# MIXED PRECISION ORTHOGONALIZATION-FREE PROJECTION METHODS FOR EIGENVALUE AND SINGULAR VALUE PROBLEMS

TIANSHI XU[*], ZECHEN ZHANG[†], JIE CHEN[‡], YOUSEF SAAD[†], AND YUANZHE XI[*]

**Abstract.** Mixed-precision arithmetic offers significant computational advantages for large-scale matrix computation tasks, yet preserving accuracy and stability in eigenvalue problems and the singular value decomposition (SVD) remains challenging. This paper introduces an approach that eliminates orthogonalization requirements in traditional Rayleigh-Ritz projection methods. The proposed method employs non-orthogonal bases computed at reduced precision, resulting in bases computed without inner-products. A primary focus is on maintaining the linear independence of the basis vectors. Through extensive evaluation with both synthetic test cases and real-world applications, we demonstrate that the proposed approach achieves the desired accuracy while fully taking advantage of mixed-precision arithmetic.

**Key words.** Mixed precision, singular value decomposition, eigenvalue problem, Rayleigh-Ritz, orthogonalization-free, GPU

**AMS subject classifications.** 15A23, 65F25, 65Y05, 68W10

## 1. Introduction.

Recent research on themes related to high-performance computing indicates a strong surge of interest in low-precision and mixed-precision arithmetic. There are compelling performance incentives to work with lower precision. Two related benefits of working in reduced precision are the *lower energy consumption* and the *lighter storage requirement*, which also leads to less communication [3]. Using mixed-precision arithmetic has long been an effective approach in many areas. Scientists and engineers in scientific computing have traditionally leaned toward double-precision arithmetic by default, but this is now being questioned as more studies are being undertaken, and a better understanding is emerging on the impact of low precision on common computations, see, e.g., [2, 3, 12].

Mixed-precision arithmetic provides notable computational benefits, yet maintaining accuracy when using reduced precision remains challenging. In this paper, we analyze the impact of mixed-precision arithmetic on eigenvalue and singular value computations. Eigenvalue decomposition and Singular Value Decompositions (SVD) constitute fundamental numerical linear algebra kernels, which are widely used in diverse scientific and data science applications [7, 11, 14, 19, 34, 29, 30, 33]. For large-scale problems, these decompositions are typically replaced by partial eigenvalue or singular value problems where a few eigenvalues or singular values need to be computed along with their associated eigenvectors or singular vectors. This is often achieved via projection-type methods [24, 18, 4]. These methods rely on matrix-vector multiplications (MatVecs) and other simple linear algebra operations to build a suitable subspace and then extract eigenvalue and singular value approximations from this subspace [26]. In this paper, we first identify and discuss the specific challenges associated with the use of mixed-precision arithmetic and then propose effective strategies to address them.

In the past few years, researchers have already explored a few methods to mitigate

[*]Department of Mathematics, Emory University, Atlanta, GA, (tianshi.xu@emory.edu, yxi26@emory.edu). Research is supported by NSF DMS-2208412 and NSF DMS-2513118.

[†]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, (zhan5260@umn.edu,saad@umn.edu). Research supported by the NSF DMS-2513117.

[‡]MIT-IBM Watson AI Lab, IBM Research. (chen.future.jie@gmail.com)

*This manuscript is for review purposes only.*

accuracy loss in mixed-precision computations of eigenvalues and singular values. One popular approach relies on iterative refinement originating from Newton's method for mixed-precision refinement in standard eigenvalue computations [9, 10]. This strategy has been effectively extended to symmetric eigenvalue problems [21, 25, 31] and the SVD [22]. Another direction of research involves identifying optimal precision levels for storing matrices without compromising performance [23]. Recent studies also recommend using probabilistic error analysis to determine precision requirements throughout computational phases, emphasizing high-precision reorthonormalization for maintaining accuracy [16]. One such example is the mixed-precision, single-pass Nyström method proposed in [6], which executes computationally intensive matrix multiplication operations at lower precision.

In this paper, we focus on the Rayleigh-Ritz (RR) projection framework [26] for efficiently computing subsets of eigenvalues and singular values. The RR projection method involves two main stages: the first stage constructs a subspace basis encapsulating essential matrix information, typically an orthogonal basis derived via QR factorization; the second stage projects the original matrix onto this subspace, extracting the targeted eigenvalue or singular value approximations. Since classical QR-based orthogonalization methods can suffer from substantial orthogonality loss when performed in low precision, the effectiveness of the RR projection method will also be undermined in this case. To address this issue, we introduce a refined RR projection approach to improve both accuracy and computational performance with mixed precision arithmetic.

Our main contributions are the following:

**1**. We introduce the Orthogonalization-Free Rayleigh-Ritz (OFRR) procedure, specifically designed to address the challenges of maintaining numerical accuracy in eigenvalue and singular value computations performed with mixed-precision arithmetic. Traditional approaches, reliant on QR-based orthogonalization, often suffer significant accuracy degradation in low-precision environments. To overcome this limitation, OFRR eliminates the explicit orthogonalization step, enabling the extraction of accurate spectral information from non-orthogonal basis vectors.

**2**. We investigate the use of various approaches for generating non-orthogonal bases for the proposed OFRR procedure. Furthermore, we show that the Hessenberg process—a variant of LU factorization—outperforms Gram–Schmidt orthogonalization due to its inner-product-free feature.

**3**. To evaluate the performance and accuracy of the OFRR algorithm, we conduct extensive numerical experiments using a diverse set of matrices. These includ challenging real-world problems from the SuiteSparse Matrix Collection [8], as well as kernel matrices arising in Gaussian processes [5, 17]. The results indicate that OFRR significantly enhances approximation accuracy compared to traditional approaches that rely on orthogonalization steps. In addition, we implement OFRR on GPU architectures to assess its practical performance. The GPU-accelerated OFRR implementation highlights the algorithm's scalability and applicability in large-scale matrix computations.

The remaining sections are organized as follows. In Section 2, we use subspace iteration as an illustrative example to demonstrate the challenges posed by low-precision arithmetic in eigenvalue computations. We then introduce the Orthogonalization-Free Rayleigh-Ritz (OFRR) procedure in Section 3 and examine several strategies for gen-

*This manuscript is for review purposes only.*

erating non-orthogonal bases in Section 4. The effectiveness and accuracy of the proposed OFRR algorithm are verified through extensive numerical experiments in Section 5, and concluding remarks are drawn in Section 6.

Following `MATLAB` syntax, we use subscripts to access elements and submatrices of matrices and vectors. For a matrix $\mathbf{A}$, the notation $\mathbf{A}_{i,:}$ represents the entire $i$-th row, while $\mathbf{A}_{:,j}$ represents the entire $j$-th column and $\mathbf{A}_{i,j}$ is the entry at the $i$-th row and $j$-th column. Similarly, for a vector $\mathbf{v}$, $\mathbf{v}_i$ indicates the $i$-th entry. More generally, for integers $p \leq q$ and $r \leq s$, $\mathbf{A}_{p:q,r:s}$ denotes the submatrix of $\mathbf{A}$ consisting of rows $p$ through $q$ and columns $r$ through $s$. The colon ':' in a subscript indicates selecting all indices along that dimension. If $\mathbf{p}$ is a permutation vector, then, $\mathbf{A}_{\mathbf{p},j}$ denotes the $j$-th column of $\mathbf{A}$ with its entries permuted according to $\mathbf{p}$. Furthermore, we let $\mathbf{e}_i$ denote the $i$-th column of an identity matrix. Finally, we represent a general subspace by $\mathcal{K}$ and use $\mathcal{K}_m(\mathbf{v}, \mathbf{A})$ to denote the $m$-th Krylov subspace:

$$\mathcal{K}_m(\mathbf{v}, \mathbf{A}) := \text{span}\{\mathbf{v}, \mathbf{A}\mathbf{v}, \cdots, \mathbf{A}^{m-1}\mathbf{v}\}.$$

## 2. Challenges in Low Precision Eigenvalue Computations.

In this section, we use the subspace iteration with Rayleigh–Ritz (RR) projection as an example to identify the difficulties that contribute to the accuracy loss in low-precision eigenvalue computations.

Subspace iteration is widely used to approximate the dominant eigenpairs of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$. This algorithm, akin to a 'block' version of the power method, begins with a randomly chosen initial set of vectors $\mathbf{X}_0 \in \mathbb{R}^{n\times k}$. Each iteration applies a power of $\mathbf{A}$ to $\mathbf{X_0}$

$$\mathbf{X}_{iter} = \mathbf{A}^{iter}\mathbf{X}_0, \tag{2.1}$$

where the power *iter* is typically fixed a priori by the user or determined dynamically. To ensure numerical stability, column scaling should follow each matrix-vector multiplication to prevent overflow or underflow. In addition, the QR algorithm is typically applied to $X_{iter}$ to preserve linear independence among the vectors. See Algorithm 2.1 for a summary of this procedure.

**Algorithm 2.1** *Multiple Step Subspace Iteration*

```
1: ▷ input: $\mathbf{A} \in \mathbb{R}^{n\times n}$, $k$, $m$, and $iter$
2: ▷ output: $\mathbf{X_0}$
3: ▷ Generate a set of random vectors $\mathbf{X}_0 \in \mathbb{R}^{n\times k}$
4: for $i = 1 : m$ do
5:     ▷ Compute $\mathbf{X}_{iter} = \mathbf{A}^{iter}\mathbf{X}_0$
6:     ▷ Perform QR factorization $\mathbf{X}_{iter} = \mathbf{QR}$
7:     ▷ Set $\mathbf{X}_0 = \mathbf{Q}$
8:     ▷ Update $iter$
9: end for
```

Algorithm 2.1 generates an orthonormal basis $\mathbf{Q}$ intended to approximate the dominant invariant subspace of $\mathbf{A}$. It is worth noting that alternative approaches exist for approximating a few eigenvalues and vectors of a matrix. Most prominent among these is the family of Krylov subspace methods (e.g., the Lanczos algorithm for symmetric $\mathbf{A}$ or the Arnoldi process for non-symmetric $\mathbf{A}$). Krylov methods are usually faster for such tasks, see Section 4.1, but subspace iteration has a number of other advantages when the goal is to compute an invariant subspace, e.g., in electronic structure calculations [35].

Algorithm 2.1 is always used in conjunction with RR projection step (Algorithm 2.2). This modification involves updating $\mathbf{X}_0$ in Line 7 of Algorithm 2.1 using the output $\tilde{\mathbf{U}}$ from Algorithm 2.2. The inputs for Algorithm 2.2 are the matrix $\mathbf{A}$ and the matrix $\mathbf{Q}$, which is generated in Line 6 of Algorithm 2.1.

**Algorithm 2.2** *Rayleigh-Ritz Projection with Orthogonal Bases*

```
1: ▷ input: $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{Q} \in \mathbb{R}^{n\times k}$ with orthonormal columns
2: ▷ output: $\tilde{\mathbf{\Lambda}}$ and $\tilde{\mathbf{U}}$          {Approximate eigenvalues and Schur vectors}
3: ▷ Compute $\mathbf{B} = \mathbf{Q}^\top \mathbf{A}\mathbf{Q}$
4: ▷ Compute Schur decomposition $\mathbf{B}\mathbf{Y} = \mathbf{Y}\tilde{\mathbf{\Lambda}}$
5: ▷ Compute $\tilde{\mathbf{U}} = \mathbf{Q}\mathbf{Y}$.
```

As illustrated in Algorithms 2.1 and 2.2, subspace iteration with RR projection primarily relies on two fundamental linear algebra operations: MatVecs and vector orthogonalization. In reduced-precision environments, maintaining the orthonormality of $\mathbf{Q}$ as required by classical Rayleigh-Ritz projection presents significant challenges. To investigate the impact of orthogonality loss on eigenvalue approximations, we conducted a series of experiments using Gaussian kernel matrices defined by

$$\mathbf{A}_{ij} = f(\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/(2l^2)) + s\delta_{ij}), \tag{2.2}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ in $\mathbb{R}^d$ are the $i$-th and $j$-th data points, respectively, from a dataset $\mathbf{D} \in \mathbb{R}^{n\times d}$. Here, $f$ represents the scale parameter, $l$ represents the length scale parameter, $s$ is the variance parameter, and $\delta_{ij}$ is a Kronecker delta function that is 1 when $i = j$ and 0 otherwise. We uniformly sampled 1000 data points from a square area with side length $\sqrt{1000}$, setting $f = 1$, $l = 10$, and $s = 0.01$ to generate a test matrix $\mathbf{A}$.

We then performed subspace iteration with RR projection using $k = 40$, $m = 3$, and $iter = 2$ and the Modified Gram-Schmidt QR factorization in Line 6 of Algorithm 2.1. We examined various precision configurations: `double` (double-precision floating-point, `FP64`), `single` (single-precision floating-point, `FP32`), and `half` (half-precision floating-point, `FP16`). Note that all computations for `FP16` are done in `FP32` and the results are then truncated to `FP16`. This setting is common since it reflects the behavior of many optimized routines, such as certain `cuBLAS` functions (like `cublasDotEx`), which utilize `FP32` for internal accumulations. Precision settings for the two operations, MatVecs and QR factorization, are customized for each experiment and designated by labels such as `[MatVec Precision]-[QR Precision]`. For example, in the `single-double` configuration, MatVec operations are carried out in single precision while QR factorization is done in double precision. We always use double precision to solve the projected eigenvalue problem in Line 3 of Algorithm 2.2.

Figure 1 reports the relative errors for the 20 largest eigenvalues under various precision configurations. As anticipated, the configurations with double/single precision MatVecs achieve errors close to the machine epsilon for `FP64`/`FP32`, confirming that employing high precision in both MatVecs and QR operations effectively minimizes numerical inaccuracies. In contrast, the `half-half` configurations exhibit substantially larger errors, highlighting the inherent challenges of relying exclusively on half precision. Comparisons among the `half-double`, `half-single`, and `half-half` configurations suggest that lowering the precision of MatVec operations to half precision alone does not significantly compromise the overall accuracy, as long as the QR factorization is performed in higher precision. For example, the approximation accuracy of the `half-double` configuration surpasses that of full half precision and the error is smaller than $10^{-4}$. A similar trend holds for the `half-single` configuration.

Figure 1 shows that even when MatVecs are carried out in low precision, the dominant spectral subspace can still be reconstructed accurately once the resulting vectors are post-processed. Because high-precision MatVecs (or the high-precision storage they require) impose heavy penalties in memory traffic and run-time, large-scale solvers already lean toward reduced precision for these operations. The evidence in Figure 1 therefore motivates embedding low- or mixed-precision arithmetic not only in the MatVecs but throughout the basis-generation and projection stages. Our goal is to capture the speed and memory advantages of low precision while reserving full precision for the small, projected problem so that the eigenvalue approximation accuracy is not compromised much. The Orthogonalization-Free Rayleigh–Ritz Projection framework, introduced next, is built precisely for this purpose.

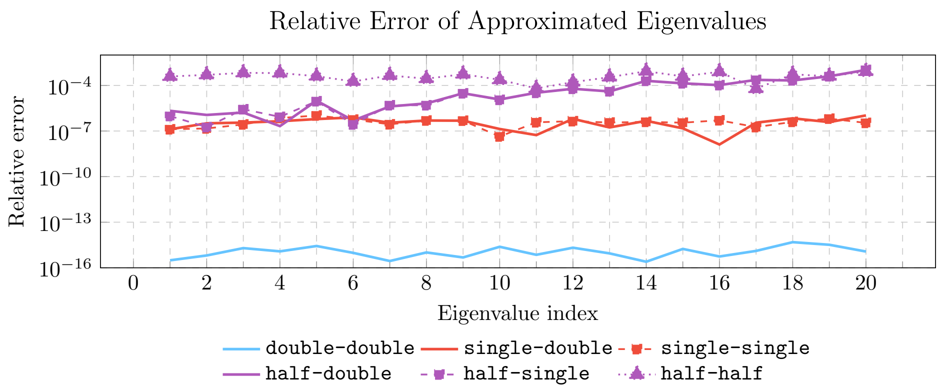

FIGURE 1. *Relative error plot of subspace iteration with Rayleigh-Ritz projection under different precision options. The test matrix is a Gaussian kernel matrix of size* $1000 \times 1000$*. A concise naming convention is used to denote different options:* `[MatVec Precision]-[QR Precision]`.

**3. Orthogonalization-Free Rayleigh-Ritz Projection.** In this section, we introduce a generalization of the Rayleigh-Ritz projection designed specifically for half-precision and lower. This variant, which does not require an orthogonal input basis, is referred to as the Orthogonalization-Free Rayleigh-Ritz (OFRR) projection method.

**3.1. OFRR for Eigenvalue Problems.** We first consider the eigenvalue problem

$$\mathbf{A}\mathbf{u} = \lambda \mathbf{u}, \tag{3.1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$. Given a $k$-dimensional subspace $\mathcal{K}$, the orthogonal projection method seeks approximate eigenpairs $\tilde{\lambda} \in \mathbb{C}$, $\tilde{\mathbf{u}} \in \mathcal{K}$ of $\mathbf{A}$ such that the following Galerkin condition is satisfied:

$$\mathbf{A}\tilde{\mathbf{u}} - \lambda\tilde{\mathbf{u}} \perp \mathcal{K}, \tag{3.2}$$

or, equivalently,

$$\langle \mathbf{A}\tilde{\mathbf{u}} - \tilde{\lambda}\tilde{\mathbf{u}}, \mathbf{u}\rangle = 0, \qquad \forall \mathbf{u} \in \mathcal{K}. \tag{3.3}$$

The standard Rayleigh-Ritz process discussed in Algorithm 2.2 assumes that an orthonormal basis of $\mathcal{K}$ is available. Now, assume that we only have a linearly independent basis $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_k]$ of $\mathcal{K}$ (which might not be orthogonal). Then the Galerkin condition leads to the following equations:

$$\langle \mathbf{A}\tilde{\mathbf{u}} - \tilde{\lambda}\tilde{\mathbf{u}}, \mathbf{u}_i \rangle = 0, \qquad i = 1, \cdots, k.$$

Since the approximate solution $\tilde{\mathbf{u}}$ is sought within the subspace $\mathcal{K}$, it can be represented as $\mathbf{U}\mathbf{y}$, with $\mathbf{y}$ being a unique vector in $\mathbb{C}^k$. Accordingly, we can transform (3.3) into a set of equations involving $\tilde{\lambda}$ and $\mathbf{y}$:

$$\langle \mathbf{A}\mathbf{U}\mathbf{y} - \tilde{\lambda}\mathbf{U}\mathbf{y}, \mathbf{u}_i \rangle = 0, \qquad i = 1, \cdots, k.$$

which is equivalent to:

$$\mathbf{U}^*\mathbf{A}\mathbf{U}\mathbf{y} = \tilde{\lambda}\mathbf{U}^*\mathbf{U}\mathbf{y}. \tag{3.4}$$

This approach, detailed in Algorithm 3.1, shifts from the traditional eigenvalue problem to a generalized one when $\mathbf{U}^*\mathbf{U} \neq \mathbf{I}$. The effectiveness of this method is closely linked to the condition number of $\mathbf{U}^*\mathbf{U}$. We will explore methods for constructing well-conditioned non-orthogonal bases in Section 4.

**Algorithm 3.1** *Orthogonalization-Free Rayleigh-Ritz Projection*

```
1: ▷ input: $\mathbf{A}$, $\mathbf{U} \in \mathbb{C}^{n\times k}$
2: ▷ output: $\tilde{\mathbf{\Lambda}}$ and $\tilde{\mathbf{U}}$
3: ▷ Compute $\mathbf{B} = \mathbf{U}^*\mathbf{A}\mathbf{U}$
4: ▷ Compute $\mathbf{M} = \mathbf{U}^*\mathbf{U}$
5: ▷ Compute eigendecomposition $\mathbf{B}\mathbf{Y} = \mathbf{M}\mathbf{Y}\tilde{\mathbf{\Lambda}}$
6: ▷ Compute $\tilde{\mathbf{U}} = \mathbf{U}\mathbf{Y}$.
```

**3.2. OFRR for Singular Value Decomposition.** In this section, we extend the OFRR method to Singular Value Decomposition (SVD). We first describe the orthogonal Rayleigh-Ritz projection for SVD. Consider the following SVD

$$\begin{cases} \mathbf{A}\mathbf{v} & = \sigma\mathbf{u} \\ \mathbf{A}^\top\mathbf{u} & = \sigma\mathbf{v} \end{cases} \tag{3.5}$$

where $\mathbf{A} \in \mathbb{R}^{n_1\times n_2}$, $\mathbf{v}$ is the right singular vector and $\mathbf{u}$ is the left singular vector. Given two subspaces $\mathcal{K}_1$ of dimension $k_1$ and $\mathcal{K}_2$ of dimension $k_2$, the Rayleigh-Ritz projection for SVD seeks $\tilde{\sigma} \in \mathbb{R}$, $\tilde{\mathbf{u}} \in \mathcal{K}_1$, and $\tilde{\mathbf{v}} \in \mathcal{K}_2$ such that

$$\begin{cases} \langle \mathbf{A}\tilde{\mathbf{v}} - \tilde{\sigma}\tilde{\mathbf{u}}, \mathbf{u} \rangle = 0, & \forall \mathbf{u} \in \mathcal{K}_1, \\ \langle \mathbf{A}^\top\tilde{\mathbf{u}} - \tilde{\sigma}\tilde{\mathbf{v}}, \mathbf{v} \rangle = 0, & \forall \mathbf{v} \in \mathcal{K}_2. \end{cases} \tag{3.6}$$

When we have an orthonormal basis $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{k_1}]$ for $\mathcal{K}_1$ and an orthonormal basis $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{k_2}]$ for $\mathcal{K}_2$, the classical Rayleigh-Ritz projection simply uses the SVD of $\mathbf{U}^\top\mathbf{A}\mathbf{V}$ to generate approximate singular values and singular vectors, as shown in Algorithm 3.2.

**Algorithm 3.2** *Rayleigh-Ritz Projection for SVD*

```
1: ▷ inputs: $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2}$, and $\mathbf{U} \in \mathbb{R}^{n_1 \times k_1}$, $\mathbf{V} \in \mathbb{R}^{n_2 \times k_2}$, both with orthonormal columns.
2: ▷ output: $\tilde{\mathbf{S}}$, $\tilde{\mathbf{U}}$, and $\tilde{\mathbf{V}}$   {%Approximate singular values and singular vectors}
3: ▷ Compute $\mathbf{B} = \mathbf{U}^\top \mathbf{A}\mathbf{V}$
4: ▷ Compute SVD $\mathbf{B} = \mathbf{Z}\tilde{\mathbf{S}}\mathbf{W}^\top$
5: ▷ Compute $\tilde{\mathbf{U}} = \mathbf{U}\mathbf{Z}$, $\tilde{\mathbf{V}} = \mathbf{V}\mathbf{W}$
```

Now assume that only linearly independent bases are available for $\mathcal{K}_1$ and $\mathcal{K}_2$ instead of orthonormal ones. For the Galerkin condition in (3.6) to be satisfied, we need to impose the following equations:

$$\begin{cases} \langle \mathbf{A}\tilde{\mathbf{v}} - \tilde{\sigma}\tilde{\mathbf{u}}, \mathbf{u}_i \rangle = 0, & i = 1, \cdots, k_1, \\ \langle \mathbf{A}^\top \tilde{\mathbf{u}} - \tilde{\sigma}\tilde{\mathbf{v}}, \mathbf{v}_i \rangle = 0, & i = 1, \cdots, k_2. \end{cases}$$

We can again express any vector $\tilde{\mathbf{u}} \in \mathcal{K}_1$ as $\tilde{\mathbf{u}} = \mathbf{U}\mathbf{y}$ with a unique $\mathbf{y} \in \mathbb{R}^{k_1}$ and $\tilde{\mathbf{v}} \in \mathcal{K}_2$ as $\tilde{\mathbf{v}} = \mathbf{V}\mathbf{z}$ with a unique $\mathbf{z} \in \mathbb{R}^{k_2}$. We then transform the original problem into the following system of equations in terms of $\tilde{\sigma}$, $\mathbf{y}$, and $\mathbf{z}$

$$\begin{cases} \langle \mathbf{A}\mathbf{V}\mathbf{z} - \tilde{\sigma}\mathbf{U}\mathbf{y}, \mathbf{u}_i \rangle = 0, & i = 1, \cdots, k_1, \\ \langle \mathbf{A}^\top \mathbf{U}\mathbf{y} - \tilde{\sigma}\mathbf{V}\mathbf{z}, \mathbf{v}_i \rangle = 0, & i = 1, \cdots, k_2, \end{cases}$$

which is equivalent to:

$$\begin{cases} \mathbf{U}^\top \mathbf{A}\mathbf{V}\mathbf{z} = \tilde{\sigma}\mathbf{U}^\top \mathbf{U}\mathbf{y}, \\ (\mathbf{U}^\top \mathbf{A}\mathbf{V})^\top \mathbf{y} = \tilde{\sigma}\mathbf{V}^\top \mathbf{V}\mathbf{z}. \end{cases} \tag{3.7}$$

The above system of equations can be reformulated as a block 2-by-2 generalized eigenvalue problem:

$$\begin{bmatrix} 0 & \mathbf{U}^\top \mathbf{A}\mathbf{V} \\ (\mathbf{U}^\top \mathbf{A}\mathbf{V})^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \tilde{\sigma} \begin{bmatrix} \mathbf{U}^\top \mathbf{U} & 0 \\ 0 & \mathbf{V}^\top \mathbf{V} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}. \tag{3.8}$$

When both $\mathbf{U}^\top \mathbf{U}$ and $\mathbf{V}^\top \mathbf{V}$ are identity matrices, i.e., the bases are orthonormal, this method is equivalent to the orthogonal Rayleigh-Ritz projection described in Algorithm 3.2.

It is straightforward to see that if $[\mathbf{y}^\top, \mathbf{z}^\top]^\top$ is an eigenvector of the generalized eigenvalue problem (3.8) associated with a positive eigenvalue $\tilde{\sigma}$, then $[-\mathbf{y}^\top, \mathbf{z}^\top]^\top$ is an eigenvector associated with $-\tilde{\sigma}$. Additionally, all other eigenvalues are zero. Therefore, the positive eigenvalues of (3.8) correspond exactly to the singular values that are sought.

The remaining task is to determine the approximate orthonormal singular vectors. The following theorem illustrates how these singular vectors can be constructed from the eigenvectors of (3.8).

THEOREM 3.1. *Assume the columns of $[\mathbf{Y}^\top, \mathbf{Z}^\top]^\top$ contain all the eigenvectors associated with the positive eigenvalues of (3.8) and the corresponding eigenvalues are stored in the diagonal matrix $\tilde{\mathbf{S}}$, such that*

$$\begin{bmatrix} 0 & \mathbf{U}^\top \mathbf{A}\mathbf{V} \\ (\mathbf{U}^\top \mathbf{A}\mathbf{V})^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Y} \\ \mathbf{Z} \end{bmatrix} = \begin{bmatrix} \mathbf{U}^\top \mathbf{U} & 0 \\ 0 & \mathbf{V}^\top \mathbf{V} \end{bmatrix} \begin{bmatrix} \mathbf{Y} \\ \mathbf{Z} \end{bmatrix} \tilde{\mathbf{S}}.$$

*This manuscript is for review purposes only.*

*Then, the columns of* $\tilde{\mathbf{U}} = \sqrt{2}\mathbf{UY}$ *and* $\tilde{\mathbf{V}} = \sqrt{2}\mathbf{VZ}$ *are orthonormal.*

*Proof.* According to the theory of the generalized eigenvalue problem, the eigenvectors of different eigenvalues are orthogonal under the A-inner product defined by the mass matrix. For the two different eigenpairs $(\tilde{\sigma}_i; [\mathbf{y}_i^\top, \mathbf{z}_i^\top]^\top)$ and $(\tilde{\sigma}_j; [\mathbf{y}_j^\top, \mathbf{z}_j^\top]^\top)$ of (3.8) where $i \neq j$, we have:

$$\begin{bmatrix}\mathbf{y}_j^\top & \mathbf{z}_j^\top\end{bmatrix} \begin{bmatrix}\mathbf{U}^\top\mathbf{U} & 0 \\ 0 & \mathbf{V}^\top\mathbf{V}\end{bmatrix} \begin{bmatrix}\mathbf{y}_i \\ \mathbf{z}_i\end{bmatrix} = 0 \Rightarrow \mathbf{y}_j^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i = -\mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i. \tag{3.9}$$

Since $(\tilde{\sigma}_i; [\mathbf{y}_i^\top, \mathbf{z}_i^\top]^\top)$ is an eigenpair of (3.8), we also have

$$\mathbf{V}^\top \mathbf{A}^\top \mathbf{U}\mathbf{y}_i = \tilde{\sigma}_i \mathbf{V}^\top \mathbf{V}\mathbf{z}_i \Rightarrow \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{A}^\top \mathbf{U}\mathbf{y}_i = \tilde{\sigma}_i \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i. \tag{3.10}$$

Similarly, we know that

$$\mathbf{y}_i^\top \mathbf{U}^\top \mathbf{A}\mathbf{V}\mathbf{z}_j = \tilde{\sigma}_j \mathbf{y}_i^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_j = \tilde{\sigma}_j \mathbf{y}_j^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i. \tag{3.11}$$

Given that $\mathbf{z}_j^\top \mathbf{V}^\top \mathbf{A}^\top \mathbf{U}\mathbf{y}_i = \mathbf{y}_i^\top \mathbf{U}^\top \mathbf{A}\mathbf{V}\mathbf{z}_j$, we can combine (3.10) and (3.11) and obtain

$$\tilde{\sigma}_i \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = \tilde{\sigma}_j \mathbf{y}_j^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i. \tag{3.12}$$

By integrating (3.9) with (3.12), we obtain

$$\tilde{\sigma}_i \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = \tilde{\sigma}_j \mathbf{y}_j^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i = -\tilde{\sigma}_j \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i \Rightarrow (\tilde{\sigma}_i + \tilde{\sigma}_j)\mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = 0,$$

where the second equal sign is due to (3.9). Since both $\tilde{\sigma}_i$ and $\tilde{\sigma}_j$ are positive, $\tilde{\sigma}_i + \tilde{\sigma}_j \neq 0$ which implies $\mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = 0$. We can show a similar property for $\mathbf{y}$ as

$$\mathbf{y}_j^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i = \frac{\tilde{\sigma}_i}{\tilde{\sigma}_j} \mathbf{z}_j^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = 0. \tag{3.13}$$

Next, we discuss the situation when $i = j$. In this case, we have:

$$\begin{bmatrix}\mathbf{y}_i^\top & \mathbf{z}_i^\top\end{bmatrix} \begin{bmatrix}\mathbf{U}^\top\mathbf{U} & 0 \\ 0 & \mathbf{V}^\top\mathbf{V}\end{bmatrix} \begin{bmatrix}\mathbf{y}_i \\ \mathbf{z}_i\end{bmatrix} = 1 \Rightarrow \mathbf{y}_i^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i + \mathbf{z}_i^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = 1. \tag{3.14}$$

Note that obtaining (3.12) does not require $i \neq j$, so we also have

$$\tilde{\sigma}_i \mathbf{z}_i^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = \tilde{\sigma}_i \mathbf{y}_i^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i \Rightarrow \mathbf{z}_i^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = \mathbf{y}_i^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i. \tag{3.15}$$

Combining (3.14) and (3.15), we have

$$\mathbf{y}_i^\top \mathbf{U}^\top \mathbf{U}\mathbf{y}_i = \mathbf{z}_i^\top \mathbf{V}^\top \mathbf{V}\mathbf{z}_i = 1/2 \tag{3.16}$$

From the results in (3.13) and in (3.16), it is obvious that $\tilde{\mathbf{U}} = \sqrt{2}\mathbf{UY}$ and $\tilde{\mathbf{V}} = \sqrt{2}\mathbf{VZ}$ are orthonormal. ∎

We conclude this section by summarizing the final algorithm in Algorithm 3.3. The proposed orthogonalization-free Rayleigh–Ritz projection requires solving a generalized eigenvalue problem of dimension $k_1+k_2$, in contrast to the standard two-sided Rayleigh–Ritz projection, which involves an SVD on a $k_1 \times k_2$ matrix. Nonetheless, this approach can preserve good accuracy even under loss of orthogonality in low-precision computations.

**Algorithm 3.3** *Orthogonalization-Free Rayleigh-Ritz Projection for SVD*

```
1: ▷ input: $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{U} \in \mathbb{R}^{n_1 \times k_1}$, $\mathbf{V} \in \mathbb{R}^{n_2 \times k_2}$
2: ▷ output: $\tilde{\mathbf{S}}$, $\tilde{\mathbf{U}}$, and $\tilde{\mathbf{V}}$   {%Approximate singular values and singular vectors}
3: ▷ Solve the following generalized eigenvalue problem for all positive eigenvalues
       $\begin{bmatrix} 0 & \mathbf{U}^\top \mathbf{A}\mathbf{V} \\ (\mathbf{U}^\top \mathbf{A}\mathbf{V})^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} = \tilde{\sigma} \begin{bmatrix} \mathbf{U}^\top \mathbf{U} & 0 \\ 0 & \mathbf{V}^\top \mathbf{V} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$
4: ▷ Assemble the eigenvectors associated with positive eigenvalues into the columns
   of matrices $\mathbf{Y}$ and $\mathbf{Z}$, and insert these eigenvalues into the diagonal of the diagonal
   matrix $\tilde{\mathbf{S}}$.
5: ▷ Compute $\tilde{\mathbf{U}} = \sqrt{2}\mathbf{U}\mathbf{Y}$, $\tilde{\mathbf{V}} = \sqrt{2}\mathbf{V}\mathbf{Z}$
```

## 4. Construction of Linearly Independent Bases.

In the previous section, we proposed the OFRR method for eigenvalue problems and SVD. Unlike orthogonal projection methods, which require an orthogonal basis, OFRR allows for more flexibility with non-orthogonal bases. This section will focus on various strategies to enhance the linear independence of bases for effective integration with OFRR. The procedure for SVD is similar to the eigenvalue computations, with the only difference being the additional matrix-vector multiplications with $\mathbf{A}^\top$. For the sake of conciseness, we omit the discussion of the SVD algorithm.

### 4.1. Linearly Independent Basis for Krylov Subspace Methods.

In this section, we will focus on generating linearly independent bases for the Krylov subspace $\mathcal{K}_k(\mathbf{v}, \mathbf{A})$ for $\mathbf{A} \in \mathbb{R}^{n\times n}$. Under the OFRR framework, where orthogonality is not required, the simplest approach is to directly use the matrix $\mathbf{K} := [\mathbf{v}, \mathbf{A}\mathbf{v}, \cdots, \mathbf{A}^{k-1}\mathbf{v}]$ without any modification. The generalized eigenvalue problem using OFRR would then be:

$$\mathbf{K}^\top \mathbf{A}\mathbf{K}\mathbf{y} = \tilde{\lambda}\mathbf{K}^\top \mathbf{K}\mathbf{y}. \tag{4.1}$$

However, there are several reasons why this approach is generally not recommended.

First, some columns of $\mathbf{K}$ might be nearly linearly dependent, especially when the original matrix $\mathbf{A}$ is numerically low-rank. Direct use of $\mathbf{K}$ could result in a mass matrix $\mathbf{K}^\top\mathbf{K}$ that is extremely ill-conditioned in this case, which adversely affects the numerical stability of the eigenvalue algorithm. Second, overflow can arise in computations especially with reduced precision. While column scaling might be applied to normalize the infinity norm of each column of $\mathbf{K}$ to one, the magnitudes of the columns' 2-norms can remain large. This can potentially lead to overflow when forming $\mathbf{K}^\top\mathbf{A}\mathbf{K}$ and $\mathbf{K}^\top\mathbf{K}$ in reduced-precision environments. Therefore, it is still essential to use algorithms that avoid poorly conditioned bases.

#### 4.1.1. Arnoldi Method.

A straightforward approach is to employ standard methods for constructing an orthogonal basis, simply executing them using reduced precision arithmetic. For instance, the Arnoldi method – the most widely adopted technique for building an orthogonal basis of the Krylov subspace associated with a general matrix – could be applied to construct linearly independent bases. One variant of the Arnoldi algorithm is shown in Algorithm 4.1, where Modified Gram-Schmidt (MGS) is used to build an orthogonal basis for the Krylov subspace. It is worth noting that when the input matrix $\mathbf{A}$ is symmetric, applying this general

Arnoldi procedure becomes computationally equivalent to the Lanczos algorithm with full orthogonalization, a variant often employed for enhanced numerical stability. In some applications implemented using double precision, where orthogonality is critical, re-orthogonalization is typically enabled. With re-orthogonalization, Lines 8–10 in Algorithm 4.1 are repeated once if the 2-norm of $\mathbf{v}$ after projection is reduced by more than a certain tolerance.

**Algorithm 4.1** *Computing orthogonal bases from the Arnoldi Process with MGS*

```
1: input: $\mathbf{A} \in \mathbb{C}^{n\times n}$, $\mathbf{v}$, $k$
2: output: $\mathbf{V}$                    {linearly independent basis for $\mathcal{K}_k(\mathbf{A}, \mathbf{v})$}
3: ▷ Initialize matrix $\mathbf{V}$
4: ▷ Update $\mathbf{v} := \mathbf{v}/\|\mathbf{v}\|_2$
5: ▷ Set $\mathbf{V}_{:,1} = \mathbf{v}$
6: for $j = 1 : k-1$ do
7:     ▷ Compute $\mathbf{v} = \mathbf{A}\mathbf{V}_{:,j}$
8:     for $i = 1 : j$ do
9:         ▷ Compute $\mathbf{v} := \mathbf{v} - \langle \mathbf{V}_{:,i}, \mathbf{v}\rangle \mathbf{V}_{:,i}$
10:    end for
11:    ▷ Update $\mathbf{v} := \mathbf{v}/\|\mathbf{v}\|_2$
12:    ▷ Set $\mathbf{V}_{:,j+1} = \mathbf{v}$
13: end for
```

Classical Gram-Schmidt (CGS) with re-orthogonalization is also commonly used in scientific computing because it can leverage BLAS level-2 operations for computational performance and significantly reduces the number of parallel reduction operations (required in computing inner products) compared to MGS. Note that in the context of OFRR, re-orthogonalization may not be needed.

For $\mathbf{A} \in \mathbb{R}^{n\times n}$, computing the $j$-th column of $\mathbf{V}$ requires approximately $4nj$ FLOPs, leading to a total cost of roughly $2nk^2$ excluding the matrix-vector multiplication with $\mathbf{A}$. While CGS has the same approximate FLOP count ($2nk^2$), it differs structurally from MGS by using BLAS level-2 operations. Specifically, it employs matrix-vector products to compute sets of inner products $\langle \mathbf{V}_{:,i}, \mathbf{v}\rangle$ and perform vector updates, contrasting with the BLAS level-1 operations used in MGS. Re-orthogonalization would double the cost to $4nk^2$ for both methods. Alternatively, Householder reflectors can also be used to generate the orthonormal basis, offering superior numerical stability. However, explicit formation of $\mathbf{V}$, necessary for certain applications, makes the total FLOP count approximately $4nk^2$.

Therefore, in reduced-precision environments with OFRR where strict orthonormality may not be required, CGS or MGS without re-orthogonalization offer improved FLOP efficiency and are often preferred to Householder reflectors. In the following sections, we will only discuss the use of CGS and MGS.

While the Arnoldi process using Gram-Schmidt orthogonalization can provide good numerical stability for OFRR, its reliance on full orthogonalization is often computationally expensive. A primary reason for this expense is the frequent requirement for inner product computations inherent in Gram-Schmidt methods.

Furthermore, performing these inner products in low-precision formats, such as half precision, presents significant challenges beyond just the computational cost. The limited dynamic range increases the risk of overflow or underflow during summation, and precision loss can severely compromise the numerical stability of the orthogonalization process. While strategies like accumulating inner products in higher precision or applying dynamic vector scaling can mitigate these issues, they introduce additional computational overhead.

To reduce computational demands, we will explore alternative methods or modifications that mitigate the cost and numerical issues arising from inner product computations in reduced precision in the next section.

**4.1.2. Krylov-Hessenberg Process.** In this section, we propose to adopt the Hessenberg process as an alternative to generate linearly independent bases. This method is derived from the Generalized Hessenberg process, as detailed in Wilkinson's classical book [32, Chap. 6]. Unlike traditional methods that depend on inner products to compute the projection coefficients, the Hessenberg process obtains projection coefficients by extracting entries directly from previously computed bases. More specifically, the procedure generates the basis vectors $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_k$ of the Krylov subspace in the usual Arnoldi-like fashion except that orthogonality is enforced against a preselected set of vectors $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_k$ instead of the $\mathbf{v}_i$'s themselves. Thus, at step $j$ of the procedure we compute the vector $\mathbf{A}\mathbf{v}_j$ and orthogonalize it against $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_j$, leading to a vector $\mathbf{v}_{j+1}$ that satisfies the usual relation among Krylov basis vectors:

$$\mathbf{h}_{j+1,j}\mathbf{v}_{j+1} = \mathbf{A}\mathbf{v}_j - \sum_{i=1}^{j} \mathbf{h}_{i,j}\mathbf{v}_i. \tag{4.2}$$

In this paper, we consider the simplest case where we choose $\mathbf{z}_i = \mathbf{e}_i$, and scale all $\mathbf{v}_i$'s so that $\|\mathbf{v}_i\|_\infty = 1$. This procedure has been advocated in [27] as an alternative to GMRES for solving linear systems of equations iteratively. Later it was also exploited for solving *dense* linear systems, see, for example, [15]. The Hessenberg algorithm for generating a non-orthogonal basis is summarized in Algorithm 4.2.

**Algorithm 4.2** *Computing non-orthogonal bases from the Krylov-Hessenberg Process*

```
 1: input: $\mathbf{A} \in \mathbb{C}^{n\times n}$, $\mathbf{v}$, $k$
 2: output: $\mathbf{V}$                    {linearly independent basis for $\mathcal{K}_k(\mathbf{A}, \mathbf{v})$}
 3: ▷ Initialize matrix $\mathbf{V}$
 4: ▷ Initialize permutation vector $\pi$
 5: ▷ Find $r$ the index of element in $\mathbf{v}$ with largest magnitude
 6: ▷ Update $\mathbf{v} := \mathbf{v}/\mathbf{v}_r$
 7: ▷ Set $\pi_1 := r$
 8: ▷ Set $\mathbf{V}_{:,1} = \mathbf{v}$
 9: for $j = 1 : k-1$ do
10:     ▷ Compute $\mathbf{v} = \mathbf{A}\mathbf{V}_{:,j}$
11:     for $i = 1 : j$ do
12:         ▷ Compute $\mathbf{v} := \mathbf{v} - \mathbf{v}(\pi_i)\mathbf{V}_{:,i}$
13:     end for
14:     ▷ Find $r$ the index of element in $\mathbf{v}$ with largest magnitude.
15:     ▷ Update $\mathbf{v} := \mathbf{v}/\mathbf{v}_r$
16:     ▷ Set $\pi_{j+1} := r$
17:     ▷ Set $\mathbf{V}_{:,j+1} = \mathbf{v}$
18: end for
```

As can be seen the Hessenberg process is inner-product free. Only one reduction operator to obtain the index of the entry with the largest magnitude is needed during each outer step (Line 14 of Algorithm 4.1). This constitutes a significant advantage over the Arnoldi process. The arithmetic operations involved are also less prone to numerical stability issues as will be seen later. Note also that he FLOP count for the Hessenberg process when $n \gg k$ is roughly $nk^2$ excluding the matrix-vector multiplication with $\mathbf{A}$.

**4.2. Linearly Independent Basis for Subspace Iteration.** The previous discussions have focused on strategies for constructing linearly independent bases for the Krylov subspace. We now shift focus to subspace iteration.

Subspace iteration, in contrast, adopts a block-oriented approach. It allows for the potential use of higher-level `BLAS` operations (e.g., `BLAS` level-3 for the matrix-block product if $\mathbf{A}$ is dense) and can lead to improved computational efficiency on modern architectures compared to the Krylov methods. Similar to the challenges encountered in naive Krylov-based implementations, directly applying $\mathbf{X}_{\text{iter}} = \mathbf{A}^{\text{iter}}\mathbf{X}_0$ without modification may result in severe numerical difficulties.

**4.2.1. QR factorization.** A straightforward way to construct a linearly independent basis with an improved condition number is to run a QR factorization with column pivoting using either CGS or MGS. Although the arithmetic work is identical to that in Arnoldi—$2mn^2$ FLOPs for a single sweep and $4mn^2$ FLOPs with re-orthogonalization—the practical speed can differ greatly.

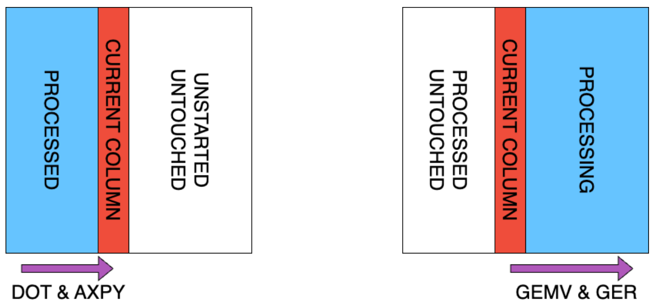

FIGURE 2. *The MGS sweep used within Arnoldi (left) and a “right-looking” variant of MGS for subspace iteration (right). In Arnoldi each processed basis vector updates the single current column via repeated `DOT` & `AXPY` operations. In the ‘righ-looking’ version, once the current column is orthogonal, a single `GEMV` & `GER` applies its correction to the entire trailing block at once.*

As shown in the left panel of Figure 2, the MGS sweep embedded in Arnoldi is “left-looking”: at step $j$ only the columns already computed are available, so the projection must be carried out through $j$ successive `DOT`–`AXPY` pairs (`BLAS` level-1), making the computation memory-bound. In contrast, for subspace iteration the entire block is resident in memory; the sweep can therefore be organized in a “right-looking” manner (right panel of Figure 2), where one `GEMV` followed by a rank-1 `GER` updates all trailing columns at once. Packaging the same FLOPs into `BLAS` level-2 calls raises arithmetic intensity and yields markedly higher sustained performance on modern CPUs and GPUs.

**4.2.2. The Hessenberg Process.** The Krylov-Hessenberg process introduced in the previous section can be readily modified for building a linearly independent basis for the subspace iteration. Similar to MGS, the Hessenberg process for subspace iteration could also be implemented in a ‘right-looking’ way, as detailed in Lines 13-15 of Algorithm 4.3. The output of this algorithm returns a linearly independent basis $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots]$. Since the columns of $\mathbf{A}$ may be nearly linearly dependent, it is crucial to skip columns with negligible magnitude during factorization as implemented in Line 9 of Algorithm 4.3. Specifically, when selecting row pivots, if the largest magnitude entry in a column is close to the working precision, this column should be skipped, and the factorization should continue with the next column. The objective is to ensure that span($\mathbf{Q}$) closely approximates span($\mathbf{A}$). Therefore, zero or near-zero columns should be omitted rather than padded with standard basis vectors to reflect the correct numerical rank.

**Algorithm 4.3** *Computing non-orthogonal bases from the Hessenberg Process*

```
 1: input: $\mathbf{A} \in \mathbb{C}^{n\times k}$, $tol$          {$tol$ is to exclude zero columns}
 2: output: $\mathbf{Q}$                                          {linearly independent basis}
 3: ▷ Initialize matrix $\mathbf{Q} = \mathbf{A}$
 4: ▷ Initialize nonzero column indicator vector $\mathbf{s}$ to TRUE
 5: ▷ Initialize permutation vector $\pi$
 6: for $j = 1 : k$ do
 7:    ▷ Set $\mathbf{q} = \mathbf{Q}_{:,j}$
 8:    ▷ Find $r$ the index of element in $\mathbf{q}$ with largest magnitude.
 9:    if $|\mathbf{q}_r| \geq tol$ then
10:       ▷ Update $\mathbf{q} := \mathbf{q}/\mathbf{q}_r$
11:       ▷ Set $\pi_j = r$
12:       ▷ Set $\mathbf{Q}_{:,j} = \mathbf{q}$
13:       for $i = j+1 : k$ do
14:          ▷ Update $\mathbf{Q}_{:,i} := \mathbf{Q}_{:,i} - \mathbf{Q}_{\pi_i,i}\mathbf{q}$
15:       end for
16:    else
17:       ▷ Set $\mathbf{s}_j$ to FALSE
18:       ▷ Set $\pi_j = 1$
19:    end if
20: end for
21: ▷ Set $\mathbf{Q} = \mathbf{Q}_{:,\mathbf{s}}$
```

Finally, we discuss the connection between the Hessenberg process and the LU factorization. For a given $\mathbf{A} \in \mathbb{C}^{n\times k}$ with full column rank, LU factorization with row pivoting computes

$$\mathbf{PA} = \mathbf{LU} \Rightarrow \mathbf{A} = (\mathbf{P}^\top \mathbf{L})\mathbf{U}, \tag{4.3}$$

where $\mathbf{L} \in \mathbb{C}^{n\times k}$, $\mathbf{U} \in \mathbb{C}^{k\times k}$, and $\mathbf{P} \in \mathbb{R}^{n\times n}$ is a permutation matrix, i.e., a matrix obtained by reordering the rows of an identity matrix. The matrix $\mathbf{P}^\top\mathbf{L}$ now has the same range as $\mathbf{A}$, and its columns could be used as a linearly independent basis for the column space of $\mathbf{A}$.

Although both the Hessenberg process and the LU factorization have been widely used, their direct algorithmic relationship is worth highlighting. An interesting observation is that the output matrix produced by the Hessenberg process in Algorithm 4.3 is identical to the output matrix obtained from the row-pivoted LU factorization. Thus, the numerical stability analysis of the Hessenberg process is supported by existing results on the LU factorization.

Motivated by this connection, we next examine advances in mixed-precision LU factorization, which has become a topic of significant interest due to its potential for accelerating the solution of large-scale linear systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$. One development was made by Haidar et al. [13], who proposed a low-precision LU factorization strategy within an iterative refinement framework. Their work relied on a partitioned, 'right-looking' LU algorithm designed to maximize data locality and arithmetic intensity. More recently, Lopez et al. [20] proposed a highly efficient mixed-precision approach based on a partitioned 'left-looking' LU factorization. The core idea behind such partitioned approaches is to perform the factorization on $r \times r$ blocks, allowing the algorithms to leverage high-performance `BLAS` level-3 operations. Another contribution [28], explores pre-pivoting strategies within mixed-precision frameworks, where the ultimate goal is to achieve effective `FP64` accuracy.

Despite the focus of these efforts on solving linear systems, rather than subspace generation, the methodological innovations developed therein are highly relevant to our proposed work. Specifically, the structured block-wise execution and memory-aware optimizations introduced in partitioned LU schemes may be adapted to enhance the efficiency of the Hessenberg process when used for constructing basis matrices in Krylov subspace and subspace iteration methods. Such a generalization would not only expand the utility of the Hessenberg process but could also yield significant performance benefits on modern computing architectures. A rigorous exploration of these extensions—especially in the context of block Krylov methods—presents a promising direction for future research.

**4.2.3. Condition Number Comparison of Computed Bases.** In this subsection, we compare the condition numbers of the bases generated by various methods discussed in the previous subsections. We choose Gaussian kernel matrices with data uniformly sampled within a square of edge length $\sqrt{1000}$ in the experiment. This time, we fix $s = 0.01$ and vary $l$ from 1 to 100 in (2.2) to test matrices with different spectral properties. When $l$ is close to 1, the eigenvalues of the matrix decay slowly, and as $l$ approaches 100, most of its eigenvalues would be close to $s$. To be specific, when $l = 1$ the 20-th largest eigenvalue is larger than 6, while when $l = 100$ the 7-th largest eigenvalue is already close to 0.01. We perform subspace iteration with $m = 1$, iter $= 3$, and $k = 20$ to compute $\mathbf{X}_{iter}$, and then compute the condition number of the postprocessed basis matrix $\mathbf{Q}$. Specifically, we calculate the condition numbers of $\mathbf{X}_{iter}$, $\mathbf{Q}$ obtained through Modified Gram-Schmidt (MGS) with re-orthogonalization, Classical Gram-Schmidt (CGS) with re-orthogonalization, and the one returned from the Hessenberg process. The subspace iteration and basis computations are performed in double, single, and half precision. As the results in single precision are consistently close to those in double precision, we exclude them from the figure to improve readability. Notably, for the half-precision configuration in this experiment, the computations are carried out using native `FP16` arithmetic without intermediate calculations in `FP32`, presenting a more challenging scenario for maintaining numerical stability. The final condition numbers are calculated in double precision for accuracy.

As we can see from Figure 3, the condition number of the original $\mathbf{X}_{iter}$ increases significantly as the problem becomes numerically low-rank, necessitating the use of QR-like strategies for building a well-conditioned basis. On the other hand, double-precision MGS and CGS (with re-orthogonalization) consistently produce a basis with a condition number close to 1. However, as the matrices become numerically low-rank, MGS and CGS begins to lose orthogonality under half-precision, leading to higher condition numbers. Although the condition numbers for the Hessenberg process are slightly larger than those for MGS and CGS in most cases, they exhibit consistently

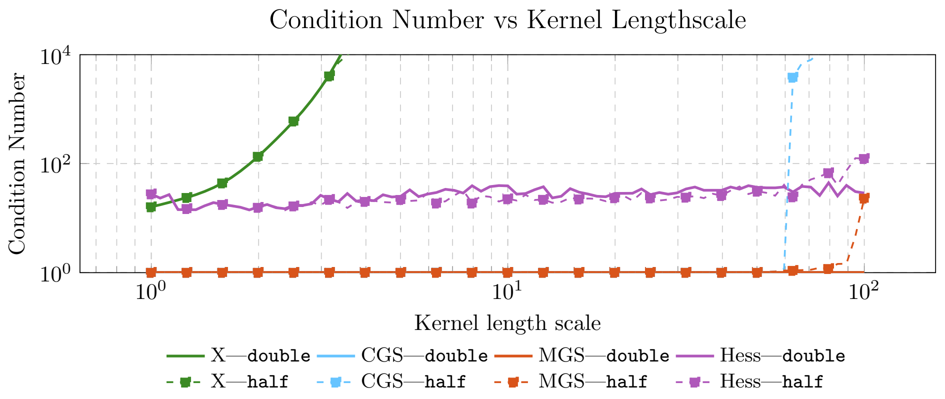

FIGURE 3. *Condition number for bases computed by four different methods: no stabilization (-X), MGS with re-orthogonalization, CGS with re-orthogonalization, and Hessenberg. Tests are performed on multiple kernel matrices, each sized* $1000 \times 1000$*, with length scales varying from* 1 *to* 100.

low variance across different length scales and precisions. This stability, combined with its efficiency, makes it a robust and efficient option for building the bases with reduced precision arithmetic.

Next, Table 1 provides a comparative summary of the dominant `BLAS` operations, FLOPs, parallelism, and numerical stability across different variants of Modified Gram-Schmidt (MGS), Classical Gram-Schmidt (CGS), and the Hessenberg process.

TABLE 1
*Comparison of MSG, CGS and Hessenberg Process when applied to an* $n \times k$ *matrix* $\mathbf{A}$ *when* $n \gg k$.

| Method Variant | Dominant `BLAS` | FLOPs (Order) | Parallelism | Stability |
|---|---|---|---|---|
| **Modified Gram-Schmidt (MGS)** | | | | |
| Left-Looking MGS | Level 1 | $2nk^2$ | Low | Moderate |
| Left-Looking MGS re-orth | Level 1 | $4nk^2$ | Low | Good |
| Right-Looking MGS | Level 2 | $2nk^2$ | Moderate | Moderate |
| **Classical Gram-Schmidt (CGS)** | | | | |
| CGS | Level 2 | $2nk^2$ | Moderate | Low |
| CGS2 | Level 2 | $4nk^2$ | Moderate | Moderate |
| **Hessenberg Process** | | | | |
| Left-Looking | Level 1 | $nk^2$ | Moderate | Good |
| Right-Looking | Level 2 | $nk^2$ | Moderate | Good |
| Block | Level 3 | $nk^2$ | Good | Good |

To conclude this section, we outline the mixed-precision strategies employed within the OFRR framework. These strategies are adapted based on the precision in which the input matrix $\mathbf{A}$ is available. Figure 4 serves to illustrate the data flow and suggested precision choices for one important scenario: applying subspace iteration to a matrix $\mathbf{A}$ provided only in half precision (`FP16`). When employing Krylov subspace methods within the OFRR framework, the core principle of major steps remains similar.

As illustrated in Figure 4, memory conservation is prioritized by storing the basis vectors $\mathbf{V}$ and intermediate results like $\mathbf{W} := \mathbf{AV}$ primarily in `FP16` format. Computations such as applying matrix-matrix multiplication with $\mathbf{A}$ and performing the basis construction (OFRR Hessenberg/QR) can often use `FP16` compute precision, potentially enhanced with `FP32` accumulation. A critical step is the projection step to form the matrices $\mathbf{B} := \mathbf{V}^\top\mathbf{W}$ and $\mathbf{M} := \mathbf{V}^\top\mathbf{V}$. This step takes input matrices (like $\mathbf{V}$ and $\mathbf{W}$) in `FP16`, performs the matrix multiplications and accumulations using `FP32` compute precision, and stores the resulting small matrices $\mathbf{B}$ and $\mathbf{M}$ in `FP32`. Finally, these `FP32` matrices are promoted to `FP64` to solve the generalized eigenvalue problem defined by matrix pencil $(\mathbf{B}, \mathbf{M})$ using standard double-precision solvers.

The strategy is much simpler for subspace iteration with higher-precision inputs. If $\mathbf{A}$ is `FP64`, all storage and computations throughout the process typically remain in `FP64`. If $\mathbf{A}$ is `FP32`, the framework generally operates with `FP32` as the working precision for both storage and computation, with the final projected generalized eigenvalue problem solved in `FP64` to maximize the accuracy of the resulting eigenpairs.

In summary, the OFRR framework flexibly integrates mixed-precision strategies, leveraging low-precision storage and computation where feasible, while strategically increasing precision for numerically sensitive stages like projection and the final solve.

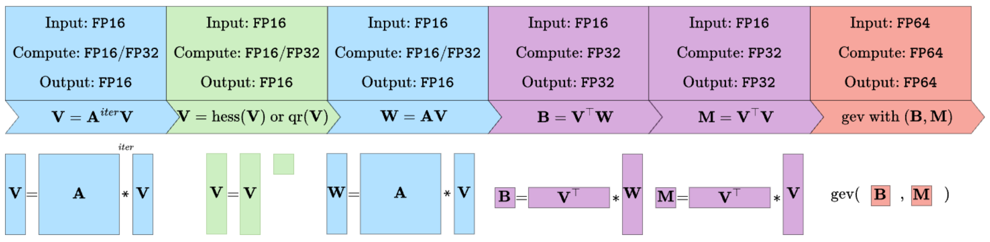
Input: FP16
Compute: FP16/FP32
Output: FP16
$\mathbf{V} = \mathbf{A}^{iter}\mathbf{V}$
Input: FP16
Compute: FP16/FP32
Output: FP16
$\mathbf{V}$ = hess($\mathbf{V}$) or qr($\mathbf{V}$)
Input: FP16
Compute: FP16/FP32
Output: FP16
$\mathbf{W} = \mathbf{AV}$
Input: FP16
Compute: FP32
Output: FP32
$\mathbf{B} = \mathbf{V}^\top\mathbf{W}$
Input: FP16
Compute: FP32
Output: FP32
$\mathbf{M} = \mathbf{V}^\top\mathbf{V}$
Input: FP64
Compute: FP64
Output: FP64
gev with $(\mathbf{B}, \mathbf{M})$
$\mathbf{V} = \mathbf{A}^{iter} * \mathbf{V}$
$\mathbf{V} = \mathbf{V}$
$\mathbf{W} = \mathbf{A} * \mathbf{V}$
$\mathbf{B} = \mathbf{V}^\top * \mathbf{W}$
$\mathbf{M} = \mathbf{V}^\top * \mathbf{V}$
gev( $\mathbf{B}$ , $\mathbf{M}$ )

FIGURE 4. *Mixed-precision strategy within the OFRR framework using subspace iteration for an* `FP16` *input matrix* $\mathbf{A}$*. Each stage indicates typical precision choices for data storage (Input/Output) and computation (Compute), illustrating the progression from* `FP16` *working precision to* `FP32` *projection and an* `FP64` *final solve.*

## 5. Numerical Experiments.

In this section, we test our OFRR framework on different problems that require (partial) eigenvalue or singular value decompositions under various numerical precision settings. Our evaluation consists of two parts: (i) numerical accuracy is assessed using simulations implemented in `MATLAB` (version 2024b); (ii) computational efficiency is evaluated using an optimized `C++` implementation accelerated by CUDA, compiled with `nvcc` (version 12.8). All experiments were conducted on a hardware platform running Ubuntu 24.04.2 LTS, equipped with an Intel(R) Core(TM) i7-12700K CPU (8 Performance-cores @ 3.60 GHz and 4 Efficient-cores @ 2.70 GHz), 64 GB of system memory, and a `NVIDIA` GeForce RTX 3070 Ti GPU (8 GB GDDR6X VRAM, compute capability 8.6, and 6144 `CUDA` cores). For reproducibility, our research code is publicly available on `GitHub`[1].

### 5.1. Implementation Details.

Our `MATLAB` implementation was designed to rigorously evaluate the numerical accuracy of the proposed OFRR framework across

[1] https://github.com/Hitenze/MixedPrecisionOFRR

different precision settings.

We simulated half-precision arithmetic using `MATLAB`'s built-in `half` datatype and added custom functions to precisely control numerical precision. A key detail is that `MATLAB`'s standard operations on `half` arrays often perform intermediate computations in higher precision, and only convert the final result back to `FP16`. This type of mixed-precision behavior is common in libraries like `cuBLAS`, although some GPU routines support full `FP16` execution.

To ensure that all computations adhered strictly to our intended precision model, we required full control over every arithmetic step. For this reason, we avoided using `MATLAB`'s built-in `qr` function and instead implemented our own versions of the MGS algorithm with re-orthogonalization to serve as the QR-based baseline. This approach allowed us to guarantee that every operation followed the specified precision path, with no hidden accuracy promotions or conversions. For MGS, we used the standard threshold $\sqrt{2}/2$ to detect loss of orthogonality and trigger re-orthogonalization.

Furthermore, several other custom functions were necessary because `MATLAB` lacks native half-precision support for certain operations. We implemented a custom 2-norm function specifically for low precision, using the standard technique of scaling the vector by its infinity norm before computing the 2-norm, i.e., $\|\mathbf{x}\|_2 = \|\mathbf{x}\|_\infty \|\mathbf{x}/\|\mathbf{x}\|_\infty\|_2$. This technique mitigates overflow and underflow issues that are common in low-precision arithmetic. For sparse matrix operations, we implemented custom routines based on the Compressed Sparse Row (CSR) format, chosen for its implementation simplicity. For experiments conducted in single precision and double precision (FP64), we used standard `MATLAB` data types and built-in functions.

In addition to the `MATLAB` implementation for accuracy studies, we developed `C++/CUDA` implementations of key subroutines to evaluate runtime performance on GPUs. The `C++` implementation employs the standard `FP16` data type defined in `cuda_fp16.h`, and integrates functions from `cuBLAS`, standard `BLAS`, and `LAPACK`. All dense matrices are stored in column-major order (`Fortran`-style), and all `cuBLAS` calls and custom kernels are executed on the default `CUDA` stream. Unless otherwise specified, the primary matrix data resides in device memory during computation. Scalar parameters (e.g., weights for linear combinations or column scaling factors) used in `cuBLAS` routines are passed from host memory by configuring the `cuBLAS` pointer mode `CUBLAS_POINTER_MODE_HOST`.

For 'left-looking' MGS implementation, we utilized `cuBLAS` routines `cublasDotEx`, `cublasAxpyEx`, and `cublasScaleEx`. For 'right-looking' MGS and CGS, we employed `cublasGemmEx` for efficient column updates. Note that we use `BLAS` level-3 routine rather than a Level-2 `GEMV`-based approach, primarily because `cuBLAS` does not provide a `GEMV` routine with the same flexibility in mixed-precision configurations as `cublasGemmEx`. We implemented custom `CUDA` kernels for most operations in the 'right-looking' version of Hessenberg process. The first kernel is used to identify the index $\pi_j$ of the element possessing the largest magnitude within the relevant subvector of a given column $j$ since the standard `cuBLAS` `cublasI<t>amax` routines lack `FP16` support. The resulting index $\pi_j$ is stored directly in device memory. Following the identification of $\pi_j$, the $j$-th column is scaled based on the value of the element at this index. Subsequently in the 'right-looking' version, using this scaled vector, we compute the necessary scaling weights required for updating subsequent columns. Finally, these computed weights are used to apply the transformation to all subsequent columns ($j + 1$ to $n$) through a linear combination, which is executed by another custom `CUDA` update kernel.

It is important to note that while our custom kernels correctly implement the required functionality and demonstrate effective performance in our experiments, they were developed as prototypes. Unlike the highly tuned routines available in libraries such as `cuBLAS`, our kernels have not undergone extensive performance optimization. For instance, we employed fixed kernel block sizes across all tests and did not undertake architecture-specific tuning to identify the most efficient configurations for different GPUs. Also, we did not use block update as in the LU factorization routines discussed earlier based on `MAGMA` [1]. Consequently, although the current implementation already achieves a notable level of performance, we anticipate that substantial further speedups could be realized through dedicated optimization efforts targeting these custom kernels.

**5.2. Approximation Accuracy.** We first evaluated the accuracy of OFRR across three problem classes using `MATLAB`. For all experiments, we followed a consistent methodology. Each experiment was repeated with three precisions: `FP64`, `FP32` and `FP16` for MatVecs and building bases. Note that we always used double precision `FP64` when solving the final small (generalized) eigenvalue problems in the projected space. We used single precision to form this problem for the half-precision cases, as described in Figure 4. The tolerances to exclude columns were set dynamically relative to the machine epsilon of the respective precision, defined as $\varepsilon_{\texttt{FP64}} \approx 2.22 \times 10^{-16}$ for `FP64`, $\varepsilon_{\texttt{FP32}} \approx 1.19 \times 10^{-7}$ for `FP32`, and $\varepsilon_{\texttt{FP16}} \approx 9.77 \times 10^{-4}$ for `FP16`. Random initial matrices/vectors with entries drawn from the uniform distribution $\mathcal{U}(0,1)$ were generated in FP64 and cast to the target precision; same sample was reused within each group of tests.

**5.2.1. Eigenvalue problems with kernel matrices.** In our first set of experiments, we tested the performance of OFRR on eigenvalue problems with the Gaussian kernel matrices defined in Section 2 using a large length scale parameter $l$ and a small variance parameter $s$. This setup ensures that all the problems we test have only a few eigenvalues with large magnitudes. Recall that for a dataset $\mathbf{D} \in \mathbb{R}^{n\times d}$, if we denote by $\mathbf{x}_i$ the $i$-th data point, the Gaussian kernel matrix is defined as $\mathbf{A}_{ij} = f(\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/(2l^2)) + s\delta_{ij})$, where $\delta_{ij}$ is a Kronecker delta function.

In the first test, we uniformly sampled 1000 data points from a square area with side length $\sqrt{1000}$, setting $f = 0.2$, $l = 10$, $s = 0.01$, and $f = 0.2$, $l = 100$, $s = 0$ to generate two test matrices. The first test problem is not strictly numerically low-rank, as the eigenvalues only decay to 0.01, as illustrated in subplot (2,2) of Figure 5. The second problem has a faster decay to 0.0, as illustrated in subplot (4,2) of Figure 5. We compared three different combinations of algorithms: the classical Rayleigh-Ritz projection with QR, OFRR with QR, and OFRR with the Hessenberg process. We used MGS with re-orthogonalization to perform the QR factorization since it is the most accurate option. For all the tests on the first matrix, we set the subspace dimension $k = 50$, ran $m = 10$ iterations with a step size $iter = 3$, and reported the accuracy of the 20 largest eigenvalues. For the tests on the second matrix, since the eigenvalue decays faster, we set the subspace dimension $k = 20$, ran $m = 5$ iterations with a step size $iter = 2$, and reported the accuracy of the 6 largest eigenvalues.

As we can see from Figure 5, under double precision, all three methods achieve high accuracy; the two QR-based schemes are marginally superior because they employ orthonormal bases. Under single precision, both QR-based options exhibited relative errors larger than one. This deterioration arises from the loss of orthogonality in the single-precision QR basis, which introduces several spurious large Ritz values and shifts the remaining eigenvalue approximations by at least one index. Under half

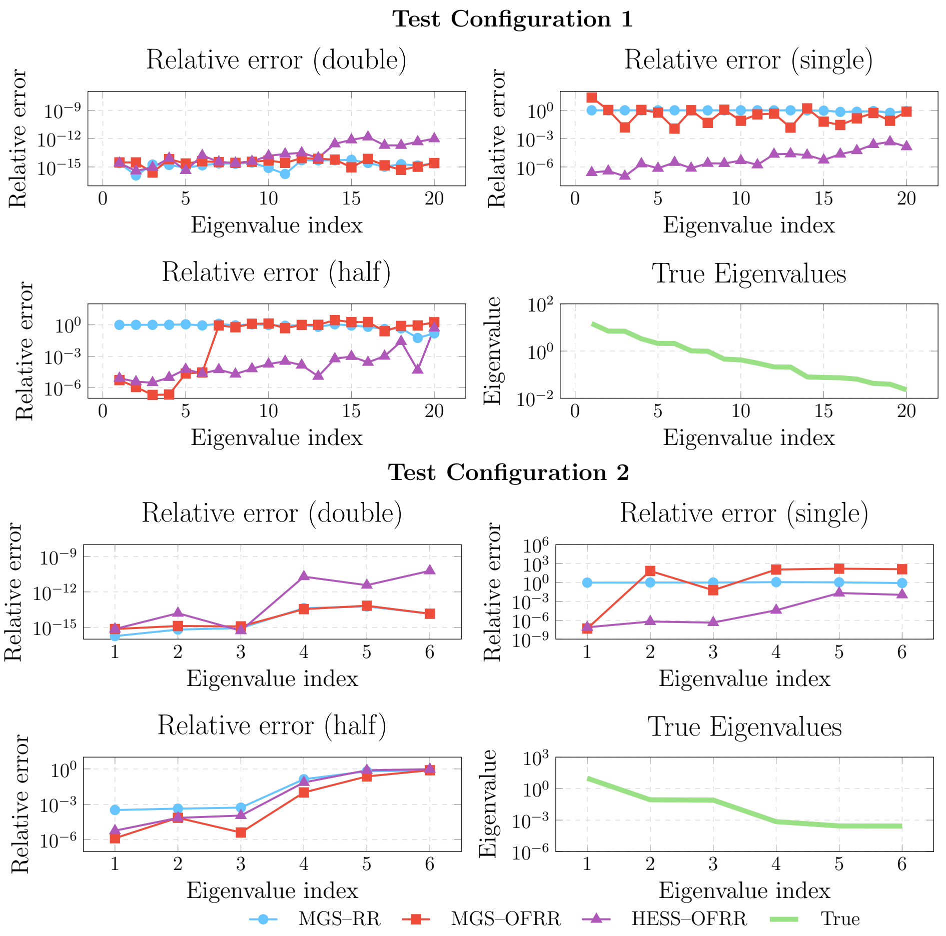

FIGURE 5. *Relative approximation accuracy using different algorithms with different precisions and true leading eigenvalues. The test matrices are Gaussian kernel matrices of size* $1000 \times 1000$ *with* $f = 0.2$, $l = 10$, $s = 0.01$ *(test configuration 1) and* $f = 0.2$, $l = 100$, $s = 0$ *(test configuration 2).*

precision, the use of a large tolerance $\varepsilon_{\mathtt{FP16}} \approx 9.77 \times 10^{-4}$ eliminates more columns in the basis construction process, so the accuracy of the two QR-based algorithms both improved in the second test configuration. Even so, the orthogonal Rayleigh-Ritz-based method is still less accurate than OFRR with the Hessenberg process. On the other hand, even OFRR with QR produces results comparable to those of OFRR with the Hessenberg process in the second test configuration, the Hessenberg variant is significantly more efficient.

**5.2.2. Eigenvalue problems with sparse matrices.** In the next set of experiments, we evaluated the performance of several Krylov subspace methods within the OFRR framework, using sparse matrices from the Suite-Sparse matrix collection [8]. Specifically, we compared three algorithmic combinations: the classical Rayleigh-Ritz projection with the Lanczos method, the OFRR with the Lanczos method, and the OFRR with the Krylov-Hessenberg process. Here, we use Lanczos with full orthogonalization, which is equivalent to the Arnoldi method for symmetric matrices. For tests with restart turned on, we always restarted with the single Ritz vector corresponding to the largest Ritz value. For these tests, we selected three matrices: `BCSSTK01`, `BCSSTK03`, and `1138_BUS`. Each matrix was scaled so that the largest eigenvalue is below 100 to avoid overflow in half precision. Key properties of these matrices are summarized in Table 2. Because the Krylov subspace might not contain all leading eigenvectors, direct comparison of the computed Ritz values against the exact leading eigenvalues is not meaningful. Instead, we assess the accuracy of computed approximate eigenpairs $(\lambda, \mathbf{v})$ by reporting the relative residual norm $\|\mathbf{A}\mathbf{v} - \lambda\mathbf{v}\|_2/|\lambda|$. For `BCSSTK01` we set the Krylov subspace dimension to 20 and report the 5 largest Ritz pairs. For `BCSSTK03` the dimension is 50 with 4 restarts, and we report the 10 largest Ritz pairs. For `1138_BUS` the dimension is 100 with 4 restarts, and we report the 20 largest Ritz pairs.

TABLE 2
*Matrices from the SuiteSparse matrix collection. Here, n is the matrix size, and* nnz *denotes the number of nonzeros.*

| | `BCSSTK01` | `BCSSTK03` | `1138_BUS` |
|---|---|---|---|
| $n$ | 48 | 112 | 1128 |
| nnz | 400 | 640 | 4054 |
| Application | Structural | Structural | Power Network |

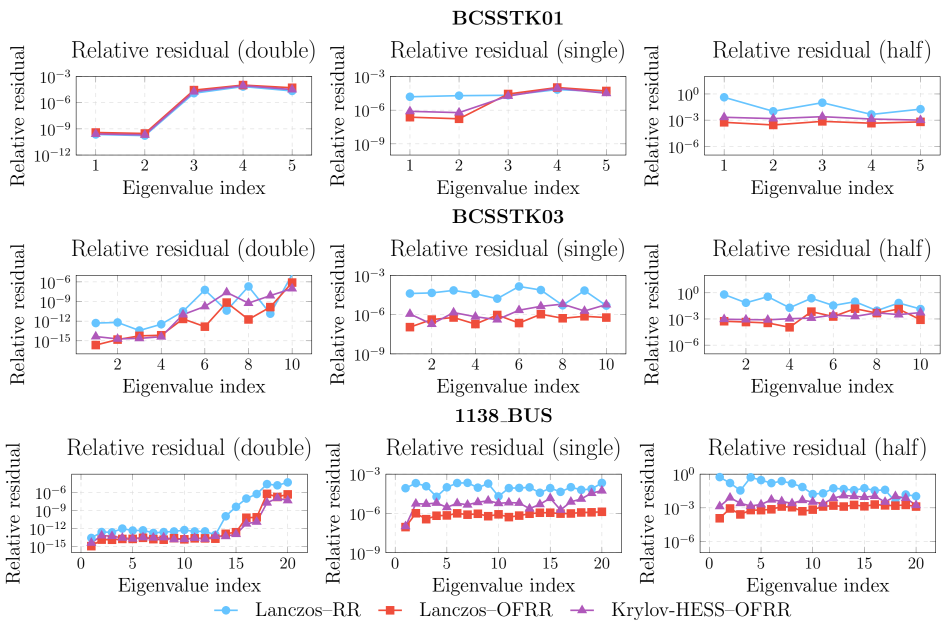

FIGURE 6. *Relative residual norm using different algorithms with different precisions. The test matrices are from SuiteSparse matrix collection.*

Figure 6 confirms the trend established in Section 5.2.1. All solvers achieve high accuracy in double precision. Consistent with results from previous half-precision tests, under single and half precision, the standard Rayleigh-Ritz method combined with Lanczos proved less accurate than the two methods utilizing the OFRR framework across all three test problems. Furthermore, comparing the two OFRR variants, the Hessenberg-based approach yielded an accuracy comparable to that of the Lanczos-based approach, reaffirming the benefits of using the Hessenberg process with OFRR in reduced precision.

**5.2.3. Singular value decomposition with kernel matrices.** In our next set of experiments, we tested the performance of OFRR on SVD with Gaussian kernel matrices. For the SVD experiments, we utilized the dataset comprising $n = 1000$ data points previously generated for the eigenvalue tests presented in Section 5.2.1. We denote this dataset as $\mathbf{D}_x \in \mathbb{R}^{n\times d}$. Subsequently, a second dataset, $\mathbf{D}_y \in \mathbb{R}^{m\times d}$ where $m = 200$, was created by randomly selecting $m$ points from $\mathbf{D}_x$ without replacement. We then constructed two $1000\times 200$ kernel matrices $\mathbf{A}$ defined by $\mathbf{A}_{ij} = f(\exp(-\|\mathbf{x}_i - \mathbf{y}_j\|_2^2/(2l^2)))$ for our test with $f = 0.2$, $l = 10$, and $f = 0.2$, $l = 100$. For both SVD test matrices, we performed $m = 10$ iterations with a step size $iter = 1$. For the first matrix, we used a subspace dimension of $k = 20$ and reported the accuracy of the 10 largest approximate singular values. For the second matrix, the subspace dimension was set to $k = 10$, and we reported the accuracy of the 5 largest approximate singular values.

The results for the SVD approximation tests are presented in Figure 7. In both double precision and single precision, all algorithmic approaches provided accurate approximations of the singular values, with the two QR-based methods exhibiting slightly higher accuracy due to the use of orthogonal bases. In half precision, the results continued to align with the primary trend observed in earlier experiments. The two methods based on the OFRR framework achieved better accuracy than the Rayleigh-Ritz-based method similar to the eigenvalue tests. Furthermore, within the OFRR framework, the Hessenberg variant yielded accuracy comparable to the QR variant in half precision. This similarity in the achieved accuracies, combined with the known computational advantages of the Hessenberg process, bolsters its appeal as a tool for OFRR in low-precision computations.

**5.3. Speedup.** Having validated the numerical accuracy using the `MATLAB` implementation, we now shift our focus to quantifying the computational performance of key algorithmic components. To this end, we leverage the C++/`CUDA` implementations developed specifically for execution on GPU architectures.

This section focuses on analyzing the performance of routines responsible for generating linearly independent bases, as this step is a core computational kernel in the OFRR framework. Whether OFRR is integrated with Arnoldi-type iterations or embedded in subspace iteration schemes for solving eigenvalue or SVD problems, the basis generation step remains the dominant performance-critical component. Therefore, we specifically compare the runtime performance of the following algorithms designed for this task: (i) Left-looking Modified Gram-Schmidt without re-orthogonalization (`MGS-L`); (ii) Right-looking Modified Gram-Schmidt (`MGS-R`); (iii) Left-looking Classical Gram-Schmidt without re-orthogonalization (`CGS`); (iv) Left-looking Classical Gram-Schmidt with re-orthogonalization (`CGS-2`); (v) Left-looking Hessenberg Process (`Hess-L`); (vi) Right-looking Hessenberg Process (`Hess-R`). Benchmarking the efficiency of these routines provides a direct and representative assessment of the overall performance of the OFRR framework, without the confounding effects of outer iteration strategies or back-end solvers, which typically rely on standard, highly optimized `BLAS` libraries.

The tests utilized three distinct fixed-size input matrices with entries drawn from the uniform distribution $\mathcal{U}(0,1)$ with dimensions $25000\times 200$, $50000\times 200$, and

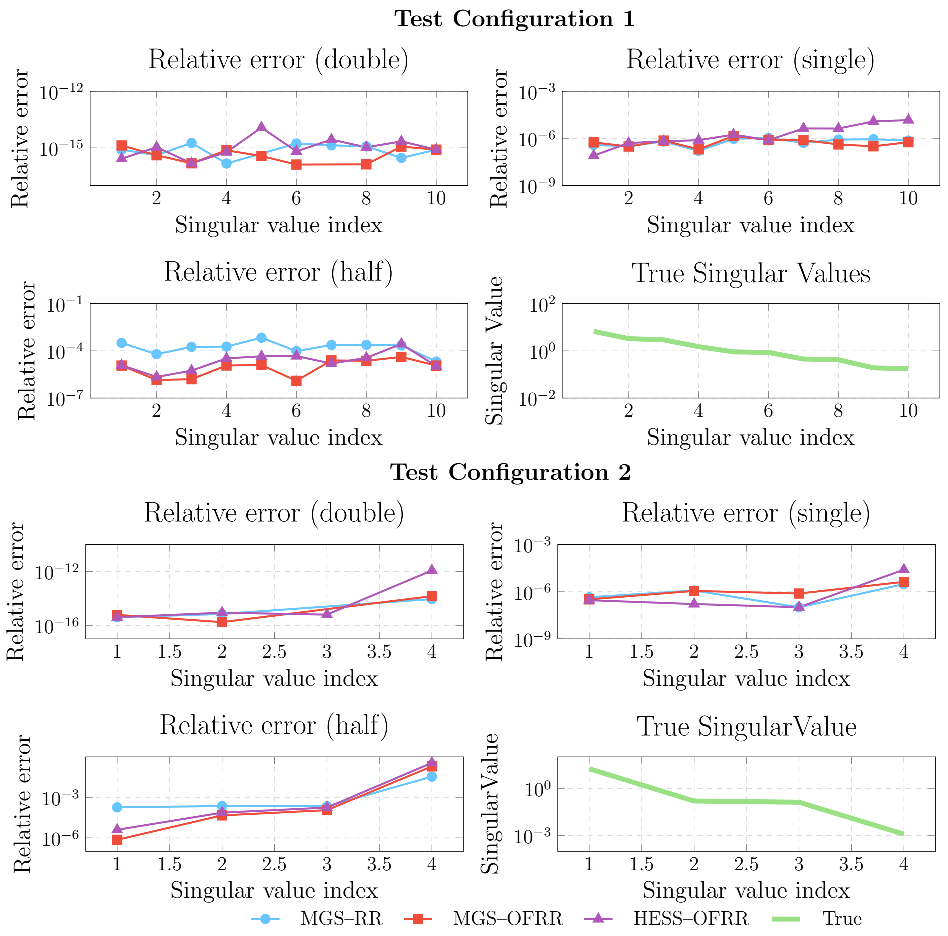

FIGURE 7. *Relative approximation accuracy using different algorithms with different precisions (left) and true leading singular values (right). The test matrices are Gaussian kernel matrices of size* $1000 \times 200$ *with* $f = 0.2$, $l = 10$ *(test configuration 1) and* $f = 0.2$, $l = 100$ *(test configuration 2).*

$50000 \times 400$, and were conducted under `FP64`, `FP32`, and `FP16` precision. For `FP16`, all computations were internally carried out in `FP32`. Each test was repeated five times, and the average runtime is reported.

Table 3 presents the runtimes (in milliseconds) of various basis generation methods across three matrix sizes and three floating-point precisions on GPU. We first observe a consistent performance advantage for right-looking algorithms ("-R") over their left-looking counterparts ("-L"). This is especially pronounced for MGS: `MGS-R` achieves more than $10\times$ speedup over `MGS-L` across all tested sizes, confirming that right-looking structures are significantly more GPU-friendly due to better memory access and data locality.

Comparing the Hessenberg process to MGS, we find that right-looking Hessenberg (`Hess-R`) achieves comparable or better performance than `MGS-R` in most configurations. For instance, at size $50000 \times 400$ under `FP32`, `Hess-R` completes in 77.9ms versus 99.6ms for `MGS-R`. This is particularly encouraging given that the Hessenberg imple-

Table 3
*Runtime (in milliseconds) of MGS (without re-orthogonalization), CGS, and Hessenberg process on GPU across multiple precisions and matrix sizes. Here, "-L" and "-R" denote left- and right-looking variants, respectively, and "`CGS-2`" indicates CGS with re-orthogonalization. For `FP16`, all computations were internally carried out in `FP32`.*

| Precision | Method | Matrix Dimensions ($m \times n$) | | |
|---|---|---|---|---|
| | | $25000 \times 200$ | $50000 \times 200$ | $50000 \times 400$ |
| `FP64` | `MGS-L` | 359.805 | 369.862 | 1402.376 |
| | `MGS-R` | 42.936 | 67.040 | 219.855 |
| | `CGS` | 46.322 | 65.882 | 182.747 |
| | `CGS-2` | 55.346 | 82.591 | 243.225 |
| | `Hess-L` | 119.293 | 132.390 | 480.391 |
| | `Hess-R` | 24.677 | 39.528 | 138.033 |
| `FP32` | `MGS-L` | 334.379 | 335.997 | 1331.892 |
| | `MGS-R` | 23.062 | 27.906 | 99.559 |
| | `CGS` | 14.824 | 21.894 | 72.658 |
| | `CGS-2` | 18.987 | 30.062 | 102.819 |
| | `Hess-L` | 111.796 | 121.182 | 446.195 |
| | `Hess-R` | 16.571 | 24.286 | 77.928 |
| `FP16` | `MGS-L` | 334.164 | 335.369 | 1328.612 |
| | `MGS-R` | 29.499 | 17.821 | 57.887 |
| | `CGS` | 12.971 | 15.894 | 47.782 |
| | `CGS-2` | 15.025 | 20.742 | 64.865 |
| | `Hess-L` | 111.314 | 120.955 | 441.900 |
| | `Hess-R` | 12.754 | 17.273 | 49.223 |

mentation relies on a custom GPU kernel, which has not yet been fully optimized. Further performance gains are expected with improved kernel-level optimizations, including memory fusion, architecture-aware block sizing, and better occupancy tuning.

The performance advantage of the Hessenberg process over MGS also holds for their left-looking variants. Across all tested sizes and precisions, `Hess-L` consistently outperforms `MGS-L`, with speedups ranging from 2–3× at larger problem scales. This improvement stems largely from the inner-product-free nature of the Hessenberg process. Moreover, while CGS is often considered a more efficient alternative to MGS in left-looking settings, its numerical instability under finite precision can be problematic. For instance, Figure 1 demonstrates a case where the instability of CGS impacts the condition number of the basis, highlighting the advantage of `Hess-L`. In such cases, `Hess-L` offers both better runtime and improved robustness.

To analyze scaling with respect to matrix dimensions, we compare cases with increasing $m$ and $n$. Doubling $m$ (e.g., $25000 \times 200$ to $50000 \times 200$) leads to negligible runtime growth for left-looking methods like `MGS-L`, reflecting the limited parallelism of reduction-based operations such as inner products. In contrast, doubling $n$ (e.g., $50000 \times 200$ to $50000 \times 400$) yields near 4× runtime increase, consistent with the $O(mn^2)$ cost.

Precision-wise, we observe meaningful runtime reductions from `FP64` to `FP32` and `FP16`, but the improvement is highly method-dependent. Right-looking methods benefit most from reduced precision, with `Hess-R` and `MGS-R` showing clear speedups. In contrast, left-looking methods, especially `MGS-L`, show little to no performance gain. For instance, `MGS-L` takes 334ms under both `FP32` and `FP16` at $25000 \times 200$, essentially unchanged from its `FP64` time. This can be attributed to the reliance on `BLAS` level-1 operations, which are limited by memory bandwidth and cannot exploit the arithmetic acceleration from lower-precision units.

## 6. Conclusion.

In this paper, we investigated the use of the non-orthogonal Rayleigh–Ritz projection method for computing selected eigenvalues and singular values under varying levels of numerical precision. Our study highlights the advantages of the Hessenberg process in low-precision settings, as an alternative to the Modified Gram–Schmidt (MGS) procedure. Specifically, the Hessenberg process demonstrates not only competitive accuracy but also improved efficiency in GPU implementation. While our current implementation of the Hessenberg process already delivers competitive performance, it also reveals untapped potential for further optimization—particularly through the development of custom GPU kernels. These findings position the Hessenberg-based OFRR framework as a promising direction for developing efficient and scalable eigensolvers on modern hardware.

## REFERENCES

[1] A. Abdelfattah, N. Beams, R. Carson, P. Ghysels, T. Kolev, T. Stitt, A. Vargas, S. Tomov, and J. Dongarra, *Magma: Enabling exascale performance with accelerated blas and lapack for diverse gpu architectures*, The International Journal of High Performance Computing Applications, 38 (2024), pp. 468–490.

[2] A. Abdelfattah, S. Tomov, and J. Dongarra, *Progressive optimization of batched lu factorization on gpus*, in 2019 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2019, pp. 1–6.

[3] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, *Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers*, Concurrency and Computation: Practice and Experience, 31 (2019), p. e4460.

[4] W. E. Arnoldi, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quarterly of applied mathematics, 9 (1951), pp. 17–29.

[5] D. Cai, E. Chow, and Y. Xi, *Posterior covariance structures in gaussian processes*, 2025, https://arxiv.org/abs/2408.07379, https://arxiv.org/abs/2408.07379.

[6] E. Carson and I. Daužickaitė, *Single-pass nyström approximation in mixed precision*, SIAM Journal on Matrix Analysis and Applications, 45 (2024), pp. 1361–1391.

[7] C.-C. Chang, P. Tsai, and C.-C. Lin, *Svd-based digital image watermarking scheme*, Pattern Recognition Letters, 26 (2005), pp. 1577–1586.

[8] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), 38 (2011), pp. 1–25.

[9] J. J. Dongarra, *Improving the accuracy of computed matrix eigenvalues*, tech. report, Argonne National Lab.(ANL), Argonne, IL (United States), 1980.

[10] J. J. Dongarra, *Improving the accuracy of computed singular values*, SIAM Journal on Scientific and Statistical Computing, 4 (1983), pp. 712–719.

[11] Q. Guo, C. Zhang, Y. Zhang, and H. Liu, *An efficient svd-based method for image denoising*, IEEE transactions on Circuits and Systems for Video Technology, 26 (2015), pp. 868–880.

[12] A. Haidar, H. Jagode, P. Vaccaro, A. Yarkhan, S. Tomov, and J. Dongarra, *Investigating power capping toward energy-efficient scientific applications*, Concurrency and Computation: Practice and Experience, 31 (2019), p. e4485.

[13] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 603–613.

[14] N. Halko, P.-G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM review, 53 (2011), pp. 217–288.

[15] M. Heyouni and H. Sadok, *A new implementation of the cmrh method for solving dense linear systems*, Journal of Computational and Applied Mathematics, 213 (2008), pp. 387–399.

[16] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, Acta Numerica, 31 (2022), pp. 347–414.

[17] H. Huang, T. Xu, Y. Xi, and E. Chow, *HiGP: A high-performance python package for gaussian process*, 2025, https://arxiv.org/abs/2503.02259.

[18] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, Journal of research of the National Bureau of Standards, 45 (1950), pp. 255–282.

[19] R. Li, Y. Xi, L. Erlandson, and Y. Saad, *The eigenvalues slicing library (evsl): Algorithms, implementation, and software*, SIAM Journal on Scientific Computing, 41 (2019), pp. C393–C415.

[20] F. Lopez and T. Mary, *Mixed precision lu factorization on gpu tensor cores: reducing data movement and memory footprint*, The International Journal of High Performance Computing Applications, 37 (2023), pp. 165–179.

[21] T. Ogita and K. Aishima, *Iterative refinement for symmetric eigenvalue decomposition*, Japan Journal of Industrial and Applied Mathematics, 35 (2018), pp. 1007–1035.

[22] T. Ogita and K. Aishima, *Iterative refinement for singular value decomposition based on matrix multiplication*, Journal of Computational and Applied Mathematics, 369 (2020), p. 112512.

[23] R. Ooi, T. Iwashita, T. Fukaya, A. Ida, and R. Yokota, *Effect of mixed precision computing on h-matrix vector multiplication in bem analysis*, in Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2020, pp. 92–101.

[24] B. N. Parlett and W. G. Poole, Jr, *A geometric theory for the qr, lu and power iterations*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 389–412.

[25] M. Petschow, E. S. Quintana-Ortí, and P. Bientinesi, *Improved accuracy and parallelism for mrrr-based eigensolvers—a mixed precision approach*, SIAM Journal on Scientific Computing, 36 (2014), pp. C240–C263.

[26] Y. Saad, *Numerical methods for large eigenvalue problems: revised edition*, SIAM, 2011.

[27] H. Sadok, *Cmrh: A new method for solving nonsymmetric linear systems based on the hessenberg reduction algorithm*, Numerical Algorithms, 20 (1999), pp. 303–321.

[28] N. Sahraneshinsamani, S. Catalán, and J. R. Herrero, *Mixed-precision pre-pivoting strategy for the lu factorization*, The Journal of Supercomputing, 81 (2025), p. 87.

[29] J. Shi, R. Li, Y. Xi, Y. Saad, and M. V. de Hoop, *Computing planetary interior normal modes with a highly parallel polynomial filtering eigensolver*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 894–906.

[30] J. Shi, R. Li, Y. Xi, Y. Saad, and M. V. de Hoop, *A non-perturbative approach to computing seismic normal modes in rotating planets*, Journal of Scientific Computing, 91 (2022), p. 67.

[31] Y. M. Tsai, P. Luszczek, and J. Dongarra, *Mixed-precision algorithm for finding selected eigenvalues and eigenvectors of symmetric and hermitian matrices*, in 2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH), IEEE, 2022, pp. 43–50.

[32] J. H. Wilkinson, *The algebraic eigenvalue problem*, Oxford University Press, Inc., 1988.

[33] Y. Xi, R. Li, and Y. Saad, *Fast computation of spectral densities for generalized eigenvalue problems*, SIAM Journal on Scientific Computing, 40 (2018), pp. A2749–A2773.

[34] T. Xu, V. Kalantzis, R. Li, Y. Xi, G. Dillon, and Y. Saad, *pargemslr: A parallel multilevel schur complement low-rank preconditioning and solution package for general sparse matrices*, Parallel Computing, 113 (2022), p. 102956.

[35] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, *Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration*, Phy. rev. E, 74 (2006), p. 066704.

*This manuscript is for review purposes only.*