ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Design and Implementation of Optimized Eigendecomposition Algorithms For Symmetric Block-tridiagonal and Symmetric Broad-arrowhead Matrices and their Applications in Ring-polymer Instanton Theory

Bachelor's Thesis

Marcel Ferrari

January 1., 2024

Advisors: Prof. Dr. Jeremy Richardson, Mr. Imaad Ansari

Theoretical Molecular Quantum Dynamics Group,
Department of Chemistry D-CHAB, ETH Zürich

Abstract

Eigendecomposition is a fundamental tool in numerous scientific and engineering disciplines, playing a critical role in a large number of problems. Efficient numerical methods for eigendecomposition are of crucial importance due to the complexity and computational cost of this process, especially for large-scale matrices. This thesis discusses the theoretical aspects of optimized eigendecomposition algorithms for symmetric block-tridiagonal and symmetric broad arrowhead matrices, as well as documenting the empirical results obtained by developing a robust and high-performance implementation of these methods. Additionally, practical applications of these techniques in numerical simulations of ring-polymer instanton theory are discussed and evaluated with a set of real-world test problems. The performance and accuracy results are very promising, showcasing up to 20× speedup, for exact solutions, compared to the state-of-the-art eigensolver routines implemented in the Intel oneAPI MKL library. Moreover, up to 30× faster runtimes were recorded for approximated solutions with a deviation of less than 10% from the exact results, demonstrating the validity of the proposed methods.

Contents

Chapter 1

Introduction

1.1 Introduction

Eigendecomposition, also known as diagonalisation, is a fundamental tool in various scientific and engineering disciplines, playing a critical role in a large number of problems. Its applications range from machine learning and data science tasks, such as principal component analysis (PCA) [1], to studying the physical properties of harmonic oscillator chains [2], to solving radial Schrödinger's equation in quantum mechanics [3]. Efficient numerical methods for eigendecomposition are crucial due to the complexity and computational cost of this process, especially for large-scale matrices. Although very efficient eigensolvers for generic dense matrices are already implemented in numerical libraries such as LAPACK [4] or Intel oneAPI MKL [5], many physical systems can be described with matrices that exhibit specific structural traits, which can be exploited to significantly reduce computational expenses.

This work discusses the theory behind various optimized eigendecomposition algorithms for certain classes of symmetric real matrices, as well as documenting the empirical results obtained by developing a robust and high-performance implementation of the discussed methods. Additionally, practical applications of these techniques in numerical simulations of ring-polymer instanton theory, as described in [6], are discussed and evaluated with a set of real-world test problems.

Specifically, this research focuses on efficiently solving the eigenproblem for two specific matrix classes: block-tridiagonal/banded and broad arrowhead matrices. Block-tridiagonal matrices can be diagonalised efficiently by using the Block divide and conquer algorithm (BD&C), which operates on the off-diagonal blocks of the matrix directly by employing a binary tree divide and conquer strategy. The eigendecomposition of broad arrowhead matrices is instead obtained from an intermediate decomposition known as arrowhead

factorization (AF), which can then be exploited to compute the eigenvalues and eigenvectors of the original matrix. Two additional methods, used to diagonalise diagonal plus rank one (DPR1) and simple arrowhead matrices, are also discussed as they are required internally by both the BD&C algorithm and the AF eigensolver.

The thesis is structured into six main chapters: chapters 2-5 are each dedicated to the theoretical discussion of a different eigendecomposition algorithm. More precisely, chapter 2 describes the BD&C algorithm for block-tridiagonal matrices, chapter 3 details an efficient method for DPR1 matrices, chapter 4 explains how the arrowhead factorization eigensolver works and chapter 5 illustrates an optimized approach for simple arrowhead matrices. Chapter 6 showcases the results of numerical experiments and benchmarks used to evaluate the performance and accuracy of the BD&C and AF methods. Finally, chapter 7 is dedicated to final remarks and conclusions.

Chapter 2

Eigendecomposition of symmetric banded and block-tridiagonal matrices

2.1 Introduction

In the realm of computational science, we are often confronted with the problem of computing the eigenvalues and eigenvectors of a matrix, especially in the context of large-scale simulations. Banded and block-tridiagonal matrices, in particular, often emerge as the natural mathematical representation of many physical systems. This is the case, for examples, of problems related to the path-integral formulation of quantum mechanics [7]. More precisely, numerical simulations based on instanton theory, as described in [6], very often rely on an iterative optimization process that involves the diagonalisation of block-tridiagonal matrices. Given the expensive nature of this operation, it is essential to develop optimized strategies that allow to reduce the computational cost of eigendecomposition.

This chapter focuses on the theoretical aspects required for the implementation of an optimized algorithm used to compute the full eigendecomposition of banded and block-tridiagonal matrices.

2.2 State-of-the-art eigensolver algorithms

The algorithms currently implemented in industry-leading numerical libraries, such as Intel oneAPI MKL [5] and LAPACK [4], rely on a three step eigendecomposition process: first, the matrix is reduced to tridiagonal form using a series of orthogonal transformations [8]. Then, its eigenvalues and eigenvectors are calculated with one of three available algorithms for the tridiagonal eigenproblem: the QR algorithm [9], Cuppen's divide and conquer (DC) algorithm [10] [11] or the Multiple Relatively Robust (MRR) algorithm [12]. Finally, the eigenvectors of the tridiagonal matrix are backtransformed

to reconstruct the final solution. In general, the cost of this procedure is dominated by the last backpropagation step, which has a complexity of $\mathcal{O}(n^3)$ for dense matrices and $\mathcal{O}(n^2 b)$ for banded matrices with semi-bandwidth b [13]. LAPACK offers two main driver functions that implement this process: the DSYEVR routine for dense matrices and the DSBEVD routine for banded matrices. This work, however, considers an alternative approach, known as the Block Divide and Conquer algorithm (BD&C), that is a direct generalisation of Cuppen's DC algorithm for the tridiagonal problem. With this strategy, it is possible to operate directly on tri-block diagonal matrices without the need of reduction to tridiagonal form. Figure 2.1 shows a diagram of the different eigendecomposition strategies.

The work published in [13] discusses an in-depth comparison between the previously mentioned methods, including the BD&C algorithm. Their findings highlight the strength and limitations of both tridiagonalisation and block operation based techniques.

In the upcoming sections, a detailed description of the BD&C algorithm is given. The description of this algorithm has appeared in the literature since the early late 90's/early 2000's and it is difficult to attribute it to a single author. The work published by Gansterer in [14] is one of the earliest papers discussing the topic. However, it is important to note that many ideas, both theoretical and practical, proposed in this work were developed independently.

2.3 Problem description

Consider a symmetric block tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ of the following form:

$$T := \begin{bmatrix} A_1 & B_1^T & & & \\ B_1 & A_2 & B_2^T & & \\ & B_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{p-1}^T \\ & & & B_{p-1} & A_p \end{bmatrix} \tag{2.1}$$

where A_i, B_i are dense matrices of size $f \times f$ and p is the number of diagonal blocks. If T is banded, then the blocks B_i are upper triangular matrices.

The goal is to compute the eigenvalues and eigenvectors of T, that is, find $n \times n$ matrices Λ and Q such that:

$$T = Q \Lambda Q^\top \tag{2.2}$$

Moreover, since T is a symmetric real matrix, the eigenvectors can always be chosen so that Q is orthogonal. This additional constraint guarantees that the inverse Q^{-1} must never be explicitly computed as it is equal to $Q^\top$.

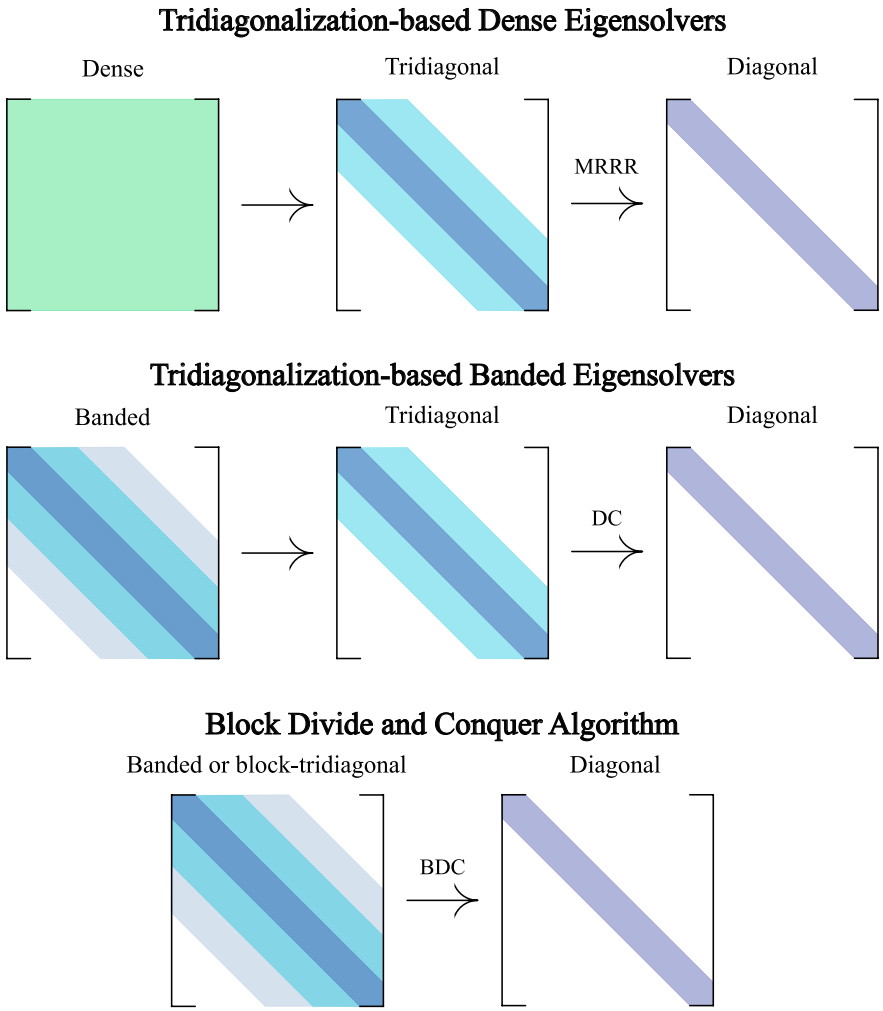

Figure 2.1: Diagram of the different methods for diagonalisation. After tridiagonalisation, dense solver DSYEVR uses the MRRR algorithm with complexity $O(n^2)$, while the banded solver DSBEVD implements the divide and conquer algorithm with complexity $O(n^3)$. The BD&C algorithm has complexity $\mathcal{O}(n^3)$.

2.4 Overview of the algorithm

The BD&C algorithm computes the eigendecomposition of a symmetric tri-block diagonal matrix in a two-phase strategy of division and conquering.

1. The division step revolves around splitting the matrix T into two smaller similar sub-problems, which are then diagonalised either by recursively applying the BD&C algorithm again, or by invoking a dense eigensolver routine.

2. Once the eigendecomposition of the sub-problems is known, the algorithm transitions to its next step: conquering. During this phase, the

partial solutions are merged in a tree-like order to generate the final eigendecomposition of the matrix.

At the most fundamental level, BD&C is a classic binary tree divide and conquer algorithm. This is a remarkable characteristic as by breaking the matrix down into smaller and more manageable blocks, it is possible to significantly reduce the work-depth of the algorithm and take advantage of the many cores offered by modern CPUs. This trait emerges as a result of the sparsity structure of the intermediate results, allowing for a hierarchical approach to problem-solving similar to much simpler problems, such as, for example, sorting a list.

In the subsequent sections, we will discuss the mathematical details of the two steps of the BD&C algorithm.

2.5 Computation of the leaf problems

For simplicity, we first consider the minimal case that arises when diagonalising block-tridiagonal matrices such as T.

Consider the following matrix:

$$T := \begin{bmatrix} A_1 & B^T \\ B & A_2 \end{bmatrix} \tag{2.3}$$

Again we focus on computing the eigendecomposition of T.

2.5.1 The division step

The first step is to express T as a low rank modification of a block diagonal matrix:

$$T = \tilde{T} + WW^\top \tag{2.4}$$

Where $\tilde{T}$ is a block diagonal matrix and $WW^\top$ is a rank-f modification. The trick here lays in computing the singular value decomposition $B = U\Sigma V^\top$. This allows us to express T as:

$$\begin{aligned} T &= \begin{bmatrix} A_1 - V\Sigma V^\top & 0 \\ 0 & A_2 - U\Sigma U^\top \end{bmatrix} + \begin{bmatrix} V\Sigma V^\top & V\Sigma U^\top \\ V\Sigma U^\top & U\Sigma U^\top \end{bmatrix} = \\ &= \begin{bmatrix} A_1 - V\Sigma V^\top & 0 \\ 0 & A_2 - U\Sigma U^\top \end{bmatrix} + \begin{bmatrix} V\Sigma^{\frac{1}{2}} \\ U\Sigma^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} \Sigma^{\frac{1}{2}} V^\top & \Sigma^{\frac{1}{2}} U^\top \end{bmatrix} = \tilde{T} + WW^\top \end{aligned} \tag{2.5}$$

We diagonalise $\tilde{T}$ by operating on the diagonal blocks with a dense eigensolver:

$$\begin{aligned} \tilde{A}_1 &= \tilde{Q}_1 \tilde{\Lambda}_1 \tilde{Q}_1^\top \\ \tilde{A}_2 &= \tilde{Q}_2 \tilde{\Lambda}_2 \tilde{Q}_2^\top \\ \tilde{T} = Q_0 \Lambda_0 Q_0^\top &= \begin{bmatrix} \tilde{Q}_1 & \\ & \tilde{Q}_2 \end{bmatrix} \begin{bmatrix} \tilde{\Lambda}_1 & \\ & \tilde{\Lambda}_2 \end{bmatrix} \begin{bmatrix} \tilde{Q}_1^\top & \\ & \tilde{Q}_2^\top \end{bmatrix} \end{aligned} \tag{2.6}$$

We now have a partial solution of the form:

$$T = \tilde{T} + WW^\top = Q_0 \Lambda_0 Q_0^\top + WW^\top \tag{2.7}$$

2.5.2 The synthesis step

Given the partial solution 2.7, we now seek to compute the full solution $T = Q\Lambda Q^\top$. From 2.7 we obtain:

$$\begin{aligned} T &= \tilde{T} + WW^\top = Q_0 \Lambda_0 Q_0^\top + WW^\top = \\ &= Q_0(\Lambda_0 + Q_0^\top WW^\top Q_0) Q_0^\top = Q_0(\Lambda_0 + ZZ^\top) Q_0^\top \end{aligned} \tag{2.8}$$

The synthesis matrix S is defined as $S := \Lambda_0 + ZZ^\top$. Given that Q_0 is orthogonal, T and S are similar matrices, sharing the same eigenvalues, and as such we focus on solving the eigenproblem for S.

Computing the Eigenvalues and Eigenvectors of the Synthesis Matrix

We can express S as:

$$S = \Lambda_0 + ZZ^\top = \Lambda_0 + \sum_{i=1}^{f} z_i z_i^\top \tag{2.9}$$

With z_i representing the i-th column of Z. This paves the way for an iterative approach where the rank f modification is represented as a series of f rank-1 updates. At each iteration, a diagonal plus rank 1 (DPR1) eigenproblem is solved and the eigenvalues and eigenvectors are updated accordingly. The solution of each DPR1 eigenproblem can computed efficiently in $O(n^2)$, as will be explained in the upcoming chapters. This yields the following recursive definition:

$$\begin{cases} \Lambda_i, Q_i = \text{DPR1\_eigh}(\Lambda_{i-1}, \tilde{z}_i) & i \geq 1 \\ \tilde{z}_i = \left(\prod_{j=1}^{i-1} Q_j \right)^\top z_i & i \geq 2 \\ \tilde{z}_1 = z_1 \end{cases} \tag{2.10}$$

The final solution $T = Q\Lambda Q^\top$ is given by:

$$T = Q\Lambda Q^\top$$
$$Q = \prod_{j=0}^{f} Q_j,\ \Lambda = \Lambda_f$$

In practice, Q is computed by accumulating the product $\prod_{j=0}^{f} Q_j^\top$. Algorithm 1 showcases the pseudocode for implicit construction of S and solution of the low rank update eigenproblem in $\mathcal{O}(n^3)$.

Algorithm 1: Compute eigenvalues of S

Input : Λ_0, Q_0, W
Output: Λ, Q
$\Lambda := \Lambda_0$
$Q := Q_0$
for $i := 0$ **to** *f-1* **do**
  $\tilde{z}_i := Q^\top W\,.\,\mathrm{col}(i)$
  `// Update Λ and Q.`
  $\Lambda_{\text{tmp}}, Q_{\text{tmp}} := \mathrm{DPR1\_eigh}(\Lambda, \tilde{z}_i)$
  `// Update Λ`
  $\Lambda = \Lambda_{\text{tmp}}$
  `// Accumulate Q`
  $Q := Q \cdot Q_{\text{tmp}}$
end
return Λ, Q

The most expensive operation of the whole algorithm is the accumulation of the Q_i matrices to form the eigenvector matrix Q. Figure 2.2 shows a diagram of the full computation for this merge operation.

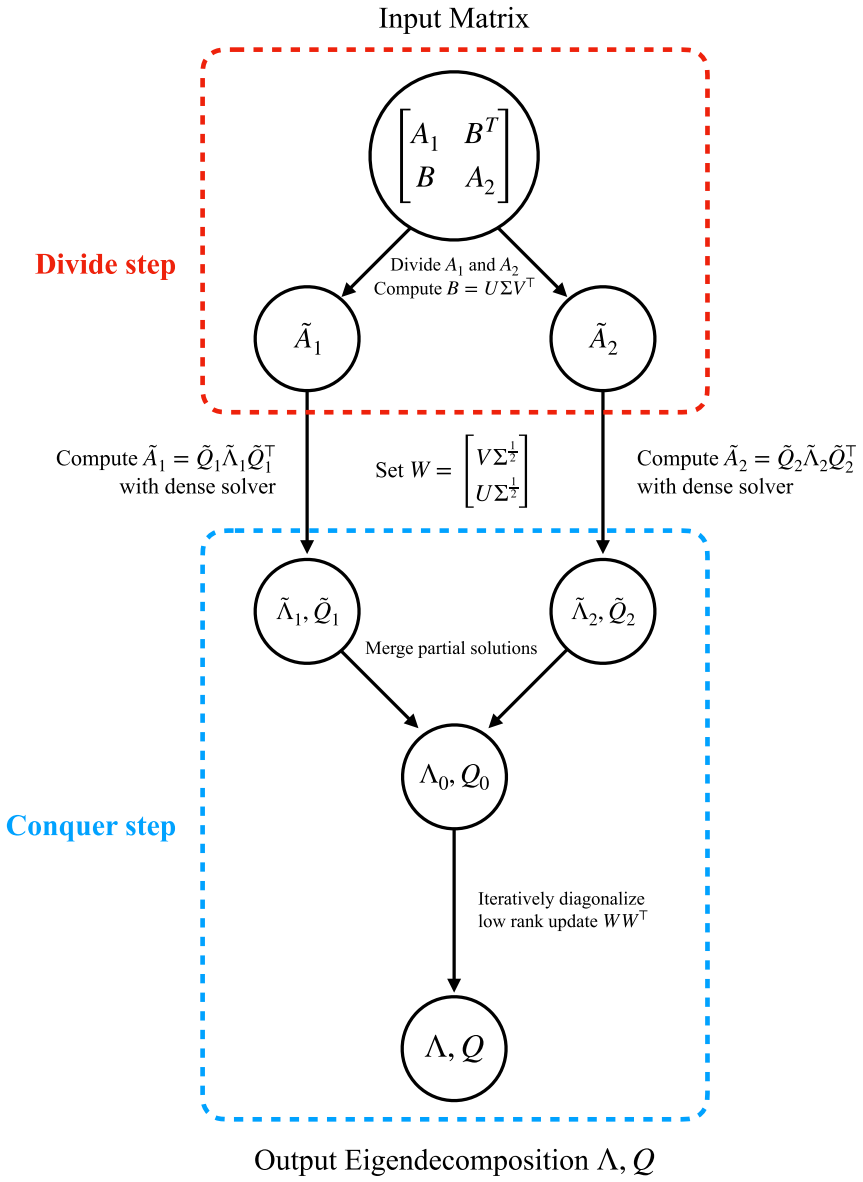

Figure 2.2: Graphical representation of the computational graph for the merge operation of two partial solutions.

2.6 Increasing the problem size

Let us now consider the general case as defined in 2.1. The same principles apply and we start by splitting the matrix T into a series of rank f updates as follows:

$$
\begin{aligned}
T = &\begin{bmatrix} A_1 - V_1\Sigma_1 V_1^\top & & & \\ & A_2 - U_1\Sigma_1 U_1^\top - V_2\Sigma_2 V_2^\top & & \\ & & \ddots & \\ & & & A_p - U_p\Sigma_p U_p^\top \end{bmatrix} + \\
&+ \begin{bmatrix} V_1\Sigma_1 V_1^\top & B_1^\top & 0 & \cdots \\ B_1 & U_1\Sigma_1 U_1^\top & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 0 & V_2\Sigma_2 V_2^\top & B_2\top & \cdots \\ 0 & B_2 & U_2\Sigma_2 V_2^\top & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} + \ldots = \\
&= \begin{bmatrix} \tilde{A}_1 & & & & \\ & \tilde{A}_2 & & & \\ & & \tilde{A}_3 & & \\ & & & \ddots & \\ & & & & \tilde{A}_p \end{bmatrix} + \begin{bmatrix} V_1\Sigma_1^{\frac{1}{2}} \\ U_1\Sigma_1^{\frac{1}{2}} \\ 0 \\ 0 \\ \vdots \end{bmatrix} \begin{bmatrix} V_1\Sigma_1^{\frac{1}{2}} & U_1\Sigma_1^{\frac{1}{2}} & 0 & 0\ldots \end{bmatrix} + \\
&+ \begin{bmatrix} 0 \\ V_2\Sigma_2^{\frac{1}{2}} \\ U_2\Sigma_2^{\frac{1}{2}} \\ 0 \\ \vdots \end{bmatrix} \begin{bmatrix} 0 & V_2\Sigma_2^{\frac{1}{2}} & U_2\Sigma_2^{\frac{1}{2}} & 0 & \ldots \end{bmatrix} + \ldots = \\
&= \tilde{T} + \sum_{i=1}^{p-1} W_i W_i^\top = \tilde{T} + \sum_{i=1}^{p-1}\sum_{j=1}^{f} z_{i,j} z_{i,j}^\top
\end{aligned}
\tag{2.11}
$$

This process can be rewritten equivalently as:

$$
\begin{cases} \tilde{A}_i = A_i - U_{i-1}\Sigma_{i-1}U_{i-1}^\top - V_i\Sigma_i V_i^\top \quad \text{for } 2 \leq i \leq p-1 \\ \tilde{A}_1 = A_1 - V_1\Sigma V_1^\top \\ \tilde{A}_p = A_p - U_{p-1}\Sigma_{p-1}U_{p-1}^\top \end{cases}
\tag{2.12}
$$

The synthesis step does not change significantly, however in this case there are $p-1$ synthesis matrices $S_i = \Lambda_{i,0} + W_i W_i^\top$ of varying size. Given the sparsity structure introduced by the 0-padding present in the W_i matrices from Eq. 2.11, it is possible to merge the partial solution in parallel using a binary tree strategy to significantly reduce the work-depth of the algorithm. Figure 2.3 displays am example diagram showcasing the computational tree for a block-tridiagonal matrix with 8 blocks of size 5×5.

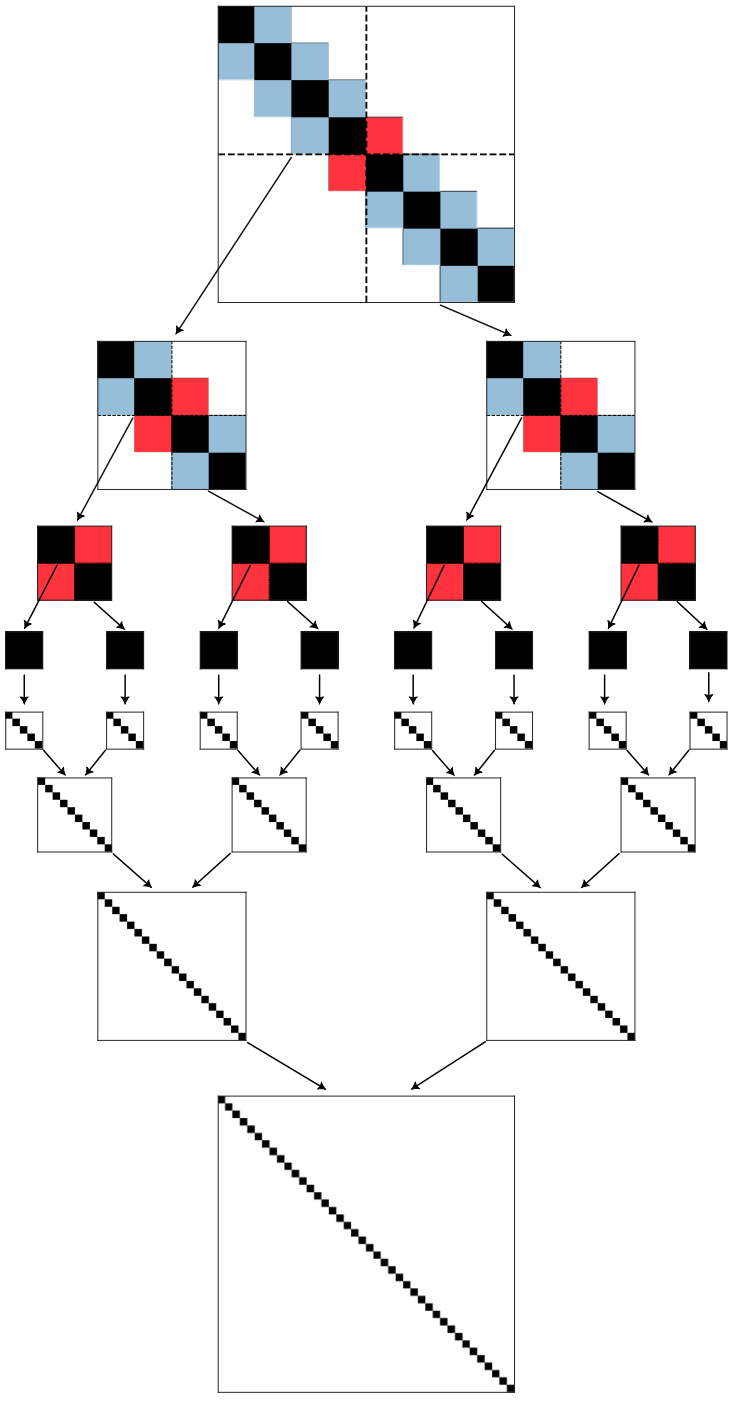

Figure 2.3: Computational graph of the BD&C algorithm for a block-tridiagonal matrix with 8 blocks of size 5×5. The gray blocks represent off-diagonal blocks of smaller subproblems. The black dashed lines show the splitting location for the division step. The red off-diagonal blocks represent the low rank update applied to the two partial solution during the merge operation.

2.7 Computing approximate eigenpairs

In many cases, full accuracy when computing the eigendecomposition of a matrix is not required and it is sufficient to guarantee orthogonality of the eigenvectors. This is specifically true, for example, when dealing with iterative optimization algorithms that are inherently influenced by a certain level of heuristics and approximations. One of the major advantages of the BD&C algorithm over tridiagonalisation-based solvers is that it is possible to sacrifice a certain degree of accuracy in exchange for significant performance improvements. Recall that the SVD decomposition of a matrix yields its best low rank approximation [15]. Consequently, the first r columns of the W_i matrices of Eq. 2.11 already implicitly represent the best rank-r approximation of their respective off-diagonal blocks B_i. Given that the cost of merging two partial solutions is determined virtually exclusively by the number of DPR1 updates computed, approximating each B_i block with the first r columns of its W_i matrix can lead to significant performance improvements, especially when r is chosen to be much smaller than f. Moreover, given that merging two partial solutions of size $n \times n$ returns a matrix of quadruple size $2n \times 2n$, it is easy to see that the computational cost of the merge operations gets progressively more expensive for nodes closer to the root of the tree. We can leverage this fact to conditionally decide when to approximate the W_i updates with lower rank matrices: by ensuring that only merge operations on large matrices are approximated, it is possible to maximize performance while minimizing the error introduced by the approximation. The performance-accuracy of the tradeoff is thus controlled by two parameters: the rank $r < f$ for the low rank approximation of the B_i matrices, and the threshold τ used to determine the minimum problem size where approximation should be applied.

Chapter 3

Eigendecomposition of DPR1 matrices

3.1 Introduction

This chapter focuses on the stable and efficient eigendecomposition of diagonal plus rank one (DPR1) matrices. These special matrices are of particular interest as they appear in the "conquer" step the BD&C algorithm described in the previous chapter. The accuracy of the BD&C strategy depends on the precision in solving the DPR1 eigenproblem. This is because, when merging two partial solutions, the eigendecomposition of the synthesis matrix is calculated by diagonalising a series of rank-one updates of a diagonal matrix. Inaccuracies in this step can propagate errors to the final solution and drastically impact both the accuracy of the eigenvalues, as well as the orthogonality of the eigenvectors. This chapter outlines the DPR1 eigenproblem and details the methods implemented to achieve accurate solutions.

3.2 Problem description

The eigenproblem for DPR1 matrices consists in computing the eigendecomposition of an $n \times n$ symmetric real matrix of the form:

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

where D is a diagonal matrix, ρ is a non-zero scalar and z is a vector.

Historically, this has been a notably difficult task to solve accurately, mainly due to numerical instability and orthogonality issues that arise when the eigenvalues are not computed with sufficient accuracy [16]. LAPACK implements the ?LAED family of routines to solve this problem for the tridiagonal divide and conquer algorithm [17]. However, given that these are internal computational routines with a very specific purpose, their use cases are limited and it was impossible to repurpose them for the BD&C algorithm.

Thus, a more recent approach to solving this problem was chosen and implemented for this work. This algorithm was proposed by Jakovčević Stor, Slapničar and Barlow in 2013 [16] and offers many attractive qualities such as better numerical accuracy than LAPACK's ?LAED routines, low $\mathrm{O}(n^2)$ complexity and a high degree of parallelism, while still being simpler than the alternatives [16]. At the time of writing, only an early development Julia implementation of this algorithm exists, authored by the original designers of the algorithm. As part of this work, a high-performance C++ implementation of this algorithm was written.

The following sections are dedicated to discussing the most important concepts and equations needed for a stable and efficient implementation of this algorithm. While theoretical discussions on accuracy and error analysis are omitted, as they are thoroughly covered in [16], the focus is set on the details required for a practical implementation of the algorithm.

3.3 Solving the eigenproblem for irreducible DPR1 matrices

We first discuss the diagonalisation of irreducible DPR1 matrices. Further explanations on the topics of the general DPR1 eigenproblem, as well as the deflation process will be provided in the upcoming sections.

Without loss of generality, consider the eigenproblem generated by the irreducible matrix

$$A = D + \rho z z^\top \tag{3.1}$$

where

$$D = \mathrm{diag}(d_1,\ d_2,\ ...,\ d_n) \quad z = [\ \xi_1,\ \xi_2, ...,\ \xi_n\ ]^\top \tag{3.2}$$

such that:

- $\rho > 0$
- $\xi_i \neq 0\ \forall i$
- $i \neq j\ \Rightarrow\ d_i \neq d_j\ \forall i, j$ (the elements of D are unique)
- $i < j\ \Rightarrow\ d_i > d_j\ \forall i, j$ (D is sorted decreasingly)

Given the special structure of this matrix, the eigenvalues can be computed as the roots of the secular function according to Theorem 3.1.

Theorem 3.1 *The eigenvalues of an irreducible DPR1 problem are given by the zeros of the secular function:*

$$f(\lambda) = 1 + \rho \sum_{i=1}^{n} \frac{\xi_i^2}{d_i - \lambda} \tag{3.3}$$

Proof The following proof was adapted from [11]. The eigenvalues of the DPR1 matrix are the roots of its characteristic polynomial:

$$\det(A - \lambda I) = \det(D + \rho z z^\top - \lambda I) = 0$$

Notice that $D - \lambda I$ is invertible as λ cannot be an eigenvalue of D, otherwise there would be a zero entry in z making the problem not irreducible as described in Eq. 3.2.

Thus, we can rewrite the left side of the above equation as:

$$\begin{aligned}
&\det(D + \rho z z^T - \lambda I) = \\
&= \det((D - \lambda I)(I + \rho(D - \lambda)^{-1} z z^T)) = \\
&= \det(D - \lambda I) \det(I + \rho(D - \lambda)^{-1} z z^T)
\end{aligned}$$

Using again the fact that $D - \lambda I$ is invertible, it is clear that $\det(D - \lambda I)$ can never be zero. Thus we obtain the following equation:

$$\det(I + \rho(D - \lambda)^{-1} z z^T) = 0$$

Now we can leverage the fact that $\det(I + xy^\top) = 1 + x^\top y$
with $x = \rho(D - \lambda)^{-1} z$, $y = z$ and obtain:

$$\begin{aligned}
&\det(I + \rho(D - \lambda)^{-1} z z^T) = \\
&= 1 + \rho z^T (D - \lambda I)^{-1} z = \\
&= 1 + \rho \sum_{i=1}^{n} \frac{\zeta_i^2}{d_i - \lambda}
\end{aligned}$$

Thus finally giving:

$$1 + \rho \sum_{i=1}^{n} \frac{\zeta_i^2}{d_i - \lambda} = 0 \qquad \square$$

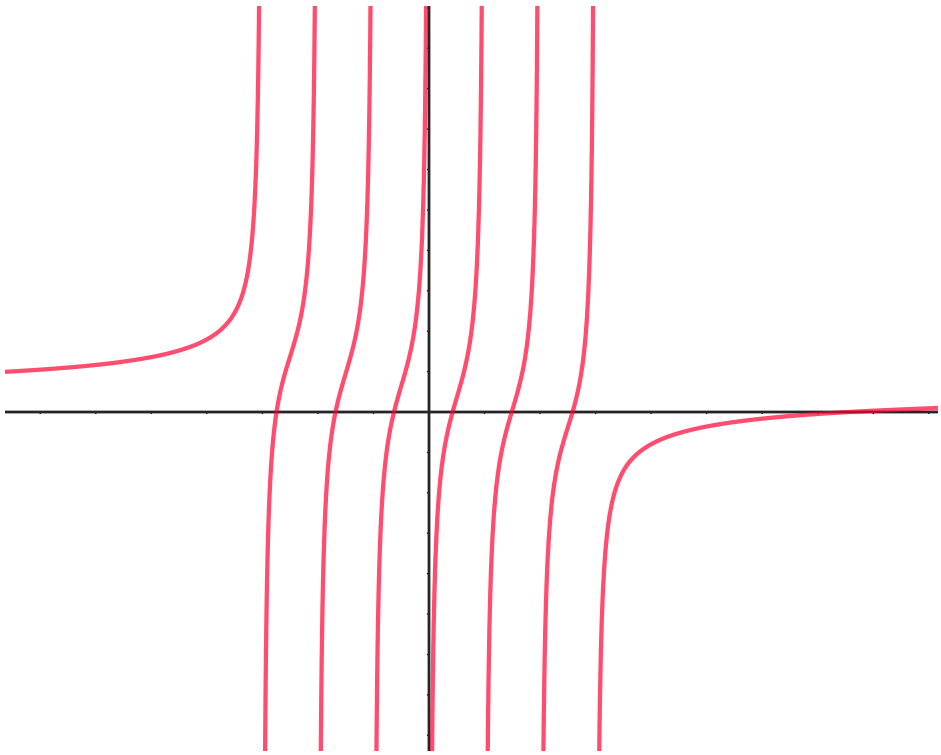

Figure 3.1: Graphical representation of the secular function from Theorem 3.1.

As pointed out in [16], it is important to notice that the diagonal elements of the matrix D are the poles of the secular function and that for $\rho > 0$, f is strictly increasing. This in turns guarantees the strict interlacing property:

$$\lambda_1 > d_1 > \lambda_2 > d_2 > ... > \lambda_n > d_n \tag{3.4}$$

This property is also clearly visible from figure 3.1: the largest eigenvalue λ_1 is the rightmost root of the function, while all other eigenvalues are bracketed by the poles $d_1,\ d_2,\ ...,\ d_n$.

The eigenvectors can then be computed efficiently as well.

Theorem 3.2 *The normalized eigenvectors* $[\ v_1,\ v_2,\ ...,\ v_n\ ]$ *of an irreducible DPR1 problem are given by the following formula:*

$$v_i = \frac{(D - \lambda_i I)^{-1} z}{||(D - \lambda_i I)^{-1} z||} \quad \forall i \text{ s.t. } : 1 \leq i \leq n \tag{3.5}$$

Proof The following proof was adapted from [11]. Consider the vector $(D-\lambda I)^{-1}z$, then for a DPR1 matrix $D+\rho zz^\top$ we have:

$$\begin{aligned}
&\left(D+\rho zz^T\right)\left[(D-\lambda I)^{-1}z\right] = \\
&= \left(D-\lambda I+\lambda I+\rho zz^T\right)(D-\lambda I)^{-1}z = \\
&= z+\lambda(D-\lambda I)^{-1}z+z\left[\rho z^T(D-\lambda I)^{-1}z\right] = \\
&= z+\lambda(D-\lambda I)^{-1}z+z\left[-1+\underbrace{1+\rho z^T(D-\lambda I)^{-1}z}_{=0 \text{ as } \lambda \text{ satisfies 3.3}}\right] = \\
&= z+\lambda(D-\lambda I)^{-1}z-z \\
&= \lambda[(D-\lambda I)^{-1}z]
\end{aligned}$$

Thus $(D-\lambda I)^{-1}z$ is an eigenvector of $D+\rho zz^\top$. Normalization finally yields:

$$v_i = \frac{(D-\lambda_i I)^{-1}z}{||(D-\lambda_i I)^{-1}z||} \qquad \square$$

In principle, it is possible to compute all eigenvalues directly from Eq. 3.3 however, in practice, computing the roots of the secular function directly results in numerically unstable results: it is impossible to guarantee that the eigenvalues are computed with high enough accuracy to ensure the orthogonality of the eigenvectors computed with Theorem 3.2.

The cause for this is the ill conditioned nature of the secular function itself. Figure 3.2 shows how even when the values of D, ρ and z are sufficiently "nice", the secular function can be ill-conditioned with exploding first derivative close to its interior roots.

3.3.1 Overview of the algorithm

This section details the algorithm proposed in [16]. The core idea is to apply a shift and invert technique to the irreducible DPR1 matrix A from Eq. 3.2 in order to compute all eigenvalues as functions of the extremal eigenvalues (either smallest or largest in absolute value) of the shifted inverted problem. Mathematically, this is achieved with the following three step process for each eigenpair:

1. The shifted matrix $A_i = (A - d_i I)$ is computed. This matrix shares the same eigenvectors as the original matrix A and its eigenvalues are tied to those of A via the following relation $\mu_i = \lambda_i - d_i$.
2. The inverse of the shifted matrix A_i^{-1} is computed. The eigenvalues ν_j, $1 \leq j \leq n$ of this matrix are the inverses of the eigenvalues of the

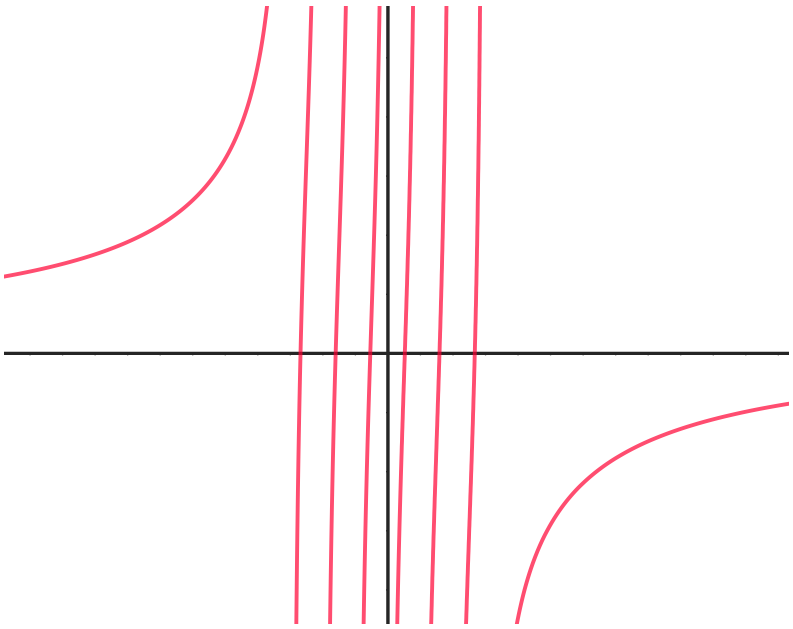

Figure 3.2: Graphical representation of the secular function generated by $D = [-3,-2,-1,0,1,2,3]$, $z = [3,3,3,3,3,3,3]$, $\rho = 1$. The largest eigenvalue corresponds to the right-most root, which is not visible. The left side of the function never converges to zero.

shifted matrix, i.e.: $\nu_j = \frac{1}{\mu_j}\ \forall j$. Only one extremal ν_i is computed as it is the only value that can be calculated in a numerically sound way.

3. Once ν_i is computed with sufficient accuracy, compute μ_i and the corresponding eigenvector v_i using Theorem 3.2, then compute the eigenvalue $\lambda_i = \mu_i + d_i$.

3.3.2 Numerical Implementation

Consider an irreducible DPR1 matrix A as described in 3.2 and let $\lambda,\ v$ be an eigenpair of A. Additionally, let d_i be the closest pole to λ, then by property 3.4 it is guaranteed that $\lambda = \lambda_i$ or $\lambda = \lambda_{i+1}$.

We apply the three step process described in the previous section. The shifted matrix A_i is given by:

$$A_i = A - d_i I = \begin{bmatrix} D_1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & D_2 \end{bmatrix} + \rho \begin{bmatrix} z_1 \\ \zeta_i \\ z_2 \end{bmatrix} \begin{bmatrix} z_1^T & \zeta_i & z_2^T \end{bmatrix} \tag{3.6}$$

with:

$$
\begin{aligned}
D_1 &= \operatorname{diag}(d_1 - d_i, \ldots, d_{i-1} - d_i) \\
D_2 &= \operatorname{diag}(d_{i+1} - d_i, \ldots, d_n - d_i) \\
z_1 &= \begin{bmatrix} \zeta_1 & \zeta_2 & \cdots & \zeta_{i-1} \end{bmatrix}^T \\
z_2 &= \begin{bmatrix} \zeta_{i+1} & \zeta_{i+2} & \cdots & \zeta_n \end{bmatrix}^T
\end{aligned}
$$

The inverse A_i^{-1} is computed according to Theorem 3.3.

Theorem 3.3 *The inverse of a shifted DPR1 matrix is a permuted arrowhead matrix of the form:*

$$
A_i^{-1} = \begin{bmatrix} D_1^{-1} & w_1 & 0 \\ w_1^T & b & w_2^T \\ 0 & w_2 & D_2^{-1} \end{bmatrix} \tag{3.7}
$$

with:

$$
\begin{aligned}
w_1 &= -D_1^{-1} z_1 \frac{1}{\zeta_i} \\
w_2 &= -D_2^{-1} z_2 \frac{1}{\zeta_i} \\
b &= \frac{1}{\zeta_i^2} \left(\frac{1}{\rho} + z_1^T D_1^{-1} z_1 + z_2^T D_2^{-1} z_2 \right).
\end{aligned}
$$

Proof The proof can be carried out by multiplication. □

The eigenvalues of A are tied to those of A_i and A_i^{-1} by the following equation:

$$
\lambda_i = \mu_i + d_i = \frac{1}{\nu_i} + d_i \tag{3.8}
$$

A_i^{-1} is a permuted arrowhead matrix with a very specific structure and similarly to the DPR1 case, its eigenvalues can be computed as the zeros of a secular function.

Theorem 3.4 *Let M be an irreducible arrowhead matrix of the form*

$$
M = \begin{bmatrix} D & u \\ u^\top & \alpha \end{bmatrix}
$$

where:

$$D = \text{diag}(d_1,\ d_2,\ ...,\ d_n)$$
$$u = [\ v_1,\ v_2,\ ...,\ v_n\ ]^\top$$

such that:

- $v_i \neq 0\ \forall i$ *s.t.:* $1 \leq i \leq n$
- $i \neq j\ \Rightarrow\ d_i \neq d_j\ \forall i, j$ *(the elements of D are unique)*
- $i < j\ \Rightarrow\ d_i > d_j\ \forall i, j$ *(D is sorted decreasingly)*

then its eigenvalues are the roots of the secular function:

$$f(\lambda) := \alpha - \lambda - \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda} \tag{3.9}$$

Proof The proof is similar to that of Theorem 3.1. The eigenvalues of M are the roots of its characteristic polynomial:

$$\det(M - \lambda I) = \det\left(\begin{bmatrix} D - \lambda I & u \\ u^\top & \alpha - \lambda \end{bmatrix}\right) = 0$$

Notice that $D - \lambda I$ is never singular; otherwise, there would be a zero element in u, making the matrix M not irreducible.

Now, applying the Schur determinant formula, we rewrite:

$$\det\left(\begin{bmatrix} D - \lambda I & u \\ u^\top & \alpha - \lambda \end{bmatrix}\right) = \det(D - \lambda I)\det(\alpha - \lambda - u^\top (D - \lambda I)^{-1} u)$$

Knowing that $\det(D - \lambda I)$ is never zero as $D - \lambda I$ is never singular, we can further simplify:

$$\det(\alpha - \lambda - u^\top (D - \lambda I)^{-1} u) = \alpha - \lambda - \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda}$$

finally giving:

$$\alpha - \lambda - \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda} = 0 \qquad \square$$

Theorem 3.4 can be easily generalized to include permuted arrowhead matrices such as those arising when A_i^{-1} is computed (Theorem 3.3).

Consider a permuted arrowhead matrix $\widehat{M}$ of the form:

$$\widehat{M} = \begin{bmatrix} D_1 & u_1 & 0 \\ u_1^\top & \alpha & u_2^\top \\ 0 & u_2 & D_2 \end{bmatrix}$$

$\widehat{M}$ can be recast to an irreducible arrowhead matrix via a similarity transform with a permutation matrix P.

$$M = P\widehat{M}P^\top = \begin{bmatrix} D_1 & 0 & u_1 \\ 0 & D_2 & u_2 \\ u_1^\top & u_2^\top & \alpha \end{bmatrix} = \begin{bmatrix} D & u \\ u^\top & \alpha \end{bmatrix}$$

Given that P is an orthogonal matrix, $\widehat{M}$ will have the same eigenvalues as M. By leveraging this fact alongside Theorem 3.4, we can conclude that the eigenvalues of A_i^{-1} from Theorem 3.3 are the zeros of the secular function:

$$g(\nu) := b - \nu - w^T(\Delta - \nu I)^{-1}w \tag{3.10}$$

where:

$$\Delta = \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Once the correct extremal eigenvalue ν_i is determined from g, its inverse $\mu_i = \frac{1}{\nu_i}$ is computed. μ_i is then used to compute the corresponding eigenvector v_i using Theorem 3.2. Finally, $\lambda_i = \mu_i + d_i$ is calculated.

3.3.3 Ensuring numerical stability

When computing the eigendecomposition of DPR1 matrices, as described in the previous section, particular care must be given to ensure numerical stability during certain steps of the algorithm. This section briefly describes which operations are problematic and what solutions can be implemented to ensure correct numerical results.

Computing the matrix A_i^{-1}

The evaluation of the b element of A_i^{-1} from Theorem 3.3 requires special considerations in order to ensure that the inverse is calculated in a numerically sound way. This in turn is critical for an accurate computation of λ. In

certain cases it is necessary to evaluate b using quadruple working precision datatypes such as, for example, GCC's `__float128`. This is considerably slower than using standard floating point operations, however when considering sufficiently "nice" inputs it is rarely if ever required. To determine when quadruple precision arithmetic is needed, two conditioning values K_b and K_z are introduced in [16] as follows:

$$K_b = \frac{1 + \rho z_1^T D_1^{-1} z_1 - \rho z_2^T D_2^{-1} z_2}{\left|1 + \rho z_1^T D_1^{-1} z_1 + \rho z_2^T D_2^{-1} z_2\right|}$$
$$K_z = \frac{1}{|\zeta_i|} \sum_{j=1,\ j\neq i}^{n} |\zeta_j|$$

K_b measures whether the element b is computed accurately enough, while K_z measures whether the computation of b affects $\|A_i^{-1}\|_2$ and the final result. Quadruple arithmetic precision is needed whenever $K_b \gg 1$ and $K_z \gg \mathcal{O}(N)$, i.e. when b is not computed accurately enough and its value greatly affects the result.

Ill-conditioned shift d_k

For certain ill-conditioned inputs it is possible that a shift d_i is chosen that is not the closest to the eigenvalue λ that is being computed. This results in ν not being the absolute largest eigenvalue of A_i^{-1}. The conditioning number K_ν, introduced in [16], is indicative of how far ν is from $\nu_{\max}$:

$$K_\nu = \frac{\|A_i^{-1}\|_2}{|\nu|} = \frac{|\nu_{\max}|}{|\nu|} \tag{3.11}$$

Given that the spectral norm $\|A_i^{-1}\|_2$ is not readily available, it is possible to compute K_ν using either the Frobenius norm or 1-norm. This will still yield correct results as $\|A_i^{-1}\|_2 \leq \|A_i^{-1}\|_F$ and $\|A_i^{-1}\|_2 \leq \|A_i^{-1}\|_1$ [18]. If $K_\nu \gg 1$ there is no guarantee that the final result λ will be computed accurately enough. The solution to this problem is to compute a non-standard shift σ that is close to λ, yet different from the neighbouring poles. ν can then be computed accurately as the largest eigenvalue of a shift-inverted DPR1 matrix A_σ^{-1}. In this case, the inverse is another DPR1 matrix, which can be computed efficiently via the Sherman-Morrison-Woodbury formula as:

$$A_\sigma^{-1} = D_\sigma^{-1} + \gamma D_\sigma^{-1} z z^T D_\sigma^{-1}, \quad \gamma = -\frac{\rho}{1 + \rho z^T D_\sigma^{-1} z} \tag{3.12}$$

Here it is possible that the computation of γ must be done using quadruple precision. In this case the conditioning number K_ρ is given as:

$$K_\rho = \frac{|\frac{1}{\rho} + z^\top D_\sigma z|}{\frac{1}{|\rho|} + z^\top \operatorname{abs}(D_\sigma^{-1})z} \tag{3.13}$$

where $\operatorname{abs}(D_\sigma^{-1})$ is the coefficient-wise absolute value of D_σ^{-1}.

Quasi-zero eigenvalue

In certain rare situations there is a possibility that, for an eigenvalue λ that is much closer to 0 than to any pole d_i, Eq. 3.8 could involve large cancellation. The solution is to recompute λ as the largest value of A^{-1} directly. This can be achieved efficiently with a two-step procedure:

1. Compute A^{-1} using Eq. 3.12 with $\sigma = 0$.
2. Recompute λ_i using inverse iteration: given that the eigenvector v_i has already been calculated accurately, it is possible to use it as the initial guess of inverse iteration in order to compute λ_i accurately in very few iterations.

3.4 Solving the eigenproblem for general DPR1 matrices

This section addresses the eigenproblem for general DPR1 matrices. The conditions for diagonalising an irreducible DPR1 matrix (Eq. 3.4) as previously described are rarely met in real-world problems derived from physical systems. Therefore, finding a numerically stable and efficient method to transform general DPR1 matrices into irreducible ones is crucial. This process is called "deflation."

3.4.1 Overview

The goal of deflation is to ensure that two conditions are met: first, that each element of D is unique and second, that no zero element appears in the vector z. These two conditions are enforced in separate steps, known as type 1 deflation and type 2 deflation. The ordering condition set on D is achieved by a preliminary permutation step. The following subsections describe the deflation process in detail.

3.4.2 Type 1 deflation

Type 1 deflation aims to ensure that all elements in the D matrix of a DPR1 problem are unique. Without loss of generality, we assume that the elements in D are sorted decreasingly. Suppose that $D = \operatorname{diag}(d_1,\ d_2,\ ...,\ d_n)$ with $d_i = d_j$ for some i and j such that $i < j$. Given that D is ordered, it can only be the case that $d_i = d_k\ \forall k \in \{i,\ i+1,\ ...,\ j\}$. Let $d_i = \delta$, then the matrix D is:

$$D = \begin{bmatrix} d_1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & d_{i-1} & & & & & & \\ & & & \delta & & & & & \\ & & & & \ddots & & & & \\ & & & & & \delta & & & \\ & & & & & & d_{j+1} & & \\ & & & & & & & \ddots & \\ & & & & & & & & d_n \end{bmatrix} = \begin{bmatrix} D_1 & & \\ & \delta I & \\ & & D_2 \end{bmatrix} \tag{3.14}$$

It is possible to find a Householder reflection that annihilates all but one elements of z that correspond to elements δ of the matrix D.

Theorem 3.5 *For a DPR1 matrix $D + zz^\top$ with D as described in Eq. 3.14, there exists a Householder matrix H such that*

$$H(D + zz^\top)H = HDH + Hzz^\top H = D + \tilde{z}\tilde{z}^\top \tag{3.15}$$

where

$$\tilde{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_{i-1} \\ -\sqrt{\sum_{p=i}^{j} z_p^2} \\ 0 \\ \vdots \\ 0 \\ z_{j+1} \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} z_{1:i-1} \\ -\|z_\delta\| e_0 \\ z_{j+1:n} \end{bmatrix} \tag{3.16}$$

Proof Let z_δ be the part of the z vector that corresponds to the δ elements in D. We first construct a "local" Householder matrix that annihilates all but the first element of z_δ. Householder reflection matrices are of the form:

$$\tilde{H} = I - 2\frac{uu^\top}{\|u\|^2} \tag{3.17}$$

Let:

$$u = z_\delta + \|z_\delta\| e_0 \tag{3.18}$$

where $e_0 = [1, 0, 0, ..., 0]^\top$. We now have a Householder matrix $\tilde{H}$ that satisfies the following property:

$$\begin{aligned}\tilde{H}z_\delta &= (I - 2\frac{uu^\top}{\|u\|^2})z_\delta = \\ &= z_\delta - 2\frac{(z_\delta + \|z_\delta\|e_0)(z_\delta + \|z_\delta\|e_0)^\top}{\|z_\delta + \|z_\delta\|e_0\|^2}z_\delta\end{aligned}$$

Notice that the denominator $\|z_\delta + \|z_\delta\|e_0\|^2$ can be further simplified to

$$\begin{aligned}\|z_\delta + \|z_\delta\|e_0\|^2 &= (z_\delta + \|z_\delta\|e_0)^\top(z_\delta + \|z_\delta\|e_0) = \\ &= z_\delta^\top z_\delta + 2\|z_\delta\|z_{\delta,0} + \|z_\delta\|^2 = \\ &= 2(\|z_\delta\|^2 + \|z_\delta\|z_{\delta,0})\end{aligned}$$

Where $z_{\delta,0}$ is the first element of z_δ. We plug this result back into the previous expression and continue simplifying:

$$\begin{aligned}&z_\delta - 2\frac{(z_\delta + \|z_\delta\|e_0)(z_\delta + \|z_\delta\|e_0)^\top}{\|z_\delta + \|z_\delta\|e_0\|^2}z_\delta = \\ &= z_\delta - 2\frac{(z_\delta + \|z_\delta\|e_0)}{2(\|z_\delta\|^2 + \|z_\delta\|z_{\delta,0})}(\|z_\delta\|^2 + \|z_\delta\|z_{\delta,0}) = \\ &= z_\delta - (z_\delta + \|z_\delta\|e_0) = -\|z_\delta\|e_0\end{aligned}$$

Hence we are able to construct a Householder reflection that annihilates all but the first element of z_δ by setting u from Eq. 3.18 in Eq. 3.17.

Now we construct a global Householder matrix H that is able to deflate the problem. Let:

$$H = \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} \tag{3.19}$$

Clearly H is still a reflection matrix as it satisfies:

$$HH = \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix}\begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} = \begin{bmatrix} I & & \\ & \tilde{H}\tilde{H} & \\ & & I \end{bmatrix} = \begin{bmatrix} I & & \\ & I & \\ & & I \end{bmatrix}$$

Yet it also holds that:

$$HDH = \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} \begin{bmatrix} D_1 & & \\ & \delta I & \\ & & D_2 \end{bmatrix} \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} =$$
$$= \begin{bmatrix} D_1 & & \\ & \delta \tilde{H}\tilde{H} & \\ & & D_2 \end{bmatrix} = \begin{bmatrix} D_1 & & \\ & \delta I & \\ & & D_2 \end{bmatrix} = D$$

and that:

$$Hz = \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} \begin{bmatrix} z_{1:i-1} \\ z_\delta \\ z_{j+1:n} \end{bmatrix} = \begin{bmatrix} z_{1:i-1} \\ \tilde{H} z_\delta \\ z_{j+1:n} \end{bmatrix} = \begin{bmatrix} z_{1:i-1} \\ -\|z_\delta\| e_0 \\ z_{j+1:n} \end{bmatrix} \tag{3.20}$$

Which in turn guarantee that H satisfies the property:

$$H(D + zz^\top)H = HDH + Hzz^\top H = D + \tilde{z}\tilde{z}^\top \qquad \square$$

In practice, most of the mathematical operations shown for type 1 deflation are done implicitly to save the cost of computation. Snippet 2 shows the pseudocode for a possible implementation, yet further considerations are required to fully guarantee numerical stability of this process.

3.4.3 Type 2 deflation

Type 2 deflation ensures that no zero elements appear in z. This is done by partitioning the z vector in zero and non-zero elements, and reordering D accordingly. Consider the DPR1 problem:

$$D + zz^\top = \begin{bmatrix} \tilde{D} & \\ & \widehat{D} \end{bmatrix} + \begin{bmatrix} \tilde{z} \\ 0 \end{bmatrix} \begin{bmatrix} \tilde{z} \\ 0 \end{bmatrix}^\top = \begin{bmatrix} \tilde{D} + \tilde{z}\tilde{z}^\top & \\ & \widehat{D} \end{bmatrix} \tag{3.21}$$

Clearly some of the eigenvalues can be read directly, as the matrix is diagonal in $\widehat{D}$. Thus it is sufficient to compute the diagonalisation $\tilde{D} + \tilde{z}\tilde{z}^\top = \tilde{Q}\tilde{\Lambda}\tilde{Q}^\top$ and the full solution will be given by:

$$D + zz^\top = Q\Lambda Q^\top = \begin{bmatrix} \tilde{Q} & \\ & I \end{bmatrix} \begin{bmatrix} \tilde{\Lambda} & \\ & \widehat{D} \end{bmatrix} \begin{bmatrix} \tilde{Q}^\top & \\ & I \end{bmatrix} \tag{3.22}$$

By leveraging these observations, it is possible to implement an efficient process that allows for reduction and diagonalisation of general DPR1 matrices:

Algorithm 2: Type 1 deflation algorithm

Initialize : multiplicity := []
Initialize : $H := I$
// D is assumed ordered decreasingly
$i := 0$
while $i < N$ **do**
  $j := i + 1$
  $\delta := D(i)$
  while $j \leq N$ ***and*** $\delta = D(j)$ **do**
    $j := j + 1$
  end
  if $j - i > 1$ **then**
    multiplicity.append(pair(i, $j - i$))
  end
  $i := j$
end
foreach *pair (i, m) in* multiplicity **do**
  $u := z_{i:i+m}$
  $u(0) := u(0) + \|z_{i:i+m}\|$
  $H_{i:i+m,i:i+m} := I - 2\frac{uu^\top}{\|u\|^2}$
  $z(i) := -\|z_{i:i+m}\|$
  $z_{i+1:i+m} := 0$
end

1. If needed, compute a permutation P that sorts the matrix D in decreasing order. Apply P to D and to z.
2. Compute type 1 deflation as previously described. This introduces new zero entries in z, while leaving D unchanged.
3. Compute type 2 deflation. This partitions the zero and non-zero elements of z and reorders D as described in Eq. 3.21.
4. The irreducible solution is computed as described in section 3.3.
5. The deflated solution is computed by setting $\lambda_i = d_i$ and $v_i = e_i$ for all i where $z_i = 0$ as described in Eq. 3.22.
6. The final solution is computed by applying the deflation Householder matrix to the eigenvectors and permuting both eigenvalues and eigenvectors back to their original positions.

Chapter 4

Eigendecomposition of broad arrowhead matrices

4.1 Introduction

This chapter discusses the efficient eigendecomposition of broad arrowhead matrices and its contents are adapted and simplified from a forthcoming paper submitted for publication [19]. Arrowhead matrices are frequently encountered in many scientific and engineering problems [20]. While an efficient eigendecomposition algorithm, which will be discussed in the next chapter, for such matrices is already exists [20], many physical systems must be described by matrices that are generalizations of simple arrowhead matrices. This is the case for numerical simulations implementing instanton theory. Specifically, simulations based on Golden-rule instanton theory, as described in [21] and [22], involve an iterative optimization process that requires repeated diagonalisation of a banded arrowhead Hessian matrix with arrowhead-width 1. These matrices can easily become very large, making the eigendecomposition process expensive, and it is essential to develop and implement optimized algorithms which can enable the simulation of new larger physical systems.

4.2 Problem description

Consider an $n \times n$ broad arrowhead matrix of the following form:

$$M = \begin{bmatrix} \widetilde{M}_{l\times l} & W_{l\times g} \\ W_{g\times l}^{\top} & R_{g\times g} \end{bmatrix} \tag{4.1}$$

where $\widetilde{M}_{l\times l}$ is either banded or block-tridiagonal, $W_{l\times g}$ is a tall and thin matrix and $R_{g\times g}$ is a small dense square matrix. Additionally, it holds that $l + g = n$, where g is known as the "width" of the arrowhead of the matrix.

The goal is to compute the eigendecomposition of M efficiently by exploiting the structure of the matrix. Figure 4.1 displays a more intuitive visualisation of the broad arrowhead matrix of Eq. 4.1.

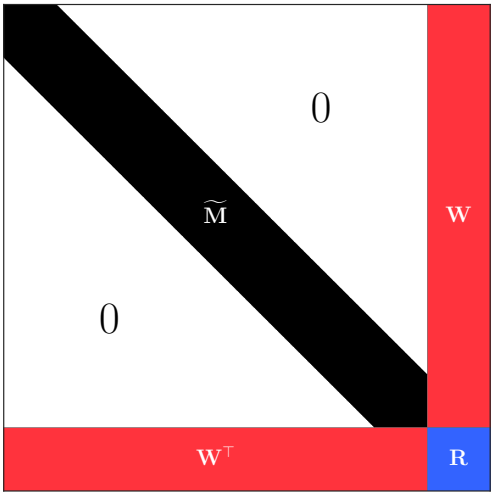

Figure 4.1: Structure of a banded broad arrowhead matrix as described in Eq. 4.1.

4.3 Overview of the algorithm

The eigendecomposition strategy for broad arrowhead matrices relies on a two-step process:

1. An intermediate factorization of the matrix M, known as *Arrowhead Factorization* (AF), is computed by diagonalising the $\widetilde{M}$ matrix with an optimized eigensolver that can exploit its structure, such as for example the LAPACK banded eigensolver routine DSBEVD or the BD&C algorithm discussed in chapter 2.
2. Once the arrowhead factorization is known, it is possible to reformulate the problem as a series of g simple arrowhead matrices, which can be diagonalised efficiently in $\mathcal{O}(n^2)$ using the method described in [20]. The full solution is then reconstructed by backtransforming the eigenvectors of the simple arrowhead matrices.

4.4 Numerical implementation

The previous section introduced the eigendecomposition strategy as a two-phase process. This section will delve into the essential details necessary for implementing both steps of this method.

4.4.1 Arrowhead factorization

The first step of the process involves computing an intermediate factorization known as arrowhead factorization (AF). Theorem 4.1 gives a definition and a proof of existence, both taken from the work proposed in [19], of arrowhead factorization for any real symmetric matrix.

Theorem 4.1 (Existence of arrowhead factorization) *Given any square symmetric real matrix $M \in \mathbb{R}^{n\times n}$, there exists a pair of matrices $A \in \mathbb{R}^{n\times n}$ and $Q \in \mathbb{R}^{n\times n}$ such that:*

$$M = QAQ^\top \tag{4.2}$$

Where Q is an orthogonal matrix and A is a symmetric arrowhead matrix of the form:

$$A = \begin{bmatrix} D & u \\ u^\top & \alpha \end{bmatrix}$$
$$D = \operatorname{diag}(d_1, d_2, ..., d_{n-1})$$
$$u \in \mathbb{R}^{n-1},\ \alpha \in \mathbb{R}$$

Proof We give proof of AF by construction. Consider the following representation of the matrix M:

$$M = \begin{bmatrix} \widetilde{M} & z \\ z^\top & \rho \end{bmatrix} = \begin{bmatrix} \widetilde{M} & \\ & 0 \end{bmatrix} + \begin{bmatrix} 0 & z \\ z^\top & \rho \end{bmatrix} \tag{4.3}$$

Here $\widetilde{M} \in \mathbb{R}^{n-1\times n-1}$ is the top-left $n-1 \times n-1$ block of M, $z \in \mathbb{R}^{n-1}$ is a vector containing first $n-1$ entries of the last column of M and $\rho \in \mathbb{R}$ is the scalar value found in the bottom right corner of M. Notice that the second term of the sum is an arrowhead matrix with zero shaft. We proceed by computing the eigensystem of the symmetric real matrix $\widetilde{M} = \widetilde{Q}\widetilde{D}\widetilde{Q}^\top$ and defining the augmented solution:

$$Q = \begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix}, \quad D = \begin{bmatrix} \widetilde{D} & \\ & 0 \end{bmatrix}, \tag{4.4}$$

Using Q and D from Eq. 4.4, it is possible to reconstruct the first term of the right-hand side of Eq. 4.3:

$$QDQ^\top = \begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix} \begin{bmatrix} \widetilde{D} & \\ & 0 \end{bmatrix} \begin{bmatrix} \widetilde{Q}^\top & \\ & 1 \end{bmatrix} = \begin{bmatrix} \widetilde{M} & \\ & 0 \end{bmatrix} \tag{4.5}$$

Substituting the results of Eq. 4.4 in Eq. 4.3 yields the following:

$$\begin{aligned} M &= \begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix} \begin{bmatrix} \widetilde{D} & \\ & 0 \end{bmatrix} \begin{bmatrix} \widetilde{Q}^\top & \\ & 1 \end{bmatrix} + \begin{bmatrix} 0 & z \\ z^\top & \rho \end{bmatrix} = \\ &= \begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix} \left(\begin{bmatrix} \widetilde{D} & \\ & 0 \end{bmatrix} + \begin{bmatrix} 0 & \widetilde{Q}^\top z \\ z^\top \widetilde{Q} & \rho \end{bmatrix} \right) \begin{bmatrix} \widetilde{Q}^\top & \\ & 1 \end{bmatrix} = \\ &= \begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix} \left(\begin{bmatrix} \widetilde{D} & \\ & 0 \end{bmatrix} + \begin{bmatrix} 0 & w \\ w^\top & \rho \end{bmatrix} \right) \begin{bmatrix} \widetilde{Q}^\top & \\ & 1 \end{bmatrix} = \\ &= \underbrace{\begin{bmatrix} \widetilde{Q} & \\ & 1 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} \widetilde{D} & w \\ w^\top & \rho \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \widetilde{Q}^\top & \\ & 1 \end{bmatrix}}_{Q^\top} \end{aligned} \tag{4.6}$$

□

The proof of Theorem 4.1 showcases how to construct the arrowhead factorization of M by diagonalising the $\widetilde{M}$ matrix. Since $\widetilde{M}$ has a special structure, this can be done very efficiently as the only additional operation required is a single matrix-vector product to construct the head of the arrowhead matrix.

4.4.2 Reconstructing the full solution

Once the arrowhead factorization $M = Q_0 A Q_0^\top$ is known, it is possible to retrieve the full eigendecomposition by diagonalising $A = Q_1 D Q_1^\top$ and backtransforming its eigenvectors:

$$M = Q_0 A Q_0^\top = Q_0 Q_1 D Q_1^\top Q_0^\top = Q D Q^\top \tag{4.7}$$

Notice that if $\widetilde{M}$ does not offer an exploitable structure right away, it is possible to apply arrowhead factorization recursively to diagonalise $\widetilde{M}$ as well. Alternatively, this can also be seen as a direct application of Theorem 4.1 to the top-left $l + 1 \times l + 1$ corner block of M (where $l + g = n$):

$$M = \begin{bmatrix} \widetilde{M}_{l \times l} & \\ & \mathbf{0}_{g \times g} \end{bmatrix} + \begin{bmatrix} \mathbf{0}_{l \times l} & W_{l \times g} \\ W_{g \times l}^\top & R_{g \times g} \end{bmatrix} \tag{4.8}$$

In practice it is convenient to store the full arrowhead implicitly in a matrix $S \in \mathbb{R}^{n \times g}$ constructed by taking the last g columns of M:

$$S = M[:, -g:]^1 = \begin{bmatrix} W \\ R \end{bmatrix} \tag{4.9}$$

[1] We use `Numpy` style indexing. See `https://numpy.org/doc/stable/user/basics.indexing.html`.

Algorithm 3 showcases the pseudocode for full diagonalisation based on arrowhead factorization using this representation of the S matrix.

Algorithm 3: AF Eigensolver

Input: Matrix $\widetilde{M}$, Matrix S
$g := S$.cols(), $l := \widetilde{M}$.cols()
$n := l + g$
$D :=$ Zero Vector(n)
$Q :=$ Identity Matrix(n)

`// diagonalise` $\widetilde{M}$ `using optimized solver`
OptimizedEigenSolver solver$(\widetilde{M})$
$D[: l] :=$ solver.eigenvalues()
$Q[: l, : l] :=$ solver.eigenvectors()

`// Arrowhead factorization to eigendecomposition`
for $k = 0$ *to* $g - 1$ **do**
  $w := Q[: (l + k), : (l + k)]^{\top} * S[: (l + k), k]$
  $\rho := S[(l + k), k]$
  ArrowheadEigenSolver solver$(D[: (l + k)], w, \rho)$
  $D[: (l + k + 1)] :=$ solver.eigenvalues()
  $Q[: (l + k + 1), : (l + k + 1)] *=$ solver.eigenvectors()
end
return D, Q

The final missing link in algorithm 3 is implementing an efficient eigensolver for simple arrowhead matrices, which will be discussed in the upcoming chapter.

Chapter 5

Eigendecomposition of arrowhead matrices

5.1 Introduction

Arrowhead matrices frequently emerge when describing physical systems in different scientific and engineering domains. Their specific structure can be exploited in order to compute eigenvalues and eigenvectors much more efficiently. In this chapter we give a brief description of a recently proposed algorithm for the solution of the eigenproblem for arrowhead matrices. This method is of dual significance: not only does it address the direct challenges posed by arrowhead matrices, but more importantly it also enables the generalized arrowhead factorization based eigensolver described in the previous chapter. This strategy was proposed, again, by Jakovčević Stor, Slapničar and Barlow in [20] and is strictly tied to the DPR1 eigenproblem. Again, the focus is set on the equations required for an efficient implementation as a more in-depth discussion is available in the original publication. For the sake of brevity and conciseness and given the many similarities of this chapter with chapter 3, only the key differences will be highlighted.

5.2 Problem description

The eigenproblem for arrowhead matrices consists of computing the eigendecomposition of an $n \times n$ symmetric real matrix of the form:

$$A = \begin{bmatrix} D & u \\ u^\top & \alpha \end{bmatrix} \tag{5.1}$$

where D is a diagonal matrix, α is a scalar and u is a vector.

5.3 Solving the eigenproblem for irreducible arrowhead matrices

Similarly to chapter 3, we first discuss the solution of the eigenproblem for irreducible arrowhead matrices and subsequently we generalize the deflation process to the arrowhead case.

Without loss of generality consider the eigenproblem generated by an irreducible (in the sense of Theorem 3.4) arrowhead matrix A as defined in equation 5.1. As we have proven in chapter 3, it is possible to exploit the special structure of A to compute its eigenvalues. According to Theorem 3.4, the eigenvalues of A are the roots of the secular function:

$$f(\lambda) := \alpha - \lambda - \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda} \tag{5.2}$$

Notice that once again, the following interlacing property is satisfied:

$$\lambda_1 > d_1 > \lambda_2 > d_2 > \cdots > d_{n-2} > \lambda_{n-1} > d_{n-1} > \lambda_n \tag{5.3}$$

However, issues related to numerical instability still make it impractical to compute all eigenvalues directly from Eq. 5.2.

The eigenvectors can be computed efficiently as well.

Theorem 5.1 *The normalized eigenvectors* $[\, v_1,\ v_2,\ \ldots,\ v_n \,]$ *of an irreducible arrowhead matrix are given by the following formula:*

$$v_i = \frac{x_i}{\|x_i\|_2}, \quad x_i = \begin{bmatrix} (D - \lambda_i I)^{-1} u \\ -1 \end{bmatrix}, \quad i = 1, \quad \ldots, n \tag{5.4}$$

Proof Consider the vector x_i as defined in Eq. 5.4, then, for an arrowhead matrix A as defined in Eq. 5.1, given an eigenvalue λ_i of A, we have:

$$Ax_i = \begin{bmatrix} D & u \\ u^\top & \alpha \end{bmatrix} \begin{bmatrix} (D - \lambda_i I)^{-1} u \\ -1 \end{bmatrix} = \begin{bmatrix} D(D - \lambda_i)^{-1} u - u \\ u^\top (D - \lambda_i)^{-1} u - \alpha \end{bmatrix}$$

Notice that $(D - \lambda_i I)$ is invertible as λ_i can never be an eigenvalue of D. Otherwise there would be some element of u that is zero, making A not irreducible as defined in Theorem 3.4.

We first prove that $D(D - \lambda_i I)^{-1} u - u = \lambda_i (D - \lambda_i I)^{-1} u$:

$$
\begin{aligned}
&D(D-\lambda_i I)^{-1}u - u = \\
&= D(D-\lambda_i I)^{-1}u - (D-\lambda_i I)(D-\lambda_i I)^{-1}u = \\
&= (D-(D-\lambda_i I))(D-\lambda_i I)^{-1}u = \\
&= \lambda_i (D-\lambda_i I)^{-1}u
\end{aligned}
$$

Next we show that $u^\top (D-\lambda_i I)^{-1}u - \alpha = \lambda_i(-1)$

$$
\begin{aligned}
&u^\top (D-\lambda_i I)^{-1}u - \alpha = \\
&= (-1)(\alpha - u^\top (D-\lambda_i I)^{-1}u) = \\
&= (-1)(\alpha - \lambda_i + \lambda_i - u^\top (D-\lambda_i I)^{-1}u) = \\
&= (-1)(\underbrace{\alpha - \lambda_i - \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda_i}}_{=0 \text{ by Th. 3.4}} + \lambda_i) \\
&= \lambda_i(-1)
\end{aligned}
$$

Hence we have verified that $Ax_i = \lambda_i x_i$ and as such that x_i is an eigenvector of A. Normalization finally gives:

$$
v_i = \frac{x_i}{\|x_i\|_2} \qquad \square
$$

5.3.1 Overview of the algorithm

The algorithm follows the same shift and invert strategy described in chapter 3. The eigenvalues are calculated from the extremal roots of the secular equation generated by the shifted inverted problem. This is again achieved with a three step procedure:

1. The shifted matrix $A_i = (A - d_i I)$ is computed. Its eigenvalues are tied to those of A via the following relation $\mu_i = \lambda_i - d_i$.
2. The inverse of the shifted matrix A_i^{-1} is computed. The eigenvalues ν_j, $1 \leq j \leq n$ of this matrix are the inverses of the eigenvalues of the shifted matrix.
3. Once ν_i is computed with sufficient accuracy, compute μ_i and the corresponding eigenvector v_i using Theorem 5.1, then compute the eigenvalue $\lambda_i = \mu_i + d_i$.

5.3.2 Numerical implementation

Consider an irreducible arrowhead matrix A as described in 5.1 and let λ, v be an eigenpair of A. Additionally, let d_i be the closest pole to λ, then by property 5.3 it is guaranteed that $\lambda = \lambda_i$ or $\lambda = \lambda_{i+1}$. We apply the three step process described in the previous section.

The shifted matrix A_i is given by:

$$A_i = A - d_i I = \begin{bmatrix} D_1 & 0 & 0 & z_1 \\ 0 & 0 & 0 & \zeta_i \\ 0 & 0 & D_2 & z_2 \\ z_1^\top & \zeta_i & z_2^\top & a \end{bmatrix} \tag{5.5}$$

with:

$$\begin{aligned} D_1 &= \operatorname{diag}\left(d_1 - d_i, \ldots, d_{i-1} - d_i\right), \\ D_2 &= \operatorname{diag}\left(d_{i+1} - d_i, \ldots, d_{n-1} - d_i\right), \\ z_1 &= \begin{bmatrix} \zeta_1 & \zeta_2 & \cdots & \zeta_{i-1} \end{bmatrix}^\top, \\ z_2 &= \begin{bmatrix} \zeta_{i+1} & \zeta_{i+2} & \cdots & \zeta_{n-1} \end{bmatrix}^\top, \\ a &= \alpha - d_i. \end{aligned}$$

The inverse A_i^{-1} is computed according to Theorem 5.2.

Theorem 5.2 *The inverse of a shifted arrowhead matrix is a permuted arrowhead matrix of the form:*

$$A_i^{-1} = \begin{bmatrix} D_1^{-1} & w_1 & 0 & 0 \\ w_1^\top & b & w_2^\top & 1/\zeta_i \\ 0 & w_2 & D_2^{-1} & 0 \\ 0 & 1/\zeta_i & 0 & 0 \end{bmatrix} \tag{5.6}$$

with:

$$\begin{aligned} w_1 &= -D_1^{-1} z_1 \frac{1}{\zeta_i} \\ w_2 &= -D_2^{-1} z_2 \frac{1}{\zeta_i} \\ b &= \frac{1}{\zeta_i^2}\left(-a + z_1^\top D_1^{-1} z_1 + z_2^\top D_2^{-1} z_2\right) \end{aligned}$$

Proof The proof is carried out by multiplication. □

The eigenvalues of A are tied to those of A_i and A_i^{-1} by the following equation:

$$\lambda_i = \mu_i + d_i = \frac{1}{\nu_i} + d_i \tag{5.7}$$

By Theorem 3.4, the eigenvalues of A_i^{-1} are the zeros of the secular function:

$$g(\nu) := b - \nu - w^\top (\Delta - \nu I)^{-1} w \tag{5.8}$$

where:

$$\Delta = \begin{bmatrix} D_1 & & \\ & 0 & \\ & & D_2 \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ \frac{1}{\zeta_i} \\ w_2 \end{bmatrix}$$

Once the correct extremal eigenvalue ν_i is determined from g, its inverse $\mu_i = \frac{1}{\nu_i}$ is computed. μ_i is then used to compute the corresponding eigenvector v_i using Theorem 5.1. Finally, $\lambda_i = \mu_i + d_i$ is calculated.

5.3.3 Ensuring numerical stability

Similarly to the DPR1 case, when computing the eigendecomposition of arrowhead matrices, as described in the previous section, particular care must be given to ensure numerical stability during certain steps of the algorithm. This section briefly describes which operations are problematic and what solutions can be implemented to ensure correct numerical results.

Computing the matrix A_i^{-1}

The evaluation of the b element of A_i^{-1} from Theorem 5.2 requires special considerations: in certain cases it is necessary to evaluate b using quadruple working precision datatypes. To determine when quadruple precision arithmetic is needed, two conditioning values K_b and K_z are introduced in [20] as follows:

$$K_b = \frac{|a| + \left| z_1^\top D_1^{-1} z_1 \right| + \left| z_2^\top D_2^{-1} z_2 \right|}{\left| -a + z_1^\top D_1^{-1} z_1 + z_2^\top D_2^{-1} z_2 \right|}$$

$$K_z = \frac{1}{|\zeta_i|} \sum_{j=1,\ j \neq i}^{n} |\zeta_j|$$

K_b measures whether the element b is computed accurately enough, while K_z measures whether the computation of b affects $\|A_i^{-1}\|_2$ and the final result. Quadruple arithmetic precision is needed whenever $K_b \gg 1$ and $K_z \gg \mathcal{O}(N)$.

Ill-conditioned shift d_k

For certain ill-conditioned inputs it is possible that a shift d_i is chosen that is not the closest to the eigenvalue λ that is being computed. This results in ν not being the absolute largest eigenvalue of A_i^{-1}. The conditioning number K_ν is indicative of how far ν is from the real $\nu_{\max}$:

$$K_\nu = \frac{\|A_i^{-1}\|_2}{|\nu|} = \frac{|\nu_{\max}|}{|\nu|} \tag{5.9}$$

Given that the spectral norm $\|A_i^{-1}\|_2$ is not readily available, it is possible to compute K_ν using either the Frobenius norm or 1-norm. This will still yield correct results as $\|A_i^{-1}\|_2 \leq \|A_i^{-1}\|_F$ and $\|A_i^{-1}\|_2 \leq \|A_i^{-1}\|_1$ [18]. If $K_\nu \gg 1$ there is no guarantee that the final result λ will be computed accurately enough. The solution to this problem is to compute a non-standard shift σ that is close to λ, yet different from the neighbouring poles. ν can then be computed accurately as the largest eigenvalue of a shift-inverted arrowhead matrix A_σ^{-1}. In this case, the inverse is a DPR1 matrix of the form:

$$A_\sigma^{-1} = (A - \sigma I)^{-1} = \begin{bmatrix} (D - \sigma I)^{-1} & \\ & 0 \end{bmatrix} + \rho u u^\top \tag{5.10}$$

with:

$$u = \begin{bmatrix} z^\top (D - \sigma I)^{-1} & -1 \end{bmatrix}^\top, \quad \rho = \frac{1}{a - z^\top (D - \sigma I)^{-1} z}$$

The proof of this statement can again be carried out by multiplication. The largest eigenvalue of A_σ^{-1} can then be computed with Eq. 3.3 from chapter 3.

5.4 Solving the eigenproblem for general arrowhead matrices

This section addresses the eigenproblem for general arrowhead matrices. This process is a direct generalization to the DPR1 deflation process described in chapter 3.

5.4.1 Overview

The deflation process for arrowhead matrices is very similar to that for DPR1 matrices. The dual goal of deflation remains the same: first, ensuring uniqueness of the elements in D and second, ensuring that no zero element appears in the vector z. Again, two types of deflation are distinguished. The following subsections describe the process in detail.

5.4.2 Type 1 deflation

Type 1 deflation aims to ensure that all elements in the D matrix of an arrowhead problem are unique. Without loss of generality, we assume that the elements in D are ordered decreasingly. Suppose that $D = \text{diag}(d_1,\ d_2,\ ...,\ d_n)$ with $d_i = d_j$ for some i and j such that $i < j$. Given that D is ordered, it can only be the case that $d_i = d_k\ \forall k \in \{i,\ i+1,\ ...,\ j\}$. Let $d_i = \delta$, then the matrix D is

$$D = \begin{bmatrix} d_1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & d_{i-1} & & & & & & \\ & & & \delta & & & & & \\ & & & & \ddots & & & & \\ & & & & & \delta & & & \\ & & & & & & d_{j+1} & & \\ & & & & & & & \ddots & \\ & & & & & & & & d_n \end{bmatrix} = \begin{bmatrix} D_1 & & \\ & \delta I & \\ & & D_2 \end{bmatrix} \tag{5.11}$$

in chapter 3 (Theorem 3.5) we have proven, that it is possible to find a Householder reflection that annihilates all but one elements of z corresponding to elements δ of the matrix D. This can be achieved by constructing the householder matrix H as follows:

$$\begin{aligned} u &= z_\delta + \|z_\delta\|\, e_0 \\ \tilde{H} &= I - 2\frac{uu^\top}{\|u\|^2} \\ H &= \begin{bmatrix} I & & \\ & \tilde{H} & \\ & & I \end{bmatrix} \end{aligned}$$

It remains to prove that applying H to the arrowhead matrix does not destroy its structure.

Theorem 5.3 *For an arrowhead matrix A with D as described in Eq. 5.11, there exists a Householder matrix $\widehat{H}$ such that*

$$\widehat{H}A\widehat{H} = \widehat{H}\begin{bmatrix} D & z \\ z^\top & \alpha \end{bmatrix}\widehat{H} = \begin{bmatrix} D & \tilde{z} \\ \tilde{z}^\top & \alpha \end{bmatrix} \tag{5.12}$$

where

$$\tilde{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_{i-1} \\ -\sqrt{\sum_{p=i}^{j} z_p^2} \\ 0 \\ \vdots \\ 0 \\ z_{j+1} \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} z_{1:i-1} \\ -\|z_\delta\|e_0 \\ z_{j+1:n} \end{bmatrix} \tag{5.13}$$

Proof Let $\widehat{H} = \begin{bmatrix} H & 0 \\ 0 & 1 \end{bmatrix}$

The proof is very similar to that for the DPR1 case:

$$\widehat{H}A\widehat{H} = \begin{bmatrix} H & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} D & z \\ z^\top & \alpha \end{bmatrix}\begin{bmatrix} H & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} HDH & Hz \\ z^\top H & 1\alpha \end{bmatrix} = \begin{bmatrix} D & \tilde{z} \\ \tilde{z}^\top & \alpha \end{bmatrix} \qquad \square$$

5.4.3 Type 2 deflation

Type 2 deflation ensures that no zero elements appear in z. Consider the eigenproblem of the arrowhead matrix:

$$A = \begin{bmatrix} \widehat{D} & 0 & 0 \\ 0 & \tilde{D} & \tilde{z} \\ 0 & \tilde{z}^\top & \alpha \end{bmatrix} \tag{5.14}$$

Clearly some of the eigenvalues can be read directly, as the matrix is diagonal in $\widehat{D}$. Hence, it is sufficient to compute the diagonalisation

$$\begin{bmatrix} \tilde{D} & \tilde{z} \\ \tilde{z}^\top & \alpha \end{bmatrix} = \tilde{Q}\tilde{\Lambda}\tilde{Q}^\top$$

and then the full solution will be given by:

$$A = Q\Lambda Q^\top = \begin{bmatrix} I & \\ & \tilde{Q} \end{bmatrix} \begin{bmatrix} \widehat{D} & \\ & \tilde{\Lambda} \end{bmatrix} \begin{bmatrix} I & \\ & \tilde{Q}^\top \end{bmatrix}$$

Therefore, it is possible to implement the same efficient process, that allows for reduction and diagonalisation of general DPR1 matrices, for arrowhead matrices.

Chapter 6

Numerical experiments and benchmarks

This chapter presents a comparative performance analysis of the eigensolver algorithms discussed in this work. The BD&C algorithm is compare against Intel oneAPI MKL's dense eigensolver (DSYEVR) and banded eigensolver (DSBEVD) using a mix of synthetic and real-world test cases. Additionally, this chapter includes a brief comparison between the broad arrowhead eigensolver and the DSYEVR routine, using two synthetic benchmarks for assessment.

6.1 Experimental setup

The benchmarks were conducted on compute nodes equipped with dual 64-core AMD EPYC 7742 CPUs (total of 128 cores) and 512 GB DDR4 memory. The runtime baseline for the performance evaluation was set by the best performing routine, either DSYEVR or DSBEVD, offered by the Intel oneAPI MKL 2022.2.0 library [5]. Our eigensolver implementations are built on top of the Eigen 3.4.0 C++ library [23], backed by the Intel oneAPI MKL backend. Multithreading was handled by the Intel oneAPI Threading Building Blocks 2021.7.0 library [24]. All the software used for these tests was compiled via the Spack package manager [25] with the GCC 12.2.0 compiler. Each benchmark presented in this chapter was evaluated with 5 runs and the median performance results are reported.

6.2 BD&C algorithm

This section is dedicated to the benchmark set for the BD&C algorithm, which specializes in diagonalising tri-block diagonal and banded matrices. The

algorithm is tested against both dense and banded eigensolver routines for a comprehensive comparison. The benchmarks include:

1. A set of synthetic benchmarks consisting of block-tridiagonal matrices with uniformly randomly generated values in the interval between -1 and 1. The test dimensions range from 1024×1024 to 16384×16384 and semi-bandwidth (b)/block size (f) varies taking values between 8, 16, 32, 64, and 128.
2. Two real-world examples derived from simulation data. These benchmarks test the capabilities of the eigensolvers by diagonalising ring-polymer action Hessian matrices for two distinct physical systems: a Malondialdehyde molecule and a molecular system featuring an interaction between an oxygen atom and a water molecule.

6.2.1 Synthetic benchmarks

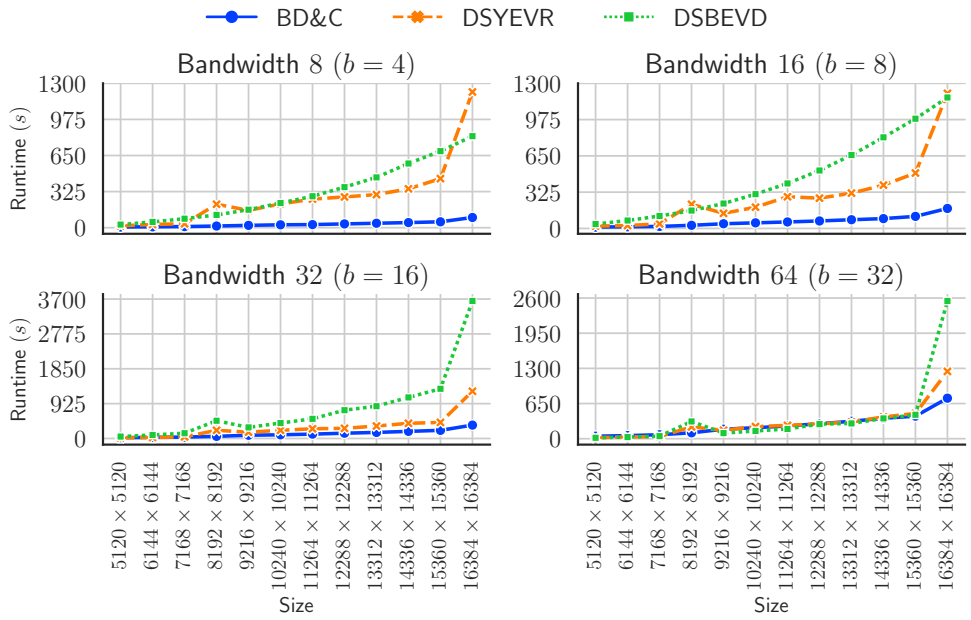

Figure 6.1: Graphs showcasing the runtime of the BD&C, the DSYEVR the DSBEVD eigensolvers for synthetic input matrices. Only the results for matrices of size greater or equal to 5120×5120 are showcased in order highlight the asymptotic behaviour more clearly. A bandwidth of 64 represents the threshold after which the BD&C algorithm underperforms compared to the traditional routines.

Figure 6.1 showcases the asymptotic runtime behaviour of the three eigensolver algorithms for randomly generated input matrices without employing any eigenvalue spectrum approximation techniques. For matrices with very thin bandwidths, the BD&C algorithm displays substantially lower runtimes

compared to both DSYEVR and DSBEVD routines, especially with large matrix sizes.

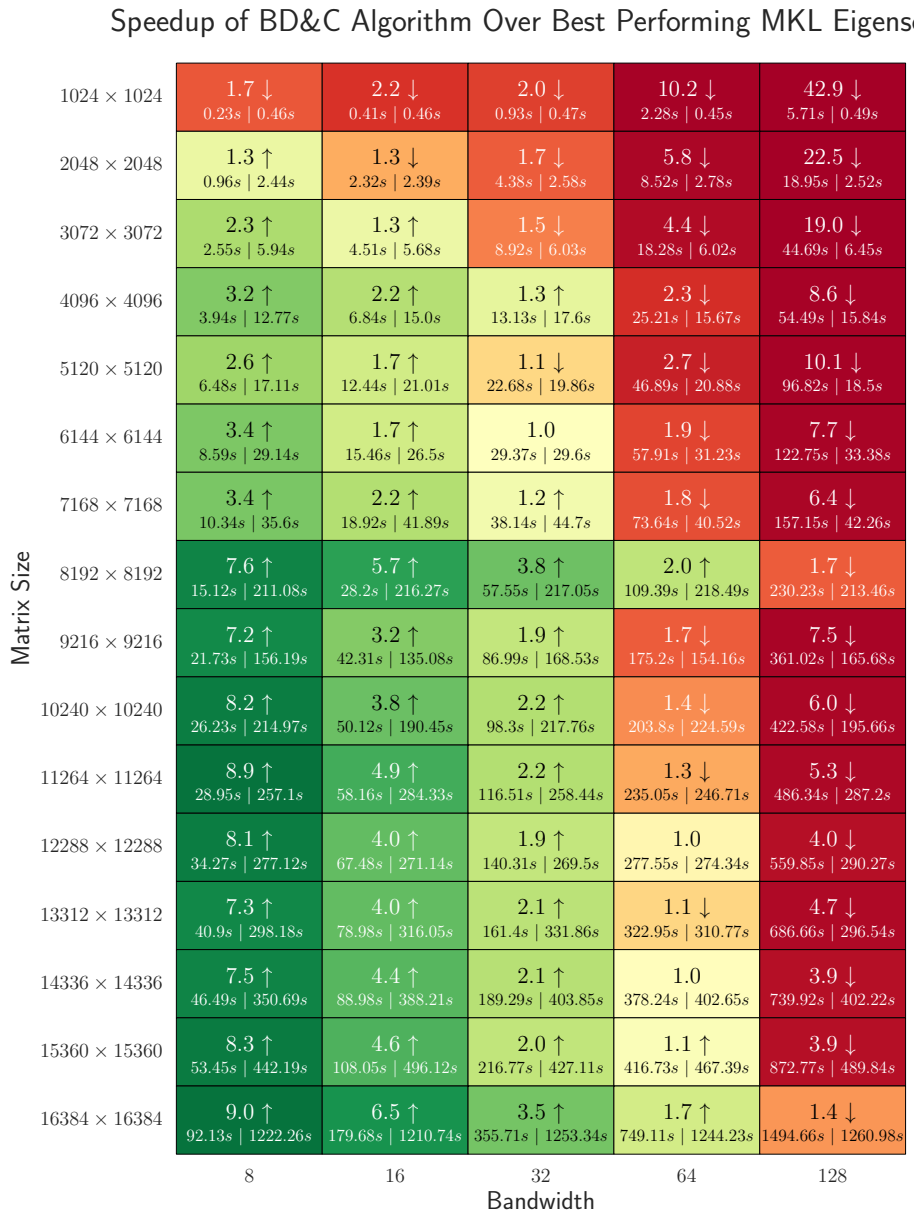

Figure 6.2: Heatmap showcasing the speedup of the BD&C algorithm compared to the best performing routine offered by Intel oneAPI MKL. Each cell contains both the value of the speedup, as well as the runtime in seconds of the BD&C eigensolver (on the left) and the reference baseline set by MKL (on the right).

As expected, increasing the bandwidth of the matrix has a drastic impact on performance and for matrices with bandwidth 64, all three eigensolvers exhibit very similar execution times. Interestingly, although DSBEVD has a much lower theoretical complexity compared to DSYEVR, the results shows that the banded eigensolver never outperforms the dense eigensolver for small bandwidth values. Moreover, the runtime of the DSBEVD routine seems to improve with increasing bandwidth: this is an unexpected result as the number of operations required for tridiagonalisation increases with growing bandwidth. It is difficult to pinpoint the exact reason for this abnormal behaviour, however, the work proposed in [13] suggests that denormalized floating point arithmetic could be the cause of the performance degradation.

Figure 6.2 displays a speedup heatmap of the BD&C algorithm compared to the best performing solver implemented in Intel oneAPI MKL. Each cell contains the speedup value, as well as the runtime in seconds of the two routines (BD&C on the left and Intel MKL on the right). Again, we observe that the BD&C algorithm outperforms Intel oneAPI MKL when handling matrices with extremely narrow bandwidths, particularly in larger test cases, while it faces challenges as the bandwidth increases. We also observe a performance degradation for small matrix sizes. This is likely caused by optimization factors not directly linked with the complexity of the algorithm, such as memory management inefficiencies or overheads in the initial setup of the computational environment.

6.2.2 Real-world benchmarks

Two examples derived from real world simulation data are presented in order to prove the robustness and efficacy of the methods discussed in this work. These example matrices are derived from numerical simulations of instanton theory, as described in [6], where they are repeatedly diagonalised as part of an iterative optimization procedure.

Scientific background

The test data was derived from two distinct physical systems, each comprising a set of connected beads arranged to form a chain, known as a ring-polymer, within a specific potential energy landscape. These ring-polymers represent a discrete model of the most probable path taken by particles when tunneling through the potential energy surface, also known as an instanton trajectory [26]. Figure 6.3 showcases an example of a ring-polymer, depicted as a set of interconnected blue beads, as well as the underlying two-dimensional potential energy surface. Mathematically, each bead is modelled as a harmonic oscillator linked to its immediate neighbours via a set of harmonic springs. The optimization process aims to relax the ring-polymer by minimizing the discretised action in order to calculate the best possible approximation of

the instanton trajectory. Additionally, other properties of interest can be computed, such as transition rates of electrons through the potential energy wall and solutions to tunneling splitting problems, offering insights into the frequency and conditions under which quantum tunneling occurs [6].

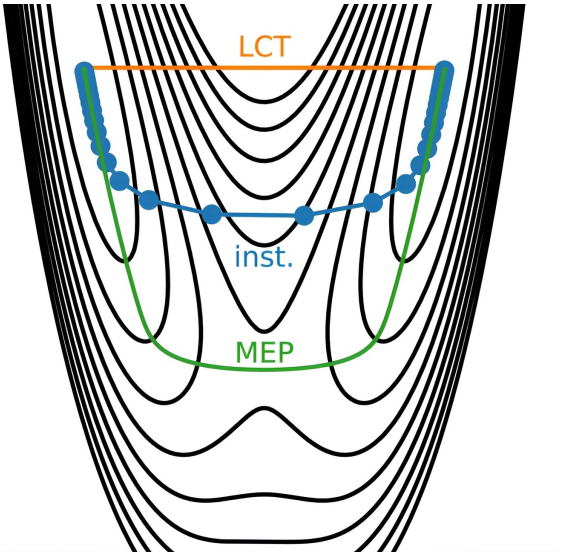

Figure 6.3: Diagram proposed in [26] describing the instanton trajectory (inst.), the minimum-energy path (MEP) and the large-curvature tunneling approximation (LCT). The blue circles represent the beads of the ring-polymer and the black contour lines showcase the potential energy surface.

The Hessian matrix of the discretized ring-polymer action is a tri-block diagonal matrix with full rank off-diagonal blocks. The size f of the diagonal and off-diagonal blocks is determined by the number of degrees of freedom of the system [6]. Moreover, the off-diagonal blocks consist of purely diagonal matrices. Figure 6.4 showcases the structure of the first 5 diagonal blocks of the Hessian matrix for the Malondialdehyde molecule ring-polymer discussed as first real-world test-case. Given that the ring-polymer is modelled as a chain of oscillators linked by harmonic springs, the structure of the Hessian matrix holds specific information about the system. Namely, the diagonal blocks contain the information regarding the resonance frequencies and vibrational modes of the beads, while the off-diagonal blocks encode the information about the coupling between beads, defining how energy and motion are transferred across the chain.

Malondialdehyde Hessian matrix

The first real-world test case is a Hessian matrix derived from the discretized action of a Malondialdehyde molecule ring-polymer. The matrix describes

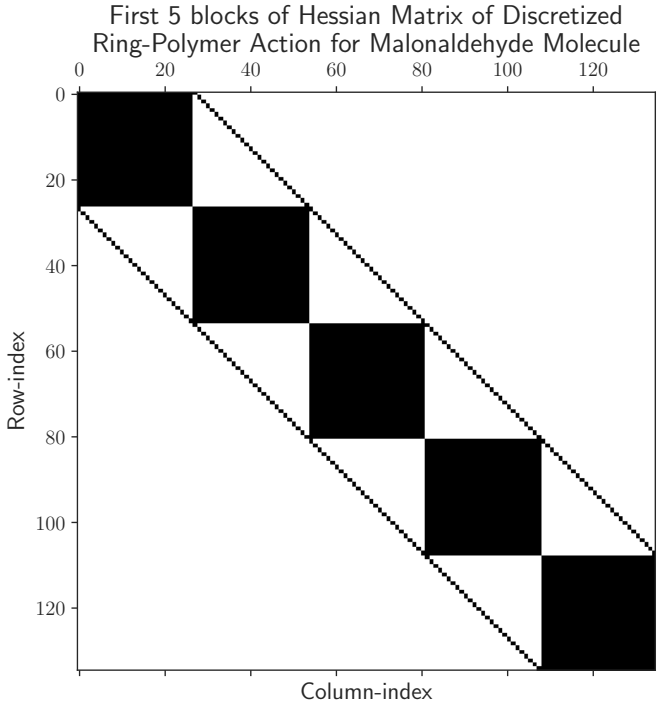

Figure 6.4: Structure of the first 5 diagonal blocks of the Hessian matrix of the discretized Malondialdehyde ring-polymer action. There are 512 diagonal blocks and each block has a size of 27×27. The total matrix size is 13824×13824

a system of 512 beads and as such consists of 512 diagonal blocks of size 27×27, for a total matrix size of 13824×13824. The matrix was diagonalised using both the BD&C algorithm as well as the dense (DSYEVR) and banded (DSBEVD) eigensolver routines offered by Intel oneAPI MKL in order to obtain a performance and accuracy comparison. Different approximated solutions, as described in section 2.7, were also computed and the resulting spectrums were analyzed. Moreover, we define the following modified determinant operator to serve as an aggregate comparison metric for the different approximations:

$$\det'(H) = \sum_{i=2}^{N} \log(|\lambda_i|) \tag{6.1}$$

That is, $\det'(H)$ is the product of the natural logarithm of the magnitude of all eigenvalues of H, except the first. This is necessary for two reasons:

1. The first eigenvalue of the Hessian matrix is know to be exactly zero as it accounts for time translation in the tunneling process, and it is always neglected when computing electron transition rates.

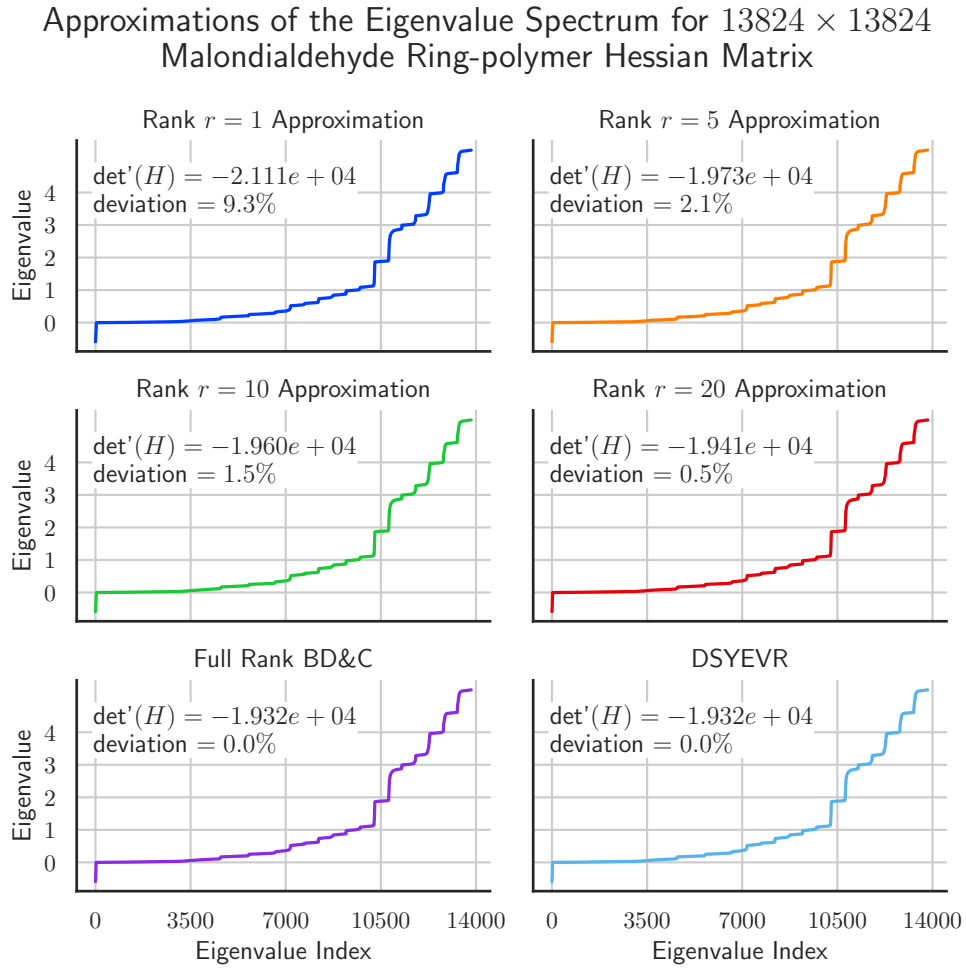

Figure 6.5: Eigenvalue spectrum diagrams of the Malondialdehyde ring-polymer Hessian Matrix. The first four plots showcase approximate solutions computed with rank 1, 5, 10 and 20 approximations of the off-diagonal blocks. The approximation threshold τ was set to 500. The last two plots showcase exact solutions. The values for det$'(H)$ as well as the deviation from the reference solution set by DSYEVR are shown.

2. Given the large problem size, computing the determinant of the Hessian matrix by multiplying the eigenvalues directly would result in a number that is too large to fit in standard double precision datatypes. Instead we opt to compute the sum of the natural logarithms of the eigenvalues which is equivalent to the natural logarithm of the determinant.

Figure 6.5 displays the eigenvalue spectrums obtained from diagonalising the Hessian matrix using different levels of approximation, as well as the full-accuracy solutions computed by the BD&C algorithm and the DSYEVR dense solver routine. These spectrums appear remarkably similar and overall seem to share the same common structure. The values of the modified

determinant are also similar, showing a maximum deviation of less than 10%. The similarity in the results demonstrates that the approximation strategy employed by the BD&C algorithm yields competitive results in terms of accuracy. Even with very aggressive approximation settings, the BD&C algorithm manages to closely match the results calculated by the DSYEVR routine.

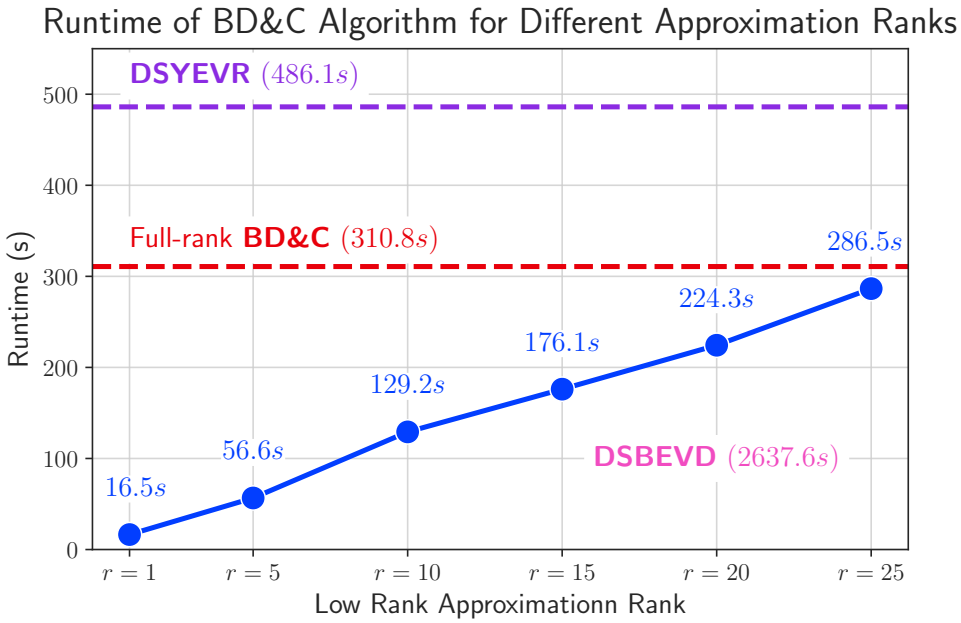

Figure 6.6: Runtime of the BD&C algorithm for the Malondialdehyde ring-polymer Hessian matrix for varying levels of approximation. The values for the full-rank computation as well as the runtime of the DSYEVR routine are marked by horizontal lines. The runtime of the DSBEVD routine is reported as an annotation.

Figure 6.6 displays the the runtime of the BD&C algorithm for different degrees of approximation and includes reference values for the DSYEVR and DSBEVD routines. The BD&C algorithm performed very well in this benchmark, particularly given the large size and the thin bandwidth of the test matrix. The runtime of the full accuracy computation is already 40% lower than the DSYEVR routine, yet approximation can lower it by over an order of magnitude.

Figure 6.7 presents the speedup of the BD&C algorithm over the DSYEVR eigensolver with various approximation levels. The performance demonstrated is quite impressive, with the speedup peaking at nearly 30 times faster than DSYEVR. Not surprisingly, increasing the solver's accuracy decreases its performance. Despite this, as previously discussed, even the most aggressive approximation settings yielded solutions that closely approximated the

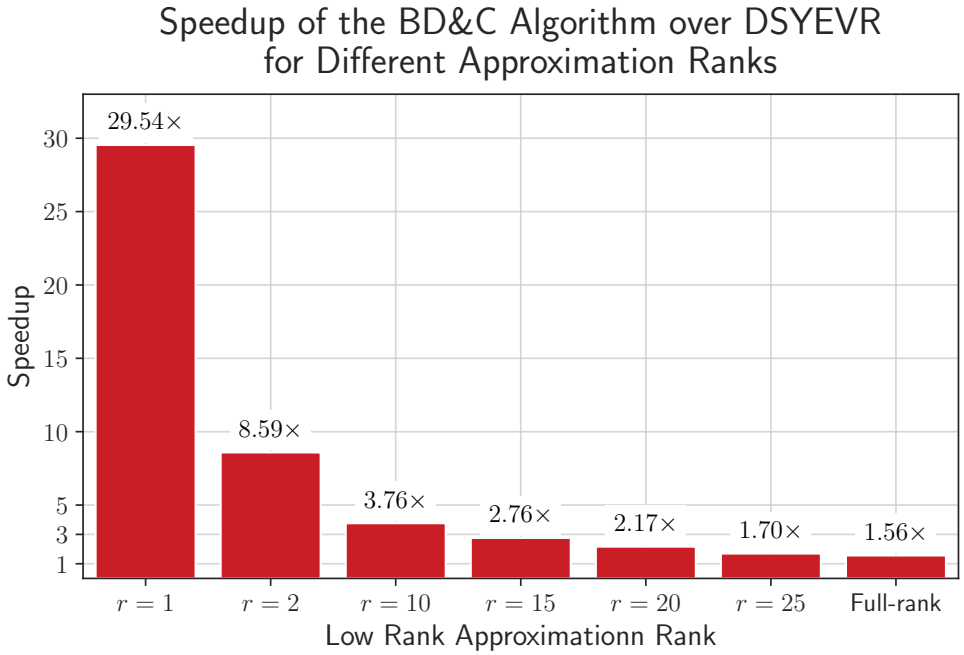

Figure 6.7: Speedup of the BD&C algorithm over DSYEVR for the Malondialdehyde ring-polymer Hessian matrix for different approximation ranks. The DSYEVR routine was chosen as the baseline as it is much faster than the banded eigensolver DSBEVD for this test case.

reference. This balance between speed and accuracy highlights the efficiency of the BD&C algorithm, demonstrating that the algorithm can deliver results with considerable performance, offering a practical solution in scenarios where time efficiency is as crucial as precision.

Oxygen and Water ring-polymer Hessian matrix

The second real-world benchmark is a Hessian matrix derived from the discretized action of a ring-polymer describing the interaction between an oxygen atom with a water molecule. The system comprises 764 beads, but the number of diagonal blocks contained in the Hessian can be halved by exploiting symmetric properties of the ring-polymer. As such, the matrix contains 382 blocks, each of size 27×27, for a total size of 5745×5745. Again, the matrix was diagonalised using both the BD&C algorithm as well as the dense (DSYEVR) and banded (DSBEVD) eigensolver routines offered by Intel oneAPI MKL in order to obtain a performance and accuracy comparison.

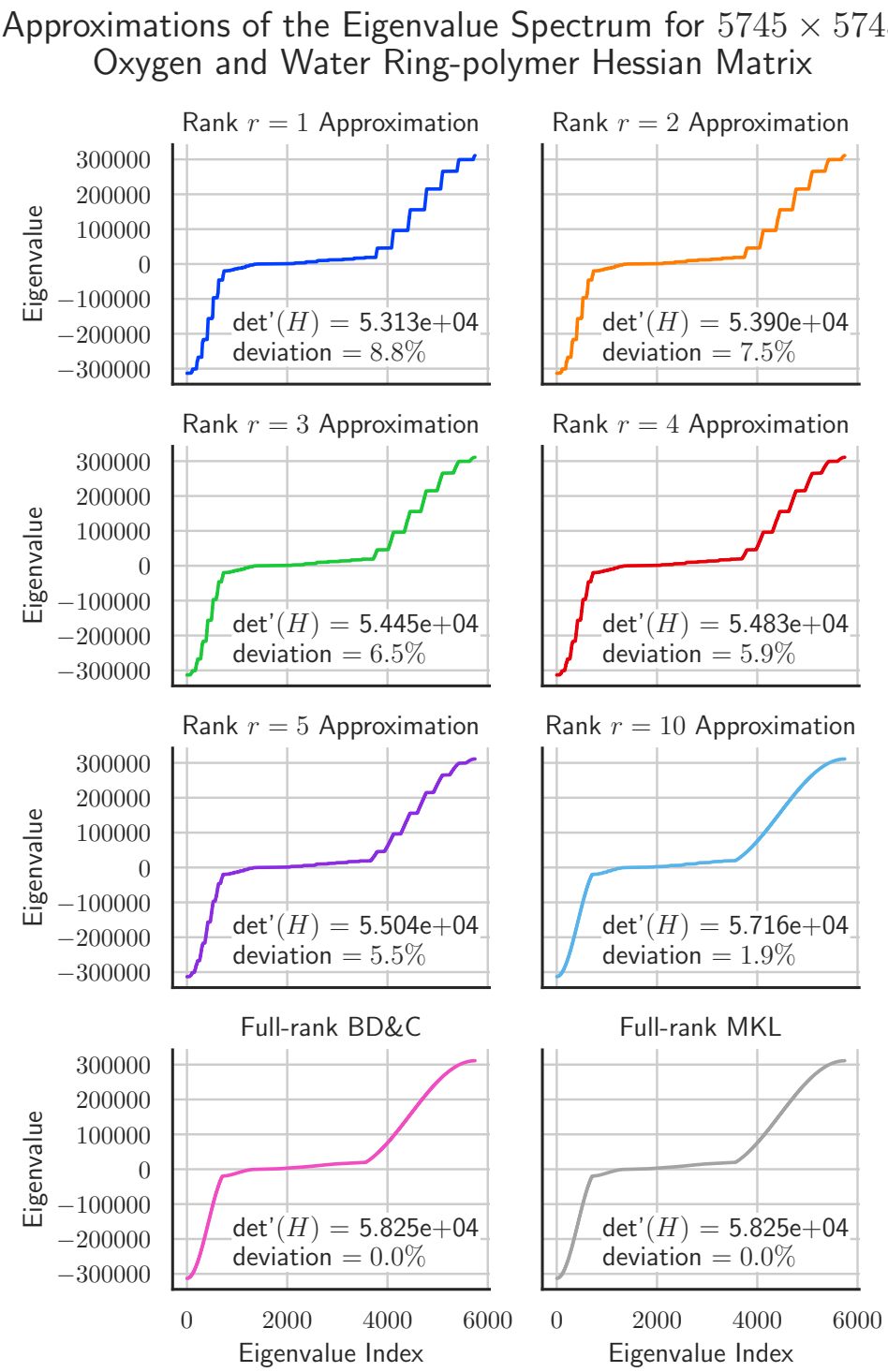

Figure 6.8: Eigenvalue spectrum diagrams of the Oxygen and Water ring-polymer Hessian Matrix. The first six plots showcase approximate solutions, while the last two plots showcase exact solutions. The approximation threshold τ was set to 500. Values for $\det'(H)$ as well as the deviation from the reference solution set by DSYEVR are shown.

Different approximated solutions, as described in section 2.7, were also computed and the resulting spectrums were analyzed. The same modified determinant operator described in the previous section is used as an aggregate value for comparison.

Figure 6.8 displays the eigenvalue spectrums calculated by diagonalising the Hessian matrix using different levels of approximation, as well as the full-accuracy solutions computed by the BD&C algorithm and the DSYEVR dense solver routine. Once again, the spectrums appear remarkably similar and overall seem to share the same common structure. The values of $\det'(H)$ are also similar, showing again a maximum deviation of less than 10%. These results further confirm that the approximation strategy employed by the BD&C algorithm yields competitive results in terms of accuracy, even with very aggressive approximation settings.

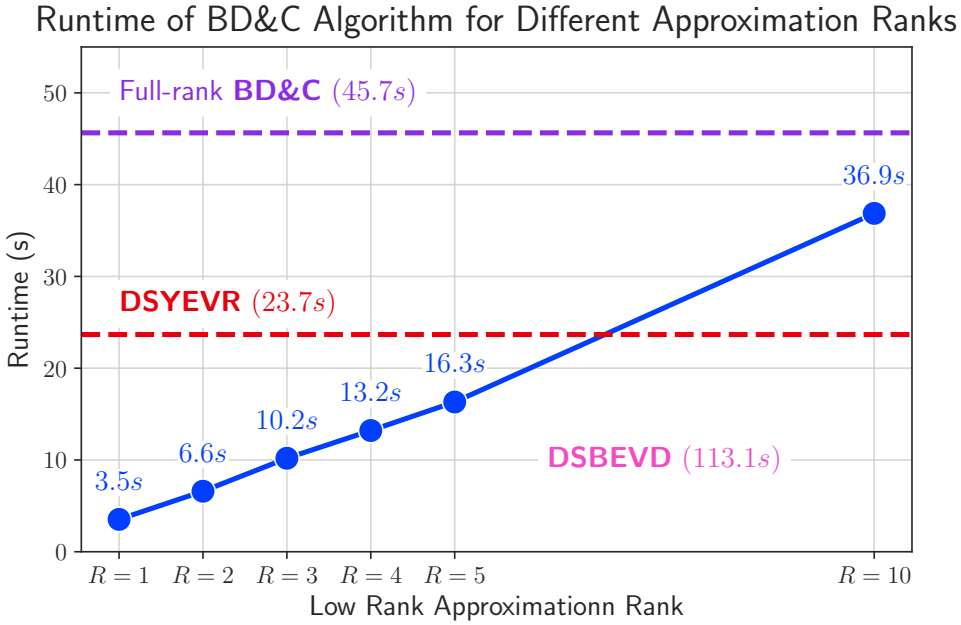

Figure 6.9: Runtime of the BD&C algorithm for the Oxygen and Water ring-polymer Hessian matrix for varying levels of approximation. The values for the full-rank computation as well as the runtime of the DSYEVR routine are marked by horizontal lines. The runtime of the DSBEVD routine is reported as an annotation.

Figure 6.9 shows the runtime of the BD&C algorithm for different levels of approximation and includes comparison data for the DSYEVR and DSBEVD routines. In this test, the BD&C algorithm's performance was decent, but not as impressive as in the previous real-world example. This difference can be attributed to the smaller size of the test matrix. When running at full accuracy, the BD&C algorithm took significantly longer than the DSYEVR

routine. However, in this case as well, approximation significantly lowered the runtime. This suggests that while the BD&C algorithm might be slower for full accuracy diagonalisation of smaller matrices, its capability to speed up calculations through approximation is still an important asset.

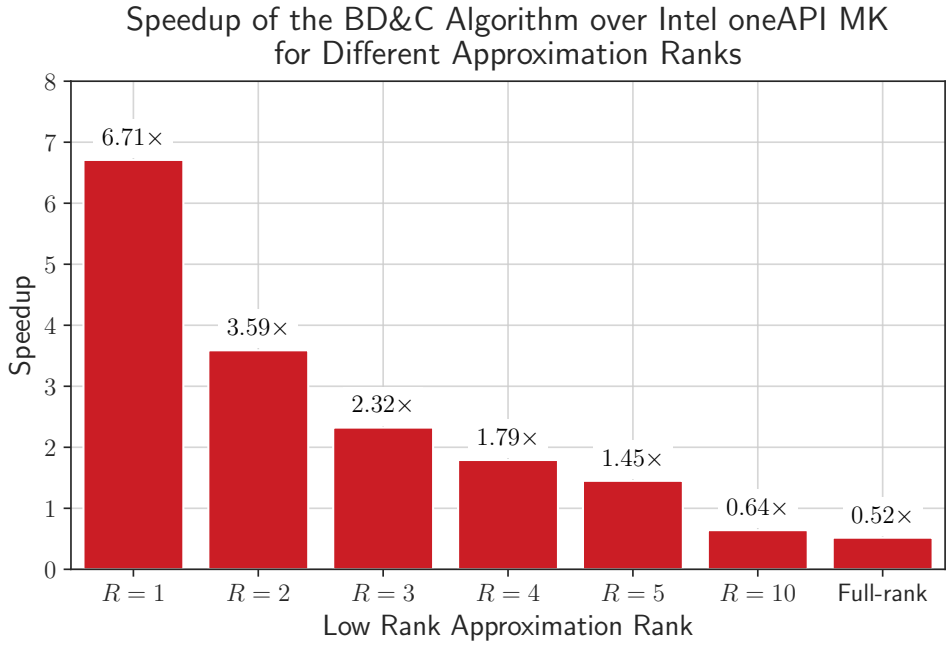

Figure 6.10: Speedup of the BD&C algorithm over DSYEVR for the Oxygen and Water ring-polymer Hessian matrix for different approximation ranks. The DSYEVR routine was chosen as the baseline as it is much faster than the banded eigensolver DSBEVD for this test case.

Figure 6.10 displays the speedup of the BD&C algorithm over the DSYEVR eigensolver with various degrees of approximation. Despite the exact computation being 2× slower, the performance demonstrated for the approximated solutions is still quite impressive, with the speedup peaking at nearly 7 times faster than DSYEVR. Once again, increasing the solver's accuracy decreases its performance, however, considering the reliable results obtained even with aggressive approximations, the BD&C algorithm still proves very attractive for applications where the exact solution is not strictly required.

6.3 Banded and block-tridiagonal broad arrowhead matrices

This section is an adaptation of the benchmarks section proposed in [19], and discusses the results of numerical experiments designed to evaluate the broad arrowhead eigensolver detailed in chapter 4. The empirical data, as

well as the figures, are directly borrowed from the paper. This was done because the benchmarks were performed with the same codebase developed in this work.

6.3.1 Synthetic benchmarks

Two synthetic benchmarks are presented for this algorithm using different optimized methods for $\widetilde{M}$. The first consists of a banded arrowhead matrix: in this case $\widetilde{M}$ was diagonalised using the DSBEVD LAPACK routine. The second is a block-tridiagonal arrowhead matrix with rank-one off-diagonal blocks: in this example $\widetilde{M}$ was diagonalised using the BD&C algorithm. Both benchmarks were executed using matrices with uniformly randomly generated values in the interval between -1 and 1 and with different arrowhead widths g. A more detailed analysis of the performance of this method is available in [19].

Banded Arrowhead Matrices

The first test case consists of a banded arrowhead matrix. Banded broad arrowhead matrices are a direct generalization of arrowhead matrices where both the shaft and the arrowhead are wider than a single element. Figure 6.11 showcases a diagram describing the structure of a banded arrowhead matrix.

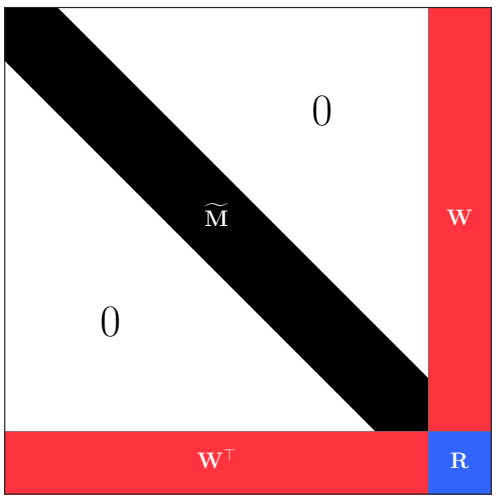

Figure 6.11: Structure of a banded broad arrowhead matrix as described in Eq. 4.1. The shaft $\widetilde{M}$ is a banded matrix.

In this case the matrix $\widetilde{M}$ is banded and as such it is possible to reduce the complexity of diagonalisation from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2 b)$, where b is the semi-bandwidth, by using the routine DSBEVD offered by Intel oneAPI MKL [13].

From a theoretical standpoint, this should yield a considerable performance improvement, especially for small values of b; however empirical measurements proposed in [19] show that this is not always the case. Moreover, the maximum speedup recorded for DSBEVD over DSYEVR was only of about 3.5× for a 10k×10k matrix. Given that AF-based eigendecomposition relies on an optimized eigensolver for $\widetilde{M}$, the bandwidth parameter was set to 600, which was empirically determined to be the optimal bandwidth value in terms of speedup.

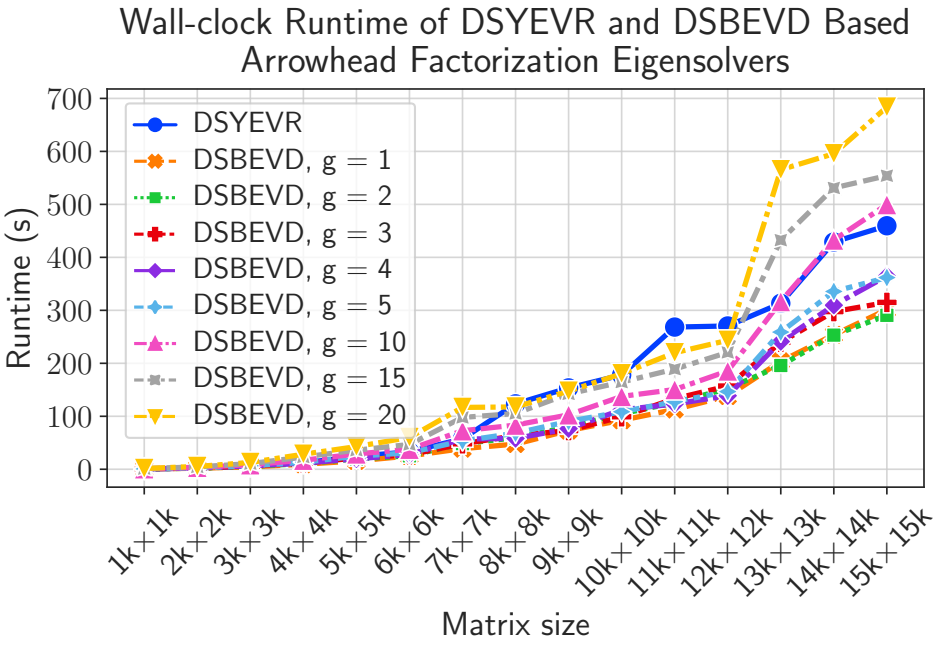

Figure 6.12: Wall-clock runtime of DSYEVR and DSBEVD based AF eigensolver for different arrowhead widths g.

Figure 6.12 displays the runtime for both DSYEVR and DSBEVD based AF. Little if any performance gain is observed for matrices of size up to 7k × 7k, however with growing matrix size we observe an improvement of the performance of the AF-based method, which surpasses that of DSYEVR, especially for small values of g.

Figure 6.13 presents a heatmap illustrating the performance of the DSBEVD AF eigensolver in terms of speedup and slowdown compared to DSYEVR, along with the median runtime and the 95% confidence intervals for the median. Notably, the highest speedup recorded is about 2.5×, with matrix sizes ranging from 8k × 8k to 12k × 12k showing particularly strong performance. Nonetheless, the overall speedup is limited by the maximal speedup achievable by DSBEVD relative to DSYEVR.

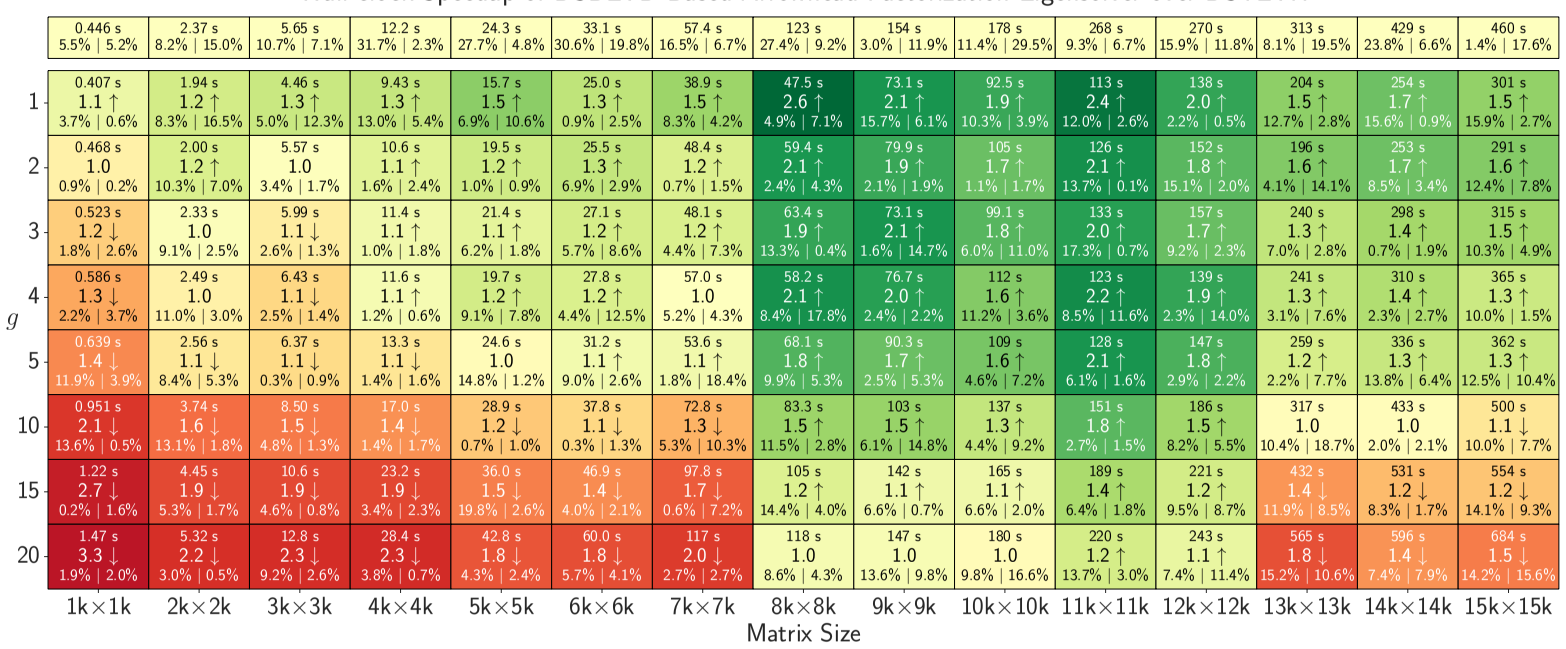

Figure 6.13: Speedup heatmap showcasing the results of the DSBEVD AF eigensolver. Each cell details, from top to bottom, median execution time, speedup over DSYEVR, lower bound and upper bound of the 95% confidence interval of the median. The first row reports the baseline set by DSYEVR.

Block-tridiagonal arrowhead matrices

The second test case consists of a Block-tridiagonal Arrowhead matrix with rank 1 off-diagonal blocks. Block-tridiagonal matrices are a special case of banded arrowhead matrices where the sparsity of the shaft results in a block-tridiagonal structure. In this case, however, the off-diagonal blocks have the additional property of being of rank-1 (or low rank). Figure 6.14 displays an example of the structure of a block-tridiagonal arrowhead matrix.

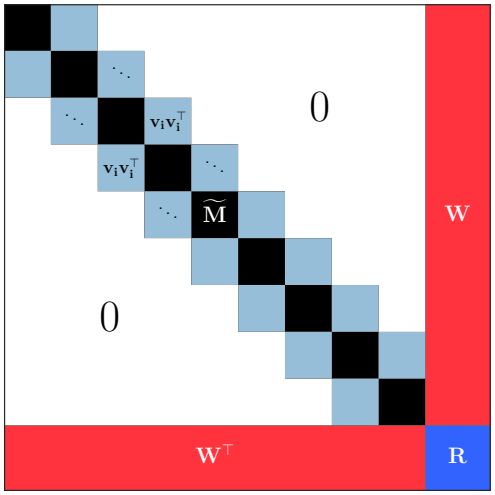

Figure 6.14: Structure diagram of a block-tridiagonal arrowhead matrix. The gray blocks represent the rank 1 off-diagonal blocks. The components of the matrix are named according to Eq. 4.8

Although the eigendecomposition of $\widetilde{M}$ could be computed by treating it as a generic banded matrix, the cost of diagonalisation can be greatly reduced by exploiting the low rank structure of the off-diagonal blocks with the BD&C algorithm. This test case is particularly interesting as it simulates the possible performance gain achievable by using the approximation techniques of the BD&C algorithm together with the AF eigensolver method. Similarly to the previous benchmark, the size of the diagonal and off-diagonal blocks was set to 200, which corresponds to a banded arrowhead matrix with a bandwidth of 600.

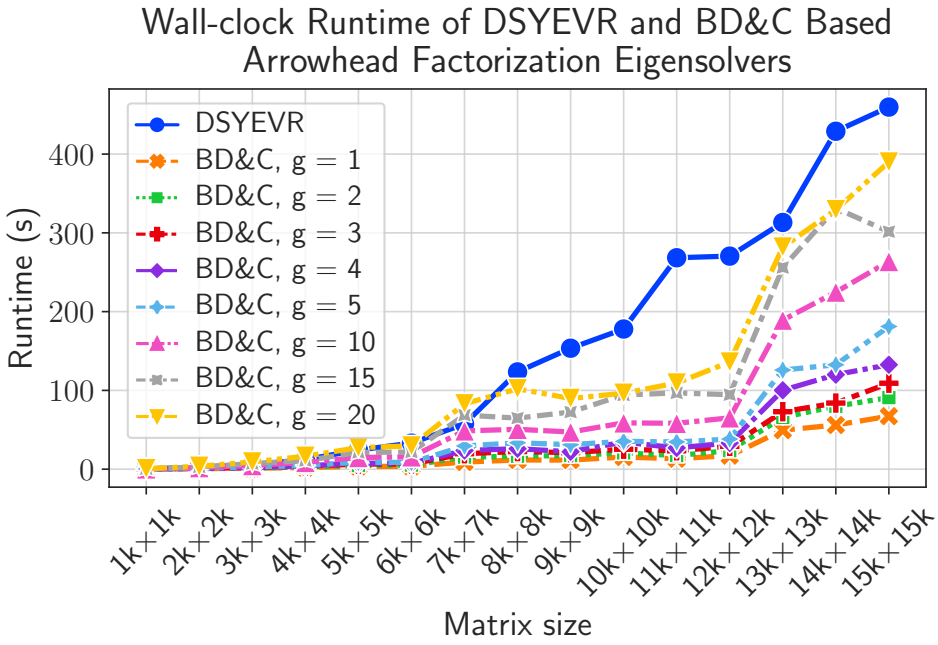

Figure 6.15: Wall-clock runtime of DSYEVR and BD&C based AF eigensolver for different arrowhead widths g.

Figure 6.15 highlights the runtime of the BD&C based AF eigensolver. We observe significant performance improvements across most scenarios, particularly when dealing with matrices of large size and small arrowhead widths. Additionally, the heatmap in Figure 6.16 illustrates the comparative speedup of the BD&C based AF eigendecomposition method against the DSYEVR routine, revealing remarkable execution times that are up to 20× faster. This data provides a clear demonstration of the significant performance gains achievable with this method, underscoring its robustness and its potential to significantly reduce computational times in practical applications, making it a valuable tool for high-performance computing environments and numerical simulations.

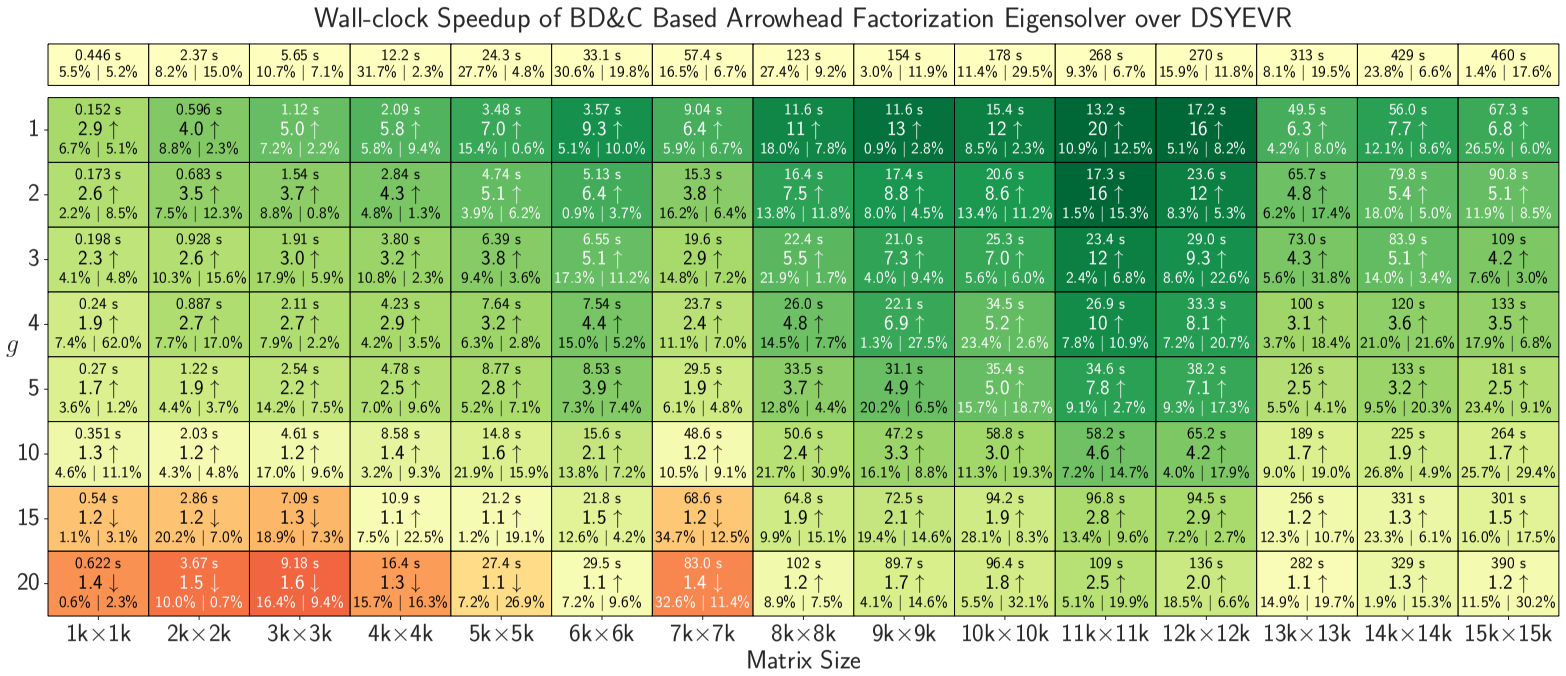

Figure 6.16: Speedup heatmap showcasing the results of the BD&C AF eigensolver. Each cell details, from top to bottom, median execution time, speedup over DSYEVR, lower bound and upper bound of the 95% confidence interval of the median. The first row reports the baseline set by DSYEVR.

Chapter 7

Conclusions

7.1 Conclusions

In this work, we explored different optimized eigendecomposition algorithms, implementing efficient diagonalisation strategies for banded, block-tridiagonal, DPR1, broad arrowhead and arrowhead matrices. We carried out numerical experiments and benchmarks for the BD&C algorithm and the arrowhead factorization method. The BD&C algorithm was tested using synthetic inputs, in order to highlight its performance with different problem sizes and bandwidths, and real-world examples, showcasing its capabilities with practical simulation data. The results were very promising, particularly for matrices with smaller bandwidths and larger sizes. Moreover, our experiments with approximation techniques revealed that the BD&C algorithm can achieve significantly shorter runtimes, showcasing speedups of up to 30x over the best eigensolver offered by the Intel oneAPI MKL library, while keeping the deviation from the exact solution under 10%. This result is significant as it demonstrates the potential of this method for simulations where exact results are not strictly necessary and runtimes are often very long. The arrowhead factorization eigensolver was tested using two distinct synthetic benchmarks, in order to showcase the performance when using the banded eigensolver DSBEVD and the BD&C algorithm back-ends. In general, the results showed that the algorithm was particularly fast with large matrices and small arrowhead widths. Although the performance of AF with DSBEVD was satisfactory, the combination of AF with the BD&C algorithm was much more impressive, demonstrating up to $20\times$ faster runtimes compared to DSYEVR. This work provides a strong argument for the robustness and practicality of the discussed methods, demonstrating that they are a valuable asset in complex numerical simulations. Our analysis and empirical results show the potential of these algorithms to lower computational costs, enabling the exploration of new larger systems. This work adds to the continuous development of computational techniques for eigendecomposition, delivering

both novel theoretical strategies, as well as strong practical results.

7.2 Online Resources

All experiments discussed in this work were conducted using the Freccia eigensolver library, which offers optimized eigensolver algorithms for specific matrix structures [27]. Freccia is built on top of the Eigen C++ library [23] and can use both OpenBLAS [28] and Intel oneAPI MKL backends [5]. The project is still in very early development, however it has already produced very promising results.

7.3 Future work

Developing efficient, stable and robust numerical algorithms is an extremely complex and time consuming task. Given the very promising results produced by this first implementation of the discussed methods, we are convinced that the forthcoming iterations will showcase significant improvements. We are working on making these algorithms publicly available by actively developing the Freccia eigensolver library. We hope that in the future, our work will be easily accessible to scientists and researches in various fields. Lastly, we are also evaluating whether it is possible to modify some of the techniques discussed in this work and apply them to general dense symmetric real matrices. In principle, this can already be done both with the BD&C algorithm and with the AF eigensolver, however, with the current implementation of the code, this results in a complexity of $\mathcal{O}(n^4)$. Thus, to make these methods practically viable for the dense problem, the complexity must be reduced to at least $\mathcal{O}(n^3)$. This would be an interesting achievement as it would introduce a new class of eigendecomposition algorithms for dense matrices, which do not rely on reduction to tridiagonal form.

Acknowledgements

My deepest gratitude goes to Professor Dr. Jeremy Richardson (ETHZ) for welcoming me into his research group, providing the inspiration and insight for this research topic and for allowing me to work on such an problem. I also thank him for the encouragement and for supervising this project.

My sincerest thanks go to Mr. Imaad Ansari (ETHZ) for his mentorship during the full development of this thesis. Our many insightful discussions have been of tremendous help in overcoming challenges and for the general success of this project.

I also thank Mr. Hussein N. el Harake (CSCS/ETHZ) for his technical support when developing and testing the Freccia library.

Finally I thank my family for the continuous support during the three years of my Bachelor's degree.

Bibliography

[1] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37–52, 1987, Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists, ISSN: 0169-7439. DOI: https://doi.org/10.1016/0169-7439(87)80084-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169743987800849.

[2] L. Bandeira and C. C. Ramos, "Non-homogeneous chain of harmonic oscillators," *Mathematics in Computer Science*, vol. 16, no. 1, p. 3, Mar. 18, 2022. DOI: 10.1007/s11786-022-00522-x. [Online]. Available: https://doi.org/10.1007/s11786-022-00522-x.

[3] P. S. Barklem, "Excitation and charge transfer in low-energy hydrogen-atom collisions with neutral atoms: Theory, comparisons, and application to ca," *Phys. Rev. A*, vol. 93, p. 042705, 4 Apr. 2016. DOI: 10.1103/PhysRevA.93.042705. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.93.042705.

[4] E. Anderson *et al.*, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback).

[5] Intel Corporation, *Intel oneAPI Math Kernel Library (MKL)*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, [Online; accessed 18-November-2023], 2023.

[6] J. O. Richardson, "Ring-polymer instanton theory," *International Reviews in Physical Chemistry*, vol. 37, no. 2, pp. 171–216, 2018. DOI: 10.1080/0144235X.2018.1472353. eprint: https://doi.org/10.1080/0144235X.2018.1472353. [Online]. Available: https://doi.org/10.1080/0144235X.2018.1472353.

[7] R. P. Feynman, A. R. Hibbs, and D. F. Styer, *Quantum mechanics and path integrals*. Courier Corporation, 2010.

[8] L. S. Blackford *et al.*, *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997, p. 47. DOI: 10.1137/1.9780898719642. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719642. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719642.

[9] G. Golub and F. Uhlig, "The qr algorithm: 50 years later its genesis by john francis and vera kublanovskaya and subsequent developments," *IMA Journal of Numerical Analysis*, vol. 29, no. 3, pp. 467–485, 2009. DOI: 10.1093/imanum/drp012.

[10] 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. [Online]. Available: https://doi.org/10.1007/BF01396757.

[11] J. D. Rutter, "A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem," Department of Mathematics, University of California at Berkeley, Berkeley, CA 94720, Computer Science Division (EECS) Technical Report UCB/CSD 94/799, 1994.

[12] 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. [Online]. Available: https://doi.org/10.1145/1186785.1186788.

[13] M. Moldaschl and W. N. Gansterer, "Comparison of eigensolvers for symmetric band matrices," *Science of Computer Programming*, vol. 90, pp. 55–66, 2014, Special issue on Numerical Software: Design, Analysis and Verification, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2014.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642314000112.

[14] 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 Transactions on Mathematical Software (TOMS)*, vol. 28, no. 1, pp. 45–58, 2002.

[15] N. K. Kumar and J. Shneider, *Literature survey on low rank approximation of matrices*, 2016. arXiv: 1606.06511 [math.NA].

[16] N. Jakovčević Stor, I. Slapničar, and J. Barlow, "Forward stable eigenvalue decomposition of rank-one modifications of diagonal matrices," *Linear Algebra and its Applications*, vol. 487, pp. 301–315, 2015, ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2015.09.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0024379515005406.

[17] LAPACK Development Team, *LAPACK online documentation - DLAED1 routine*, Accessed: 2023-December-26, 2023. [Online]. Available: https://netlib.org/lapack/explore-html/df/d9e/group__laed1_gac1b3937f595e08f9bab9966fa1a007a5.html#gac1b3937f595e08f9bab9966fa1a007a5.

[18] K. B. Petersen and M. S. Pedersen, *The matrix cookbook*, Version 20121115, Nov. 2012. [Online]. Available: http://www2.compute.dtu.dk/pubdb/pubs/3274-full.html.

[19] M. Ferrari, F. Cavalli, H. N. el Harake, C. Lompa, and N. Lo Russo, "Arrowhead factorization of real symmetric matrices and its applications in optimized eigendecomposition," Manuscript submitted for publication, 2023.

[20] N. Jakovčević Stor, I. Slapničar, and J. L. Barlow, "Accurate eigenvalue decomposition of real symmetric arrowhead matrices and applications," *Linear Algebra and its Applications*, vol. 464, pp. 62–89, 2015, Special issue on eigenvalue problems, ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2013.10.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0024379513006265.

[21] I. M. Ansari, E. R. Heller, G. Trenins, and J. O. Richardson, "Instanton theory for fermi's golden rule and beyond," *Philosophical Transactions of the Royal Society A*, vol. 380, no. 2223, p. 20 200 378, 2022.

[22] J. O. Richardson, "Ring-polymer instanton theory of electron transfer in the nonadiabatic limit," *The Journal of Chemical Physics*, vol. 143, no. 13, p. 134 116, Oct. 2015, ISSN: 0021-9606. DOI: 10.1063/1.4932362. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4932362/15503176/134116_1_online.pdf. [Online]. Available: https://doi.org/10.1063/1.4932362.

[23] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[24] Intel Corporation, *Intel oneapi threading building blocks*, version 2021.7.0, [Online; accessed 22-November-2023], 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[25] T. Gamblin *et al.*, "The Spack Package Manager: Bringing Order to HPC Software Chaos," ser. Supercomputing 2015 (SC'15), LLNL-CONF-669890, Austin, Texas, USA, Nov. 2015. DOI: 10.1145/2807591.2807623. [Online]. Available: https://github.com/spack/spack.

[26] J. O. Richardson, "Perspective: Ring-polymer instanton theory," *The Journal of Chemical Physics*, vol. 148, no. 20, p. 200 901, May 2018, ISSN: 0021-9606. DOI: 10.1063/1.5028352. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.5028352/15541718/200901_1_online.pdf. [Online]. Available: https://doi.org/10.1063/1.5028352.

[27] M. Ferrari, *Freccia Eigensolver Library*, https://github.com/MarcelFerrari/Freccia, [Online; accessed 18-November-2023], 2023.

[28] OpenBLAS Project, *OpenBLAS*, https://www.openblas.net, [Online; accessed 18-November-2023], 2023.

[29] L. Dalcin and Y.-L. L. Fang, "Mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021. DOI: 10.1109/MCSE.2021.3083216.

[30] G. M. J. Barca *et al.*, "Recent developments in the general atomic and molecular electronic structure system," en, *The Journal of Chemical Physics*, vol. 152, no. 15, p. 154 102, Apr. 2020, ISSN: 0021-9606, 1089-7690. DOI: 10.1063/5.0005188. [Online]. Available: http://aip.scitation.org/doi/10.1063/5.0005188 (visited on 06/18/2020).

[31] F. Neese, F. Wennmohs, U. Becker, and C. Riplinger, "The ORCA quantum chemistry program package," *The Journal of Chemical Physics*, vol. 152, no. 22, p. 224 108, Jun. 2020, ISSN: 0021-9606. DOI: 10.1063/5.0004608. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0004608/16740678/224108_1_online.pdf. [Online]. Available: https://doi.org/10.1063/5.0004608.

[32] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[33] L. S. Blackford *et al.*, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997, ISBN: 0-89871-397-8 (paperback).

Appendix A

Development of an MPI load balancing scheduler

An additional unrelated task that was tackled during this project was the design and implementation of an MPI load-balancing scheduler for the ring-polymer instanton simulation code (Instopt) developed by the Theoretical Molecular Quantum Dynamics Group at ETH Zurich. This appendix briefly discusses the most important details of this task.

A.1 Current implementation and limitations of Instopt

The current implementation of the Instopt codebase is written in Python and leverages the mpi4py library [29] to achieve MPI distributed memory parallelism. However in practice, only a very small part of the code benefits from the additional computational resources. Instopt relies on a set of external software for electron calculations, such as GAMESS [30] and ORCA [31]. These external packages are used for point-wise evaluation of energy potentials and gradients, which is an extremely expensive process. To speed up this process, the points are split over multiple MPI ranks, each of which then invokes the required external routines. The rest of the code relies on the NumPy library [32], which does not directly offer support for the shared memory linear algebra routines implemented in ScaLAPACK [33] and Intel oneAPI MKL [5], and as such can only run on a single process. The problem is that, instead of using a single process, these computations are run redundantly by all MPI ranks, significantly increasing the resource usage, degrading performance and increasing power consumption. Given that these simulations can run for days or even weeks, it is extremely important to reduce resource consumption as much as possible by eliminating redundant computations.

A.2 Design of an MPI load balancing scheduler

Currently, Instopt executes the same exact computations on all ranks, except when evaluating potentials and gradients, where the points are split evenly across all ranks. Not only is this inefficient due to the number redundant calculations, but also because this approach is vulnerable to load balancing issues. This is because splitting the inputs evenly across ranks does not guarantee an even distribution of the workload. Figure A.1 illustrates how distributing a work vector with a linear schedule can lead to imbalances among ranks. In this example case, the work-depth becomes 11 units as ranks 1 and 2 will have to idle and wait for rank 3, before continuing with other computations.

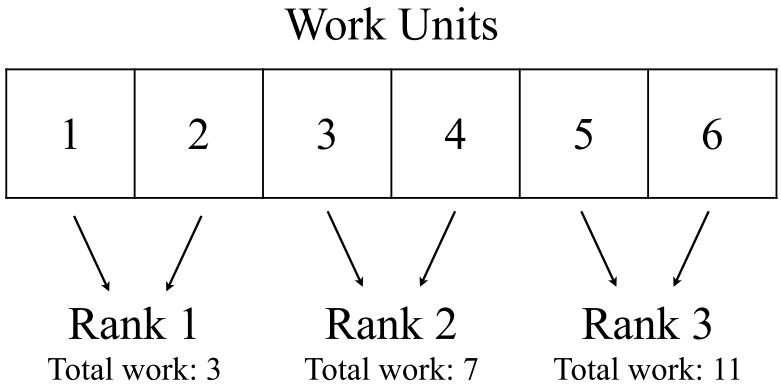

Figure A.1: Linear schedule of tasks. Each cell represents an independent work unit that can be scheduled to a rank.

The goal is thus to design and implement a scheduler object that can satisfy a dual purpose: save resources by eliminating redundant computations and improve load balancing by scheduling tasks in a more intelligent way. The strategy employed to reach this goal is to designate a "master" rank that is responsible for both running the single-process portions of the code, as well as scheduling tasks to "worker" ranks. These worker processes are set to idle until they receive useful work, minimizing the amount of computational resources and power consumed. Listing A.1 shows the compute function called by the master rank in order to wake up worker processes and distribute tasks. Snippet A.2 shows the implementation for the loop function responsible for the idle-work functionality of the worker ranks.

```
def compute(self, tasks):
    # Partition tasks
    jobs = self.partition(tasks)

    # Wake up workers
    self.comm.Barrier()

    # Scatter jobs
    jobs = self.comm.scatter(jobs, root=self.root)

    # Do work
    rax = self.work(jobs)

    # Collect results from worker nodes
    # rax is a list so this reduction concatenates all results.
    rax = self.comm.reduce(rax, root=self.root, op=MPI.SUM)

    return rax
```

Listing A.1: Internal compute function called by the master rank in order to wake up worker processes and distribute tasks.

```
def loop(self):
    # Work loop
    while True:
        # Wait for signal from master rank
        self.comm.Barrier()

        # Get tasks
        tasks = self.comm.scatter(None, root=self.root)

        # Check for termination signal
        if isinstance(tasks, TerminateScheduler):
            break

        # Do work
        # rax is a list of results computed by this rank.
        rax = self.work(tasks)

        # Return results
        self.comm.reduce(rax, root=self.root, op=MPI.SUM)
```

Listing A.2: Internal loop function that implements idle-work functionality for worker-ranks.

A.2.1 Scheduling strategies

Three different scheduling strategies were implemented in order to improve the load balancing capabilities of the scheduler.

Linear schedule

The linear schedule that was already implemented in the code was ported to the scheduler class as a extensively tested and trusted fail-safe option. As previously discussed, this schedule is not the most efficient as it partitions the tasks uniformly across ranks.

Random schedule

This strategy is similar to linear scheduling in the sense that each rank receives the same number of tasks, however it attempts to minimize the work-depth by randomizing the order in which the tasks are assigned to each rank.

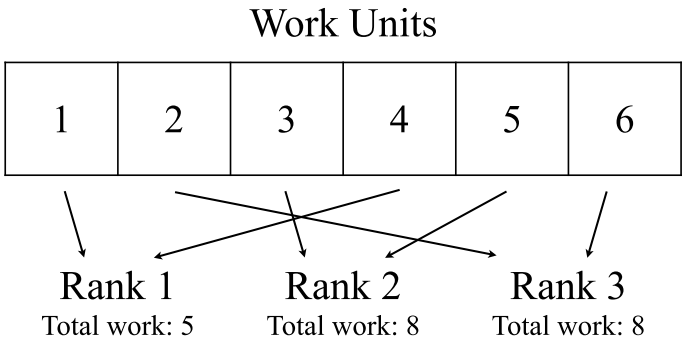

Figure A.2: Random schedule of tasks. Each cell represents an independent work unit that can be scheduled to a rank.

Figure A.2 shows an example of how this strategy might schedule a work vector. Despite each rank still receiving two tasks, the work-depth is reduced from 11 to 8 by rearranging the order in which they are assigned. Although random scheduling does not offer any strict guarantees, it is a simple heuristic that solves the problem of using a fixed schedule with very little computational overhead.

A.2.2 Greedy schedule

The greedy schedule is fundamentally different from the other scheduling strategies, as it does not divide the number of tasks evenly across ranks. Instead, the master rank is sacrificed and assumes a purely coordinative function by interactively scheduling work. Whenever a worker rank is done with a task, it communicates the result to the master rank, which replies with a new task as soon as there is useful work to do. Figure A.3 shows an

illustration of this process. Even if rank 1 is not doing any useful work, the work-depth is still much lower compared to the linear schedule and almost identical to the random schedule.

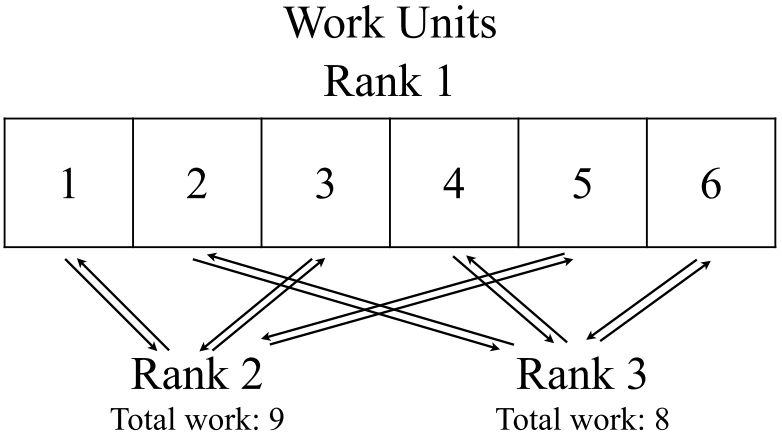

Figure A.3: Greedy schedule of tasks. Each cell represents an independent work unit that can be scheduled to a rank.

This scheduling strategy requires more complex logic in order to handle the master-worker relationship between ranks. Snippet A.3 shows the implementation of the compute function called by the master rank, while listing A.4 displays the code for the loop function called by the worker processes.

A.3 Controlling the scheduler

Work is submitted to the scheduler via the `MPIScheduler.submit` method, which accepts a function and optionally additional arguments to be passed during evaluation. The returned value is an `MPIFuture` object, which wraps the result that will be available later. Once the tasks have been submitted, the `MPIScheduler.compute` method should be called in order to execute the workload. Only after this, it is possible to retrieve the results from the `MPIFuture` objects by calling the `MPIFuture.results` function.

```
def compute(self, tasks):
    rax = []
    ntasks = len(tasks)

    # Start worker pool and schedule at least one task per rank
    for worker in range(min(len(tasks), self.size)):
        if worker != self.root:
            task = tasks.pop()
            # No need to store request
            self.comm.isend(task, dest=worker)

    # Schedule additional work if needed
    while tasks:
        # Receive result from any rank
        status = MPI.Status()
        result = self.comm.recv(source=MPI.ANY_SOURCE,
    status=status)
        rax.append(result)

        # Schedule next task to the worker that just finished
        task = tasks.pop()
        # No need to store request
        self.comm.isend(task, dest=status.Get_source())

    # Receive the remaining results
    reqs = []
    for _ in range(len(rax), ntasks):
        reqs.append(self.comm.irecv(source=MPI.ANY_SOURCE))

    # Wait for all irecvs and append results to rax
    results = MPI.Request.waitall(reqs)
    rax.extend(results)

    return rax
```

Listing A.3: Internal compute function called by the master rank in order to wake up worker processes and distribute tasks.

```
def loop(self):
        while True:
            # Wait for task from master rank
            task = self.comm.recv(source=self.root)

            # Check for termination signal
            if isinstance(task, TerminateScheduler):
                break

            # Do work
            result = self.work(task)

            # Return result to master worker
            self.comm.isend(result, dest=self.root)
```

Listing A.4: Internal loop function that implements idle-work functionality for worker-ranks.

A.4 Lazy evaluation

High level functions are often expressed as a series of calls to independent lower level functions. In order to enable automatic and efficient scheduling of multiple high level computations, a lazy evaluation strategy was implemented. This was done by introducing the `@lazy` decorator, which is used to wrap functions that can be both called lazily or not. For example, the `bead_potentials` method from listing A.5 can be called both directly, in which case it should be evaluated immediately, or as part of the `bead_potentials_gradients_and_hessians` method, in which case the computation should be delayed until the callee function returns. Methods decorated with `@lazy` inherit a new optional `lazy` argument, which controls lazy evaluation. By default, the scheduler computes and unwraps the results as soon as the called function returns.

```
@lazy
def bead_potentials(self, x):
    return scheduler.submit_multiple(self.PES.potential, x)

@lazy
def bead_gradients(self, x):
    return scheduler.submit_multiple(self.PES.gradient, x)

@lazy
def bead_hessians(self, x):
    return scheduler.submit_multiple(self.PES.hessian, x)

@lazy
def bead_gradients_and_hessians(self, x):
    return self.bead_gradients(x, lazy=True),
           self.bead_hessians(x, lazy=True)

@lazy
def bead_potentials_and_gradients(self, x):
    return self.bead_potentials(x, lazy=True),
           self.bead_gradients(x, lazy=True)

@lazy
def bead_potentials_gradients_and_hessians(self, x):
    return self.bead_potentials(x, lazy=True),
           self.bead_gradients(x, lazy=True),
           self.bead_hessians(x, lazy=True)
```

Listing A.5: Examples of lazy methods implemented in the Instopt code.

Lazy methods should always and only be called with `lazy=True` from within other lazy methods, otherwise they will return `MPIFuture` objects that wrap the results.