UMEÅ UNIVERSITY

# Robust Parallel Eigenvector Computation for the Non-Symmetric Eigenvalue Problem

**Angelika Schwarz, Carl Christian Kjelgaard Mikkelsen, Lars Karlsson**

**UMINF 20.02**

Department of Computing Science

ISSN 0348-0542

# Robust Parallel Eigenvector Computation For the Non-Symmetric Eigenvalue Problem

Angelika Schwarz, Carl Christian Kjelgaard Mikkelsen, Lars Karlsson

*Department of Computing Science, Umeå University, Sweden*

**Abstract**

A standard approach for computing eigenvectors of a non-symmetric matrix reduced to real Schur form relies on a variant of backward substitution. Backward substitution is prone to overflow. To avoid overflow, the LAPACK eigenvector routine DTREVC3 associates every eigenvector with a scaling factor and dynamically rescales an entire eigenvector during the backward substitution such that overflow cannot occur. When many eigenvectors are computed, DTREVC3 applies backward substitution successively for every eigenvector. This corresponds to level-2 BLAS operations and constitutes a bottleneck. This paper redesigns the backward substitution such that the entire computation is cast as tile operations (level-3 BLAS). By replacing LAPACK's scaling factor with tile-local scaling factors, our solver decouples the tiles and sustains parallel scalability even when a lot of numerical scaling is necessary.

**Keywords:** Overflow protection, eigenvectors, real Schur form, tiled algorithms

## 1 Introduction

Eigenvalue problems are essential to many applications. In continuum mechanics, the principal stresses and the principal directions are the eigenvalues and eigenvectors of the stress tensor. In quantum mechanics, the eigenvalues are the possible energy levels and the eigenvectors determine the corresponding spatial probability distributions. The distribution of eigenvalues and eigenvectors is important in random matrix theory, which is used in finance and risk management. In machine learning, eigenvalues and eigenvectors are used in the principal component analysis to reduce the dimensionality of data. In vibration analysis, the eigenvalues are the natural frequencies and the eigenvectors are the mode shapes. In particular, the design and construction of automobiles, trains, ships, aircrafts, buildings and bridges hinges on our ability to solve eigenvalue problems.

Eigenvalues and eigenvectors can be very sensitive to small changes in the original matrix. The theory is particularly clear when $(\lambda, \boldsymbol{x})$ is a simple eigenpair of the matrix $\boldsymbol{A}$ [1]. In this case, there is a non-singular matrix $\begin{bmatrix}\boldsymbol{x} & \boldsymbol{X}\end{bmatrix}$ with inverse $\begin{bmatrix}\boldsymbol{y} & \boldsymbol{Y}\end{bmatrix}^H$ such that

$$\begin{bmatrix}\boldsymbol{y}^H \\ \boldsymbol{Y}^H\end{bmatrix} \boldsymbol{A} \begin{bmatrix}\boldsymbol{x} & \boldsymbol{X}\end{bmatrix} = \begin{bmatrix}\lambda & \boldsymbol{0}^H \\ \boldsymbol{0} & \boldsymbol{M}\end{bmatrix}. \tag{1}$$

Here the superscript $H$ denotes the conjugate transpose. If $\boldsymbol{E}$ is a sufficiently small perturbation, then $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{E}$ has a simple eigenpair $(\tilde{\lambda}, \tilde{\boldsymbol{x}})$ which tends to $(\lambda, \boldsymbol{x})$ as $\boldsymbol{E}$ tends to zero. In fact, the first-order expansion of the eigenpair $(\lambda, \boldsymbol{x})$ with respect to the perturbation $\boldsymbol{E}$ is

$$(\tilde{\lambda}, \tilde{\boldsymbol{x}}) \simeq (\lambda + \boldsymbol{y}^H \boldsymbol{E}\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{X}(\lambda \boldsymbol{I} - \boldsymbol{M})^{-1} \boldsymbol{Y}^H \boldsymbol{E}\boldsymbol{x}). \tag{2}$$

We observe that

$$|\lambda - \tilde{\lambda}| \lessapprox \kappa_2(\lambda) \|\boldsymbol{E}\|_2 \tag{3}$$

where the spectral condition number of $\lambda$ is given by

$$\kappa_2(\lambda) = \left( \frac{\boldsymbol{x}^H \boldsymbol{y}}{\|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2} \right)^{-1}. \tag{4}$$

When seeking to bound the error on $\lambda$, a very reasonable first step is therefore to compute both a left eigenvector $\boldsymbol{y}$ and a right eigenvector $\boldsymbol{x}$ and then compute the spectral condition number $\kappa_2(\lambda)$. However, there exist matrices for which the components of each eigenvector grow rapidly and eventually exceed the representational floating-point range. We say that a solver is *robust* if, given a valid input, all intermediate and final results are in the representational range. In particular, a robust solver delivers a final result which is free from infinities (overflows) and NaNs and can therefore be evaluated in the context of the user's application. Since eigenvectors are only determined up to a scaling it is possible to avoid overflow by systematically scaling the eigenvectors during the computation. Indeed, this is the strategy used by LAPACK's eigenvector solvers xTREVC3 [2].

This paper adds overflow protection to the non-robust tiled algorithms for computing eigenvectors of non-symmetric matrices reduced to real Schur form presented in [3]. By adding overflow protection, the same set of problems is now solvable as by DTREVC3. This LAPACK routine expresses a significant part of the computation with level-2 BLAS operations. Our robust tiled algorithms, by contrast, cast the entire eigenvector computation as level-3 BLAS operations such that the algorithms run efficiently in parallel.

In the following, we summarize how eigenvectors can be computed and analyze what problems can be encountered during the computation. The standard eigenvalue problem aims at finding an eigenvalue $\lambda$ and an eigenvector $\boldsymbol{y} \neq \boldsymbol{0}$ so that $\boldsymbol{A}\boldsymbol{y} = \lambda \boldsymbol{y}$ holds. Throughout this paper we assume that the matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is not symmetric. Eigenvalues and eigenvectors are commonly computed through a three-stage procedure [4]. The first stage reduces the matrix $\boldsymbol{A}$ to Hessenberg form $\boldsymbol{H} = \boldsymbol{Q}_0^T \boldsymbol{A} \boldsymbol{Q}_0$, where $\boldsymbol{Q}_0 \in \mathbb{R}^{n \times n}$ is an orthogonal matrix.

Th Hessenberg reduction reduces the computational cost of the second stage: the QR algorithm, which is an iterative procedure that reduces $\boldsymbol{H}$ further with similarity transformations. In real arithmetic, the QR algorithm converges to upper quasi-triangular form instead of upper triangular form when using complex arithmetic. The resulting real Schur decomposition of $\boldsymbol{A}$

$$\boldsymbol{Q}^T \boldsymbol{A} \boldsymbol{Q} = \boldsymbol{T} = \begin{bmatrix} \boldsymbol{T}_{11} & \boldsymbol{T}_{12} & \dots & \boldsymbol{T}_{1m} \\ & \boldsymbol{T}_{22} & \dots & \boldsymbol{T}_{2m} \\ & & \ddots & \vdots \\ & & & \boldsymbol{T}_{mm} \end{bmatrix} \tag{5}$$

is obtained, where $\boldsymbol{Q}$ accumulates all orthogonal transformations. The eigenvalues are given by the blocks $\boldsymbol{T}_{kk}$, either 1-by-1 or 2-by-2, on the diagonal of $\boldsymbol{T}$. A 1-by-1 block corresponds to a real eigenvalue, whereas a 2-by-2 block corresponds to a complex conjugate pair of eigenvalues.

The third stage addresses the eigenvector computation and will be the focus of this paper. A standard approach to computing an eigenvector $\boldsymbol{y}_k$ that corresponds to a simple eigenvalue $\lambda_k$ is derived as follows. Substituting $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T$ into $\boldsymbol{A}\boldsymbol{y}_k = \lambda_k \boldsymbol{y}_k$ gives us $\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T\boldsymbol{y}_k = \lambda_k \boldsymbol{y}_k$. We multiply with $\boldsymbol{Q}^T$ from the left and obtain $\boldsymbol{T}(\boldsymbol{Q}^T\boldsymbol{y}_k) = \lambda_k(\boldsymbol{Q}^T\boldsymbol{y}_k)$. With $\boldsymbol{x}_k := \boldsymbol{Q}^T\boldsymbol{y}_k$, the problem problem is to find $\boldsymbol{x}_k \neq \boldsymbol{0}$ such that

$$(\boldsymbol{T} - \lambda_k \boldsymbol{I})\boldsymbol{x}_k = \boldsymbol{0}, \tag{6}$$

which is possible since the matrix is singular by the definition of $\lambda_k$. A non-trivial solution $\boldsymbol{x}_k$ can be computed through a modified backward substitution method that can handle the 2-by-2 blocks on the diagonal of $\boldsymbol{T}$. We refer to this step as the *backsolve*. Finally, an eigenvector of the original matrix $\boldsymbol{A}$ is obtained through the *backtransform* $\boldsymbol{y}_k \leftarrow \boldsymbol{Q}\boldsymbol{x}_k$.

The backsolve starts with a block partitioning of (6):

$$\left( \begin{bmatrix} \boldsymbol{T}_{11} & \boldsymbol{t}_{12} & \boldsymbol{T}_{13} \\ & \lambda_k & \boldsymbol{t}_{23}^T \\ & & \boldsymbol{T}_{33} \end{bmatrix} - \lambda_k \boldsymbol{I} \right) \begin{bmatrix} \boldsymbol{x}_1 \\ 1 \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{0}. \tag{7}$$

The 2nd and 3rd block equations are satisfied and the 1st can be satisfied by solving for $\boldsymbol{x}_1$ in

$$(\boldsymbol{T}_{11} - \lambda_k \boldsymbol{I})\boldsymbol{x}_1 = -\boldsymbol{t}_{12}. \tag{8}$$

(The coefficient matrix is non-singular by the assumption that $\lambda_k$ is a simple eigenvalue.) We thus obtain an admissible solution to (6). A basic scalar algorithm for solving (8) in complex arithmetic is listed in Algorithm 1.

**Algorithm 1:** Solution of (8) (complex arithmetic)

```
x_{1:k-1} ← −t_{12};
for j ← k − 1, k − 2, ..., 1 do
    x_j ← x_j/(t_jj − λ_k);
    x_{1:j-1} ← x_{1:j-1} − t_{1:j-1,j} x_j;
```

As stated by Stewart [1] this very basic algorithm suffers from two fundamental problems.

**Multiple eigenvalues** If $\boldsymbol{T}$ has a multiple eigenvalue, then there exists $\ell < k$, such that $\lambda_\ell = \lambda_k$. When we attempt to compute an eigenvector corresponding to $\lambda_k$, we encounter a division by zero when attempting to compute $x_\ell$. In LAPACK, if a diagonal entry $t_{jj} - \lambda_k$ drops below a certain threshold, it is replaced by a small non-zero number before applying a robust substitution algorithm. We should always proceed with caution in the presence of multiple eigenvalues. It is straightforward to verify that a matrix which has a multiple eigenvalue is close to a matrix which has an ill-conditioned eigenvalue. Conversely, a matrix which has an ill-conditioned eigenvalue is close a matrix which has a multiple eigenvalue [5].

**Overflow** Substitution is vulnerable to overflow and the worst case behavior exhibits exponential growth [6]. If there are many eigenvalues near the eigenvalue $\lambda_k$, then the solution of (8) will involve many small diagonal entries and overflow is a distinct possibility. However, clustering is not a necessary condition for rapid growth as there exist matrices with well-separated eigenvalues for which the eigenvectors grow rapidly and almost all eigenvectors exceed the representational range. A specific example is exhibited in Section 3 and used in our numerical experiments.

Since the eigenproblem (6) permits any scaling of the eigenvector, overflow can be avoided by rescaling the vector. LAPACK contains the robust eigenvector routine xTREVC3. This routine solves a scaled version of (8):

$$(\boldsymbol{T}_{11} - \lambda_k \boldsymbol{I})\hat{\boldsymbol{x}}_1 = \xi_k(-\boldsymbol{t}_{12}).$$

The scalar $\xi_k \in (0, 1]$ is computed dynamically alongside $\hat{\boldsymbol{x}}_1$ and allows for avoiding overflow. The resulting (scaled) eigenvector is given by

$$\begin{bmatrix} \hat{\boldsymbol{x}}_1 \\ \xi_k^{-1} \\ \boldsymbol{0} \end{bmatrix} = \xi_k^{-1} \begin{bmatrix} \boldsymbol{x}_1 \\ 1 \\ \boldsymbol{0} \end{bmatrix}.$$

The scaling factor effectively extends the class of systems that can be solved with floating-point arithmetic. As a consequence, the eigenvectors necessary to evaluate the spectral condition number (4) can be computed in order to assess what accuracy can be expected from the computed solution.

The main contribution of this paper is a robust, level-3 BLAS-based, scalable algorithm for computing eigenvectors in real arithmetic from the real Schur form. While this does not solve the problem of ill-conditioned eigenvectors, the user will always be able to evaluate what accuracy can be expected based on the corresponding spectral condition numbers.

Our algorithms can solve the same class of problems as LAPACK's DTREVC3 while using mostly level-3 BLAS operations and expressing numerical scaling such that parallel scalability is not impeded. We highlight the following advancements.

- Every statement that can potentially overflow is guarded with overflow protection logic.
- LAPACK's DTREVC3 [2] expresses the backsolve with level-2 BLAS operations. This bottleneck has been removed by expressing the entire computation with level-3 BLAS operations. To the best of our knowledge, only Elemental's TriangEig [7, 8] expresses the backsolve in terms of level-3 BLAS operations. However, TriangEig can handle only triangular matrices, which restricts the routine to either complex arithmetic or to real matrices with only real eigenvalues. In contrast, our implementation allows the eigenvectors to be computed from the real Schur form using real arithmetic.
- The usage of global scaling factors yields a bottleneck in a parallel implementation. By using local scaling factors, this bottleneck is removed.

The rest of this paper is structured as follows. Section 2 presents how eigenvectors can be computed without numerical scaling. Section 3 exhibits a class of matrices for which almost all eigenvectors exceed the representational range. Robust algorithms will be developed and parallelized with tasks in Section 5. For this purpose, Section 4 introduces the concept of scaling factors and dynamic scaling to avoid overflow. Overflow protection logic that renders the algorithms robust is presented. The robust task-parallel eigenvector solver is extended to distributed memory systems in Section 6. Section 7 compares our approach with existing eigenvector solvers. Section 8 describes the numerical experiments, whose results are presented in Section 9.

## 2 Non-robust backsolve in real arithmetic

Algorithm 1 uses complex arithmetic to solve (8) using regular backward substitution. But when working with the real Schur form, the coefficient matrix can be quasi-triangular with 2-by-2 blocks on the main diagonal. In this section, we describe the modifications of the algorithm which are necessary to handle this complication. While real arithmetic makes the computation more complicated, it saves flops and memory compared to reverting to complex arithmetic. A real matrix can have both real and complex eigenvalues and, in turn, real and complex eigenvectors. In real arithmetic, the zero imaginary parts of real eigenvectors need neither be stored nor processed, which saves both memory and flops.

In real arithmetic, $\boldsymbol{T}$ is upper quasi-triangular and has 1-by-1 and 2-by-2 blocks on the diagonal. Let $\lambda_k = t_{kk}$ be a real eigenvalue associated with the 1-by-1 block $t_{kk}$. A corresponding eigenvector can be obtained by solving (8). This is done by a modified backward substitution method that can handle 2-by-2 blocks on the diagonal of $\boldsymbol{T}_{11}$. Every encounter with a 2-by-2 block requires the solution of a 2-by-2 real-valued linear system. We solve these systems through Gaussian elimination with complete pivoting.

Every 2-by-2 block representing a complex conjugate pair of eigenvalues can be normalized[1] by an orthogonal transformation such that it attains the format

$$\begin{bmatrix} \alpha & \beta \\ \gamma & \alpha \end{bmatrix} \in \mathbb{R}^{2\times 2}, \tag{9}$$

where $\beta$ and $\gamma$ have opposite signs. Analogously to LAPACK, we assume that all 2-by-2 blocks on the diagonal of $\boldsymbol{T}$ have this format. The eigenvalues of (9) are $\lambda_k = \alpha + i\sqrt{|\beta\gamma|}$ and $\overline{\lambda}_k = \alpha - i\sqrt{|\beta\gamma|}$. Corresponding eigenvectors can be obtained from

[1] This transformation is implemented in DLANV2 in LAPACK 3.7.0.

$$\left( \begin{bmatrix} \boldsymbol{T}_{11} & \boldsymbol{t}_{12} & \boldsymbol{t}_{13} & \boldsymbol{T}_{14} \\ & \alpha & \beta & \boldsymbol{t}_{24}^T \\ & \gamma & \alpha & \boldsymbol{t}_{23}^T \\ & & & \boldsymbol{T}_{44} \end{bmatrix} - \lambda_k \boldsymbol{I} \right) \begin{bmatrix} \boldsymbol{x}_1 \\ x_2 \\ x_3 \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{0}. \tag{10}$$

An eigenvector $\boldsymbol{x}_k = [\boldsymbol{x}_1^T, x_2, x_3, \boldsymbol{0}^T]^T$ can be computed by choosing the mutually dependent $x_2$ and $x_3$ and solving the 1st block equation

$$(\boldsymbol{T}_{11} - \lambda_k \boldsymbol{I})\boldsymbol{x}_1 = -\boldsymbol{t}_{12}x_2 - \boldsymbol{t}_{13}x_3 \tag{11}$$

for $\boldsymbol{x}_1$ through modified backward substitution. When in the process a 1-by-1 block is encountered on the diagonal of $\boldsymbol{T}_{11}$, a complex division is executed in real arithmetic. When a 2-by-2 block is encountered, a 2-by-2 complex linear system is solved through Gaussian elimination with pivoting exploiting that the 2-by-2 block is normalized.

In summary, the linear system (8) for a real $\lambda_k$, and the linear system (11) for a complex $\lambda_k$ can be solved through a modified backward substitution method that can handle 2-by-2 blocks on the diagonal. A full algorithm is listed as Algorithm 1 in [3].

The computation of a single eigenvector corresponds to a level-2 BLAS operation. When many eigenvectors are computed simultaneously, the eigenvector computation can benefit from level-3 BLAS operations. We express the computation as a tiled algorithm which is rich in matrix-matrix multiplications. The upper quasi-triangular matrix $\boldsymbol{T}$ is partitioned into an $n_\mathrm{b}$-by-$n_\mathrm{b}$ grid of tiles such that (a) the 2-by-2 blocks $\boldsymbol{T}_{kk}$ are not split across tiles and (b) diagonal tiles are square. In order to simplify the presentation, we assume that all eigenvectors of $\boldsymbol{T}$ are sought. Our algorithms can handle the computation of an arbitrary subset of eigenvectors. Using complex numbers, the algorithm solves $\boldsymbol{T}\boldsymbol{X} = \boldsymbol{X}\boldsymbol{\Lambda}$, where $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n) \in \mathbb{C}^{n \times n}$. The eigenvector matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m] \in \mathbb{C}^{n \times m}$ is partitioned analogously to $\boldsymbol{T}$. (Recall that $m$ denotes the number of blocks on the diagonal of $\boldsymbol{T}$.) Algorithm 2 [3] expresses the eigenvector computation as a tiled algorithm using the following subroutines.

- Backsolve. This subroutine takes a diagonal tile $\boldsymbol{T}_{kk}$ as input and computes the eigenvectors $\boldsymbol{X}_{kk}$ of $\boldsymbol{T}_{kk}$. Hence, it solves $\boldsymbol{T}_{kk}\boldsymbol{X}_{kk} = \boldsymbol{X}_{kk}\boldsymbol{\Lambda}_{kk}$, where $\boldsymbol{\Lambda}_{kk}$ denotes the eigenvalues of $\boldsymbol{T}_{kk}$.
- Solve. This subroutine takes the eigenvector tile $\boldsymbol{X}_{jk}$ with $j < k$, the diagonal tile $\boldsymbol{T}_{jj}$, and the eigenvalues $\boldsymbol{\Lambda}_{kk}$ of the diagonal tile $\boldsymbol{T}_{kk}$ as inputs and solves $\boldsymbol{T}_{jj}\boldsymbol{X}_{jk} = \boldsymbol{X}_{jk}\boldsymbol{\Lambda}_{kk}$ through multi-shift (modified) backward substitution.
- Update. This subroutine computes a linear update $\boldsymbol{X}_{hk} \leftarrow \boldsymbol{X}_{hk} - \boldsymbol{T}_{hj}\boldsymbol{X}_{jk}$ with $h < j$.

Algorithm 2 can be parallelized with tasks [3] by encapsulating each function call in a separate task. A Backsolve task processing $\boldsymbol{X}_{kk}$ has outgoing dependences to Update tasks modifying $\boldsymbol{X}_{hk}$, $h = 1, \ldots, k-1$. The incoming dependences of a Solve task on $\boldsymbol{X}_{jk}$ are satisfied when all Update tasks writing to $\boldsymbol{X}_{jk}$ have been completed. To avoid race conditions, updates of $\boldsymbol{X}_{hk}$ are executed with exclusive write access.

# 3 Triangular matrices whose eigenvectors overflow

In this section, we present a class of triangular matrices for which the components of all eigenvectors exhibit rapid growth and almost all eigenvectors exceed the representational range. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be the upper triangular matrix given by

$$a_{ij} = \begin{cases} -c & i < j, \\ a + jb & i = j, \\ 0 & i > j. \end{cases} \tag{12}$$

**Algorithm 2:** Tiled Multi-Backsolve

**Data:** $\boldsymbol{T}$ as in (5), partitioned into an $n_\mathrm{b}$-by-$n_\mathrm{b}$ grid of tiles such that 2-by-2 blocks on the diagonal are not split and diagonal tiles are square

**Result:** Eigenvector matrix $\boldsymbol{X}$ such that $\boldsymbol{T}\boldsymbol{X} = \boldsymbol{X}\boldsymbol{\Lambda}$

```
TiledBacksolve(T)
    X ← 0;
    for k ← n_b, n_b − 1, ..., 1 do
        for j ← k, k − 1, ..., 1 do
            if k = j then
                X_kk ← Backsolve(T_kk);
            else
                X_jk ← Solve(X_jk, T_jj, Λ_kk);
            for h ← 1, ..., j − 1 do
                X_hk ← Update(X_hk, T_hj, X_jk);
    return X;
```

The eigenvalues $\lambda_j = a + jb$ can be read off from the diagonal. Let $\gamma$ denote the ratio $\gamma = \frac{b}{c}$. The eigenvectors are determined up to a scaling by the sequence $\{z_k\}_{k=1}^{\infty}$ given by

$$z_k = \binom{\gamma + k - 1}{k}. \tag{13}$$

Specifically, if $\boldsymbol{X} = [x_{ij}] \in \mathbb{R}^{n \times n}$ is the upper triangular matrix given by

$$x_{ij} = \begin{cases} z_{j-i} & i < j \\ 1 & i = j \\ 0 & i > j, \end{cases} \tag{14}$$

then the $j$th column of $\boldsymbol{X}$ is an eigenvector of $\boldsymbol{A}$ corresponding to the eigenvalue $\lambda_j$. We emphasize that eigenvectors of $\boldsymbol{A}$ are controlled entirely by the value of $\gamma$. It is irrelevant if the eigenvalues are clustered near $a$, i.e., $a \neq 0$ and small values of $\left|\frac{b}{a}\right|$, or if they are well-separated as in the case of $a = 0$ and large values of $|b|$. In the case of $\gamma = n$, we have the lower bound $z_k \geq 2^k$ for $k < n$ and most columns of $\boldsymbol{X}$ will exceed the representational range for modest values of $n$. Moreover, if $\gamma = \frac{1}{2}$, then $z_k \to 0$ as $k \to \infty$. In this case, the elements of $\boldsymbol{X}$ are all well inside the representational range.

# 4 Overflow-avoiding computations

This section addresses the robust computation of eigenvectors from the real Schur form. For this purpose, we associate a scaling factor $\xi_k \in (0, 1]$ with an eigenvector $\boldsymbol{x}_k$. The scaling factor $\xi_k$ is computed dynamically alongside with the solution $\boldsymbol{x}_k$ of the scaled triangular equation

$$(\boldsymbol{T}_{11} - \lambda_k \boldsymbol{I})\boldsymbol{x}_1 = \xi_k(-\boldsymbol{t}_{12}) \tag{15}$$

in the case of a real eigenvector or

$$(\boldsymbol{T}_{11} - \lambda_k \boldsymbol{I})\boldsymbol{x}_1 = \xi_k(-\boldsymbol{t}_{12}\boldsymbol{x}_2 - \boldsymbol{t}_{13}\boldsymbol{x}_3) \tag{16}$$

in the case of a complex eigenvector without ever exceeding the overflow threshold $\Omega > 0$.

In the following, previous work on robustness is reviewed. Then the overflow protection logic necessary to render the eigenvector computation robust is introduced. Finally, a robust version of Algorithm 2 is presented.

## 4.1 Robust solution of triangular linear systems

A robust backward substitution algorithm for the solution of an upper triangular system was developed by Anderson [9]. It solves the scaled triangular linear system

$$\boldsymbol{T}\boldsymbol{y} = \xi\boldsymbol{b}. \tag{17}$$

The scaling factor $\xi \in (0, 1]$ is computed alongside with the solution $\boldsymbol{y}$ such that overflow is avoided. LAPACK 3.7.0 implements Anderson's algorithms in the form of xLATRS. This routine uses upper bounds on the growth to determine ahead of time if overflow is impossible. If so, xLATRS falls back to its fast non-robust counterpart xTRSV in the BLAS. Otherwise, (17) is solved using dynamic scaling to prevent overflow. Since xLATRS keeps the solution $\boldsymbol{y}$ consistently scaled throughout the computation, frequent scaling events incur significant overhead.

The overhead of scaling events can be reduced as shown in [10, 11]. There, the single right-hand side is partitioned into segments. Each segment is associated with a local scaling factor. Scaling is reduced to those segments involved in an operation rather than the entire right-hand side. A consistently scaled solution is computed in a post-processing step. For this purpose, the global scaling factor $\xi$ is chosen as the smallest local scaling factor and the right-hand side segments are rescaled with respect to $\xi$.

The robust routine xLATRS supports only a single right-hand side. An extension to multiple right-hand sides has been realized in [6]. Each right-hand side $\boldsymbol{b}_k$ is associated with a global scaling factor $\xi_k$. When arranged as $\boldsymbol{B} = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n]$ and $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_n]$, the system $\boldsymbol{T}\boldsymbol{Y} = \boldsymbol{B}\,\mathrm{diag}(\boldsymbol{\xi})$ is solved for $\boldsymbol{Y}$. The solver Kiya [6] partitions the right-hand side matrix into tiles and has a separate local scaling factor for every column of every tile of the right-hand side matrix. With this approach, linear tile updates can still be executed as matrix-matrix multiplications. The local scaling factors are reduced to a vector of global scaling factors $\boldsymbol{\xi}$. We extend this approach to the eigenvector computation and use *augmented vectors*, introduced in [11, 10], as a means to represent scaled vectors.

**Definition 1** *An augmented vector $\langle \xi, \boldsymbol{x} \rangle$ consists of a scalar $\xi \in (0, 1]$ and a vector $\boldsymbol{x} \in \mathbb{R}^n$ and represents the vector $\boldsymbol{y} = \xi^{-1}\boldsymbol{x}$. We say that two augmented vectors $\langle \xi, \boldsymbol{x} \rangle$ and $\langle \zeta, \boldsymbol{y} \rangle$ are equivalent if and only if $\xi^{-1}\boldsymbol{x} = \zeta^{-1}\boldsymbol{y}$.*

**Definition 2** *We say that two augmented vectors $\langle \xi, \boldsymbol{x} \rangle$, $\langle \zeta, \boldsymbol{y} \rangle$ are consistently scaled if $\xi = \zeta$.*

Scaling factors effectively increase the exponent field of double-precision numbers and thereby avoid overflow during the computation. If the scaling factors attain the format $2^{\xi}$ as suggested in [10] and $\xi$ is stored as a 64 bit integer, the largest representable number is increased by a factor of $2^{2^{64}-1}$.

## 4.2 Overflow protection logic

Overflow protection logic can be evaluated prior to the potentially hazardous operation to obtain a scaling factor. Once this scaling factor has been applied, the operation can be executed safely. The eigenvector computation requires robust linear updates, divisions and summations. The following routines compute scaling factors such that these operations cannot exceed the overflow threshold.

- ProtectUpdate. This routine was introduced in [11] and computes a scaling factor $\xi \in (0, 1]$ such that the linear update $\xi\boldsymbol{Y} - \boldsymbol{T}(\xi\boldsymbol{X})$ cannot overflow. ProtectUpdate receives upper bounds $||\boldsymbol{Y}||_\infty$, $\|\boldsymbol{T}\|_\infty$ and $\|\boldsymbol{X}\|_\infty$ and returns a scaling factor $\xi$ which guarantees $\xi(\|\boldsymbol{Y}\|_\infty + \|\boldsymbol{T}\|_\infty\|\boldsymbol{X}\|_\infty) \leq \Omega$. We use ProtectUpdate for complex-valued matrices $\boldsymbol{X}$, $\boldsymbol{Y}$.
- ProtectDivision. This routine, presented in [11], computes a scaling factor $\xi \in (0, 1]$ such that the scalar division $(\xi b)/t$, $t \neq 0$ does not overflow, i.e., $|\xi b| \leq |t|\Omega$.
- ProtectSum. This routine computes a scaling factor $\xi \in \{\frac{1}{2}, 1\}$ such that the scalar summation $\xi(x + y)$ does not overflow, i.e., $|\xi(x + y)| \leq \Omega$. Section 4.4 derives the decision tree that leads to the scaling factor.

**Algorithm 3:** Robust solution of (15) (real arithmetic)

**Data:** $\boldsymbol{T}$ as in (5) with $||\boldsymbol{T}_{k\ell}||_\infty \leq \Omega$, $k, \ell = 1, \ldots, m$, position $k$ of the 1-by-1 block $t_{kk} = \lambda_k$
**Result:** An augmented vector $\langle \xi, \boldsymbol{x} \rangle$ with $\boldsymbol{T}(\xi^{-1}\boldsymbol{x}) = t_{kk}(\xi^{-1}\boldsymbol{x})$

```
BacksolveReal(T, k)
    λ ← t_kk;
    ⟨ξ, x⟩ ← ⟨1, −t_{1:k−1,k}⟩;
    for j ← k − 1, k − 2, ..., 1 do
        if T_jj is a 1-by-1 block then
            Compute δ with ProtectSum such that (δt_jj) − (δλ) can be executed safely;
            Compute ν with ProtectDivision such that (νδx_j)/((δt_jj) − (δλ)) can be executed
              safely;
            y ← (νδx_j)/((δt_jj) − (δλ));
            x ← νδx;
            x_j ← y;
            γ ← ProtectUpdate(||x_{1:j−1}||_∞, ||t_{1:j−1}||_∞, |x_j|);
            x_{1:j−1} ← (γx_{1:j−1}) − t_{1:j−1,j}(γx_j);
            ξ ← νδγξ;
        if T_jj is a 2-by-2 block then
            Compute δ with ProtectSum such that (δT_jj) − (δλI_2) can be executed safely.;
            Solve 2-by-2 system [(δT_jj) − (δλI_2)]y = νδx_j robustly for ν^{−1}y;
            x ← νδx;
            x_j ← y;
            γ ← ProtectUpdate(||x_{1:j−1}||_∞, ||T_{1:j−1,j}||_∞, ||x_j||_∞);
            x_{1:j−1} ← (γx_{1:j−1}) − T_{1:j−1,j}(γx_j);
            ξ ← νδγξ;
    return ⟨ξ, x⟩;
```

### 4.3 Robust computation of a single eigenvector

The routines ProtectUpdate, ProtectDivision and ProtectSum allow us to compute the scaling factor $\xi$ alongside with the eigenvector such that overflow is never encountered. Using augmented vectors, Algorithm (3) BacksolveReal solves the scaled linear system (15). Each operation that can overflow is preceded with overflow protection logic, which computes a scaling factor. After applying the scaling, the operation can be executed safely. These scaling factors are accumulated into the global scaling factor $\xi$. When a 2-by-2 block on the diagonal of $\boldsymbol{T}$ is encountered, a 2-by-2 system is computed robustly. We do this with Gaussian elimination with complete pivoting. The system is reduced to row echelon form; its construction and the transformations are protected with ProtectUpdate. The resulting triangular system is solved robustly as in [10].

The scaled complex system (16) can be solved similarly, yielding a robust routine BacksolveComplex. When during the backward substitution a 1-by-1 block is encountered, the complex division is executed in real arithmetic. We augment the corresponding routine DLADIV in LAPACK 3.7.0 [12] by rigorously adding overflow protection logic to every operation that can overflow. The real and imaginary part of the result are computed separately and yield two distinct scaling factors. Since we associate one single scaling factor with a complex eigenvector, the common scaling factor corresponds to the smaller one. When the backward substitution encounters a 2-by-2 block, a complex 2-by-2 system is solved. We again use Gaussian elimination with complete pivoting and reduce the system to row echelon form. Complex linear updates are split into robust updates of the real part and the robust updates of the imaginary part. The complex backward substitution reuses the robust complex division in real arithmetic. Hence, the solution of a complex

2-by-2 system involves more overflow-protected operations than the real counterpart.

**Remark.** The computation of an overflow-free result is susceptible to the execution order of instructions. In tests of our implementation we experienced that compiler optimization sometimes changed the intended execution order in a hazardous way. First, consider $d \leftarrow (ab)c$. When associative math is enabled, this explicitly bracketed expression may be executed as $d \leftarrow a(bc)$. We choose the compiler optimization flags such that associative math is disabled. Second, the successive execution of $d \leftarrow bc$ and $e \leftarrow ad$ was simplified to $e \leftarrow abc$. When executed as $(ab)c$, the intended execution order is not adhered to. We enforce the generation of intermediate variables like $d$ and thereby prescribe the execution order. In our C implementation, we achieve this by declaring all intermediate variables as `volatile`.

### 4.4 Robust scalar sum

In this section, we provide the logic to compute a scaling factor such that the scalar summation $z \leftarrow x + y$ can be executed robustly. Hence, we seek a scaling factor $\xi \in (0, 1]$ such that $|(\xi x) + (\xi y)| \leq \Omega$. Theorem 1 shows that a scaling factor $\xi \in \{\frac{1}{2}, 1\}$ suffices.

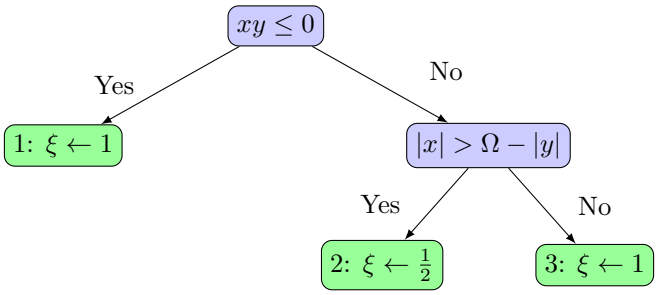


Figure 1: Computing a scaling factor $\xi$ such that the sum cannot overflow.

**Theorem 1** *Let $|x| \leq \Omega$ and $|y| \leq \Omega$ where $x, y \in \mathbb{C}$. The scaling factor $\xi$ computed in Figure 1 ensures that the summation $z \leftarrow \xi x + \xi b$ satisfies $|z| \leq \Omega$.*

*Proof.* The three cases are given by the leaves in Figure 1.

1. By interchanging $x$ and $y$ we can assume $x \leq 0 \leq y$. We then have $|x + y| = ||y| - |x|| \leq |y| \leq \Omega$.
2. We have $|x| > \Omega - |y|$. The choice $\xi = \frac{1}{2}$ yields $\xi|x| + \xi|y| \leq \frac{1}{2}\Omega + \frac{1}{2}\Omega \leq \Omega$.
3. We have $|x| \leq \Omega - |y|$, i.e., $|x| + |y| \leq \Omega$.

Analogously to the flow charts in [11, 10], Figure 1 shows the three cases that have to be considered for the summation of two numbers. Algorithm 4 ProtectSum lists the code to compute the scaling factor.

This concludes the routines needed to render the eigenvector computation robust.

## 5 Robust tiled eigenvector computation

The goal of this section is to develop the robust tiled algorithm EigVecs for computing eigenvectors from the real Schur form. We first introduce algorithms for the backsolve and the backtransform step and then couple these.

**Algorithm 4:** Scaling Factor for Robust Scalar Summation

**Data:** $x \in \mathbb{C}$ with $|x| \leq \Omega$, $y \in \mathbb{C}$ with $|y| \leq \Omega$
**Result:** A scaling factor $\xi$ such that $|(\xi x) + (\xi y)| \leq \Omega$

```
ProtectSum(x, y)
    ξ ← 1;
    if (x > 0 ∧ y > 0) ∨ (x < 0 ∧ y < 0) then
        if |x| > Ω − |y| then
            ξ ← 1/2;
    return ξ;
```

## 5.1 Backsolve

We present two algorithms. The first algorithm extends the computation to all eigenvectors of $\boldsymbol{T}$ and relies on level-2 BLAS operations. The second algorithm expresses the eigenvector computation as a tiled algorithm rich in matrix-matrix multiplications and uses the first algorithm as a basic building block.

Augmented vectors were used in the computation of a single eigenvector to represent a scaled eigenvector. This concept can be extended to many eigenvectors. Each eigenvector $\boldsymbol{x}_k$ is associated with a scaling factor $\xi_k$. The idea of associating one scaling factor with every right-hand side has been introduced in [10] and is referred to as an *augmented matrix*.

**Definition 3** *An augmented matrix $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ consists of a vector of scaling factors $\boldsymbol{\xi} = [\xi_1, \xi_2, \ldots, \xi_m]$, where $\xi_i \in (0, 1]$, and a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m] \in \mathbb{C}^{n \times m}$. It represents $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle = \left[\xi_1^{-1}\boldsymbol{x}_1, \xi_2^{-1}\boldsymbol{x}_2, \ldots, \xi_m^{-1}\boldsymbol{x}_m\right]$.*

Given the notion of an augmented matrix, the scaled system

$$\boldsymbol{T}(\boldsymbol{X}\,\mathrm{diag}(\boldsymbol{\xi}^{-1})) = (\boldsymbol{X}\,\mathrm{diag}(\boldsymbol{\xi}^{-1}))\boldsymbol{\Lambda} \tag{18}$$

is solved for $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$. Algorithm 5 solves (18) and extends the robust computation of a single eigenvector to the robust computation of all eigenvectors of $\boldsymbol{T}$.

**Algorithm 5:** Robust Multi-Backsolve of (18)

**Data:** $\boldsymbol{T}$ as in (5)
**Result:** An eigenvector matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m] \in \mathbb{C}^{n \times m}$ and scaling factors $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_m] \in (0, 1]^m$ such that $\boldsymbol{T}(\xi_i^{-1}\boldsymbol{x}_i) = \lambda_i(\xi_i^{-1}\boldsymbol{x}_i)$

```
Backsolve(T)
    for k ← m, m − 1, ..., 1 do
        Form the initial right-hand side x_k;
        if T_kk is a 1-by-1 block then
            ⟨ξ_k, x_k⟩ ← BacksolveReal(T_{1:k,1:k}, k);
        if T_kk is a 2-by-2 block then
            ⟨ξ_k, x_k⟩ ← BacksolveComplex(T_{1:k,1:k}, k);
    return ⟨[ξ_1, ..., ξ_m], [x_1, ..., x_m]⟩;
```

In order to render the tiled Algorithm 2 robust, we express the eigenvector computation with augmented matrices. By using augmented matrices in a tiled algorithm, we split up the global scaling factors into local scaling factors. Each eigenvector $\boldsymbol{x}_k$ is associated with $n_\mathrm{b}$ local scaling factors, one per tile row. In a post-processing step, these local scaling factors are reduced to the global scaling factor $\xi_k$. The non-robust routines

**Algorithm 6:** Thread-Safe Robust Tile Update

**Data:** $\boldsymbol{Y} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n] \in \mathbb{C}^{m \times n}$ with $||\boldsymbol{y}_j||_\infty \leq \Omega$ for $j = 1, \ldots, n$, $\boldsymbol{\nu} = [\nu_1, \ldots, \nu_n] \in (0, 1]^n$, $\boldsymbol{T} \in \mathbb{R}^{m \times k}$ with $||\boldsymbol{T}||_\infty \leq \Omega$, $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n] \in \mathbb{C}^{k \times n}$ with $||\boldsymbol{x}_j||_\infty \leq \Omega$ for $j = 1, \ldots, n$, $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_n] \in (0, 1]^n$

**Result:** Updated $\boldsymbol{Y}$ and $\boldsymbol{\nu}$ such that $(\nu_j^{-1}\boldsymbol{y}_j) \leftarrow (\nu_j^{-1}\boldsymbol{y}_j) - \boldsymbol{T}(\xi_j^{-1}\boldsymbol{x}_j)$

1 UPDATE($\langle \boldsymbol{\nu}, \boldsymbol{Y} \rangle, \boldsymbol{T}, \langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$)

2 lock($\langle \boldsymbol{\nu}, \boldsymbol{Y} \rangle$);

**for** $j \leftarrow 1, \ldots, n$ **do**

$\gamma_j \leftarrow \min\{\xi_j, \nu_j\}$;

$\zeta_j \leftarrow$ PROTECTUPDATE $\left(\frac{\gamma_j}{\nu_j}||\boldsymbol{y}_j||_\infty, ||\boldsymbol{T}||_\infty, \frac{\gamma_j}{\xi_j}||\boldsymbol{x}_j||_\infty\right)$;

$\delta_j \leftarrow \zeta_j\gamma_j$;

$\boldsymbol{Y} \leftarrow \left[\boldsymbol{Y} \operatorname{diag}\left(\frac{\delta_1}{\nu_1}, \ldots, \frac{\delta_n}{\nu_n}\right)\right] - \boldsymbol{T}\left[\boldsymbol{X} \operatorname{diag}\left(\frac{\delta_1}{\xi_1}, \ldots, \frac{\delta_n}{\xi_n}\right)\right]$;

3 $\boldsymbol{\nu} \leftarrow \boldsymbol{\delta}$;

4 unlock($\langle \boldsymbol{\nu}, \boldsymbol{Y} \rangle$);

5 **return** $\langle \boldsymbol{\delta}, \boldsymbol{Y} \rangle$;

**for** $j \leftarrow 1, \ldots, n$ **do**

$\zeta_j \leftarrow$ PROTECTUPDATE$(0, ||\boldsymbol{T}||_\infty, ||\boldsymbol{x}_j||_\infty)$;

$\boldsymbol{W} \leftarrow \boldsymbol{T}\,[\boldsymbol{X} \operatorname{diag}(\zeta_1, \ldots, \zeta_n)]$;

lock($\langle \boldsymbol{\nu}, \boldsymbol{Y} \rangle$);

**for** $j \leftarrow 1, \ldots, n$ **do**

$\gamma_j \leftarrow \min\{\nu_j, \xi_j\zeta_j\}$;

$\eta_j \leftarrow$ PROTECTSUM $\left(\frac{\gamma_j}{\nu_j}||\boldsymbol{y}_j||_\infty, \frac{\gamma_j}{\xi_j\zeta_j}||\boldsymbol{w}_j||_\infty\right)$;

$\delta_j \leftarrow \eta_j\gamma_j$;

$\boldsymbol{Y} \leftarrow \left[\boldsymbol{Y} \operatorname{diag}\left(\frac{\delta_1}{\nu_1}, \ldots, \frac{\delta_n}{\nu_n}\right)\right] - \left[\boldsymbol{W} \operatorname{diag}\left(\frac{\delta_1}{\xi_1\zeta_1}, \ldots, \frac{\delta_n}{\xi_n\zeta_n}\right)\right]$;

BACKSOLVE, SOLVE and UPDATE are replaced with robust counterparts. A robust version of BACKSOLVE is given in Algorithm 5. It solves the scaled system (18) and arranges the result as an augmented matrix. Similarly, SOLVE can be rendered robust. In order to compute an UPDATE robustly, we require consistently scaled augmented matrices as defined in [10].

**Definition 4** *We say that two augmented matrices $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ and $\langle \boldsymbol{\zeta}, \boldsymbol{Y} \rangle$ are consistently scaled if $\boldsymbol{\xi} = \boldsymbol{\zeta}$.*

A robust UPDATE is given in Algorithm (6). Scaling factors for the linear update are computed by simulating consistent scaling. Then the scaling factors are applied prior to executing the linear update. When executed in parallel, updates of $\langle \boldsymbol{\nu}, \boldsymbol{Y} \rangle$ require exclusive write access. We achieve this with locks, one lock per augmented tile. The lock-protected region can be pruned when the update $\boldsymbol{Y} \leftarrow \boldsymbol{Y} - \boldsymbol{T}\boldsymbol{X}$ is split into the accumulation into a buffer $\boldsymbol{W} \leftarrow \boldsymbol{T}\boldsymbol{X}$ and writing back $\boldsymbol{Y} \leftarrow \boldsymbol{Y} - \boldsymbol{W}$. In this case, the overflow protection logic is split and guards each operation separately. While this reduces the length of the lock-protected region, it increases the overhead from overflow protection logic. A discussion on the two options is given later in this section.

Given robust routines BACKSOLVE, SOLVE and UPDATE, Algorithm 7 solves (18) in a robust, tiled fashion. Since the algorithm uses local scaling factors in the form of augmented matrices, the eigenvector tiles are decoupled. As a consequence, the computed eigenvector segments may be inconsistently scaled. A consistent scaling is computed in a post-processing step. Algorithm 8 lists how a consistent scaling can be computed. The $n_\mathrm{b}$ local scaling factors are reduced to a global scaling factor for each eigenvector. Then the eigenvector segments are rescaled with respect to the global scaling factor.

Algorithm 7 can be parallelized with tasks. We separate the consistency scaling with a synchronization point. Then the computation (lines 3-11) can be taskified identically to its non-robust counterpart Algorithm 2. Rescaling is realized with perfectly parallel SCALE tasks, processing entire eigenvector tile columns.

**Algorithm 7:** Robust Tiled Multi-Backsolve

**Data:** $\boldsymbol{T}$ as in (5), partitioned into an $n_\mathrm{b}$-by-$n_\mathrm{b}$ grid of tiles such that 2-by-2 blocks on the diagonal are not split and diagonal tiles are square

**Result:** Augmented eigenvector matrix $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ such that $\boldsymbol{T}(\boldsymbol{X}\,\mathrm{diag}(\boldsymbol{\xi})^{-1}) = (\boldsymbol{X}\,\mathrm{diag}(\boldsymbol{\xi})^{-1}))\boldsymbol{\Lambda}$

1 RobustTiledBacksolve($\boldsymbol{T}$)
2 $\quad \boldsymbol{X} \leftarrow \boldsymbol{0}$;
3 $\quad$ **for** $k \leftarrow n_\mathrm{b}, n_\mathrm{b}-1, \ldots, 1$ **do**
4 $\quad\quad$ **for** $j \leftarrow k, k-1, \ldots, 1$ **do**
5 $\quad\quad\quad \boldsymbol{\xi}_{jk} \leftarrow [1, \ldots, 1]$;
6 $\quad\quad\quad$ **if** $k = j$ **then**
7 $\quad\quad\quad\quad \langle \boldsymbol{\xi}_{jk}, \boldsymbol{X}_{kk} \rangle \leftarrow$ Backsolve($\boldsymbol{T}_{kk}$);
8 $\quad\quad\quad$ **else**
9 $\quad\quad\quad\quad \langle \boldsymbol{\xi}_{jk}, \boldsymbol{X}_{jk} \rangle \leftarrow$ Solve($\langle \boldsymbol{\xi}_{jk}, \boldsymbol{X}_{jk} \rangle, \boldsymbol{T}_{jj}, \Lambda(\boldsymbol{T}_{kk})$);
10 $\quad\quad\quad$ **for** $h \leftarrow 1, 2, \ldots, j-1$ **do**
11 $\quad\quad\quad\quad \langle \boldsymbol{\xi}_{hk}, \boldsymbol{X}_{hk} \rangle \leftarrow$ Update($\langle \boldsymbol{\xi}_{hk}, \boldsymbol{X}_{hk} \rangle, \boldsymbol{T}_{hj}, \langle \boldsymbol{\xi}_{jk}, \boldsymbol{X}_{jk} \rangle$);
12 $\quad$ Reduce scaling factors to $\boldsymbol{\xi}$ and scale $\boldsymbol{X}$ consistently;
13 $\quad$ **return** $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$

**Algorithm 8:** Consistency scaling

**Data:** Augmented matrices $\langle \boldsymbol{\xi}_{hk}, \boldsymbol{X}_{hk} \rangle$

**Result:** Augmented matrix $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ where $\boldsymbol{X} \in \mathbb{C}^{n \times m}$

1 `// Compute global scaling factor` $\xi_k$ `of` $\boldsymbol{x}_k$
2 **for** $k \leftarrow 1, 2, \ldots, m$ **do**
3 $\quad \xi_k^{\min} \leftarrow \min_{1 \le h \le n_\mathrm{b}} \{\xi_{hk}\}$ (column-wise minimum);
4 $\quad$ `// Consistency scaling`
5 $\quad$ **for** $h \leftarrow 1, 2, \ldots, n_\mathrm{b}$ **do**
6 $\quad\quad \boldsymbol{x}_{hk} \leftarrow \left(\frac{\xi_k^{\min}}{\xi_{hk}}\right) \boldsymbol{x}_{hk}$;
7 $\boldsymbol{\xi} \leftarrow \left[\xi_1^{\min}, \ldots, \xi_m^{\min}\right]$;

The overflow protection logic used to guard linear updates relies on upper bounds. We provide these upper bounds in the form of matrix norms $||\boldsymbol{T}_{hj}||_\infty$. Since matrix norms are expensive to compute, we precompute and record $||\boldsymbol{T}_{hj}||_\infty$ with perfectly parallel Bound tasks. Note that we only require upper bounds for tiles above the diagonal, so the magnitude changes arising from shifts $\boldsymbol{T} - \lambda \boldsymbol{I}$ have no impact. We separate the computation of $||\boldsymbol{T}_{hj}||_\infty$ from the remaining computation by a synchronization point. Thereby the amount of dependences handled by the runtime system is reduced.

Small linear updates in Backsolve and Solve tasks also rely on norms $||\boldsymbol{x}_{1:j-1}||_\infty$ as upper bounds, see lines 9 and 17 in Algorithm 3. The eigenvector norm is subject to change in every linear update and has to be recomputed. To delimit the cost of recomputing the norm, we use recursive blocking. We cut Backsolve and Solve into small internal tiles and replicate the structure of Algorithm 7.

We conclude this section by discussing under what circumstances either of the two implementations of an Update task, given as Algorithm 6 (left) or (right), is advantageous. The runtime of an Update task is dominated by the matrix-matrix multiplication. Algorithm 6 (left) computes the update with the minimum amount of operations and executes the matrix-matrix multiplication within the lock-protected region. Algorithm 6 (right), by contrast, reduces the length of the lock-protection region to a quick matrix subtraction at the expense of doubling the overflow protection logic and incurring additional computation.

Hence, the two implementations reflect the trade-off between a length of the lock-protected region and additional computation.

The case that a thread executing a task cannot acquire a lock can be handled in two ways. The thread can block until the lock becomes available. The resulting idling and synchronization overhead depends on the length of the critical section and how contended the lock is. Alternatively, the UPDATE task can be suspended (e.g., using `#pragma omp taskyield`) to allow the thread to execute other ready tasks. However, this incurs additional scheduling overhead. Based on experiments, blocking appears to be favorable for Algorithm 6 (right) when the critical section is short. Algorithm 6 (left), by contrast, results in substantial idling overhead already for small processor counts when blocking is used and in these cases task suspension is superior.

It clearly depends on the runtime system, the processor count, and the locking approach (blocking or task-suspending) whether Algorithm 6 (left) or (right) is advantageous. Our measurements indicate that Algorithm 6 (left) with task suspension is advantageous for small core counts. Out of theoretical considerations we expect Algorithm 6 (right) with blocking to be superior for sufficiently large core counts. In our experiments, however, we were unable to increase the processor count enough to observe this behavior.

## 5.2 Backtransform

The computation of eigenvectors of a real matrix $\boldsymbol{A}$ with the Schur decomposition $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T$ comprises a backsolve and a backtransform step. The previous section derived algorithms for the backsolve step and computed an augmented eigenvector matrix $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ of $\boldsymbol{T}$. This section addresses the backtransform of $\langle \boldsymbol{\xi}, \boldsymbol{X} \rangle$ to eigenvectors of $\boldsymbol{A}$.

Under the assumption that the backtransform cannot encounter overflow, it is legitimate to drop the scaling factors $\boldsymbol{\xi}$. Then the backtransform computes $\boldsymbol{Y} \leftarrow \boldsymbol{Q}\boldsymbol{X}$ and can be realized as a tiled, structure-exploiting matrix-matrix multiplication. The computation can be taskified over the tiles in $\boldsymbol{Y}$ such that $\boldsymbol{Y}_{hj} \leftarrow \boldsymbol{Q}_{h,1:j}\boldsymbol{X}_{1:j,j}$ constitutes a task, see Algorithm 5 TILEDBACKTRANSFORM in [3].

Although the backtransform multiplies with an orthogonal matrix $\boldsymbol{Q}$, overflow can nevertheless occur in pathological cases. Consider, for example, the Householder matrix $\boldsymbol{Q} = \boldsymbol{I} - \tau \boldsymbol{v}\boldsymbol{v}^T$, where $\boldsymbol{v}$ is a unit norm vector and $\tau$ is a scalar. Then an eigenvector $\boldsymbol{x}$ is backtransformed $\boldsymbol{Q}\boldsymbol{x} = ||\boldsymbol{x}||_2 \boldsymbol{e}_1$, where $\boldsymbol{e}_1$ is the first standard unit vector. Since we may have $\|\boldsymbol{x}\|_2 > \Omega$ even though all $|x_j| \leq \Omega$, overflow in the backtransform is possible. Normalization prior to the backtransform solves this problem.

## 5.3 Normalization

The eigenvalue equation $\boldsymbol{A}\boldsymbol{y} = \lambda \boldsymbol{y}$ allows the eigenvector $\boldsymbol{y}$ to have any magnitude. For further processing, however, it is advantageous to bound the magnitude. First, we review the normalization done in LAPACK. Then we advocate using the 2-norm and show how normalization and consistency scaling can be applied jointly.

LAPACK backsolves, backtransforms and then normalizes the eigenvectors. A real eigenvector is normalized

$$\boldsymbol{x} \leftarrow \boldsymbol{x}/||\boldsymbol{x}||_\infty, \quad \|\boldsymbol{x}\|_\infty = \max_{1 \leq j \leq n} \{|x_j|\} \tag{19}$$

so that the entry with the largest magnitude becomes one. A complex eigenvector $\boldsymbol{z} = \boldsymbol{x} + i\boldsymbol{y}$, where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ is normalized

$$\boldsymbol{z} \leftarrow \boldsymbol{z}/\|\boldsymbol{z}\|_\infty, \quad \|\boldsymbol{z}\|_\infty = \left\| \begin{bmatrix} \boldsymbol{x} & \boldsymbol{y} \end{bmatrix} \right\|_\infty = \max_{1 \leq j \leq n} \{|x_j| + |y_j|\} \tag{20}$$

so that the infinity norm of the matrix $\begin{bmatrix} \boldsymbol{x} & \boldsymbol{y} \end{bmatrix} \in \mathbb{R}^{n \times 2}$ becomes one. We propose to normalize the eigenvectors with respect to the 2-norm. While the robust computation of the 2-norm is computationally expensive, it guarantees that the backtransform is free from overflow. The normalization with respect to the 2-norm

also incurs two computational benefits. First, since we normalize the eigenvectors in between backsolve and backtransform, the triangular shape of $\boldsymbol{X}$ can be exploited. This halves the work compared to the normalization of backtransformed eigenvectors. Recall that the backtransform multiplies an orthonormal matrix with the eigenvector matrix. Hence, the eigenvectors are still normalized after the backtransform. Second, the amount of data loads can be reduced even further when normalization and consistency scaling are applied simultaneously.

The computation of the 2-norm of a real eigenvector $\|\boldsymbol{x}\|_2 = \sqrt{x_1^2 + \ldots + x_n^2}$ is problematic because squaring can exceed the overflow threshold. The LAPACK 3.7.0 routine DNRM2 avoids overflow by computing

$$||\boldsymbol{x}||_2 = x_{\max}\sqrt{\sum_{j=1}^{n}\left(\frac{x_j}{x_{\max}}\right)^2}, \quad x_{\max} = \max_{1\leq j\leq n}\{|x_j|\}. \tag{21}$$

This can be generalized to complex eigenvectors, which are normalized

$$\begin{aligned} ||\boldsymbol{x}||_2 = x_{\max}\sqrt{\sum_{j=1}^{n}\left(\left(\frac{\Re(x_j)}{x_{\max}}\right)^2 + \left(\frac{\Im(x_j)}{x_{\max}}\right)^2\right)}, \\ x_{\max} = \max_{1\leq j\leq n}\{|\Re(x_j)|, |\Im(x_j)|\}. \end{aligned} \tag{22}$$

Since Algorithm 7 represents the solution as augmented matrices, postprocessing is necessary to restore

**Algorithm 9:** Merged consistency scaling and normalization

```
1  // Compute global scaling factor ξ_k of x_k
2  for k ← 1, 2, ..., m do
3      ξ_k^min ← min_{1≤h≤n_b} {ξ_hk} (column-wise minimum);
4  // Compute l_k = ||x_k||_2 robustly simulating consistent scaling
5  l ← [0, ..., 0];
6  for k ← 1, 2, ..., m do
7      l_k ← ROBUST2NORM([ξ_k^min/ξ_1,k, ..., ξ_k^min/ξ_n_b,k], [x_1,k, ..., x_n_b,k]);
8  // Normalize and scale consistently
9  for k ← 1, 2, ..., m do
10     for h ← 1, 2, ..., n_b do
11         x_hk ← (ξ_k^min/ξ_hk) (1/l_k) x_hk;
```

consistent scaling. Consistency scaling and normalization can be applied simultaneously. Algorithm 9 demonstrates this using the 2-norm and can be used in line 12 of Algorithm 7. Normalization as done in LAPACK (19), (20) works analogously. Algorithm 9 reduces the local scaling factors to a global scaling, one per eigenvector (line 2, 3). The robust norm computation as done in DNMR2 can be extended to tiled scaled vectors. An illustration of the index mapping between vector tiles and absolute indices is given in Figure 2. For a real vector, the tiled version of (21) is

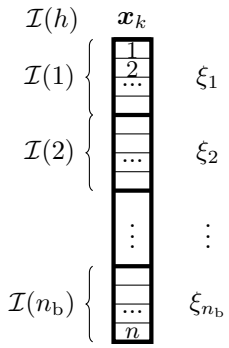


Figure 2: Mapping between tile indices and global indices.

$$||\boldsymbol{x}||_2 = x_{\max}\sqrt{\sum_{h=1}^{n_\mathrm{b}}\left(\frac{\xi^{\min}}{\xi_h}\right)^2\sum_{j\in\mathcal{I}(h)}\left(\frac{x_j}{x_{\max}}\right)^2},$$

$$x_{\max} = \max_{1\le h\le n_\mathrm{b}}\left\{\frac{\xi^{\min}}{\xi_h}\max_{j\in\mathcal{I}(h)}\{|x_j|\}\right\}.$$

For a complex vector, the tiled version of (22) is

$$||\boldsymbol{x}||_2 = x_{\max}\sqrt{\sum_{h=1}^{n_\mathrm{b}}\left(\frac{\xi^{\min}}{\xi_h}\right)^2\sum_{j\in\mathcal{I}(h)}\left(\left(\frac{\Re(x_j)}{x_{\max}}\right)^2+\left(\frac{\Im(x_j)}{x_{\max}}\right)^2\right)},$$

$$x_{\max} = \max_{1\le h\le n_\mathrm{b}}\left\{\frac{\xi^{\min}}{\xi_h}\max_{j\in\mathcal{I}(h)}\{|\Re(x_j)|,|\Im(x_j)|\}\right\}.$$

Note that both calculations simulate consistent scaling by applying the consistency scaling factor $\xi^{\min}/\xi_h$ after computing the partial norm. With this, the routine for the robust norm computation ROBUST2NORM is a natural generalization of the LAPACK implementation of DNMR2.

The 2-norm of a representable vector $\boldsymbol{z} = \boldsymbol{x} + i\boldsymbol{y} \in \mathbb{C}^n$ can exceed the representational range. In general, we have

$$\|\boldsymbol{z}\|_2 \le \sqrt{n}\|\boldsymbol{z}\|_\infty.$$

Moreover, equality is possible as demonstrated by the case of

$$\forall j \in \{1, 2\ldots, n\} \;:\; (|x_j| + |y_j| = t \wedge |x_j||y_j| = 0)\,. \tag{23}$$

We are currently using a *practical* overflow threshold $\Omega_p$ for which $\Omega_p \le \frac{\Omega}{\sqrt{n}}$.

Once the normalization factor and the global scaling factor are known, the eigenvector tiles are rescaled (line 11). In our implementation we exploit the upper triangular structure of the eigenvector matrix and prune the loops appropriately.

Coupling backsolve, backtransform and normalization gives us the routine EIGVECS, listed in Algorithm 10, for computing all eigenvectors in a tiled, scalable way.

# 6 Robust eigenvectors on distributed memory systems

This section addresses the extension of the robust shared-memory implementation to distributed memory systems. Our algorithm is based on the non-robust Algorithm 6 from [3], to which overflow protection is

**Algorithm 10:** Robust Tiled Eigenvector Computation

**Data:** $\boldsymbol{T}$ as in (5), partitioned into an $n_\mathrm{b}$-by-$n_\mathrm{b}$ grid of tiles such that 2-by-2 blocks on the diagonal are not split and diagonal tiles are square, orthogonal matrix $\boldsymbol{Q}$, where $\boldsymbol{T}$ and $\boldsymbol{Q}$ form $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T$.

**Result:** Eigenvector matrix $\boldsymbol{Y}$ such that $\boldsymbol{A}\boldsymbol{Y} = \boldsymbol{Y}\boldsymbol{\Lambda}$.

```
1 EigVecs(T, Q)
2     ⟨ξ, X⟩ ← RobustTiledBacksolve(T) using Algorithm 9;
3     Y ← TiledBacktransform(Q, X);
4     Normalize Y (unnecessary if ||·||_2 is used);
5     return Y;
```

added. The goal is to use one MPI rank per node and task parallelism node-internally. Here we focus on robustness and refer to [3] for a detailed description of the algorithm.

The initial data distributions are as follows. The matrices $\boldsymbol{T}$ and $\boldsymbol{Q}$ are 2D tile cyclically distributed; the eigenvectors are distributed column-wise among the ranks. By replicating $\boldsymbol{T}$ and $\boldsymbol{Q}$ we can reuse the task-parallel eigenvector routine for the node-internal computation.

As the eigenvectors are distributed column-wise, RobustTiledBacksolve is restricted to the computation of some eigenvectors. The desired eigenvectors are specified by the set of selected blocks $\mathcal{S} \subseteq \{\boldsymbol{T}_{11}, \ldots, \boldsymbol{T}_{mm}\}$ of the diagonal blocks of $\boldsymbol{T}$ defined in (5). The eigenvectors corresponding to the selected blocks are computed with BacksolveReal or BacksolveComplex. The computed eigenvectors and associated scaling factors are stored compactly. Analogously, Solve can be restricted to compute selected eigenvectors. Update tasks remain as in Algorithm 6. Then the tiled algorithm for computing selected eigenvectors has the same structure as Algorithm 7, only restricting the computation to selected eigenvectors. We call this algorithm RobustTiledBacksolveSel.

Using RobustTiledBacksolveSel, Algorithm 11 computes all eigenvectors on a distributed memory system. Once $\boldsymbol{T}$ has been replicated, each rank independently computes its share of the eigenvectors of $\boldsymbol{T}$. Since the eigenvectors are distributed column-wise, overflow protection and rescaling happen node-internally. Hence, robustness does not incur additional communication or synchronization between ranks. Once $\boldsymbol{Q}$ has been replicated, each rank backtransforms its share of eigenvectors. Again, no communication between ranks is needed assuming a column-wise distribution of $\boldsymbol{Y}$.

The replication of $\boldsymbol{T}$ prior to the computation is a bottleneck. Since the overflow protection relies on upper bounds $||\boldsymbol{T}_{ij}||_\infty$, our algorithm overlaps the replication of $\boldsymbol{T}$ with computing upper bounds. As soon as a tile $\boldsymbol{T}_{ij}$ has been received, the corresponding Bound task is inserted into the task queue. Hence, the bounds $||\boldsymbol{T}_{ij}||_\infty$ are computed redundantly on each node. The replication of $\boldsymbol{Q}$ can in the ideal case be fully overlapped with the backsolve.

# 7 Related work

In this section we summarize existing approaches, preparing for a quantitative comparison in Section 8.

**MAGMA** The MAGMA 2.4.0 routine magma_dtrevc3_mt [13] parallelizes the LAPACK routine DTREVC3. A user's selection of eigenvectors is computed column-by-column through robust backward substitution of (15) in case of a real eigenvector and (16) in case of a complex eigenvector. The solution of each scaled shifted system with backward substitution constitutes a task. The parallel computation is possible because the shift $\boldsymbol{T} - \lambda_k \boldsymbol{I}$ is applied on-the-fly without modifying $\boldsymbol{T}$. Hence, the backsolve corresponds to parallel level-2 BLAS operations. The backtransform is realized with task-parallel, tiled matrix-matrix multiplications. In a post-processing step, the eigenvectors are normalized serially such that the largest entry has magnitude 1.

**Algorithm 11:** Robust Distributed Memory + OpenMP Eigenvector Computation

**Data:** 2D tile cyclically distributed upper quasi-triangular matrix $\boldsymbol{T}$, 2D tile cyclically distributed orthogonal matrix $\boldsymbol{Q}$, where $\boldsymbol{T}$ and $\boldsymbol{Q}$ form $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T$.
**Result:** Eigenvector matrix $\boldsymbol{Y}$ such that $\boldsymbol{A}\boldsymbol{Y} = \boldsymbol{Y}\boldsymbol{\Lambda}$.

```
1  DistributedEigVecs(T, Q)
2      Compute partitioning of X, Y into n_r tile columns and distribute the tile columns among the r
         processes;
3      Replicate T through asynchronous all-to-all broadcast;
4      Wait for broadcast of T to complete;
5      if Q ≠ I then
6          Replicate Q through asynchronous all-to-all broadcast;
7      for k ← 1, ..., n_r do
8          if my rank owns X_:k then
9              Allocate X_:k, Y_:k;
10             Thread-parallel ⟨ξ_k, X_:k⟩ ← RobustTiledBacksolveSel(T, blocks(T_:k)) using
                 Algorithm 9;
11     if Q ≠ I then
12         Wait for broadcast of Q to complete;
13         for k ← 1, ..., n_r do
14             if my rank owns X_:k then
15                 Thread-parallel Y_:k ← Q X_:k;
16                 Normalize Y_:k (unnecessary if ||·||_2 is used);
17     else
18         Y_:k ← X_:k;
19     return Y;
```

GPUs can optionally accelerate the backtransform. Backsolve tasks, data transfers between CPU and GPU and backtransform tasks can be processed in parallel for distinct eigenvectors and therefore be overlapped.

**Elemental** The distributed memory library Elemental [7] release 0.87.7 includes two shifted backward substitution routines [8]. The first routine is SafeMultiShiftTrsm. It solves the scaled shifted systems $(\boldsymbol{T} - \lambda_k \boldsymbol{I})\boldsymbol{x}_k = \xi_k \boldsymbol{b}_k$ for $\boldsymbol{x}_k$. The matrix $\boldsymbol{T}$ is upper quasi-triangular and shifted with $\lambda_k$. The $\boldsymbol{b}_k$ form a rectangular right-hand side. Robustness is realized through global scaling factors $\xi_k \in (0, 1]$, one per right-hand side $\boldsymbol{b}_k$, see [3] for a qualitative comparison with our approach to robustness. Here we focus on the second routine TriangEig.

TriangEig solves (18) for a triangular matrix $\boldsymbol{T}$. The eigenvector matrix $\boldsymbol{X}$ and $\boldsymbol{T}$ are equally distributed in an element-wise fashion. An iteration of TriangEig proceeds as follows. The matrix $\boldsymbol{T}$ is partitioned logically into a window on the diagonal and an off-diagonal block above the window. This partitioning is geared towards two operations: a multi-shift backward substitution and a linear update. The former processes the window and the corresponding conformally partitioned eigenvector block row. The memory is redistributed such that the window of $\boldsymbol{T}$ is replicated across all involved ranks. The eigenvector matrix is redistributed such that one rank is in charge of (at least) one eigenvector. Then the eigenvectors can be computed in parallel through a multi-shift backward substitution. This corresponds to a Solve task in Algorithm 7. The linear update uses the corresponding off-diagonal block of $\boldsymbol{T}$ to update the eigenvectors. This block and the eigenvectors are replicated such that one rank holds all data necessary to execute the linear update as a matrix-vector multiplication. For a reasonable matrix size relative to the processor count, a rank

is in charge of many eigenvectors. Hence, the linear update corresponds to a matrix-matrix multiplication. After the linear update, the updated eigenvectors are again distributed element-wise. The algorithm moves the window of $\boldsymbol{T}$ up along the diagonal and advances with the next iteration.

Robustness is realized through global scaling factors $\boldsymbol{\xi}$, one per eigenvector. Since eigenvectors are kept consistently scaled throughout the computation, scaling events trigger rescaling of the entire eigenvector. Our tiled solver, by contrast, uses local scaling factors, which limits scaling to those tiles involved in an operation.

Elemental implements memory movements between ranks with synchronous MPI communication. As a consequence, the multi-shift backward substitution and the linear block updates are executed alternatingly. Elemental parallelizes the eigenvector computation by the independent processing of eigenvector columns. We, by contrast, split the linear block update into smaller linear tile updates and leave it to a runtime system to schedule parallel tasks.

This concludes the qualitative comparison. A quantitative comparison between Magma and Elemental on the one hand and our solver on the other hand is given in Section 9.

# 8 Numerical experiments

This section describes how the numerical experiments were set up and executed.

**Hardware** Our experiments are executed on Intel Xeon Gold 6132 (Skylake) nodes of Kebnekaise, a supercomputer at High Performance Computing Center North (HPC2N), Umeå, Sweden. Each node has 192 GB RAM and 14 cores on each of the 2 NUMA islands. In double-precision arithmetic the theoretical peak performance is 83.2 GFLOPS/s per core and 2329.6 GFLOPS/s per node. The nodes have dynamic frequency scaling enabled. The bandwidth was measured with the STREAM triad benchmark at 12.7 GB/s for one core and 162 GB/s for a full node.

**Software and configuration** We use the Intel compiler 18.0.3 with optimization level `-O2`, enable AVX-512 instructions and activate interprocedural optimizations `-ipo`. We link against the single-threaded MKL 2018.3.222 BLAS implementation. The shared memory API is OpenMP 4.5. We forbid migration of threads by setting `KMP_AFFINITY` to `compact`.

The following six routines were used in the numerical experiments. The first three routines have been presented in other work and are used for comparison.

- TriangEig. The Elemental 0.87.7 library contains the robust routine TriangEig [8]. This routine computes all eigenvectors of a triangular matrix on a distributed memory system. The triangular matrix and the eigenvector matrix are distributed identically in an element-wise fashion (`DistMatrix<double,MC,MR>`). The eigenvectors are normalized with respect to the 2-norm exploiting the upper triangular structure. We tuned the algorithmic block size through a sweep over 80:8:272 for every core count and use the default processor grid. The performance results of the sequential runs use the specialized sequential solver.
- magma_dtrevc3_mt. MAGMA 2.4.0 contains the robust implementation magma_dtrevc3_mt [13], a thread-parallel implementation of the LAPACK routine DTREVC3.
- TiledBacksolve. This non-robust solver [3] computes the eigenvectors of a quasi-triangular matrix on a shared memory system and casts the computation entirely with level-3 BLAS operations. This routine serves as reference on how fast eigenvectors can be computed.

The following routines have been presented in this paper.

- RobustTiledBacksolve. This solver is the robust counterpart of TiledBacksolve. Due to the usage of local scaling factors, both implementations have the same degree of parallelism. We tune the tile size by a sweep over 260:32:900.

Table 1: Overview of the eigenvector routines – presence of overflow protection, supported matrices in the backsolve and availability of a backtransform.

| | robust | backsolve | backtransform |
|---|---|---|---|
| TRIANGEIG | yes | triangular | no |
| MAGMA_DTREVC3_MT | yes | quasi-triangular | yes |
| TILEDBACKSOLVE | no | quasi-triangular | no |
| ROBUSTTILEDBACKSOLVE | yes | quasi-triangular | no |
| EIGVECS | yes | quasi-triangular | yes |
| DISTRIBUTEDEIGVECS | yes | quasi-triangular | yes |

- EIGVECS. This robust routine computes the eigenvectors of a general real matrix. It uses ROBUSTTILEDBACKSOLVE for the backsolve and TILEDBACKTRANSFORM for the backtransform. The used tile size is identical to ROBUSTTILEDBACKSOLVE.
- DISTRIBUTEDEIGVECS. This solver extends EIGVECS to distributed memory systems. It uses OpenMP tasks node-internally and MPI to communicate between nodes. The used tile size is identical to ROBUSTTILEDBACKSOLVE.

Table 1 summarizes what each of the six eigenvector routines computes. Driven by the different compute capabilities, we design our experiments such that the routines solve the same problem.

**Test problems** The experiments solve three types of test problems. The **first test problem** employs the family of matrices described in Section 3. All experiments set $a = 0$ and $b = 1$. We consider the two subsystems defined by $c = n$ and $c = \frac{1}{2}$, respectively. Recall that the choice $c = n$ ensures that the eigenvalues grow rapidly and frequent scalings are necessary. Moreover, the choice of $c = \frac{1}{2}$ ensures that numerical scaling is never necessary. There is no backtransform.

The **second test problem** processes an upper triangular matrix $\boldsymbol{B}$ with 1-by-1 and 2-by-2 blocks on the diagonal, see (5). The upper triangular part of $\boldsymbol{B}$ is set to random numbers in $[0, 1)$. The diagonal blocks are

$$\boldsymbol{B}_{jj} = \begin{cases} n+j & \text{if a 1-by-1 block} \\ \begin{bmatrix} n+j-\frac{1}{2} & -1 \\ 1 & n+j-\frac{1}{2} \end{bmatrix} & \text{if a 2-by-2 block,} \end{cases} \tag{24}$$

$j = 1, \ldots, m$.

The **third test problem** takes as input the real Schur decomposition $\boldsymbol{C} = \boldsymbol{Q}\boldsymbol{B}\boldsymbol{Q}^T$, where $\boldsymbol{B}$ is the upper quasi-triangular matrix from the second test problem. We define $\boldsymbol{Q}$ as a Householder matrix constructed from a random, normalized Householder vector. Given $\boldsymbol{B}$ and $\boldsymbol{Q}$, the matrix $\boldsymbol{C}$ is computed.

**Reliability** The runtime measurements were conducted with exclusive node access. Each problem was solved three times; the median runtime was used for the performance results. Following [14], the validation procedure evaluates the relative normwise backward error

$$\frac{||\boldsymbol{A}\boldsymbol{x} - \lambda\boldsymbol{x}||_F}{||\boldsymbol{A}||_F||\boldsymbol{x}||_F + |\lambda|||\boldsymbol{x}||_F} \tag{25}$$

for each computed eigenvector.

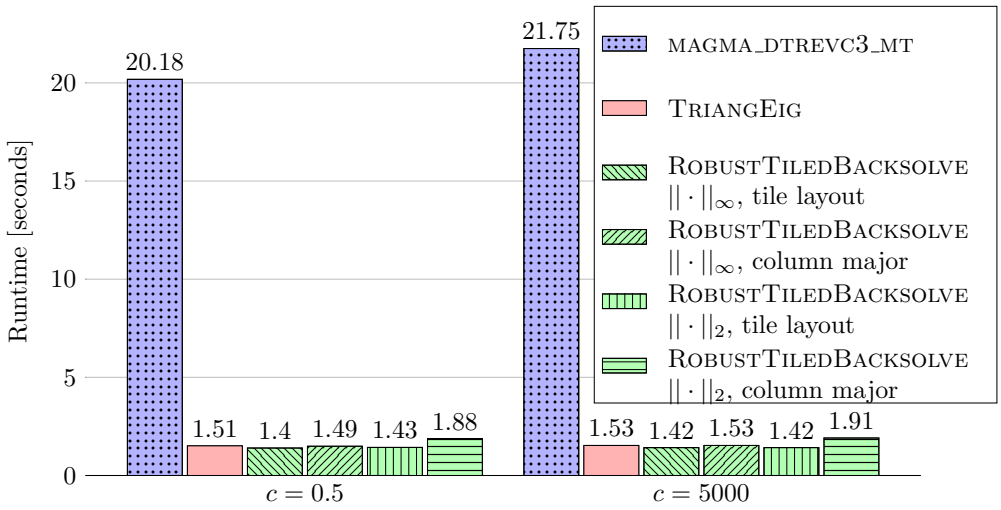


Figure 3: Serial execution of the robust solvers on $\boldsymbol{A}$ with $n = 5000$.

# 9 Performance results

This section presents the results of the numerical experiments. We conduct five experiments. The first two experiments perform a quantitative comparison between TRIANGEIG, MAGMA_DTREVC3_MT and ROBUSTTILEDBACKSOLVE. The third and fourth experiment analyze the cost of robustness. The fifth experiment considers extreme cases for the eigenvector computation and delivers lower and upper bounds of the runtime.

**Quantitative serial comparison** The first experiment compares TRIANGEIG, MAGMA_DTREVC3_MT and ROBUSTTILEDBACKSOLVE sequentially on the first test problem. Figure 3 shows the results on a matrix with dimension 5000. Since this experiment does not backtransform the eigenvectors, MAGMA_DTREVC3_MT relies entirely on level-2 BLAS operations, whereas TRIANGEIG and ROBUSTTILEDBACKSOLVE use level-3 BLAS. Since the performance difference widens with increasing matrix sizes, the next parallel experiments focus on a comparison between the former two TRIANGEIG and ROBUSTTILEDBACKSOLVE.

**Quantitative parallel comparison** The second experiment compares the parallel scalability of TRIANGEIG and ROBUSTTILEDBACKSOLVE on the first test problem. The experiments fix the problem size and increase the number of cores. Figure 4 compare the solvers on one shared memory node on a matrix with dimension 10000 (top) and 40000 (bottom). Figure 5 extends the experiment to distributed memory for a matrix with dimension 40000. Four nodes correspond to a total of 112 cores.

In all experiments ROBUSTTILEDBACKSOLVE is at least as fast as TRIANGEIG. Our solver achieves a similar performance for $c = 0.5$ and $c = n$. This suggests that numerical scaling does not impede the parallel scalability. In the case of TRIANGEIG, numerical scaling results in a noticeable performance difference. While the sequential runs with $c = n$ and $c = 0.5$ achieve a similar performance, there is a widening performance gap when the number of cores is increased. Recall that TRIANGEIG uses the default processor grid. Some core counts cause degenerated data distributions, which explains the runtime fluctuations.

**Analysis of overhead from overflow protection** The third experiment aims at quantifying the cost of robustness. For this purpose, the non-robust TILEDBACKSOLVE and its robust counterpart ROBUSTTILEDBACKSOLVE are compared on the first test problem. Figure 6 decomposes the runtime into task types for a sequential run (left) and a parallel run (right). The experiment with $c = 0.5$ does not require numerical scaling and reveals the overhead due to the evaluation of overflow protection logic. BACKSOLVE/SOLVE tasks

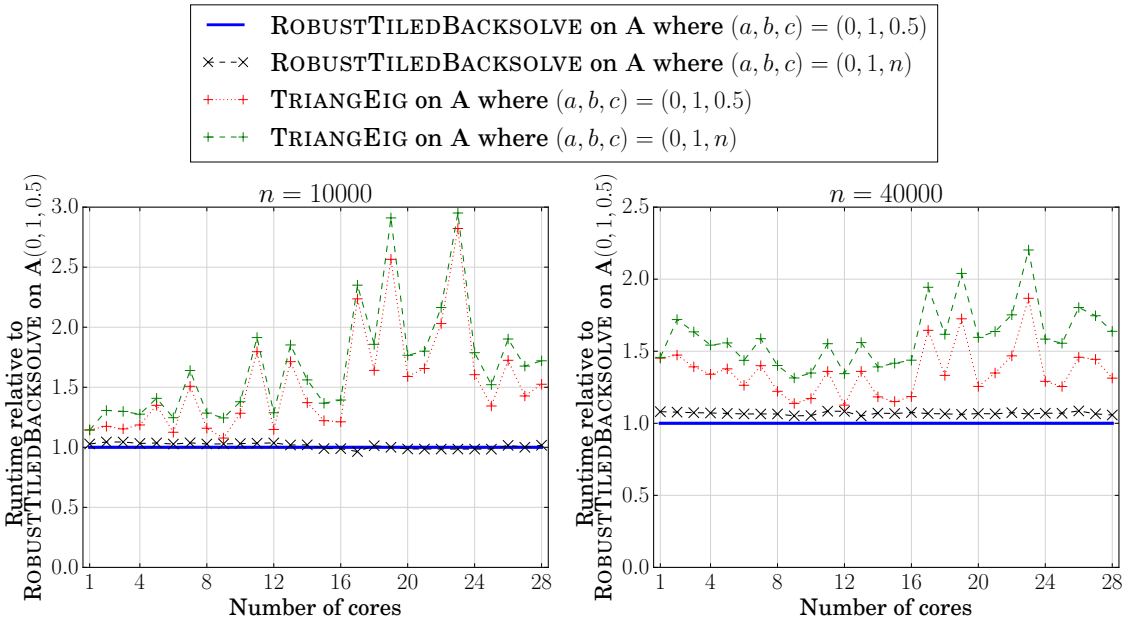


Figure 4: Relative parallel scalability on a shared memory node.

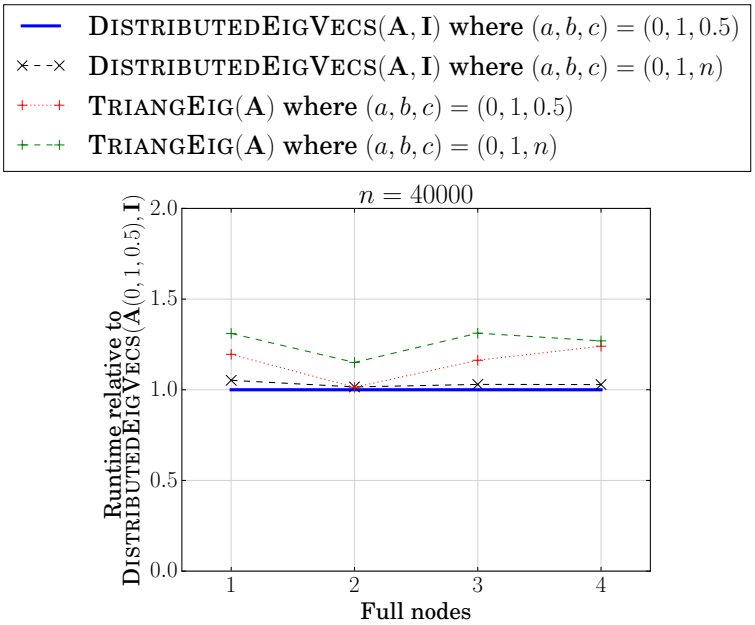


Figure 5: Relative parallel scalability on a distributed memory system.

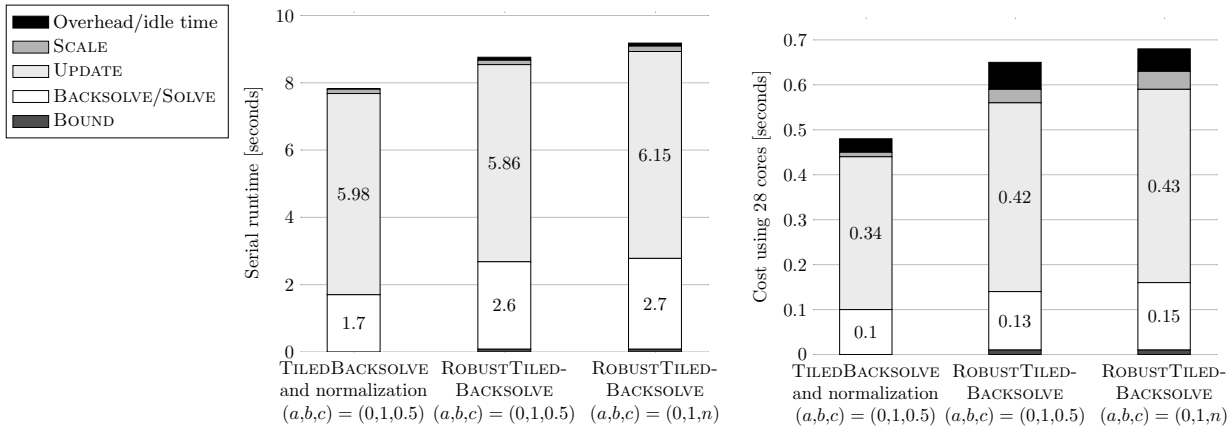


Figure 6: Runtime decomposition on $\boldsymbol{A}$ for $n = 10000$ on one shared memory node.

are more expensive. This traces back to the computation of vector norms and the evaluation of overflow protection logic. Update tasks, by contrast, are not visibly affected. The cost of overflow protection logic of an Update task is negligible compared to the cost of the matrix-matrix multiplication as long as numerical scaling is not necessary. Even when numerical scaling is necessary ($c = n$), the slowdown is moderate.

The parallel speedup of the compute-bound Backsolve/Solve and Update tasks ranges in 14 – 17. Due to reduced frequency boosts at higher core counts, the best obtainable speedup using AVX-512 is 20.8. The parallel speedup of the memory-bound Bound and Scale tasks ranges in 4 – 5. Recall that Bound and Scale tasks are not overlapped with the computation. Bound tasks add constant overhead to EigVecs and can be amortized over the computation of many eigenvectors. The increased overhead/idle time in the parallel run can be attributed to the synchronization points when threads become idle.

The eigenvector computation on larger matrices is dominated by Update tasks. Since Update tasks are only moderately affected by numerical scaling, the impact of robustness on the entire computation is limited, increasing the runtime 15%-20%. The next experiments quantifies the fraction by which the entire eigenvector computation is slowed down relative to the machine capabilities.

**Performance relative to the peak performance** The fourth experiments examines the strong scalability of EigVecs relative to the machine capabilities on one shared memory node. For comparison, the equivalent non-robust computation is added. Figure 7 shows the strong scaling results on the third test problem with dimension 10000 and 20000. Recall that the compute node has dynamic frequency scaling enabled. The increased frequency boost for small core counts explains why the computation can exceed the theoretical peak performance. We approximate the flop count for both solvers alike. An eigenvector, real or complex, of length $\ell$ contributes $\ell^2 + \ell$ flops for the backsolve and $n(2\ell - 1)$ for the backtransform. The flops due to numerical scaling or the evaluation of overflow protection logic are disregarded. The robust solver is slower by a constant factor. Since the performance curves follow the non-robust counterpart, robustness does not affect the parallel scalability.

**Bounding the eigenvector computation** The fifth experiment aims at bounding the runtime of RobustTiledBacksolve. The experiment uses the second test problem and varies two parameters. First, we consider the ratio of 2-by-2 blocks relative to the total number of blocks $\boldsymbol{B}_{kk}$, $k = 1, \ldots, m$. A 50% ratio of 2-by-2 blocks, for example, means that $\boldsymbol{B} \in \mathbb{R}^{n \times n}$ has $n/2$ 1-by-1 and $n/4$ 2-by-2 blocks. Second, the ratio of selected blocks determines how many blocks $\boldsymbol{B}_{kk}$ are randomly selected and, in turn, how many eigenvectors are computed. Recall that one eigenvector is computed per selected block, no matter if the block is 1-by-1 or 2-by-2.

Figure 8 shows the results on a matrix of dimension 10000. The runtime differences for the three select

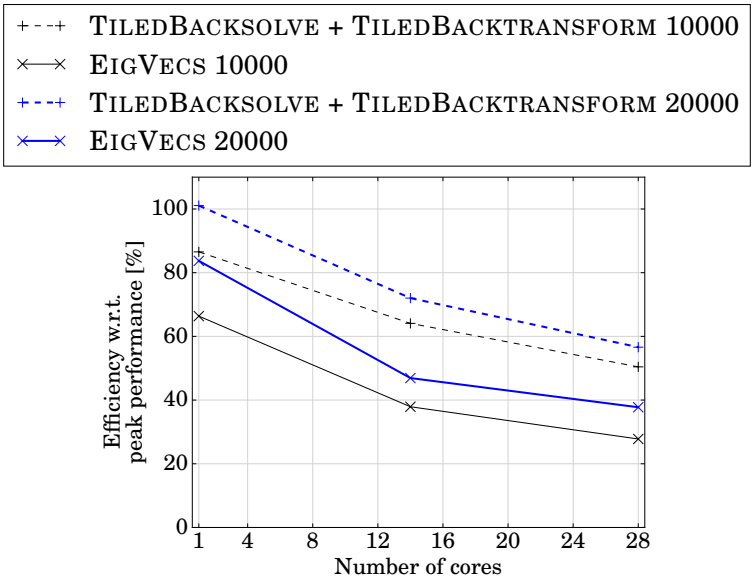


Figure 7: Strong scaling of successive execution of backsolve and backtransform on one shared memory node.

ratios reflect the different work load. A blend of 1-by-1 and 2-by-2 blocks indicates that the most expensive execution path is associated with a complex eigenvalue encountering a 1-by-1 block.

# 10 Conclusion

Eigenvectors can be computed from the real Schur form through a modified backward substitution method which can handle 2-by-2 blocks on the diagonal, followed by a backtransform to the original basis. The computation is prone to growth in the eigenvector components and can exceed the floating-point range. Such an overflow can be avoided by scaling the eigenvectors dynamically such that every arithmetic operation is guaranteed to yield a valid floating-point number. This paper presented a tiled eigenvector solver with overflow protection.

Our algorithm computes the backsolve mostly with efficient matrix-matrix multiplications (level-3 BLAS). As a result, there is a noticeable performance improvement over the LAPACK routine DTREVC3 and its derivatives, which use less efficient level-2 BLAS. Our algorithm relies on local scaling factors, which allow us to decouple tiles and compute scaled partial solutions. Thereby our task-centric solvers sustains parallel scalability even when numerical scaling is necessary. As a result, the parallel scalability is improved over the tiled distributed memory solver TriangEig available in Elemental 0.87.7, which uses global scaling factors.

Compared to a non-robust solver, the robust solver is around 15% - 20% slower when numerical scaling is not needed. In turn, the user is guaranteed a result which can be evaluated. When numerical scaling is needed, the additional rescaling increases the overhead by an additional constant factor of up to 10%. In particular, the spectral condition number of each eigenvalue can be computed.

# Acknowledgements

The authors wish to confirm that there are no known conflicts of interest associated with this publication.

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671633. Support was also received by eSSENCE, a collaborative

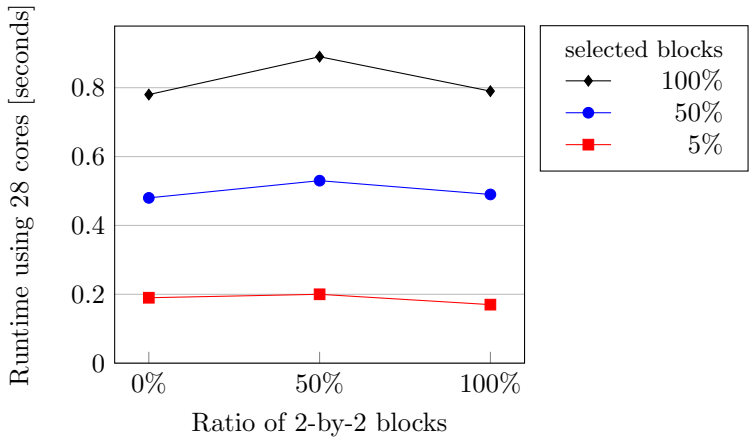


Figure 8: Impact of the proportion of 2-by-2 blocks and the proportions of selected blocks on the runtime.

e-Science programme funded by the Swedish Government via the Swedish Research Council (VR) grant no UFV 2010/149.

# References

[1] G. W. Stewart, Matrix Algorithms. Volume II: Eigensystems, Vol. 2, SIAM, 2001.

[2] M. R. Fahey, New Complex Parallel Eigenvalue and Eigenvector Routines, LAPACK Working Note 153 (August 2001).

[3] A. Schwarz, L. Karlsson, Scalable eigenvector computation for the non-symmetric eigenvalue problem, Parallel Computing 85 (2019) 131–140.

[4] G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd Edition, John Hopkins University Press, 1996.

[5] A. Ruhe, Properties of a Matrix with a Very Ill-conditioned Eigenproblem, Numerische Mathematik 15 (1) (1970) 57–60.

[6] C. C. Kjelgaard Mikkelsen, A. B. Schwarz, L. Karlsson, Parallel robust solution of triangular linear systems, Concurrency and Computation: Practice and Experience (2018) e5064.

[7] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, N. A. Romero, Elemental: A new framework for distributed memory dense matrix computations, ACM Transactions on Mathematical Software (TOMS) 39 (2) (2013) 13.

[8] T. Moon, J. Poulson, Accelerating eigenvector and pseudospectra computation using blocked multi-shift triangular solves (July 2016). arXiv:1607.01477.

[9] E. Anderson, Robust Triangular Solves for Use in Condition Estimation, LAPACK Working Note 36, Cray Research Inc. (August 1991).

[10] C. C. Kjelgaard Mikkelsen, L. Karlsson, Blocked Algorithms for Robust Solution of Triangular Linear Systems, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2017, pp. 68–78.

[11] C. C. Kjelgaard Mikkelsen, L. Karlsson, Robust Solution of Triangular Linear Systems, NLAFET Working Note 9 (May 2017).

[12] M. Baudin, R. L. Smith, A Robust Complex Division in Scilab (October 2012). arXiv:1210.4539.

[13] M. Gates, A. Haidar, J. Dongarra, Accelerating computation of eigenvectors in the dense nonsymmetric eigenvalue problem, in: International Conference on High Performance Computing for Computational Science, Springer, 2014, pp. 182–191.

[14] D. J. Higham, N. J. Higham, Structured backward error and condition of generalized eigenvalue problems, SIAM Journal on Matrix Analysis and Applications 20 (2) (1998) 493–512.