases} \tag{5.19}$$

[5] The inertia consists of (π, ν, ζ) with π and ν being the number of positive and negative eigenvalues respectively and ζ being the dimension of the kernel. In the following $\nu(M)$ and $\zeta(M)$ will denote the number of positive and zero (to working precision) eigenvalues of the matrix M respectively.

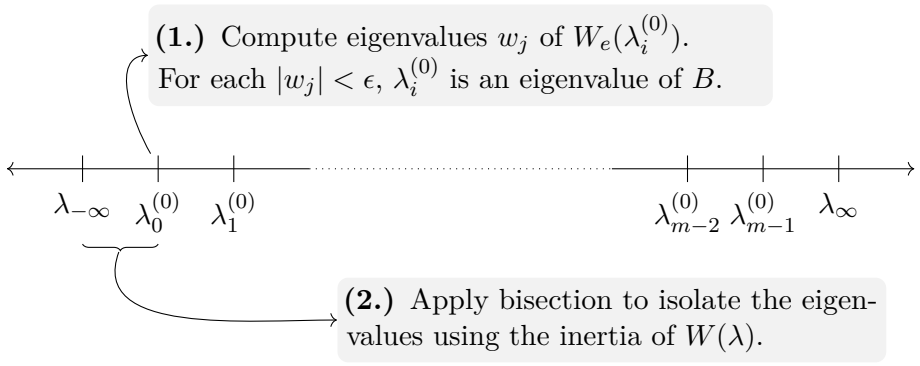


Figure 5.2.: The process of finding the eigenvalues of B.

5.1.2. Isolating the Eigenvalues of B

The sub-routine `isolateEigenvalues` uses bisection to find M (as calculated using eq. (5.16)) intervals each containing exactly one eigenvalue of B. The current implementation uses a recursive approach, see algorithm 9. As long as $M > 1$ the algorithm bisects the current interval and determines M for both halves, see line 7. Each iteration requires the computation of one *Weinstein matrix*, that is $W((a+b)/2)$, and the inertia thereof. Once an interval with $M = 1$ is found `findEigenpairInertia` is called.

It may happen that the interval (a, b) cannot by divided further, since its already minimal to working precision, that is $|b-a| < \epsilon|a|$. In this case an eigenvalue of multiplicity $\mu > 1$ is found. For matrices of type `R2` this does not occur and the corresponding code is hence not includes in algorithm 9. For a further discussion of this issue see section 6.6.2.

Algorithm 9 isolateEigenvalues

```
1:  procedure ISOLATEEIGENVALUES(a, b, M)
2:      if M = 1 then
3:          call findEigenpairInertia
4:      else if M > 1 then
5:          m ← (a + b)/2
6:          calculate interia of W(m)
7:          calculate M for both intervals (a, m) and (m, b) yielding, M_a and M_b
8:          call isolateEigenvalues(a, m, M_a)
9:          call isolateEigenvalues(m, b, M_b)
10:     end if
11: end procedure
```

5.1.3. Locating an Isolated Eigenvalue

Once the eigenvalues are isolated more efficient algorithms can be applied, see section 5.2. However, for this baseline version *bisection* is also used to locate the eigenvalues. The routine `findEigenpairInertia`, see algorithm 10, locates an eigenvalue in a given interval (a, b) and computes the corresponding eigenvector. This bisection based method has two stopping criterion. First and foremost it stops as soon as it has located an eigenvalue, that is if the *Weinstein matrix* $W(m)$ is singular (see line 5). Eigenvectors calculated hereafter seem to be accurate[6] to double-precision. The second stopping criterion (line 2) prevents an endless loop in case the interval gets minimal, that is the next floating point number to working precision of a is b. Note that this stopping criterion is not sufficient to guarantee accurate eigenvectors, since $W(m)$ will not be singular. Indeed this happens quite often and the computed eigenvectors may be completely inaccurate. In section 6.2 we will analyses the correlation between the singularity of the *Weinstein matrix* $W(\lambda)$ and the residuals $||Bq_i - \lambda_i q_i||/||B||_2$, where (λ_i, q_i) is an eigenpair of B.

Note that both stopping criterion ensure accurate eigenvalues. The computation of the corresponding eigenvector is discussed next.

Algorithm 10 findEigenpairInertia

```
1:  procedure FINDEIGENPAIRINERTIA(a, b)
2:      while b − a > |a|ε_double do
3:          m ← (a + b)/2
4:          calculate interia of W(m)
5:          if ω̂_i < ε_double then
6:              break
7:          end if
8:          if M_a > 0 then
9:              b ← m
10:         else
11:             a ← m
12:         end if
13:     end while
14:     Compute the Weinstein matrix W(m) and its eigendecomposition   ▷ Using
    L/dsyevd
15:     Compute corresponding eigenvector, see eq. (5.20)
16: end procedure
```

[6] In the sense that is they produce a residual $||Bq_i - \lambda_i q_i||_2/||B||_2 < \epsilon \cdot n \approx 2.22 \cdot 10^{-16} n$.

5.1.4. Computing Eigenvectors for Known Eigenvalues

This paragraph is concerned with the computation of eigenvectors for known $\lambda \in \sigma(B)$. They are computed at two points in the algorithm: algorithm 8 line 11 for the case $\lambda \in \sigma(B_0)$ and in algorithm 10 line 15 otherwise. In both cases a call to `weinsteinMatrix` precedes the computation of the eigenvector, as the eigenvectors of $W(\lambda)$ – or rather $W_e(\lambda)$ if $\lambda \in \sigma(B_0)$ – are required. After the call to `weinsteinMatrix` the matrices U_1^{copy} and U_2^{copy} contain $(\Lambda_1 - \lambda)^{-1}U_1$ and $(\Lambda_2 - \lambda)^{-1}U_2$ respectively.[7]

In case $\lambda \notin \sigma(B_0)$ the computation of the eigenvalues is done via

$$y = \begin{pmatrix} y_1 \\ y_0 \\ y_2 \end{pmatrix} = \begin{pmatrix} -(B_1 - \lambda I)^{-1}V_1 x_2 \\ x_2 \\ -(B_2 - \lambda I)^{-1}V_2 x_2 \end{pmatrix} = \begin{pmatrix} -Q_1(\Lambda_1 - \lambda I)^{-1}U_1 x_2 \\ x_2 \\ -Q_2(\Lambda_2 - \lambda I)^{-1}U_2 x_2 \end{pmatrix} \tag{5.20}$$

$x_2 \in \mathcal{N}(W(\lambda))$, where $\mathcal{N}$ denotes the nullspace (compare with eq. (3.26)). To compute y first $y_0 = x_2$ is set. Afterwards y_1 and y_2 are computed via four `dgemv` calls. First

$$y_1 = U_1^{copy} \cdot x_2, \qquad y_2 = U_2^{copy} \cdot x_2 \tag{5.21}$$

followed by the transformation of the eigenvalues

$$y_1 = Q_1 y_1, \qquad y_2 = Q_2 y_2. \tag{5.22}$$

Note the here the eigenvalues of B are calculated, whereas section 3.2.2 discusses the permutated matrix $\bar{B}$.

In case $\lambda \in \sigma(B_0)$ one gets (see eq. (3.28))

$$y = \begin{pmatrix} y_1 \\ y_0 \\ y_2 \end{pmatrix} = \begin{pmatrix} -(B_1 - \lambda)^{-1}V_1 x_2 + Q_1 x_3 \\ x_2 \\ -(B_2 - \lambda)^{-1}V_2 x_2 + Q_2 x_3 \end{pmatrix} = \begin{pmatrix} -Q_1(\Lambda_1 - \lambda I)^{-1}U_1 x_2 + Q x_3 \\ x_2 \\ -Q_2(\Lambda_2 - \lambda I)^{-1}U_2 x_2 + Q x_3 \end{pmatrix} \tag{5.23}$$

thus the procedure is the same except for the addition of Qx_3. Here $Q \in \mathbb{R}^{n \times \mu}$ are the eigenvectors of B_0 that span the eigenspace corresponding to λ, with μ denoting the multiplicity of λ in B_0. Therefore the addition of Qx_3 takes μ `daxpy` calls.

To finish the computation y needs to be normalised which takes one `dnrm2` and one `dscal` call.

[7] The matrices U_1^{copy} and U_2^{copy} are needed in the computation of the *Weinstein matrix*, see section 5.1.5.

Algorithm 11 weinsteinMatrix

```
1: procedure WEINSTEINMATRIX(λ, j, ext)
2:     W = Δ − λ
3:     for i ← 1, 2 do
4:         copy U_i to U_i^copy
5:         for h ← 1, 2, ... k_i do
6:             scale row h of U_i^copy by 1/(Λ_i − λ)
7:         end for
8:         W = W − U_i^⊤ U_i^copy
9:     end for
10:    if ext then
11:        W(r + 1, r + 1) = 0
12:        W(r + 1, 1 : r) = W(1 : r, r + 1)^⊤ = U_j(j, 1 : r)
13:    end if
14: end procedure
```

5.1.5. The Computation of Weinstein Matrices

This subsection covers the computation of a *Weinstein matrix* which concludes the discussion of the baseline implementation. To locate an eigenvalue several *Weinstein matrices* need to be computed. This add up to *cn Weinstein matrices* for one rank-r update, where c is the average number of iterations needed to locate an eigenvalue. An efficient implementation of the subroutine `weinsteinMatrix` is thus essential. Algorithm 11 summarises the procedure. It takes three parameters: a double-precision value λ, a boolean *ext* and an integer j. In case $ext = true$ the *extended Weinstein matrix*

$$W_e(\lambda) = \begin{pmatrix} \Delta - \lambda - U^t((\Lambda_1 - \lambda I_{b1}) \oplus (\Lambda_2 - \lambda I_{b2}))^+ U & V^t Q \\ Q^t V & 0_\mu \end{pmatrix}, \tag{5.24}$$

is computed, otherwise

$$W(\lambda) = \Delta - \lambda - U^\top((\Lambda_1 - \lambda I_{b1}) \oplus (\Lambda_2 - \lambda I_{b2}))^{-1} U. \tag{5.25}$$

If *ext* is *true* λ is an eigenvalue of B_0 and the parameter j specifies to which subproblem it belongs ($\lambda \in \sigma(B_j)$). The main effort is the computation of

$$U^\top((\Lambda_1 - \lambda I_{b1}) \oplus (\Lambda_2 - \lambda I_{b2}))^{-1} U, \tag{5.26}$$

which is done the following way. U_i ($i = 1, 2$) is copied into an additional array U_i^{copy}, which is scaled by $(\Lambda_i - \lambda I_{b_i}))^{-1}$ in lines 5 to 7. Afterwards two `dgemm` call computes

$U_i^\top U_i^{copy}$ in line 8.

Note that since $(\Lambda_i - \lambda)^{-1}$ has negative values one cannot scale U_i^{copy} by $(\Lambda_i - \lambda)^{-1/2}$ without using complex numbers. Thus $U_i^\top U_i^{copy}$ cannot be computed via the symmetric rank-r operation `dsyrk`[8], but a `dgemm` call needs to be used.

5.1.6. Accuracy of the Baseline Implementation

The residual $\mathfrak{R}$ (eq. (4.2)) and orthogonality error $\mathfrak{O}$ (eq. (4.3)) for the matrices of type `R2` are plotted in fig. 5.3. Clearly the results are not sufficiently accurate. `LAPACK`'s eigensolvers achieve $\mathfrak{R} < \epsilon nc$ and $\mathfrak{O} < \epsilon nc$, where c is a small constant [19].

In contrast all eigenvalues match those computed via `L/dsyevd` up to at least $\approx 10^{-14}$. The resulting inaccuracies thus seem to stem from the computation of the eigenvectors.

5.2. Version 1 - Zeroin

As a first improvement `dsbkdcV1` (Version 1) incorporates *Brent's zeroin* algorithm [14] to speed up *bisection* when searching for an already isolated eigenvalue. Thus if an interval (a, b) with $M = 1$ is found in algorithm 9 line 3 the subroutine `findEigenpair` is called instead of `findEigenpairInertia`. This superlinear zerofinder will be introduced in the next subsection.

In this version a few additional functions are add, see fig. 5.4. Most importantly the new subroutine `findEigenpair` implements the *zeroin* method to locate an isolated eigenvalue. If it does not succeed or if *zeroin* cannot be applied the older *bisection* based methods (`findEigenpairInertia`) is used. The code for the computation of eigenvectors is moved to a separate function called `computeEigenvector`. Finally `detW` computes the determinant of a *Weinstein matrix*, which is needed by *zeroin*. In section 5.2.2 the computation of $\det(W(\cdot))$ will be discussed.

5.2.1. Brent's Root-Finding Algorithm

Brent's algorithm is a modification from *Dekker's zeroin method* developed in 1969.[9] It combines bisection, linear (the secant method) and inverse quadratic interpolation. Additionally a minimal step guarantees a certain improvement. To start an interval (a, b) is needed on which the function $f(\cdot)$ changes sign. As long as f is continuous the intermediate value theorem can be applied and guarantees a zero in the interval. The

[8] Calculates $\alpha AA^\top + \beta C$, with a symmetric matrix $C \in \mathbb{R}^{n\times n}$ and an update matrix $A \in \mathbb{R}^{n\times k}$.

[9] See [16] for a comparison of Dekker's initial algorithm and Brent's improved one.

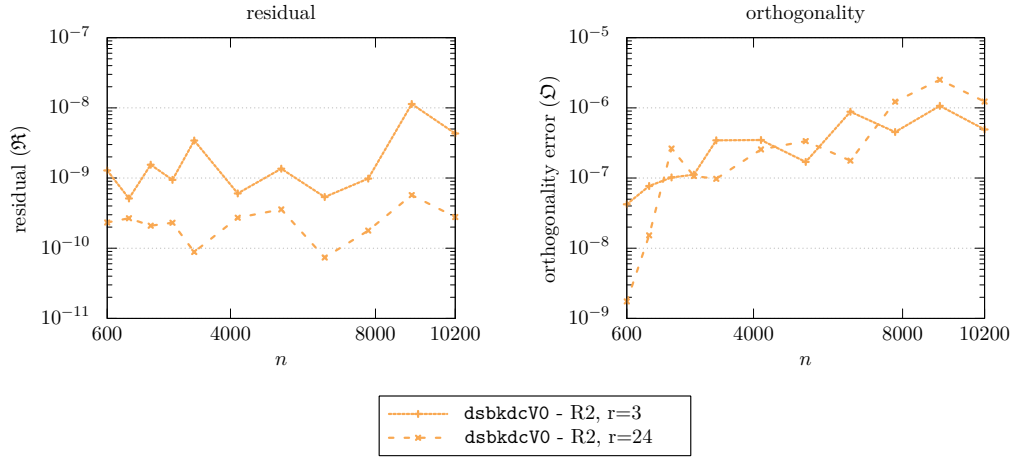


Figure 5.3.: Accuracy of `dsbkdcV0`. Both the residual $\mathfrak{R}$ (left) and the orthogonality error $\mathfrak{D}$ (right) for test matrices of type `R2` are shown. The bandwidth is $r = 3$ and $r = 24$.

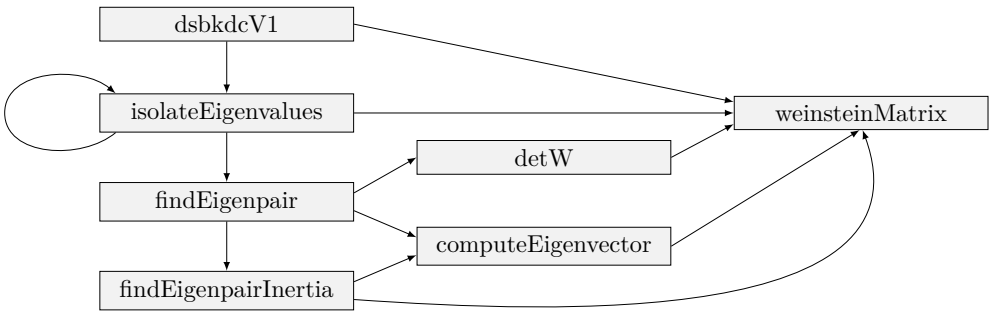


Figure 5.4.: Call graph for `dsbkdcV1`.

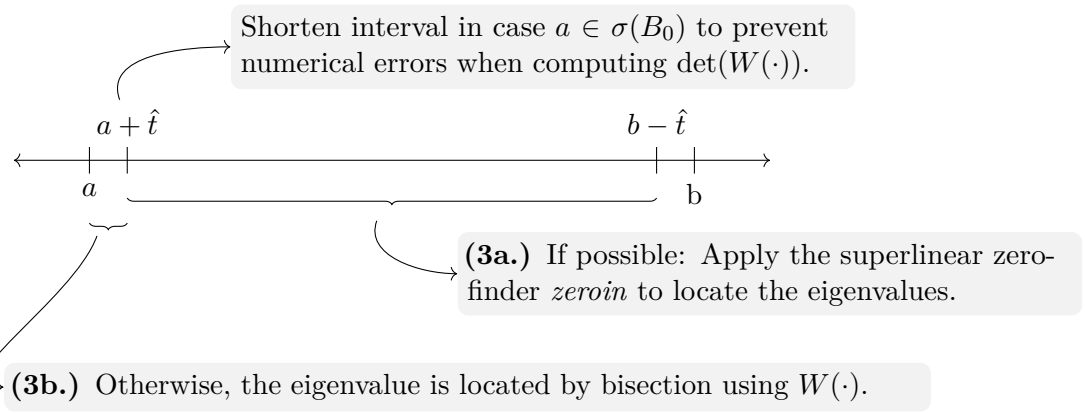


Figure 5.5.: Using *zeroin* to locate an isolated eigenvalue.

goal is to minimise the interval so that $b - a < \varepsilon$ while conserving $f(a) \cdot f(b) < 0$, hence to located a zero up to $\pm\varepsilon$. In the context of `dsbkdcV1` $f(\cdot) = det(W(\cdot))$ and the zero corresponds to an eigenvalue of B.

Consider an isolated eigenvalue in the interval (a, b). Since $W(\lambda)$ has poles for $\lambda \in \sigma(B_0)$ it is necessary to tighten the interval by a small margin if a or b are also eigenvalues of B_0, see fig. 5.5. Currently a is set to

$$a = \begin{cases} a + 25|a| \cdot \epsilon_{double} & \text{if } a \in \sigma(B_0), \\ a & \text{else,} \end{cases} \tag{5.27}$$

and analogously b to

$$b = \begin{cases} b - 25|b| \cdot \epsilon_{double} & \text{if } b \in \sigma(B_0), \\ b & \text{else .} \end{cases} \tag{5.28}$$

As the interval is tightened two prerequisites of *zeroin* need to checked. Firstly that $a < b$, hence that the interval did not vanish and secondly, that $\det(W(a)) \cdot \det(W(b)) < 0$, that is that the interval still contains the zero of $\det(W(\cdot))$. Compare with fig. 5.5.

Version 0 of `dsbkdc` uses bisection to find all eigenvalues. Let $1/2 \leq |b/a| \leq 2$, e.g. let a and b have similar exponents. Then for a stopping criterion $b - a < \epsilon|a|$ bisection takes approximately 52 iterations, since the interval is cut in half in every iteration and the size of mantissa is 52.[10] A faster method is linear interpolation, where the next point is

[10]Due to the interlacing property of B_0 and since $||B||_2 = 1$ less than 52 iterations are needed in `findEigenpairInertia`.

calculated as where the straight line through (a, f_a) and (b, f_b) intersects the x-axis. An iteration step with linear interpolation is calculated via

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \cdot f(x_n). \tag{5.29}$$

Unfortunately this method can escape the initial interval after a few iterations, since $x_{n+1} \in (x_{n-1}, x_n)$ is not guaranteed. In that case it would not find the correct zero, or any at all, since $\det(W)$ is a function with n zeros and $b_1 + b_2$ poles.

To get faster convergence as for linear interpolation it is possible to employ quadratic interpolation, which is known as *Muller's method* [24, p. 192]. Therefore three distinct points are needed, e.g. the current interval plus a previous point. These can be obtained via bisection, yielding x_n, x_{n-1} and x_{n-2}. Unfortunately calculating the intersection with the x-axis may result in complex numbers since a square-root is needed. This is solve in *Brent's method* by using quadratic interpolation on the inverse of f. Making use of the *Lagrange interpolation formula* to interpolate f_a, f_b and f_c one gets

$$f^{-1}(y) = \frac{(y - f_b)(y - f_c)}{(a - b)(a - c)}a + \frac{(y - f_a)(y - f_c)}{(b - a)(b - c)}b + \frac{(y - f_a)(y - f_b)}{(c - a)(c - b)}c. \tag{5.30}$$

For $y = 0$ this yields[11]

$$x = \frac{f_b f_c}{(a - b)(a - c)}a + \frac{f_a f_c}{(b - a)(b - c)}b + \frac{f_a f_b}{(c - a)(c - b)}c. \tag{5.31}$$

The algorithm Dekker defined uses linear interpolation whenever it is save and falls back to bisection when needed. Brent added a few improvements, as for example the use of inverse quadratic interpolation. In algorithm 12 an overview of the algorithm *zeroin* is given. The following variables keep track of the progress:

- (a, b) contains the zero
- c is the previous value of b
- d holds the current step size
- e holds the last step size

Brent stores the last step size in e and uses bisection whenever a minimal step was taken the last time, thus ensuring that at most twice as many steps are performed as

[11] This was adopted from [37], however, similar is show in [24].

when using only bisection, see line 14. Otherwise interpolation is used. If in the current iteration the algorithms has three distinct points a, b and c *zeroin* uses inverse quadratic interpolation. If $a = c$ and thus only 2 distinct points exist linear interpolation is used, see line 18. In line 30 *zeroin* checks whether the intersection point is outside the interval. In line 37 it is ensured that the step has at least size *tol*. Finally the determinant of the new point is calculated in line 42.

The algorithm has two stopping criterion: Either it found a zero of f or the interval is minimal, see line 11. As already mentioned in section 5.1.3 a minimal interval is not sufficient for computing accurate eigenvectors. After the eigenvalue is found the corresponding eigenvector is computed as described in section 5.1.4.

5.2.2. Calculating the Weinstein Determinant

It remains to be discussed how to compute the *Weinstein determinant*, that is the determinant of $W(\cdot)$. One possibility is to calculate the LU-decomposition. This gives $W(\cdot) = W = LU$ with L having all ones on the diagonal. Thus the determinant is simply the product of the diagonal elements of U:

$$\det(W) = \det(LU) = \det(L)\det(U) = \det(U) = \prod_i u_{ii}. \tag{5.32}$$

Alternative one can compute the spectral decomposition $W = VDV^\top$ and thus compute the determinant as

$$\det(W) = \det(VDV^\top) = \det(V)\det(D)\det(V^\top) = \prod_i d_{ii}. \tag{5.33}$$

`L/dgetrf` which computes the LU-decomposition has a runtime complexity $\mathcal{O}(n^2)$, whereas `L/dsyevd` needs $\mathcal{O}(n^3)$ flops. However, as only the eigenvalues are required the computation is quite fast, see chapter 2. Furthermore the *Weinstein matrices* are small $(r \times r)$ and thus the computation of the eigendecomposition of $W(\cdot)$ had little influence on the overall runtime of `dsbkdc`.

As can be seen in section 6.1 the accuracy of the determinant, or rather the accuracy of the sign of the determinant, is essential. The actual difference in runtime between the LU- and spectral decomposition is small in comparison to the benefit of using *zeroin*. Hence the more accurate method was chosen, which in our experiments seems to be `L/dsyevd`.

The absolute values of the determinant is in fact not as important, see section 5.5.

Algorithm 12 Brent's zeroin methods as used in the subroutine `findEigenpair`

$f_a \leftarrow \det(W(a)), \quad f_b \leftarrow \det(W(b))$
while true **do**
  if $\operatorname{sgn}(f_a) = \operatorname{sgn}(f_b)$ **then** ▷ *guarantee sign change in* (a, b)
    $a \leftarrow c, \quad f_a \leftarrow f_c, \quad d \leftarrow b - c, \quad e \leftarrow d$
  end if
  if $|f_a| < |f_b|$ **then** ▷ *make sure* $f(b)$ *is best approximation so far*
    $c \leftarrow b, \quad b \leftarrow a, \quad a \leftarrow c, \quad f_c \leftarrow f_b, \quad f_b \leftarrow f_a, \quad f_a \leftarrow f_c$
  end if
  $m = (a - b)/2$
  $tol = 2.0\epsilon \cdot \max(|b|, 1)$
  if $m < tol$ OR $f_b = 0$ **then** ▷ *test for convergence*
    `break`
  end if
  if $e < tol$ OR $f_c <= f_b$ **then** ▷ *bisection*
    $d = m, \quad e = m$
  else ▷ *interpolation*
    $s = f_b/f_c$
    if $a = c$ **then** ▷ *linear interpolation*
      $p = 2ms, \quad q = 1 - s$
    else ▷ *inverse quadratic interpolation*
      $q = f_c/f_a, \quad k = f_b/f_a$
      $p = s[2m \cdot q(q - k) - (b - c)(k - 1)]$
      $q = (q - 1)(k - 1)(s - 1)$
    end if
    if $p > 0$ **then**
      $q = -q$
    else
      $p = -p$
    end if
    if $2p < 3mq - |tol \cdot q|$ AND $p < |0.5eq|$ **then**
      $e \leftarrow d, \quad d \leftarrow p/q$
    else
      $d \leftarrow m, \quad e \leftarrow m$
    end if
  end if
  $c \leftarrow b, \quad f_c \leftarrow f_b$
  if $|d| > tol$ **then** ▷ *use minimal step if step size is too small*
    $b = b + d$
  else
    $b = b - \operatorname{sgn}(b - c) \cdot tol$
  end if
  $f_b = \det(W(b))$ ▷ *calculate new value for* $f(b)$
end while

5.2.3. Runtime of the Baseline Implementation and Version 1

In fig. 5.6 the runtime of the baseline implementation `dsbkdcV0` is compared with the improved version `dsbkdcV1`. On the left the total runtime for each routine is shown. The test data are matrices of type `R2` with either semi-bandwidth $r = 3$ or $r = 24$. One can see that the speedup is much more severe for the larger bandwidth. This is due to the fact that as the bandwidth gets smaller the *extended Weinstein matrices* in algorithm 8 line 11 are more likely to be singular. In these cases no root-finder is applied and hence *zeroin* cannot yield a speedup.

On the right in fig. 5.6 the runtime is scaled by n^3. Clearly the complexity is little below $\mathcal{O}(n^3)$ for $r = 3$. For matrices with larger bandwidth the runtime is higher, but the complexity seems lower. Generally the algorithm has a complexity of $\mathcal{O}(n^3)$ which comes from the back-transformation of the eigenvector. However it seems that the factors with lower complexity have a higher constant which dominate the runtime, hence it appears that the algorithm has a complexity $< \mathcal{O}(n^3)$. The version (`dsbkdcV2`) will use level 3 `BLAS` operations for the back-transformation which further reduces the constant for the n^3 factor, yielding a better runtime even for small bandwidths.

5.3. Version 2 - Blocked Back-Transformation

In this version (`dsbkdcV2`) the back-transformation of the eigenvalues during the rank-r update is block. Instead of calculating each eigenvector completely by

$$y = \begin{pmatrix} -Q_1(\Lambda_1 - \lambda I)^{-1} U_1 x_2 \\ x_2 \\ -Q_2(\Lambda_2 - \lambda I)^{-1} U_2 x_2 \end{pmatrix} \quad \text{or} \quad y = \begin{pmatrix} -Q_1(\Lambda_1 - \lambda I)^{+} U_1 x_2 + Q x_3 \\ x_2 \\ -Q_2(\Lambda_2 - \lambda I)^{+} U_2 x_2 + Q x_3 \end{pmatrix}, \tag{5.34}$$

as described in section 5.1.4 only the first matrix-vector product is computed, which yields the vector

$$\hat{y} = \begin{pmatrix} (\Lambda_1 - \lambda I)^{-1} U_1 x_2 \\ x_2 \\ (\Lambda_2 - \lambda I)^{-1} U_2 x_2 \end{pmatrix} \quad \text{or} \quad \hat{y} = \begin{pmatrix} (\Lambda_1 - \lambda I)^{+} U_1 x_2 - E x_3 \\ x_2 \\ (\Lambda_2 - \lambda I)^{+} U_2 x_2 - E x_3 \end{pmatrix}. \tag{5.35}$$

The right equations in eq. (5.34) and eq. (5.35) need to be used in case $\lambda \in \sigma(B_0)$, the left ones otherwise. As defined in theorem 3.2.1 $Q \in \mathbb{R}^{n \times \mu}$ are the eigenvalues of B_0 spanning $\mathcal{N}(B_0 - \lambda)$, where μ is the multiplicity of λ. Thus $E = (Q_1^\top \oplus Q_2 \top) Q$ in eq. (5.35). Let $q_i^{(0)}$ denote the columns of $Q_0 = (Q_1 \oplus Q_2)$, then the update $\hat{y} = \hat{y} - E x_3$ can be written

as

$$\hat{y} = \hat{y} - x_3 \sum_{i=0}^{\mu} e_{I(i)} \tag{5.36}$$

where e_j is a unit vector of the standard basis and I is a set of indices, such that

$$\forall i \in I: \quad q_i^{(0)} \in \mathcal{N}(B_0 - \lambda), \tag{5.37}$$

holds. This update takes μ flops.

To keep all interim result additional workspace is needed, more precisely a $n \times n$ matrix $\hat{Y}$ with columns $\hat{y}_i$. At the end of `dsbkdcV2`, that is after all eigenvalue are located and all interim eigenvectors are computed, $\hat{Y}$ needs to be back-transformed. This is done with two `dgemm` calls which together compute

$$Y = -(Q_1 \oplus 0_{r\times r} \oplus Q_2)\hat{Y} = -\begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 0_{r\times r} & 0 \\ 0 & 0 & Q_2 \end{pmatrix} \hat{Y}. \tag{5.38}$$

The improved algorithm `dsbkdcV2` uses level 3 `BLAS` operations, whereas `dsbdcV1` uses only level 2.

5.3.1. Speedup and Runtime Complexity for Version 2

Figure 5.7 shows the improved runtime. On the left one can see the speedup in comparison to the baseline implementation. On the right the runtime divided by n^3 is plotted, suggesting a runtime complexity below $\mathcal{O}(n^3)$. Once again the complexity for small bandwidth seem higher than for large bandwidths. This is mostly due to the fact that the sub-problems are solved using `L/dsyevd` which has a complexity $\mathcal{O}(n^3)$. The addition of recursion preserves the lower complexity for smaller bandwidths, see section 5.6.

5.4. Version 3 - Packed Storage

In this version (`dsbkdcP0`[12]) the switch to packed storage is carried out which improves the storage requirements. In all version so far the matrix B is stored in an array of size n^2, disregarding the sparse structure of B. For the improved storage scheme B is

[12]The `Px` stands for *packed x*.

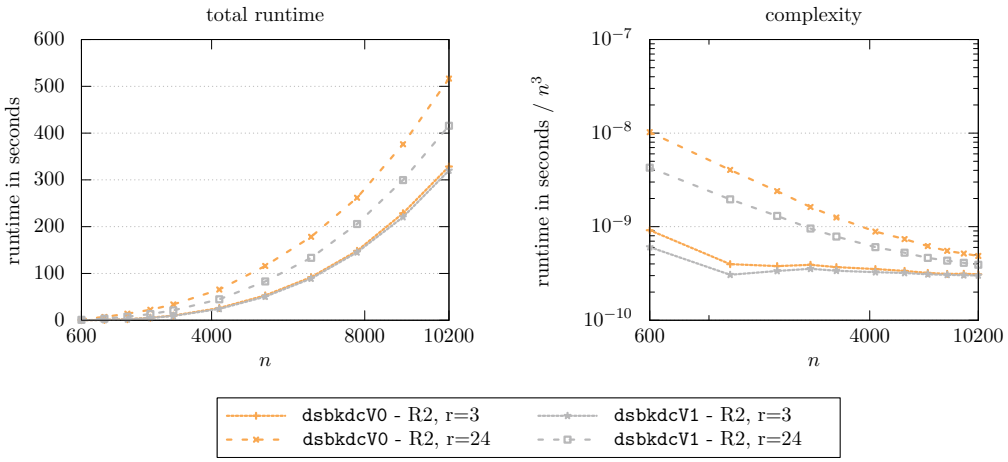


Figure 5.6.: Runtime of `dsbkdcV0` and `dsbkdcV1` for matrices of type `R2`. On the left the total runtime in seconds for increasing n. On the right the runtime scaled by $1/n^3$ and plotted on a log-log scale. Hence a horizontal line would suggest a runtime complexity of $\mathcal{O}(n^3)$.

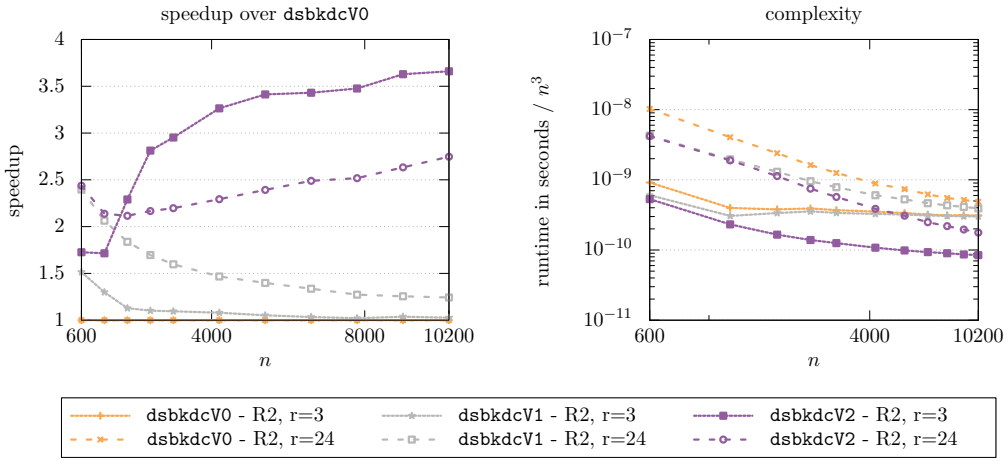


Figure 5.7.: Runtime and speedup of `dsbkdcV0`, `dsbkdcV1` and `dsbkdcV2`.

interpreted as a symmetric block tridiagonal matrix

$$B = \begin{pmatrix} D_1 & E_1^\top & & & \\ E_1 & D_2 & E_2^\top & & \\ & E_2 & D_3 & \ddots & \\ & & \ddots & \ddots & E_{p-1}^\top \\ & & & E_{p-1} & D_p \end{pmatrix} \in \mathbb{R}^{n\times n}, \qquad D_i, E_i \in \mathbb{R}^{r\times r}.$$

The array D stores the main block diagonal consisting of the symmetric $r \times r$ blocks D_i. The size of D is $r \times r \times p$ with $p = n/r$. E stores the fist block subdiagonal with $p-1$ non-symmetric blocks E_i. Even though the block elements E_i are triangular the whole blocks are save. This version only supports matrices $n \mod r = 0$, which simplifies the implementation. Each block is stored in column major order, see fig. 5.8. The storage requirements to store B are thus $\approx 2nr$ instead of n^2. Note that the workspace requirements are unchanged and increase with n^2. This is due to the blocked computation of the eigenvectors, which requires the matrix $\hat{Y}$, see section 5.3.

The runtime is not affected by this change (see fig. 5.9) as the algorithm operates mainly on the band structure of B. Hence most zeros of the dense storage are never accessed in `dsbkdcVx`.

5.5. Version 4 - Improving the Root-Finding Method

As a final performance improvement the *Weinstein determinant* is scaled near the poles to speed up the root-finding method *zeroin*. Arbenz [4] applied *zeroin* to the function

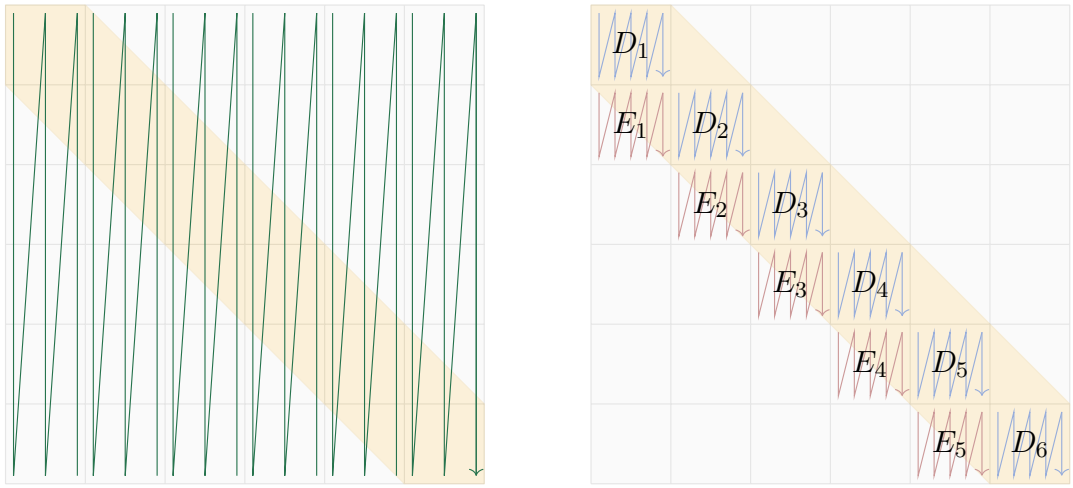
$$f(\lambda) = (\lambda_{j-1}^{(0)} - \lambda)^{\mu_{j-1}} (\lambda_j^{(0)} - \lambda)^{\mu_j} \det W(\lambda), \qquad \lambda \in (\lambda_{j-1}^{(0)}, \lambda_j^{(0)}), \tag{5.39}$$

where μ_i denotes the multiplicity of $\lambda_i^{(0)}$ as an eigenvalue of B_0. Alternatively the function

$$\hat{f}(\lambda) = \min_i \omega_i \in \sigma(W(\lambda)), \tag{5.40}$$

was tested, with little difference in performance.

Version 4 (`dsbkdcP1`) uses eq. (5.39). The runtime of `dsbkdcV2`, `dsbkdcP0` and `dsbkdcP1` is shown in fig. 5.9. As mentioned in section 5.2 a larger bandwidth decreases the probability that the *extended Weinstein matrices* are singular, hence this improvements has a larger impact on matrices with $r = 24$.



(a) full (b) block tridiagonal

Figure 5.8.: Different storage schemes. The full storage (left) is used in `dsbkdcVx`, whereas the more economical block tridiagonal storage scheme (right) is used for `dsbkdcPx`. The orange band illustrates the non-zero elements of the matrix.

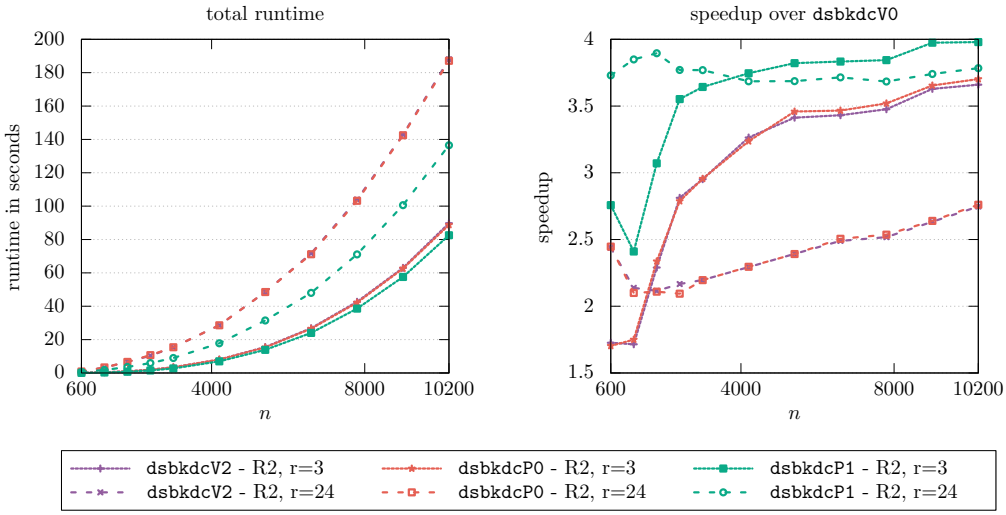


Figure 5.9.: Runtime and speedup for `dsbkdcV2`, `dsbkdcP0` and `dsbkdcP1`. Note that `dsbkdcV2` and `dsbkdcP0` have almost the same runtime since only the storage scheme changed.

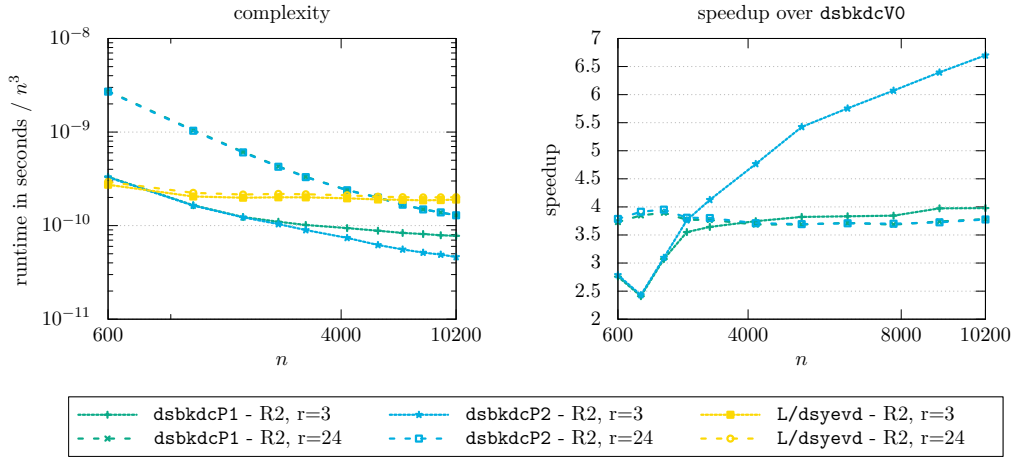


Figure 5.10.: On the left the runtime of `dsbkdcP1` and `dsbkdcP2` in comparison to `L/dsyevd`. On the runtime the speedup of `dsbkdcP1` and `dsbkdcP2` over the baseline implementation.

5.6. Version 5 - Recursion

In this version (`dsbkdcP2`) recursion is added. To see whether a recursive approach will improve the runtime the performance of `dsbkdcP1` and `L/dsyevd` are compared and a crossover point is determined. That is the size $\hat{n}$ for which `dsbkdcP1` is faster than `L/dsyevd` and should thus be used to solve the subproblem. For matrices $2\hat{n}$ a recursive approach will improve the performance.

For `dsbkdcP2` recursion is used if $n \geq 2\hat{n}$ with

$$\hat{n} = \begin{cases} 1000, & \text{if } r = 3, \\ 5000, & \text{if } r = 24. \end{cases} \tag{5.41}$$

5.6.1. Runtime

Figure 5.10 shows the runtime for `dsbkdcP2`. On the left the runtime complexity of `L/dsyedc`, `dsbkdcP1` and `dsbkdcP2` is plotted. The line corresponding to `L/dsyedc` is completely horizontal, indicating a $\mathcal{O}(n^3)$ complexity. `dsbkdcP1` displays a lower complexity, however, it flattens for increasing n for $r = 3$. This is due to the computation of the spectral decomposition of the subproblems with `L/dsyevd`. By using an recursive approach the lower complexity is preserved for larger n, see fig. 5.10 (`dsbkdcP2`).

The right plot in fig. 5.10 shows the speedup compared to `dsbkdcV0`.

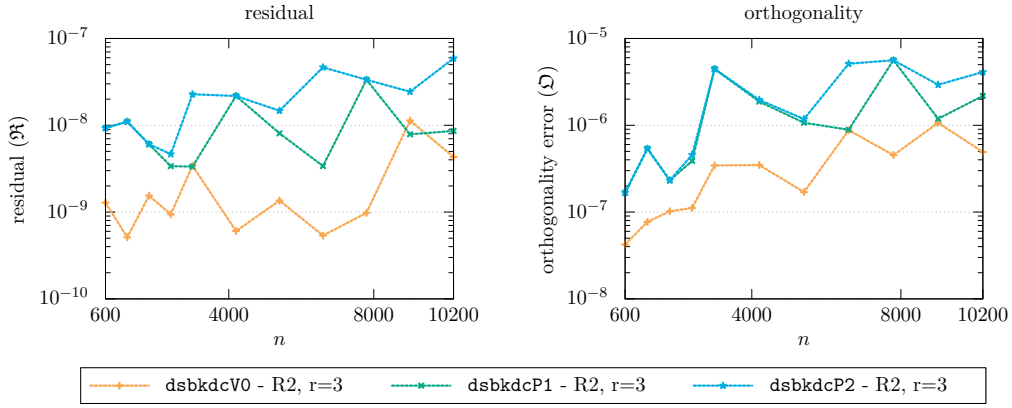


Figure 5.11.: Accuracy of `dsbkdcV0`, `dsbkdcP1` and `dsbkdcP2`.

5.6.2. Accuracy

The accuracy of `dsbkdcP2` is plotted in fig. 5.11. The difference in the accuracy between `dsbkdcP1` and `dsbkdcV0` comes from a slightly different stopping criterion when using *zeroin* instead of *bisection*. The eigenvalues of these two methods differ by $\pm 2\epsilon|\lambda|$, which explains the difference in accuracy as the algorithm is highly sensitive to perturbations in the eigenvalues. The accuracy of the recursive approach is only slightly worse.

6. Towards Improving the Accuracy of DSBKDC

The implementations from chapter 5 (e.g. `dsbkdcP1`) produce both residuals and orthogonality errors that are not sufficiently accurate in comparison to standard eigensolvers. This chapter is concerned with understanding the cause of these numerical inaccuracies that occur during the computation of the eigenvectors. The first sections present two small adaptations which lead to a more stable implementation and allow for the integration of new test data which failed for previous versions. Using these new test matrices section 6.2 analyses different metrics that correlation well to the "individual" residuals $\mathfrak{R}_i(B) = ||Bq_i - \lambda_i q_i||_2,\ i = 1, 2, \ldots, n$. A near perfect correlation to the smallest eigenvalue

$$\hat{\omega}_i \coloneqq \min_j |\omega_j| \in \sigma(W(\lambda_i)) \tag{6.1}$$

of the corresponding *Weinstein matrix* is identified. This proves that the calculation of the eigenvectors is extremely sensitive to perturbations in the eigenvalue. A similar problem is described in section 2.4.2 for the case of a rank-one modification for *Cuppen's divide and conquer* method. The rest of the chapter explores different approaches to improving the results. Section 6.3 discusses the modifications implemented by Arbenz [4] that circumvent the inaccuracies during the computation of the eigenvectors. Section 6.4 takes a different approach to improving the accuracy of the algorithm by using extended precision to calculate more precise eigenvalues. In section 6.5 multiplicities $\mu > 1$ in $\sigma(B_0)$ are discussed. Section 6.6 concludes the chapter by presenting some adaptions needed for test data with tightly clustered eigenvalue.

6.1. Version 6 - Adaptation of two Thresholds

In this section two small modifications are presented that enable the integration of further test data. In section 6.1.1 the amount by which an interval is tightened before applying *Brent's zeroin* method is increased. Section 6.1.2 slightly modifies the threshold which determines the singularity of a *Weinstein matrix*. These modifications are done by trail

and error and hence depend on the test matrices. This could be problematic as they may not work for new data. This is mainly problematic for the first subsection as the threshold discussed in section 6.1.2 can e.g. be found in [14].

6.1.1. Tightening the Interval Near the Poles of a Weinstein Matrix

When locating an eigenvalue in $(\lambda_i^{(0)}, \lambda_{i+1}^{(0)})$ (see algorithm 8 line 14) *bisection* is used until an interval (a, b) containing exactly one eigenvalue is found. Once this is the case `findEigenpair` uses *Brent's zeroin* method to speed up the root-finding process. However, as described in section 5.2 (see also fig. 5.5) if a or b are eigenvalues of B_0 the interval has first to be tightened, since the *Weinstein determinant* has poles at $\lambda \in \sigma(B_0)$. In `dsbkdcP1` the interval (a, b) is tightened by

$$\bar{x} = \begin{cases} x \pm 25|x| \cdot \epsilon_{double} & \text{if } x \in \sigma(B_0), \\ x & \text{else,} \end{cases}, \qquad x \in \{a, b\}, \tag{6.2}$$

yielding the new interval $(\bar{a}, \bar{b})$. Unfortunately in some cases `findEigenpair` fails to locate the zero in the interval $(\bar{a}, \bar{b})$ even though the prerequisites are met. A further analysis of the problem shows that *zeroin* converges to one of the interval borders. It turns out that in these cases the sign of the *Weinstein determinant* of $\bar{a}$, $\bar{b}$ or both are wrong due to numerical errors. These numerical inaccuracies stem from the cancellation in $(\Lambda - \lambda I)^{-1},\ \lambda \in \{\bar{a}, \bar{b}\}$ when calculating the corresponding *Weinstein matrix*, as

$$\exists\ \Lambda_i \in \sigma(B_0) : \lambda = \Lambda_i \pm 25|\Lambda_i|\epsilon_{double}, \tag{6.3}$$

where Λ_i are the diagonal elements of Λ. By adjusting the added step from

$$25|x| \cdot \epsilon_{double} \qquad \text{to} \qquad 300|x| \cdot \epsilon_{double} \tag{6.4}$$

these inaccuracies no longer occur. To make sure that no numerical errors occur a further tightening of the interval may be practical, however, an overly large step t, with $\bar{x} = x \pm t$, will hinder the use of *Brent's zeroin* method. For this version (`dsbkdcA1`) $\bar{x} = x \pm 300|x| \cdot \epsilon_{double}$ in case $x \in \sigma(B_0)$ is used.

6.1.2. Threshold for the Singularity of a Weinstein matrix

So far the *Weinstein matrix* corresponding to λ_i is considered singular if

$$\hat{\omega}_i < \epsilon. \tag{6.5}$$

This is hence a stopping criterion for both *bisection* (in `findEigenpairInertia`) and *Brent's zeroin* method (in `findEigenpair`). The same threshold is used to determine whether an *extended Weinstein matrix* is singular. As it turns out the threshold needs to be adapted for matrices B with a spectral radius $||B||_2 > 1$, e.g. the matrices of type `R1`. As a new threshold

$$\hat{\omega}_i < \max(|\lambda_i|, 1.0)\epsilon, \tag{6.6}$$

is chosen, which is similar to the stopping criterion of *Brent's zeroin* method.

The `dsbkdc` algorithm fails to computed all eigenpairs for the matrix `W2_9000_3_8795` when using the old threshold. More precisely `dsbkdcP1` produces a residual of 0.037 if eq. (6.5) is used to determine whether the *extended Weinstein matrices* are singular. To understand the problem let a be an eigenvalue of B_0 and let the corresponding *extended Weinstein matrix* be singular with regard to eq. (6.6) but not eq. (6.5). Then `isolateEigenvalues` will find an interval (a, b) (or (b, a)) containing exactly one eigenvalue, since the eigenvalue is clearly close to a. The problem occurs if a is already be the best approximation of the eigenvalue. Consequently *bisection* will be used to determine the eigenvalue in the interval (a, b) and it will converges to $a \in \sigma(B_0)$. As a result the eigenvector is calculated with eq. (3.26) instead of eq. (3.28) which leads to a large residual.

Our experiments suggest that the *(extended) Weinstein matrix* may be considered singular if $\hat{\omega}_i < \max(|\lambda_i|, 1.0)\epsilon$. Even a larger threshold, e.g. $50\max(|\lambda_i|, 1.0)\epsilon$ works for our test data, see fig. 6.4. However, to avoid overfitting eq. (6.6), which is more natural, is used for future versions.

6.1.3. Results for Version 6

With these two adaption the new version (`dsbkdcA1`) achieves a residual of $\mathcal{R} < \epsilon_{single} \cdot n$ for all matrix types defined in section 4.2 that are generated with $\varepsilon = \sqrt{\epsilon_{double}}$, excluding `S1` and `S2`. The orthogonality error depends both on the residual and on the matrix type and may be larger than $\epsilon_{single} \cdot n$. Figure 6.1 shows the residual (left) and the orthogonality error (right) for some test data.

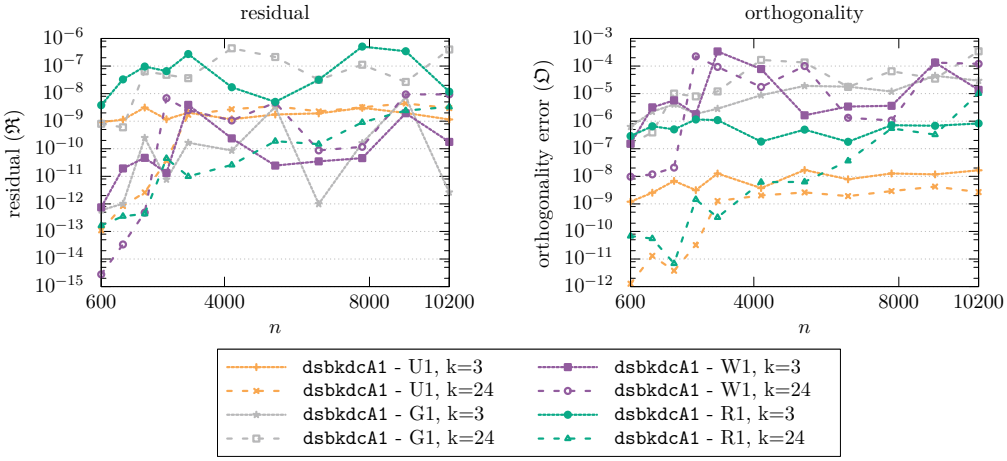


Figure 6.1.: Accuracy of `dsbkdcA1`.

6.2. Searching for Correlations

This section gives insight to the origin of the numerical instabilities that occur during the computation of the eigenvalue. Each section presents a metric and analyses the correlation with the residual $\mathfrak{R}(B)$ or the "individual" residuals

$$\mathfrak{R}_i(B) = \frac{||Bq_i - \lambda_i q_i||_2}{||B||_2}, \tag{6.7}$$

where $\mathfrak{R}_i$ is the residual corresponding to the i-th eigenpair (λ_i, q_i).

6.2.1. Correlation Between Residual and Orthogonality Error

First the correlation between the residual ($\mathfrak{R}$) and the orthogonality error ($\mathfrak{O}$), as defined in section 4.3, is analysed. Theorem 2.1.2 shows that a small residual is sufficient to guarantee orthogonal eigenvectors if the eigenvalues are well separated and thus the gaps γ_i between neighbouring eigenvalue are large. Figure 6.2 depicts the correlation between the two metrics. Each dot represents a test matrix. The matrices have bandwidth $r = 3$ or $r = 24$ and dimension $n = 600$ to $n = 10200$. For some matrix types the correlation is clearly visible as e.g. for `R1`. However, the orthogonality error depends also on the distribution of the eigenvalue. For test data with tightly clustered eigenvalues (e.g. `W1`) a double precision residual produced only a single precision orthogonal error. For extreme test cases (e.g. `S1`) the orthogonality error may approach 1 independent of the residual.

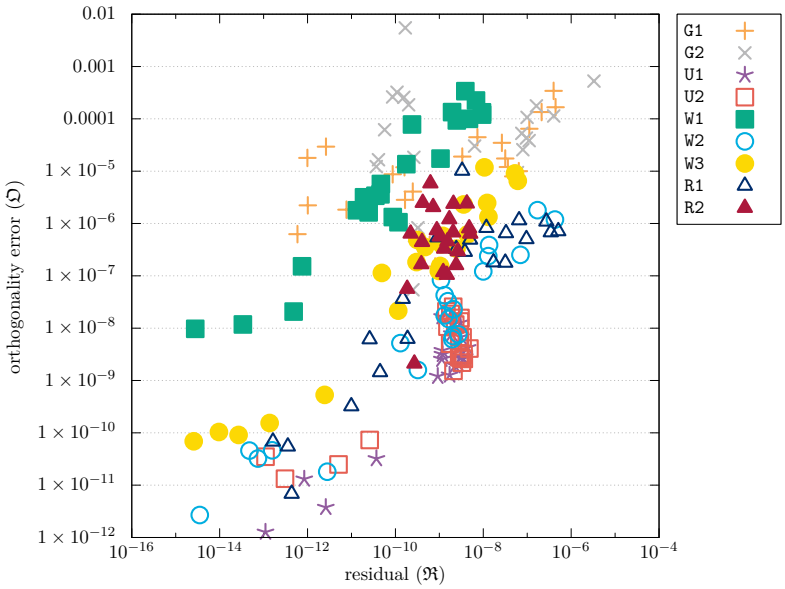


Figure 6.2.: Correlation between the residual ($\mathfrak{R}$) and the orthogonality error ($\mathfrak{O}$), as calculated by `dsbkdcA1`. Each dot represents a test matrix with semi-bandwidth $r = 3$ or $r = 24$ and a dimension between $n = 600$ and $n = 10200$.

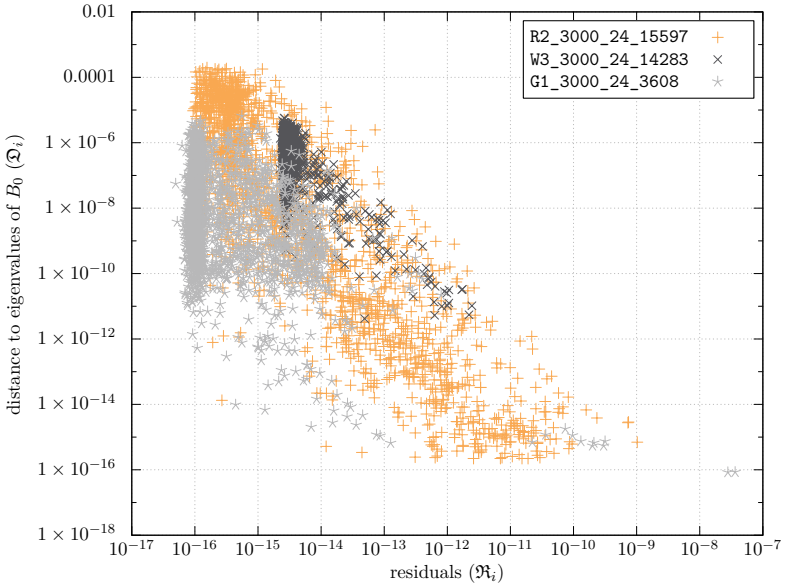


Figure 6.3.: Correlation between the individual residual $\mathfrak{R}_i$ and the distance $\mathfrak{D}_i$ between the corresponding eigenvalue λ_i to the closest eigenvalue in $\sigma(B_0)$. Each point corresponds to one eigenpair (λ_i, q_i).

6.2.2. The Impact of the Distance to Eigenvalues of the Subproblem

In this subsection the correlation between the residual $\mathfrak{R}_i$ (see eq. (6.7)) and

$$\mathfrak{D}_i(B) = \mathfrak{D}_i = \min_j \frac{|\lambda_i - \lambda_j^{(0)}|}{\max(1.0, |\lambda_i|)}, \tag{6.8}$$

that is the distance between an eigenvalue $\lambda_i \in \sigma(B)$ and the closest eigenvalue $\lambda_j^{(0)} \in \sigma(B_0)$ is analysed. In fig. 6.3 this correlation is depicted for three matrices, that are to some extent representative for the tested data. Each point corresponds to one eigenpair (λ_i, q_i).

The negative correlation visible in fig. 6.3 suggests that the numerical inaccuracies stem from the cancellation and hence from the loss in precision during in the computation of the *Weinstein matrix*. More precisely during the scaling operation

$$(\Lambda_1 \oplus \Lambda_2 - \lambda I)^{-1} U \tag{6.9}$$

in `weinsteinMatrix`. It is noteworthy that even a relatively large distance $\mathfrak{D}_i \approx 10^{-6}$ may lead to mediocre residuals.

6.2.3. The Singularity of a Weinstein Matrix

It often happens that an eigenvalue is approximated to full working precision, that is that the eigenvalue is located in (a, b), with $b - a < 2|b|\epsilon$. Since $\hat{\omega}_i < \epsilon$ is the main stopping criterion of *zeroin/bisection*, it follows that in these cases the smallest absolute eigenvalue $\hat{\omega}_i$ of the corresponding *Weinstein matrix* is not zero to working precision. This issue was already mentioned in section 5.1.3 and section 5.2.1.

Figure 6.4 shows the correlation between the individual residual ($\mathfrak{R}_i$) and the smallest absolute eigenvalue of the corresponding *Weinstein matrix*

$$\tilde{\omega}_i = \min_j \frac{|\omega_j|}{\max(1.0, |\lambda_i|)}, \quad \omega_j \in \begin{cases} \sigma(W(\lambda_i)), & \text{if } \lambda_i \notin \sigma(B_0), \\ \sigma(W_e(\lambda_i)), & \text{if } \lambda_i \in \sigma(B_0). \end{cases} \tag{6.10}$$

Note that $\tilde{\omega}_i = \hat{\omega}_i / \max(1.0, |\lambda_i|)$ which allows the stopping criterion for *Brent's zeroin* and *bisection* to be written as

$$\tilde{\omega}_i < \epsilon, \tag{6.11}$$

see eq. (6.6). The correlation is almost perfect suggesting the relation[1]

$$\tilde{\omega}_i < \varepsilon \implies \mathfrak{R}_i < \max(\varepsilon, \epsilon), \tag{6.12}$$

where ϵ is the machine precision. For the three matrices plotted in fig. 6.4 the *Pearson correlation coefficient* evaluates to 0.899, 0.93 and 0.925 for `R2_3000_24_15597`, `W3_3000_24_14283` and `G1_3000_24_3608` respectively. Note that points with $\hat{\omega}_i < 10^{-18}$ are not plotted. For those points the residual stays at approximately ϵ which hinders the correlation and thus effects the correlation coefficient.

The relation eq. (6.12) suggests that eigenvectors computed with a *Weinstein matrix* singular to working precision have a residual $\mathfrak{R}_i < \epsilon$. Version 7 (`dsbkdcA3`) takes advantage of this insight by locating the eigenvalues up to quad-precision to guarantee $\tilde{\omega} < \epsilon_{double}$, see section 6.4.

Figure 6.5 correlates $\mathfrak{D}_i$ with $\tilde{\omega}_i$. This suggests that a small distance $\mathfrak{D}_i$ makes it less probable to find an eigenvalue λ_i that yields a singular *Weinstein matrix*. A non-singular *Weinstein matrix* in turn leads to inaccurate eigenvalues. In other words, the correlation depicted in fig. 6.3 may be an indirect one.

6.3. Inverse Iteration and the Extended Weinstein Matrix

Arbenz [4] reported similar residual as seen in fig. 6.1. He proposed a second version using *inverse iteration* for the computation of the eigenvectors. Unfortunately, this algorithm is not competitive in terms of runtime.

To evaluate *inverse iteration* a reference implementation `dsbein` is presented in section 3.1. It uses a tridiagonalisation based `LAPACK` routine to calculate all eigenvalues and afterwards computes the corresponding eigenvectors using a banded *inverse iteration* method. Note that computing all eigenvalues using a tridiagonalisation based method is extremely fast since the rotations do not need to be accumulated to form Q, see section 2.3.1.2. Hence there is no advantage in using the *divide and conquer by extension* method to locate the eigenvalues as compared to using e.g. `L/dsbedc`, which uses a bulge chasing procedure followed by the tridiagonal QR algorithm.[2] As expected `dsbein` is fast for test data with well separated eigenvalue but relies on re-orthogonalisation for matrices with clustered eigenvalue. For these test matrices the complexity degenerates to $\mathcal{O}(n^3)$.

[1] For most points in fig. 6.4 the relation is even stronger, e.g. $\tilde{\omega}_i < \varepsilon \implies \mathfrak{R}_i < c\max(2\varepsilon, \epsilon)$, where c is a constant.

[2] This is true for a sequential algorithm. A more detailed analysis is needed to evaluate this statement for a parallel machine.

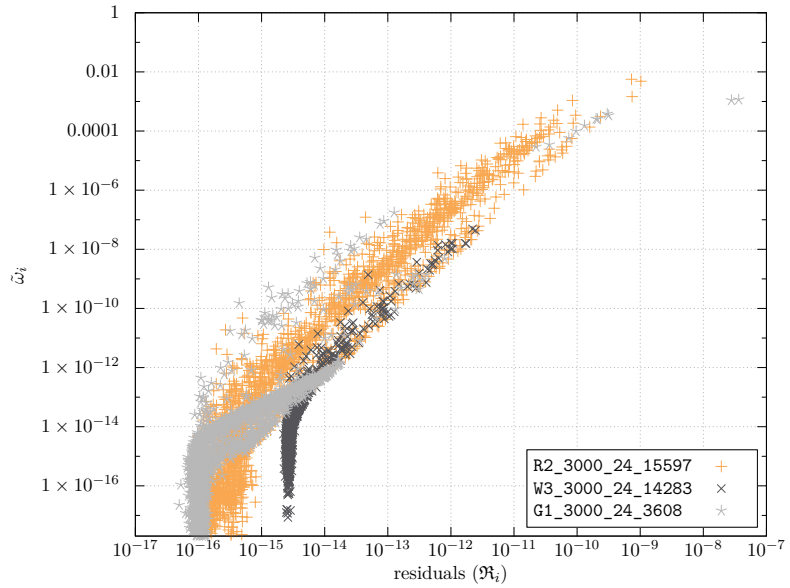


Figure 6.4.: Correlation between residual $\mathfrak{R}_i$ and the smallest eigenvalue of the corresponding *Weinstein matrix* ($\tilde{\omega}_i$).

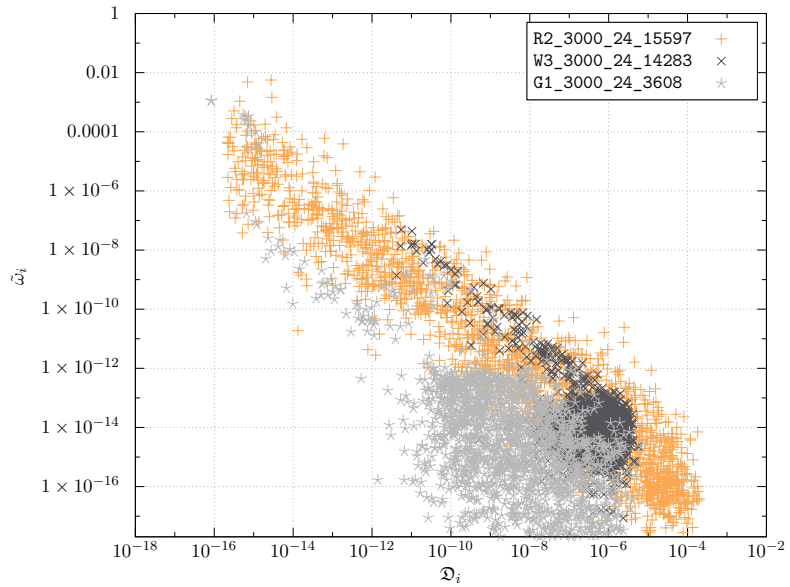


Figure 6.5.: Correlation between $\mathfrak{D}_i$ and $\tilde{\omega}_i$.

This can be seen in **??** which shows the total runtime (left) and the complexity (right) of `dsbein`. The matrices of type `U1` have well separated eigenvalues for small n but as all eigenvalues lie in the interval $(0, 1)$ the gaps between the eigenvalues decreases with increasing n making re-orthogonalisation necessary. This shows that the approach taken by [4] is not competitive in terms of both runtime and accuracy compared to *Cuppen's divide and conquer* algorithm. See chapter 7 for a more detailed analysis of the reference implementation `dsbein`.

The second modification Arbenz [4] described is to use the *extended Weinstein matrix* when locating an eigenvalue not only for value $\lambda_i^{(0)} \in \sigma(B_0)$ but also for values in the neighbourhood thereof: $\lambda \in \mathcal{U}(\lambda_i^{(0)})$. To justify this he showed that the matrices in eq. (3.17) are congruent in an neighbourhood $\mathcal{U}(\lambda)$ with

$$G_e = \begin{pmatrix} I - QQ^t & -(B_0 - \lambda)^\dagger V & Q \\ 0 & I & 0 \\ Q^t & 0 & 0 \end{pmatrix} \tag{6.13}$$

and

$$(B_0 - \lambda)^\dagger = \begin{cases} (I - QQ^\top)(B_0 - \lambda)^{-1}(I - QQ^\top), & \xi \in \mathbb{U}(\lambda)\backslash\{\lambda\}, \\ (B_0 - \lambda)^+. \end{cases} \tag{6.14}$$

For the given test data this modification yields no improvements. Hence we saw no need to use the computationally more expensive *extended Weinstein matrix* in the neighbourhoods of $\lambda_i^{(0)} \in \sigma(B_0)$.

6.4. Version 7 - Extended Precision

As fig. 6.4 shows there is a strong correlation between the smallest absolute eigenvalue $(\tilde{\omega}_i)$ of the *Weinstein matrix* $W(\lambda_i)$ and the accuracy of the eigenvector corresponding to λ_i. This version (`dsbkdcA3`) takes advantage of this insight by determining the eigenvalues to higher precision if needed. The approach is outlined in the following:

An eigenvalue is located as usual in double-precision. If $\hat{\omega} > \epsilon$ on exit of e.g. *Brent's zeroin* and thus the interval (a, b) is minimal with

$$|b - a| < \epsilon_{double}|b| = 2.2 \cdot 10^{-16}|b|, \tag{6.15}$$

the interval variables get copied into `__float128` variables, here denoted as a_q and b_q.[3] In a next step the eigenvalue is located up to a precision of

$$|b_q - a_q| < \epsilon_{quad}|b| = 1.9 \cdot 10^{-34}|b|, \tag{6.16}$$

where ϵ_{quad} denotes the quad-precision machine epsilon. To keep the algorithm competitive it is important to minimise the use of extended precision operations. It turns out to be sufficient to calculate the scaling operation $(\Lambda - \lambda I))^{-1}$ in `weinsteinMatrix` (see algorithm 11 line 5) in higher precision. More precisely, only the calculation of the scaling factor $1/(\lambda_i^{(0)} - \lambda)$ needs to be done in quad-precision, the vector scaling operation is done via `dscal` in double-precision. In total $b_1 + b_2$ subtractions and divisions need to be performed in quad-precision for the computation of a *Weinstein matrix*. Since the resulting *Weinstein matrix* of λ_q has double-precision `LAPACK` can further be used to calculate $\sigma(W(\lambda_q))$.

This approach is used in both `findEigenpair` which uses *Brent's zeroin* and `findEigenpairInertia` which uses *bisection*. In each iteration of *zeroin*/*bisection* $b_1 + b_2 + d$ floating point operations are performed in quad-precision, where d denotes the number of flop needed to updated the interval, e.g. $d = 3$ in case of *bisection*.

6.4.1. Results

The results are shown in fig. 6.6. The residuals have drop significantly but are still not as good as those of e.g. *Cuppen's divide and conquer* algorithm. The orthogonality errors in fig. 6.6 (right) are slightly worse than those achieved by the *multiple relatively robust representation* method, see chapter 7. Unfortunately, the orthogonality is highly dependent on the input data and may approach 1 for extreme cases, e.g. type `S1`.

Two further details are noteworthy. First note that `R1` has a higher residual then `R2` for $r = 24$. This is expected up to a factor of 25, since matrices of type `R2` have a spectral norm of 1 whereas the largest eigenvalue of `R1` is approximately 25 (in case $r = 24$), however, the difference is larger. A clear difference can also be seen in fig. 6.7 and fig. 6.8. The magnitude of these differences is surprising.

The second interesting fact is that the potentially easiest test data, meaning `U1` and `U2`, have that largest residuals. The matrix `U1_600_24_21343` is depicted in fig. 6.7 and fig. 6.8, which shows that cancellation is not particular high in the computation of the *Weinstein matrices*. For example the eigenpair of `U1_600_24_21343` with the largest residual $\mathfrak{R}_i \approx 10^{-13}$ in fig. 6.8 has $\mathfrak{D}_i > 10^{-5}$.

[3] In the following a subscript q denoted a quad-precision variable.

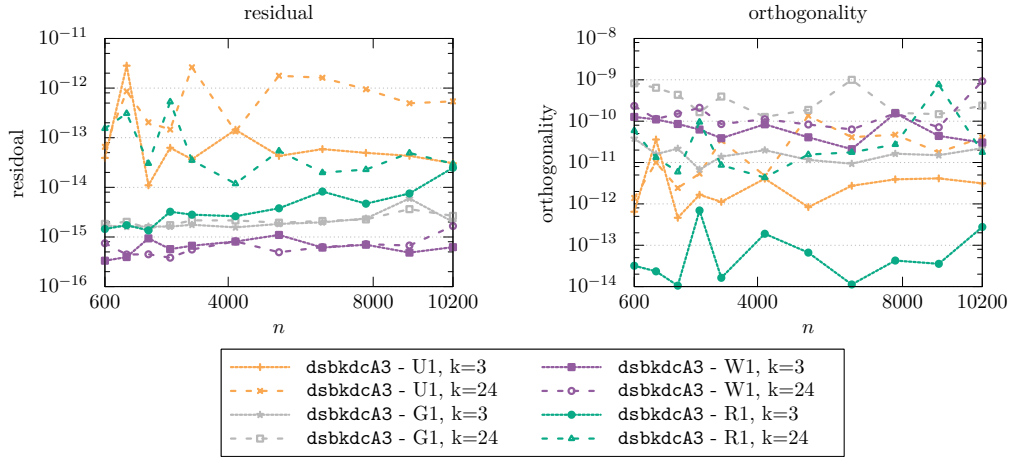


Figure 6.6.: Accuracy of `dsbkdcA3`.

Figure 6.7 shows the individual residuals and the minimal eigenvalues of the corresponding *Weinstein matrices* for three matrices. Figure 6.8 depicts the correlation between $\tilde{\omega}_i$ and $\mathfrak{D}_i$. The test matrices `U1_600_24_21343` and `R1_600_24_20884` represent the worst cases from fig. 6.6, whereas `R2_600_24_27679` has a residual of $\mathfrak{R} < 10^{-14}$. Most residuals $\mathfrak{R}_i$ are accurate to double-precision, but a few inaccurate eigenpairs yield the high residuals in fig. 6.6. A possible solution is be to further increase the precision, which would only affect eigenpairs with $\tilde{\omega}_i > \epsilon$, however, this would not improve the orthogonality error for matrices with tightly clustered eigenvalues, see chapter 7.

6.5. Version 8 - Subproblems with Eigenvalues of Multiplicities Larger One

The test data analysed so far have no eigenvalue $\lambda_i^{(0)} \in \sigma(B_0)$ with multiplicity $\mu > 1$, meaning that

$$|\lambda_i^{(0)} - \lambda_{i-1}^{(0)}| > \epsilon_{double} \cdot \max(1.0, |\lambda_i^{(0)}|), \quad \forall i. \tag{6.17}$$

However this situation does occurs for test matrices with tightly clustered eigenvalues as e.g. `S2`. In these cases the dimension of the *extended Weinstein matrix* corresponding to this particular eigenvalue $\lambda_i^{(0)}$ increases, see theorem 3.2.1 or more specifically eq. (3.18).

If two eigenvalues $\lambda_i^{(0)}$ and $\lambda_j^{(0)}$ of B_0 are close but not equal the corresponding eigenvectors may not be orthogonal. This section evaluates the approach of considering two close eigenvalues of B_0 as one with $\mu > 1$. Therefore this version (`dsbkdcA4`)

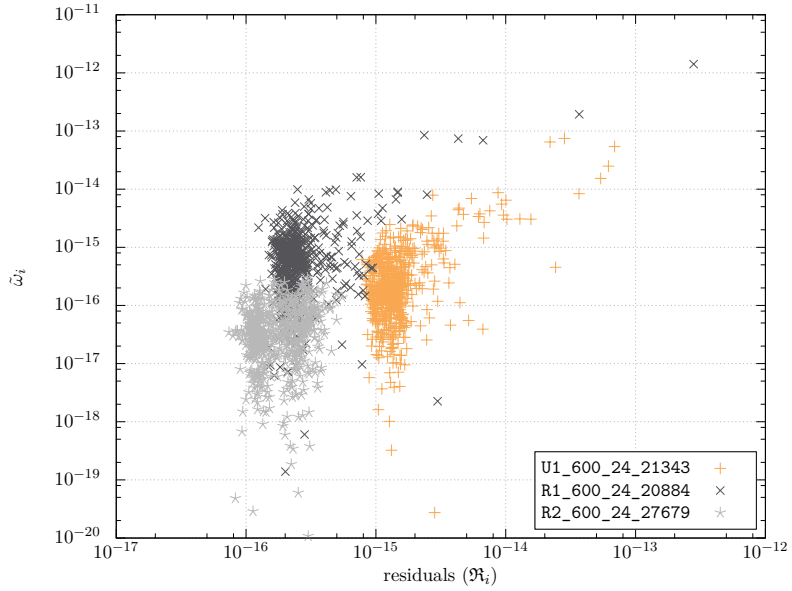


Figure 6.7.: Correlation between residual $\mathfrak{R}_i$ and the smallest eigenvalue of the corresponding *Weinstein matrix* ($\tilde{\omega}_i$). Calculated using `dsbkdcA3`.

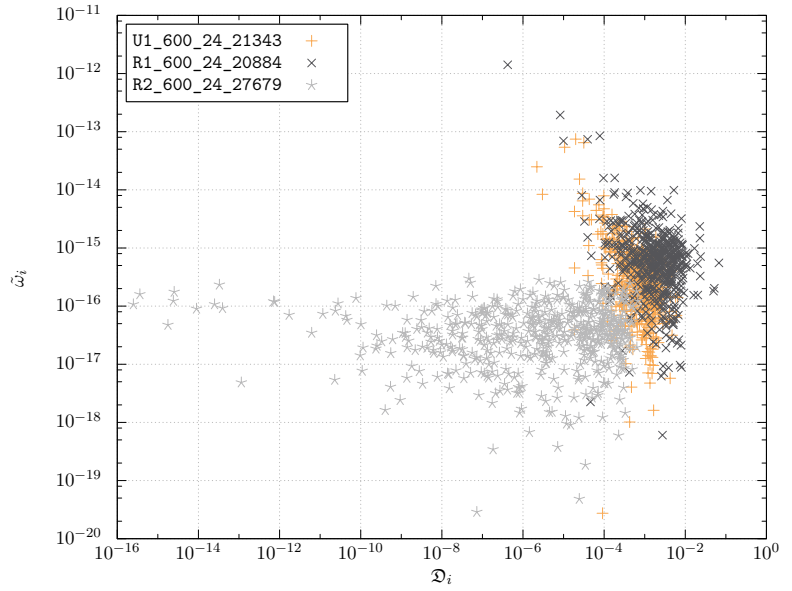


Figure 6.8.: Correlation between distance $\mathfrak{D}_i$ and the smallest eigenvalue of the corresponding *Weinstein matrix* ($\tilde{\omega}_i$). Calculated using `dsbkdcA3`.

subsumes two eigenvalue of B_0 into one with $\mu > 1$ if

$$|\lambda_i^{(0)} - \lambda_{i-1}^{(0)}| \leq \epsilon \cdot |\lambda_i^{(0)}|, \tag{6.18}$$

which is the smallest possible threshold. The matrix `W3_21_3_1234` which is generated with $\varepsilon = \epsilon_{double}$, see section 4.2, has such eigenvalues:

$$|\lambda_{15}^{(0)} - \lambda_{14}^{(0)}| = 2.22 \cdot 10^{-16} < \epsilon \cdot |\lambda_i^{(0)}|, \quad \lambda_{14}^{(0)} \approx 1. \tag{6.19}$$

Unfortunately, this approach only leads to an increase of the residual from $6 \cdot 10^{-16}$ to $5 \cdot 10^{-14}$ and hence also to and increase of the orthogonality error (from $9 \cdot 10^{-3}$ to 0.9). This increase is caused by a slight perturbation of the eigenvalues, which differ by up to $5 \cdot 10^{-14}$ from those calculated without an artificial multiplicity. To conclude this approach yields less accurate eigenvalue and thus inaccurate eigenvectors.

6.6. Version 9 - Adaptions for Matrices with Tightly Clustered Eigenvalues

This last version of `dsbkdc` implements three modifications that only take effect for matrices with tightly clustered eigenvalues.

6.6.1. Numerical Instabilities in the Calculation of M

For each neighbouring pair in eq. (5.13), e.g. two eigenvalue of $\sigma(B_0)$, the number of eigenvalue $\lambda \in \sigma(B)$ in such an interval is given by M as calculated by eq. (5.16). For test data with tightly clustered eigenvalues the case $M < 0$ occurs, which indicates that an eigenvalue is set twice. An example of such a matrix is `S1_21_3_1234`. The residual for this particular matrix is $1.4 \cdot 10^{-16}$ and the orthogonality error is 0.844. No special difficulties could be observed for this matrix, e.g. the subproblem B_0 has no multiplicities. As a consequence the only possibility is to skip the intervals with $M < 1$ with the consequence that some eigenvalue are set twice.

Since the case $M < 0$ only occurs for matrices with extremely clustered eigenvalue it is likely that the problem is caused by the loss in precision due to the cancellation in $(\Lambda - \lambda I)^+$ when calculating the *extended Weinstein matrices*. Cancellation occurs because multiple $\Lambda_i \in \sigma(B_0)$ are close to $\lambda \in \sigma(B_0)$.

6.6.2. Minimal Interval while Isolating Eigenvalues

To calculate the eigenvalue in between a neighbouring pairs of eq. (5.13), e.g. in the interval $(\lambda_{i-1}^{(0)}, \lambda_i^{(0)})$, the subroutine `isolateEigenvalues` is called. This subroutine applies bisection until M intervals each containing one eigenvalue are found. However, for matrices with tightly clustered eigenvalues it may happen that the interval (a, b) gets minimal with

$$|b - a| < |b| \cdot \epsilon_{double}, \tag{6.20}$$

which makes it impossible to isolate the eigenvalues. The only possibility is to threat a as an eigenvalue of multiplicity $\mu = M$. In this case the M corresponding eigenvectors are set via eq. (5.20) with $x_2^{(j)}$ being the M eigenvectors of $W(a)$ corresponding to the M smallest absolute eigenvalues $\omega_j \in \sigma(W(a))$. Note that $\omega_j < \epsilon|a|$ is not guaranteed which may lead to inaccurate results.

A matrix where this problem occurs is `S1_5400_24_17555`. The corresponding residual is $4 \cdot 10^{-15}$ and the orthogonality error is 1.

6.6.3. Eigenvalues with Multiplicity Large One

In this section `isolateEigenvalues` is adapted to check for eigenvalues with multiplicity $\mu > 1$ to improve the orthogonality of the computed eigenvectors. The subroutine `isolateEigenvalues` applies *bisection* using the inertia of *Weinstein matrix* in an interval (a, b) until it has found M distinct intervals each containing one eigenvalue. Therefore in each step the *Weinstein matrix* $W(m)$ of $m = (a + b)/2$ and its inertia

$$\text{inertia}(W(m)) = (\pi, \nu, \zeta), \tag{6.21}$$

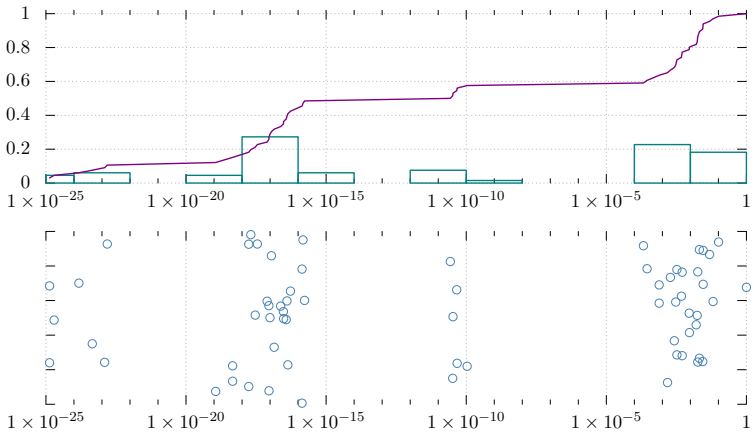
where ζ denotes the number of eigenvalues $\omega < \epsilon|m|$, is calculated. It often happens that $\zeta > 0$ and thus $W(m)$ is found to be singular. In this case m is an eigenvalue of B, but it cannot be set and *bisection* needs to be applied further. However, for matrices with tightly clustered eigenvalues it may happen that $\zeta > 1$ and thus an eigenvalue with multiplicity $\mu = \zeta$ is found. In previous versions these eigenvalues are isolated further and ultimately set separately. To instead set these eigenvalues together *bisection* only has to be continued as long as $\zeta < M$, hence until the eigenvalue with multiplicity $\mu > \zeta$ is isolated. As soon as $\zeta = M$ the eigenvalue can be set and the corresponding eigenvectors can be computed via eq. (5.20), where $x_2^{(j)} \in \mathcal{N}(W(m))$ are orthogonal and $N(W(m))$ has dimension M.

This situation occur e.g. for the matrix `W1_12_3_23413` which is generated with $\varepsilon = \epsilon$,

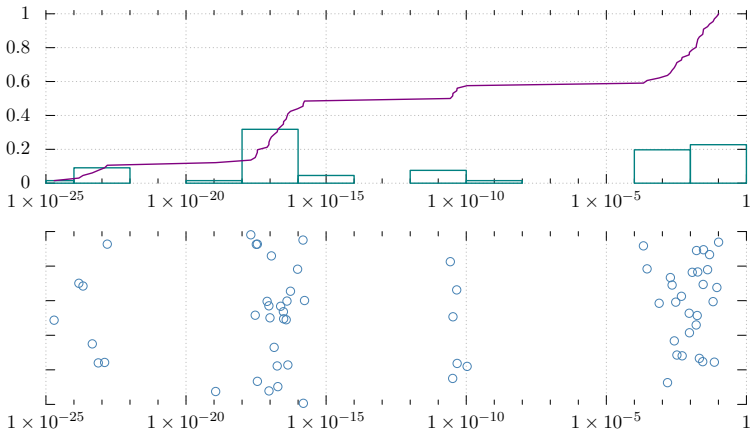
see section 4.2. If *bisection* is applied further and thus the eigenvalues are set separately the algorithm successfully finds all eigenpairs with a residual $\mathfrak{R} = 1.19 \cdot 10^{-16}$ and an orthogonality error $\mathfrak{O} = 1$. The orthogonality error is that high since two eigenvalues that are extremely close are set using the same equation. If, however, two eigenpairs are computed together the residual evaluates to $\mathfrak{R} = 1.23 \cdot 10^{-16}$ and the orthogonality error drops to $\mathfrak{O} = 0.101$. Note that the residual slightly increased, because the eigenvalue is not as accuracy as if *bisection* is applied further.

See fig. 6.9 for a detailed visualisation of the orthogonality error for the matrix `W1_12_3_23413`. Figure 6.9b depicts the case in which *bisection* is applied further until M distinct intervals, each containing one eigenvalue, are found. Figure 6.9b shows the improved version where an eigenvalue of multiplicity $\mu = 2$ is set in `isolateEigenvalues`. The main difference is that the dot product (the point furthest to the right) of the two eigenvectors calculated together drops from 1 to $\approx 10^{-16}$, however, orthogonality of the remaining eigenvectors is unchanged.

Unfortunately this change only takes affect for test data with highly clustered eigenvalues and the positive effect on the orthogonality error $\mathfrak{O}$ gets often masked by other inaccurate eigenvectors.



(a) Eigenvectors are calculated separately.



(b) Eigenvalues with multiplicity 2.

Figure 6.9.: The elements of the matrix $Q^\top Q - I$ are plotted as points where the x-axis is absolve value of the element and the y-axis is a random number for visualisation purposes. Above the distribution of the points is displayed using a histogram and a cumulative line. The original eigenproblem is `W1_12_3_23413`.

7. Experimental Comparison of Eigensolvers for the Bandsymmetric Eigenproblem

In this chapter the state-of-the-art approach of computing the spectral decomposition of a banded matrix B, that is using a tridiagonal reduction process followed by a tridiagonal eigensolver, is compared to bandsymmetric methods that operate only within the banded structure of B. The results are also of interest for the dense symmetric eigenproblem S, since modern numerical libraries use a two-stage tridiagonalisation approach to compute the spectral decomposition of S.

Note that the two-stage tridiagonalisation approach is only more efficient than the traditional one stage reduction if run on a parallel system. Therefore the approach of omitting the second reduction stage and using a bandsymmetric eigensolver instead of a tridiagonal one is only favourable if the algorithm runs efficiently on such a parallel machine. Fortunately the *divide and conquer* methods as proposed by [4] have properties that make them well parallelisable. For the tridiagonal analogue, that is *Cuppen's divide and conquer* algorithm, [50] and [51] presented efficient parallel implementations for a distributed memory architecture and a hybrid GPU-accelerated multicore system respectively. Haidar, Ltaief and Dongarra [33] presented a implementation of the *divide and conquer by modification* method for shared memory systems. It can be expected that similar speedups can be achieved with a parallel implementation of `dsbkdc`. However, in the following we will restrict the comparison to sequential algorithms.

Section 7.1 summarises the routines used throughout this chapter. Section 7.2 compares the tridiagonalisation based eigensolvers available in `LAPACK` and `PLASMA`. In section 7.3 the bandsymmetric eigensolvers introduced in chapter 3 are compared to `L/dsyevd`, which is the fastest method available in `LAPACK`.

7.1. Routines

In this subsection the tested routines are listed. They can be split in tridiagonalisation-based algorithms which are available in `LAPACK` or `PLASMA` and those working directly

on the banded structure of the test data. The first group consists of four algorithms. They differ either in the routine used for the tridiagonal reduction or in which tridiagonal eigensolver is used. As discussed in section 2.4.4 only the tridiagonal eigensolver `MR` and `DC` need to be considered. The `LAPACK` routines are denoted by the name of the driver routines, however, the tridiagonalisation routine, the tridiagonal eigensolver and the routine for the back-transformation are called separately to enable time measurements of the individual steps. The four algorithms are:

- `L/dsbevd`: Uses the band-to-tridiagonal reduction routine `L/dsbtrd` (bulge chasing algorithm using *Givens rotations*). The spectral decomposition is computed using `L/dstedc` (`DC`). The back-transformation is performed by `L/dstedc`.
- `L/dsbevr`[1]: As `L/dsbevd` but uses `L/dstemr` (`MR`) to solve the tridiagonal problem. The back-transformation is done via `dgemm`.
- `L/dsyevd`: Uses the full-to-tridiagonal reduction routine `L/dsytrd` (*Householder reduction*). `L/dstedc` (`DC`) is used to compute the spectral decomposition of the tridiagonal problem. The back-transformation is performed by `L/dormtr`.
- `P/dsyevd`: Similar to `L/dsyevd` but uses a two-stage reduction process, see section 2.3.2.

The second group of algorithms operate directly on the band-structure of the test matrices. Two of those algorithms were introduced by Arbenz [4] and can be thought of as generalisations of the tridiagonal divide and conquer algorithm. The last one, namely `dsbein`, is a reference implementation used to evaluate the *inverse iteration* method for banded matrices. The tested routines are:

- `dsb1dc`: *Divide and conquer by modification*, see section 3.2.3 and [27]. Uses r rank-one modifications.
- `dsbkdc`: *Divide and conquer by extension*, see section 3.2.2, chapter 5 and chapter 6. Uses one rank-r modification.
- `dsbkdcQP`: As `dsbkdc` but uses quad-precision to determine the eigenvalues to higher precision, see section 6.4.
- `dsbkdcRC`: As `dsbkdc` but uses recursion, with $\hat{n} = 1300$ for $r = 3$, see section 5.6.
- `dsbein`: A simple implementation of *inverse iteration* for banded matrices, see section 3.1.

[1] This driver routine does not exist in `LAPACK`.

7.2. Tridiagonalisation Based Methods

The runtime of tridiagonalisation based methods is dominated by the tridiagonalisation and back-transformation steps, see fig. 2.1. The `LAPACK` library contains one routine to reduce a banded matrix to tridiagonal form. This routine (`L/dsbtrd`) uses a bulge chasing procedure to maintain the banded form. A second routine exists (`L/dsytrd`) that tridiagonalises a dense matrix using *Householder reductions*. It turns out that `L/dsbtrd` is not competitive compared to `L/dsytrd` (not even for $r = 2$) if the transformation matrix V is required. Figure 7.1 shows the runtime for `L/dsyevd` which uses a full-to-tridiagonal reduction and `L/dsbevd` which uses a bulge chasing procedure. Both methods have a runtime complexity of $\mathcal{O}(n^3)$, however, `L/dsyevd`, is faster for all bandwidths and will thus be used to evaluate the bandsymmetric eigensolvers in the following section.

`PLASMA` uses a two-stage reduction process with a banded matrix as interim result. Unfortunately, no routine for the bandsymmetric eigenproblem is available in `PLASMA`, hence `P/dsyevd` is used instead. The runtime of `P/dsyevd` is depicted in fig. 7.1. Although `P/dsyevd` uses a full-to-band-to-tridiagonal reduction it is faster than `L/dsbevd`. This shows that the band-to-tridiagonal reduction (second stage) used in `P/dsyevd` is much more efficient than `L/dsbevd`.

Note that the runtime of the tridiagonal eigensolvers is independent of the matrix type, hence fig. 2.1 is representative for all tested matrices.

The eigensolvers in `LAPACK` and `PLASMA` achieve a very high level of accuracy if *Cuppen's divide and conquer* algorithm is used to solve the tridiagonal eigenvalue problem. The tridiagonal reduction and the back-transformation preserve the orthogonality of the computed eigenvectors, see e.g. section 2.3.1.3.

In comparison `MR` produces less accurate eigenvectors. The orthogonality error is bound by $\mathcal{O}(n\epsilon)$, but with a much higher constant as for `DC`. An in depth analysis of the tridiagonal eigensolvers is given by [19]. On our testing system `MR` failed to compute the spectral decomposition of some matrices of type `S2`. The runtime and accuracy for all other matrix types can be seen in fig. A.3 and fig. A.4 respectively.

7.3. Bandsymmetric Eigensolvers

7.3.1. Inverse Iteration

Arbenz [4] proposed a second algorithm that uses *inverse iteration* to compute the eigenvectors of a banded matrix. The implementation `dsbein` is used to evaluate this approach. Figure 7.3 shows the runtime-complexity for matrices of type `U1` (left) and

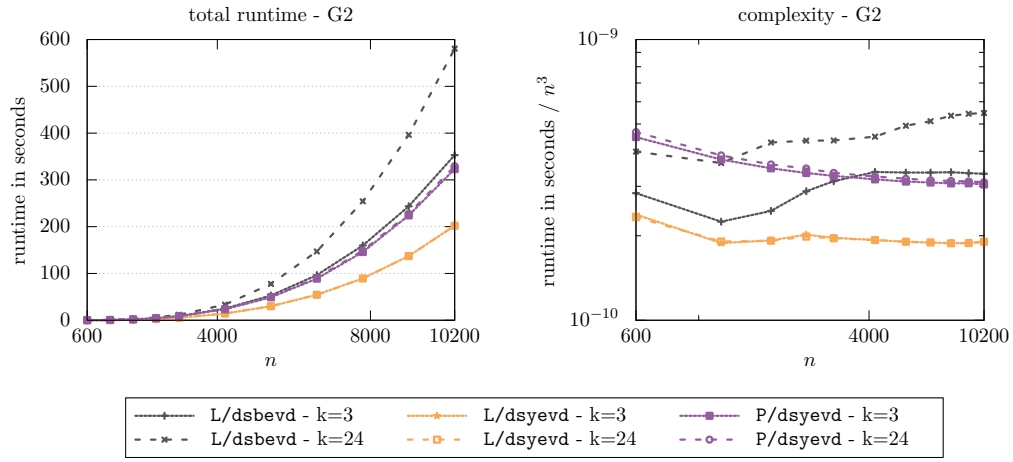


Figure 7.1.: Runtime of tridiagonalisation based methods for test matrices of type `G2`. `L/dsbevd` used a band-to-tridiagonal reduction, whereas `L/dsyevd` uses a full-to-tridiagonal reduction. `P/dsyevd` on the other hand uses the two-stage tridiagonalisation approach.

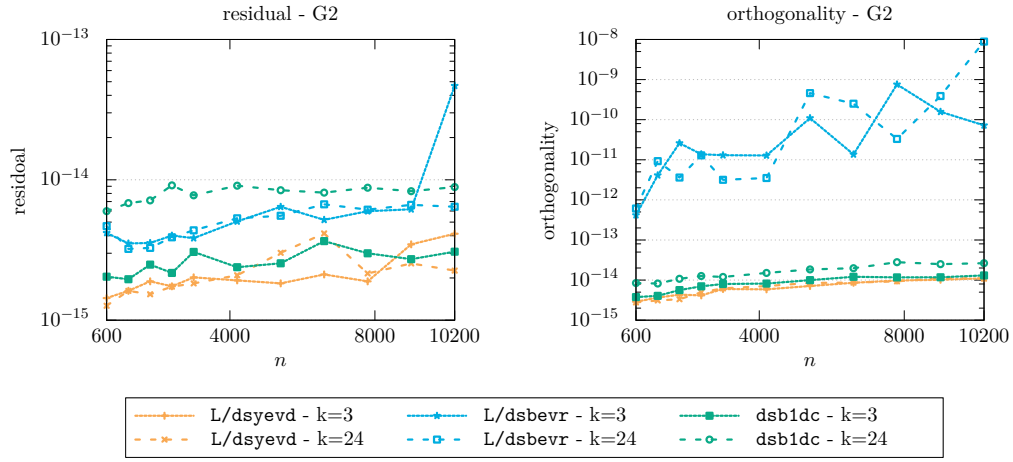


Figure 7.2.: Accuracy of `L/dsyevd`, `L/dsbevr` and `dsb1dc`, for test matrices of type `G2`.

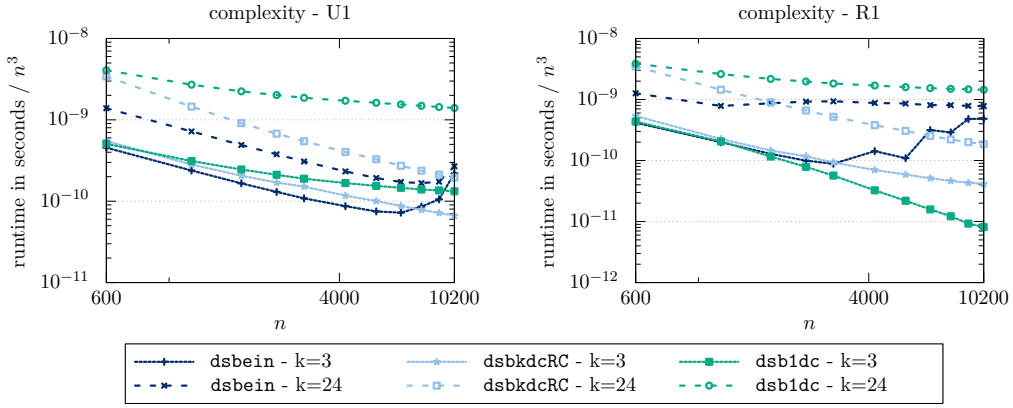


Figure 7.3.: Runtime of `dsbein`, `dsbkdcRC` and `dsb1dc`.

`R1` (right). As expected the complexity degenerates to $\mathcal{O}(n^3)$ as the dimension increases. Note that the gaps between eigenvalues decreases for all our test data as the dimension increases. That is because the spectral norm of the test matrices is independent of n, hence `dsbein` relies on re-orthogonalisation for large matrices.

Inverse iteration does not achieve the same level of accuracy as `DC` or `dsb1dc`, see fig. A.16 and [19]. Therefore it is not favourable to use `dsbein` over a tridiagonalisation based algorithm.

7.3.2. Divide and Conquer by Modification

The *divide and conquer by modification* algorithm proposed by Gansterer, Ward, Muller *et al.* [27] needs $2(r-1)n^3$ flops to compute the spectral decomposition of a banded matrix B. For small bandwidths the observed complexity is often lower (e.g. $< O(n^3)$ for $r = 3$) due to *deflation*, which simplifies the back-transformation of the eigenvectors. The runtime is shown in fig. 7.3 for matrices of type `U1` (left) and `R1` (right). Note that more *deflation* occurs for matrices of type `R1`, hence illustrating that the runtime is highly dependent on the problem (e.g. matrix type). Figure 7.4 depicts the runtime complexity for matrices of type `G2` (left) and `W2` (right) in comparison to `L/dsyevd`. It clearly shows that `dsb1dc` is only competitive for small bandwidths.

Finally fig. 7.5 illustrates the runtime dependency on r. For $n = 3000$ `dsb1dc` is faster than `L/dsyevd` for matrices with a semi-bandwidth $r < \hat{r} = 7$. This crossover point ($\hat{r}$) is independent of n, if the same amount of *deflation* occurs. However, as n increases the gaps between the eigenvalues of our test matrices decrease, which increases the amount of *deflation*.

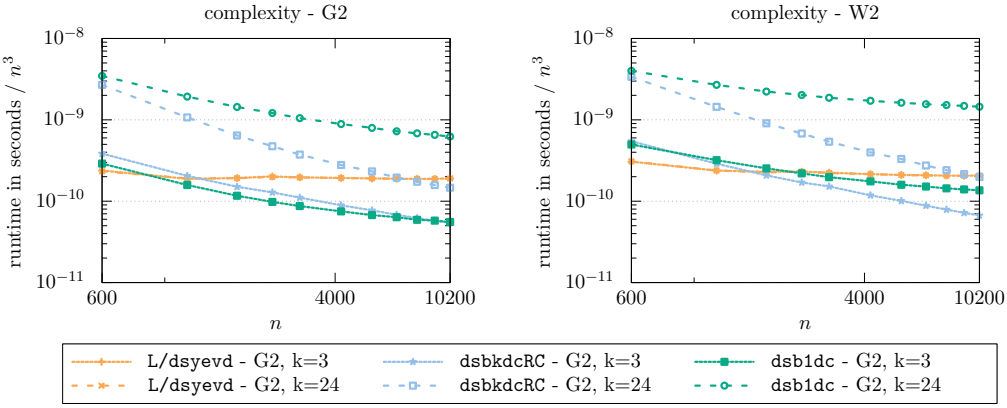


Figure 7.4.: Runtime of `dsbkdcRC`, `dsb1dc` and `L/dsyevd`.

The accuracy of `dsb1dc` is shown in fig. 7.2. Since the same techniques as in `DC` are applied the accuracy is also similar.

7.3.3. Divide and Conquer by Extension

The *divide and conquer by extension* method, as proposed by [4], uses one rank-r modification, whereas `dsb1dc` uses r rank-*one* modifications. The advantage of this approach is that the complexity is given by $4/3n^3$, and is hence independent of r. However, the root-finding method to locate the eigenvalue is much more expensive as for `dsb1dc` and needs $\mathcal{O}(n^2r^2)$ flops for the last rank-r update. In our testing runs the runtime of the zerofinder dominated the runtime up to $n = 10200$. The observed complexity it thus $< \mathcal{O}(n^3)$, see fig. 7.3. Figure 7.4 shows the runtime in relation to `L/dsyevd`. Note that in contrast to `dsb1dc` the runtime of `dsbkdcRC` is only slightly influenced by the matrix type.

The dependency on r is illustrated by fig. 7.5. It suggests that `dsbkdcRC` is fast in comparison to standard eigensolvers if the ratio r/n is small. Finally fig. 7.6 depicts the speedup over `L/dsyevd` (left) and `L/dsbevd` (right). To conclude, the runtime of `dsbkdc` is promising for small to medium bandwidths, however, only for large n.

7.3.4. Accuracy of DSBKDC

Unfortunately the spectral decomposition computed using `dsbkdcRC` (and `dsbkdc`, e.g. without recursion) is at best accurate to single-precision, meaning $\mathfrak{O}, \mathfrak{R} < \epsilon_{single} \cdot n$. In section 6.4 an improved version, which uses quad-precision to calculate more precise eigenvalues, was proposed. As can be seen in fig. 7.8 the eigenvectors of this improved

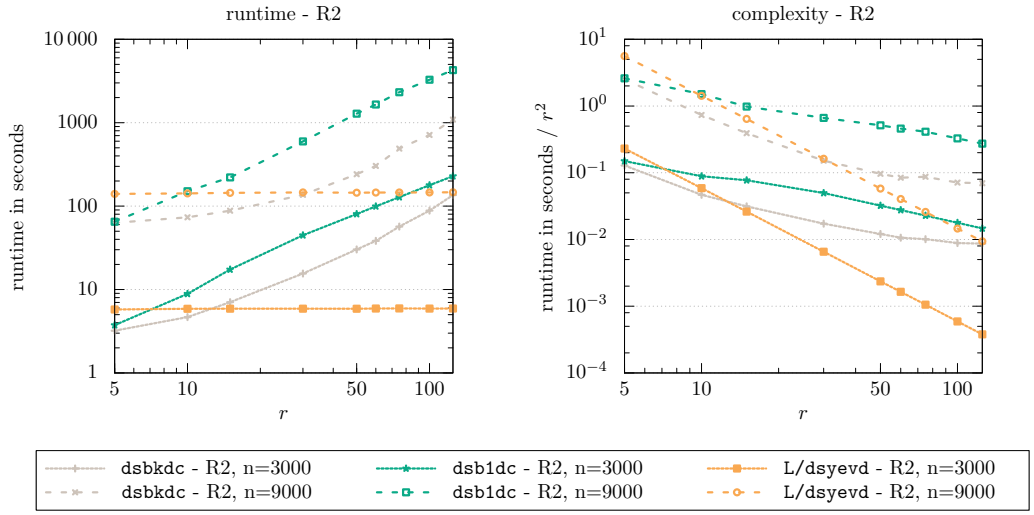


Figure 7.5.: Runtime for `dsbkdc` and `dsb1dc` in comparison to `L/dsyevd` for varying r. Note that a log-log scale is used for the total runtime as well, since the difference in runtime are that large. The matrices are of type `R2`.

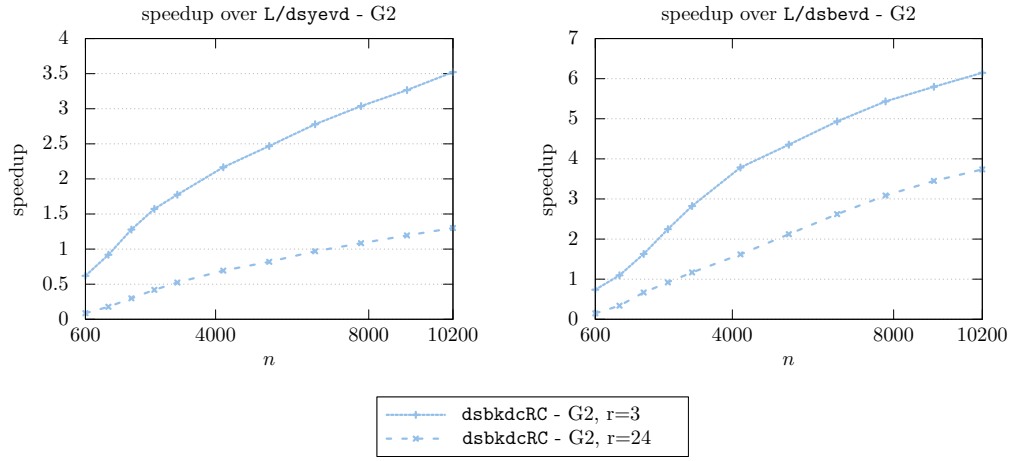


Figure 7.6.: Speedup of `dsbkdcRC` over `L/dsyevd` (left) and `L/dsbevd` (right) for matrices of type `G2`.

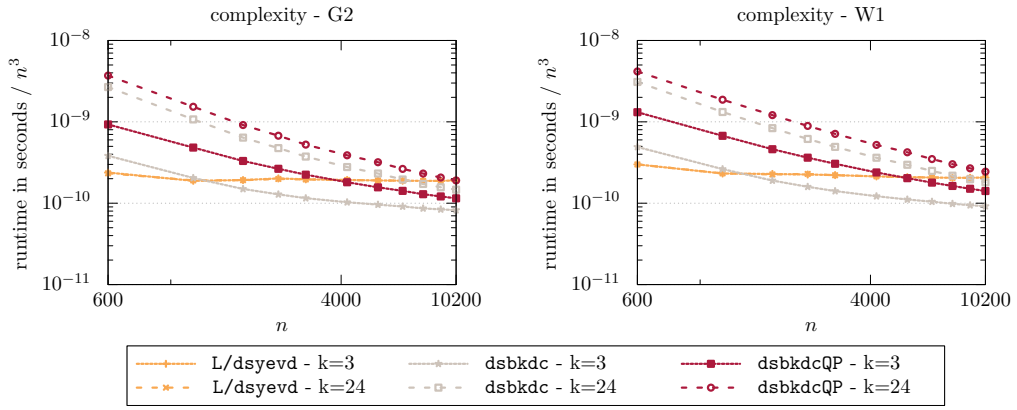


Figure 7.7.: Runtime of `dsbkdcQP` in comparison to `dsbkdc` and `L/dsyevd`.

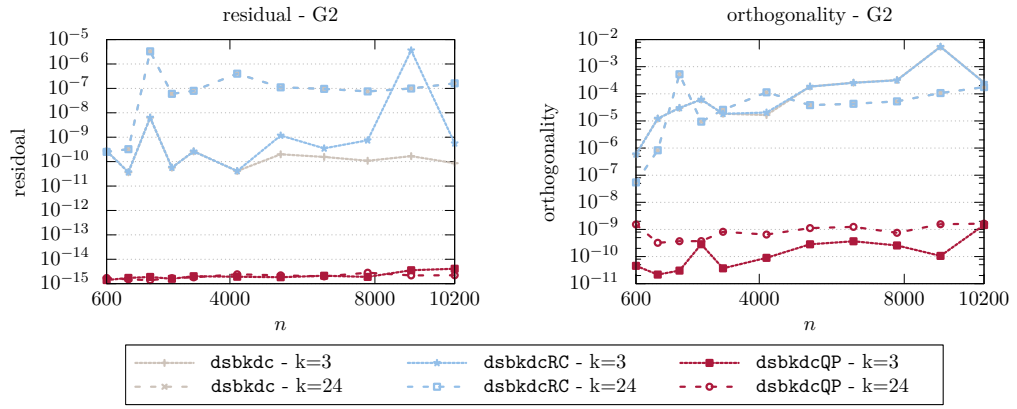


Figure 7.8.: Accuracy of `dsbkdc`, `dsbkdcRC` and `dsbkdcQP`.

version (`dsbkdcQP`) are accurate to double-precision, e.g. $\mathfrak{R} < \epsilon_{double} \cdot n$. For most test cases the orthogonality of the computed eigenvectors in similar to those computed with `MR`. However, for test matrices with tightly clustered eigenvalues, e.g. matrices of type `S1` or `S2`, the orthogonality error may still evaluate to one, see fig. A.12.

Figure 7.7 shows the runtime of `dsbkdcQP` in comparison to `dsbkdc` and `L/dsyevd` for matrices of type `G2` (left) and `W1` (right). Note that the costs for this higher accuracy increases with n^2 and hence does not change the complexity.

8. Conclusio

The first part of this thesis reviewed the state-of-the-art approach for solving a real symmetric eigenproblem. That is reducing a dense matrix S to tridiagonal form and solving the tridiagonal eigenproblem T. To get the spectral decomposition of the original problem a back-transformation of the eigenvectors is needed. The runtime for this approach is dominated by the tridiagonal reduction and the later back-transformation which both need $\mathcal{O}(n^3)$ flops. Modern tridiagonal eigensolvers on the other hand have a runtime complexity below $\mathcal{O}(n^3)$, hence they can be neglected in the overall runtime analysis.[1] The `LAPACK` library offers two tridiagonalisation routines: `L/dsytrd` for the full-to-tridiagonal and `L/dsbtrd` for the band-to-tridiagonal reduction. The latter uses a so called bulge chasing procedure to maintain the banded form. On our system `L/dsytrd` turned out to be faster for all bandwidths if the transformation matrix V is accumulated. If only eigenvalues are required, and hence V is not calculated, `L/dsbtrd` has a complexity of $\mathcal{O}(n^2 r)$ and is thus an order of magnitude faster than `L/dsytrd`.

Haidar, Tomov, Dongarra *et al.* [35] showed that a two-stage reduction approach is needed to exploit parallelism on modern computer architectures. `PLASMA` offers an eigensolver (`P/dsyevd`) that uses this two-stage tridiagonalisation scheme. Note that a two-stage reduction yields no advantage on a sequential machine and is thus slower than `L/dsytrd`. However, the second stage, that is the band-to-tridiagonal reduction, used in `PLASMA` is more efficient as `L/dsbtrd`, even on a sequential machine. Unfortunately, no isolated band-to-tridiagonal routine exists and the `LAPACK` analogue is not fully implemented yet. In conclusion, omitting the second stage of the two-stage tridiagonal reduction and using a bandsymmetric eigensolver instead of a tridiagonal one may lead to a significant speedup.

In the second part, starting with chapter 3, the bandsymmetric eigensolvers were introduced and analysed. The *inverse iteration* method was found to be not competitive when computing the complete spectral decomposition. Especially for matrices with clustered eigenvalues the runtime complexity quickly degenerated to $\mathcal{O}(n^3)$. In these cases tridiagonalisation based algorithms are more favourable as they have stronger

[1] E.g. `DC` has a runtime complexity of $\mathcal{O}(n^{2.3})$ for most matrices [43]. See also [19].

accuracy guarantees while being faster.

In section 3.2 the *divide and conquer algorithms* proposed by Arbenz [4] were discussed. The work done by Gansterer, Ward, Muller *et al.* [27] showed that the costs for the *divide and conquer by modification* method with r rank-one updates is highly dependent on the bandwidth of the given problem. The complexity evaluates to $2(r-1)n^3$ for the highest order term. For eigenproblems with small bandwidth the algorithm is fast, however, already for $r > 6$ a traditional tridiagonalisation based approach may be faster. The advantage of using r rank-one modifications instead of one rank-r modification is that the computed spectral decomposition is as accurate as if computed by `L/dsyevd` (which uses *Cuppen's divide and conquer algorithm*).

On the other hand the *divide and conquer by extension* algorithm – as proposed in [4] and implemented in chapter 5 and chapter 6 – uses one rank-r update. The runtime complexity of this algorithm is independent of r with the highest order term being $4/3n^3$. However, up to $n = 10200$ it was found that the runtime is dominated by the n^2 term which has an quadratic dependency on the semi-bandwidth r. This suggests that the ratio b/n is crucial to estimate the performance of `dsbkdc`. The crossover point for which `dsbkdc` is faster than `L/dsyevd` is approximately $n = 1200$ for $r = 3$ and $n = 7500$ for $r = 24$. Unfortunately, it is still unclear how to guarantee accurate eigenvectors for matrices with tightly clustered eigenvalues.

The implementation and optimisation of `dsbkdc` was discussed in chapter 5. The most notable speedup compared to the baseline implementation was achieved by blocking the back-transformation of the eigenvectors, enabling the use of level 3 `BLAS` operations. This reduced the runtime of the code which gives rise to the n^3 complexity of the algorithm. In section 5.6 it is shown that when recursion is fully utilised the observed complexity is below $\mathcal{O}(n^3)$ for matrices up to $n = 10200$ (even for $r = 3$).

To improve the accuracy of the calculated spectral decomposition the numerical instabilities were analysed in section 6.2. It was shown that the residual of an eigenpair (λ_i, q_i), e.g. $||Bq_i - \lambda_i q_i||_2/||B||_2$, correlates strongly with the smallest absolute eigenvalue of the *Weinstein matrix* corresponding to λ_i. This insight lead to an improved version (`dsbkdcQP`) that uses quad-precision to calculate more precise eigenvalues. For most test cases this modified version achieved similar levels of accuracy as the *Multiple Relatively Robust Representations* method. E.g. a residual $\mathfrak{R} < \epsilon n$ could be achieved for most matrices. However, for eigenproblems with tightly clustered eigenvalues $\mathfrak{O}$ may still evaluate to one.

In chapter 7 both tridiagonalisation based algorithms as well as bandsymmetric eigensolvers were numerically compared using banded matrices. It was shown that

`dsb1dc` is fast for small bandwidths, but not competitive for $r > 10$. The `dsbkdc` algorithm, on the other hand, is less sensitive to the bandwidth of the given matrix. For matrices with small bandwidth `dsbkdc` achieved speedups up to 3.5 in comparison to `LAPACK`. Unfortunately, the numerical instabilities that occur during the computation of the eigenvectors may still lead to high orthogonality errors. To make `dsbkdc` a viable alternative to state-of-the-art eigensolvers these instabilities need to be resolved first.

A. Additional Measurements

In the following additional measurements for matrices of all types are listed. The matrices have a semi-bandwidth of either $r = 3$ or $r = 24$. The dimension varies between $n = 600$ and $n = 10200$. Plots for the following routines are included (in that order): `L/dsbevd`, `L/dsbevr`, `L/dsyevd`, `dsb1dc`, `dsbkdc`, `dsbkdcQP`, `dsbkdcRC` and `dsbein`. See section 7.1 for short description of the routines.

For each routine 6 plots are shown. Three illustrating the runtime and three the accuracy of the routine. The runtime-plots show both the total runtime as well as the runtime scaled by $1/n^3$. The plots depicting the accuracy show on the left the residual $\mathfrak{R}$ and on the right the orthogonality error $\mathfrak{O}$.

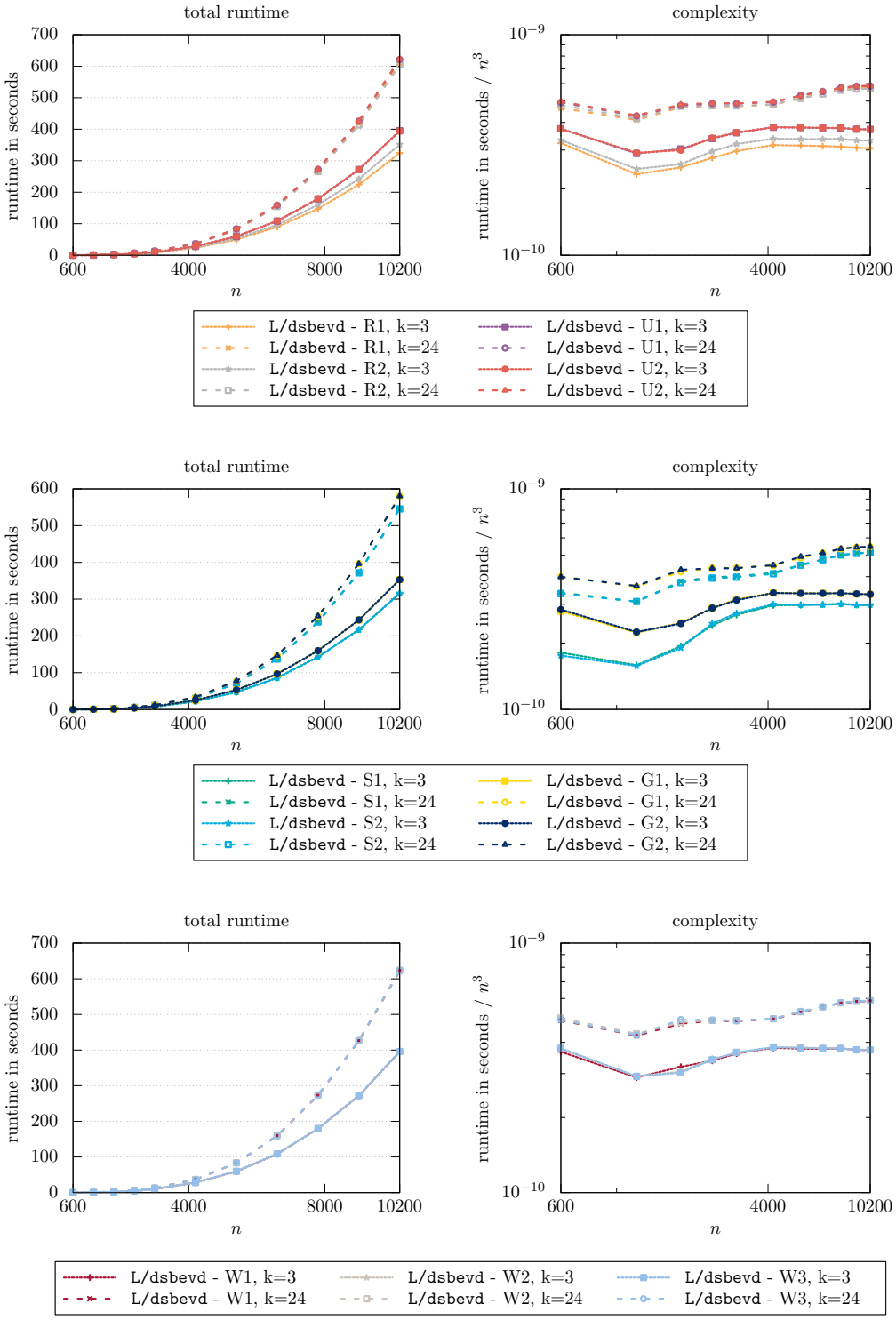


Figure A.1.: Runtime of `L/dsbevd`.

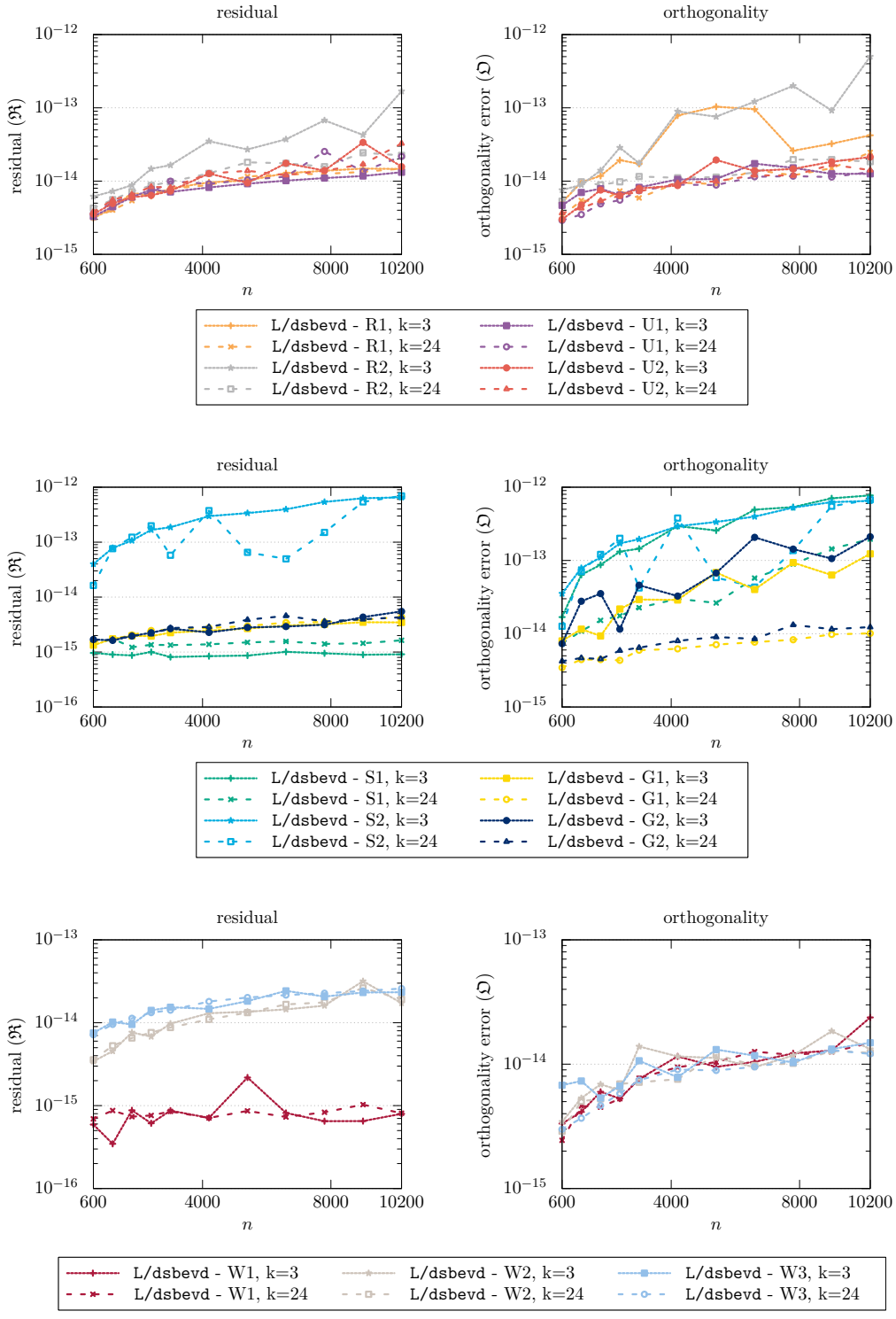


Figure A.2.: Accuracy of `L/dsbevd`.

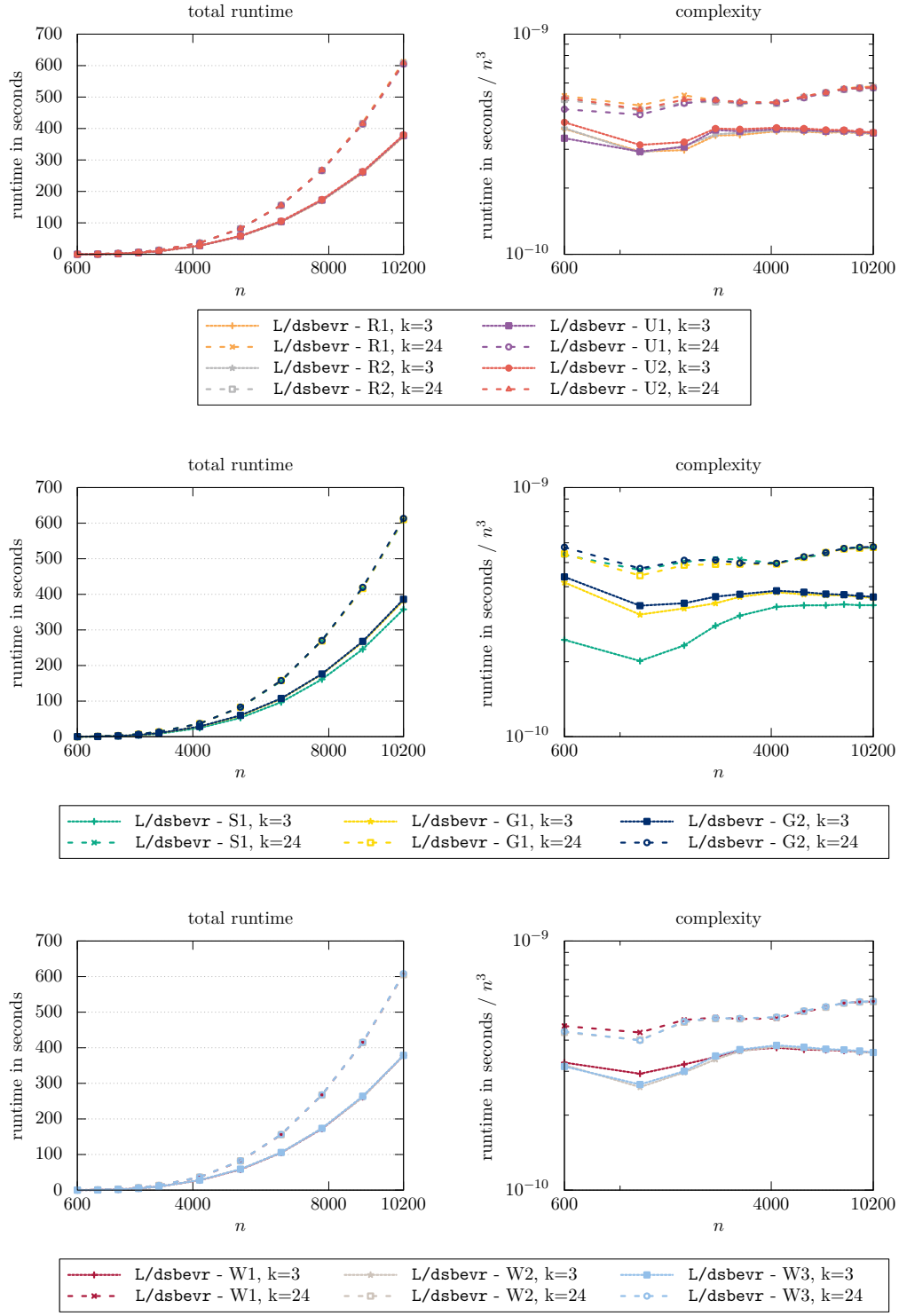


Figure A.3.: Runtime of `L/dsbevr`. No measurements are shown for matrices of type `S2`, since `MR` fails to computed the complex eigendecomposition for some matrices of this type.

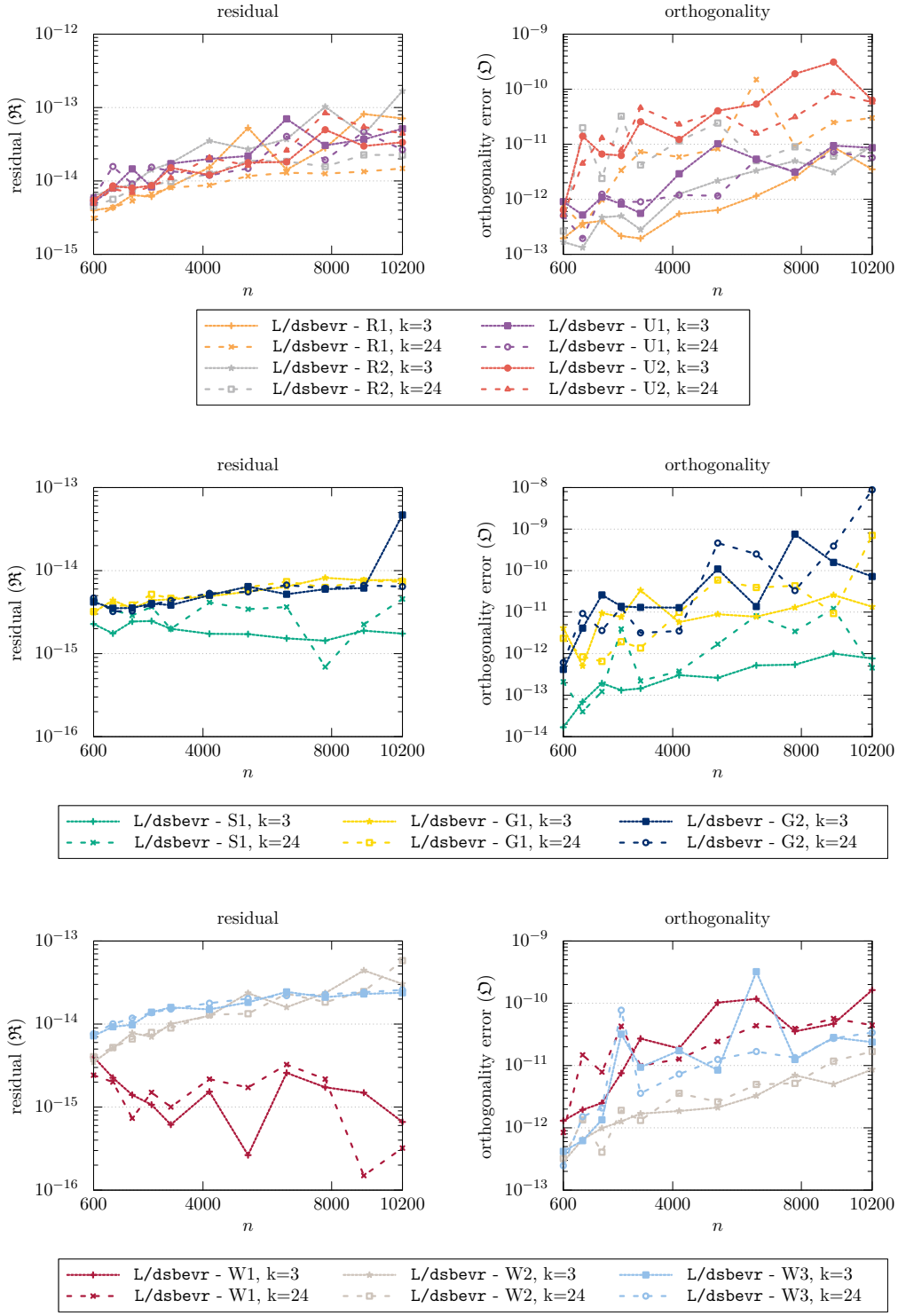


Figure A.4.: Accuracy of `L/dsbevr`. No measurements are shown for matrices of type `S2`, since `MR` fails to computed the complex eigendecomposition for some matrices of this type.

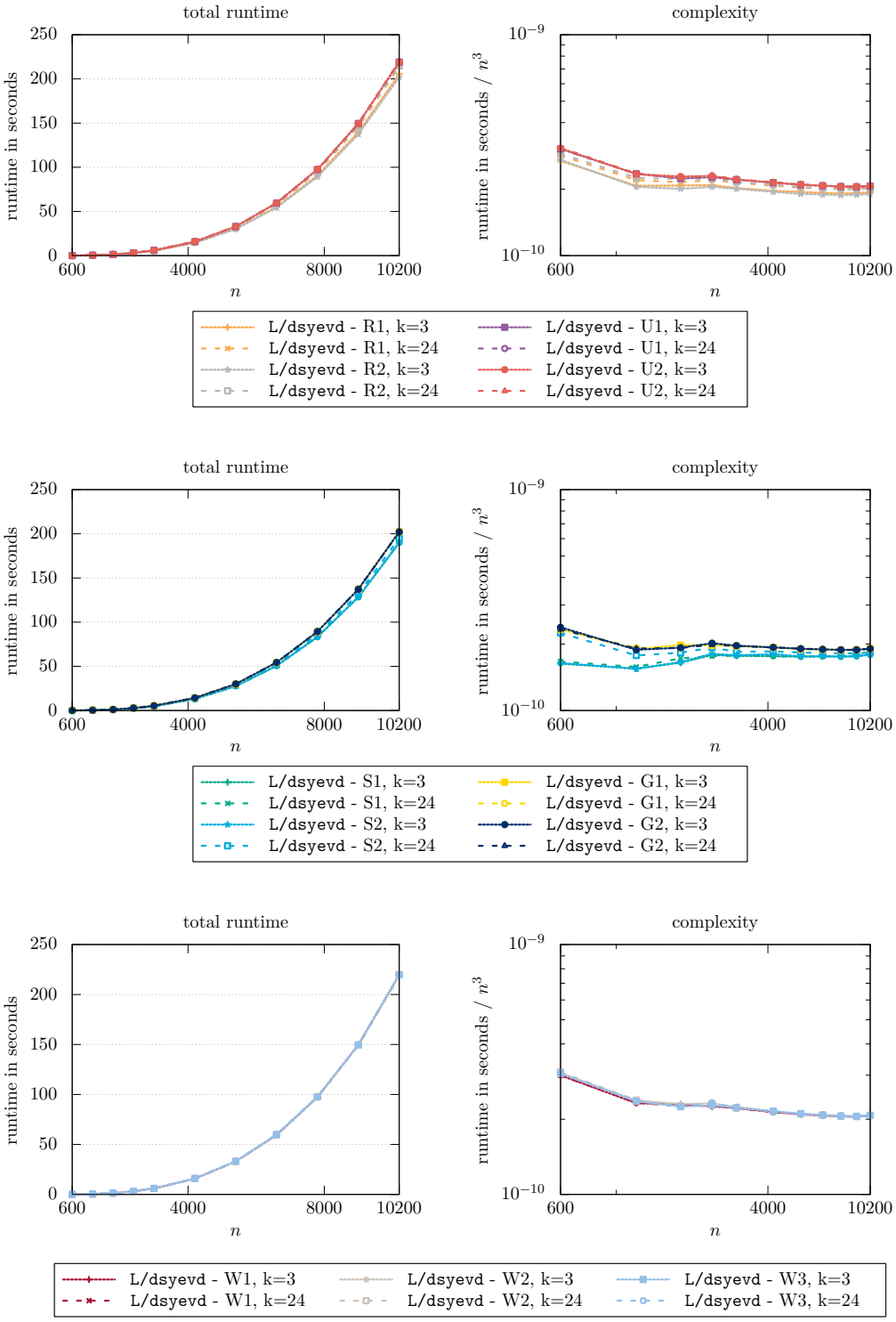


Figure A.5.: Runtime of `L/dsyevd`.

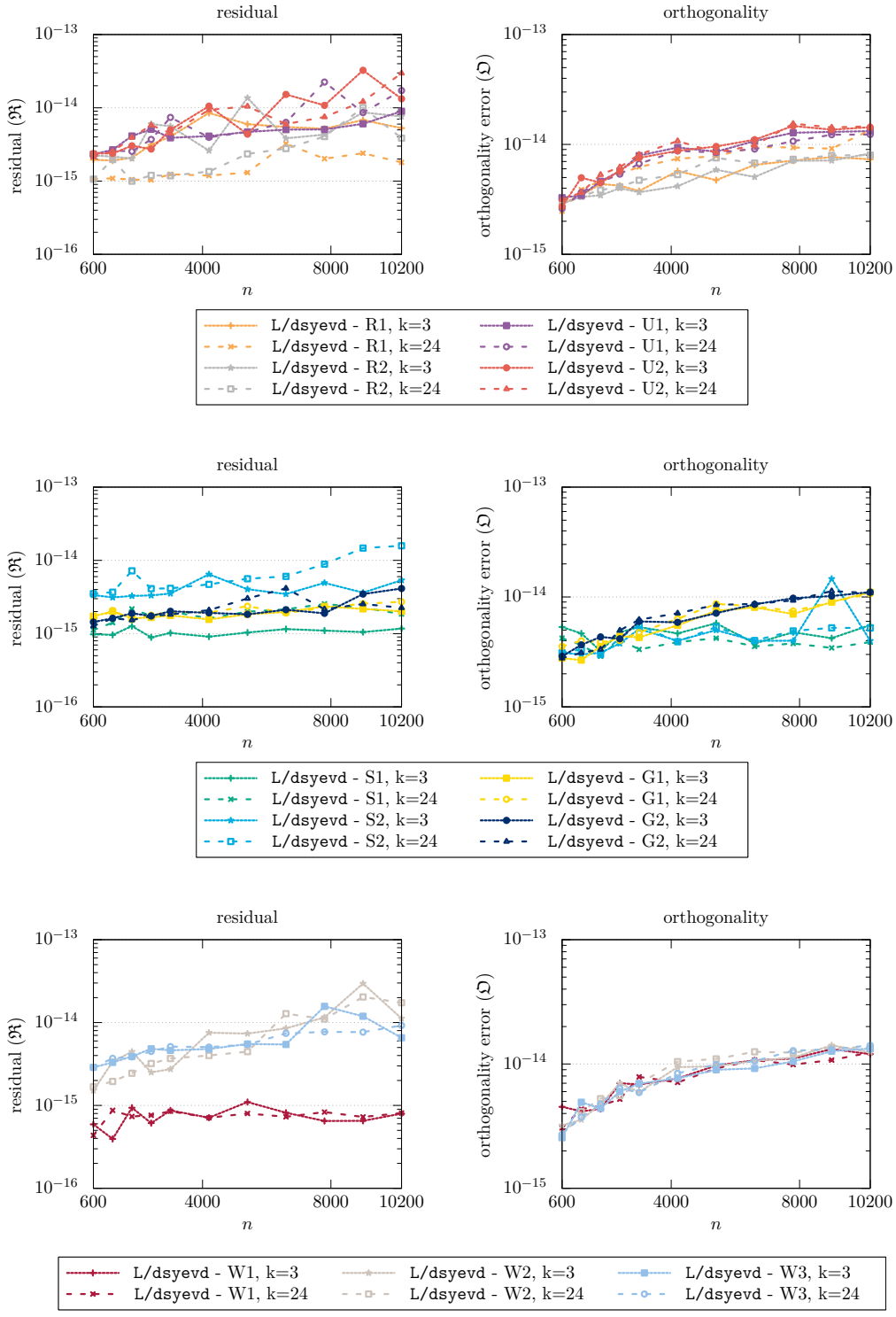


Figure A.6.: Accuracy of `L/dsyevd`.

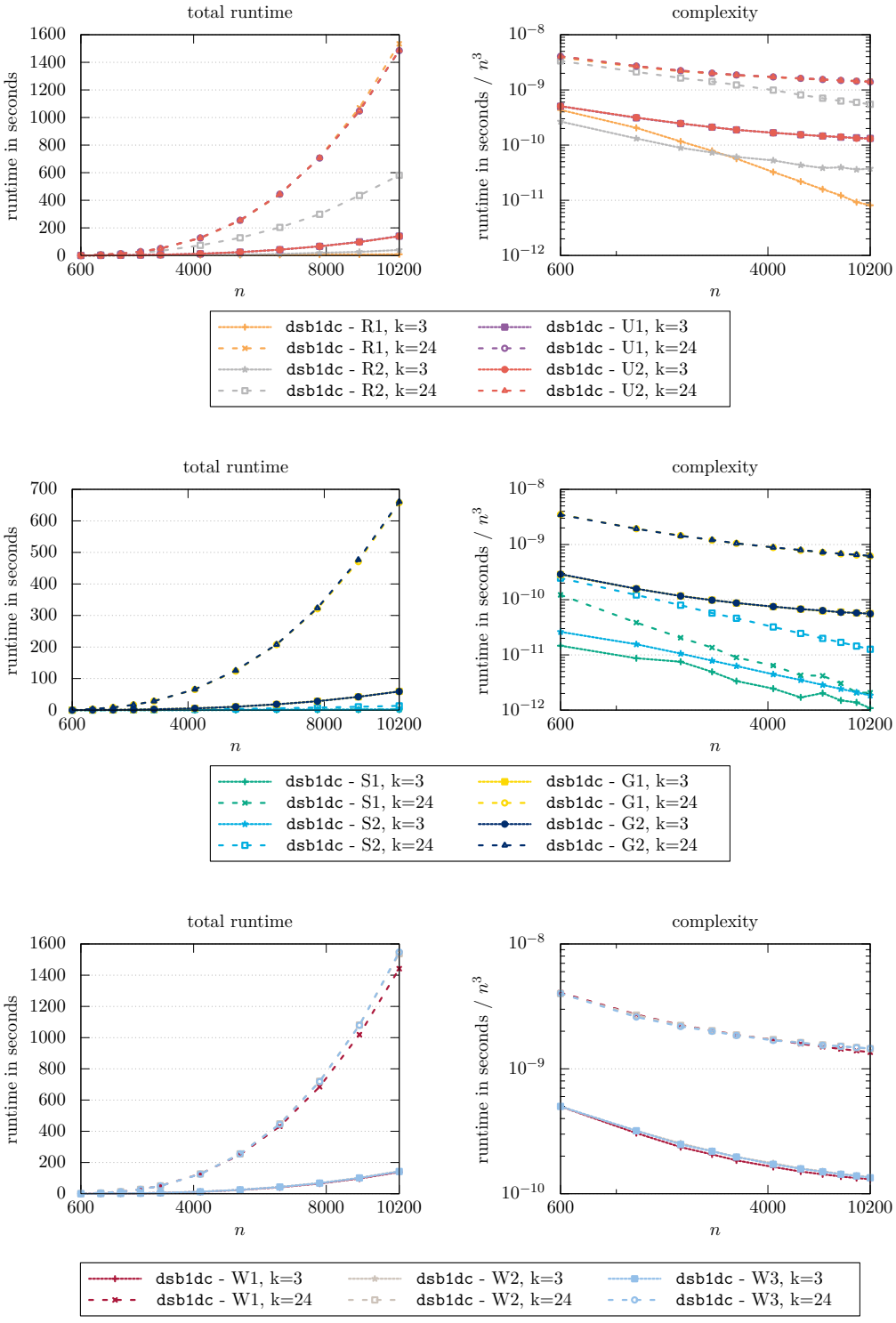


Figure A.7.: Runtime of `dsb1dc`.

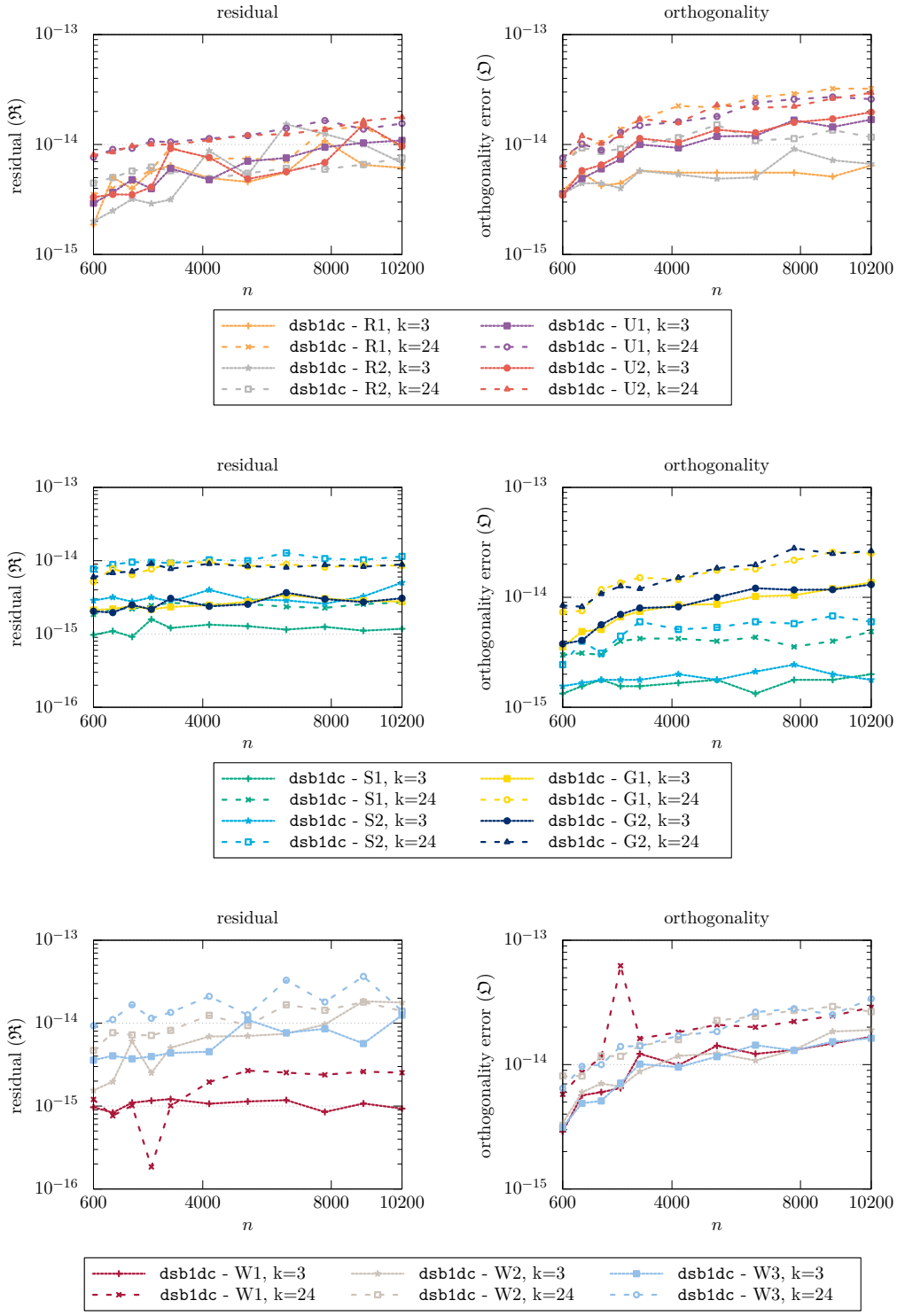


Figure A.8.: Accuracy of `dsb1dc`.

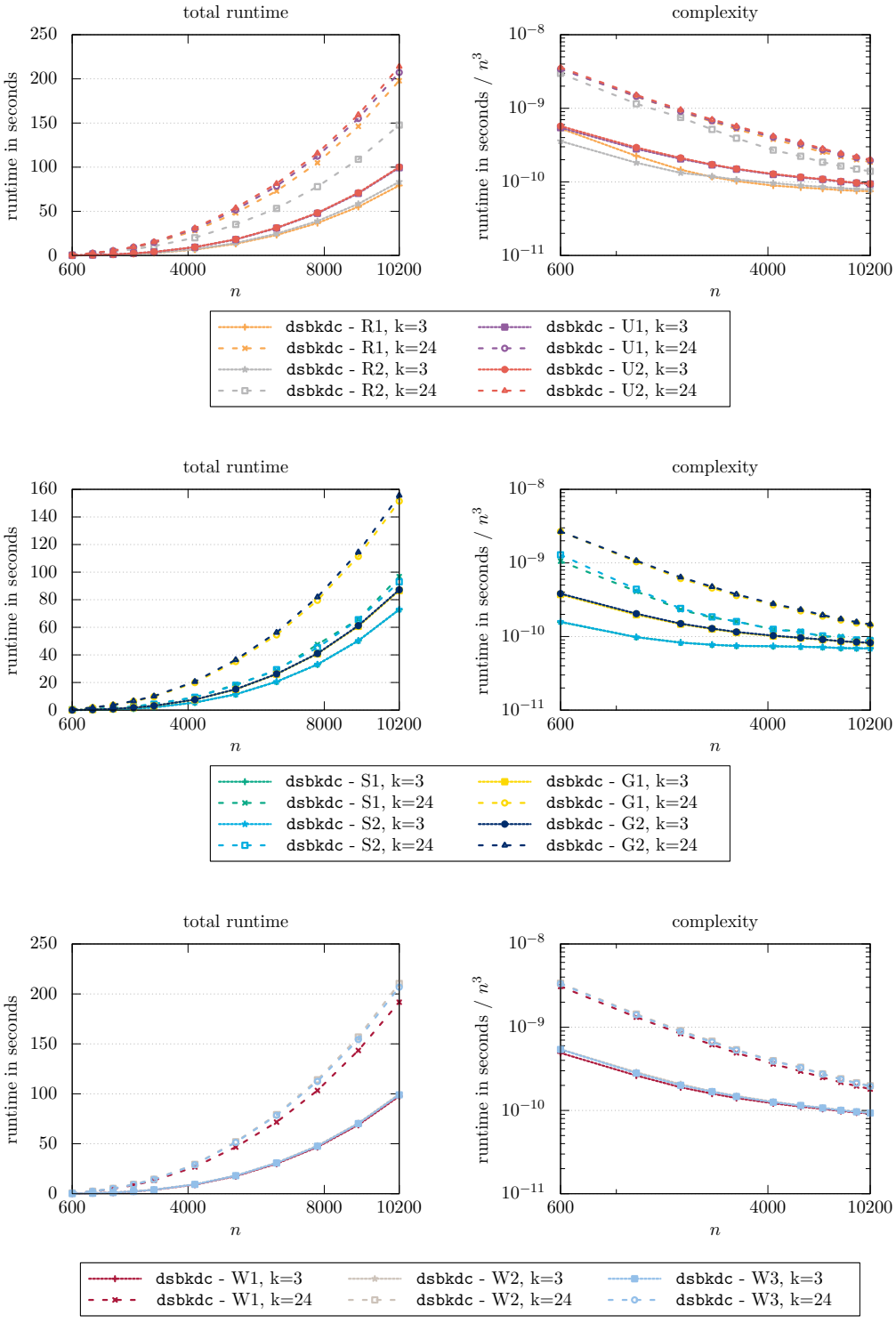


Figure A.9.: Runtime of `dsbkdc`.

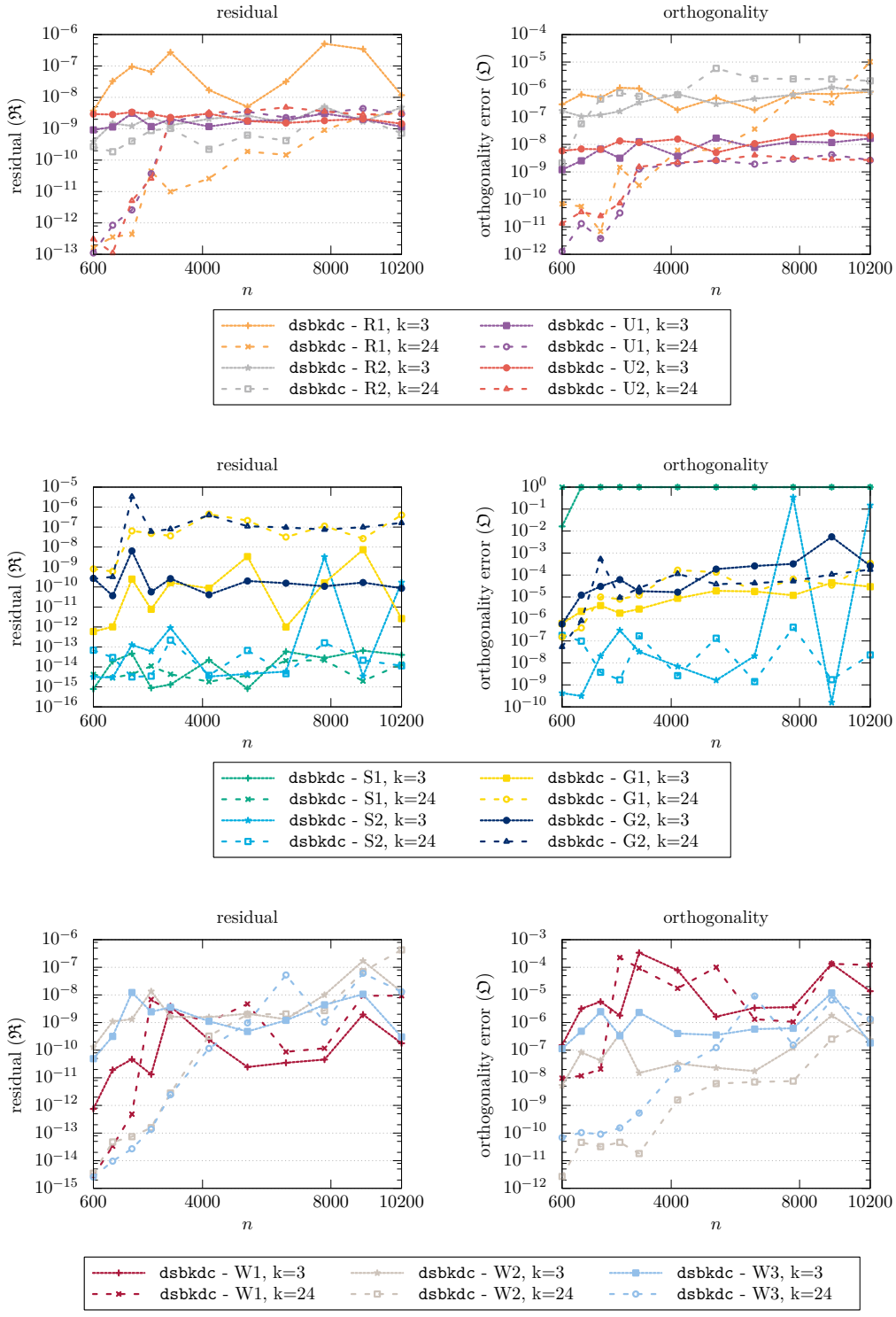


Figure A.10.: Accuracy of `dsbkdc`.

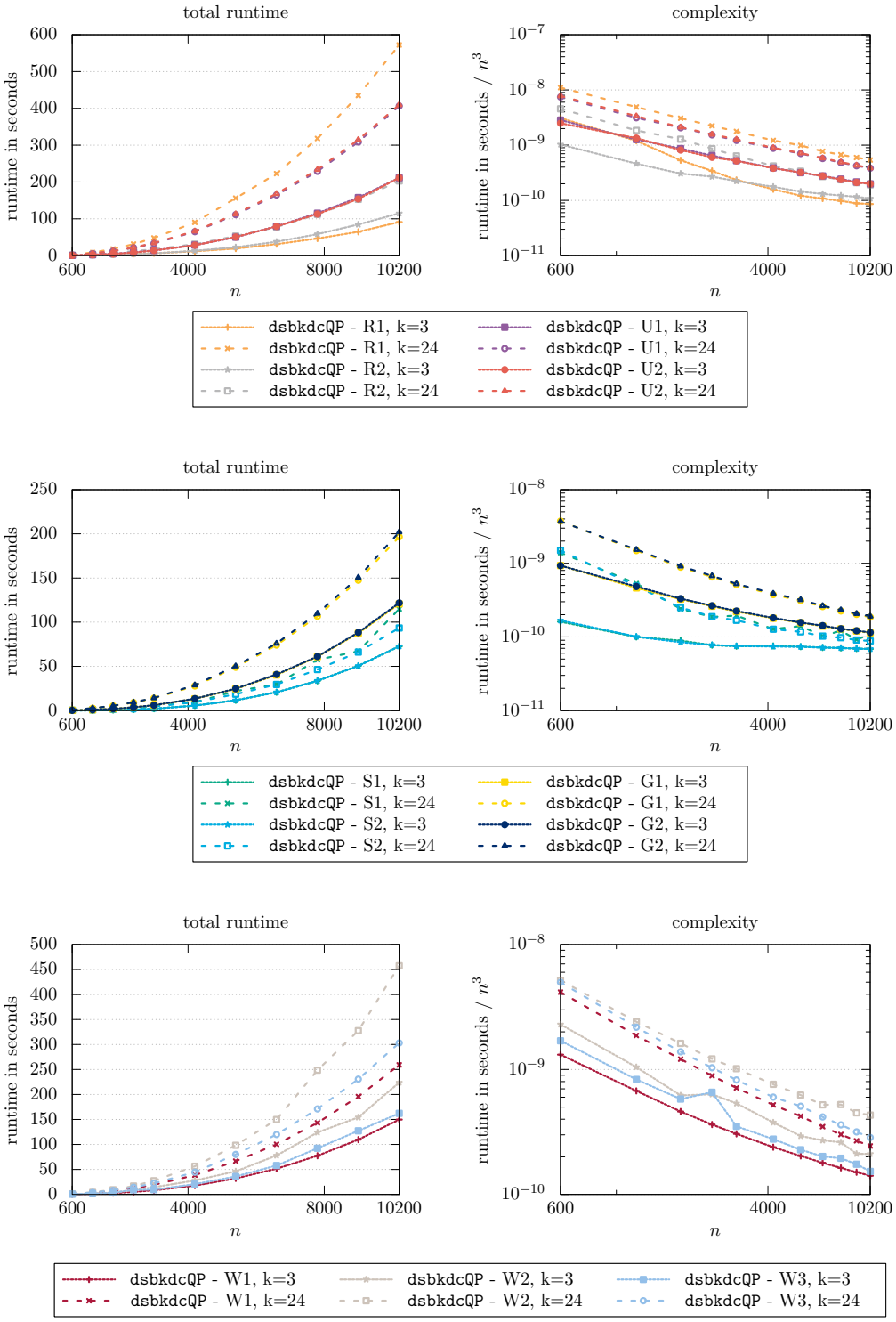


Figure A.11.: Runtime of `dsbkdcQP`.

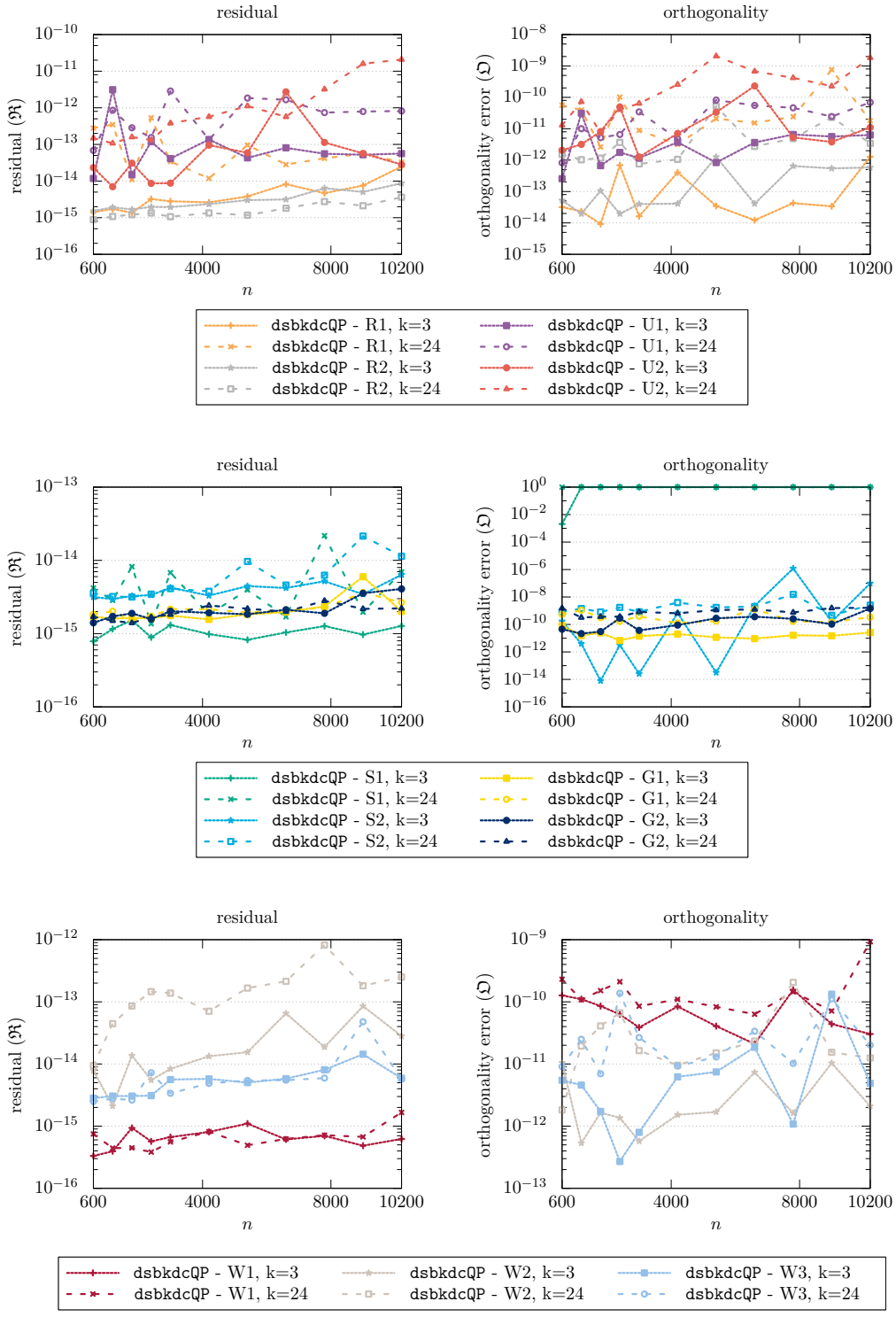


Figure A.12.: Accuracy of `dsbkdcQP`.

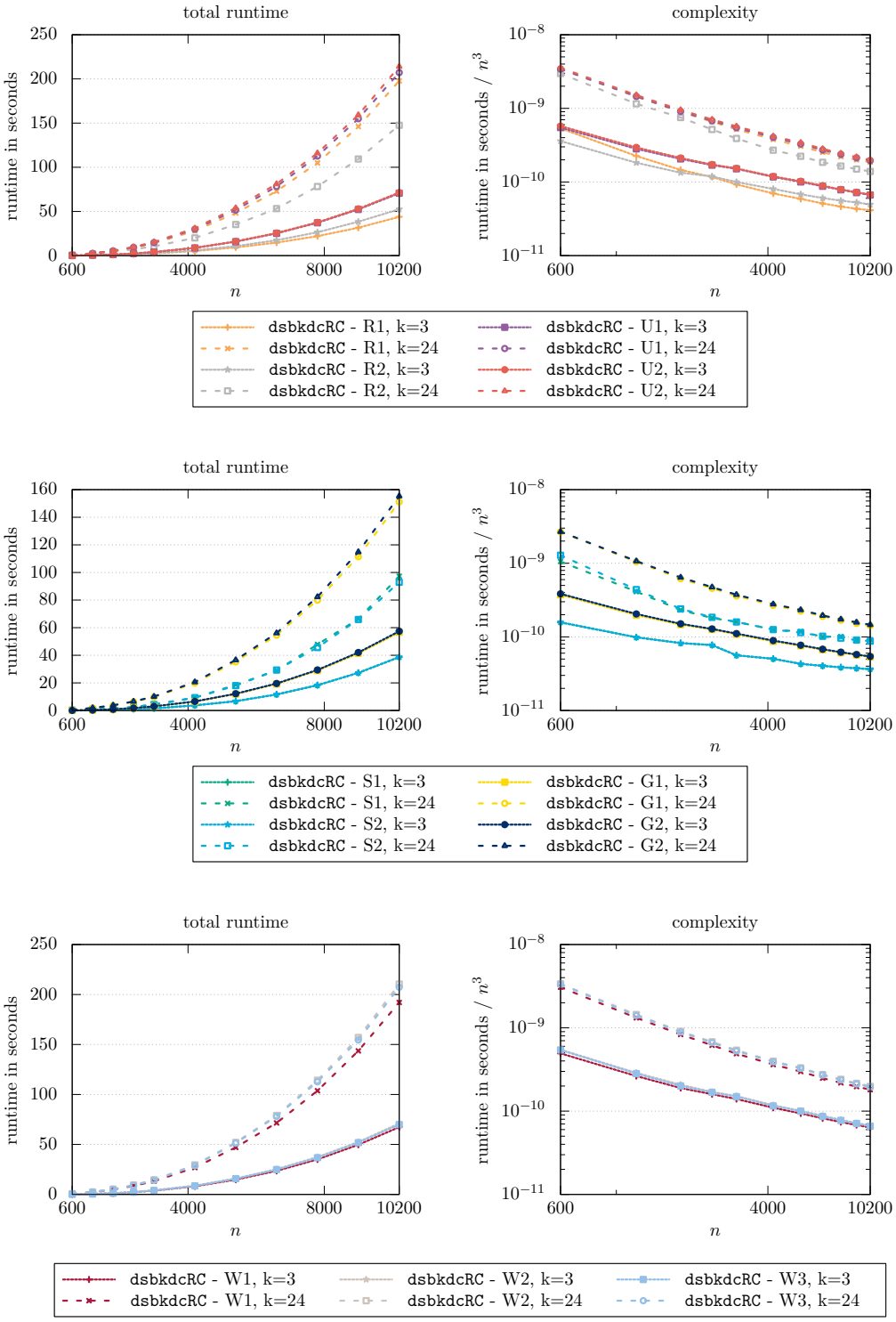


Figure A.13.: Runtime of `dsbkdcRC`.

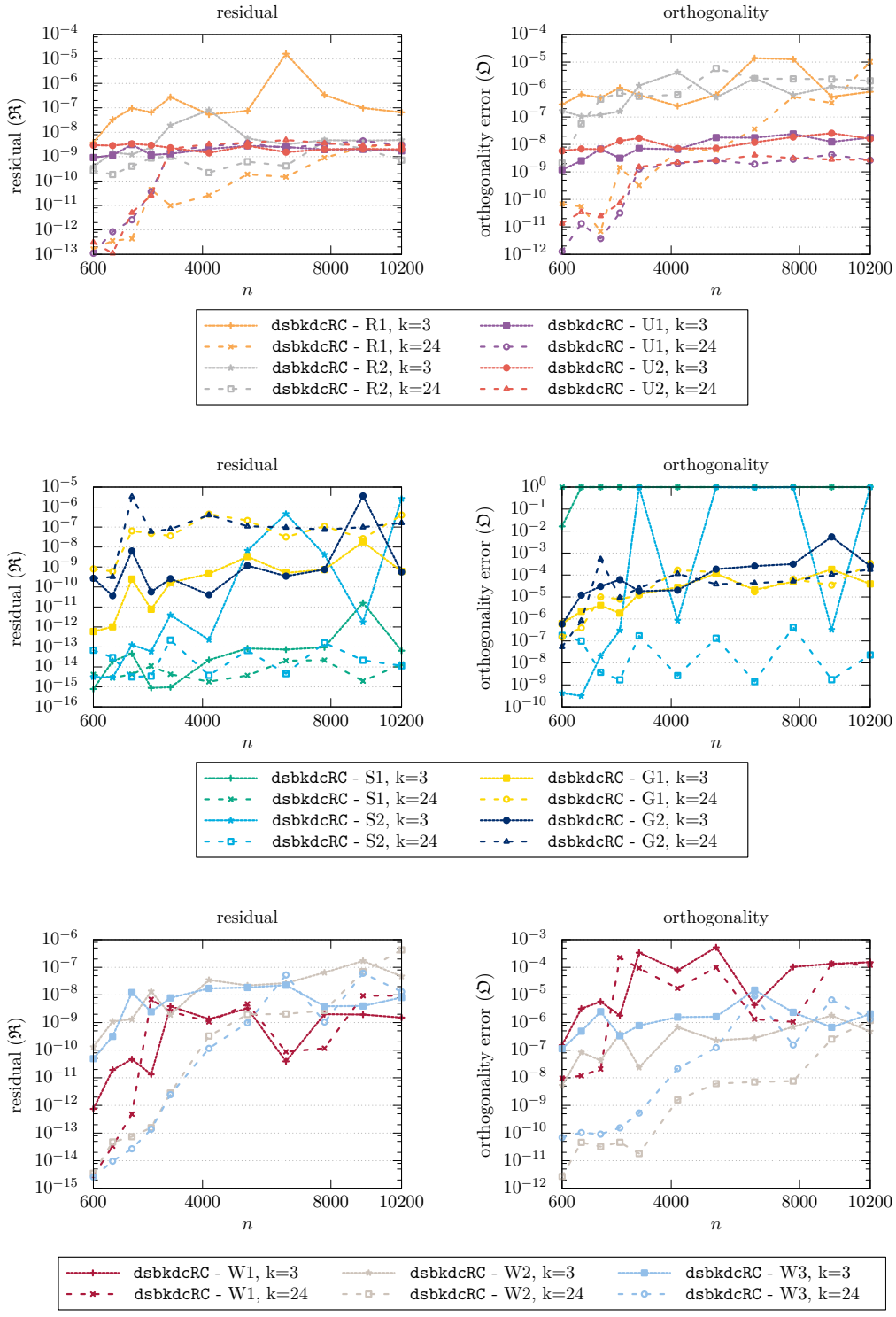


Figure A.14.: Accuracy of `dsbkdcRC`.

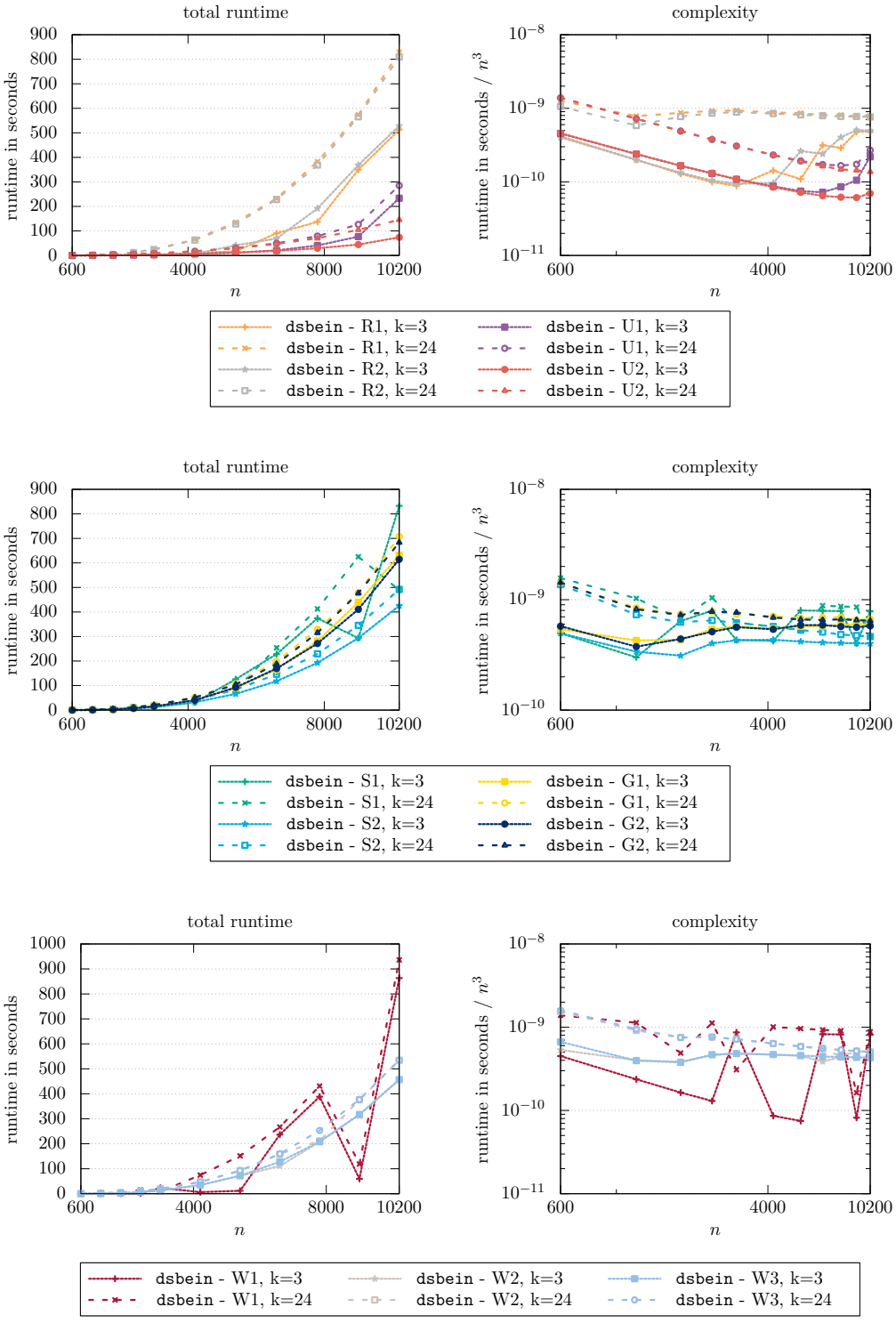


Figure A.15.: Runtime of `dsbein`.

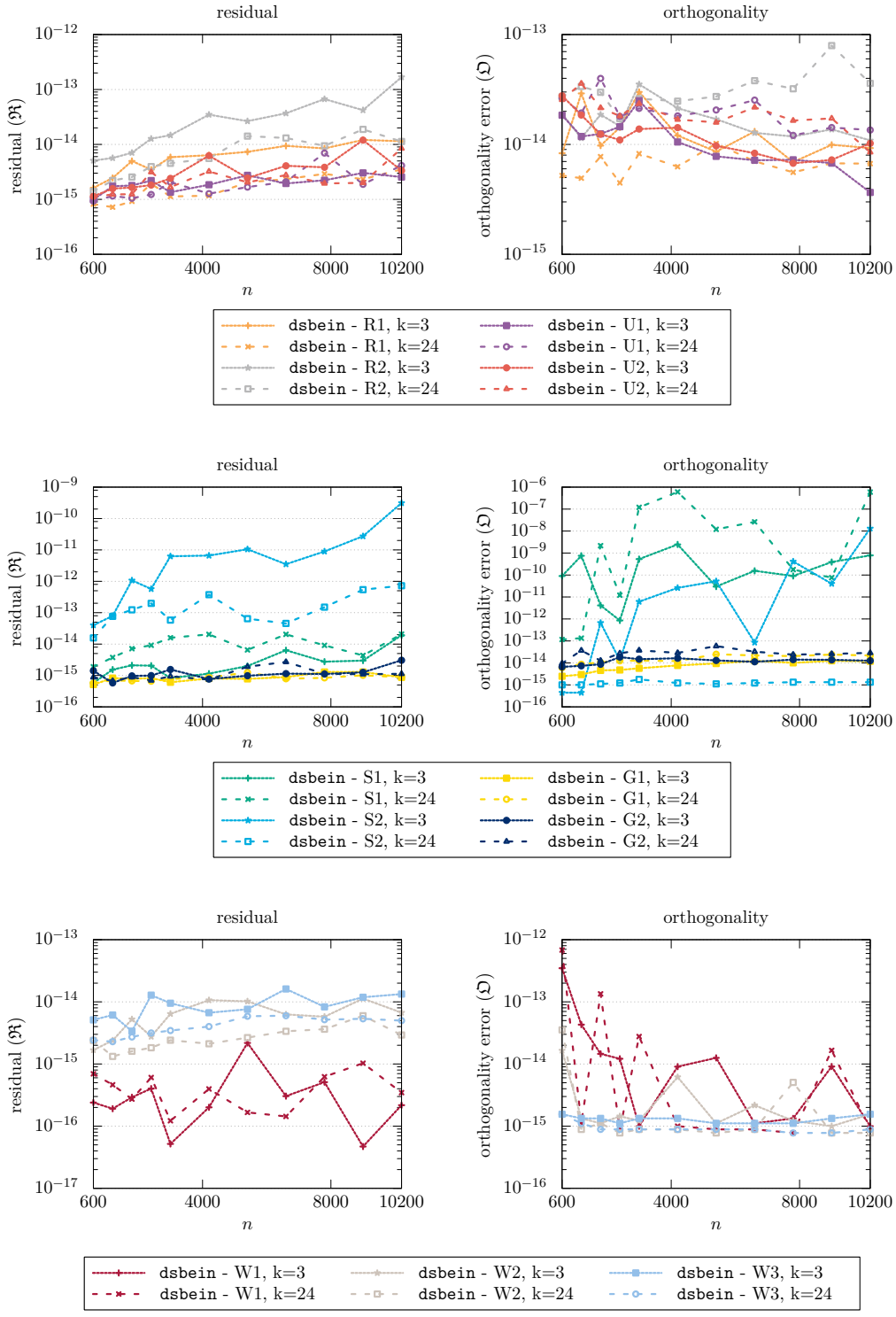


Figure A.16.: Accuracy of `dsbein`.

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects", *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012 037, 2009.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney *et al.*, *LAPACK Users' guide*. SIAM, 1999.

[3] P. Arbenz, "Computing eigenvalues of banded symmetric toeplitz matrices", *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 4, pp. 743–754, 1991. DOI: 10.1137/0912039.

[4] P. Arbenz, "Divide and conquer algorithms for the bandsymmetric eigenvalue problem", *Parallel computing*, vol. 18, no. 10, pp. 1105–1128, 1992. DOI: 10.1016/0167-8191(92)90059-G.

[5] P. Arbenz, W. Gander and G. H. Golub, "Restricted rank modification of the symmetric eigenvalue problem: Theoretical considerations", *Linear Algebra and its Applications*, vol. 104, pp. 75–95, 1988, ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(88)90309-6.

[6] G. Ballard, J. Demmel and N. Knight, "Communication avoiding successive band reduction", in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New Orleans, Louisiana, USA: ACM, 2012, pp. 35–44, ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145822.

[7] C. Beattie and D. W. Fox, "Schur complements and the weinstein-aronszajn theory for modified matrix eigenvalue problems", *Linear Algebra and its Applications*, vol. 108, pp. 37–61, 1988, ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(88)90178-4.

[8] P. Bientinesi, I. Dhillon and R. van de Geijn, "A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations", *SIAM Journal on Scientific Computing*, vol. 27, no. 1, pp. 43–66, 2005. DOI: 10.1137/030601107.

[9] C. Bischof, X. Sun and B. Lang, "Parallel tridiagonalization through two-step band reduction", in *Proceedings of IEEE Scalable High Performance Computing Conference*, May 1994, pp. 23–27. DOI: 10.1109/SHPCC.1994.296622.

[10] C. H. Bischof, B. Lang and X. Sun, "A framework for symmetric band reduction", *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 581–601, Dec. 2000, ISSN: 0098-3500. DOI: 10.1145/365723.365735.

[11] ——, "Algorithm 807: The sbr toolbox — software for successive band reduction", *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 602–616, Dec. 2000, ISSN: 0098-3500. DOI: 10.1145/365723.365736.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[13] R. F. Boisvert, R. Pozo and K. A. Remington. (1996). The matrix market exchange formats: Initial design, [Online]. Available: https://math.nist.gov/MatrixMarket/reports/MMformat.ps (visited on 14/07/2018).

[14] R. P. Brent, "An algorithm with guaranteed convergence for finding a zero of a function", *The Computer Journal*, vol. 14, no. 4, pp. 422–425, 1971. DOI: 10.1093/comjnl/14.4.422.

[15] J. R. Bunch, C. P. Nielsen and D. C. Sorensen, "Rank-one modification of the symmetric eigenproblem", *Numerische Mathematik*, vol. 31, no. 1, pp. 31–48, Mar. 1978, ISSN: 0945-3245. DOI: 10.1007/BF01396012.

[16] J. C. P. Bus and T. J. Dekker, "Two efficient algorithms with guaranteed convergence for finding a zero of a function", *ACM Trans. Math. Softw.*, vol. 1, no. 4, pp. 330–345, Dec. 1975, ISSN: 0098-3500. DOI: 10.1145/355656.355659.

[17] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem", *Numerische Mathematik*, vol. 36, no. 2, pp. 177–195, Jun. 1980, ISSN: 0945-3245. DOI: 10.1007/BF01396757.

[18] T. A. Davis and S. Rajamanickam, "Algorithm 8xx: Piro band, pipelined plane rotations for blocked band reduction", *Submitted to ACM Transactions on Mathematical Software, Available at*, 2010.

[19] J. Demmel, O. Marques, B. Parlett and C. Vömel, "Performance and accuracy of lapack's symmetric tridiagonal eigensolvers", *SIAM Journal on Scientific Computing*, vol. 30, no. 3, pp. 1508–1526, 2008. DOI: 10.1137/070688778.

[20] J. W. Demmel, "Applied numerical linear algebra", vol. 56, 1997.

[21] I. S. Dhillon and B. N. Parlett, "Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices", *Linear Algebra and its Applications*, vol. 387, pp. 1–28, 2004, ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2003.12.028.

[22] I. S. Dhillon, B. N. Parlett and C. Vömel, "The design and implementation of the mrrr algorithm", *ACM Trans. Math. Softw.*, vol. 32, no. 4, pp. 533–560, Dec. 2006, ISSN: 0098-3500. DOI: 10.1145/1186785.1186788.

[23] J. Dongarra and D. Sorensen, "A fully parallel algorithm for the symmetric eigenvalue problem", *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 2, s139–s154, 1987. DOI: 10.1137/0908018.

[24] J. F. Epperson, *An introduction to numerical methods and analysis*. John Wiley & Sons, 2013.

[25] J. G. F. Francis, "The qr transformation a unitary analogue to the lr transformation—part 1", *The Computer Journal*, vol. 4, no. 3, pp. 265–271, 1961. DOI: 10.1093/comjnl/4.3.265.

[26] W. N. Gansterer, J. Schneid and C. W. Ueberhuber, "A low-complexity divide-and-conquer method for computing eigenvalues and eigenvectors of symmetric band matrices", *BIT Numerical Mathematics*, vol. 41, no. 5, pp. 967–976, Dec. 2001, ISSN: 1572-9125. DOI: 10.1023/A:1021933127041.

[27] W. Gansterer, R. Ward, R. Muller and W. Goddard, "Computing approximate eigenpairs of symmetric block tridiagonal matrices", *SIAM Journal on Scientific Computing*, vol. 25, no. 1, pp. 65–85, 2003. DOI: 10.1137/S1064827501399432.

[28] W. N. Gansterer, R. C. Ward and R. P. Muller, "An extension of the divide-and-conquer method for a class of symmetric block-tridiagonal eigenproblems", *ACM Trans. Math. Softw.*, vol. 28, no. 1, pp. 45–58, Mar. 2002, ISSN: 0098-3500. DOI: 10.1145/513001.513004.

[29] G. Golub, "Some modified matrix eigenvalue problems", *SIAM Review*, vol. 15, no. 2, pp. 318–334, 1973. DOI: 10.1137/1015032.

[30] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[31] M. Gu and S. Eisenstat, "A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem", *SIAM Journal on Matrix Analysis and Applications*, vol. 15, no. 4, pp. 1266–1276, 1994. DOI: 10.1137/S089547989223924X.

[32] ——, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem", *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 172–191, 1995. DOI: 10.1137/S0895479892241287.

[33] A. Haidar, H. Ltaief and J. Dongarra, "Toward a high performance tile divide and conquer algorithm for the dense symmetric eigenvalue problem", *SIAM Journal on Scientific Computing*, vol. 34, no. 6, pp. C249–C274, 2012. DOI: 10.1137/110823699.

[34] A. Haidar, H. Ltaief and J. Dongarra, "Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels", in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: ACM, 2011, 8:1–8:11, ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063394.

[35] A. Haidar, S. Tomov, J. Dongarra, R. Solcà and T. Schulthess, "A novel hybrid cpu–gpu generalized eigensolver for electronic structure calculations based on fine-grained memory aware tasks", *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 196–209, 2014. DOI: 10.1177/1094342013502097.

[36] M. Hanke-Bourgeois, *Grundlagen der numerischen Mathematik und des wissenschaftlichen Rechnens*, 3. Springer, 2002, vol. 178.

[37] (2017). Inverse quadratic interpolation, [Online]. Available: https://en.wikipedia.org/wiki/Inverse_quadratic_interpolation (visited on 18/08/2018).

[38] E. Jessup and I. Ipsen, "Improving the accuracy of inverse iteration", *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 550–572, 1992. DOI: 10.1137/0913031.

[39] H. Ltaief, P. Luszczek and J. Dongarra, "Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures", in *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)(IPDPS)*, vol. 00, May 2011, pp. 944–955. DOI: 10.1109/IPDPS.2011.91.

[40] O. A. Marques, C. Vömel, J. W. Demmel and B. N. Parlett, "Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers", *ACM Trans. Math. Softw.*, vol. 35, no. 1, 8:1–8:13, Jul. 2008, ISSN: 0098-3500. DOI: 10.1145/1377603.1377611.

[41] M. Moldaschl and W. N. Gansterer, "Comparison of eigensolvers for symmetric band matrices", *Science of computer programming*, vol. 90, pp. 55–66, 2014.

[42] M. L. Overton, *Numerical computing with IEEE floating point arithmetic*. Siam, 2001, vol. 76.

[43] G. Pichon, A. Haidar, M. Faverge and J. Kurzak, "Divide and conquer symmetric tridiagonal eigensolver for multicore architectures", in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 51–60. DOI: 10.1109/IPDPS.2015.51.

[44] S. Rajamanickam and T. A. Davis, "Blocked band reduction for symmetric and unsymmetric matrices", *Submitted to ACM Trans. Math. Softw*, 2009.

[45] ——, "User guide for piro_band: General band reduction using blocked givens rotations", Tech. Rep., 2009. [Online]. Available: https://www.cise.ufl.edu/~srajaman/PIRO_BAND/Doc/UserGuide.pdf (visited on 14/07/2018).

[46] H. R. Schwarz, "Tridiagonalization of a symmetric band matrix", *Numerische Mathematik*, vol. 12, no. 4, pp. 231–241, Nov. 1968, ISSN: 0945-3245. DOI: 10.1007/BF02162505.

[47] D. Sorensen and P. Tang, "On the orthogonality of eigenvectors computed by divide-and-conquer techniques", *SIAM Journal on Numerical Analysis*, vol. 28, no. 6, pp. 1752–1775, 1991. DOI: 10.1137/0728087.

[48] G. W. Stewart, *Matrix Algorithms Volume II: Eigensystems*. Siam, 2001, vol. 2.

[49] (2018). The gcc quad-procision math library, [Online]. Available: https://gcc.gnu.org/onlinedocs/libquadmath.pdf (visited on 15/07/2018).

[50] F. Tisseur and J. Dongarra, "A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures", *SIAM Journal on Scientific Computing*, vol. 20, no. 6, pp. 2223–2236, 1999. DOI: 10.1137/S1064827598336951.

[51] C. Vömel, S. Tomov and J. Dongarra, "Divide and conquer on hybrid gpu-accelerated multicore systems", *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C70–C82, 2012. DOI: 10.1137/100806783.