VECTORIZATION OF THE JACOBI-TYPE SINGULAR VALUE DECOMPOSITION METHOD

VEDRAN NOVAKOVIĆ†

Abstract. The eigenvalue decomposition (EVD) of (a batch of) Hermitian matrices of order two has a role in many numerical algorithms, of which the one-sided Jacobi method for the singular value decomposition (SVD) is the prime example. In this paper the batched EVD is vectorized, with a vector-friendly data layout and the AVX-512 SIMD instructions of Intel CPUs, alongside other key components of a real and a complex OpenMP-parallel Jacobi-type SVD method, inspired by the sequential xGESVJ routines from LAPACK. These vectorized building blocks should be portable to other platforms, with unconditional reproducibility guaranteed for the batched EVD and several others. No avoidable overflow of the results can occur with the proposed EVD or SVD. The measured accuracy of the proposed EVD often surpasses that of the xLAEV2 routines from LAPACK. While the batched EVD outperforms the matching sequence of xLAEV2 calls, speedup of the parallel SVD is modest but can be improved and is already beneficial with enough threads. Regardless of their number, the proposed SVD method gives identical results, but of somewhat lower accuracy than xGESVJ.

Key words. batched eigendecomposition of Hermitian matrices of order two, SIMD vectorization, singular value decomposition, parallel one-sided Jacobi-type SVD method

AMS subject classifications. 65F15, 65F25, 65Y05, 65Y10

1. Introduction. The eigenvalue decomposition (EVD) of a Hermitian or a symmetric matrix of order two [24] is a part of many numerical algorithms, some of which are implemented in the LAPACK [3] library, like the one-sided Jacobi-type algorithm [15, 16, 17] for the singular value decomposition (SVD) of general matrices. It also handles the $2 \times 2$ terminal cases in the QR-based [18, 19] and the MRRR-based [12, 13] algorithms for the eigendecomposition of Hermitian/symmetric matrices. Its direct application, the two-sided Jacobi-type EVD method for symmetric [24] and Hermitian [20] matrices, has not been included in LAPACK but is widely known.

The first part of the paper aims to show that reliability of the $2 \times 2$ symmetric/Hermitian EVD, produced by the LAPACK routines xLAEV2, can be improved by scaling the input matrix $A$ by an easily computable power of two. This preprocessing not only prevents the scaled eigenvalues from overflowing (and underflowing if possible), but also preserves the eigenvectors of a complex $A$ from becoming inaccurate and non-orthogonal in certain cases of subnormal components of $a_{12} = \bar{a}_{21}$. The proposed EVD formulas are branch-free, implying no if-else statements, but relying instead on the standard-conformant [21] handling of the special floating-point values by the min/max functions. If the formulas are implemented in the SIMD fashion, they can compute several independent EVDs in one go, instead of one by one in a sequence.

An algorithm that performs the same operation on a collection (a “batch”) of inputs of the same or similar (usually small) dimensions is known as batched (see [1, Sect. 10] for an overview). A vectorized, batched EVD of Hermitian matrices of order two is thus proposed, in a similar vein as the batched $2 \times 2$ SVD from [30]. Recall that a SIMD vector instruction performs an operation on groups (called vectors) of scalars, packed into the lanes of vector registers, at roughly the cost of one (or few) scalar operation. For example, two vectors, each with $l$ lanes, can be multiplied, each lane of

---

*This work has been supported in part by Croatian Science Foundation under the project IP–2014–09–3670 “Matrix Factorizations and Block Diagonalization Algorithms” (MFBDA). The prototype implementation is available in a GitHub repository https://github.com/venovako/VecJac.

†ORCID: https://orcid.org/0000-0003-2964-9674; completed a part of this research as an independent collaborator on the MFBDA project, 10000 Zagreb, Croatia (venovako@venovako.eu)
the first vector by the corresponding lane of the second one, at a fraction of the price of \( l \) scalar multiplications. If a batch of input and output matrices is represented as a set of separate data streams, each containing the same-indexed elements of a particular sequence of matrices (e.g., all \( a_{11}^{(r)} \)), then a stream can be handled as one or more vectors, depending on the hardware’s vector width. Every major CPU architecture, like those from Intel, AMD, IBM, and ARM, offers vector operations with varying, but ever expanding\(^1\) vector widths and instruction subsets. Also, there are specialized vector engines, like NEC SX-Aurora TSUBASA with 2048-bit-wide registers. Thus, it is not a question should an algorithm be vectorized, but how to do that, if possible.

The final, second part of the paper focuses on vectorization of the one-sided Jacobi-type SVD for general matrices (but with at least as many rows as there are columns), in its basic form [15], as implemented in the xGESVJ LAPACK routines, without the rank-revealing QR factorization and other preprocessing [16, 17] of the more advanced xGEJSV routines. Both sets of routines are sequential by design, even though the used BLAS/LAPACK subroutines might be parallelized. The Jacobi-type method is not the fastest SVD algorithm available, but it provides the superior accuracy of the singular vectors and high relative accuracy of the singular values [11]. It is shown how the whole method (but with a parallel pivot strategy), i.e., all of its components, can be vectorized with the Intel AVX-512 instruction subset. Those BLAS-like components that are strategy-agnostic can be retrofitted to the xGESVJ routines.

Certain time-consuming components of the Jacobi-type SVD method, like the column pair updates, have for long been considered for vectorization [10]. In fact, any component realized with calls of optimized BLAS or LAPACK routines is automatically vectorized if the routines themselves are, such as dot-products and post-multiplications of a column pair by a rotation-like transformation. In the sequential method, however, only one transformation is generated per a method’s step, in essence by the EVD of a pivot Grammian matrix of order two, while a parallel method generates up to \( \lfloor n/2 \rfloor \) transformations per step, with \( n \) being the number of columns of the iteration matrix. This sequence of independent EVDs is a natural candidate for vectorization that computes several EVDs at once, and for its further parallelization, applicable for larger \( n \). Also, some vector platforms provide no direct support for complex arithmetic with vectors of complex values in the typical, interleaved \( z = (\Re z, \Im z) \) representation, and thus the split one [36], with an accompanying set of BLAS-like operations, is convenient.

Batched matrix factorizations (like the tall-and-skinny QR) and decompositions (like a small-order Jacobi-type SVD) in the context of the blocked Jacobi-type SVD on GPUs have been developed in [29, 8], while the batched bidiagonalization on GPUs has been considered in [14]. Efficient batched kernels are ideal for acceleration of the blocked Jacobi-type SVD methods, but the non-blocked (pointwise) ones induce only batches of the smallest, \( 2 \times 2 \) EVD problems for the one-sided, and the SVD ones for the two-sided [26, 30] methods, what motivated developing of the proposed batched EVD.

The two mentioned parts of the paper further subdivide as follows. In section 2 the vectorized batched EVD of Hermitian matrices of order two is developed and its numerical properties are assessed. In section 3 the robust principles of (re-)scaling of the iteration matrix in the SVD method are laid out, such that the matrix can never overflow under inexact transformations, and a majority of the remaining components of the method is vectorized. In section 4 the developed building blocks are put together

\(^{1}\)A reader interested in porting the proposed method to another platform is advised to consult the up-to-date architecture manuals for the vectorization support of a particular generation of CPUs.
to form a vectorized one-sided Jacobi-type SVD method (with a parallel strategy),
that can be executed single- or multi-threaded, with identical outputs guaranteed in
either case. Numerical testing is presented in section 5. The paper concludes with a
summary and some directions for further research in section 6. The supplementary
material contains most proofs and additional algorithms, code, and numerical results.

2. Vectorization of eigendecompositions of order two. Let \( A \) be a symmetric
(Hermitian) matrix of order two, \( U \) an orthogonal (unitary) matrix of its
eigenvectors, i.e., its diagonalizing Jacobi rotation, real [24] or complex [20], and \( \Lambda \)
a real diagonal matrix of its eigenvalues. In the complex case,

\[
A = U \Lambda U^*, \quad U = \begin{bmatrix}
\cos \varphi & -e^{-i\alpha} \sin \varphi \\
e^{i\alpha} \sin \varphi & \cos \varphi
\end{bmatrix}, \quad \Lambda = \begin{bmatrix}
\lambda_1 & 0 \\
0 & \lambda_2
\end{bmatrix},
\]

with \( \varphi \in [-\pi/4, \pi/4] \) and \( \alpha \in (-\pi, \pi] \), while \( \alpha = 0 \) in the real case. From (2.1) it
follows by two matrix multiplications that

\[
A = \cos^2 \varphi \begin{bmatrix}
\lambda_1 + \lambda_2 \tan^2 \varphi & e^{-i\alpha} \tan \varphi (\lambda_1 - \lambda_2) \\
e^{i\alpha} \tan \varphi (\lambda_1 - \lambda_2) & \lambda_1 \tan^2 \varphi + \lambda_2
\end{bmatrix}.
\]

Should \( A \) be constructed from (2.2), e.g., for testing purposes, with its eigenvalues
prescribed, then it suffices to ensure that \( |\lambda_1| + |\lambda_2| \leq \omega/8 \), where \( \omega \) is the largest finite
floating-point value, to get \( \max |a_{ij}| \leq \omega/(4\sqrt{2}) \). In Proposition 2.4 it is shown that
such a bound on the magnitudes of the elements of \( A \) guarantees that its eigenvalues
will not overflow. If the bound does not hold, \( A \) has to be downscaled to compute \( \Lambda \).

One applicable power-of-two scaling algorithm was proposed in [30, subsection 2.1]
in the context of the SVD of a general real or complex \( 2 \times 2 \) matrix, and is adapted
for the EVD of a symmetric or Hermitian matrix \( A \) of order two in subsection 2.3.

Contrary to the standalone EVD, for orthogonalization of a pivot column pair
in the Jacobi SVD algorithm it is sufficient to find the eigenvectors \( U \) of their pivot
Grammian matrix \( A \), while its eigenvalues in \( \Lambda \) are of no importance. If the orthogonal-
ized pivot columns are also to be ordered in the iteration matrix non-increasingly
with respect to their Frobenius norms, a permutation \( P \) (as in, e.g., [29]) has to be
found such that \( \lambda'_1 \geq \lambda'_2 \geq 0 \), where \( A' = P^* U^* A U P \), i.e., the pivot columns have
to be transformed by multiplying them from the right by \( U P \) instead of by \( U \). For a
comparison of the eigenvalues of \( A \) to be made, it is only required that the eigenvalue
with a smaller magnitude is finite, while the other one may be allowed to overflow if
floating-point exceptions do not trap (what is the default “non-stop” regime).

2.1. Branch-free computation of the Jacobi rotations. Assume that \( A \),
represented by its lower triangle elements \( a_{11}, a_{21}, \) and \( a_{22} \), has already been scaled.
From the annihilation condition \( U^* A U = \Lambda \), as shown in Appendix A.1 of the sup-
plementary material similarly to [20], but using the fused multiply-add operation
\( \text{fma}(a, b, c) = \text{RN}(a \cdot b + c) \) with a single rounding of its result [21], and assuming
\( \arg 0 = 0 \) for determinacy, it follows

\[
\lambda_1 / \cos^2 \varphi = (a_{22} \cdot \tan \varphi + 2|a_{21}|) \cdot \tan \varphi + a_{11},
\]

\[
\lambda_2 / \cos^2 \varphi = (a_{11} \cdot \tan \varphi - 2|a_{21}|) \cdot \tan \varphi + a_{22},
\]

and for \( \alpha \) from (2.1), in the complex case computed as in (2.11),

\[
\alpha = \arg a_{21}, \quad e^{i\alpha} = a_{21}/|a_{21}|.
\]
In the real case \( (2.4) \) leads to \( e^{i\alpha} = \text{sign} \, a_{21} = \pm 1 \) (note, sign \( \pm 0 = \pm 1 \)). Also,

\[
(2.5) \quad \tan \varphi = \frac{\tan(2\varphi)}{1 + \sqrt{\tan(2\varphi) \cdot \tan(2\varphi) + 1}}, \quad \cos \varphi = \frac{1}{\sqrt{\tan \varphi \cdot \tan \varphi + 1}}.
\]

With the methods from [30], \( \tan(2\varphi) \) is computed branch-free, as

\[
(2.6) \quad \tan(2\varphi) = \min \left( \text{fmax} \left( \frac{2|a_{21}|}{|a|}, 0 \right), \text{fl}(\sqrt{\omega}) \right) \cdot \text{sign} \, a, \quad a = a_{11} - a_{22}.
\]

The \text{fmax} call returns its second argument if the first one is \text{NaN} (when \( |a_{21}| = |a| = 0 \)).

The upper bound of \( \text{fl}(\sqrt{\omega}) \) on \( |\tan(2\varphi)| \) enables simplifying \( (2.5) \), since, when rounding to nearest, tie to even, \( \text{fma}(\text{fl}(\sqrt{\omega}), \text{fl}(\sqrt{\omega}), 1) \) cannot overflow in single or double precision, so its square root in \( (2.5) \) can be computed faster than resorting to the \text{hypot}(x, y) = \sqrt{x^2 + y^2} \) function. It can be verified (by a C program, e.g.) that the same bound ensures that \( (2.5) \) gives the correct answer for \( \tan \varphi \) (i.e., \( \pm 1 \)) instead of \text{NaN} when the unbounded \( \tan(2\varphi) \) would be \( \pm \infty \) due to \( |a| = 0 \) and \(|a_{21}| > 0\).

By factoring out \( \cos \varphi \) from \( U \) in \( (2.1) \), \( \sin \varphi = \tan \varphi \cdot \cos \varphi \) does not have to be computed, unless the eigenvectors are not required to be explicitly represented, since they can postmultiply a pivot column pair in the Jacobi SVD algorithm in this factored form, with \( \cos \varphi \) and \( \tan \varphi \), as in \( (3.1) \). In the Jacobi SVD algorithm, the ordering of the (always positive) eigenvalues of a pivot matrix is important for ordering the transformed pivot columns of the iteration matrix, i.e., for swapping them if \( \lambda_1 < \lambda_2 \), while the actual eigenvalues have to be computed for the standalone, generic EVD only.

### 2.1.1. On \text{invsqrt}. The \text{invsqrt}(x) = \text{fl}(1/\sqrt{x}) \) intrinsic function (a vectorized, presently not correctly rounded implementation of the standard’s recommended \text{rSqrt} function [21]) is appropriate for computation of the cosines directly from the squares of the secent \( (\tan^2 \varphi + 1) \) in \( (2.5) \), but it may not be available on another platform. A slower but unconditionally reproducible alternative, as explained in [33], is to compute

\[
(2.7) \quad \text{fl}(\sec^2 \varphi) = \text{fma}(\text{fl}(\tan \varphi), \text{fl}(\tan \varphi), 1), \quad \text{fl}(\sec \varphi) = \text{fl}(\sqrt{\text{fl}(\sec^2 \varphi)}),
\]

\[
\text{fl}(\cos \varphi) = \text{fl}(1/\text{fl}(\sec \varphi)), \quad \text{fl}(\sin \varphi) = \text{fl}(\text{fl}(\tan \varphi)/\text{fl}(\sec \varphi)),
\]

and optionally replace \( (2.3) \) by the potentially more accurately evaluated expressions

\[
(2.8) \quad \lambda_1 = ((a_{22} \cdot \tan \varphi + 2|a_{21}|) \cdot \tan \varphi + a_{11})/\sec^2 \varphi, \\
\lambda_2 = ((a_{11} \cdot \tan \varphi - 2|a_{21}|) \cdot \tan \varphi + a_{22})/\sec^2 \varphi.
\]

Let \( \delta_1 \) be the relative rounding error accumulated in the process of computing \( \text{fl}(\cos \varphi) \) from \( \text{fl}(\sec^2 \varphi) \). If two elementary arithmetic operations from \( (2.7) \) are used,

\[
(2.9) \quad \delta_1^- = \frac{1 - \varepsilon}{1 + \varepsilon} \leq \delta_1 \leq \frac{1 + \varepsilon}{1 - \varepsilon} = \delta_1^+,
\]

where the denominators and the numerators come from the bounds on the relative errors of the square root and the division, respectively, while \( \varepsilon = 2^{-(p+1)} = 0.5 \text{ulp}(1) \) is the machine precision when rounding to the nearest, with \( p \) being the number of bits of significand (\( p = 23 \), \( p = 52 \), or \( p = 112 \) for single, double, or quadruple precision).
2.2. Implementation of complex arithmetic. Let $a$, $b$, $c$, and $d$ be complex values such that $d = a \cdot b + c$. By analogy with the real fused multiply-add operation, let such a ternary complex function and its value in floating-point be denoted as $\tilde{d} = \text{fma}(d, e, f)$. Then, $d$ can be computed with only two roundings for each of its components (as suggested in the cuComplex.h header in the CUDA toolkit\(^2\)), as

$$
\tilde{d} = \text{fma}(\Re a, \Re b, \text{fma}(\Im a, \Re b, \Im c)) \quad \text{and} \quad \Im \tilde{d} = \text{fma}(\Re a, \Im b, \text{fma}(\Im a, \Im b, \Im c)).
$$

A branch-free way of computing the polar form of a complex value $z$ as $|z|e^{i\beta}$, with $\beta = \arg z$ and $\text{fl}(|z|) = \text{hypot}(\Re z, \Im z) \leq \omega$, was given in [30, Eq. (1)] as

$$
\text{fl}(\cos \beta) = \text{fmin}(\text{fl}(\Re z), 1) \cdot \text{sign} \Re z, \quad \text{fl}(\sin \beta) = \Im z / \max\{\text{fl}(|z|), \mu\},
$$

where $\mu$ is the smallest subnormal positive non-zero real value. Floating-point operations must not trap on exceptions and subnormal numbers have to be supported as both inputs and outputs. The $\text{fmin}$ and $\text{fmax}$ functions have to return their second argument if the first one is NaN, but the full C language [23] semantics is not required.

2.2.1. On hypot. The hypot vector intrinsic function may not be available in another environment. It can be substituted [30] by a naïve yet fully vectorized and unconditionally reproducible Algorithm 2.1 (using the notation from subsection 2.4), based on the well-known relations $\text{hypot}(0, 0) = 0$ and (abusing the symbols $m$ and $M$)

$$
\sqrt{x^2 + y^2} = M \sqrt{\left(\frac{x}{M}\right)^2 + \left(\frac{y}{M}\right)^2} = M \sqrt{\left(\frac{m}{M}\right)^2 + 1} = M \sqrt{q^2 + 1},
$$

$$
q = m/M, \quad m = \min\{|x|, |y|\}, \quad \omega \geq M = \max\{|x|, |y|\} > 0.
$$

**Algorithm 2.1** A naïve vectorized hypot.

**Input:** vectors $x$ and $y$ with finite values in each lane  
**Output:** $\approx \sqrt{x^2 + y^2}$, not correctly rounded but without undue overflow in each lane

1: \( -0 = \text{set1}(-0.0); \quad 0 = \text{setzero}(); \quad 1 = \text{set1}(1.0); \)
2: \( x = \text{andnot}(-0, x); \quad // x = |x| \) (execute unless $x$ holds positive values only)
3: \( y = \text{andnot}(-0, y); \quad // y = |y| \) (execute unless $y$ holds positive values only)
4: \( m = \text{min}(x, y); \quad M = \text{max}(x, y); \)
5: \( q = \text{max}(\text{div}(m, M), 0); \quad \max \) replaces $0/0 \rightarrow \text{NaN}$ with $0$
6: \( \text{return } \text{mul}(\text{sqrt}(\text{fmadd}(q, q, 1)), M); \)

**Lemma 2.1** gives the relative error bounds for $\text{hypot}(x, y)$ if computed in the standard [21] floating-point arithmetic as in Algorithm 2.1, with (exactly representable) scalars $x$ and $y$ instead of vectors. In (2.12) and in the scalar Algorithm 2.1, $M > 0$ implies $\text{fl}(\text{hypot}(x, y)) > 0$, while $M = 0$ implies $q = 0 = \text{fl}(\text{hypot}(x, y))$ in the latter.

**Lemma 2.1.** Let $\text{hypot}(x, y)$ be defined by (2.12), and assume that neither overflow nor underflow occurs in the final multiplication of $M$ by $\text{fma}(\text{fl}(q), \text{fl}(q), 1)$. Then,

$$
\text{fl}(\text{hypot}(x, y)) = \delta_2 \text{hypot}(x, y),
$$

$$
\delta_2^- = (1 - \varepsilon)^{5/2} \sqrt{1 - \varepsilon(2 - \varepsilon)/2} < \delta_2 < (1 + \varepsilon)^{5/2} \sqrt{1 + \varepsilon(2 + \varepsilon)/2} = \delta_2^+.
$$

**Proof.** See Appendix A.1.1 in the supplementary material.

---

\(^2\)The CUDA toolkit is available at [https://developer.nvidia.com/cuda-toolkit](https://developer.nvidia.com/cuda-toolkit).
Remark 2.2. For all complex, representable values \( z \neq 0 \), \( \|z/\text{fl}(|z|)| \leq \sqrt{2} \) in the standard rounding modes, since, from (2.12), \( \text{fl}(|z|) \geq \max\{|\Re z|, |\Im z|\} \), what, together with (2.11), gives \( 1 \geq \|\text{fl}(\cos \beta)\| \geq \|\text{fl}(\sin \beta)\| \) if \( |\Re z| \geq |\Im z| \), or \( 1 \geq \|\text{fl}(\sin \beta)\| \geq \|\text{fl}(\cos \beta)\| \) otherwise. In certain cases that do not satisfy the assumption of Lemma 2.1, \( \text{fl}(e^{i\alpha}) \) from (2.4), computed by (2.12) or by another hypot, might be quite inaccurate, with \( \|\text{fl}(e^{i\alpha})\| \leq \sqrt{2} \). Remark 2.3 shows that this bound is sharp.

Remark 2.3. For all representable \( x = \text{fl}(x) \), \( \|\text{fl}(\text{hypot}(x,x))\| = \|\text{fl}(\sqrt{2})\| |x| \) if computed as in (2.12). Let \( \Psi_T \) be a set of subnormal non-zero \( x \), that depends on the implementation of the hypot, the datatype \( T \), and the rounding mode \( \circ \) in effect, on which \( \text{fl}(\text{hypot}(x,x)) = |x| \). This set is non-empty with the default rounding (e.g., it contains \( x = \tilde{\mu} \) for Algorithm 2.1 and for the tested math library’s hypot), but it should be empty when rounding to \( +\infty \). For \( y = x \in \Psi_T \), \( \delta_2 = 1/\sqrt{2} \) in (2.13). If for a complex \( z \) holds \( |\Re z| = |\Im z| = |x| \in \Psi_T \), then \( \|\text{fl}(z)\| = |x| \), and so \( \|\text{fl}(\cos \beta)\| = \|\text{fl}(\sin \beta)\| = 1 \) in (2.11). This has serious consequences for accuracy of the eigenvectors computed by the complex LAPACK routines, as shown in subsection 5.2. If \( \text{fl}(U) \) comes from Algorithm 2.2 or its single precision version, an inaccurate \( \text{fl}(e^{i\alpha}) \) from (2.4), for which \( \|\text{fl}(e^{i\alpha})\| = \sqrt{2} \), is avoided in many cases if (2.17) implies a large enough upscaling.

2.3. Almost exact scaling of a Hermitian matrix of order two. A scaling of \( A \) sufficient for finiteness of \( \Lambda \) in floating-point is given in Proposition 2.4.

Proposition 2.4. If for a Hermitian matrix \( A \) of order two holds

\[
\hat{a} = \max_{1 \leq j \leq 2} |a_{ij}| \leq \omega/(4\sqrt{2}) = \tilde{\omega},
\]

then no output from any computation in (2.4)–(2.8), including the resulting eigenvalues, can overflow, assuming the standard [21] non-stop floating-point arithmetic in at least single precision \( (p \geq 23) \) and rounding to the nearest, tie to even.

Moreover, barring any underflow of the results of those computations, the following relative error bounds hold for the quantities computed as in (2.4)–(2.7):

\[
\begin{align*}
\text{fl}(\cos \alpha) & = \delta'_\alpha \cos \alpha, \\
\text{fl}(\sin \alpha) & = \delta'_\alpha \sin \alpha, \\
& 1 - 4.000000 \varepsilon < \frac{\delta'_\alpha, \delta'_\alpha}{\|\text{fl}(e^{i\alpha})\|} < 1 + 4.000001 \varepsilon, \\
\text{fl}(\tan \varphi) & = \delta'_\varphi \tan \varphi, \\
& 1 - 5.500000 \varepsilon < \delta'_\varphi < 1 + 5.500001 \varepsilon, \\
\text{fl}(\cos \varphi) & = \delta'_\varphi \cos \varphi, \\
& 1 - 8.000000 \varepsilon < \delta'_\varphi < 1 + 8.000002 \varepsilon,
\end{align*}
\]

where \( \mathbb{F} \) indicates whether \( A \) is real or complex. For different ranges of \( \delta_1 \) or \( \delta_2 \) the bounds can be recalculated as described in Appendix A.3, as well as for a different rounding mode (while lowering the upper bound of \( \text{fl}(\sqrt{\omega}) \) for \( \text{fl}(\tan 2\varphi) \) if required).

Proof. The proof is presented in Appendix A.2 of the supplementary material. 

Corollary 2.5 gives a practical scaling bound in the terms of the magnitudes of the components of the elements of \( A \) and an exactly representable value based on \( \omega \).

Corollary 2.5. If a Hermitian matrix \( A \) of order two is scaled such that

\[
\max_{1 \leq j \leq 2} \max\{|\Re a_{ij}|, |\Im a_{ij}|\} \leq \tilde{\omega}/\sqrt{2} = \omega/8,
\]

then the assumption (2.14) of Proposition 2.4 holds for \( A \).
VECTORIZATION OF THE JACOBI SVD

Proof. From max\{ |ℜa_{ij}|, |ℑa_{ij}| \} ≤ ω/8 it follows |a_{ij}| ≤ ω\sqrt{2}/8 = ω/(4\sqrt{2}) = ω/4.

With η = ⌊\log(ω/8)⌋, (2.16) implies the scaling factor 2^ζ of a A (A' = 2^ζ A), where
\[ ζ_i = η - \log|a_{ii}|, \quad ζ_{22} = η - \log|ℜa_{21}|, \quad ζ_{21} = η - \log|ℑa_{21}|, \]
while \( \log 0 = -\infty \) and ζ is an integer exactly representable as a floating-point value³.

The eigenvalues of \( A' \) in \( \Lambda' \) cannot overflow, but in \( \Lambda \approx 2^{-ζ} A' \) might, where the approximate sign warns of a possibility that the values in \( A \) small enough in magnitude become subnormal and lose their least significant bits when downscaling it with \( ζ < 0 \). Otherwise, the scaling of \( A \) is exact, and some subnormal values might be raised into the normal range when \( ζ > 0 \). The scaled eigenvalues in \( \Lambda' \) could be kept alongside ζ in the cases where \( A \) is expected to overflow or underflow, and compatibility with the LAPACK's xLAEV2 routines is not required. Else, \( A \) is returned by backscaling.

2.4. Intel AVX-512 vectorization. In the following, a function in the template font is as shorthand for the Intel AVX-512 C intrinsic⁴ _mm512_function_pd_. A variable in the sans-serif font or a bold Greek letter without an explicitly specified type is assumed to be a vector of the type _mm512d_, with \( s = 8 \) double precision values (lanes). A variable named with a letter from the Fraktur font (e.g., m) represents an eight-bit lane mask of the type _mm_mask8_ (see [22] for the datatypes and the vector instructions). Most, but not all, intrinsics correspond to a single instruction.

The chosen vectorization approach is Intel-specific for convenience, but it is meant to determine the minimal set of vector operations required (hardware-supported or emulated) on any platform that is considered to be targeted, either by that platform’s native programming facilities, or via some multi-platform interface in the spirit of the SLEEF [34] vectorized C math library, e.g., or, if possible, by the compiler’s auto-vectorizer, as it has partially been done with the Intel Fortran compiler for the batched generalized eigendecomposition of pairs of Hermitian matrices of order two, one of them being positive definite, by the Hari-Zimmermann method [35, subsection 6.2].

2.4.1. Data storage and the split complex representation. For simplicity and performance, a real vector \( x \) of length \( m \) is assumed to be contiguous in memory, aligned to a multiple of the larger of the vector size and the cache-line size (on Intel CPUs they are both 64 B), and zero-padded at its end to the optimal length \( \tilde{m} \), where
\[ \tilde{m} = \begin{cases} m, & \text{if } m \mod s = 0, \\
 m + (s - (m \mod s)), & \text{otherwise}. \end{cases} \]

If \( z \) is a complex array, it is kept as two real non-overlapping ones, \( ℜ\tilde{z} \) and \( ℑ\tilde{z} \) (for the real and the imaginary parts, respectively), laid out in this split form as
\[ ℜ\tilde{z} = \begin{bmatrix} ℜz \\ 0_{\tilde{m} - m} \end{bmatrix}, \quad ℑ\tilde{z} = \begin{bmatrix} ℑz \\ 0_{\tilde{m} - m} \end{bmatrix}, \]
where the blocks \( 0_{\tilde{m} - m} \) of zeros are the minimal padding that makes the length of the real as well as of the imaginary block a multiple of the number of SIMD lanes (\( s \)), with \( \tilde{m} \) from (2.18). A complex \( m \times n \) matrix \( G \) is kept as two real \( \tilde{m} \times n \) matrices,
\[ ℜG = \begin{bmatrix} ℜg_1 \\ 0_{\tilde{m} - m} \end{bmatrix} \begin{bmatrix} ℜg_2 \\ 0_{\tilde{m} - m} \end{bmatrix} \cdots \begin{bmatrix} ℜg_n \\ 0_{\tilde{m} - m} \end{bmatrix}, \quad ℑG = \begin{bmatrix} ℑg_1 \\ 0_{\tilde{m} - m} \end{bmatrix} \begin{bmatrix} ℑg_2 \\ 0_{\tilde{m} - m} \end{bmatrix} \cdots \begin{bmatrix} ℑg_n \\ 0_{\tilde{m} - m} \end{bmatrix}, \]

³For all standard floating-point datatypes, the value represented by \( ω \) is an integer.
⁴https://www.intel.com/content/www/us/en/docs/intrinsics-guide/
i.e., each column \( y_j \) of \( G \), where \( 1 \leq j \leq n \), is split into the real \((\Re g_j)\) and the imaginary \((\Im g_j)\) part, as in (2.19). The leading dimensions of \( \Re G \) and \( \Im G \) may differ, but each has to be a multiple of \( s \) to keep all padded real columns properly aligned.

The split form has been used for efficient complex matrix multiplication kernels [30] and for the generalized SVD computation by the implicit Hari–Zimmermann algorithm on GPUs [32]. Intel CPUs have no native complex-specific arithmetic instructions operating on vectors of at least single precision complex numbers in the customary, interleaved representation, so a manual implementation of the vectorized complex arithmetic is inevitable, for what the split representation is more convenient.

A conversion of the customary representation of complex arrays to and back from the split form, in both vectorized and parallel fashion, is described in Appendix E.

An input batch \((A^{(t)})_{\ell=1}^r\) of Hermitian matrices of order two is kept as a collection of one-dimensional real arrays \(\tilde a_{11}, \tilde a_{22}, \bar{\Re} \tilde a_{21}, \bar{\Im} \tilde a_{21}\), of length \(\hat{r}\) and with layout (2.19), where \(\hat{r}\) is calculated from \(r\) as in (2.18). The corresponding Jacobi rotations' parameters are kept in the arrays \(\cos \tilde \phi, \cos \alpha \tan \tilde \phi, \sin \alpha \tan \tilde \phi\). If \(\lambda_1^{(t)} < \lambda_2^{(t)}\) then the permutation \(P^{(t)} = [1 \ 0]\), else \(P^{(t)} = [0 \ 1]\), what is compactly encoded by setting the \(((\ell - 1) \mod s)\)-th bit of a bitmask \(p\) to 1 or 0, respectively.

2.4.2. Vectorized eigendecomposition of a batch of Hermitian matrices of order two. Based on subsections 2.1 to 2.3, Algorithm 2.2 implements a vectorized, branch-free, unconditionally reproducible eigendecomposition method for at most \(a\) Hermitian matrices of order two in double precision. Although presented with the AVX512DQ instruction subset, only the basic, AVX512F subset is required\(^5\). The order of the statements slightly differs from the one in the actual code, for legibility.

Algorithm 2.3 builds on Algorithm 2.2 and computes the eigendecomposition of a batch of Hermitian matrices of order two in parallel, where each OpenMP thread executes Algorithm 2.2 in sequence over a (not necessarily contiguous) subset of the input’s vectors. No dependencies exist between the iterations of the parallel-for loop of Algorithm 2.3, so it can be generalized by distributing its input over several machines.

Algorithms B.1 and B.2, detailed in Appendix B of the supplementary material, are the specializations of Algorithms 2.2 and 2.3, respectively, for real symmetric matrices. There, \(e^{\alpha} \tan \tilde \varphi = \cos \alpha \tan \tilde \varphi = \text{sign} \tilde a_{21} \cdot \tan \tilde \varphi\) elementwise.

Converting Algorithms 2.2, 2.3, B.1 and B.2 to single precision, with \(s = 16\), requires redefining \(\omega = \text{FLT\_MAX}, \bar{t} = 1.844674297E+19, \bar{\mu} = \text{FLT\_TRUE\_MIN}, \eta = \text{FLT\_MAX\_EXP} - 4\) and switching to \_\_\_\_\_m512_\_\_\_ps inintrinsics, \_\_\_\_m512 vectors, and \_\_\_\_mmsk16 bitmasks, while the relative error bounds in Proposition 2.4 still hold.

The algorithms can be implemented in a scalar fashion \((s = 1)\), thus enabling access to the higher-precision, scalar-only datatypes, such as extended or quadruple precision, or in a pseudo-scalar way of, e.g., GPU programming models, where each thread executes the scalar code over a different data in the same layout as proposed here.

3. Column transformations, matrix scaling, and dot-products. A naïve formation of the Grammian pivot matrices by computing the three required dot-products without prescaling the columns is susceptible to overflow and underflow [15] and thus severely restricts the admissible exponent range of the elements of the itera-

\(^5\)With the AVX512F instruction subset only, the bitwise operations require reinterpreting casts to and from 64-bit integer vectors (no values are changed, converted, or otherwise acted upon); e.g., \(\text{and}(x, y) = \text{castsi512}(_\_\_\_\_m512_\_\_\_ps(_\_\_\_\_m512_castpd_si512(x), _\_\_\_\_m512_castpd_si512(y)))\).
Algorithm 2.2 z8jac2: a vectorized eigendecomposition of at most s double precision Hermitian matrices of order two with the Intel’s AVX-512 intrinsics.

Input: \( \tilde{a}_{11}, \tilde{a}_{22}, \tilde{R}_{\tilde{a}_{21}}, \tilde{A}_{\tilde{a}_{21}}, \cos \varphi, \cos \alpha \tan \varphi, \sin \alpha \tan \varphi, \lambda_1, \lambda_2, p \)

Output: \( \cos \varphi, \cos \alpha \tan \varphi, \sin \alpha \tan \varphi, \lambda_1, \lambda_2; p \) a permutation-indicating bitmask

// vectors with all lanes set to a compile-time constant
1. \(-0 = \text{set}1(-0.0); 0 = \text{setzero}(); 1 = \text{set}1(1.0); \omega = \text{set}1(\text{DBL}_\text{MAX}); \)
2. \(\sqrt{\omega} = \text{set}1(1.34078079299425956E+154); \quad // f(\sqrt{\omega}) \text{ from (2.6)} \)
3. \(\mu = \text{set}1(\text{DBL}_\text{TRUE}_\text{MIN}); \eta = \text{set}1(1020.0); // \eta = (\text{DBL}_\text{MAX}_\text{EXP} - 1) - 3 \)

// aligned loads of the ith input vectors
4. \(a_{11} = \text{load}(\tilde{a}_{11} + i); a_{22} = \text{load}(\tilde{a}_{22} + i); \quad // \text{all lanes have to } \ldots \)
5. \(\tilde{R}_{\tilde{a}_{21}} = \text{load}((\tilde{R}_{\tilde{a}_{21}} + i); \quad \tilde{A}_{\tilde{a}_{21}} = \text{load}(\tilde{A}_{\tilde{a}_{21}} + i); \quad // \ldots \ \text{hold } \text{finite } \text{values} \)

// the scaling exponents \(\zeta\)
6. \(\zeta_{11} = \text{sub}(\eta, \text{getexp}(a_{11})); \quad \zeta_{22} = \text{sub}(\eta, \text{getexp}(a_{22})); \quad // \text{getexp}(x) = \lfloor \log |x| \rfloor \)
7. \(\zeta_{21} = \text{sub}(\eta, \text{getexp}(\tilde{R}_{\tilde{a}_{21}})); \quad \zeta_{31} = \text{sub}(\eta, \text{getexp}(\tilde{A}_{\tilde{a}_{21}})); \quad // \ldots \ \lfloor \log 0 \rfloor = -\infty \)
8. \(\zeta = \text{min}(\omega, \text{min}(\zeta_{11}, \zeta_{22}, \min(\zeta_{21}, \zeta_{31}))); \quad // \text{finalize } \zeta \text{ from (2.17)} \)
9. \(-\zeta = \text{xor}(\zeta, -0); \quad // \text{xor}(x, -0) \text{ flips the sign bits in } x; \text{ optionally, store } -\zeta \)

// the scaling of \(A \rightarrow 2^n A\)
10. \(\tilde{R}_{\tilde{a}_{21}} = \text{scalef}(\tilde{R}_{\tilde{a}_{21}}, \zeta); \quad \tilde{A}_{\tilde{a}_{21}} = \text{scalef}(\tilde{A}_{\tilde{a}_{21}}, \zeta); \quad // a_{21} = 2^s a_{21} \)
11. \(a_{11} = \text{scalef}(a_{11}, \zeta); \quad a_{22} = \text{scalef}(a_{22}, \zeta); \quad // a_{ii} = 2^s a_{ii} \)

// the polar form of \(a_{21}\) using (2.11)
12. \(\|\tilde{R}_{\tilde{a}_{21}}\| = \text{andnot}(\tilde{a}_{21} \land \tilde{a}_{21}); \quad // \text{andnot}(\tilde{a}, x) = x \land \lnot x \ldots \)
13. \(\|\tilde{A}_{\tilde{a}_{21}}\| = \text{andnot}(\tilde{A}_{\tilde{a}_{21}}); \quad // \ldots \ \text{clears the sign bits} \)
14. \(\text{sgn}(\tilde{R}_{\tilde{a}_{21}}) = \text{and}(\tilde{R}_{\tilde{a}_{21}} - 0); \quad // \text{and}(x, -0) \text{ extracts the sign bits} \)
15. \(a_{21} = \text{hypot}(\|\tilde{R}_{\tilde{a}_{21}}\|, \|\tilde{A}_{\tilde{a}_{21}}\|); \quad // \text{inline Algorithm 2.1} \)
16. \(\cos \alpha = \text{min}(\text{div}(\|\tilde{R}_{\tilde{a}_{21}}\|, |a_{21}|), 1); \quad // \min \text{ replaces } 0/0 \rightarrow \text{NaN} \text{ with 1} \)
17. \(\cos \alpha = \text{or}(\cos \alpha, \text{sgn}(\tilde{R}_{\tilde{a}_{21}})); \quad // \text{or } \text{transfers the sign bits to positive values} \)
18. \(\sin \alpha = \text{div}(\tilde{A}_{\tilde{a}_{21}}, \text{max}|a_{21}|, \mu); \quad // \text{max } \text{replaces } 0 \text{ with } \mu \)

// \cos \varphi \text{ and } e^{\alpha \tan \varphi} \text{ (or } e^{\alpha \sin \varphi} \text{)}
19. \(o = \text{scalef}|a_{21}|, 1; a = \text{sub}(a_{11}, a_{22}); \quad // (2.6) \)
20. \(|a| = \text{andnot}(\neg a, 0); \quad \text{sgn}(a) = \text{and}(a, 0); \quad // (2.6) \)
21. \(\tan 2\varphi = \text{or}(\text{min}(\text{max}(\text{div}(|a|, 0), \sqrt{\omega}), \text{sgn}(a)); \quad // (2.6) \)
22. \(\sec^2 2\varphi = \text{fmad}(\text{tan} 2\varphi, \text{tan} 2\varphi, 1); \quad // \sec^2 2\varphi < \infty \)
23. \(\tan \varphi = \text{div}(\tan 2\varphi, \text{add}(1, \text{sqrt}(\sec^2 2\varphi)); \quad // (2.5) \)
24. \(\sec^2 \varphi = \text{fmad}(\tan \varphi, \tan \varphi, 1); \quad // (2.7) \)
25. \(\cos \varphi = \text{sqrt}(\sec^2 \varphi); \quad \cos \varphi = \text{div}(\text{tan} 1, \text{sec} \varphi); \quad // (2.7) \)
26. \(\cos \alpha \tan \varphi = \text{mul}(\cos \alpha, \tan \varphi); \quad \text{store}(\cos \varphi + i, \cos \varphi); \quad // \text{aligned } \ldots \)
27. \(\sin \alpha \tan \varphi = \text{mul}(\sin \alpha, \tan \varphi); \quad \text{store}(\cos \alpha \tan \varphi + i, \cos \alpha \tan \varphi); // \ldots \ \text{stores} \)

// \(e^{\alpha \sin \varphi} \text{, } \cos \alpha \sin \varphi = \text{div}(\cos \alpha \tan \varphi, \sec \varphi); \text{ } \sin \alpha \sin \varphi = \text{div}(\sin \alpha \tan \varphi, \sec \varphi); \quad // \text{the eigenvalues} \)
28. \(\lambda_{1} = \text{div}(\text{fmad}(\text{tan} \varphi, \text{fmad}(a_{22}, a_{22}, a_{22}), \sec^2 \varphi); \quad // (2.8) \)
29. \(\text{store}(\sin \alpha \tan \varphi + i, \sin \alpha \tan \varphi); // \text{below: } \text{fmsub}(x, y, z) \equiv \text{fmad}(x, y, -z) \)
30. \(\lambda_{2} = \text{div}(\text{fmad}(\text{tan} \varphi, \text{fmsub}(a_{11}, a_{22}, a_{22}), \sec^2 \varphi); \quad // (2.8) \)
31. \(\lambda_{1} = \text{scalef}(\lambda_{1}, -\zeta); \quad \text{store}(\lambda_{1} + i, \lambda_{1}); // \text{backscales and stores } \lambda_{1} \)
32. \(p = \text{mm512_cmlpl_pd_mask}(\lambda_{1}, \lambda_{2}) // \text{lane-wise check if } \lambda_{1} < \lambda_{2} \)
33. \(\lambda_{2} = \text{scalef}(\lambda_{2}, -\zeta); \quad \text{store}(\lambda_{2} + i, \lambda_{2}); // \text{backscales and stores } \lambda_{2} \)
34. \(p[1] = -\text{ctvmask}_u32(p); // \text{store } p \text{ to } p, x = 8 \text{ (16 for AVX512F)} \)

// alternatively, in lines 31 and 33, return \(\lambda_{1}\) and \(\lambda_{2}\), resp., instead of backscales
### Algorithm 2.3 \( \text{zbjac2} \): an OpenMP-parallel, AVX-512-vectorized eigendecomposition of a batch of \( \tilde{r} \) double precision Hermitian matrices of order two.

**Input:** \( \tilde{r}; \tilde{a}_{11}, \tilde{a}_{22}, \tilde{R}_{21}, \tilde{S}_{21}; p \) also, in the context of Algorithm 4.1 only.

**Output:** \( \cos \tilde{\varphi}, \cos \tilde{\alpha} \tan \tilde{\varphi}, \sin \tilde{\alpha} \tan \tilde{\varphi}; \tilde{\lambda}_1, \tilde{\lambda}_2; p \) // unsigned array of length \( \tilde{r}/s \)

```c
1: #pragma omp parallel for default(shared) // optional
2: for i = 0 to \( \tilde{r} - 1 \) step s do // i = \( \ell - 1 
3: 3: \text{if } p[i/s] = 0 \text{ then continue; } // \text{skip this vector on request of Algorithm 4.1}
4: 4: \text{zbjac2}(i, \tilde{a}_{11}, \tilde{a}_{22}, \tilde{R}_{21}, \tilde{S}_{21}, \cos \tilde{\alpha} \tan \tilde{\varphi}, \sin \tilde{\alpha} \tan \tilde{\varphi}, \cos \tilde{\varphi}, \tilde{\lambda}_1, \tilde{\lambda}_2, p[i, -\tilde{\xi}]);
5: 5: \text{// Algorithm 2.2 should be inlined above to avoid a function-call overhead}
6: 6: \text{end for } // \text{optional backscaling of } \tilde{\lambda}_1 \text{ and } \tilde{\lambda}_2 \text{ if skipped (} -\tilde{\xi} \text{ has to be returned)}
```

3.1. Effects of the Jacobi rotations on the elements’ magnitudes. Transforming a pair of columns as \( \{g_p, g_q\} U = \{g'_p, g'_q\} \), by any Jacobi rotation \( U \), cannot raise the largest magnitude of the elements from any row \( i \) of the pair by more than \( \sqrt{2} \) in exact arithmetic. For any \( \varphi, |\cos \varphi| + |\sin \varphi| \leq \sqrt{2} \), and for any \( \alpha, |\pm e^{+i\alpha}| = 1 \), and for any index pair \( (p, q) \), where \( p < q \) and \( g'_p \) and \( g'_q \) are not to be swapped,

\[
g'_{ip} = g_{ip} \cos \varphi + g_{iq} e^{i\alpha} \sin \varphi, \quad g'_{iq} = g_{iq} \cos \varphi - g_{ip} e^{-i\alpha} \sin \varphi, \quad \max\{|g'_{ip}|, |g'_{iq}|\} \leq \max\{|g_{ip}|, |g_{iq}|\} (|\cos \varphi| + |\sin \varphi|) \leq \sqrt{2} \max\{|g_{ip}|, |g_{iq}|\}.
\]

All quantities involving \( \alpha \) and \( \varphi \) are computed, and therefore may not be exact. Applying a computed Jacobi rotation, where \( e' = \text{fl}(\text{fl}(e^{i\alpha}) \text{fl}(\tan \varphi)) \), is in fact done as

\[
\begin{align*}
g'_{ip} &= (g_{ip} + g_{iq} e') \text{fl}(\cos \varphi), \\
g'_{iq} &= (g_{iq} - g_{ip} e') \text{fl}(\cos \varphi), \\
\text{fl}(g'_{ip}) &= \text{fl}(\text{fma}(g_{ip}, e', g_{ip}) \text{fl}(\cos \varphi)), \\
\text{fl}(g'_{iq}) &= \text{fl}(\text{fma}(g_{iq}, -e', g_{iq}) \text{fl}(\cos \varphi)),
\end{align*}
\]  

using the complex fused multiply-add from (2.10). However, the components of the result of either \text{fma} from (3.1) are larger in magnitude by the factor \( \approx 1/\text{fl}(\cos \varphi) \) than those of the final result. Overflow in those intermediate computations is avoided by rescaling the iteration matrix as in subsection 3.2. Similar holds for real transformations, with \( e^{+i\alpha} = \pm 1 \) and the real fused multiply-add in (3.1). Lemmas 3.1 and 3.2, proven in Appendix D of the supplementary material, bound, in the real and the complex case, respectively, relative growth of all, or the “important”, transformed element’s magnitudes in (3.1), caused by the rounding errors, by a modest multiple of \( \varepsilon \).

**Lemma 3.1.** Assume that all input floating-point values in (3.1) are real and finite, and neither overflow nor underflow occurs in any rounding of those computations. Let \( \text{fl}(g'_{ip}) = g_{ip}(1 + \varepsilon'_p) \) and \( \text{fl}(g'_{iq}) = g_{iq}(1 + \varepsilon'_q) \). Then,

\[
(1 - \varepsilon)^2 \leq 1 + \varepsilon'_p \leq (1 + \varepsilon)^2, \quad (1 - \varepsilon)^2 \leq 1 + \varepsilon'_q \leq (1 + \varepsilon)^2.
\]

With no other assumptions than computing with the standard \[21\] floating-point arithmetic, if \( \max\{|g_{ip}|, |g_{iq}|\} \leq \omega/2 \) then \( \max\{|\text{fl}(g'_{ip})|, |\text{fl}(g'_{iq})|\} \leq \omega \).

**Lemma 3.2.** Assume that the complex \text{fma} from (2.10) is used in (3.1), no input value has a non-finite component, and neither overflow nor underflow occurs in any rounding of those computations. With \( p \geq 23, \varepsilon = 3.000001 \varepsilon, \) and \( \varepsilon'' = 5.658656 \varepsilon, \)
1. if $|g'_{ip}| > 0$ and $|g'_{ip}|/\|\text{fl}(\cos \varphi)\| \geq |\Re g_{iq}|$, then $|\text{fl}(g'_{ip}) - g'_{ip}|/|g'_{ip}| \leq \varepsilon''$, else, if $|g_{iq}| \leq \omega/(1 + \sqrt{2})$, then $|\text{fl}(g'_{ip})| < \omega$;
2. if $|g'_{iq}| > 0$ and $|g'_{iq}|/\|\text{fl}(\cos \varphi)\| \geq |\Re g_{iqp}|$, then $|\text{fl}(g'_{iq}) - g'_{iq}|/|g'_{iq}| \leq \varepsilon''$, else, if $|g_{ip}| \leq \omega/(1 + \sqrt{2})$, then $|\text{fl}(g'_{iq})| < \omega$.

With no other assumptions than computing with the standard [21] floating-point arithmetic, if $\max\{|\text{fl}(g'_{ip})|, |\text{fl}(g'_{iq})|\} \leq \omega/4$ then $\max\{|\text{fl}(g'_{ip})|, |\text{fl}(g'_{iq})|\} \leq \omega$.

Proposition 3.3 follows directly from the last statements of Lemmas 3.1 and 3.2.

At the start of each step $k \geq 1$ it is assumed anew that $G_k = \text{fl}(G_k)$, i.e., the current floating-point representation of the iteration matrix is taken as exact.

**Proposition 3.3.** Assume $G_k = \text{fl}(G_k)$. If $\|G_k\|_{\max} \leq \omega/\zeta^\Re$, where $\zeta^\Re = 2$ for $G_k$ real and $\zeta^\Re = 4$ for $G_k$ complex, then $\|\text{fl}(G_{k+1})\|_{\max} \leq \omega$.

### 3.2. Periodic rescaling of the iteration matrix in the robust SVD.

The state-of-the-art construction of the diagonalizing Jacobi rotation for a pivot Grammian matrix, prescaled by the inverse of the Frobenius pivot column norms, from [15, Eq. (2.13) and Algorithm 2.5] and LAPACK, overcomes all range limitations except a possible overflow/underflow of a quotient of those norms. However, it requires a procedure for computing the column norms without undue overflow, like the ones from the reference BLAS routines (S/D)NRM2 [2, Algorithm 2] and (SC/DZ)NRM2, which are not straightforwardly vectorizable.

Therefore, there are several options for the Frobenius norm computation in the robust Jacobi SVD, as shown in Appendix F in the supplementary material. Three of them, with their advantages and drawbacks, are:

1. rely on the reference or the vendor-optimized BLAS routines, while ensuring that the column norms cannot overflow by an adequate rescaling of the iteration matrix, and bearing in mind that those routines are at best conditionally reproducible on any fixed platform (depending on the selected instruction subset, e.g.), or;
2. compute the norms as the square roots of dot-products in a higher-precision datatype with a wide-enough exponent range, e.g., in the hardware-supported Intel’s extended 80-bit (platform-dependent) datatype or in the standard [21] quadruple precision (128-bit) datatype for double precision inputs, with the latter choice being portably vectorizable with the SLEEF’s quad-precision math library, or;
3. extend a typical vectorized dot-product procedure such that it computes the norms with the intermediate and the output data represented as having the same precision (i.e., the significands’ width) as the input elements, but almost unbounded exponents, as proposed in [29, Appendix A] for the Jacobi-type SVD on GPUs (however, platform-independent in principle, and reproducible up to a choice of $\alpha$).

In Appendix F the third option above is described in the context of SIMD processing and all options are evaluated, with the conclusion that on Intel’s platforms the first one, but with the MKL’s routines instead of the reference ones, is the most performant and quite accurate. Also, for all options the following representation of the computed norms is proposed. Let $|x|_F = (e, f)$, where $e$ and $f$ are quantities of the input’s datatype $D$, such that $(e, f)$ represents the value $2^e f$. Let for a non-zero finite value hold $-\infty < e < \infty$ and $1 \leq f < 2$, while $0 = (-\infty, 1)$. All such values from $D$ can thus be represented exactly, while the results (but only those computed with wider exponents) that would overflow if rounded to $D$ are preserved as finite.

Since $\|G\|_F \leq \sqrt{mn}\|G\|_{\max}$, to ensure $\|G\|_F \leq \omega/\zeta^\Re$ it suffices to scale $G$ such that $\|G\|_{\max} \leq \omega/(\zeta^\Re\sqrt{mn})$. With $G$ complex, $\|G\|_{\max} \leq \sqrt{2}\max\{\|\Re G\|_{\max}, \|\Im G\|_{\max}\}$, so it suffices to have $\max\{\|\Re G\|_{\max}, \|\Im G\|_{\max}\} \leq \omega/(\zeta^\Re\sqrt{2mn})$. The Jacobi transformations are unitary, and the Frobenius norm is unitarily invariant, so $\|G_k\|_F = \|G\|_F$.
for all $k \geq 1$, if the rounding errors are ignored. No overflow can occur in (3.1) with $G$ scaled as described, due to Proposition 3.3 and $\|G_k\|_{\max} \leq \|G\|_F = \|G\|_F \leq \omega/\varsigma^F$.

Without rounding errors, the previous paragraph would define the required initial scaling of $G$ in terms of $\|G\|_{\max}$ (or $\|RG\|_{\max}$ and $\|SG\|_{\max}$), and no overflow checks should be needed, since $\|(G_k)_j\|_F \leq \|G\|_F$ for any column $j$. However, as discussed, some algorithms for the Frobenius norm might overflow even when the result should be finite, and the rounding within the transformations could further raise the magnitudes of the elements. In the absence of another theoretical bound for a particular Frobenius norm routine, assume\(^6\) that for any $k \geq 1$ and $1 \leq j \leq n$, $\|(G_k)_j\|_{\max} \leq \omega/(\varsigma^F m)$, and so $\|G_k\|_{\max} \leq \omega/(\varsigma^F m)$, implies $\text{fl}(\|G\|_F) \leq \omega$. Note that $m \geq \sqrt{mn}$ since $m \geq n$. At the start of each step $k$ it then suffices to have $\|G_k\|_{\max} \leq \omega/(\varsigma^F m)$ in the real, and $\max\{\|RG\|_{\max}, \|SG\|_{\max}\} \leq \omega/(\varsigma^C m \sqrt{2})$ in the complex case, to avoid any overflow in that step. Since $\varsigma^F$ is a largest of two, $\omega/\varsigma^F$ is an exactly representable quantity that, for its exponent, has the largest significand possible (all ones), while this property might not hold for the other upper bounds for the max-norms.

Let $2^{s[i]}(G_k)$ be a scaling of $G_k$, where the exponent $s[i] \in \mathbb{Z}$ is

\[
(3.2) \quad s[i] = \lfloor \log(\omega/(\varsigma^R m)) \rfloor - 1, \quad s[i] = \lfloor \log(\omega/(\varsigma^C m \sqrt{2})) \rfloor - 1,
\]

with $\tilde{M}_k^R = \|G_k\|_{\max}$ and $\tilde{M}_k^C = \max\{\|RG\|_{\max}, \|SG\|_{\max}\}$. Instead of comparing the significand of the upper bound (that should have then been rounded downwards) with that of $\tilde{M}_k^R$ and deciding whether to subtract unity from the scaling exponent if the former is smaller than the latter, the easiest but potentially suboptimal way to build the scaling exponents in (3.2) and (3.3) below are the unconditional subtractions of unity when the upper bound is not $\omega/\varsigma^F$. If a computation of the column norms cannot overflow, then, due to Proposition 3.3, a more relaxed scaling $2^{s[i]}(G_k)$, where

\[
(3.3) \quad s[i] = \lfloor \log(\omega/(\varsigma^R)) \rfloor - 1, \quad s[i] = \lfloor \log(\omega/(\varsigma^C \sqrt{2})) \rfloor - 1,
\]

is sufficient to protect the transformations from overflowing when forming $G_{k+1}$.

Observe that (3.3) protects from a destructive action, i.e., from overflowing while replacing a pivot column pair of the iteration matrix with its transformed counterpart. No recovery is possible from such an event without either keeping a copy of the original column pair or checking the magnitudes of the transformed elements before storing them, both of which slow down the execution. In contrast, overflow of a computed norm is non-destructive, and can be recovered (and protected) from by downscaling $G_k$ according to (3.2) and computing the norms of the scaled columns.

It is expensive to rescale $G_k$ at the start of every step. The following rescaling heuristic is thus proposed, that delays scaling unless a destructive operation is possible:

1. Let $G_0 = G$ and $s_0 = s[i]$, before the iterative part of the Jacobi SVD. Then, let $G_1 = 2^{s_0}G_0$ be the initial iteration matrix. If all elements of $G$ are small enough by magnitude, this can imply upscaling ($s_0 > 0$) and as many subnormal values as safely possible, if they exist in $G$, become normal\(^7\) in $G_1$. Otherwise, $s_0 \leq 0$. Also, let $M_0^F = 2^{s_0}M_0^F$, where $M_0^F$ has been found by a method described below.

\(6\)This assumption can be turned into a user-provided parameter to the Jacobi SVD routine that holds a theoretically or empirically established upper bound on the array elements’ magnitudes for non-overflowing execution of the chosen Frobenius norm procedure, with (3.2) adjusted accordingly.

\(7\)This upscaling, i.e., raising of the magnitudes tries to keep the elements of the rotated column pairs from falling into the subnormal range if a huge but not total cancellation in (3.1) occurs.
2. At the start of each step \( k \geq 1 \), compute \( \tilde{s}^y_{(k)} \) from (3.3) and \( \tilde{s}^x_{[k]} \) from (3.2), using 
\( \tilde{M}^y_k \) determined at the end of previous step. If \( s^y_{(k)} < 0 \), \( G_k \) has to be downscaled 
to \( G'_k \) and \( \tilde{M}^y_k \) updated. For that, take the lower exponent \( s^y'_{(k)} = s^y_{[k]} \), since it will 
protect the subsequent computation of the column norms as well. Otherwise, let \( s^y_{(k)} = 0 \) and \( G'_k = G_k \). Define \( s_k = s_{k-1} + s^y'_{(k)} \) as the effective scaling exponent of the 
initial \( G \), i.e., \( 2^{-s_k}G'_k \) is what the iteration matrix would be without any scaling.

3. If any column norm of \( G''_k \) overflows, rescale \( G'_k \) to \( G''_k \) using \( s''_k = s''_{[k]} - s^y'_{(k)} \), let 
\( s_k = s_k + s''_k \), recompute the norms, and update \( \tilde{M}^y_k \). Else, \( G''_k = G'_k \) and \( s''_k = 0 \).

4. While applying the Jacobi rotations to transform \( G''_k \) to \( G_{k+1} \), compute \( \tilde{M}^y_{k+1} \).

This can be done efficiently by reusing portions of a transformed pivot column 
pair already present in the CPU registers, but at the expense of implementing the 
custom rotation kernels instead of relying on the BLAS routines xROT and xSCAL. 
If \( \tilde{M}^y_0 = \infty \), i.e., if \( G \) contains a non-finite value, the Jacobi SVD algorithm fails. It is 
assumed that \( G \) is otherwise of full column rank, so \( \tilde{M}^y_0 > 0 \) (else, the routine stops).

When the Jacobi process has numerically converged after \( K \) steps, for some \( K \), 
the scaled singular values \( \Sigma' \) of \( G \) have to be scaled back by \( 2^{-s} \), \( s = s_K \). If \( \sigma^j \) 
were represented as an ordinary floating-point value, such a backscaling could have 
caused the result’s overflow or an undesired underflow [30]. However, \( \sigma^j \) is computed as the 
Frobenius norm of the \( j \)th column of the final iteration matrix and is thus 
represented as \( \sigma^j = (e^j, f^j) \), making any overflow or underflow of the backscaled 
\( \sigma^j = 2^{-s} \sigma^j = (e^j - s, f^j) \) impossible, unless \( \sigma^j \) is converted to a floating-point value.

Rescaling of each column of \( G_k \) is trivially vectorizable by the \texttt{scalef} intrinsic, 
and the columns can be processed concurrently. The max-norm of a real matrix is 
computed as a parallel max-reduction of the columns’ max-norms. For a column \( x \) let \( x \) 
be a vector of zeros, and load consecutive vector-sized chunks \( y \) of \( x \) in a loop. For each 
loaded chunk, update the partial maximums in \( x \) as \( x = \max(x, \min(\text{abs}(y), \infty)) \), 
converting any encountered NaN in \( y \) into \( \infty \). After the loop, let \( \|x\|_{\text{max}} = \text{reduce\_max}(x) \).

The max-norm of a complex matrix \( G_k \) is approximated, as described, by the 
maximum of two real max-norms, \( \|G_k\|_{\text{max}} = \sqrt{2} \max\{\|\Re G_k\|_{\text{max}}, \|\Im G_k\|_{\text{max}}\} \). The 
Frobenius norm of a complex column \( x \) is obtained as \( \|x\|_F = \text{hypot}(\|\Re x\|_F, \|\Im x\|_F) \).

3.3. The scaled dot-products. Let \( [g_p, g_q] \), \( p < q \), be a pivot column pair from 
\( G''_k \), and \( 0 < ||g_j||_F = (e_j, f_j) \), \( g_j = g_j/||g_j||_F \), for \( j \in \{p, q\} \). The scaled complex dot-
product \( \tilde{g}^*_p \tilde{g}_q \) is computed as in Algorithm 3.1 (the real scaled dot-product being its 
obvious simplification), with a single division operation. In the loop of Algorithm 3.1, 
\( g_j \) is prescaled to \( g_j/2^{e_j} \), with its Frobenius norm \( \approx f_j \). The components of the resulting 
\( \tilde{z} = \text{fl}((g_q/2^{e_q}) \cdot (g_p/2^{e_p}))) \) and \( z = \text{fl}(\tilde{g}^*_p \tilde{g}_q) = \text{fl}(\tilde{z} / (f_q : f_p)) \) thus cannot overflow.

3.3.1. The convergence criterion. Following [15] and the LAPACK’s \texttt{xGESVJ} 
routines, a pivot column pair \( (p, q) \) of the iteration matrix is not transformed (but the 
columns and their norms might be swapped) if it is numerically orthogonal, i.e., if 
\begin{equation}
\text{fl}(\tilde{g}^*_p \tilde{g}_q) < \text{fl}(\epsilon \sqrt{m}) = \nu.
\end{equation}

The Jacobi process stops successfully if no transformations in a sweep over all 
pivot pairs have been performed (and thus the convergence has been detected), or 
unsuccessfully if the convergence has not been detected in the prescribed number of 
sweeps \( S \). In LAPACK, \( S = 30 \), but this might be insufficient, as shown in section 5.

Assume that, for a chosen sequence of \( S \) pivot index pairs \( (p_1, q_1), \cdots, (p_S, q_S) \), the
Algorithm 3.1 vdpstd: a vectorized complex scaled dot-product routine.

Input: \( g_q = (Rg_q, \Im m g_q), 0 < \| g_q \|_F = (e_q, f_q); g_p = (Rg_p, \Im m g_p), 0 < \| g_p \|_F = (e_p, f_p). \)

Output: \( z = (R\hat{z}, \Im m \hat{z}) = \hat{z}_g^* g_p + g_q^* q_p/(\| g_q \|_F \| g_p \|_F) \).

1. \( \hat{R}z = \text{setzero}(); \; \hat{z} = \text{setzero}(); \; -e_j = \text{set1}(-e_j); \quad j \in \{p, q\} \)

2. for \( i = 0 \) to \( m - 1 \) step \( s \) do

3. \( \Re g_{ij} = \text{load}(Rg_{ij} + 1); \; \Im m g_{ij} = \text{load}(\Im m g_{ij} + 1); \quad j \in \{p, q\} \) here and below

4. \( \Re g_{ij} = \text{scale}(\Re g_{ij} - e_j); \; \Im m g_{ij} = \text{scale}(\Im m g_{ij} - e_j); \quad j \in \{p, q\} \)

5. \( \Re z = \text{fmad}(\Re g_{ij}, q_p, \Re z); \; \Im m z = \text{fmad}(\Im m g_{ij}, q_p, \Im m z); \)

6. \( \Re z = \text{fmad}(\Re g_{ij}, q_p, \Re z); \; \Im m z = \text{fmad}(\Im m g_{ij}, q_p, \Im m z); \)

7. end for

8. \( \Re z = \text{reduce_add}(\Re z); \; \Im m z = \text{reduce_add}(\Im m z); \quad \) above: single \( 2 \times 64 \)-bit division

9. \( \text{mm_store}pd(\Re z, \text{mm_div}pd(\text{mm_set}pd(\Re z, \Re z), \text{mm_set}pd(f_p, f_p))); \)

10. return \( z; \quad \) above: single \( 2 \times 64 \)-bit division

respective scaled dot-products have already been computed and packed into vectors

\( \Re a_{21}^\ell = \text{fl}(\Re (g_{q}, g_{pl}))_\ell, \; \Im m a_{21}^\ell = \text{fl}(\Im m (g_{q}, g_{pl}))_\ell, \quad 1 \leq \ell \leq s. \)

Algorithm 3.2 checks the convergence criterion over all vectors’ lanes, encodes the result as a bitmask, and counts how many transformations should be performed.

Algorithm 3.2 Vectorized checking of the convergence criterion.

Input: The vectors \( \Re a_{21}^\ell \) and \( \Im m a_{21}^\ell; \) \( v. \)

Output: The bitmask \( c \); \( c_\ell = 1 \iff (p_\ell, q_\ell) \) should be transformed; \( \sum c_\ell = S_\ell \)

1. \( v = \text{set1}(v); \quad |a_{21}^\ell| = \text{hypot}(\Re a_{21}^\ell, \Im m a_{21}^\ell); \; \) above: single \( 2 \times 64 \)-bit division

2. \( c = \text{mm}512_c\_\text{cmp}le\_pd\_\text{mask}(v, |a_{21}^\ell|); \quad \) above: single \( 2 \times 64 \)-bit division

3. return \( \_\text{mm}512_c\_\text{popcnt}u32(\_\text{cvtmaskX}u32(c)); \quad X = 8 \) (16 for AVX512F)

3.4. Formation of the scaled Grammians. Let \( [g_p, g_q], p < q \), be a pivot column pair from \( G^p_j \), and \( 0 < \| g_j \|_F = (e_j, f_j) \) for \( j \in \{p, q\} \). The scaled Grammian

\[ A_{[pq]}' = \frac{1}{\| g_q \|_F \| g_p \|_F} [g_p g_q]^* [g_p g_q] = \begin{bmatrix} a_{11}' & a_{12}' \\ a_{21}' & a_{22}' \end{bmatrix} \]

can be expressed in the terms of \( \| g_j \|_F = 2^e_j f_j \) as

\[ a_{11}' = \frac{\| g_q \|_F}{\| g_p \|_F} = 2^{e_q} f_p f_q, \quad a_{22}' = \frac{\| g_q \|_F}{\| g_p \|_F} = 2^{e_q} f_q f_p, \quad a_{21}' = \frac{\| g_q \|_F}{\| g_p \|_F} = 2^{e_q} f_p f_q. \]

Algorithm 3.1 computes \( a_{21}' \) without overflow, and neither \( a_{11}' = (e_p - e_q, f_p/f_q) \) nor \( a_{22}' = (e_q - e_p, f_q/f_p) \) can overflow in this “non-normalized” representation (but can as floating-point values). To call the batched EVD routine with scaled Grammians as inputs, they have to be scaled further to the representable range by \( 2^{s_{\[pq\]}}, s_{\[pq\]} \leq 0 \), i.e., to \( A'_{\[pq\]} = 2^{s_{\[pq\]}} A_{\[pq\]}' \). Let \( e' = \lfloor \log(f_p/f_q) \rfloor \) and \( e'' = \lfloor \log(f_q/f_p) \rfloor \). Then, define

\[ e_{p/q} = e_p - e_q + e', \quad e_{q/p} = e_q - e_p + e'', \quad f_{p/q} = 2^{-e'} f_p f_q, \quad f_{q/p} = 2^{-e''} f_q f_p. \]
The normalized representations of the diagonal elements of $A'_{pq}$ are $a'_{11} = (e_{p/q}, f_{p/q})$ and $a'_{22} = (e_{q/p}, f_{q/p})$, where the fractional parts lie in $[1, 2)$ and the exponents are still finite. Let $\eta = \text{DBL\_MAX\_EXP} - 1$ be the largest exponent of a finite floating-point value, and $e_{\geq} = \max\{v_{p/q}, v_{q/p}\}$. Then, $s_{pq} = \min(\eta - e_{\geq}, 0)$. With a shorthand

$$F(x) = \text{getmant}(x, \_\_\_MM\_MANT\_NORM\_1\_2, \_\_\_MM\_MANT\_SIGN\_zero)$$

for extraction of the fractional parts of $x$ in $[1, 2)$, this formation procedure is vectorized in Algorithm 3.3 for complex Grammians, while the real ones do not require $3a''_{pq}$.

Algorithm 3.3 Vectorized formation of non-overflowing scaled Grammians.

**Input:** The vectors $\Re a'_{21}, \Im a'_{21}; e_1, f_1; e_2, f_2$ of the scaled dot-products and the first and the second column norms, resp., of the chosen pivot pairs $(p_1, q_1), \ldots, (p_{s}, q_{s})$.

**Output:** $a''_{11}, a''_{22}, \Re a''_{21}, \Im a''_{21}$, where the $\ell$th lane of $a''_{ij}$ is $(A''_{ij})_{\ell j}$.

1. $f_{12} = \text{div}(f_1, f_2); e_{12} = \text{sub}(e_1, e_2); f_{21} = \text{div}(f_2, f_1); e_{21} = \text{sub}(e_2, e_1)$; // (3.5)
2. $e_{12} = \text{add}(e_{12}, \text{getexp}(f_{12})); f_{12} = F(f_{12}); (e_{p/q}, f_{p/q})$ from (3.6) with (3.7)
3. $e_{21} = \text{add}(e_{21}, \text{getexp}(f_{21})); f_{21} = F(f_{21}); (e_{q/p}, f_{q/p})$ from (3.6) with (3.7)
4. $s_{A} = \min(\text{sub}(\text{set1}(	ext{DBL\_MAX\_EXP}) - 1, \text{max}(e_{12}, e_{21})), \text{setzero}()); // s_{pq}$
5. $e_{21} = \text{add}(e_{21}, s_{A}); e_{21} = \text{add}(e_{21}, s_{A}); // each lane $\leq \eta$
6. $a''_{12} = \text{scalef}(f_{12}, e_{12}); a''_{22} = \text{scalef}(f_{21}, e_{21}); // scale the diagonal
7. $\Re a''_{21} = \text{scalef}(\Re a''_{21}, s_{A}); \Im a''_{21} = \text{scalef}(\Im a''_{21}, s_{A}); // scale the off-diagonal

### 3.5. The Jacobi transformations

In the $k$th step, the postmultiplication of a pivot column pair $\begin{bmatrix} x_p^{(k)} \\ x_q^{(k)} \end{bmatrix}$ by the Jacobi rotation $U = \begin{bmatrix} e_{pq}^{(k)} \\ e_{pq}^{(k)} \end{bmatrix}$ is performed as in Algorithm 3.4, with $l = m$ for $X = G''_k$ and $l = n$ for $X = V_k$, where $V_1 = V_0 = I$ if the right singular vectors are to be accumulated in the final $V$; otherwise, $V_0$ can be set to any matrix that is to be multiplied by them. In the former case, the max-norm approximation is not needed, but in the latter one it might be required to prevent the overflow of the elements of $V$ due to rounding errors, similarly as it is done with $G$.

The real Jacobi transformation, $\text{djrrot}$, is a direct simplification of Algorithm 3.4.

In Algorithm 3.4, $P$ is a permutation matrix that is not identity if the transformed columns have to be swapped. In the (unconditionally reproducible) implementation there are further optimizations, like skipping the multiplications by $c = 1$ and applying two real transformations, on $(\Re x_p, \Re x_q)$ and $(\Im x_p, \Im x_q)$, if the rotation is real ($S = 0$).

No conditionals are present in the loop, instead, each branch has a specialized version of the loop. The return value is $\max\{|\Re x_p', |\Im x_p', |\Re x_q', |\Im x_q'\}$.

#### 3.5.1. The Gram–Schmidt orthogonalization

In [15, Definition 2.7] the conditions and the formula for the Gram–Schmidt orthogonalization of $g_q$ against $g_p$ are given, that replaces their Jacobi transformation when $\|g_q\|_F \ll \|g_p\|_F$ and tan $\varphi$ underflows. Using (3.5), the real-case orthogonalization [15, Eq. (2.20)] is defined as

$$\|g_p\|_F > 2^m v\|g_q\|_F \implies g'_q = g_q \wedge g'_q = \left( g_q \|g_q\|_F - a'_{21} g_p \|g_p\|_F \right) \|g_q\|_F ,$$

what, by representing $\|g_q\|_F = 2^m v f_j$ and after moving $f_q$ within the parenthesis, gives

$$g'_q = \left( \frac{g_q}{2^{m+1}} - \psi \frac{g_p}{2^{m+1}} \right) 2^m v , \quad \psi = a'_{21}(f_q / f_p) , \quad 1/2 < f_q / f_p < 2 .$$

---

8slightly simplified and with a different instruction order than in the prototype implementation.
Output: \([x_p \ x_q]\) of length \(n\) in the split form, by the Jacobi transformation \(\Phi = U(\alpha_{pq}, \varphi_{pq})P\).

\(\Phi =\) a parallel one has to be chosen. It is then natural to select the maximal number of repeating the same pivot sequence in each sweep) strategy, with

\[\text{subset of pivot pairs more than once in a sweep. An example is a quasi-cyclic strategy}\]

\[\text{any number of OpenMP threads can be requested,}\]

\[\text{Any number of OpenMP threads can be requested,}\]

Algorithm 3.4 \(z\) rot: a vectorized postmultiplication of a pair of complex columns \([x_p \ x_q]\) of length \(n\), by the method \(\Phi = U(\alpha_{pq}, \varphi_{pq})P\).

\(\text{Input:}\) \((\Re x_p, \Im x_p)\), \((\Re x_q, \Im x_q)\); \(c = \cos \varphi_{pq}; C = \cos \alpha_{pq} \tan \varphi_{pq}, S = \sin \alpha_{pq} \tan \varphi_{pq}\).

\(\text{Output:}\) \([x'_p \ x'_q]\) \(\Phi\) and, optionally, an approximation of its max-norm.

1. \(C = \text{set1}(C); \ S = \text{set1}(S); \ c = \text{set1}(c); \quad // -C = \text{set1}(-C)\) or ...
2. \(-0 = \text{set1}(-0.0) ; \ M = \text{setzero}(); \quad // \ldots \ \text{fmmadd}(\cdot, \cdot, \cdot)\) in *
3. for \(i = 0\) to \(l - 1\) step \(s\) do // optimal in the sense of (2.18)
4. \(\Re x_{ij} = \text{load}(\Re x_j + i) ; \ \Im x_{ij} = \text{load}(\Im x_j + i) ; \quad // j \in \{p, q\}\)
5. \(\Re x'_{ip} = \text{mul}(\text{fmmadd}(\Re x_{iq}, C, \text{fmmadd}(\Im x_{iq}, S, \Re x_{ip})), c) ; \quad // \text{fma}(x_q, e', x_p)\)
6. \(\Im x'_{ip} = \text{mul}(\text{fmmadd}(\Re x_{iq}, S, \text{fmmadd}(\Im x_{iq}, C, \Im x_{ip})), c) ; \quad // e' = C + iS\)
7. \(\text{store}(\Re x_j + i, \Re x'_{ip}) ; \ \text{store}(\Im x_j + i, \Im x'_{ip}) ; \quad \// j = p\) if \(P = I\), else \(j = q\)
8. \(M = \text{max}(\{M, \text{max}(\text{andnot}(\text{andnot}(0, \Re x'_{ip}), \text{andnot}(0, \Im x'_{ip}))\)));\)
9. \(\Re x'_{iq} = \text{mul}(\text{fmmadd}(\Re x_{ip}, -C, \text{fmmadd}(\Im x_{ip}, S, \Re x_{iq})), c) ; \quad // \text{fma}(x_p, -e', x_q)\)
10. \(\Im x'_{iq} = \text{mul}(\text{fmmadd}(\Re x_{ip}, S, \text{fmmadd}(\Im x_{ip}, -C, \Im x_{iq})), c) ; \quad // -e' = -C + iS\)
11. \(\text{store}(\Re x_j + i, \Re x'_{iq}) ; \ \text{store}(\Im x_j + i, \Im x'_{iq}) ; \quad \// j = q\) if \(P = I\), else \(j = p\)
12. \(M = \text{max}(\{M, \text{max}(\text{andnot}(\text{andnot}(0, \Re x'_{iq}), \text{andnot}(0, \Im x'_{iq}))\)));\)
13. end for // after each iteration: \(M = \text{max}(\{M, |\Re x'_{ip}|, |\Im x'_{ip}|, |\Re x'_{iq}|, |\Im x'_{iq}|\})\) // maximum of the lanes of \(M\)

If the roles of \(g_p\) and \(g_q\) are reversed in (3.8), i.e., if \(\|g_p\|_F > 2^n v \|g_q\|_F\), the resulting \(g'_q\) and \(g'_p = g_q\) should be swapped to keep the non-increasing order of their norms.

\(\text{Algorithm H.1 in the supplementary material vectorizes (3.8). The complex variant}\)

\(\text{has not yet been implemented, and both variants remain to be tested.}\)

4. A parallel Jacobi-type SVD method. In this section the previously developed building blocks are put together to form a robust, OpenMP-parallel Jacobi-type SVD method. It is applicable to any \(m \times n\) input matrix \(G\) of full column rank with finite elements, where \(\hat{m}\) satisfies (2.18) and \(\hat{n} \mod 2s = 0\). If the dimensions of \(G\) do not satisfy these constraints, \(G\) is assumed to be bordered beforehand, as explained in [31], e.g. The workspace required is \(\hat{n}/s\) integers and \(4n\) or \(5n\) reals for the real or the complex variant, respectively. Any number of OpenMP threads can be requested, but at most \(\hat{n}\) will be used at any time. Only double precision versions of the method have been implemented, but the single precision ones should be conceptually identical.

4.1 Parallel Jacobi strategies. Even a sequential but vectorized method processes \(s > 1\) pivot column pairs in each step. No sequential pivot strategy (such as de Rijk’s [10], \(\text{dr}\), in LAPACK), that selects a single pivot pair at a time, suffices, and a parallel one has to be chosen. It is then natural to select the maximal number of \(\hat{n}/2\) pivot pairs each time, where all pivot indices are different. Since the number of all index pairs \((p, q)\) where \(p < q\) is \(\hat{n}(\hat{n} - 1)/2\), the chosen strategy is expected to require at least \(K = \hat{n} - 1\) steps in a sweep. It may require more (e.g., \(K = \hat{n}\)), and transform a subset of pivot pairs more than once in a sweep. An example is a quasi-cyclic strategy called the modified modulus [31] (henceforth, \text{MM}), applicable for \(\hat{n}\) even. A cyclic (i.e., repeating the same pivot sequence in each sweep) strategy, with \(\hat{n} - 1\) steps, could be, e.g., the Mantharam–Eberlein [27] one, or its generalization \(\text{ME}\) beyond \(\hat{n}\) being a power of two, from [29]. Available in theory for all even \(\hat{n}\), \(\text{ME}\) is restricted in practice to \(\hat{n} = 2^o\), \(l \geq 1\), \(o \leq 21\) odd, with a noticeably faster convergence than \(\text{MM}\) [29, 32].

The method is executed on a shared-memory system, so the cost of “ comunications...
tion” could be visible only if the data spans more than one NUMA domain; otherwise, any communication topology underlying a strategy can be disregarded when looking for a suitable one. More important is to assess if a strategy is convergent (provably, as \( \texttt{MM} \), or at least in practice, as \( \texttt{ME} \)) and the cost of its implementation (a lookup table of at most \( \tilde{n}^2 \) integers encoding the pivot pairs in each step of a sweep for \( \texttt{MM} \) and \( \texttt{ME} \) is set up before the execution for a given \( \tilde{n} \) in \( O(\tilde{n}^2) \) and \( O(\tilde{n}^2 \log \tilde{n}) \) time, respectively).

A dynamic ordering \([5, 6]\) would be a viable alternative to cyclic parallel strategies, but it is expensive for pointwise (i.e., non-blocked) one-sided methods (for a two-sided, pointwise Kogbetliantz-type SVD method with a dynamic ordering, see \([33]\)).

Among other advantages of the sequential Jacobi-type methods, the quasi-cubic convergence speedup of Mascarenhas \([28]\) remains elusive with a parallel strategy. A pointwise one-sided method is the most ungrateful Jacobi-type SVD for parallelization, with no performance benefits of blocking but with all the issues such a constrained choice of parallel strategies brings, as the slow convergence and a probably excessive amount of slightly non-orthogonal transformations in subsection 5.3 show.

4.2. Data representation. Complex arrays are kept in the split form. Splitting the input matrix \( G \) and merging the output matrices \( U \) (occupying the space of \( G \)) and \( V \) happen before and after the method is invoked, respectively, and take less than 1\% of the method’s run-time. The resulting singular values are kept as two properly aligned arrays, \( e \) and \( f \), such that \( \sigma_j = (e_j, f_j) \). The integer work arrays are \( p \) and \( c \), each with \( \tilde{n}/(2s) \) elements, while the real workspace \( w \) is divided into several properly aligned subarrays that are denoted by a tilde over their names in Algorithm 4.1.

4.2.1. Column norms and the left singular vectors. The Frobenius norms of the columns of the iteration matrix are held in \( (e, f) \). If two columns of \( G''_k \) (and the same ones of \( V_k \)) are swapped, so are their norms. If a column is transformed, its norm will be recomputed (not updated, as in \([15]\)) at the beginning of the \((k + 1)\)th step.

If the method converges, the iteration matrix, holding \( U\Sigma' \), has to be normalized to \( U \). For all \( j \) in parallel, each component of every element of the \( j \)th column of the iteration matrix is scaled by \( 2^{-e_j} \) and divided by \( f_j \). Finally, \( e_j = e_j - s \).

4.3. The method. Algorithm 4.1 shows a simplified implementation of the complex double precision method. The real variant, \( \texttt{dvjsvd} \), is derived straightforwardly.

If the assumption from subsection 3.2 on a safe upper bound of the magnitudes of the columns’ elements for the Frobenius norm computation is adequate, line 11 in Algorithm 4.1 cannot cause an infinite loop, but an inadequate assumption can. In the testing from section 5, after the initial scaling of \( G_0 \rightarrow G_1 \), no rescaling of the iteration matrix was ever triggered, except of \( U\Sigma' \rightarrow U \), so it should be a rare event.

Even an inadequate assumption can be incrementally improved. If a norm overflow is detected in line 11 more than once in succession, the assumed upper bound can be divided by two each time, until the iteration matrix is downscaled enough to prevent overflow and break this \texttt{goto}-loop. Appendix C gives the lower bounds on the assumption that would eventually be reached, when the column norms could be computed by ordinary dot-products as \( \sqrt{x^T x} \). If this happens, the flawed norm-computing routine can be replaced by a wrapper around the dot-product by a function pointer swap, without stopping the execution. This safeguard has not yet been implemented.

All loops of Algorithm 4.1 are parallel and, at least in the first few sweeps over a general matrix, have a balanced workload across all threads. Near the end of the execution, in the last sweeps, the number of pivot pairs that have to be processed should diminish, depending on the asymptotic convergence rate of the pivot strategy.
Algorithm 4.1 zvjsvd: a vectorized, OpenMP-parallel Jacobi-type SVD method.

Input: \((\mathbb{R}G, \mathbb{Z}G); \mathbf{S}, \mathbf{k}; \mathbf{w}, \mathbf{p}, c;\) a strategy lookup table \(J((k-1) \mod \mathbf{k}, \ell) \rightarrow (p^k_{\ell}, q^k_{\ell})\).

Output: \((\mathbb{R}U', \mathbb{Z}U'), (\mathbb{R}V, \mathbb{Z}V'), (\mathbf{e}, \mathbf{f});\) the actual number of sweeps performed \(C \leq \mathbf{S}\).

1. Determine \(\tilde{M}^C_0, s^C_0\) from \(G_0\) and scale \(G_0, \tilde{M}^C_0\) to \(G_1, \tilde{M}^C_1\) as in subsection 3.2.
2. Set \(V_1 = I\) in parallel over its columns and let \(k = 1\). // \(k\) is the sweep counter.
3. for \(C = 0 \to \mathbf{S} - 1\) do
   // sweep loop
   4. \(T = 0;\) // \(T\) counts the number of transformations in the whole sweep
   5. for \(k' = 0 \to k - 1\) do // step loop, \(k' = (k - 1) \mod \mathbf{k}\)
      6. Find \(s^C_{(k)}, s^C_{(k)}\) from \(\tilde{M}^C, (3.2)\) and (3.3); downscale \(G_k \to G_k'\) if \(s^C_{(k)} < 0\).
      7. for \(\ell = 1 \to \tilde{n}/2\) do // parallel; + reduce \(o\), the overflow count (initially 0)
          \(\|g_{\ell}\|_F = \text{hypot}(\|Rg_{\ell}\|_F, \|\Im g_{\ell}\|_F);\) // \(j \in \{p, q\}\); use the MKL’s DNRM2
          8. if \(\|g_{\ell}\|_F = \infty\) then \(o = o + 1\) else \((e_{\ell}, f_{\ell}) = \|g_{\ell}\|_F;\) // convert to \((e, f)\)
      9. end for // \((p_t, q_t) = J(k', \ell);\) see also subsection 4.2.1.
   10. if \(o > 0\) then // downscale \(G_k' \to G_k''\) as in subsection 3.2 and goto line 7;
   11. for \(\ell = 1 \to \tilde{n}/2\) do // parallel; compute the scaled dot-products
          \(z = g_{\ell}p_{\ell}g_{\ell}p_{\ell} = \text{zdpscl}(Rg_{\ell, p}, Rg_{\ell, q}, e_p, e_q, f_p, f_q);\) // Algorithm 3.1
          \(\Re a_{21} \equiv \|Rz_{21}\|_{\ell} = \|\Re z_{21}\|_{\ell} = \|z_{21}\|;\) // pack the results contiguously in \(w\)
          \(e_{\ell} = e_{\ell};\) \(f_{\ell} = e_{\ell};\) \(f_{\ell} = f_{\ell};\) \(f_{\ell} = f_{\ell};\) // pack the norms
      12. end for // \((p_t, q_t) = J(k', \ell);\) see also subsection 4.2.4.
   13. for \(i = 0 \to \tilde{n}/2\) step \(s\) do // parallel; + reduce \(t\) (initially 0)
          \(\Re a_{21} = \text{load}(\Re a_{21} + 1);\) \(\Im a_{21} = \text{load}(\Im a_{21} + 1);\) // load \(a_{21}\) from \(w\)
          \(P_t = \text{Algorithm 3.2}(\ell \to c_j, \nu);\) // check the convergence criterion; \(j = i/s\)
          if \(p_t > 0\) then \(t = t + p_t\) else continue; // skip the rest if \(p_t = 0\)
          \(f_1 = \text{load}(f_1 + 1); f_2 = \text{load}(f_2 + 1); e_1 = \text{load}(e_1 + 1); e_2 = \text{load}(e_2 + 1);\)
          \(a_{12} = \text{Algorithm 3.3}(\Re a_{21}, \Im a_{21}, e_1, e_1, e_2, e_2);\) // \((A''_{p, q})_{\ell}\)
          store(a_{11} + 1, a_{11} + 1); store(a_{22} + 1, a_{22} + 1); // store the scaled 
          \(\text{store}(\Re a_{21} + 1, \Im a_{21} + 1);\) \(\ldots\) Grammians to \(w\)
      14. end for // \(t\) counts the number of transformations in the current step
   15. // Algorithm 2.3 computes the EVDs of the scaled Grammians in parallel:
   16. zbac2(\(\Re a_{21}, \Re a_{21}, \Im a_{21}; \cos \phi, \cos \alpha \tan \phi, \sin \alpha \tan \phi, \lambda_1, \lambda_2; p);\) // \((\Phi_t)_{\ell}\)
   17. for \(\ell = 1 \to \tilde{n}/2\) do // parallel; + max-reduce \(M\) (initially 0)
          if \(|c_{j}| = 0\) then // \(j = (\ell - 1)/s;\) bit index \(l = (((\ell - 1) \mod s) + 1)
          if \(p_{[j]} = 1\) then swap \(g_{\ell}, g_{\ell}, v_{\ell}, v_{\ell}, (p_{\ell}, f_{\ell});\) \(e_{\ell}, f_{\ell}_{\ell};\)
          \(\text{else transformation needed; postmultiply the column pairs by } \Phi_{\ell};\)
          \(\text{else if } p_{[j]} = 1 \text{ then } P_t = I;\) // \((\Phi_t = U_t P_t)\) implicitly
          \(\text{Algorithm 3.4 applied to } (g_{\ell, p}, g_{\ell, q}, \tilde{n}\text{ rows, and to } (v_{\ell, p}, v_{\ell, q}), \tilde{n}\text{ rows;)}\)
          \(M' = \text{zjrot}(Rg_{p, q}, Rg_{q, q}, Rg_{q, q}, \cos \phi, \cos \alpha \tan \phi, \sin \alpha \tan \phi, P_t);\)
          \(\text{zjrot}(Rv_{p, q}, Rv_{q, q}, Rv_{q, q}, \cos \phi, \cos \alpha \tan \phi, \sin \alpha \tan \phi, P_t);\)
          \(M = \max\{(M, M');\} / \mathbf{0 < M' < \infty} \text{ should hold}\)
      18. end if // \(M\) unchanged if not transforming
   19. end for // \((p_t, q_t) = J(k', \ell);\) see also subsection 3.5
   20. \(T = T + t;\) \(\tilde{M}^{C+1} = \max(\tilde{M}^C, M);\) // might be an overestimate, but safe
   21. end for // \(k = k + 1\)
   22. if \(T = 0\) then break; // convergence if no transformations in a sweep
   23. end for // \(C = C + 1\)
   24. if \(C < \mathbf{S}\) then normalize \(U \Sigma' \to U\) and scale \(\Sigma' \to \Sigma\) as in subsection 4.2.1;
   25. return \(C;\) // \(C = \mathbf{S} \iff \text{no convergence}\)

// For referencing, let lines 7–10 = ⭖, 12–14 = ⭖, 15–25 = ⭖, 26 = ⭖, and 27–38 = ⭖.
4.3.1. Reproducibility. Apart from the Frobenius norm computation and Algorithm 3.1, the results of which are reproducible in the same environment, all other parts of the method are unconditionally reproducible. The method’s results do not depend on the requested number of threads, as long as the external routines (only DNNRM2 from MKL for now) are sequential, but do on the choice of parallel strategy.

5. Numerical testing. The testing was performed on a 64-core Intel Xeon Phi 7210 CPU, running at 1.30 GHz with TurboBoost turned off, 96 GiB of RAM and 16 GiB of MCDRAM serving as the last-level cache only. Under 64-bit CentOS Linux 7.9.2009, the Intel oneAPI C, C++, and Fortran compilers, version 2021.2.0, and the sequential MKL library were used, with the ILP64 ABI and the Conditional Numerical Reproducibility mode set to MKL_CBWR_AVX512_MIC.

The batched EVD Algorithm 2.3 (henceforth, z), Algorithm B.2 (d), and their single precision complex (c) and real (s) counterparts were tested also in isolation, on (huge, for more reliable timing) batches of Hermitian/symmetric matrices of order two, comparing them to the inlinable, manually translated C versions of the reference LAPACK routines xLAEV2, for \( x \in \{Z, D, C, S\} \), respectively, since the MKL’s and the reference implementations were slower to call, with no observed difference in accuracy.

The SVD method in Algorithm 4.1 and its real variant were compared to the ZGESVJ and DGESVJ routines, respectively, with JOBA = 'G', JOBU = 'U', and JOBV = 'V'.

The OpenMP environment was set up for all tests as OMP_PROC_BIND = SPREAD, OMP_PLACES = CORES, and OMP_DYNAMIC = FALSE. For the EVD testing 32 threads were bound to the topologically distant cores, such that each core could exploit the whole L2 cache, while for the SVD testing OMP_NUM_THREADS \( \in \{8, 16, 32, 64\} \).

All error testing was done in quadruple precision datatypes, __float128 in C and REAL(KIND=REAL128) in Fortran, including the final scaling of the singular values SVA(j)*WORK(1) from xGESVJ and \( \sigma_j = 2^{63} f_j \) from the proposed SVD method.

5.1. Matrices under test. For the EVDs, 256 batches in single and 256 batches in double precision, each with \( 2^{28} \) Hermitian and \( 2^{28} \) symmetric matrices of order two, were generated using all from random \( \lambda_1, \lambda_2, \tan \varphi, \) and \( \cos \alpha \), where the random bits were provided by the RDRAND CPU facility. For the eigenvalues \( \lambda_j, \) a 32-bit or 64-bit quantity was reinterpreted as a single or a double precision value, respectively, and accepted if \( |\lambda_j| \leq \omega/2^4 \). Random 64-bit signed integers were converted to quadruple precision, scaled by \( 2^{-63} \) to the \([-1, 1]\) range, and assigned to \( \tan \varphi \) and \( \cos \alpha \). Then, \( |\sin \alpha| = \sqrt{1 - \cos^2 \alpha} \) in quadruple precision, and the sign of \( \tan \varphi \) was absorbed into \( e^{i\alpha} \) (making \( e^{i\alpha} = \pm 1 \) in the real case). The three required matrix elements from the lower triangle were computed in quadruple precision and rounded to single or double precision without overflow. Five real values were generated in total for \( \lambda_1, \lambda_2, \tan \varphi, \) and \( \cos \alpha \): four of them for one complex and two real matrix elements, and the last one for the (2,1)-element of a real symmetric matrix, implicitly generated from the same eigenvalues and \( \tan \varphi \), but as if \( \cos \alpha = 1 \) initially. These values were stored to binary files and later read from them into memory, one batch at a time, in the layout described in subsection 2.4.1. The eigenvalues were similarly preserved for comparison.

To make \(|e^{i\alpha}|\) as close to unity as practicable, \( \cos \alpha \) was in fact rounded from quadruple to double precision (with 52 bits of significand) and converted back, before computing \( \sin^2 \alpha = 1 - \cos^2 \alpha \) with 112 bits of significand. Thus, \( \sin^2 \alpha \) was exact.

For the SVD testing, the datasets \( \Xi_{\xi}^D \) and \( \Xi_{\xi}^F \), parametrized by \( \xi = \xi_1 = -23 \) and \( \xi = \xi_2 = -52 \), respectively, were generated\(^9\), each one with complex (\( F = C \)) and real

---

\(^9\)See https://github.com/venovako/JACSD/tree/master/tgensvd for the implementation.
with both components of similar, close to unit magnitudes and close enough magnitudes. If they had been close to normal, this issue would have been avoided. A further explanation is left for Appendix G.1 in the supplementary material, along with more EVD testing results.

For example, \( \text{diag}(\Sigma|\xi = -3, n = 4, I_n|) = [1/8 \ 1/4 \ 1/2 \ 1]^T \). The permuted singular values \( \Sigma|\xi, n, P_n = P_n \Sigma|\xi, n, I_n| P_n^{-1} \) can be in ascending, descending, or any random order. In the former two cases, the smallest singular values are tightly clustered.

Let an input matrix \( G|\xi, n, P_n = U_n \Sigma|\xi, n, P_n V_n^* \). For \( \Xi_{\xi}^n \) a random \( P_n \), same for both \( \mathbb{F} \), was taken for each \( n \). For \( \Xi_{\xi}^n \), three input matrices were generated for each \( n \) and \( \mathbb{F} \), with ascending, descending, and a random \( P_n \) order of \( \Sigma \), same for both \( \mathbb{F} \). The random unitary matrices \( U_n \) and \( V_n^* \) were implicitly generated by two applications, from the left and from the right, of the LAPACK’s testing routine xLAROR, \( x \in \{ \mathbb{D}, \mathbb{Z} \} \), converted to work in quadruple precision. First, \( U_n \Sigma \), and then \( G = (U_n \Sigma) V_n^* \) were obtained. The resulting \( G \) was rounded to double precision and stored, as well as \( \Sigma \).

5.2. The batched EVD results. Figure 5.1 shows the run-time ratio, batch by batch, of calling the LAPACK-like routine for each matrix in a batch and invoking the vectorized EVD for eight \( (\mathbb{d} \text{ and } \mathbb{z}) \) or 16 \( (\mathbb{s} \text{ and } \mathbb{c}) \) matrices at once. In lines 26 and 27 of Algorithm 2.2, and in line 16 of Algorithm B.1, \( \text{fl}(e^{i\alpha} \tan \varphi) \) was further divided by \( \text{fl}(\sec \varphi) \) to get \( \text{fl}(e^{i\alpha} \sin \varphi) \), as in (2.7). This was also done in \( \mathbb{s} \) and \( \mathbb{c} \). The complex LAPACK-like routines were adapted for taking the \( (2, 1) \) matrix element, the complex conjugate of the \( (1, 2) \) element \( \mathbb{B} \), as input, and \( \text{SN1} = \text{fl}(e^{i\alpha} \sin \varphi) \) was kept in the split form, to eliminate any otherwise unavoidable pre-/post-processing overhead.

A parallel OpenMP for loop split the work within a batch evenly among the threads. Each thread thus processed \( 2^{28}/32 = 2^{23} \) matrices per batch. Every xLAEV2 invocation, had it not been inlined, would have involved several function calls, so these results are a lower bound on run-time of any semantically unchanged library routine. After the “warmup” with the first batch in a run, affecting the vectorized EVD more than the LAPACK-based one, the results are consistent and satisfactory. The single precision versions of the batched EVD are more performant than the double precision ones due to twice the number of single versus double precision lanes per widest vector.

The EVD’s relative residual\(^{10} \) is \( \rho_\mathbb{F}^\alpha = \| U \Lambda U^* - A \|_F / \| A \|_F \), where \( x \in \{ \mathbb{d}, \mathbb{z}, \mathbb{s}, \mathbb{c} \} \) for the batched EVD in the corresponding precision and datatypes, and \( x \in \{ \mathbb{d}, \mathbb{z}, \mathbb{s}, \mathbb{c} \} \) for the respective LAPACK-based one. For a batch \( \tau \), let \( \rho_\mathbb{F}^\alpha = \max_i \rho_\mathbb{F}^\alpha_{x, i}, 1 \leq i \leq 2^{28} \). Then, in Figure 5.2 the ratios \( \rho_\mathbb{F}^\alpha / \rho_\mathbb{F}^2 \) and \( \rho_\mathbb{F}^2 / \rho_\mathbb{F}^2 \) show that, on average, real batched EVDs are a bit more accurate than the LAPACK-based ones, but \( \rho_\mathbb{F}^2 / \rho_\mathbb{F}^2 \) and \( \rho_\mathbb{F}^2 / \rho_\mathbb{F}^2 \) as indicated in Remark 2.3, demonstrate that a catastrophic loss of accuracy of the eigenvectors (which in this case are no longer of the unit norm) is possible when the components of \( \mathbb{B} \) are of small subnormal and close enough magnitudes. If they had been close to normal, this issue would have been avoided. A further explanation is left for Appendix G.1 in the supplementary material, along with more EVD testing results.

Observe that the upsampling from (2.17) could have preserved accuracy of the complex LAPACK routines in many problematic instances by preventing \( \mathbb{B} / |\mathbb{B}| \) to be computed with both components of similar, close to unit magnitudes. However, certain patho-

\(^{10} \) Here and for all other error measures that involve division, assume \( 0/0 = 0 \).
logical cases are unavoidable even with Algorithm 2.2. Consider the following matrix

\[
A = \begin{bmatrix}
\omega/8 & \bar{\mu} + i\bar{\mu} \\
\bar{\mu} \pm i\bar{\mu} & \omega/8
\end{bmatrix}, \quad \text{fl}(|a_{12}|) = \bar{\mu} = \text{fl}(|a_{21}|) \implies \text{fl}(e^{i\alpha}) = 1 \pm i.
\]

Then, from (2.17), \(\zeta = 0\), while \(\text{fl}(\tan \varphi) = 1\), \(\text{fl}(\cos \varphi) = \text{fl}(1/\sqrt{2})\), and therefore

\[
\text{fl}(U) = \text{fl}(\cos \varphi) \begin{bmatrix} 1 & -1 \pm i \\
1 \pm i & 1 \end{bmatrix}, \quad \det(\text{fl}(U)) \approx \frac{3}{\sqrt{2}}, \quad \|\text{fl}(u_1)\|_F = \|\text{fl}(u_2)\|_F = \frac{3}{\sqrt{2}}.
\]
the post-scaling subnormal values were zeroed out (e.g., by setting the Flush To Zero (FTZ) CPU flag, but not DAZ, before line 10 in Algorithm 2.2), then the issues with $A$ and fully subnormal matrices would vanish, but this “fix” could turn a nonsingular ill-conditioned matrix into an exactly singular one (it depends on the context if this is an issue). Thus, if the input data range is too wide for the scaling to make all matrix elements normal, it is safest to compute the EVD in a datatype with wider exponents.

5.3. The SVD results. Only a subset of the complex variant’s results is shown here, with the rest presented in Appendix G.2 in the supplementary material.

5.3.1. Dataset $\Xi^C$. Figure 5.3 shows the speedup of $\text{zvjsvd}$ versus $\text{ZGESVJ}$. A higher one might have been expected, given that both the thread-based and the SIMD (vector) parallelism were employed in the former, but this can be explained. Foremost, $\text{zvjsvd}$ with $\text{MM}$ took $\approx 10$ sweeps more on bigger matrices (see Figure G.4) than $\text{ZGESVJ}$ with $\text{dR}$ (ME, where applicable, lowered the difference by 1–2 sweeps). This demonstrates the need for better parallel strategies for pointwise one-sided methods.

Second, as Figure 5.4 shows, $\text{zvjsvd}$ with 64 threads spent most of its run-time on transforming the columns ($\star$) and computing the Frobenius norms ($\ddagger$). For $\star$, the max-norm approximation of the transformed columns of the iteration matrix was additionally computed. Algorithm 3.4, as well as the scaled dot-products ($\clubsuit$), might be optimized further by a careful cache management that is beyond the scope of a prototype implementation. For $\ddagger$, the Frobenius norms of the transformed columns of the iteration matrix were recomputed, while $\text{ZGESVJ}$ updated them, with a periodic recomputation [15]. The prescaling ($\heartsuit$) and the EVD ($\diamondsuit$) of the Grammians jointly took less than $2\%$ of the run-time on bigger inputs and may be considered optimized.

![Fig. 5.3. Run-time ratios of $\text{ZGESVJ}$ and $\text{zvjsvd}$ with $\text{MM}$ on $\Xi^C$.](image)

Define the relative error measures $r_G = \|U\Sigma V^* - G\|_F/\|G\|_F$, $r_U = \|U^*U - I\|_F$, $r_V = \|V^*V - I\|_F$, and $r_\Sigma = \max_j |\sigma_j' - \sigma_j|/|\sigma_j|$ for $\text{ZGESVJ}$, where $\sigma_j'$ and $\sigma_j$ are the $j$th computed and exact singular value, respectively, and let $r'_G$, $r'_U$, $r'_V$, and $r'_\Sigma$ be the same measures for $\text{zvjsvd}$. Figure 5.5 suggests that the singular values are relatively accurate and the left singular vectors are orthogonal with $\text{zvjsvd}$ almost as with $\text{ZGESVJ}$, while the relative SVD residuals are somewhat worse, probably due to a mild ($\max_j r'_V < 5 \cdot 10^{-20}$) loss of orthogonality of the right singular vectors. These extra errors in the proposed method might be caused by transforming $G$ and $V$ too
many times (due to more sweeps) compared to ZGESVJ, by the “improper” rotations that near the end of the process lose orthogonality due to $\cos \varphi = 1$ and $|\tan \varphi| > 0$.

5.3.2. Dataset $\bigotimes_2^2$. For $\bigotimes_2^2$, ME was used with zvjsvd. The MKL routines required at least $\text{NSWEEP} = 30$ sweeps with a majority inputs (see Figure G.5). To circumvent that, z/dgesvj.f source files were taken from the LAPACK repository and modified to z/dnssvj.f, with NSWEEP as a function argument, instead of being a hard-coded parameter, what could in general benefit the users of the xGESVJ routines.

Both ZGESVJ and zvjsvd behaved as expected, with $r_G < 8 \cdot 10^{-2}$, $r'_G < 8 \cdot 10^{-1}$; $r_G < 5 \cdot 10^{-14}$, $r'_G < 3 \cdot 10^{-12}$; $\max\{r_U, r'_U\} < 2 \cdot 10^{-22}$; while $r_V < 9 \cdot 10^{-24}$ and $r'_V < 9 \cdot 10^{-20}$ indicate the same problem as with $\bigotimes_1^2$. The results were similar for $\bigotimes_2^2$.

6. Conclusions and future work. The strongest contribution of this paper is the vectorized algorithm for the batched EVD of Hermitian matrices of order two. It requires no branching but only the basic bitwise and arithmetic operations, with fma,
and max and min that filter out a single NaN argument. It is not applicable if trapping on floating-point exceptions is enabled, and should be tuned for non-default rounding modes, but it is faster and often more accurate in every aspect than the matching sequence of xLAEV2 calls. Also, the computed scaled eigenvalues cannot overflow.

The proposed SVD method should be several times faster than xGESVJ on modern CPUs and scale (sublinearly) with the number of cores. A fully tuned implementation, as well as ever-increasing vector lengths on various platforms, should provide significantly better speedups, while the stored column norms could be updated as in [15]. The scaling principles from subsections 3.2, 3.4 and 4.3 cost little performance-wise, do not depend on parallelism or the pivot strategy, and thus could be incorporated into xGESVJ, as well as the real variants of Algorithms 3.1, 3.4 and H.1 into DGESVJ.

Acknowledgments. The author is thankful to Zlatko Drmač for mentioning a long time ago that there might be room for low-level optimizations in his Jacobi-type SVD routines in LAPACK, and to Sanja, Saša, and Dean Singer, without whose material support this research would never have been finished.

Appendix A. Derivation and accuracy of the formulas from subsection 2.1. Let $A$, $U$, and $Λ$ be as in (2.1). In Appendix A.1 the formulas from subsection 2.1 are derived. In Appendix A.2 the relative errors induced while computing some of those formulas in finite precision are given as a part of the proof of Proposition 2.4.

A.1. Derivation of the formulas from subsection 2.1. Equating the corresponding elements on both sides of $U^*AU = Λ$ and assuming $\cos \varphi \neq 0$ it follows
\begin{align}
\lambda_{11}/\cos^2 \varphi &= a_{11} + a_{22} \tan^2 \varphi + 2R (a_{21} e^{-i\alpha}) \tan \varphi, \\
\lambda_{22}/\cos^2 \varphi &= a_{11} \tan^2 \varphi + a_{22} - 2R (a_{21} e^{-i\alpha}) \tan \varphi,
\end{align}
for the diagonal elements of $Λ$, and
\begin{align}
\frac{\lambda_{21}}{\cos^2 \varphi} e^{-i\alpha} &= a_{21} e^{-i\alpha} + (a_{22} - a_{11}) \tan \varphi - \bar{a}_{21} e^{i\alpha} \tan^2 \varphi = 0,
\end{align}
for one of the remaining off-diagonal zeros, as well as
\begin{align}
\frac{\lambda_{12}}{\cos^2 \varphi} e^{i\alpha} &= \bar{a}_{21} e^{i\alpha} + (a_{22} - a_{11}) \tan \varphi - a_{21} e^{-i\alpha} \tan^2 \varphi = 0
\end{align}
for the other.

From (A.2) the first two, and from (A.3) the last two equations in
\begin{align}
\bar{a}_{21} e^{i\alpha} \tan^2 \varphi - a_{21} e^{-i\alpha} &= (a_{22} - a_{11}) \tan \varphi = a_{21} e^{-i\alpha} \tan^2 \varphi - \bar{a}_{21} e^{i\alpha}
\end{align}
are obtained. Ignoring the middle equation and regrouping the terms, it follows
\begin{align}
a_{21} e^{-i\alpha} (1 + \tan^2 \varphi) = \bar{a}_{21} e^{i\alpha} (1 + \tan^2 \varphi),
\end{align}
and, by canceling $1 + \tan^2 \varphi < \infty$ on both sides, $a_{21} e^{-i\alpha} = \bar{a}_{21} e^{i\alpha}$, i.e., $z = \bar{z}$, what is possible if and only if $z$ is real. Therefore,
\begin{align}
α &= \arg(a_{21}), \quad e^{iα} = a_{21}/|a_{21}|.
\end{align}
If $a_{21}$ is real, $e^{iα} = \text{sign}(a_{21}) = \pm 1$. 

\[24\]
Using (A.4) and $|a_{21}| = |a_{21}|$, (A.1)–(A.3) become
\[
\begin{align*}
\lambda_{11} / \cos^2 \varphi &= a_{11} + a_{22} \tan^2 \varphi + 2|a_{21}| \tan \varphi, \\
\lambda_{22} / \cos^2 \varphi &= a_{11} \tan^2 \varphi + a_{22} - 2|a_{21}| \tan \varphi, \\
\frac{\lambda_{21}}{\cos^2 \varphi} e^{-i\alpha} &= |a_{21}| + (a_{22} - a_{11}) \tan \varphi - |a_{21}| \tan^2 \varphi = 0, \\
\frac{\lambda_{12}}{\cos^2 \varphi} e^{i\alpha} &= |a_{21}| + (a_{22} - a_{11}) \tan \varphi - |a_{21}| \tan^2 \varphi = 0.
\end{align*}
\]

The last two equations above are identical, so from either one it follows
\[
(A.5) \quad |a_{21}|(1 - \tan^2 \varphi) = (a_{11} - a_{22}) \tan \varphi.
\]

If $a_{21} = 0$, and since $a_{11}$ and $a_{22}$ are arbitrary, from (A.5) follows $\tan \varphi = 0$, i.e., $U$ is the identity matrix. Else, if $a_{11} = a_{22}$, then $\tan \varphi = \pm 1$ satisfies (A.5). In all other cases,
\[
\frac{2|a_{21}|}{a_{11} - a_{22}} = \frac{2 \tan \varphi}{1 - \tan^2 \varphi} = \tan(2\varphi),
\]
and, since $|\tan(2\varphi)| < \infty$, i.e., $|\varphi| < \pi/4$,
\[
\tan \varphi = \frac{\tan(2\varphi)}{1 + \sqrt{1 + \tan^2(2\varphi)}},
\]

**A.1.1. Proof of Lemma 2.1.** See the main paper for its statement.

**Proof.** Observe that $m$ and $M$ in (2.12) are exact, being normal or not, so any underflow in their formation is harmless. A possible underflow of $\text{fl}(q)$ is harmless as well. If $\text{fl}(q)$ is subnormal or zero, $\text{fma}(\text{fl}(q), \text{fl}(q), 1) = \text{fma}(q, q, 1)$, as if $\varepsilon_1 = 0$ below.

Let $\text{fl}(q) = q(1 + \varepsilon_1)$, where $|\varepsilon_1| \leq \varepsilon$. Then,
\[
\text{fl}(\text{fl}(q)^2 + 1) = \text{fma}(\text{fl}(q), \text{fl}(q), 1) = (q^2(1 + \varepsilon_1)^2 + 1)(1 + \varepsilon_3) = r(1 + \varepsilon_3),
\]
where $|\varepsilon_3| \leq \varepsilon$. Expressing $r$ as the wanted, exact quantity times a relative error factor, $r = (q^2 + 1)(1 + \varepsilon_2)$, and solving this equation for $\varepsilon_2$, it follows
\[
\varepsilon_2 = \frac{q^2}{q^2 + 1} \varepsilon_1(2 + \varepsilon_1).
\]
Since $0 \leq q \leq 1$, the fraction above ranges from $0$ (for $q = 0$) to $1/2$ (for $q = 1$), inclusive, irrespectively of $\varepsilon_1$. Therefore, $\varepsilon_2$, as a function of $\varepsilon_1$ when $q = 1$, attains the extremal values for $\varepsilon_1 = \mp \varepsilon$,
\[
-\frac{\varepsilon(2 - \varepsilon)}{2} \leq \frac{1}{2} \varepsilon_1(2 + \varepsilon_1) \leq \frac{\varepsilon(2 + \varepsilon)}{2}.
\]
However, if $q = 1$ (equivalently, if $m = M$), then $\varepsilon_1 = 0$ since the division is exact, and the two inequalities above are in fact strict, i.e., neither equality is possible.

Now, $\text{fl}(\sqrt{\text{fma}(\text{fl}(q), \text{fl}(q), 1)}) = \sqrt{(q^2 + 1)(1 + \varepsilon_2)(1 + \varepsilon_3)(1 + \varepsilon_4)}$, where $|\varepsilon_4| \leq \varepsilon$.

The final multiplication by $M$ gives, with $|\varepsilon_5| \leq \varepsilon$,
\[
\text{fl}(\text{hypot}(x, y)) = M \sqrt{q^2 + 1} \sqrt{1 + \varepsilon_2} \sqrt{1 + \varepsilon_3} \sqrt{1 + \varepsilon_4} \sqrt{1 + \varepsilon_5} = \delta_2 \text{hypot}(x, y),
\]
where $\delta_2 = \sqrt{1 + \varepsilon_2} \sqrt{1 + \varepsilon_3} \sqrt{1 + \varepsilon_4} \sqrt{1 + \varepsilon_5}$ is minimized for $\varepsilon_3 = \varepsilon_4 = \varepsilon_5 = -\varepsilon$ and $\varepsilon_2 = -\varepsilon(2 - \varepsilon)/2$, and maximized for $\varepsilon_3 = \varepsilon_4 = \varepsilon_5 = \varepsilon$ and $\varepsilon_2 = \varepsilon(2 + \varepsilon)/2$.

**Remark A.1.** When considering relative floating-point accuracy of a computation, any underflow by itself, in isolation, is harmless if the result is exact, since the relative error is zero. But it is too cumbersome to state this obvious exception.
A.2. Proof of Proposition 2.4. See the main paper for its statement.

Proof. If \( a_{21} \) is real, \( \text{fl}(|a_{21}|) = \delta^c_\alpha |a_{21}|, \delta^w_\alpha = 1 \), else \( \text{fl}(|a_{21}|) = \delta^c_\alpha |a_{21}| \), where \( \delta^c_\alpha = \delta_2 \) (see Lemma 2.1). Thus, in the complex case, from (2.4) it follows

\[
\text{fl}(\cos \alpha) = \cos \alpha \frac{1 + \varepsilon_1}{\delta_2} = \delta'_\alpha \cos \alpha, \quad \text{fl}(\sin \alpha) = \sin \alpha \frac{1 + \varepsilon_2}{\delta_2} = \delta''_\alpha \sin \alpha,
\]

where \( \max\{|\varepsilon_1|,|\varepsilon_2|\} \leq \varepsilon \). The error factors \( \delta'_\alpha \) and \( \delta''_\alpha \) are maximized for \( \varepsilon_1 = \varepsilon_2 = \varepsilon \) and \( \delta_2 = \delta^+_2 \), and minimized for \( \varepsilon_1 = \varepsilon_2 = -\varepsilon \) and \( \delta_2 = \delta^-_2 \). Let \( \delta^+_2 = (1 + \varepsilon)/\delta^+_2 \) and \( \delta^-_2 = (1 - \varepsilon)/\delta^-_2 \). Since \( \min\{|x|,|y|\} \leq \sqrt{x^2 \cos^2 \alpha + y^2 \sin^2 \alpha} \leq \max\{|x|,|y|\}, \)

\[
1 - 4.000000 \varepsilon \lessgtr \delta^-_2 < |\text{fl}(e^{\pm i\alpha})| < \delta^+_2 \lessgtr 1 + 4.000001 \varepsilon.
\]

The approximated multiples of \( \varepsilon \) come from evaluating \((1 - \delta^-_2)/\varepsilon \) and \((\delta^+_2 - 1)/\varepsilon \), as explained for (A.12) and (A.13) below.

From (2.6), \( \text{fl}(|a|) = |a_{11} - a_{22}|(1 + \varepsilon_3) \), where \( |\varepsilon_3| \leq \varepsilon \), and \( \text{fl}(2|a_{21}|) = 2 \text{fl}(|a_{21}|) \). First, assume that \( \text{fl}(2|a_{21}|/|a|) \leq \text{fl}(|\sqrt{\omega}|) \). Then,

\[
\text{fl}(\tan 2\varphi) = \text{fl}(2|a_{21}|/|a|) \text{sign } a = \delta^x_{2\varphi} (2|a_{21}|/|a|) \text{sign } a = \delta^x_{2\varphi} \tan 2\varphi,
\]

where \( \delta^c_{2\varphi} = \delta_2/(1 + \varepsilon_3) \) and \( \delta^w_{2\varphi} = 1/(1 + \varepsilon_3) \). Minimizing and maximizing these fractions, similarly as above, it follows

\[
\delta^c_{2\varphi} = \frac{\delta^-_2}{1 + \varepsilon} < \delta^w_{2\varphi} < \frac{\delta^+_2}{1 - \varepsilon} = \delta^c_{2\varphi},
\]

\[
\delta^w_{2\varphi} = \frac{1}{1 + \varepsilon} \leq \delta^w_{2\varphi} \leq \frac{1}{1 - \varepsilon} = \delta^w_{2\varphi}.
\]

Else, let \( \omega > \text{fl}(2|a_{21}|/|a|) > \text{fl}(|\sqrt{\omega}|) \) and assume that \(|\text{fl}(\tan 2\varphi)| \) was not bounded above by \( \text{fl}(|\sqrt{\omega}|) \). Then \(|\text{fl}(\tan \varphi)| \) would have been the exact unity if the square root in (2.5) was computed as finite, e.g., using \( \text{hypot}(1, \text{fl}(\tan 2\varphi)) \) and assuming

\[
|x| \geq \text{fl}(|\sqrt{\omega}|) \implies \text{hypot}(1, x) = |x| \land 1 + |x| = |x|
\]

for a representable \( x \). Therefore, regardless of the relative error in \( \text{fl}(\tan 2\varphi) \), the relative error in \( \text{fl}(\tan \varphi) \) could have only \emph{decreased} in magnitude from the one obtained with \( \text{fl}(\tan 2\varphi) = \text{fl}(|\sqrt{\omega}|) \), when also \(|\text{fl}(\tan \varphi)| = 1 \), since \( \tan \varphi \to \pm 1 \) monotonically for \( \tan 2\varphi \to \pm \infty \). The Jacobi rotation is built from \( \text{fl}(\tan \varphi) \) (and the functions of \( \alpha \) in the complex case), while \( \text{fl}(\tan 2\varphi) \) is just an intermediate result and thus the relative error in it is not relevant as long as the one in \( \text{fl}(\tan \varphi) \) is kept in check.

By substituting \( \text{fl}(\tan 2\varphi) \) for \( \tan 2\varphi \) in (2.5) and using the fused multiply-add for the argument of the square root, with \( \max\{|\varepsilon_4|,|\varepsilon_5|,|\varepsilon_6|,|\varepsilon_7|\} \leq \varepsilon \) it follows that

\[
\text{fl}(\tan \varphi) = \frac{\delta^x_{2\varphi} \tan 2\varphi(1 + \varepsilon_7)}{(1 + \sqrt{((\delta^x_{2\varphi} \tan 2\varphi)^2 + 1)(1 + \varepsilon_4)(1 + \varepsilon_5)(1 + \varepsilon_6)})} = \delta^x_{\varphi} \tan \varphi,
\]

where \( \varepsilon_4, \varepsilon_5, \varepsilon_6, \) and \( \varepsilon_7 \) stand for the relative rounding errors of the fma, the square root, the addition of one, and the division, respectively, and \( \delta^x_{\varphi} \) remains to be bounded.

Let \( y = \tan 2\varphi \). From (A.8), by solving the equation

\[
(\delta^x_{2\varphi})^2 y^2 + 1 = (y^2 + 1)(1 + x)
\]
for $x$, it is possible to express the relative error present in the intermediate result of the fma before its rounding, as a function of $\varphi$ (due to $y$) and $\varepsilon$ (due to $(\delta_{2\varphi}^\varepsilon)^2)$,

$$
(A.9) \quad x = \frac{y^2}{y^2+1} \varepsilon_{2\varphi}, \quad \varepsilon_{2\varphi} = (\delta_{2\varphi}^\varepsilon)^2 - 1, \quad 0 \leq |x| < |\varepsilon_{2\varphi}|,
$$

since $0 \leq y^2/(y^2 + 1) < 1$ for all $y$. From (A.7) $\varepsilon_{2\varphi}^\varepsilon$, and thus $x$, can be bounded as

$$
(A.10) \quad (\delta_{2\varphi}^\varepsilon)^2 - 1 = \varepsilon_{2\varphi}^- \leq \varepsilon_{2\varphi} \leq \varepsilon_{2\varphi}^+ = (\delta_{2\varphi}^\varepsilon)^2 - 1.
$$

By rewriting the fma operation as above, (A.8) can be expressed as

$$
\text{fl}(\tan \varphi) = \frac{\tan 2\varphi}{1 + \sqrt{\tan^2 2\varphi + 1}} \frac{\delta_{2\varphi}^\varepsilon (1 + \varepsilon_7)}{1 + \varepsilon_6},
$$

or, letting $r = \sqrt{\tan^2 2\varphi + 1}, b = \sqrt{1 + x} \sqrt{1 + \varepsilon_4 (1 + \varepsilon_5)}$, and $b' = \delta_{2\varphi}^\varepsilon (1 + \varepsilon_7)/(1 + \varepsilon_6)$,

$$
\text{fl}(\tan \varphi) = \frac{\tan 2\varphi}{1 + br} b' = \frac{\tan 2\varphi}{1 + r(1 + d)} b' = \tan \varphi \frac{b'}{1 + d} = \varepsilon_{2\varphi}^\varepsilon \tan \varphi, \quad r \geq 1.
$$

Similarly as before, the equation $1 + br = (1 + r)(1 + d)$ has to be solved for $d$ to factor out the relative error (i.e., $d$) from the remaining exact value (i.e., $1 + r$). Then,

$$
\text{fl}(\tan \varphi) = \frac{\tan 2\varphi}{1 + br} b' = \frac{\tan 2\varphi}{1 + r(1 + d)} b' = \tan \varphi \frac{b'}{1 + d} = \varepsilon_{2\varphi}^\varepsilon \tan \varphi, \quad r \geq 1.
$$

Maximizing $|\delta_{2\varphi}^\varepsilon| = \delta_{2\varphi}^\varepsilon$ is equivalent to maximizing $|\delta_{2\varphi}^\varepsilon| = \delta_{2\varphi}^\varepsilon$ as $\delta_{2\varphi}^\varepsilon$ from (A.7) and minimizing $d$, while setting $\varepsilon_6 = -\varepsilon$ and $\varepsilon_7 = \varepsilon$. Minimizing $d$ is equivalent to letting $r \to \infty$ in (A.11) and minimizing $b$, what amounts to setting $\varepsilon_4 = \varepsilon_5 = -\varepsilon$ and minimizing $x$ by letting $y^2 \to \infty$ in (A.9) and taking $\varepsilon_{2\varphi}^\varepsilon$ from (A.10) as the limiting value. Therefore, the maximal value of $\delta_{2\varphi}^\varepsilon$ is bounded above as

$$
(A.12) \quad \max \delta_{2\varphi}^\varepsilon < \delta_{2\varphi}^\varepsilon = \frac{\delta_{2\varphi}^\varepsilon (1 + \varepsilon)}{\sqrt{1 + \varepsilon_{2\varphi}^\varepsilon (1 - \varepsilon)^{5/2}}} \leq \begin{cases} 1 + 5.500001 \varepsilon, & F = \mathbb{R}, \\ 1 + 11.500004 \varepsilon, & F = \mathbb{C}. \end{cases}
$$

First, $(\delta_{2\varphi}^\varepsilon - 1)/\varepsilon$, i.e., the factors multiplying $\varepsilon$ above, were expressed as functions of $\varepsilon$ in the scripts from Appendix A.3. The factors were symbolically computed for $\varepsilon \in \{2^{-11}, 2^{-24}, 2^{-53}, 2^{-113}\}$, evaluated with 50 digits of precision, manually rounded upwards to six decimal places, and the maximums over $p \in \{23, 52, 112\}$ were taken.$^{11}$

Minimizing $\delta_{2\varphi}^\varepsilon$ is equivalent to minimizing $\delta_{2\varphi}^\varepsilon$ as $\delta_{2\varphi}^\varepsilon$ from (A.7) and maximizing $d$, while setting $\varepsilon_6 = \varepsilon$ and $\varepsilon_7 = -\varepsilon$. Maximizing $d$ is equivalent to letting $r \to \infty$ in (A.11) and maximizing $b$, what amounts to setting $\varepsilon_4 = \varepsilon_5 = \varepsilon$ and maximizing $x$ by letting $y^2 \to \infty$ in (A.9) and taking $\varepsilon_{2\varphi}^\varepsilon$ from (A.10) as the limiting value. Therefore, the minimal value of $\delta_{2\varphi}^\varepsilon$ is bounded below (the factors multiplying $\varepsilon$ come from $(1 - \delta_{2\varphi}^\varepsilon)/\varepsilon$), as

$$
(A.13) \quad \min \delta_{2\varphi}^\varepsilon > \delta_{2\varphi}^\varepsilon = \frac{\delta_{2\varphi}^\varepsilon (1 - \varepsilon)}{\sqrt{1 + \varepsilon_{2\varphi}^\varepsilon (1 + \varepsilon)^{5/2}}} \geq \begin{cases} 1 - 5.500000 \varepsilon, & F = \mathbb{R}, \\ 1 - 11.500000 \varepsilon, & F = \mathbb{C}. \end{cases}
$$

$^{11}$Even though half precision ($p = 10$) is not otherwise considered in the context of this proof, it is worth noting that the factors in that case differ from the presented ones by less than 0.1.
Due to monotonicity of all arithmetic operations involved in computing $\text{fl}(\tan \varphi)$, its absolute value cannot exceed unity, regardless of $\delta_{\varphi'}$. Therefore, the magnitude of (either component of) $e' = \text{fl}(e^{\alpha}) \text{fl}(\tan \varphi)$ cannot increase from that of (the corresponding component of) $\text{fl}(e^{\beta'})$ after its multiplication by $\text{fl}(\tan \varphi)$.

From (2.5), (2.7), and (2.9) it follows

(A.14) \[ \text{fl}(\cos \varphi) = \frac{\delta_1}{\sqrt{(\delta_{\varphi'}^2 \tan^2 \varphi + 1)(1 + \varepsilon_8)}.} \]

As done previously, let $t = \tan \varphi$ and solve $(\delta_{\varphi'}^2) t^2 + 1 = (t^2 + 1)(1 + c)$ for $c$, to get

(A.15) \[ c = \frac{t^2}{t^2 + 1} \varepsilon_{\varphi'}, \quad \varepsilon_{\varphi'}^g = (\delta_{\varphi'}^2)^2 - 1, \quad 0 \leq |c| < |\varepsilon_{\varphi'}^g|, \]

and re-express (A.14), with $|\varepsilon_{\varphi}| \leq \varepsilon$ coming from the fma’s rounding, as

(A.16) \[ \text{fl}(\cos \varphi) = \frac{1}{\sqrt{\tan^2 \varphi + 1}} \frac{\delta_1}{\sqrt{1 + \varepsilon \sqrt{1 + \varepsilon_8}}} = \delta_{\varphi} \cos \varphi. \]

Maximizing $|\delta_{\varphi}| = \delta_{\varphi}^+$ amounts to setting $\varepsilon_8 = -\varepsilon$ and $\delta_1 = \delta_1^+$, while minimizing $c$ by letting $t^2 \to \infty$ in (A.15) and taking $\varepsilon_{\varphi'}^g = (\delta_{\varphi'}^2)^2 - 1$ as the limiting value. Therefore,

(A.17) \[ \max \delta_{\varphi} < \delta_{\varphi}^+ = \frac{\delta_1^+}{\sqrt{1 + \varepsilon_{\varphi'}^g \sqrt{1 - \varepsilon}}} \begin{cases} 1 + 8.000002 \varepsilon, & F = \mathbb{R}, \\ 1 + 14.000006 \varepsilon, & F = \mathbb{C}. \end{cases} \]

Minimizing $\delta_{\varphi}^-$ amounts to setting $\varepsilon_8 = \varepsilon$ and $\delta_1 = \delta_1^-$, while maximizing $c$ by letting $t^2 \to \infty$ in (A.15) and taking $\varepsilon_{\varphi'}^g = (\delta_{\varphi'}^2)^2 - 1$ as the limiting value. Therefore,

(A.18) \[ \min \delta_{\varphi} > \delta_{\varphi}^- = \frac{\delta_1^-}{\sqrt{1 + \varepsilon_{\varphi'}^g \sqrt{1 + \varepsilon}}} \begin{cases} 1 - 8.000000 \varepsilon, & F = \mathbb{R}, \\ 1 - 14.000000 \varepsilon, & F = \mathbb{C}. \end{cases} \]

The floating-point arithmetic operations involved in computing $\text{fl}(\cos \varphi)$ are monotonic, so $0 \leq \text{fl}(\cos \varphi) \leq 1$ regardless of $\delta_{\varphi'}^g$.

Computation of the eigenvalues proceeds, with $o = 2|a_{21}|$ and $\tilde{o} = \delta_{\varphi} o$, as

(A.19) \[ \text{fl}(\lambda_1') = \text{fma}(\text{fma}(a_{22}, \tilde{t}, -\tilde{o}), \tilde{t}, a_{11}), \]

\[ \text{fl}(\lambda_2') = \text{fma}(\text{fma}(a_{11}, \tilde{t}, -\tilde{o}), \tilde{t}, a_{22}), \]

where $\tilde{t} = \text{fl}(\tan \varphi)$ and $|\tilde{t}| \leq 1$. Then, with $\max\{|\varepsilon_{10}|, |\varepsilon_{11}|, |\varepsilon_{12}|, |\varepsilon_{13}|\} \leq \varepsilon$,

\[ \text{fl}(\lambda_1') = ((a_{22} \tilde{t} + \tilde{o})(1 + \varepsilon_{10}) \tilde{t} + a_{11})(1 + \varepsilon_{12}), \]

\[ \text{fl}(\lambda_2') = ((a_{11} \tilde{t} - \tilde{o})(1 + \varepsilon_{11}) \tilde{t} + a_{22})(1 + \varepsilon_{13}), \]

what gives, after taking the absolute values, applying the triangle inequality, and using (2.14) to bound $|a_{ij}|$ from above, with $\tilde{\delta}_{\varphi} = 1$ and $\delta_{\varphi'} = \delta_{\varphi}^2$,

(A.20) \[ |\text{fl}(\lambda')| \leq ((\tilde{a} + 2a_{22} \tilde{\delta}_{\varphi}')(1 + \varepsilon) + \tilde{a})(1 + \varepsilon) = \tilde{a}((1 + 2a_{22} \tilde{\delta}_{\varphi}')(1 + \varepsilon) + 1)(1 + \varepsilon), \]

where $|\text{fl}(\lambda')| = \max\{|\text{fl}(\lambda_1')|, |\text{fl}(\lambda_2')|\}$. Bounding $\tilde{a}$ by $\tilde{\omega}$ as in (2.14) gives

\[ |\text{fl}(\lambda')| \leq \tilde{\omega}((1 + 2a_{22} \tilde{\delta}_{\varphi}')(1 + \varepsilon) + 1)(1 + \varepsilon) = \tilde{\omega} \delta_{\varphi}^2. \]
Evaluating the scripts from Appendix A.3 for all ε considered above shows that 4 < δ^2 < 4√2 ≃ 5.656854, so |f(λ′)| < ω.

Since f(sec^2 φ) ≥ 1, dividing f(λ′) and f(λ′) by it therefore cannot raise their magnitudes, and the final computed eigenvalues cannot overflow—thus far, with the assumption that no final result of any previous computation has underflowed.

Regardless of the consequences of any underflow leading to \(\hat{t}\), \(|t| \leq 1\) in (A.19) due to (2.5), and \(|f(λ′)|\) can be bounded above, with \(a_2' = \max\{|a_{11}', |a_{22}'|\} \leq \bar{ω}\), by

\[
(a_1' + \bar{o})(1 + \varepsilon) + a_2'(1 + \varepsilon),
\]

similarly as in (A.20), letting in both inequalities \(\bar{t}\) = 1. If \(a_1'\) and \(\bar{o}\) are small, no overflow occurs. If \(a_2'\) is large enough, a small enough \(\bar{o}\), no matter if accurate or not, cannot affect it by addition or subtraction in the default rounding mode, so (A.21) becomes \(a_2'(2 + \varepsilon)(1 + \varepsilon)\). Vice versa, a small enough \(a_2'\) cannot affect a large enough \(\bar{o}\), so (A.21) simplifies to \(\bar{o}(1 + \varepsilon)^2\). No overflow is possible in either case.

**A.3.** The Wolfram Language scripts used in Appendix A.2. These scripts were executed by the Wolfram Language Engine, version 12.3.1 for macOS.

**A.3.1.** A script computing the relative error bounds for a real A. In Figure A.1, \(n\) is the number of digits of precision for \(\mathbb{R}[\ldots]\), and

\[
fem = (1 - \delta_2^R)/\varepsilon, \quad fep = (\delta_2^R - 1)/\varepsilon, \quad fem = (1 - \delta_3^R)/\varepsilon, \quad fem = (\delta_3^R - 1)/\varepsilon, \quad cem = (1 - \delta_3^R)/\varepsilon, \quad cep = (\delta_3^R - 1)/\varepsilon, \quad pel = \delta_3^R.
\]

**A.3.2.** A script computing the relative error bounds for a complex A. In Figure A.2, \(n\) is the number of digits of precision for \(\mathbb{C}[\ldots]\), and

\[
fam = (1 - \delta_2^C)/\varepsilon, \quad fap = (\delta_2^C - 1)/\varepsilon, \quad fam = (1 - \delta_3^C)/\varepsilon, \quad fam = (\delta_3^C - 1)/\varepsilon, \quad cem = (1 - \delta_3^C)/\varepsilon, \quad cep = (\delta_3^C - 1)/\varepsilon, \quad pel = \delta_3^C.
\]

**Appendix B.** Vectorized eigendecomposition of a batch of real symmetric matrices of order two. Algorithms B.1 and B.2 are the real counterparts of Algorithms 2.2 and 2.3. The complex algorithms work also with real symmetric matrices as a simple special case, but the real ones are faster. Note that the real algorithms return ± tan \(\varphi = e^{iα}\) tan \(\varphi = \text{sign} a_{21}\) tan \(\varphi\), as the complex ones do with a real input.

**Appendix C.** Overflow conditions of a real and a complex dot-product.

A real dot-product, \(x^Ty\), of columns \(x\) and \(y\) of length \(m\), can be bounded in magnitude using the triangle inequality as

\[
|\mathbf{x}^T \mathbf{y}| = \left| \sum_{i=1}^{m} x_i \cdot y_i \right| \leq \sum_{i=1}^{m} |x_i| \cdot |y_i| = \sum_{i=1}^{m} |x_i||y_i| = |\mathbf{x}|^T |\mathbf{y}| \leq m M^2,
\]

where \(M = \max_{1 \leq i \leq m} M_i\) and \(M_i = \max\{|x_i|, |y_i|\}\). From (C.1) and

\[
|f(\mathbf{x}^T \mathbf{y}) - \mathbf{x}^T \mathbf{y}| \leq m \varepsilon |\mathbf{x}|^T |\mathbf{y}|,
\]

assuming no particular order of evaluation, but also no overflow or underflow when computing \(f(\mathbf{x}^T \mathbf{y})\) (see [25, Theorem 4.2]), it follows

\[
|f(\mathbf{x}^T \mathbf{y})| = |\mathbf{x}^T \mathbf{y} + (f(\mathbf{x}^T \mathbf{y}) - \mathbf{x}^T \mathbf{y})| \leq |\mathbf{x}^T \mathbf{y}| + |f(\mathbf{x}^T \mathbf{y}) - \mathbf{x}^T \mathbf{y}|
\]

\[
\leq |\mathbf{x}|^T |\mathbf{y}| (1 + m \varepsilon) \leq m M^2 (1 + m \varepsilon).
\]
Algorithm B.1 d8jac2: a vectorized eigendecomposition of at most $s$ double precision real symmetric matrices of order two with the Intel’s AVX-512 intrinsics.

Input: $i$: addresses of $a_{11}, a_{22}, a_{21}, \pm \tan \varphi, \cos \varphi, \lambda_1, \lambda_2, p$
Output: $\pm \tan \varphi, \cos \varphi, \lambda_1, \lambda_2, p$ // a permutation-indicating bitmask

// vectors with all lanes set to a compile-time constant
1: $0 = \text{setzero}(); 1 = \text{set1}(1.0); 0 = \text{set1}(-1.0); \omega = \text{set1}(\text{DBL}_{\text{MAX}});$
2: $\sqrt{\omega} = \text{set1}(1.34078079299425956E+154); \eta = \text{set1}(1020.0); // fl(\sqrt{\omega}); (2.6)$
3: $a_{11} = \text{load}(a_{11} + i); a_{22} = \text{load}(a_{22} + i); a_{21} = \text{load}(a_{21} + i); // 3a_{21} = 0$
4: $\zeta_{11} = \text{sub}(\eta, \text{getexp}(a_{11})); \zeta_{22} = \text{sub}(\eta, \text{getexp}(a_{22})); \zeta_{21} = \text{sub}(\eta, \text{getexp}(a_{21}));$
5: $\phi = \min(\min(\zeta_{11}, \zeta_{22}), \min(\zeta_{21}, \omega)); // finalize $\phi$ from (2.17)
6: $-\zeta = \text{xor}(\zeta, 0); // \text{xor}(x, 0)$ flips the sign bits in $x$; optionally, store $-\zeta$
7: $a_{21} = \text{scalef}(a_{21}, \phi); a_{11} = \text{scalef}(a_{11}, \phi); a_{22} = \text{scalef}(a_{22}, \phi); a_{ij} = 2^{\phi}a_{ij}$
8: $|a_{21}| = \text{andnot}(-0, a_{21}); // \text{andnot}(x, 0)$ clears the sign bits
9: $\text{sgn}(a_{21}) = \text{and}(a_{21}, -0); // \text{and}(x, 0)$ extracts the sign bits
10: $\cos \varphi$ and $\pm \tan \varphi$ (or $\sin \varphi = \text{div}(\pm \tan \varphi, \sec \varphi)$
11: $\sec^2 2\varphi = \text{or}(\min(\max(\text{div}(\eta, a_{ij}), 0), \sqrt{\omega}), \text{sgn}(a_{ij})); (2.6)$
12: $\sec^2 2\varphi < \infty$
13: $\tan \varphi = \text{div}(\text{tan} 2\varphi, \text{add}(1, \sqrt{\text{sec}^2 2\varphi})); (2.5)$
14: $\sec^2 \varphi = \text{fmadd}(\text{tan} \varphi, \text{tan} \varphi, 1); // \text{sec}^2 \varphi < \infty$
15: $\sec \varphi = \text{sqrt}(\text{sec}^2 \varphi); \cos \varphi = \text{div}(1, \text{sec} \varphi); (2.7)$
16: $\pm \tan \varphi = \text{xor}(\tan \varphi, \text{sgn}(a_{21})); // \text{xor}$ multiplies the signs of its arguments
17: $\text{store}(\pm \tan \varphi + i, \pm \tan \varphi); // e^{i\phi} \tan \varphi = \text{sign} a_{21} \tan \varphi$

// the eigenvalues
18: $\lambda'_1 = \text{div}(\text{fmadd}(\tan \varphi, \text{fmadd}(a_{22}, \tan \varphi, 0), a_{11}), \text{sec}^2 \varphi); (2.8)$
19: $\text{store}(\cos \varphi + i, \cos \varphi); // \text{below}: \text{fmsub}(x, y, z) \equiv \text{fmad}(x, y, -z)$
20: $\lambda'_2 = \text{div}(\text{fmadd}(\tan \varphi, \text{fmsub}(a_{11}, \tan \varphi, 0), a_{22}), \text{sec}^2 \varphi); (2.8)$
21: $\lambda' = \text{scalef}(\lambda'_1, -\zeta); \text{store}(\lambda' + i, \lambda'_1); // \text{backs}c\text{ale \text{and \text{store}} \lambda'}$
22: $p = \text{mm512}_\text{cmpl}_{\text{pd}}\_\text{mask}(\lambda'_1, \lambda'_2); // \text{lane-wise check if } \lambda'_1 < \lambda'_2$
23: $\lambda_1 = \text{scalef}(\lambda'_2, -\zeta); \text{store}(\lambda_1 + i, \lambda'_2); // \text{backs}c\text{ale \text{and \text{store}} \lambda_1}$
24: $p[1] = \_\text{cvtm}_{\text{axw}}\_\text{u32}(p); // \text{store p to } p, x = 8 (16 \text{ for AVX512}F)$

// alternatively, in lines 21 and 23, return $\lambda'_1$ and $\lambda'_2$, resp., instead of backsca}

Algorithm B.2 dbjac2: an OpenMP-parallel, AVX-512-vectorized eigendecomposition of a batch of $\tilde{r}$ double precision real symmetric matrices of order two.

Input: $\bar{f}$: $a_{11}, a_{22}, a_{21}$; $p$ also, in the context of Algorithm 4.1[\$] only.
Output: $\pm \tan \varphi, \cos \varphi, \lambda_1, \lambda_2, p$ // unsigned array of length $\tilde{r}$/s
1: #pragma omp parallel for default(shared) // optional
2: for $i = 0$ to $\tilde{r} - 1$ step $s$ do // optional
3: if $p[i] <= 0$ then continue; // skip this vector on request of Algorithm 4.1[\$]
4: d8jac2($i, a_{11}, a_{22}, a_{21}, \pm \tan \varphi, \cos \varphi, \lambda_1, \lambda_2, p[i], -\zeta]); // Algorithm B.1
5: end for // optional backsca}


Formation of the Grammian matrices of order two in the real case, if (2.14) is to be achieved without any scaling, requires $|\text{fl}(x^T y)| \leq \tilde{\omega}$, so it suffices to hold

$M \leq \sqrt{\omega/(4\sqrt{2}m(1 + m\varepsilon))} = \tau_m$.  

A complex dot-product, $w^* z$, can be decomposed into a sum of two real ones, each twice longer than the column length, as

$$w^* z = \Re(w^* z) + i\Im(w^* z)$$

$$= (\Re w)^T (\Re z) + (\Im w)^T (\Im z) + i((\Re w)^T (\Im z) - (\Im w)^T (\Re z))$$

$$= [(\Re w)^T (\Im w)^T] [\Re z \Im z] + i [(\Re w)^T (\Im z)] [\Im z \Re z]$$

$$= a^T b + i c^T d,$$

where $a$, $b$, $c$, and $d$ stand for the corresponding real arrays of length $\hat{m} = 2m$.

Let $\hat{M}_i = \max\{|\Re w_i|, |\Im w_i|, |\Re z_i|, |\Im z_i|\}$ and $\hat{M} = \max_{1 \leq i \leq m} \hat{M}_i$. Similarly to the
real case, from (C.2) and (C.5) it then follows

\[
|f(a^T b)| = |a^T b + (fl(a^T b) - a^T b)| \leq |a^T b| + |fl(a^T b) - a^T b| \\
\leq |a^T| |b| (1 + \hat{m} \varepsilon) \leq \hat{m} M^2 (1 + \hat{m} \varepsilon).
\]

The same relation holds if \(a\) and \(b\) are fully replaced by \(c\) and \(d\), respectively. Since
fl(w^T z) = fl(a^T b) + i fl(c^T d), its magnitude can be bounded above, using (C.6), by

(C.7) \[ |fl(w^T z)| = \sqrt{|fl(a^T b)|^2 + |fl(c^T d)|^2} \leq \sqrt{2\tilde{M}^2}(1 + \tilde{\omega} \varepsilon). \]

If |fl(w^T z)| \leq \tilde{\omega} is required, then due to (C.7) it suffices to hold

(C.8) \[ \tilde{M} \leq \sqrt{\omega/(8\tilde{m}(1 + \tilde{m} \varepsilon))} = \sqrt{\omega/(16m(1 + 2m \varepsilon))} = \tilde{r}_m. \]

Issues with underflow are ignored for the naïve Jacobi SVD. Let \( \tilde{r}_m = r_m \) in the real, and \( \tilde{r}_m = \tilde{r}_m \) in the complex case. The constraints (C.4) and (C.8) on the magnitudes of (the components of) the elements of the iteration matrix \( G_k \) become

(C.9) \[ \max\{\|\Re(G_k)\|_{\max}, \|\Im(G_k)\|_{\max}\} \leq \tilde{r}_m. \]

This equation establishes a “safe region” for the magnitudes of (the components of) the elements of the iteration matrix, within which it is guaranteed that both the formation of the Grammians and the calculation of the Jacobi rotations without the prescaling from subsection 2.3 will succeed. Not only that this region is relatively narrow, but the transformations of the pivot column pairs could cause the iteration matrix to fall outside it at the beginning of the following iteration, as shown in subsection 3.1.

If the iteration matrix \( G_k \) is scaled as in (C.9), no element of \( fl(G_{k+1}) \) can overflow, due to Proposition 3.3. The constraint (C.9), if re-evaluated by examining the magnitudes of the affected (components of) elements, could become violated, which is acceptable as long as all dot-products remain below the limit (2.14) by magnitude and the assumption of Proposition 3.3 holds. If it does not hold, or if, once all dot-products for the current step are obtained, at least one lands above (2.14), the iteration matrix has to be rescaled according to (C.9). These observations suffice for a fast implementation of the pointwise Jacobi-type SVD method, applicable when the input matrix so permits.

**Appendix D. Proofs of Lemmas 3.1 and 3.2.** See their statements in the main paper.

*Proof of Lemma 3.1.* From \( g_{ip}' = (g_{ip} \pm g_{iq} \text{fl}(\tan \varphi)) \text{fl}(\cos \varphi) \) it follows

(D.1) \[ \text{fl}(g_{ip}') = ((g_{ip} \pm g_{iq} \text{fl}(\tan \varphi))(1 + \varepsilon_1) \text{fl}(\cos \varphi))(1 + \varepsilon_2), \]

due to the final and only rounding performed when evaluating the fused multiply-add expression, with the relative error \( \varepsilon_1, |\varepsilon_1| \leq \varepsilon \). It can now be obtained, by rearranging the above terms and noting that, for the multiplication by the cosine, \( |\varepsilon_2| \leq \varepsilon \,

\[ \text{fl}(g_{ip}') = ((g_{ip} \pm g_{iq} \text{fl}(\tan \varphi)) \text{fl}(\cos \varphi))(1 + \varepsilon_1)(1 + \varepsilon_2) = g_{ip}'(1 + \varepsilon_1^p)(1 + \varepsilon_2), \]

and thus \((1 - \varepsilon)^2 \leq 1 + \varepsilon_1^p \leq (1 + \varepsilon)^2\), what proves the first part of the first statement of the Lemma. The second part, \((1 - \varepsilon)^2 \leq 1 + \varepsilon_1 \leq (1 + \varepsilon)^2\), is shown similarly.

From (D.1), \(|\text{fl}(\tan \varphi)| \leq 1\), the assumption that \( \max\{|g_{ip}|, |g_{iq}|\} \leq \omega/2\), and the fact that \( \omega/2 \) is exactly representable in floating-point, so \( \text{fl}(\omega/2 + \omega/2) = \omega \), it follows that \(|d(1 + \varepsilon_1)| \leq \omega\), for \( d = g_{ip} \pm g_{iq} \text{fl}(\tan \varphi) \) (or for \( d = g_{iq} \mp g_{ip} \text{fl}(\tan \varphi) \)). Here, monotonicity of the inner addition, the inner multiplication, and the (outer) rounding of the fma operation are relied upon. Multiplying \( d(1 + \varepsilon_1) \) by \( \text{fl}(\cos \varphi) \leq 1 \) and the subsequent (monotonous) rounding cannot yield a result of a magnitude strictly greater than \(|d(1 + \varepsilon_1)| \leq \omega\), what proves the last statement of the Lemma.
Proof of Lemma 3.2. Assume that $|g'_{ip}| > 0$ and $|g'_{ip}|/\text{fl}(\cos \varphi) \geq \max |\Re g_{iq}|$ hold. Let, from the first equation in (3.1), $d = g'_{ip}/\text{fl}(\cos \varphi) = a \cdot b + c$, where $a = g_{iq}$.

$c = g_{ip}$, and $\bar{d}$ is $\text{fl}(a \cdot b, c) = \text{RN}(d)$, in the context of (2.10). In this notation, $0 \neq |d| \geq |\Re a|$. Let max\{ε₁, |ε₂|, |ε₃|, |ε₄|, |ε₅|, |ε₆|\} ≤ ε. Then,

\[
\Re \bar{d} = (\Re a \cdot \Re b + (\Re c - 3a \cdot \Re b)(1 + \varepsilon₁)(1 + \varepsilon₂) = (\Re d + (\Re c - 3a \cdot \Re b)\varepsilon₁)(1 + \varepsilon₂),
\]

\[
\Im \bar{d} = (\Re a \cdot \Re b + (\Re c - 3a \cdot \Re b)(1 + \varepsilon₂)) \cdot \Re \varepsilon₂ = (\Re d + (\Re c - 3a \cdot \Re b)\varepsilon₂)(1 + \varepsilon₂).
\]

After subtracting $\Re d$ (3d), rearranging the terms, and adding a zero, it follows

\[
\Re \bar{d} - \Re d = \Re d(\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂) - \Re a \cdot \Re b \cdot \Im \varepsilon₂(1 + \varepsilon₂),
\]

\[
\Im \bar{d} - \Im d = \Im d(\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂) - \Re a \cdot \Im b \cdot \Im \varepsilon₂(1 + \varepsilon₂),
\]

i.e., after regrouping the terms and extracting $\Re d$ (3d) from the right hand sides,

\[
\Re \bar{d} - \Re d = \Re d(\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂) - \Re a \cdot \Re b \cdot \Re \varepsilon₂(1 + \varepsilon₂),
\]

\[
\Im \bar{d} - \Im d = \Im d(\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂) - \Re a \cdot \Im b \cdot \Im \varepsilon₂(1 + \varepsilon₂).
\]

Note that $|\text{fl}(\tan \varphi)| \leq 1$, so an argument similar to the one used in the proof of Proposition 2.4 ensures that $\varepsilon₁$ and $\varepsilon₂$ can be ignored (considered to be zero) when taking max\{ε₁, |ε₂|, |ε₃|, |ε₄|, |ε₅|, |ε₆|\} ≤ ε, regardless of a possible inaccuracy of $\varepsilon'$ indicated by Remarks 2.2 and 2.3, since even the components of such a pathological $\varepsilon'$ are at most unity by magnitude.

Taking the absolute values of the two previous equations, it follows

\[
|\Re \bar{d} - \Re d| \leq |\Re d| |\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂| + |\Re a| \delta^*_ε |\varepsilon₁|(1 + \varepsilon₂),
\]

\[
|\Im \bar{d} - \Im d| \leq |\Im d| |\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂| + |\Re a| \delta^*_ε |\varepsilon₂|(1 + \varepsilon₂).
\]

Dividing by $|d|$ and applying again the triangle inequality simplifies (D.2) to

\[
|\Re \bar{d} - \Re d|/|d| \leq |\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂| + |\Re a| |\delta^*_ε |\varepsilon₁|(1 + \varepsilon₂|),
\]

\[
|\Im \bar{d} - \Im d|/|d| \leq |\varepsilon₁ + \varepsilon₂ + \varepsilon₂\varepsilon₂| + |\Re a| |\delta^*_ε |\varepsilon₂|(1 + \varepsilon₂),
\]

or, denoting by $\varepsilon$ either $A$ or $B$ and bounding each $|\varepsilonᵢ|$, 1 ≤ i ≤ 4, by $|\varepsilon|$

\[
\frac{|\varepsilon \bar{d} - \varepsilon d|}{|d|} \leq \hat{\varepsilon}, \quad 3.00000 \varepsilon < \hat{\varepsilon} < 3.00000 \varepsilon,
\]

where the approximate multiples of $\varepsilon$ are computed by the script Figure A.2 as $\varepsilon'$. From (D.3) magnitude of each component of $d$ can be bound relative to $|d|$ as

\[
\frac{|\varepsilon \bar{d}|}{|d|} = \frac{|\varepsilon \bar{d} - \varepsilon d + \varepsilon d|}{|d|} \leq \frac{|\varepsilon \bar{d} - \varepsilon d|}{|d|} + \frac{|\varepsilon d|}{|d|} \leq \hat{\varepsilon} + 1.
\]

Since $g'_{ip} = d \text{fl}(\cos \varphi)$, it holds

\[
\text{fl}(\Re g'_{ip}) = \Re \bar{d} \text{fl}(\cos \varphi)(1 + \varepsilon₀), \quad \text{fl}(\Im g'_{ip}) = \Im \bar{d} \text{fl}(\cos \varphi)(1 + \varepsilon₀).
\]
Thus, from $\text{fl}(\cos \varphi) > 0$ and (D.3) and (D.4), it follows

$$
|\text{fl}(\Re g'_{ip}) - \Re g_{ip}|/|g'_{ip}| = |\Re \tilde{d}\text{fl}(\cos \varphi)(1 + \epsilon_5) - \Re \tilde{d}\text{fl}(\cos \varphi)|/|(d|\text{fl}(\cos \varphi))
\leq (|\Re \tilde{d} - \Re d|/|d| + |\Re \tilde{d} - \epsilon_5|/|d|) \leq \epsilon + (1 + \epsilon)|\epsilon_5| \leq \epsilon + \epsilon + \epsilon_5,
$$

$$
|\text{fl}(\Im g'_{ip}) - \Im g_{ip}|/|g'_{ip}| = |\Im \tilde{d}\text{fl}(\cos \varphi)(1 + \epsilon_6) - \Im \tilde{d}\text{fl}(\cos \varphi)|/|(d|\text{fl}(\cos \varphi))
\leq (|\Im \tilde{d} - \Im d|/|d| + |\Im \tilde{d} - \epsilon_6|/|d|) \leq \epsilon + (1 + \epsilon)|\epsilon_6| \leq \epsilon + \epsilon + \epsilon_6,
$$

or, with $\epsilon' = \epsilon + \epsilon + \epsilon_5$, $|\text{fl}(\mathcal{C} g'_{ip}) - \mathcal{C} g_{ip}| / |g'_{ip}| \leq \epsilon'$. Finally, with $\epsilon'' = \sqrt{2}\epsilon'$,

$$
|\text{fl}(g'_{ip}) - g_{ip}|/|g_{ip}| = \sqrt{(|\text{fl}(\Re g'_{ip}) - \Re g_{ip}|/|g'_{ip}|)^2 + (|\text{fl}(\Im g'_{ip}) - \Im g_{ip}|/|g'_{ip}|)^2} \leq \epsilon'',$

$$
5.656854 \epsilon < \epsilon'' < 5.656856 \epsilon,
$$

where the approximate multiples of $\epsilon$ are computed by the script Figure A.2 as \text{epp}. A similar proof is valid for $|\text{fl}(g'_{ip}) - g_{ip}|/|g_{ip}| \leq \epsilon''$ when $0 < |g_{ip}|/\text{fl}(\cos \varphi) \geq |\Re g_{ip}|$.

Take a look at what happens with the “else” parts of the claims 1 and 2 of Lemma 3.2. If $|g'_{ip}|/\text{fl}(\cos \varphi) < |\Re g_{ip}|$, then $|g'_{ip}| < |g_{ip}|$, and if $|g'_{ip}|/\text{fl}(\cos \varphi) < |\Re g_{ip}|$, then $|g'_{ip}| < |g_{ip}|$. In exact arithmetic, the max-norm of the iteration matrix could not have been affected by either transformation. When computing in floating-point, from $|\Re a|/|d| > 1$ and (D.2) follows that the relative error $|\mathcal{C} d - \mathcal{C} d|/|d|$ is no longer bounded above by a constant expression in $\epsilon$. However, if (D.2) is divided by $|a|$ (not by $|d|$), since $\max\{|\Re d|, |\Im d|\} \leq |d| < |\Re a| \leq |a|$, it follows

$$
|\Re \tilde{d} - \Re d|/|a| \leq \epsilon, \quad |\Im \tilde{d} - \Im d|/|a| \leq \epsilon, \quad |\tilde{d} - d|/|a| < \sqrt{2}\epsilon,
$$

and thus $|\tilde{d} - d| < \sqrt{2}|a|\epsilon$, so $\tilde{d}$ lies in a disk with the center $d$ and the radius $\sqrt{2}|a|\epsilon$. If $|a|(1 + \sqrt{2}\epsilon) \leq \omega$, then $|d| < \omega$ since $|d| < |a|$. The final transformed element is $\text{fl}(\tilde{d}\text{fl}(\cos \varphi))$, with $\text{fl}(\cos \varphi) \leq 1$, so $|\text{fl}(\tilde{d}\text{fl}(\cos \varphi))| < \omega$ (due to the monotonicity argument, applied here component-wise), what had to be proven.

Regarding the last statement of the Lemma, note that from $\max\{|\Re b|, |\Im b|\} \leq \delta_+^\ast$ and $\max\{|\Re a|, |\Im a|, |\Re c|, |\Im c|\} \leq \omega/4$ follows that neither the inner nor the outer real \text{fma} operations in (2.10) can overflow, since the magnitude of the result of the inner one cannot be greater than $\omega(1 + \delta_+^\ast)(1 + \epsilon)/4$, so the result of the outer one cannot exceed $\omega h/4$ in magnitude, where

$$
h = h(\epsilon) = (\delta_+^\ast + (1 + \delta_+^\ast)(1 + \epsilon))(1 + \epsilon).
$$

Solving numerically for $\epsilon$ such that $4 - h(\epsilon) = 0$ gives two real roots, but only one, $\epsilon_2, 6.520087 \cdot 10^{-2} < \epsilon_2 < 6.520088 \cdot 10^{-2}$, in $[0, 1/4]$. For all $\epsilon$ such that $0 \leq \epsilon \leq \epsilon_2$, what includes $\epsilon$ of all standard floating-point datatypes, it holds $h(\epsilon) \leq 4$, and thus $\omega h/4 \leq \omega$. Having obtained $\tilde{d}$, multiplying it by $\text{fl}(\cos \varphi) \leq 1$ cannot increase its components’ magnitudes, and thus no overflow occurs in any rounding.

Assume that an underflow occurs in (2.10), when rounding the result of an inner or of an outer real \text{fma}. If it is an outer \text{fma}, such a computed component (be it real or imaginary) of $\tilde{d}$ is subnormal. If it is an inner \text{fma}, when the affected component of $\tilde{d}$ is computed, it cannot be greater than $\omega \delta_+^\ast(1 + \epsilon)/4$ in magnitude. Also, if an underflow occurs when forming either component of $\text{fl}(d\text{fl}(\cos \varphi))$, each component stays below $\omega$ in magnitude. It may therefore, for the purposes of this part of the proof, be assumed in the following that no underflow occurs in any rounding of (3.1) and thus the conditions of the first, already proven part of this Lemma are met.
From (3.1), Remarks 2.2 and 2.3, and max{|g_{ip}|,|g_{iq}|} \leq \omega/4 it follows
\[|g_{ip}'| = |g_{ip} + g_{iq}e^r| \leq \left|g_{ip}\right| + \sqrt{2}|g_{iq}| \leq \omega(1 + \sqrt{2})/4.\]
If \(0 \neq |d| \geq |\Re|\), then \(|g_{ip}'| - g_{ip}' \leq |g_{ip}'|\varepsilon''\), what implies
\[|\Re(g_{ip}')| \leq |g_{ip}'|(1 + \varepsilon'') \leq \omega(1 + \sqrt{2})(1 + \varepsilon'')/4 \leq \omega,\]
and similarly for \(|\Re(g_{iq}')|\), if applicable. Else, \(|g_{iq}'| \leq \omega/(1 + \sqrt{2}\varepsilon)\) since \(\sqrt{2}\varepsilon \leq 3\). Therefore, \(|\Re(g_{ip}')| < \omega\) (and similarly for \(|\Re(g_{iq}')|\), if applicable), what concludes the proof.\]

**Appendix E. Converting the complex values to and back from the split form.** Algorithm E.1 shows how to transform the customary representation of complex numbers into the split form. This processing is sequential (could also be parallel) on each column of \(G\), and the columns are transformed concurrently. Repacking of the columns of \(\Re X\) and \(\Im X\) back to \(X\), for \(X \in \{G, V\}\), as in Algorithm E.2, is performed in parallel over the columns, and sequentially within each column.

**Algorithm E.1** Splitting a vector \(i\)g of four double complex numbers from \(g_j\), with their real and imaginary components customary interleaved, into two vectors of the packed real and imaginary parts, respectively, by a lane permutation \(ps\).

```plaintext
1: __m512i ps = __mm512_set_epi64(7, 5, 3, 1, 6, 4, 2, 0);
2: for i = 1 to m step 4 do // sequentially, but could be done in parallel
3:   ig = load((G)_{ij}); // \([\Re g_{ij}, \Im g_{ij}]^{i+3}_{r=1}\), i.e., interleaved
4:   pg = permutexvar(ps, ig);
5:   __mm256_store_pd((\Re G)_{ij}, extractf64x4(pg, 0)); // \([\Re g_{ij}]^{i+3}_{r=1}\)
6:   __mm256_store_pd((\Im G)_{ij}, extractf64x4(pg, 1)); // \([\Im g_{ij}]^{i+3}_{r=1}\)
end for // output: \(\Re g_j\) and \(\Im g_j\)
```

**Algorithm E.2** Merging, by a lane permutation \(pm\), of two vectors, \(r\) from \(\Re X\) and \(i\) from \(\Im X\), of the real and the imaginary parts, respectively, of four double complex numbers into a vector from \(X\) with the numbers’ parts customarily interleaved.

```plaintext
1: __m512i pm = __mm512_set_epi64(7, 3, 6, 2, 5, 1, 4, 0);
2: for i = 1 to #rows(X) step 4 do // sequentially, but could be done in parallel
3:   _m256d r = load((\Re X)_{ij}), i = load((\Im X)_{ij}); // \([\Re X]^{i+3}_{r=1}, [\Im X]^{i+3}_{r=1}\)
4:   ri = __mm512_insertf64x4(__mm512_zextpd256_xmm512_permute421042512d(pm, r, i)); // \([\Re X]^{i+3}_{r=1}\)
5:   il = permutexvar(pm, ri); // \([\Im X]^{i+3}_{r=1}\)
6:   store((X)_{ij}, il); // store the values with their parts interleaved
end for // output: \(x_j\)
```

**Appendix F. Vectorized non-overflowing computation of the Frobenius norm.** Here, a method is proposed in which the exponent range of all partial sums of the squares of the input array’s elements, as well as of the final result, is sufficiently widened to avoid obtaining an infinite value for any expected array length, but the number of significant digits is unaltered from that of the input’s datatype (\texttt{double}).

The method’s operation is conceptually equivalent to that of a vectorized \(x^2\) dot-product, shown in Algorithm F.1, and thus their outputs are generally identical when the number of lanes \(s\) is same for both, except in the cases of overflowing (or...
Algorithm F.1 A vectorized implementation of a dot-product $\mathbf{x}^T \mathbf{x}$ using fma.

1: $\text{ssq} = \text{setzero}()$; // initially, the partial sums of squares is a vector of zeroes
2: for $i = 1$ to $m$ step $s$ do
3: \quad $x = \text{load}(x_i);$ // $x = \left[ x_i \ x_{i+1} \cdots x_{i+s-1} \right]$ 
4: \quad $\text{ssq} = \text{fmad}(x, x, \text{ssq});$ // $\text{ssq} = x^T x + \text{ssq}$
5: end for
6: sum-reduce $\text{ssq}$ and return the result; \hspace{1em} // optionally, pre-sort $\text{ssq}$

extreme underflowing to zero) of the results of the Algorithm F.1, while the proposed method returns a finite (or non-zero) representation, respectively, by design.

The final $\text{ssq}$ could be sorted by the algorithm from [9, Appendix B], and the sum-reduction order in Algorithm F.1 could be taken from Algorithm F.3 for comparison with the results of the latter, or the reduction could proceed sequentially.

F.1. Input, output, constraints, and a data representation. Let $\mathbf{x}$ be an input array, with all its elements finite. A partial sum of their squares $r$ (including the resulting $\|\mathbf{x}\|_2^2$) is represented as $(e, f)$, where $1 < f < 2$ is the "fractional" part of $r$, while $e = \lfloor \log r \rfloor$ is the exponent of its power-of-two scaling factor, such that $r = 2^e f$. Both $e$ and $f$ are floating-point quantities, and $0 = (-\infty, 1)$. For $r > 0$, $e$ is a finite integral value. For example, $12 = 2^3 \cdot 1.5 = (3, 1.5)$ and $80 = 2^5 \cdot 1.25 = (6, 1.25)$.

To get $\|\mathbf{x}\|_F = \sqrt{\|\mathbf{x}\|_2^2} = \sqrt{(e, f)}$ when $e$ is finite and odd, take $f' = 2f$, $e' = e - 1$, and compute $\|\mathbf{x}\|_F = (e'/2, \text{fl}(\sqrt{f'}))$, since $\text{fl}(\sqrt{2}) \leq \text{fl}(\sqrt{f'}) < 2$ and $e'$ is even. For example, $\sqrt{12} = \sqrt{(3, 1.5)} = ((3 - 1)/2, \text{fl}(\sqrt{1.5})) = (1, \text{fl}(\sqrt{3}))$. If $e$ is infinite or even, set $f' = f$ and $e' = e$; e.g., $\sqrt{80} = \sqrt{(6, 1.25)} = (3, \text{fl}(\sqrt{1.25}))$. Let $e'' = e'/2$ and $f'' = \text{fl}(\sqrt{f'})$. The exact value of $2^{e''} f''$ could be greater than $\omega$, but $\|\mathbf{x}\|_F = (e'', f'')$, for $\mathbf{x} \neq 0$, remains representable by two finite quantities of the input’s datatype.

For example, let $\mathbf{x} = \mathbf{\omega} = [\omega \cdots \omega]^T$ of length $m \geq 2$. Then, $\|\mathbf{x}\|_F = \omega \sqrt{m}$ (mathematically), so $\text{fl}(\|\mathbf{x}\|_F) = \infty$, but $e''$ is finite (as well as $f''$) and constrained only by the requirement that all operations with finite exponents as arguments have to produce the exact result, as if performed in integer arithmetic, without any rounding. All finite exponents throughout the computation should thus be at most $\hat{\omega}$ in magnitude, where, in single precision, $\hat{\omega} = 2^{31}$, and in double precision [12], $\hat{\omega} = 2^{53}$. For every partial sum $r$ it has to hold $r < 2^{\hat{\omega} - 1}$, a limit that could hardly ever be reached.

F.2. Vectorized iterative computation of the partial sums. Given $\mathbf{x}$ of length $m \geq 1$ (zero-padded to $\hat{m}$ as in (2.18)), let $|x_i| = (E(x_i), F(x_i)) = (e_i, f_i)$, where $i \geq 0$, $E(\mathbf{x}) = \text{getexp}(\mathbf{x})$, $F(\mathbf{x})$ is given by (3.7), and $x_i = [x_i \ x_{i+1} \cdots x_{i+s-1}]^T$ is a vector with $s$ lanes, loaded from a contiguous subarray of $\mathbf{x}$ with the one-based indices $i$ such that $s + 1 \leq i \leq (1 + 1)s$.

Let, for $i \geq 0$, $r_i = (e_i, f_i)$, be a pair of vectors representing the $i$th partial sums, and let $r_0$ represent $s$ zeros as $(e^0, f^0)$. Conceptually, the following operation has to be defined,

$$r_{i+1} = \text{fma}(|x_i|, |x_i|, r_i),$$

i.e., $(e_i^{(i+1)}, f_i^{(i+1)}) = fma((e_i, f_i), (e_i, f_i), (e_i^{(i)}, f_i^{(i)})))$, where such a function is implemented in terms of SIMD instructions as follows.

\[12^\text{\hat{\omega}} = 2^{31} - 1 \text{ in double precision with the AVX512F instruction subset only}\]
First, define \( r'_i = (e^{(i)}, f^{(i)}) \) as a “non-normalized” representation of the same values as \( r_i \), but with each exponent even or infinite. To obtain \( r'_i \) from \( r_i \), \( e^{(i)} \) has to be converted to a vector \( \hat{e}^{(i)} \) of signed integers and then their least significant bits have to be extracted. In the AVX512F instruction subset, only a conversion from floating-point values to 32-bit integers is natively supported\(^{13}\), so the integral exponents are obtained as 
\[
\hat{e}_{\text{all}}^{(i)} = \_\text{mm512_cvtpx}_\text{epi32}(e^{(i)}), \quad X \in \{ s, d \},
\]
where \(-\infty \) gets converted to INT\_MIN = \(-2^{31}\). The least significant bit of each exponent is extracted by the bitwise-and operation as 
\[
\hat{e}_{\text{lsb}}^{(i)} = \_\text{mm256\_and\_si256}(\hat{e}_{\text{all}}^{(i)}),
\]
where a lane of the result is one if and only if the corresponding exponent is odd, and zero otherwise. Then, \( \hat{e}_{\text{lsb}}^{(i)} \) is converted to a floating-point vector \( \hat{e}^{(i)} = \text{cvtepi32}(\hat{e}_{\text{lsb}}^{(i)}) \).

It can now be defined that \( e^{(i)} = \text{sub}(e^{(i)}, \hat{e}^{(i)}) \) and \( f^{(i)} = \text{scalef}(f^{(i)}, \hat{e}^{(i)}) \). This process of computing \( (e^{(i)}, f^{(i)}) \) from \( (e^{(i)}, f^{(i)}) \) is a vectorization of the scalar computation of \( (e', f') \) from \( (e, f) \), given in Appendix F.1, but with a different purpose.

With \( 1 \) being a vector of all ones, and \( \hat{e}_i = \text{scalef}(e_i, 1) \), observe that the “normalized” exponents of \( |x_i|^2 \), lane-wise, should be either those from \( \hat{e}_i \) (all even or infinite) or greater by one, since \( 1 \leq f_i^* < 4 \). Therefore, a provisory, non-normalized exponent vector of \( |x_i|^2 + r'_i \) can be taken as \( e_{\text{max}} = \max(e_i, e^{(i)}) \), in which all exponents are even or infinite since the same holds for both \( \hat{e}_i \) and \( e^{(i)} \) by design.

In general, \( \hat{e}_i \) and \( e^{(i)} \) are not equal. To compute \( |x_i|^2 + r'_i \), the exponent of each scalar addend has to be equalized such that the fractional part of the addend having the smaller exponent \( e_{\text{min}} \) is scaled by two to the power of the difference of \( e_{\text{min}} \) and the larger exponent \( e_{\text{max}} \). Having the exponents equalized to \( e_{\text{max}} \), the computation can proceed with the scaled fractional parts and the fused multiply-add operation.

Let \( e'_i = \max(\text{sub}(\hat{e}_i, e_{\text{max}}), -\infty) \), with the maximum taken to filter out a possible NaN as the result of \(-\infty (-\infty)\), when instead \(-\infty\) is desired, and similarly let \( e''^{(i)} = \max(\text{sub}(e^{(i)}, e_{\text{max}}), -\infty) \). Since all lanes of \( \hat{e}'_i \) are even or infinite, they can be divided by two as \( \hat{e}''_i = \text{scalef}(\hat{e}'_i, -1) \). The scaling of \( f_i^* \) by \( 2^{e''_i} \) is mathematically equivalent to the scaling of \( f_i \) by \( 2^{e_i} \) before squaring the result, but the latter does not require explicit computation of \( f_i^* 2^{e''_i} \) and thus has a lower potential for underflow, since the squaring happens as a part of an fma computation, and such an intermediate result is not rounded. Let \( \hat{f}_i = \text{scalef}(f_i, \hat{e}'_i), \hat{f}^{(i)} = \text{scalef}(f^{(i)}, e''^{(i)}) \), and define

\[
r''_{i+1} = (e^{(i+1)}, f^{(i+1)}) = (e_{\text{max}}, \text{fmad}(\hat{f}_i, \hat{f}_i, \hat{f}^{(i)}))
\]
as a non-normalized representation of the value of \( r_{i+1} \). To get its normalized form, let \( F^{(i+1)} = F(f^{(i+1)}), e^{(i+1)} = \text{add}(e^{(i+1)}, E(f^{(i+1)})) \), and \( r_{i+1} = (e^{(i+1)}, f^{(i+1)}) \).

In the case of a complex input in the form (2.19), in each iteration the first \( x_i \) is taken from \( \Re \tilde{z} \), and the second one from \( \Im \tilde{z} \). After \( m = \hat{m}/s \) iterations, this main part of the method terminates, with the final partial sums \( r_n \) in the normalized form.

**Prefetching \( x_n \).** Before the main loop, \( x_n \) is prefetched to the L1 cache, as well as each \( x_{i+1} \) at the start of the \( i \)th iteration, using _mm_prefetch(…, _MM\_HINT\_TO).

**F.3. Horizontal reduction of the final partial sums.** Now \( r_n \) has to be sum-reduced horizontally, i.e.,

\[
\|x\|_F^2 = \sum_{l=1}^s (e_l^{(n)}, f_l^{(n)}) = \sum_{l=1}^s (e_l, f_l) = (e, f),
\]

where \( l \) indexes the SIMD lanes of \( e^{(n)} \) and \( f^{(n)} \).

\(^{13}\)With AVX512DQ instructions a double precision \( e^{(i)} \) can be converted to 64-bit integers.
F.3.1. Vectorized sorting of the final partial sums. Accuracy of the result \((e_i, f_i)\) might be improved by sorting the \((e_i, f_i)\) pairs lexicographically (i.e., non-decreasingly by the values they represent) before the summation, such that

\[
(F.2) \quad (e_i, f_i) \leq (e_j, f_j) \iff (e_i < e_j) \lor ((e_i = e_j) \land (f_i \leq f_j)).
\]

If \((F.1)\) is evaluated such that the values closest by magnitude are added together, there is less chance that a relatively small value gets ignored, i.e., does not affect the result. A vectorized method that establishes the ordering \((F.2)\) was designed as an extension of the AVX512F-vectorized [9] Batcher’s bitonic sort [4] of a double precision vector to the pair \((e^{(a)}, f^{(a)})\) of vectors with the comparison operator of the \(i\)th and the \(j\)th lane given by \((F.2)\).

In each sorting stage \(\ell\), \(1 \leq \ell \leq 6\), the code from Algorithm F.2 is executed, where \(e^{(a)}_{[1]} = e^{(a)}_{[1]}\) and \(f^{(a)}_{[1]} = f^{(a)}_{[1]}\), respectively; \(p_{[\ell]}\) is a permutation vector, and \(b_{[\ell]}\) is a bitmask, as given in Table F.1 and defined in [9]. The vectors \(e_{[\ell]}\) and \(f_{[\ell]}\), and their permutations \(e'_{[\ell]}\) and \(f'_{[\ell]}\), are compared according to \((F.2)\), resulting in a bitmask \(m'_{[\ell]}\), using which the lane-wise minimums \((e'_{[\ell]} \wedge f'_{[\ell]})\) and the maximums \((e'_{[\ell]} \vee f'_{[\ell]})\) and \(f'_{[\ell]}\) are extracted. Then, the inter-lane exchanges of values in \(e_{[\ell]}\) as well as in \(f_{[\ell]}\), according to \(b_{[\ell]}\), form the new sequences \(e_{[\ell+1]}\) and \(f_{[\ell+1]}\) for the next stage.

Algorithm F.2 The AVX512F compare-and-exchange operation from [9, Appendix B] extended to a pair \((e_{[\ell]}, f_{[\ell]})\) of double precision vectors according to \((F.2)\).

1: \textbf{for} \(\ell = 1 \text{ to } 6\) \textbf{do} // with \(s = 8\) double precision lanes per vector
2: \quad \(e'_{[\ell]} = \text{permutexvar}(p_{[\ell]}, e_{[\ell]});\) // permute \(e_{[\ell]}\) w.r.t. \(p_{[\ell]}\)
3: \quad \(f'_{[\ell]} = \text{permutexvar}(p_{[\ell]}, f_{[\ell]});\) // permute \(f_{[\ell]}\) w.r.t. \(p_{[\ell]}\)
4: \quad \(m'_{[\ell]} = \_\_\text{mm512}_-\text{cmpeq}_-\text{pd}_-\text{mask}(e_{[\ell]}, e'_{[\ell]});\) // where is \(e_{[\ell]} = e'_{[\ell]}\) \ldots
5: \quad \(m'_{[\ell]} = \_\_\text{mm512}_-\text{cmpx}_-\text{pd}_-\text{mask}(m'_{[\ell]}, f_{[\ell]}, f'_{[\ell]});\) // \ldots and \(f_{[\ell]} \leq f'_{[\ell]}\)
6: \quad \(m'_{[\ell]} = \_\_\text{mm512}_-\text{cmpx}_-\text{pd}_-\text{mask}(e_{[\ell]}, e'_{[\ell]});\) // where is \(e_{[\ell]} < e'_{[\ell]}\)
7: \quad \(m'_{[\ell]} = \_\_\text{mmmask8}_-\_\text{kor}_-\_\text{mask16}(m'_{[\ell]}, m'_{[\ell]});\) // bitmask of \((F.2)\)
8: \quad \(e_{[\ell]} = \_\text{mask}_-_\text{blend}(m'_{[\ell]}, e'_{[\ell]}, e_{[\ell]});\) // \(e\)-s of smaller values
9: \quad \(e_{[\ell]} = \_\text{mask}_-_\text{blend}(m'_{[\ell]}, e_{[\ell]}, e'_{[\ell]});\) // \(e\)-s of larger values
10: \quad \(f_{[\ell]} = \_\text{mask}_-_\text{blend}(m'_{[\ell]}, f'_{[\ell]}, f_{[\ell]});\) // \(f\)-s of smaller values
11: \quad \(f_{[\ell]} = \_\text{mask}_-_\text{blend}(m'_{[\ell]}, f_{[\ell]}, f'_{[\ell]});\) // \(f\)-s of larger values
12: \quad \(e_{[\ell+1]} = \_\text{mask}_-_\text{mov}(e_{[\ell]}, b_{[\ell]}, e_{[\ell]});\) // lane exchanges \(e_{[\ell]} \rightarrow e_{[\ell+1]}\)
13: \quad \(f_{[\ell+1]} = \_\text{mask}_-_\text{mov}(f_{[\ell]}, b_{[\ell]}, f_{[\ell]});\) // lane exchanges \(f_{[\ell]} \rightarrow f_{[\ell+1]}\)
14: \textbf{end for}

After the sorting has taken place, \((e^{(a)}, f^{(a)})\) is redefined as \((e_{[7]}, f_{[7]})\), since the represented values have not changed, but are now in a possibly different order. In single precision, and generally with a different power-of-two number of lanes \(s\), Algorithm F.2 and Table F.1 have to be reimplemented for the new sorting network.

Sorting \(x_i\). In Algorithm F.3, after \(x_i\) had been loaded, its absolute values could have been taken and sorted as a double precision vector by the procedure from [9, Appendix B], before forming \(|x_i|\). This way the post-iteration sorting could have been rendered redundant, at the expense of more work in the main loop.
The values of the permutation index vector \( p_\ell \) (from the highest to the lowest lane, i.e., in the order of the arguments of \_mm512_set_epi64) and of the bitmask \( b_\ell \) for each stage \( \ell \) of the bitonic sort from Algorithm F.2, as defined in [9].

| \( \ell \) | \( p_\ell \) (high-to-low) | \( b_\ell \) | \( \ell \) | \( p_\ell \) (high-to-low) | \( b_\ell \) |
|---|---|---|---|---|---|
| 1 | [6, 7, 4, 5, 2, 3, 0, 1] | (AA)\(_{16}\) | 4 | [0, 1, 2, 3, 4, 5, 6, 7] | (F0)\(_{16}\) |
| 2 | [4, 5, 6, 7, 0, 1, 2, 3] | (CC)\(_{16}\) | 5 | [5, 4, 7, 6, 1, 0, 3, 2] | (CC)\(_{16}\) |
| 3 | [6, 7, 4, 5, 2, 3, 0, 1] | (AA)\(_{16}\) | 6 | [6, 7, 4, 5, 2, 3, 0, 1] | (AA)\(_{16}\) |

**F.3.2. Vectorized pairwise reduction of the final partial sums.** For reproducibility, the summation (F.1) should always be performed in the same fashion. Let \((e_1^0, f_1^0) = (e_i, f_i)\), and define

\[
(e_i^{[j]}, f_i^{[j]}) = (e_{2i-1}^{[j-1]}, f_{2i-1}^{[j-1]}) + (e_{2i}^{[j-1]}, f_{2i}^{[j-1]})
\]

for \(1 \leq j \leq \lg s\) and \(1 \leq i \leq s/2^j\). Then, \((e, f) = (e_{[\lg s]}, f_{[\lg s]})\).

Assuming that \((e^{(s)}, f^{(s)})\) have been sorted as in (F.2), for all \(j \geq 0\) and \(i\), where \(1 \leq i \leq s/2^j\), it holds \((e_i^{[j]}, f_i^{[j]}) \leq (e_{i+1}^{[j]}, f_{i+1}^{[j]})\), what can be proven by induction on \(j\). For \(j = 0\), the statement is a direct consequence of the sorting, and for \(j > 0\), it follows from (F.3). Consequently, \(e_i^{[j]} \leq e_{i+1}^{[j]}\) for all \(j\) and odd \(i\), what simplifies the addition of the adjacent pairs from (F.3) to

\[
(e_i^{[j]} + f_i^{[j]} + e_{i+1}^{[j]} + f_{i+1}^{[j]}) = (e_i^{[j]}, f_i^{[j]}) + (e_{i+1}^{[j]}, f_{i+1}^{[j]})
\]

followed by the normalization of the result \((e_i^{[j]}, f_i^{[j]})\). The reduction (F.1) with this addition operator is vectorized as follows.

Define \(c^{[0]} = c^{(s)}\), where \(c \in \{e, f\}\). Let \(m^{[0]} = (01010101)_2\) be a lane bit-mask with every other bit set, \(\tilde{m}^{[0]} = (10101010)_2\) its bitwise complement, and \(m^{[j]}\) and \(\tilde{m}^{[j]}\) bit-masks with the lowest \(s/2^j\) bits taken from \(m^{[0]}\) and \(\tilde{m}^{[0]}\), respectively, while the higher bits are zero. Extend the masks analogously if \(s = 16\).

For each \(j \geq 1\), extract the odd-indexed and the even-indexed parts of \(e^{[j-1]}\), as in the right hand side of (F.3), into the \(s/2^j\) contiguous lower lanes of \(a^{[j-1]}_e\) and \(b^{[j-1]}_e\), respectively, where

\[
a^{[j-1]}_e = \text{maskz\_compress}(m^{[j-1]}, c^{[j-1]}) = (0, \ldots, 0, (e_i^{[j-1]}i),
\]

\[
b^{[j-1]}_e = \text{maskz\_compress}(\tilde{m}^{[j-1]}, c^{[j-1]}) = (0, \ldots, 0, (e_i^{[j-1]}i),
\]

and \(c \in \{e, f\}\). Form the non-normalized \(e^{[j]}\) as \(b^{[j-1]}_e\) and \(f^{[j]}\), from (F.4), as

\[
f_{\text{madd}}(\text{scalef}(1, \max(\text{sub}(e^{[j-1]}_e, b^{[j-1]}_e, -\infty)), a^{[j-1]}_e, b^{[j-1]}_f).
\]

Normalize the result as \(f^{[j]} = F(f^{[j]})\) and \(e^{[j]} = \text{add}(e^{[j]} b^{[j]} f^{[j]}).

After \(\lg s\) reduction stages the lowest lanes of \(e^{[\lg s]}\) and \(f^{[\lg s]}\) contain \(e\) and \(f\), respectively. Finally, \(||x||_F\) is computed from \(||x||_F^2\) using scalar operations, as described in Appendix F.1. The Frobenius-norm method is summarized in Algorithm F.3.

All operations from Algorithm F.3, except at line 24 and for the main iteration control, are branch-, division-, and square-root-free, and of similar complexity.
Algorithm F.3 Vectorization of the Frobenius norm computation of a one-dimensional double precision array, assuming $s = 8$ lanes per vector and the AVX-512 instructions.

**Input:** $x$ is a properly aligned double precision array with $m$ elements, all finite.

**Output:** a finite approximation of $\|x\|_F$ is computed, represented as $(e'', f'')$.

1. Let $r_0 = (e(0), f(0)) = (-\infty, 1)$ be a pair of vectors representing $s$ zeros;
2. For $i = 0$ to $(m = m/s) - 1$ do
   // $r_{i+1} = fma(\|x_i\|, |x_i|, r_i)$ as in Appendix F.2
   3. $x_i = [x_i \ x_{i+1} \ \cdots x_{i+s-1}]^T$, where $is + 1 \leq i \leq (i + 1)s$; // $i$ is one-based
   4. $e(i)_{\text{sb}} = _\text{mm512_and_epi64}(_\text{mm512_cvtpd_epi64}(e(i), 1))$; // AVX512DQ
   5. $e(i) = cvtepi64(e(i));$ $e(i) = \text{sub}(e(i), e(i));$ $f(i) = \text{scale}(f(i), e(i));$
   6. $r_i = (e(i), f(i));$ // representation of $r_i$ where $e(i)$ has even or infinite values
   7. $x_i = (E(x_i), F(x_i)) = (e_i, f_i);$
   8. $e_i = \text{scale}(e_i, 1);$ $e(i+1) = e_{\max} = \max(e_i, e(i));$ // $\leq$ the exponents of $r_{i+1}$
   9. $e_i = \text{sub}(e_i, e_{\max}), -\infty;$ $e(i)_{\text{sb}} = \max(\text{sub}(e(i), e_{\max}), -\infty);$ // differences
   10. $e_{i+1} = \text{scale}(e_{i+1}, 1);$ $f_{i+1} = \text{scale}(f_i, e_{i+1});$ $f(i+1) = \text{scale}(f(i), e(i+1));$
   11. $f(i+1) = \text{fmadd}(f_i, f_i, f(i));$ $f(i+1) = F(f(i+1));$ // $f_i^2 + f(i)$ and normalization
   12. $e(i+1) = \text{add}(e(i+1), E(f(i+1)));$ $r_{i+1} = (e(i+1), f(i+1));$ // exps. normalized
   13. end for
   14. Sort $r_0$ according to (F.2) from Appendix F.3.1 by Algorithm F.2;
   15. Let $c[0] = c[n]$, where $c \in \{e, f\}$, and $m[0] = (01010101)_2, \ m[0] = (10101010)_2$;
   16. For $j = 1$ to $\lg s$ do
      // reduction (F.1) of $r_0$ to $(e, f)$ as in Appendix F.3.2
      17. $a[j-1] = \text{maskz_compress}(m[j-1], c[j-1]);$ // $m[j-1]$-indexed parts of $c \in \{e, f\}
      18. b[j-1] = \text{maskz_compress}(m[j-1], c[j-1]);$ // $m[j-1]$-indexed parts of $c \in \{e, f\}
      19. f[j] = \text{fmadd}(\text{scale}(1, \max(a[j-1], b[j-1]), -\infty), a[j-1], b[j-1]);$ // (F.4)
      20. $e[j] = E(f[j]);$ $e[j] = \text{scale}(e[j], e[j]);$ $e[j] = \text{add}(e[j], E(f[j]));$ // normalization
      21. Shift the bitmasks $m[j-1]$ and $m[j-1]$ $s/2^j$ bits to the right to get $m[0]$ and $m[0]$;
      22. end for
   23. // Output: $(e[3], f[3])$
   24. Let $(e, f) = (e[5], f[5])$ and compute $(e'', f'')$ as in Appendix F.1; // scalars
   25. return $(e'', f'')$; // also $(e, f)$ and a double precision value $fl(2''', f''')$ for reference

Main loop at lines 3–14 requires at minimum 20 vector arithmetic operations, compared to only one \texttt{fmadd} in the standard dot-product computation, but less than fivefold slowdown is achieved for $m = m = 2^{30}$ as shown in Figures F.3 and F.4. For $s$ fixed, the other loops can be unrolled, and together with the remaining parts of Algorithm F.3 require a constant number of operations, independent of $m$.

**F.3.3. Sequential reduction of the final partial sums.** Instead of vectorizing it, the horizontal reduction could have been performed sequentially, from the first to the last partial sum, assuming $(e_i, f_i) \leq (e_{i+1}, f_{i+1})$, similarly to (F.4) as

\[(e_i', f_i') = (e_i, f_i) + (e_i, f_i) = (e_i, f_i) + (e_i, f_i), 1 \leq i < s.\]

If $(e_i, f_i) > (e_{i+1}, f_{i+1})$, these two pairs are swapped. Having computed (F.5) by scalar operations, $(e_{i+1}, f_{i+1})$ is replaced in-place by $(e_i', f_i')$ and $i$ is incremented.

This reduction variant was used in the numerical testing from Appendix F.4, since its order of summation had a chance of being more accurate than that of the vectorized variant from Appendix F.3.2, with the final partial sums sorted as in Appendix F.3.1. Note that the sequential reduction does not presuppose, but can benefit from this.
initial sorting. Finally, the reduction’s result is \((e'_s, f'_s) = (e, f)\).

**F.4. Numerical testing.** Section 5 gives a description of the test environment. The reference BLAS routine \(\text{DNRM2}\) was renamed, built for comparison, and verified by disassembling that it used, when possible, the fused multiply-add instructions.

For each exponent \(\xi \in \{0, 1008\}\), 65 double precision test vectors \(x_\tau, 0 \leq \tau \leq 64\), were generated. For a fixed \(\xi\), every vector had its elements in the range \([0, 2^{\xi}]\). Each element was pseudorandomly generated by the CPU’s \(\text{RDRAND}\) facility as a 64-bit quantity, with the same probability of each bit being a zero or a one. Any candidate with the magnitude falling out of the given range was discarded and generated anew. Finally, the elements were replaced by their absolute values to aid the future sorting.

An accurate approximation of \(\|x_\tau\|_F\), denoted by \(\|x'_\tau\|_F\), was obtained by computing \(x'_\tau^T x'_\tau\) in quadruple (128-bit) precision and rounding the result’s square root to double precision, where \(x'_\tau\) has the same elements as \(x_\tau\), but sorted non-decreasingly, to improve accuracy, and converted to quadruple precision. Then,

\[
\max \left\{ \frac{|e^{(o)} - \|x'_\tau\|_F|}{\|x'_\tau\|_F}, 0 \right\} \leq \infty
\]

is the relative error of an approximation \(e^{(o)} \approx \|x_\tau\|_F\) computed by a procedure \(\phi\). The relative errors are shown in Figures F.1 and F.2 for \(\xi = 0\) and \(\xi = 1008\), respectively, where BLAS \(\text{DNRM2}\) refers to the reference Fortran implementation.

![Fig. F.1. Relative errors of \(\|x_\tau\|_F\) computed by several procedures, for \(\xi = 0\).](image)

Figures F.1 and F.2 suggest that the reference BLAS routine is consistently about four orders of magnitude less accurate than the MKL’s \(\text{DNRM2}\). Also, the MKL’s routine is significantly more accurate than computing in extended precision, what indicates that the MKL does not use the reference algorithm in its published form.

Therefore, the MKL’s routine is the most accurate one, but bound to overflow at least when the actual result is not finitely representable in double precision, while Algorithm F.3 (or the “proposed \(\sqrt{x^T x}\)”) and the computation of \(\sqrt{x^T x}\) in extended precision cannot overflow for any reasonably-sized \(x\). However, from Figures F.3 and F.4 it can be concluded that Algorithm F.3 is somewhat slower than the extended dot-product, which in turn is significantly slower than the MKL’s \(\text{DNRM2}\). On the
MKL-targeted platforms, its DNRM2 is thus the method of choice for the Jacobi-type SVD, even if that entails a sporadic downscaling of the input data.

Vectorized quadruple precision. An accurate alternative way of computing $\sqrt{x^T x}$ without overflow is to use the SLEEF’s vectorized quad-precision math library\(^{14}\) by modifying Algorithm F.1 to convert the loaded double precision vectors to quadruple precision ones and then call the vectorized quad-fma on them. Such an approach gives the results virtually indistinguishable from a non-vectorized quadruple dot-product, but in the testing environment, for $\xi = 0$, it is $\approx 164 \times$ and $\approx 1.69 \times$ slower than the MKL’s DDOT and the BLAS DNRM2, respectively, maybe because SLEEF does not yet provide the inlineable versions of its functions for the Intel’s compilers.

**F.5. Conclusion.** When the MKL’s DNRM2 and the hardware-supported extended precision are not available, and the SLEEF’s vectorized quad-precision library

\(^{14}\)https://sleef.org/quad.xhtml
happens to be too slow (or unavailable), the proposed $\sqrt{x^T x}$ method can be employed as a fail-safe option, either for computing $\|x\|_F$ on its own, or to be called on the second attempt, after a faster but at least comparably accurate method overflows.

It is also easy to convert in a portable way the result of other methods in double, extended, or quadruple precision to the $(e,f)$ representation, which thus can be used for encoding the Frobenius norms of the columns of the iteration matrix in the Jacobi SVD, regardless of the chosen method of computing them. The remaining parts of the SVD algorithm do not rely on a higher precision of computation or of data, so a possible loss of it due to the rounding of $f$ to double precision is acceptable.

As $m$ becomes larger, and/or with an unfavorable distribution of the elements' magnitudes, such that the huge elements precede the small ones, at a certain point the partial sums in the proposed algorithm can become too large to be (significantly) affected by the subsequent updates, resulting in a too small final norm and hence a big relative error. It is worth exploring if, e.g., three vector accumulators instead of one—for the small, the medium, and the large magnitudes, in the spirit of the Blue’s algorithm [7]—could improve the relative accuracy. However, such an approach will inevitably reduce the amount of the vector parallelism, since the present instructions have to be trebled and masked to affect the appropriate accumulator only.

Appendix G. Addenda for section 5. Section 5 from the main paper is here expanded.

G.1. Addendum for subsection 5.2. For the eigenvector matrix $\Phi$ of a Hermitian matrix of order two holds $|\det \Phi| = c^2 + |s|^2 = 1$, where $c = \cos \varphi$ and $s = e^{i\alpha} \sin \varphi$. Therefore, $\delta^2 = |\det(\Phi)| - 1$ determines how much the real cosine and the real or complex sine, computed by a routine designated by $x$, depart from the ones that would make $\Phi$ unitary. For a batch $\tau$, let $\delta^2_{\tau, i} = \max_i \delta^2_{\tau, i}, 1 \leq i \leq 2^{28}$. Then, Figure G.1 shows that, in the real case, the batched and the LAPACK-based EVDs do not differ much in this measure of accuracy, but in the complex case the LAPACK-based ones may be several orders of magnitude less accurate. Together with Figures G.2 and G.3, described below, this further explains Figure 5.2.

If a closer look is taken at the matrix $A_{\max}$ that caused the greatest determinant’s residual with CLAEV2, and at the computed outputs $\Lambda_{\max}$ and $U_{\max}$ (all in single
VECTORIZATION OF THE JACOBI SVD

Fig. G.1. Per-batch ratios $\delta_s^\tau/\delta_s^\tau$, $\delta_d^\tau/\delta_d^\tau$, $\delta_c^\tau/\delta_c^\tau$, and $\delta_z^\tau/\delta_z^\tau$ of the determinants' residuals.

precision, but printed out with 21 digits after the decimal point for reproducibility),

$$A^\delta_{\text{max}} = \begin{bmatrix} A & B \\ B & C \end{bmatrix}, \quad \Lambda^\delta_{\text{max}} = \begin{bmatrix} \text{RT1} & 0 \\ 0 & \text{RT2} \end{bmatrix}, \quad U^\delta_{\text{max}} = \begin{bmatrix} \text{CS1} & -\text{SN1} \\ \text{SN1} & \text{CS1} \end{bmatrix},$$

where

$$A = -1.428758589291419051209 \cdot 10^{-38}, \quad B = 1.401298464324817070924 \cdot 10^{-45} - 1.4012984643248170924 \cdot 10^{-45}, \quad C = -1.429318548157763248111 \cdot 10^{-38},$$

and

$$\text{RT1} = \lambda_1 = -1.429318548157763248111 \cdot 10^{-38}, \quad \text{RT2} = \lambda_2 = -1.428758589291419051209 \cdot 10^{-38}; \quad \text{CS1} = \cos \varphi = -2.502502466086298227310 \cdot 10^{-4},$$

$$\text{SN1} = e^{i\alpha} \sin \varphi = 1 + i$$

a disastrous failure to represent $|\mathcal{B}|$ in the subnormal range by any value other than $|\Re \mathcal{B}| = |\Im \mathcal{B}|$, as explained in Remark 2.3, becomes the obvious reason for the loss of accuracy. But why was $\sin \varphi$ computed as unity? The answer is that the eigenvalues had to be swapped to make $|\lambda_1| \geq |\lambda_2|$. Since CLAEV2 calls SLAEV2 with $A$, $|\mathcal{B}|$, and $C$, the latter routine performed the exchange $\cos \varphi \leftarrow -\sin \varphi$, $\sin \varphi \leftarrow \cos \varphi$, as if the columns of its, real $U$ were swapped. Then CLAEV2 multiplied $\mathcal{B}/|\mathcal{B}| = e^{i\alpha} = 1 + i$ by the new value of $\sin \varphi$, i.e., by unity. Note that if $A$ and $C$ changed roles, $e^{i\alpha}$ would have remained $1 + i$, but it would have been multiplied by $\sin \varphi \approx 2.502502466 \cdot 10^{-4}$. Even tough $\text{CS1} = -1$ then, $|| \det U ||^{-1}$ would have been small. Algorithm 2.2 ensures $|\sin \varphi| \leq 1/\sqrt{2}$, up to the rounding errors, so a bogus $e^{i\alpha}$ is always scaled down.

As already known for the LAPACK’s EVD, the smaller eigenvalue by magnitude might not be computed with any relative accuracy, and this is also the case with the batched EVD. The normwise residuals $\lambda_F^\tau = \max_i \| \mathcal{f}(A_{\tau,i}) - A_{\tau,i} \|_F/\|A_{\tau,i} \|_F$, however, seem stable, and their ratios are shown in Figure G.2. The values below unity
indicate when the batched EVD was less accurate, while those above unity demonstrate that in a majority of batches the LAPACK-based EVD was more inaccurate.

\[
\tau (A_{\tau,i} = (A_{\tau,i}), 1 \leq i \leq 2^{28}; A_{\tau,i} \text{ random } 2 \times 2 \text{ Hermitian})
\]

Fig. G.2. Per-batch ratios \( \lambda^{e,\tau}_{s}/\lambda^{e,\tau}_{s} \), \( \lambda^{e,\tau}_{d}/\lambda^{e,\tau}_{d} \), \( \lambda^{e,\tau}_{c}/\lambda^{e,\tau}_{c} \), and \( \lambda^{e,\tau}_{z}/\lambda^{e,\tau}_{z} \) of the eigenvalues’ normwise residuals.

A similar conclusion is valid when looking only at the eigenvalue larger by magnitude, \( \lambda_{\max} \), that is always computed relatively accurate, up to a modest constant times \( \varepsilon \). The ratios of the max-residuals \( \lambda^{e,\tau}_{\max} = \max_{i} |fl(\lambda_{\max};\tau,i) - \lambda_{\max;\tau,i}|/|\lambda_{\max;\tau,i}| \) are shown in Figure G.3.

\[
\tau (A_{\tau,i} = (A_{\tau,i}), 1 \leq i \leq 2^{28}; A_{\tau,i} \text{ random } 2 \times 2 \text{ Hermitian})
\]

Fig. G.3. Per-batch ratios \( \lambda^{e,\tau}_{\max}/\lambda^{e,\tau}_{\max} \), \( \lambda^{e,\tau}_{\max}/\lambda^{e,\tau}_{\max} \), \( \lambda^{e,\tau}_{\max}/\lambda^{e,\tau}_{\max} \), and \( \lambda^{e,\tau}_{\max}/\lambda^{e,\tau}_{\max} \) of the eigenvalues’ max-residuals.

G.2. Addendum for subsection 5.3. Figures G.4 and G.5 show the number of sweeps until convergence in the real and the complex case on \( \Xi_{1}^{F} \) and \( \Xi_{2}^{F} \), respectively, while Figures G.6 to G.8 correspond to Figures 5.3 to 5.5, respectively, in the real case on \( \Xi_{1}^{R} \).

Appendix H. Vectorization of the real Gram–Schmidt orthogonalization. In Algorithm H.1 the real Gram–Schmidt orthogonalization from (3.8) is
Fig. G.4. Number of sweeps until convergence for the LAPACK’s \texttt{xGESVJ} and the proposed routines (with the MM parallel quasi-cyclic pivot strategy), on $\Xi_1^F$.

Fig. G.5. Number of sweeps until convergence for the LAPACK’s \texttt{xGESVJ} and the proposed routines (with the ME parallel cyclic pivot strategy), on $\Xi_2^F$.

vectorized.

REFERENCES

[1] A. Abdelfattah, H. Anzt, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. Yarkhan, Linear algebra software for large-scale accelerated multicore computing, Acta Numer., 25 (2016), pp. 1–160, https://doi.org/10.1017/S0962492916000015.

[2] E. Anderson, Algorithm 978: Safe scaling in the Level 1 BLAS, ACM Trans. Math. Software, 44 (2017), pp. 1–28 (art. no. 12), https://doi.org/10.1145/3061665.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK Users’ Guide, Software, Environments and Tools, SIAM, Philadelphia, PA, USA, 3rd ed., 1999, https://doi.org/10.1137/1.9780898719604.

[4] K. E. Batcher, Sorting networks and their applications, in Proceedings of the April 30–May
Fig. G.6. Run-time ratios of DGESVJ and dvjsvd with MM on $\Xi^3_1$.

Fig. G.7. Breakdown of the run-time of dvjsvd with MM on $\Xi^3_1$ with 64 threads.

Fig. G.8. Ratios of the relative error measures for the SVDs on $\Xi^3_1$. 
Algorithm H.1 \texttt{dgsscl}: a vectorized real Gram–Schmidt orthogonalization.

\textbf{Input:} $g_q, 0 < \|g_q\|_F = (e_q, f_q); g_p, 0 < \|g_p\|_F = (e_p, f_p); d_{21}$.

\textbf{Output:} $\bar{f}(g_q)$ from (3.8).

\begin{algorithmic}
\STATE $e_q = \text{set1}(e_q); -e_p = \text{set1}(-e_p); -e_q = \text{set1}(-e_q); -\psi = \text{set1}(-d_{21}(f_q/f_p));$
\FOR {$i = 0$ to $m - 1$} 
\STATE $x = \text{load}(g_q + i); y = \text{load}(g_q + i)$; // sequentially
\STATE $x = \text{scalef}(x, -e_q); y = \text{scalef}(y, -e_q); /...$ faster than full division
\STATE $y = \text{scalef}(\text{fmadd}(-\psi, x, y), e_q); \text{store}(g_q + i, y);$ \quad (3.8)
\ENDFOR
\end{algorithmic}
[23] ISO/IEC JTC1/SC22/WG14, ISO/IEC 9899:2018(en) Information technology — Programming languages — C, ISO, 4th ed., 2018. International standard.

[24] C. G. J. Jacobi, Über ein leiches Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen, J. Reine Angew. Math., 30 (1846), pp. 51–94, https://doi.org/10.1515/crll.1846.30.51.

[25] C.-P. Jeannerod and S. M. Rump, Improved error bounds for inner products in floating-point arithmetic, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 338–344, https://doi.org/10.1137/100795159.

[26] E. G. Kogbetliantz, Solution of linear equations by diagonalization of coefficients matrix, Quart. Appl. Math., 13 (1955), pp. 123–132, https://doi.org/10.1090/qam/88795.

[27] M. Mantharam and P. J. Eberlein, Block recursive algorithm to generate Jacobi-sets, Parallel Comput., 19 (1993), pp. 481–496, https://doi.org/10.1016/0167-8191(93)90001-2.

[28] W. F. Mascarenhas, On the convergence of the Jacobi method for arbitrary orderings, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 1197–1209, https://doi.org/10.1137/S0895479894272983.

[29] V. Novaković, A hierarchically blocked Jacobi SVD algorithm for single and multiple graphics processing units, SIAM J. Sci. Comput., 37 (2015), pp. C1–C30, https://doi.org/10.1137/140952429.

[30] V. Novaković, Batched computation of the singular value decompositions of order two by the AVX-512 vectorization, Parallel Process. Lett., 30 (2020), pp. 1–23 (art. no. 2050015), https://doi.org/10.1142/S0129626420500152.

[31] V. Novaković and S. Singer, A GPU-based hyperbolic SVD algorithm, BIT, 51 (2011), pp. 1009–1030, https://doi.org/10.1007/s10543-011-0333-5.

[32] V. Novaković and S. Singer, Implicit Hari–Zimmermann algorithm for the generalized SVD on the GPUs, Int. J. High Perform. Comput. Appl., 35 (2021), pp. 170–205, https://doi.org/10.1177/1094342020972772.

[33] V. Novaković and S. Singer, A Kogbetliantz-type algorithm for the hyperbolic SVD, Numer. Algorithms, (2021), https://doi.org/10.1007/s11075-021-01197-4. In press.

[34] N. Shibata and F. Petrogalli, SLEEF: A portable vectorized library of C standard mathematical functions, IEEE Trans. Parallel Distrib. Syst., 31 (2020), pp. 1316–1327, https://doi.org/10.1109/TPDS.2019.2966333.

[35] S. Singer, E. Di Napoli, V. Novaković, and G. Čaklović, The LAPW method with eigendecomposition based on the Hari–Zimmermann generalized hyperbolic SVD, SIAM J. Sci. Comput., 42 (2020), pp. C265–C293, https://doi.org/10.1137/19M1277813.

[36] F. G. V. Zee and T. M. Smith, Implementing high-performance complex matrix multiplication via the 3m and 4m methods, ACM Trans. Math. Software, 44 (2017), pp. 1–36 (art. no. 7), https://doi.org/10.1145/3086466.