MIXED PRECISION ITERATIVE REFINEMENT FOR LOW-RANK MATRIX AND TENSOR APPROXIMATIONS[*]

MARC BABOULIN[†], OGUZ KAYA[†], THEO MARY[‡], AND MATTHIEU ROBEYNS[†,§]

Abstract. We present a new mixed precision algorithm to compute low-rank matrix and tensor approximations, a fundamental task in numerous applications in scientific computing and data analysis. Our algorithm is reminiscent of the iterative refinement framework for linear systems: we first compute a low-rank approximation in low precision and then refine its accuracy by iteratively updating it. We carry out an error analysis of our algorithm which proves that we can reach a high accuracy while performing most of the operations in low precision. We measure the computational cost of the algorithm, which depends on the numerical rank of the input (matrix or tensor) as well as the speed ratio between low and high precision arithmetic. We identify two situations where our method has a strong potential: when the hardware provides fast low precision matrix multiply–accumulate units, and when the numerical rank of the input is small at low accuracy levels. We confirm experimentally the potential of our algorithm for computing various low-rank matrix and tensor decompositions such as SVD, QR, Tucker, hierarchical Tucker, and tensor-train.

Key words. Matrix and tensor computations, low-rank approximations, mixed precision algorithms, iterative refinement, randomized SVD, tensor decompositions.

AMS subject classifications. 65F55, 65G50, 65Y20, 15A69

1. Introduction.

Low-rank approximations (LRA) are a powerful tool used in many scientific applications to reduce the dimension of large scale data. For example, an $n \times n$ matrix X may be approximated by a low-rank product UV^T of $n \times r$ matrices U and V, reducing the storage cost from $O(n^2)$ to $O(nr)$. This storage cost is even more critical for tensors [22, 8, 16], the higher dimensional generalization of matrices: a dth order tensor X requires storage in $O(n^d)$ that is exponential in the number of dimensions d. This *curse of dimensionality* can be tackled via LRA by decomposing the full tensor as a product of tensors of lower order and lower rank. Various such decompositions exist, such as the Tucker [31, 11], the tensor-train (TT) [29], and the hierarchical Tucker (HT) [17, 15] formats.

However, computing matrix or tensor low-rank decompositions at a controlled approximation accuracy ε is computationally expensive; it represents the bottleneck of many LRA-based applications. Therefore, developing high performance algorithms for computing LRA is an important problem that has been the object of many studies; see, for instance, these surveys on matrices [21] and tensors [22, 16, 8].

Our work is specifically motivated by the recent and rapid emergence of *low precision arithmetics*, in particular half precision floating-point arithmetics such as the IEEE fp16 and bfloat16 formats. These low precision arithmetics provide great computational benefits, in terms of storage, speed, and energy [19]. On modern hardware, and in particular on Graphics Processing Unit (GPU) accelerators, low precision arithmetics can be orders of magnitude faster than the standard single (fp32) or double (fp64) precision arithmetics. However, using low precision also degrades the accuracy of the computations; for example half precision arithmetics provide at best between 3 and 4 digits of accuracy, depending on the format. This motivates

[*]Version of May 19, 2025.

[†]University Paris-Saclay, CNRS, LISN, Gif-sur-Yvette, France, (marc.baboulin,oguz.kaya, matthieu.robeyns@universite-paris-saclay.fr)

[‡]Sorbonne Université, CNRS, LIP6, Paris, France, (theo.mary@lip6.fr)

[§]Corresponding author.

the need for *mixed precision algorithms*, which combine multiple precision formats with the goal of achieving the high performance of the low precision while preserving the high accuracy of the higher precision. The emergence of low precision on modern hardware has generated much recent interest in mixed precision algorithms, with many successful examples in numerical linear algebra; see the survey [19] for an overview of this field.

In this paper, we seek to develop mixed precision algorithms for computing LRA. Contrary to other linear algebra routines such as the solution of linear systems, there has been relatively little work on designing mixed (or even low) precision algorithms for LRA. Amestoy et al. [2] describe a mixed precision matrix LRA that partitions the low-rank factors into several block columns stored in different precisions depending on the singular values of the matrix; this approach can make use of low precisions for matrices with rapidly decaying singular values. A similar approach is proposed by Ooi et al. [25] for $\mathcal{H}$-matrices. Connolly et al. [9] perform the rounding error analysis of the randomized singular value decomposition (SVD) for matrix LRA, in particular for the adaptive algorithm of Martinsson and Voronin [24] which builds an LRA by increasing its rank until the prescribed accuracy is reached. The analysis in [9] reveals that lower precision can be used when the norm of the approximation error becomes small enough; this once more corresponds to the case where the matrix has rapidly decaying singular values. Finally, a recent paper of Ootomo and Yokota [27] also proposes a mixed precision randomized SVD; their approach stores the random projection matrix in half precision while keeping the input matrix in single precision, and obtains speedup by emulating single precision computations with GPU tensor cores [13, 26]. There have also been very few attempts to develop mixed precision LRA for tensors. We can mention a recent work by Yang et al. [33] that proposes an iterative CP decomposition using mixed precision stochastic gradient descent. In a context different from LRA, the recent work of Agullo et al. [1] is worth mentioning: they consider the solution of linear systems $Ax = b$ via GMRES, where A can be approximated under tensor format [10]. This approach can then benefit from a mixed precision implementation of GMRES.

In this paper, we propose a new method for computing LRA in mixed precision arithmetic. Our approach is applicable to basically any LRA algorithm, involving either matrices or tensors. It is reminiscent of the iterative refinement framework used for solving linear systems: the idea is to first compute an LRA in low precision, then evaluate the error (or residual) from this first LRA, and re-apply the same LRA kernel to this error term to obtain a correction term that is used to refine the accuracy of the LRA. This can be repeated iteratively to reach any level of desired accuracy. The refined LRA is obtained as the sum of the original low precision LRA and the correction term, and is thus of larger yet still of low-rank. In order to contain the rank growth and maintain the optimal rank throughout the iterations, our method employs a "recompression" strategy that is performed in high precision but whose cost stays asymptotically smaller than that of LRA.

We carry out an error analysis of our method based on a general parameterized error model that only assumes that we have numerically stable implementations of the basic kernels used in our algorithm (LRA, matrix multiplication, and recompression). For matrices, many instances of such kernels are known to be stable. For tensors, the numerical stability of most tensor computations is still an open question, but extensive experiments certainly support the idea that they are indeed stable. Under this model, we prove that the method can reach a high accuracy while performing most of the operations in low precision. In particular, we show that the precision used for the

LRA kernel—which is the computational bottleneck of the whole method—only affects the convergence speed of the process, but not its attainable accuracy. Moreover, the method's convergence speed is independent of the condition number of the matrix or tensor, which makes it widely applicable. In order to assess under which conditions we can expect our method to be beneficial, we perform a complexity analysis that measures the cost of the method as a function of the numerical rank of the input as well as the speed ratio between the low and high precision arithmetic. We identify two situations where our method has a strong potential. The first is when the hardware provides fast low precision matrix multiply–accumulate units, also called block FMA units [6], which allow for computing the low precision LRA at very high speed. The second is when the numerical rank of the input is small at low accuracy levels, which means that the singular values of the matrix or tensor are rapidly decaying; in this case, the first iterations of our method becomes inexpensive.

We apply our method to various low-rank matrix and tensor decompositions: SVD, QR, Tucker, HT, and TT. We perform some MATLAB experiments to confirm that our method is able to compute an LRA at any desired level of accuracy, while mostly using the low precision arithmetic.

The rest of this paper is organized as follows. In section 2, we describe our main algorithm of iterative refinement for LRA and provide the corresponding error and complexity analysis. Then we apply this method to various LRA algorithms, namely QRCP and randomized SVD for matrices, and HOSVD, TTSVD, and HTSVD for tensors in section 3 and section 4, respectively. We validate our method experimentally in section 5. Finally, we provide concluding remarks in section 6.

1.1. Notations. We denote as X the object of interest, a matrix or a tensor, and as F its low-rank factors that we seek to compute. If $X \in \mathbb{R}^{m\times n}$ is a matrix, then $F = UV^T$ is usually implicitly represented by the product of two matrices $U \in \mathbb{R}^{m\times r}$, $V \in \mathbb{R}^{n\times r}$ of smaller dimensions, where r is the rank of F. Similarly, if $X \in \mathbb{R}^{n_1\times\cdots\times n_d}$ is a tensor of order d, its low-rank factors F are represented using a number of matrices and/or tensors of smaller order; the precise expression of F depends on the chosen format. In particular, in this article we will consider three different low-rank tensor decompositions:

- the Tucker format [11], which uses d matrices $U^{(1)} \in \mathbb{R}^{n_1\times r_1}$, …, $U^{(d)} \in \mathbb{R}^{n_d\times r_d}$, where $r_i \le n_i$, and a dth-order core tensor $G \in \mathbb{R}^{r_1\times\cdots\times r_d}$;
- the TT format [29], which uses d 3rd-order tensors $U^{(1)} \in \mathbb{R}^{r_0\times n_1\times r_1}$, $U^{(2)} \in \mathbb{R}^{r_1\times n_2\times r_2}$, …, $U^{(d)} \in \mathbb{R}^{r_{d-1}\times n_d\times r_d}$, where $r_0 = r_d = 1$;
- the HT format [17, 15], which recursively applies the Tucker format to the core tensor G to obtain a binary tree of tensors. Leaf nodes of the tree are matrices $U^{(1)} \in \mathbb{R}^{n_1\times r_{d-1}}$, …, $U^{(d)} \in \mathbb{R}^{n_d\times r_{2d-2}}$ and the rest are 3rd-order tensors $G^{(1)} \in \mathbb{R}^{r_0\times r_1\times r_2}, \dots, G^{(d-1)} \in \mathbb{R}^{r_d\times r_{2d-3}\times r_{2d-2}}$, where $G^{(1)}$ is the root and $r_0 = 1$.

Note that our main iterative refinement method described in the next section does not require any specific knowledge or property of the tensor decomposition, and simply denotes its low-rank factors as F.

Note that the low-rank factors F are a structured (compressed) object representing a matrix or tensor of the same order and outer dimensions as X. Therefore, the subtraction $X - F$ is well-defined; since X is uncompressed, evaluating the result of this subtraction requires to decompress F. Given two factors F_1 and F_2 representing matrices or tensors of the same order and outer dimensions, the addition $F_1 + F_2$ is also well-defined; since both F_1 and F_2 are compressed, the result of this addition can be formed by using low-rank arithmetic, that is, can be represented by a low-rank factor $F = F_1 + F_2$ with larger ranks.

To control the accuracy of the LRA, we will distinguish two types of parameters, denoted as ε and u. The truncation threshold ε controls the low-rank truncation error and thus the accuracy of the LRA in exact arithmetic. The unit roundoff u controls the rounding errors and thus the additional loss of accuracy incurred by the use of finite precision arithmetic. Actually, in our iterative refinement method, we will have four parameters ε, u, ε_ℓ, and u_ℓ, where ε_ℓ and u_ℓ denote the truncation threshold and unit roundoff used by the low precision LRA, whereas ε and u denote the truncation threshold and unit roundoff used by the rest of the operations performed in high precision, and correspond to the final target accuracy.

For a matrix X, we denote by $\textsc{Rank}(X, \varepsilon)$ the numerical rank of X at accuracy ε, which is the smallest integer r_ε such that there exists low-rank factors F of rank r_ε satisfying a relative accuracy ε, that is, $\|X - F\| \le \varepsilon\|X\|$ for a given choice of norm. We also define $\textsc{Rank}(X)$, the non-numerical (or exact) rank of a matrix X. This definition of the numerical rank extends to the Tucker, TT, and HT representations of tensors by defining $\textsc{Rank}(X, \varepsilon)$ and $\textsc{Rank}(X)$ as the vector of length $\bar{d}$ whose coefficients correspond to the "inner" dimensions connecting the underlying matrices and/or tensors in the decomposition. The length $\bar{d}$ depends on both d and the type of decomposition:

$$\bar{d} = \begin{cases} d & \text{(Tucker)} \\ d-1 & \text{(TT)} \\ 2d-2 & \text{(HT)} \end{cases} . \tag{1.1}$$

Technically, the rank vector is determined by a vector of error tolerances ε_i, $i = 1\colon \bar{d}$, whose sum yields the target accuracy $\varepsilon = \sum_{i=1}^{\bar{d}} \varepsilon_i$. In practice, the truncation error is uniformly distributed by setting $\varepsilon_i = \varepsilon/\bar{d}$ [29, 32, 15].

2. Iterative refinement for low-rank approximation.

In this section, we propose a mixed precision algorithm to compute LRA at high accuracy while performing most of the computations in low precision.

We consider the problem of computing low-rank factors F of a matrix or tensor X satisfying a relative accuracy ε, that is, $\|X - F\| \le \varepsilon\|X\|$. In finite precision arithmetic, we must carefully select the precision in order to achieve this accuracy. In a uniform precision context (where we perform all computations in the same precision), we must use a precision whose unit roundoff u is safely smaller than ε, that is, $u = \theta\varepsilon$, with $\theta \le 1$ some parameter. Indeed, we may then expect the computed factors $\widehat{F}$ to satisfy

$$\|X - \widehat{F}\| \le (\varepsilon + cu)\|X\| = (1 + c\theta)\varepsilon\|X\|$$

for some constant c, where the term $c\theta$ accounts for the effect of rounding errors and can be made as small as needed by decreasing θ (decreasing u, that is, increasing the precision).

For applications requiring relatively high accuracy (small values of ε), this uniform precision approach therefore cannot make use of lower precisions. This motivates us to propose a mixed precision method, outlined below, which uses low precision to compute a first approximate set of factors F_0, and then refines them into more accurate factors F_1 as follows.

1. Compute low-rank factors F_0 of X at low accuracy ε_ℓ and in low precision $u_\ell = \theta\varepsilon_\ell$.
2. Compute the error $E = X - F_0$ in high precision u.

3. Compute low-rank factors F_E of E at low accuracy ε_ℓ and in low precision $u_\ell = \theta\varepsilon_\ell$.
4. Update the low-rank factors of X to $F_1 = F_0 + F_E$ in high precision u.

This approach is based on the observation that, while the first factors F_0 will achieve a low accuracy ε_ℓ relative to $\|X\|$,

$$\|X - F_0\| \le \varepsilon_\ell \|X\|,$$

the factors F_E of the error E will achieve a low accuracy ε_ℓ relative to $\|E\|$,

$$\|E - F_E\| \le \varepsilon_\ell \|E\|.$$

Neglecting for now the effect of rounding errors, we have $\|E\| = \|X - F_0\| \le \varepsilon_\ell \|X\|$, and therefore the combined factors $F_0 + F_E$ will satisfy an accuracy of about ε_ℓ^2:

$$\|X - F_1\| = \|X - F_0 - F_E\| = \|E - F_E\| \le \varepsilon_\ell \|E\| \le \varepsilon_\ell \|X - F_0\| \le \varepsilon_\ell^2 \|X\|.$$

This idea is therefore based on computing additive updates $F_0 + F_E$ in mixed precision. This is reminiscent of the iterative refinement for linear systems $Ax = b$, which is also based on additive updates of the solution x. Indeed, for $Ax = b$, step 1 corresponds to computing an initial solution x_0, step 2 corresponds to computing the residual $r = b - Ax_0$, step 3 corresponds to solving another linear system $Ad = r$ for a correction term d, and step 4 updates $x_1 = x_0 + d$. The term "iterative refinement" is most commonly used for the solution of linear systems, but is also found in least squares, singular value, or eigenvalue problems. Therefore, we believe that the method proposed in this article should be called "iterative refinement for LRA".

One of the crucial insights that makes this method effective is that the error E has a low numerical rank whenever X has one too, since it is the sum of X and F_0, which is low-rank by construction. To be specific, we will prove in subsection 2.2 that $\textsc{Rank}(E, \varepsilon_\ell)$ is at most $2\,\textsc{Rank}(X, \varepsilon_\ell^2)$ and, for the same reason, the rank of the refined factors F_1 is bounded by $3\,\textsc{Rank}(X, \varepsilon_\ell^2)$. The factors F_1 can then be cheaply recompressed into factors of optimal rank $\textsc{Rank}(X, \varepsilon_\ell^2)$ in high precision u by exploiting their (suboptimal) low-rank structure.

If accuracy higher than ε_ℓ^2 is needed, we can apply the method again on F_1 to obtain an accuracy of ε_ℓ^3, and so on; we obtain Algorithm 2.1, which repeats this process until the desired accuracy is achieved.

Algorithm 2.1 is described within a general iterative refinement framework that uses an arbitrary LRA algorithm: $\textsc{Lra}(X, \varepsilon)$ returns low-rank factors F of X at accuracy ε. We also require a $\textsc{Decompress}$ kernel which transforms low-rank factors F back to a full-size object, and a $\textsc{Recompress}$ kernel which takes low-rank factors F and ε as input, and computes their optimal LRA at accuracy ε (that is, the LRA of the smallest rank that achieves an accuracy of at least ε). We consider these three abstract kernels in order for Algorithm 2.1 to be as general as possible. In particular, this will allow us to apply it to different LRA methods, both for matrices and tensors. However, to be concrete, we will also discuss specific examples in sections 3 and 4. Namely, we will consider five different $\textsc{Lra}$ kernels: QRCP and randomized SVD for matrices, and HOSVD, TTSVD, and HTSVD for tensors. For matrices, the $\textsc{Decompress}$ kernel is simply a matrix–matrix product; for tensors, it is similarly a contraction of the low-rank tensor along all inner dimensions. Finally, the $\textsc{Recompress}$ kernel depends on the $\textsc{Lra}$ kernel: for each case, we will show how to adapt the $\textsc{Lra}$ kernel (which takes as input a full object) to obtain the $\textsc{Recompress}$ kernel (which takes as input a low-rank object); see sections 3 and 4 for more details.

Algorithm 2.1 Iterative refinement for LRA.

```
Input:  $X$, the matrix or tensor of interest;
        $\varepsilon$, the desired relative accuracy for the approximation;
        $n_{\text{it}}$, the maximum number of iterations;
        LRA, a low-rank approximation algorithm;
        DECOMPRESS, RECOMPRESS: decompression and recompression algorithms.
Output: $F$, the low-rank factors $X$.
 1: Compute $F = \textsc{Lra}(X, \varepsilon_\ell)$ in precision $u_\ell$.
 2: for $i = 1$ to $n_{\text{it}}$ do
 3:     Compute $E = X - \textsc{Decompress}(F)$ in precision $u$.
 4:     Compute $\alpha = \|E\|$ in precision $u$.
 5:     If $\alpha \le \varepsilon \|X\|$, exit.
 6:     Scale $E \leftarrow \alpha^{-1} E$ in precision $u$.
 7:     Compute $F_E = \textsc{Lra}(E, \varepsilon_\ell)$ in precision $u_\ell$.
 8:     Scale back $F_E \leftarrow \alpha F_E$ in precision $u$.
 9:     Compute $F = \textsc{Recompress}(F + F_E, \varepsilon)$ in precision $u$.
10: end for
```

Algorithm 2.1 incorporates two additional steps to make the method effective. First, the error E is scaled by the inverse of its norm (Line 6) before computing its low-rank factors F_E, and then scaled back after (Line 8). This is done to prevent the elements of E from underflowing when converted to the lower precision, in the case where the arithmetic uses not only a reduced number of bits in the significand but also in the exponent, such as is the case for the IEEE half precision fp16 arithmetic. We also note that the elements of X itself are also susceptible to underflow or overflow; in this case, a similar scaling should be applied to X. Second, the factors F are recompressed (Line 9) to their optimal low-rank representation to prevent the rank from growing out of control during the iterations. Indeed, as mentioned above, the rank of F is bounded by $3\,\textsc{Rank}(X, \varepsilon_\ell^2)$ after one iteration. Therefore, in the absence of recompression, the rank would grow as $3^k\,\textsc{Rank}(X, \varepsilon_\ell^2)$ after k iterations, and quickly make the method unaffordable. This RECOMPRESS kernel can be implemented for an asymptotically negligible cost for many LRA algorithms, including all those considered in this article.

Algorithm 2.1 has four parameters that control its accuracy: ε, ε_ℓ, u, and u_ℓ.

- ε indicates the target accuracy for the final factors, and is prescribed by the user as input to Algorithm 2.1.
- u is the unit roundoff of the high precision, which should be taken to be the lowest possible such that u is still safely smaller than ε: $u = \theta\varepsilon$, $\theta \le 1$.
- u_ℓ is the unit roundoff of the low precision, which is used to perform the most expensive parts of the computation, the calls to LRA. Its choice depends on both the available arithmetics on the target hardware, and the target accuracy ε. Indeed, lowering the precision makes each iteration faster but requires more of them.
- Finally, ε_ℓ is the tolerance used by the LRA kernel; since LRA is performed in precision u_ℓ, ε_ℓ should be set such that u_ℓ is safely smaller than ε_ℓ, that is, $\varepsilon_\ell = u_\ell/\theta_\ell$, $\theta_\ell \le 1$. We note that this is necessary because using an ε_ℓ too close to u_ℓ prevents most LRA algorithms from reliably detecting the correct rank, due to the noise introduced by rounding errors.

In the rest of this section, we first perform an error analysis to determine the attainable accuracy and convergence rate of Algorithm 2.1. We then perform a complexity analysis to determine under which conditions the algorithm can be expected to be faster than a standard uniform precision Lra performed entirely in high precision u.

2.1. Error analysis. In order to carry out the error analysis of Algorithm 2.1, we will use the standard model of floating-point arithmetic [18, sect. 2.2]. In addition, we also need to make the following three assumptions on the numerical behavior of the Lra, Decompress, and Recompress kernels. First, we assume that computing $F = \textsc{Lra}(X, \varepsilon_\ell)$ in precision u_ℓ yields computed factors $\widehat{F}$ satisfying

$$\|X - \widehat{F}\| \le (\varepsilon_\ell + b_1 u_\ell)\|X\| = (1 + b_1\theta_\ell)\varepsilon_\ell\|X\|. \tag{2.1}$$

Second, we assume that computing $F = \textsc{Decompress}(F)$ in precision u yields computed factors $\widehat{F}$ satisfying

$$\|F - \widehat{F}\| \le b_2 u\|F\| = b_2\theta\varepsilon\|F\|. \tag{2.2}$$

Third, we assume that computing $F = \textsc{Recompress}(F, \varepsilon)$ in precision u yields computed factors $\widehat{F}$ satisfying

$$\|F - \widehat{F}\| \le (\varepsilon + b_3 u)\|F\| = (1 + b_3\theta)\varepsilon\|F\|. \tag{2.3}$$

In (2.1)–(2.3), the error bounds are parameterized by the constants b_1, b_2, and b_3, which will usually depend on the dimensions of X.

For simplicity, we ignore any rounding errors associated with the scaling by $\alpha = \|E\|$, which are negligible and can in fact be prevented by rounding α to the nearest power of two (in binary floating-point arithmetic).

We are now ready to prove the following result.

Theorem 2.1. *If Algorithm 2.1 is applied to X with Lra, Decompress, and Recompress kernels satisfying (2.1)–(2.3), then after k iterations, the computed factors $\widehat{F}$ satisfy*

$$\|X - \widehat{F}\| \le (\phi^{k+1} + \xi + O(\varepsilon_\ell\varepsilon))\|X\|, \tag{2.4}$$

with $\phi = (1 + b_1\theta_\ell)\varepsilon_\ell + (b_2 + 1)\theta\varepsilon$ and $\xi = \big(1 + (b_2 + b_3 + 2)\theta\big)\varepsilon$.

Proof. Defining $\widehat{F}_i$ as the computed factors after i iterations, our goal is to obtain a bound of the form

$$\|X - \widehat{F}_{i+1}\| \le \phi\|X - \widehat{F}_i\| + \xi\|X\|, \quad \phi < 1,$$

which will allow us to conclude that the error contracts by a factor ϕ at each iteration, until it converges to its maximum attainable accuracy ξ.

Given $\widehat{F}_i$, the first step is to decompress it and compute E at Line 3; by (2.2), the computed $\widehat{E}$ satisfies

$$\|X - \widehat{F}_i - \widehat{E}\| \le \theta\varepsilon\big((b_2 + 1)\|\widehat{F}_i\| + \|X\|\big), \tag{2.5}$$

with an extra term $\theta\varepsilon(\|\widehat{F}_i\| + \|X\|)$ on the right-hand side coming from the rounding error incurred by the subtraction. Using the triangle inequality $\|\widehat{F}_i\| \le \|X - \widehat{F}_i\| + \|X\|$, we can rearrange (2.5) as

$$\|X - \widehat{F}_i - \widehat{E}\| \le \theta\varepsilon\big((b_2 + 1)\|X - \widehat{F}_i\| + (b_2 + 2)\|X\|\big). \tag{2.6}$$

Then we compute $\textsc{Lra}(\widehat{E}, \varepsilon_\ell)$ at Line 7; by (2.1) the computed $\widehat{F}_E$ satisfies

$$\|\widehat{E} - \widehat{F}_E\| \le (1 + b_1\theta_\ell)\varepsilon_\ell \|\widehat{E}\|. \tag{2.7}$$

By using (2.6) and the triangle inequality, we rearrange (2.7) as

$$\begin{aligned} \|\widehat{E} - \widehat{F}_E\| &\le (1 + b_1\theta_\ell)\varepsilon_\ell \big(\|X - \widehat{F}_i\| + \|X - \widehat{F}_i - \widehat{E}\|\big) && (2.8)\\ &= (1 + b_1\theta_\ell)\varepsilon_\ell \|X - \widehat{F}_i\| + O(\varepsilon_\ell\varepsilon), && (2.9) \end{aligned}$$

where we do not keep track explicitly of high order terms in $O(\varepsilon_\ell\varepsilon)$ for the sake of readability. Finally, we obtain the next iterate F_{i+1} by recompressing $\widehat{F}_i + \widehat{F}_E$ at Line 9; by (2.3), the computed $\widehat{F}_{i+1}$ satisfies

$$\|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| \le (1 + b_3\theta)\varepsilon \|\widehat{F}_i + \widehat{F}_E\|. \tag{2.10}$$

By using (2.6), (2.9), and the triangle inequality, we rearrange (2.10) as

$$\begin{aligned} \|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| &\le (1 + b_3\theta)\varepsilon \big(\|X - \widehat{F}_i - \widehat{E}\| + \|\widehat{E} - \widehat{F}_E\| + \|X\|\big) && (2.11)\\ &= (1 + b_3\theta)\varepsilon\|X\| + O(\varepsilon_\ell\varepsilon). && (2.12) \end{aligned}$$

Using the triangle inequality together with (2.6), (2.9), and (2.12), we obtain

$$\begin{aligned} \|X - \widehat{F}_{i+1}\| &= \|X - \widehat{F}_i - \widehat{E} + \widehat{E} - \widehat{F}_E + \widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| && (2.13)\\ &\le \|X - \widehat{F}_i - \widehat{E}\| + \|\widehat{E} - \widehat{F}_E\| + \|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| && (2.14)\\ &\le \phi\|X - \widehat{F}_i\| + \xi\|X\| + O(\varepsilon_\ell\varepsilon), && (2.15) \end{aligned}$$

with

$$\phi = (1 + b_1\theta_\ell)\varepsilon_\ell + (b_2 + 1)\theta\varepsilon \tag{2.16}$$

and

$$\xi = \big(1 + (b_2 + b_3 + 2)\theta\big)\varepsilon. \tag{2.17}$$

Noting that the first factors $\widehat{F}_0$ computed at Line 1 satisfy by (2.1)

$$\|X - \widehat{F}_0\| \le (1 + b_1\theta_\ell)\varepsilon_\ell\|X\| \le \phi\|X\| \tag{2.18}$$

concludes the proof. ∎

Theorem 2.1 states that the approximation error $\|X - \widehat{F}\|$ contracts by a factor $\phi = O(\varepsilon_\ell)$ at each iteration, until it reaches its maximum attainable accuracy $\xi = O(\varepsilon)$. Thus, after k iterations, the error is of order $\varepsilon_\ell^{k+1} + \varepsilon$, which means that we can actually estimate in advance approximately how many iterations are needed to achieve the desired accuracy ε (up to constants):

$$n_{\text{it}} = \log(\varepsilon)/\log(\varepsilon_\ell) - 1. \tag{2.19}$$

It is worth noting that, unlike iterative refinement for linear systems, there is no dependence on the condition number of X and thus the only condition for Algorithm 2.1 to converge is that $\phi < 1$, which should always be the case as long as θ_ℓ is sufficiently small. This is explained by the fact that the speed of convergence of Algorithm 2.1 depends on the backward error $\|A - XY^T\|$, rather than the forward one. Therefore, Algorithm 2.1 is extremely general: it can be applied to any matrix or tensor of low numerical rank and it can make use of potentially very low precisions.

From a numerical perspective, Algorithm 2.1 is therefore quite appealing. We next discuss under which conditions it is also attractive from a computational perspective.

2.2. Complexity analysis. The cost of Algorithm 2.1 will mainly depend on two factors: the relative speed for computing in different precisions on the target hardware, and the numerical ranks of the objects encountered during the iterations (X, E, and F).

We begin by bounding these ranks solely as a function of the numerical ranks of X at given accuracies. In order to do so, we will need to bound the numerical rank of $A+B$ as a function of that of A and B by using the following lemma. In what follows, recall that when X is a tensor, $\textsc{Rank}(X, \varepsilon)$ is a vector (see subsection 1.1) and all the (in)equalities involving this quantity should be interpreted componentwise.

Lemma 2.2.

$$\textsc{Rank}(A+B, \varepsilon) \le \textsc{Rank}(A, \frac{\varepsilon\|A+B\|}{2\|A\|}) + \textsc{Rank}(B, \frac{\varepsilon\|A+B\|}{2\|B\|}). \tag{2.20}$$

Proof. For the purpose of this proof only, we introduce an alternative definition of numerical rank that measures accuracy in an absolute sense, rather than a relative one. Let $\textsc{Rankabs}(X, \varepsilon)$ be the the smallest integer r_ε such that there exists F of rank r_ε satisfying $\|X-F\| \le \varepsilon$. The $\textsc{Rankabs}$ operator satisfies

$$\textsc{Rankabs}(X, \varepsilon\|X\|) = \textsc{Rank}(X, \varepsilon)$$

and

$$\textsc{Rankabs}(A+B, \varepsilon) \le \textsc{Rankabs}(A, \varepsilon/2) + \textsc{Rankabs}(B, \varepsilon/2)$$

by the triangle inequality. Therefore, we have that

$$\begin{aligned} \textsc{Rank}(A+B, \varepsilon) &= \textsc{Rankabs}(A+B, \varepsilon\|A+B\|) \\ &\le \textsc{Rankabs}(A, \varepsilon\|A+B\|/2) + \textsc{Rankabs}(B, \varepsilon\|A+B\|/2) \\ &= \textsc{Rank}(A, \varepsilon\|A+B\|/2\|A\|) + \textsc{Rank}(B, \varepsilon\|A+B\|/2\|B\|). \quad \square \end{aligned}$$

Let us consider the ith iteration of Algorithm 2.1. Using (2.20), we can bound the rank of $E = X - F$ as

$$\textsc{Rank}(E, \varepsilon_\ell) \le \textsc{Rank}(X, \varepsilon_\ell\|E\|/2\|X\|) + \textsc{Rank}(F). \tag{2.21}$$

Since F is the low-rank factorization of X after i iterations, it achieves an accuracy of ε_ℓ^i and so

$$\textsc{Rank}(F) = \textsc{Rank}(X, \varepsilon_\ell^i) \tag{2.22}$$

(we assume here that $\varepsilon_\ell^i \le \varepsilon$ as otherwise the method would be stopped). After i iterations, $\|E\| \le \varepsilon_\ell^i\|X\|$ and so overall (2.21) becomes

$$\textsc{Rank}(E, \varepsilon_\ell) \le \textsc{Rank}(X, \varepsilon_\ell^{i+1}/2) + \textsc{Rank}(X, \varepsilon_\ell^i). \tag{2.23}$$

Since $\textsc{Rank}(X, \varepsilon') \le \textsc{Rank}(X, \varepsilon)$ for $\varepsilon \le \varepsilon'$, assuming that $\varepsilon \le \varepsilon_\ell^{i+1}/2$, we can obtain a simpler but weaker version of (2.23),

$$\textsc{Rank}(E, \varepsilon_\ell) \le 2\,\textsc{Rank}(X, \varepsilon). \tag{2.24}$$

Thus, the rank of E at any iteration is at most twice as large as the numerical rank of X at accuracy ε.

With a similar argument, the rank of the refined factors $F + F_E$ is bounded by

$$
\begin{aligned}
\textsc{Rank}(F + F_E) &\le \textsc{Rank}(F) + \textsc{Rank}(F_E) && (2.25)\\
&= \textsc{Rank}(F) + \textsc{Rank}(E, \varepsilon_\ell) && (2.26)\\
&\le \textsc{Rank}(X, \varepsilon_\ell^{i+1}/2) + 2\,\textsc{Rank}(X, \varepsilon_\ell^{i}), && (2.27)
\end{aligned}
$$

where (2.27) follows from (2.22) and (2.23). Again, a simpler but weaker bound is

$$
\textsc{Rank}(F + F_E) \le 3\,\textsc{Rank}(X, \varepsilon), \tag{2.28}
$$

showing that at any iteration the rank of the factors before recompression is at most three times the numerical rank of X at accuracy ε.

Now that we have bounded the ranks of the objects appearing in Algorithm 2.1, we are ready to analyze its cost. We denote as p and s the product and the sum of the dimensions of X, respectively. Thus, if $X \in \mathbb{R}^{m\times n}$ is a matrix, $p = mn$ and $s = m + n$; if $X \in \mathbb{R}^{n_1\times\cdots\times n_d}$ is a tensor of order d, $p = \prod n_i$ and $s = \sum n_i$. For readability, we only report the dominant term for the cost of each assumption/line, and assume a large scale setting, so that $p \gg s$.

We make the following assumptions on the flops (floating-point operations) cost of the different kernels:

$$
\begin{aligned}
\text{Flops } \textsc{Lra}(X, \varepsilon) &= c_1 p\,\textsc{Rank}(X, \varepsilon)_1 + o(p), && (2.29a)\\
\text{Flops } \textsc{Decompress}(F) &= c_2 p\,\textsc{Rank}(F)_1 + o(p), && (2.29b)\\
\text{Flops } \textsc{Recompress}(F, \varepsilon) &= o(p). && (2.29c)
\end{aligned}
$$

For tensors, the cost of Lra and Decompress depends on the order in which the dimensions are treated (see section 4). Here we assume they are treated in the natural order, without loss of generality since the dimensions can be arbitrarily reordered. In this case, the dominant cost of Lra and Decompress is proportional to the first coefficient of the rank vectors, denoted as $\textsc{Rank}(X, \varepsilon)_1$. Note that for matrices, the rank is a scalar so that $\textsc{Rank}(X, \varepsilon)_1 = \textsc{Rank}(X, \varepsilon)$.

Let us now measure the flops cost of the ith iteration of Algorithm 2.1.

- Line 3: $c_2 p\,\textsc{Rank}(X, \varepsilon_\ell^{i})_1$ flops.
- Line 4: $2p$ flops (for the Frobenius norm).
- Line 6: p flops.
- Line 7: $c_1 p\big(\textsc{Rank}(X, \varepsilon_\ell^{i+1}/2)_1 + \textsc{Rank}(X, \varepsilon_\ell^{i})_1\big)$ flops.
- Line 8: $o(p)$ flops.
- Line 9: $o(p)$ flops.

The dominant steps are the calls to Lra (Line 7), which is performed in low precision, and to Decompress (Line 3), which is performed in high precision. The computation of α and the scaling by α (Lines 4 and 6) could also be significant if the ranks are very small.

Summing the costs of these dominant steps across all n_{it} iterations, plus the cost of the initial Lra (Line 1), gives a total flops cost of

$$
\begin{aligned}
\text{Flops IR } = c_1 p\,\textsc{Rank}(X, \varepsilon_\ell)_1 + p \sum_{i=1}^{n_{\text{it}}} \Big(& c_1\,\textsc{Rank}(X, \varepsilon_\ell^{i+1}/2)_1 \\
&+ c_1\,\textsc{Rank}(X, \varepsilon_\ell^{i})_1 + c_2\,\textsc{Rank}(X, \varepsilon_\ell^{i})_1 + 3\Big).
\end{aligned} \tag{2.30}
$$

Since some of these flops are performed in low precision and some in high precision, we must account for the different speeds of different arithmetics. To do so, we ponderate the low precision flops by a weight $\omega_\ell < 1$ which indicates the relative performance ratio between the low and the high precision. We obtain

$$\begin{aligned} \text{Time IR } &\propto \omega_\ell c_1 p \,\textsc{Rank}(X, \varepsilon_\ell)_1 + p \sum_{i=1}^{n_{\text{it}}} \Big(\omega_\ell c_1 \,\textsc{Rank}(X, \varepsilon_\ell^{i+1}/2)_1 \\ &\quad + \omega_\ell c_1 \,\textsc{Rank}(X, \varepsilon_\ell^{i})_1 + c_2 \,\textsc{Rank}(X, \varepsilon_\ell^{i})_1 + 3 \Big), \end{aligned} \tag{2.31}$$

where the "Time" formula should be taken as a rough estimator of the time performance of the algorithm, although in practice the actual execution time depends on a number of other complex factors such as the arithmetic intensity of the operations, the data locality, and the parallelism.

This cost is to be compared with the cost of simply computing $\textsc{Lra}(X, \varepsilon)$ in the high precision u, given by

$$\text{Time Ref. } \propto c_1 p \,\textsc{Rank}(X, \varepsilon)_1. \tag{2.32}$$

This complexity analysis reveals two situations where Algorithm 2.1 can outperform the uniform precision approach.

Numerical ranks $\textsc{Rank}(X, \varepsilon_\ell^i)$ rapidly decreasing as ε_ℓ increases. The first situation is when the numerical ranks of X at accuracy lower than the final target ε are much smaller than $\textsc{Rank}(X, \varepsilon)$. This can certainly be the case in some applications. For example, in the case of matrices, the numerical rank of X at any given accuracy is determined by its singular values: $r = \textsc{Rank}(X, \varepsilon_\ell)$ is the number of singular values that need to be kept for the truncated SVD to approximate X with accuracy at least ε_ℓ. Thus, if the singular values decay rapidly, $\textsc{Rank}(X, \varepsilon_\ell)$ will in general be significantly smaller than $\textsc{Rank}(X, \varepsilon)$. The most extreme example is a matrix with one large singular value and all the remaining $\textsc{Rank}(X, \varepsilon) - 1$ singular values just above $\varepsilon\|X\|$. In this extreme case, all the iterations except the last will only need to compute on rank-1 matrices, so the overall cost of the method will be dominated by that of the last iteration, which is roughly $\omega_\ell c_1 p \,\textsc{Rank}(X, \varepsilon)$, and so is always smaller than (2.32). In a more realistic setting where the singular values decay exponentially, we may still expect (2.31) to be smaller than (2.32) even for reasonably traditional values of ω_ℓ.

This situation also extends to tensors, although we do not have such a simple characterization as one based on singular values. Essentially, if the underlying matrices used in the tensor representation exhibit rapidly decaying singular values, then the rank vectors $\textsc{Rank}(X, \varepsilon_\ell)$ will take much smaller values than $\textsc{Rank}(X, \varepsilon)$ when $\varepsilon_\ell \gg \varepsilon$.

Very fast low precision arithmetic (or very slow high precision arithmetic). The second situation is when the low precision arithmetic is much faster than the high precision one, that is, when $\omega_\ell \ll 1$. This is becoming increasingly common for low precisions on modern hardware, especially accelerators. For example, the fp16 and bfloat16 arithmetics can be up to 16 times faster than fp32 arithmetic on recent NVIDIA GPUs. Similarly, fp16 arithmetic can be up to 8 times faster than fp32 on the AMD Instinct MI250X GPUs.

Moreover, such hardware provides another crucial feature which allows the $\textsc{Decompress}$ kernel to also be performed in low precision. Indeed, while Algorithm 2.1 requires the $\textsc{Decompress}$ kernel Line 3 to be performed in high precision, this kernel has the particularity of taking as input the factors F, which are stored in low precision. Therefore, what we really need is to compute in high precision with low precision numbers; this happens to be an easier task than the more general problem of computing in high precision with high precision numbers. In fact, an increasing range of modern hardware provides the capability to perform this task inexpensively. For example, the NVIDIA GPUs are equipped with so-called tensor core units that can carry out matrix multiplication with half precision (16-bit) matrices at the accuracy of single precision (fp32) arithmetic but at the much higher speed of half precision arithmetic. Similar instructions are available on several other architectures, which have been analyzed by Blanchard et al. [6] under a common framework called block FMA. In our context, a block FMA unit therefore allows all the dominant operations to be performed at the speed of the low precision, which is much higher than that of the high precision. In this situation, the time cost of Algorithm 2.1 can be much lower than (2.32), even in the worst case scenario where the ranks of all objects throughout the iterations attain their upper bound of $2\,\textsc{Rank}(X,\varepsilon)$ for E. Indeed, in this case, we have

$$\text{Time IR } \propto \omega_\ell p\,\textsc{Rank}(X,\varepsilon)_1\big(c_1(2n_{\text{it}}+1)+n_{\text{it}}c_2\big)+3p. \tag{2.33}$$

Neglecting the $3p$ term, (2.33) is smaller than (2.32) if

$$\omega_\ell(2n_{\text{it}}+1+n_{\text{it}}c_2/c_1)\le 1. \tag{2.34}$$

For many LRA algorithms, including all those considered in this article, the cost of compressing (Lra kernel) a full object is higher than the cost of decompressing it (Decompress kernel), that is, $c_1>c_2$. Therefore (2.34) is certainly satisfied if $\omega_\ell(3n_{\text{it}}+1)\le 1$.

To illustrate this cost analysis, let us take the NVIDIA GPU tensor cores as an example, for which fp16 arithmetic ($u_\ell\approx 3\times 10^{-4}$) is up to 16 times faster ($\omega_\ell=1/16$) than fp32 arithmetic ($u\approx 6\times 10^{-8}$). Since $u_\ell^2\approx u$, a single refinement step ($n_{\text{it}}=1$) suffices to recover fp32 accuracy. Thus, the term $\omega_\ell(3n_{\text{it}}+1)$ is equal to $4/16=1/4$, which suggests that a speedup of up to a factor $4\times$ could be expected.

Clearly, the two situations discussed above are not exclusive, so in practice it is very possible that we have both smaller ranks and a fast low precision. To further assess under which condition Algorithm 2.1 can outperform the standard approach, we now specialize it to specific matrix or tensor algorithms in section 3 and section 4.

3. Application to matrix low-lank approximation.

In this section we specialize Algorithm 2.1 to the case where $X\in\mathbb{R}^{m\times n}$ is a matrix.

For any unitarily invariant norm, the optimal LRA algorithm for matrices is to compute their singular value decomposition (SVD) and truncate it to the desired accuracy, a result known as Eckart–Young theorem [12]. Specifically, given the SVD $X=U\Sigma V^T$, the optimal rank-k approximation of X is

$$\arg\min_{\substack{M\in\mathbb{R}^{m\times n}\\ \textsc{Rank}(M)=k}}\|X-M\|=U_k\Sigma_k V_k^T,$$

where $U_k\Sigma_k V_k^T$ is the truncated SVD of X, formed by the first k singular vectors and values only.

While the truncated SVD provides the best accuracy, it is expensive due to the need to compute the full SVD before truncating it: it requires $O(mn^2)$ flops. For this reason, low-rank matrix approximations are often computed using slightly less accurate but cheaper alternatives. In this paper, we focus on two widely popular choices: the truncated QR decomposition with column pivoting (QRCP, subsection 3.1), and the randomized SVD (subsection 3.2). For each algorithm, we discuss their use as

LRA kernel in our mixed precision iterative refinement approach (Algorithm 2.1): in particular, we check that the algorithm satisfies the assumptions (2.1)–(2.3) of the error analysis and the assumptions (2.29a)–(2.29c) of the complexity analysis. We also explain how to perform the RECOMPRESS kernel based on the specific LRA choice.

3.1. Truncated QRCP decomposition. Our first choice of LRA algorithm is a truncated QRCP decomposition. QRCP decomposes the original matrix $X \in \mathbb{R}^{m\times n}$ as

$$XP = QR$$

where $Q \in \mathbb{R}^{m\times n}$ is a matrix with orthonormal columns, $R \in \mathbb{R}^{n\times n}$ is an upper triangular matrix, and $P \in \mathbb{R}^{n\times n}$ is a permutation matrix. Thanks to pivoting, the QRCP decomposition can be used as a rank-revealing algorithm because the norm of the trailing submatrix at each step k of the QR factorization, $\|XP - Q_k R_k\|$, is monotonically decreasing. Therefore, we can stop the QR factorization as soon as this norm becomes smaller than the target tolerance ε_ℓ. We obtain

$$X \approx Q_k V_k^T, \quad Q_k \in \mathbb{R}^{m\times k}, \quad V_k = PR_k^T \in \mathbb{R}^{n\times k}, \tag{3.1}$$

where $k = \text{RANK}(X, \varepsilon_\ell)$. To avoid the need to apply the permutation P whenever we need to apply X, we form and store $V_k = PR_k^T$.

We can implement a suitable RECOMPRESS algorithm using this truncated QRCP decomposition. Several versions are possible; we describe in Algorithm 3.1 the one that we will use in this article. Given a (non-optimal) truncated QRCP decomposition $Q_{in}V_{in}^T$, Algorithm 3.1 recompresses it into an optimal LRA as follows. First, a thin QR factorization $\overline{Q}R$ of V_{in} is computed. Then, the product $Q_{in}R^T$ is formed and a truncated QRCP decomposition $Q_{out}\overline{V}^T$ is computed at the desired target accuracy ε. This yields the recompressed left factor Q_{out}; the recompressed right factor V_{out} is obtained by forming $\overline{Q}\overline{V}$.

Algorithm 3.1 RECOMPRESS algorithm using truncated QRCP decomposition.

```
Input: a truncated QRCP decomposition $Q_{in}V_{in}^T$ of the form (3.1);
       $\varepsilon$, the target accuracy.
Output: a recompressed QRCP decomposition $Q_{out}V_{out}^T$.
1: Compute the thin QR factorization $\overline{Q}R = V_{in}$.
2: Compute the truncated QRCP decomposition $Q_{out}\overline{V}^T = Q_{in}R^T$ with low-rank
   truncation threshold $\varepsilon$.
3: $V_{out} \leftarrow \overline{Q}\overline{V}$
```

We now check that the assumptions (2.1)–(2.3) of the error analysis are satisfied for truncated QRCP and determine the value of the constants in the error bounds. For (2.1), we can analyze the truncated QRCP by separating the truncation and rounding errors. As explained above, we can control the size of the truncation error by stopping the QRCP decomposition once the approximation error falls below the desired threshold ε_ℓ, so that in exact arithmetic we obtain

$$Q_k R_k = XP + E, \quad \|E\| \le \varepsilon_\ell \|X\|.$$

To account for the rounding errors, we use standard analysis of QR decomposition [18, p. 361], which shows that the computed QR factors satisfy the backward error bound

$$\widehat{Q}_k \widehat{R}_k = XP + \Delta X + E, \quad \|\Delta X\| \le \sqrt{k}\widetilde{\gamma}_{mk}\|X\|, \tag{3.2}$$

where $\widetilde{\gamma}_{mk} = cmku_\ell/(1-cmku_\ell) \simeq cmku_\ell$, for a modest constant c independent of the dimensions of the problem. Note that this bound is valid even with column pivoting, since computing the QRCP decomposition $XP = QR$ of X is equivalent to computing the unpivoted QR decomposition of XP. Assumption (2.1) is thus satisfied with $b_1 \simeq cmk\sqrt{k}$. The Decompress kernel is a standard matrix multiplication between the low-rank factors Q and V hence assumption (2.2) is satisfied with $b_2 = k$ by [18, p. 71]. Finally, to bound the error introduced by the Recompress kernel we must analyze Algorithm 3.1. Since the algorithm simply consists of standard QR, QRCP, and matrix multiplication operations, this analysis is straightforward and relies on standard error bounds from the literature. Although we omit it here for the sake of conciseness, we have performed this analysis and found that (2.3) is satisfied with $b_3 \simeq cn(k^{3/2} + \widetilde{k}^{3/2}) + k$, where k is the rank of $Q_{in}V_{in}^T$ before recompression and $\widetilde{k}$ is the rank of $Q_{out}V_{out}^T$ after recompression.

Finally, we discuss the cost of Algorithm 2.1 when using truncated QRCP decomposition. The truncated QRCP decomposition (3.1) can be computed in $4mnk + o(mnk)$ flops [14], so that assumption (2.29a) is satisfied with $c_1 = 4$. The Decompress kernel is a matrix multiplication which requires $2mnk$ flops, so that assumption (2.29b) is satisfied with $c_2 = 2$. For the Recompress kernel, Algorithm 3.1 requires $(6m + 2n)k^2 + o(mk^2 + nk^2)$ flops, so that assumption (2.29c) is satisfied since $(m + n)k^2 = o(mn)$.

3.2. Randomized SVD decomposition. Our second choice of LRA algorithm is the randomized SVD decomposition. Several variants have been proposed; in this article we use the one described in Algorithm 3.2, which was was proposed by Martinsson and Voronin [24].

Algorithm 3.2 Randomized SVD decomposition.

Input: $X \in \mathbb{R}^{m\times n}$, a target accuracy ε_ℓ, and a block size b.
Output: a truncated SVD $U\Sigma V^T$ decomposition of X.

```
1:  Initialize $Q$ and $B$ to empty matrices.
2:  $n_X = \|X\|$
3:  repeat
4:      Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times b}$.
5:      $Y = X\Omega$
6:      $Q_b = \texttt{qr}(Y - Q(Q^TY))$
7:      $B_b = Q_b^T X$
8:      $Q \leftarrow [Q\ Q_b]$
9:      $B \leftarrow \begin{bmatrix} B \\ B_b \end{bmatrix}$
10:     $X \leftarrow X - Q_bB_b$
11: until $\|X\| \le \varepsilon_\ell n_X$
12: Compute the truncated SVD decomposition $B \approx \overline{U}\Sigma V^T$ at accuracy $\varepsilon_\ell$.
13: $U = Q\overline{U}$
```

The algorithm consists of two phases. The first phase (Lines 3 to 11) iteratively builds a QB decomposition of the matrix such that $\|X - QB\| \le \varepsilon_\ell\|X\|$, where $Q \in \mathbb{R}^{m\times k}$ has orthonormal columns and where the dimension k is hopefully close to Rank(X, ε_ℓ). To do so, a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times b}$ is drawn for a given block size b and used to sample the matrix $Y = X\Omega$ (Line 5). Then Y is added to the basis Q while preserving the orthonormality: this can for example be accomplished using the Gram–Schmidt algorithm (Line 6). This process is repeated until the columns of Q are a sufficiently good approximation of the column space of X, that is, until $\|X - QQ^T X\|$ is small. In order to efficiently compute this quantity, the matrix $B = Q^T X$ is formed block by block (Line 7). Then, in the second phase (Lines 12 and 13), a truncated SVD of X can easily be obtained by computing the truncated SVD of the lower dimensional matrix $B \in \mathbb{R}^{k\times n}$: if $B = \overline{U}\Sigma V^T$, then $X = U\Sigma V^T$ with $U = Q\overline{U}$.

We chose to use this specific randomized SVD variant because it presents several advantages. It is blocked, which allows for an efficient implementation on modern hardware. Moreover it provides a cheap yet reliable error estimation. Thanks to blocking and error estimation, Algorithm 3.2 can adaptively reveal the numerical rank of the matrix at the requested accuracy ε_ℓ; no a priori knowledge on the rank is thus necessary. Finally, we have experimentally observed Algorithm 3.2 to be more accurate than other randomized SVD variants that we have tested.

In order to perform the Recompress operation, one could simply use Algorithm 3.1; however, if one wishes to use randomized SVD not only for the Lra but also the Recompress operation, then Algorithm 3.2 can still be used with some modifications. These modifications are described in Appendix A and result in Algorithm A.1.

We now check if the assumptions (2.1)–(2.3) of the error analysis are satisfied for the randomized SVD. To do so, we rely on the error analysis of Connolly, Higham, and Pranesh [9], which determines error bounds for Algorithm 3.2 in floating-point arithmetic. Note that their analysis assumes that the deterministic SVD on line 12 is computed with a numerically stable algorithm. By [9, Corollary 2.4], assumption (2.1) is satisfied with $b_1 \simeq \sqrt{n}mk$. Assumption (2.2) is satisfied with $b_2 = k + 1$ since decompressing $U\Sigma V^T$ can be achieved with a matrix–matrix product and a scaling. Finally, the analysis of [9] does not directly apply to the Recompress version of the algorithm discussed above, but we expect its numerical behavior to be similar to the original algorithm.

Next, we discuss the cost of Algorithm 3.2. If X is a full $m \times n$ matrix, Algorithm 3.2 costs $6mnk + o(mnk)$ flops [24, Eqn. (25)], so that assumption (2.29a) is satisfied with $c_1 = 6$. It is easy to extend [24, Eqn. (25)] to the case where $X = U\Sigma V^T$ is represented under low-rank form to check that the Recompress variant of Algorithm 3.2 described above has a cost in $O((m + n)k^2) = o(mn)$ flops, so that assumption (2.29c) is indeed satisfied. Finally, assumption (2.29b) is satisfied with $c_2 = 2$ since the Decompress kernel is simply a matrix–matrix product.

4. Application to tensor low-rank approximation.

In this section, we explore the application of our method to three different tensor decompositions, namely Tucker [32], TT [29], and HT [15]. All three decompositions provide direct methods for the Lra and Recompress kernels that can guarantee a prescribed accuracy ε, as well as fast Decompress routines, rendering them amenable to Algorithm 2.1. We first give a brief description of these decompositions as well as their corresponding Lra, Recompress, and Decompress kernels, then provide an error and complexity analysis with respect to the assumptions (2.1)–(2.3) and (2.29a)–(2.29c).

4.1. Tucker, TT, and HT decompositions.

The Tucker decomposition represents a tensor $X \in \mathbb{R}^{n_1\times\cdots\times n_d}$ with a core tensor $G \in \mathbb{R}^{r_1\times\cdots\times r_d}$ and with factor matrices $U = \{U^{(i)}\}$ with orthonormal columns, where $U^{(i)} \in \mathbb{R}^{n_i\times r_i}$ for $i \in \{1, \ldots, d\}$.

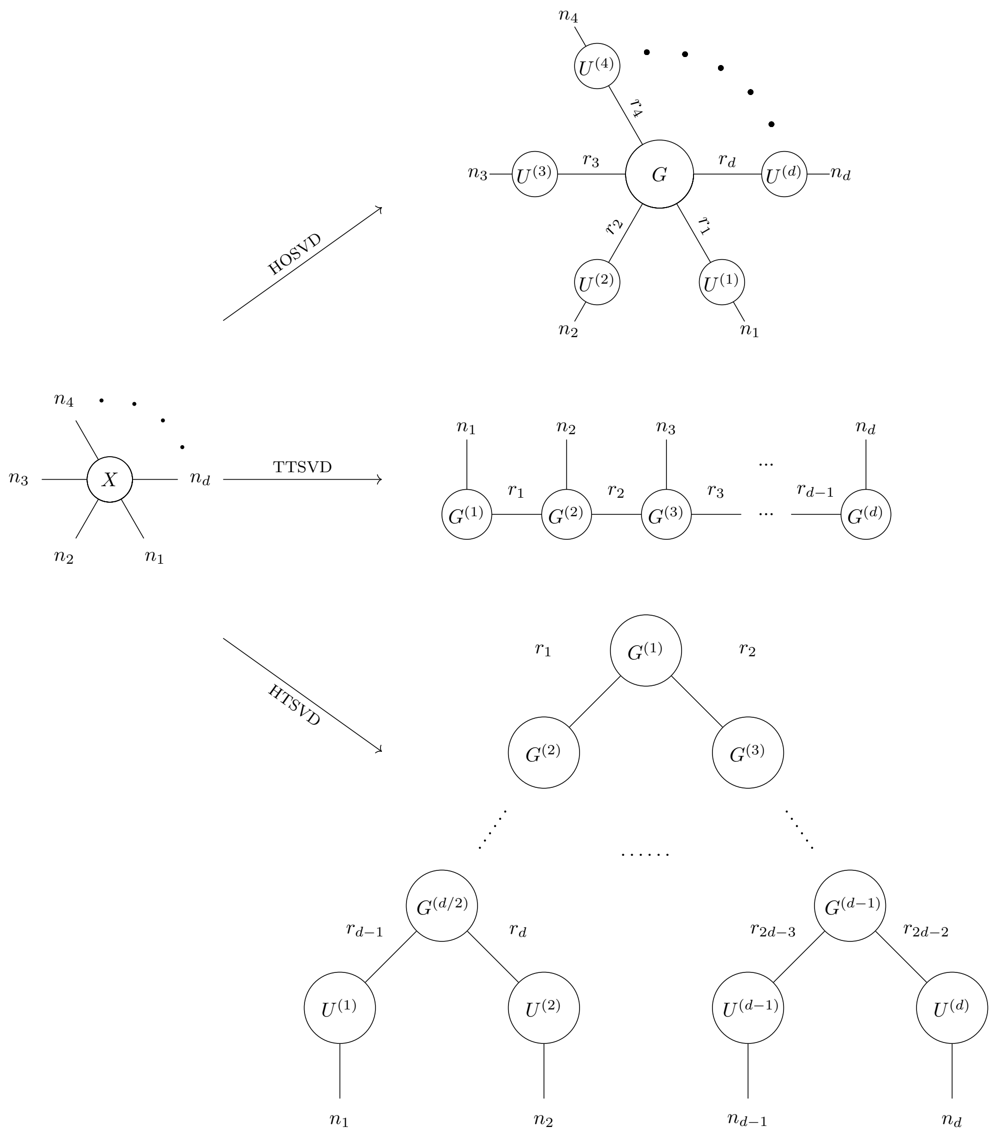


Fig. 4.1: A *d*-order tensor (left) and its representation in Tucker, TT, and HT decompositions (right) using tensor network diagram.

Each element of a Tucker tensor is given by

$$X_{i_1,\dots,i_d} = \sum_{\alpha_1,\dots,\alpha_d}^{r_1,\dots,r_d} G_{\alpha_1,\dots,\alpha_d} U^{(1)}_{i_1,\alpha_1} \dots U^{(d)}_{i_d,\alpha_d}. \tag{4.1}$$

The TT format consists of a sequence of 3rd-order tensors $G^{(i)}$ linked by the first and third dimensions that correspond to ranks. Each element of a TT tensor is given by

$$X_{i_1,\dots,i_d} = \sum_{\alpha_0,\dots,\alpha_d}^{r_0,\dots,r_d} G^{(1)}_{\alpha_0,i_1,\alpha_1} \dots G^{(d)}_{\alpha_{d-1},i_d,\alpha_d}. \tag{4.2}$$

The HT decomposition represents X as a binary tree of tensors whose leaves are matrices $U^{(i)} \in \mathbb{R}^{n_i \times r_i}$ and the rest are 3rd order tensors. Each element of an HT tensor is given by

$$X_{i_1,\dots,i_d} = \sum_{\alpha_0,\dots,\alpha_{2d-2}}^{r_0,\dots,r_{2d-2}} G^{(1)}_{\alpha_0,\alpha_1,\alpha_2} \dots G^{(d-1)}_{\alpha_{d-2},\alpha_{2d-3},\alpha_{2d-2}} U^{(1)}_{i_1,\alpha_{d-1}} \dots U^{(d)}_{i_d,\alpha_{2d-2}}. \tag{4.3}$$

We illustrate these decompositions in Figure 4.1 using a tensor network diagram [7] where each circle represents a tensor and each edge corresponds to a dimension. For each decomposition, the corresponding LRA method yields a network of core tensors (or just “cores”), whose contraction/multiplication along the inner dimensions (of size r_i, $i = 1\colon \bar{d}$) yields the full tensor. The outer dimensions (of size n_i, $i = 1\colon d$) correspond to dimensions of the full tensor X.

Algorithm 4.1 LRA algorithm for a tensor.

Input: a tensor X and a target accuracy ε.
Output: F_X: a low-rank tensor decomposition of X.

```
1: $\bar{\varepsilon} \leftarrow \varepsilon/\bar{d}$
2: $F_X \leftarrow []$
3: for $i = 1$ to $\bar{d}$ do
4:     $[U, Y] \leftarrow \mathrm{SVD}(\mathrm{MATRICIZE}(X), \bar{\varepsilon})$
5:     $F_X \leftarrow [F_X, \mathrm{RESHAPE}(U)]$
6:     $X \leftarrow \mathrm{RESHAPE}(Y)$
7: end for
8: $F_X \leftarrow [F_X, \mathrm{RESHAPE}(X)]$
```

The HOSVD [32], TTSVD [29], and HTSVD [15] algorithms can be used as LRA kernels to compute the Tucker, TT, and HT decompositions of a given full tensor X, respectively. All three are direct methods that can achieve any prescribed accuracy ε in exact arithmetic. To the best of our knowledge, the effect of rounding errors on these methods in finite precision arithmetic has not been analyzed in the literature. While these methods are therefore not known to be stable, empirical experiments suggest that they still reliably achieve an accuracy of order ε when run in a finite precision with a unit roundoff u sufficiently smaller than ε. Formally proving this result is an open problem that deserves a dedicated study.

We provide a high level pseudo-code that encapsulates these methods in Algorithm 4.1. In essence, all three algorithms perform $\bar{d}$ (defined in (1.1)) successive SVDs on matricizations of X to isolate each core of the decomposition and determine its corresponding rank, as indicated on Line 4. Then, the left orthogonal component U of the SVD is reshaped and added to the decomposition F_X, and the non-orthogonal component is reshaped into a tensor to be further factored in the subsequent iterations. At the end, the decomposed tensor F_X consisting of $\bar{d}+1$ tensors is returned.

As the LRA kernel isolates the cores through a series of $\bar{d}$ SVDs on X, the DECOMPRESS kernel performs successive tensor contractions on neighboring cores of F_X through the inner dimensions that link them to obtain the full tensor X. In Algorithm 4.2, we provide a generic DECOMPRESS kernel that performs these contractions

Algorithm 4.2 Decompress algorithm for a tensor decomposition.

```
Input: a low-rank tensor decomposition $F_X$.
Output: the full tensor $X$ corresponding to $F_X$.
1: $X \leftarrow F_X[1]$
2: for $i = \bar{d}$ to 1 in reverse order do
3:     $X \leftarrow X \times_{\{r_i\}} F_X[i+1]$
4: end for
```

in the reverse order of SVDs in Algorithm 4.1 for simplicity. This is done on Line 3 where $\times_{\{r_i\}}$ represents the contraction/multiplication across the inner dimension of size r_i. In practice, these contractions could be performed in an arbitrary order among the $\bar{d}$ inner dimensions as tensor contraction is associative.

Finally, for the TT format we use the standard Recompress kernel (called “rounding” in the tensor literature) which relies on a sequence of QRs and SVDs on the matricizations of the cores (we refer to [29] for a detailed description). For HT and Tucker decompositions, we employ an adaptation of this TT Recompress kernel to these formats, instead of the standard Recompress kernel proposed in [15] mainly to avoid the Gram matrix formation therein.

4.2. Discussion of the error and complexity. For the error analysis, as previously mentioned, tensor computations are experimentally observed to behave stably in finite precision arithmetic, but a formal proof of this fact is not known. The Lra kernel is based on a chain of SVDs, the Decompress kernel is based on a chain of matrix multiplies, and the Recompress kernel on a chain of QRs, SVDs, and matrix multiplies. Despite the fact that all of these basic linear algebra building blocks have stable implementations, it is not sufficient to directly conclude on the stability of the overall tensor computation, because the composition of stable algorithms is not necessarily stable [5]. Therefore, the stability of tensor algorithms is an open problem that deserves a dedicated study.

For the complexity analysis of the Lra step, we assume that the first SVD reduces the size of the tensor significantly, that is, to $o(p)$, which renders the cost of subsequent SVDs negligible. In this case, we can use the same constant $c_1 = 6$ as in the case of randomized SVD in subsection 3.2. With the same assumption, the cost of Decompress in Algorithm 4.2 will also be dominated by the last contraction, which is essentially a matrix multiplication on permuted tensors across the first inner dimension. Thus, we can similarly use the constant $c_2 = 2$ as in subsection 3.2, since the cost of the previous contractions is in $o(p)$ in this case as well. Even without this assumption, we can find constants $c_1 = 6t$ and $c_2 = 2t$ with the same $t > 1$, since intermediate tensor sizes in each SVD and contraction steps in Algorithm 4.1 and Algorithm 4.2 stay the same. This keeps the cost ratio of these two steps constant across all $\bar{d}$ dimensions yielding the same t for c_1 and c_2; we skip further details for brevity. Finally, the cost of Recompress remains in $o(p)$ as it involves QRs and SVDs on the matricizations of 2nd or 3rd order cores in the decomposition, whose size is in $o(p)$ by the low-rank assumption. Therefore, we satisfy the assumptions (2.29a)–(2.29c) of the complexity analysis in the tensor case.

5. Numerical experiments.

5.1. Experimental setting. We now test our iterative refinement approach experimentally. We developed a MATLAB code that implements Algorithm 2.1 and can use as Lra kernel any of the matrix and tensor LRA algorithms discussed in the previous two sections: QRCP, randomized SVD, HOSVD, TTSVD, and HTSVD.

For the matrix algorithms (QRCP and randomized SVD), we use our own implementation. For the tensor algorithms (HOSVD, TTSVD, and HTSVD), we rely on the implementations from the libraries described in [4], [28], and [23], respectively, with some adjustments. We use MATLAB version R2019a throughout the experiments.

In the experiments, the high precision u is set to double precision (fp64 arithmetic, with unit roundoff $u = 2^{-53} \approx 1 \times 10^{-16}$) and the use of various low precisions u_ℓ is simulated with the chop library of Higham and Pranesh [20]: single precision (fp32 arithmetic, with unit roundoff $u_\ell = 2^{-24} \approx 6 \times 10^{-8}$) and half precision (fp16 and bfloat16 arithmetics, with unit roundoff $u_\ell = 2^{-11} \approx 5\times10^{-4}$ and $u_\ell = 2^{-8} \approx 4\times10^{-3}$, respectively).

For the low-rank truncation thresholds, we set $\varepsilon = 10^{-13}$ and $\varepsilon_\ell = \theta_\ell u_\ell$, where $\theta_\ell \le 1$ is a scaling factor necessary to control the effect of rounding errors on the ability of the LRA to detect the correct numerical rank. We analyze in detail the role of this parameter θ_ℓ in subsection 5.3; based on the conclusions of this analysis, we have set $\theta_\ell = 2^{-1}$ for all LRA kernels except randomized SVD and QRCP, for which we have used $\theta_\ell = 2^{-2}$ and $\theta_\ell = 2^{-3}$, respectively.

To test the algorithms, we use randomly generated matrices and tensors with various singular value distributions. More precisely, we compare three types of distributions for the singular values σ_i:

$$\sigma_i = \max(\widehat{\sigma}_i, 10^{-16}), \quad \widehat{\sigma}_i = \begin{cases} 1/i & \text{(linear)} \\ i^{-10} & \text{(power)} \\ e^{-i} & \text{(exponential)} \end{cases} . \tag{5.1}$$

These three distributions are illustrated in Figure 5.1. All random matrices are square and of size 100×100. All random tensors are of fourth order ($d = 4$) and of size $100 \times 100 \times 100 \times 100$. We generate the matrices as $Q_1 \Sigma Q_2$ where Q_1, Q_2 are random orthogonal matrices and Σ is a diagonal matrix with the specified singular values as coefficients. For the tensor experiments involving Tucker and TT decompositions, we generate the low-rank tensor in the Tucker format with $Q^{(1)}$, ..., $Q^{(d)}$ random orthonormal matrices and a dth-order core tensor G whose coefficient $G_{i_1,\dots i_d}$ is given by $\sigma_{\max(i_1,\dots,i_d)}$ in (5.1). For the experiments involving the HT format, we generate the low-rank tensor in the HT format whose leaf nodes are matrices with orthogonal columns, and each element $G^{(t)}_{i,j,k}$ of its internal cores is set to $\sigma_{\max(i,j,k)}$.

In addition to these random data sets, we also report results for data sets coming from two real-life applications in section 5.4.

5.2. Experimental results. In the first experiment, we analyze the behavior of Algorithm 2.1 for the matrix case, which is provided in Figure 5.2. We consider three types of matrices based on their singular value distribution (linear, power, and exponential) and two Lra kernels (QRCP and randomized SVD). In each case, we plot the relative error

$$\eta_i = \frac{\|X - F_i\|}{\|X\|} \tag{5.2}$$

where F_i is the computed low-rank factor of X after i refinement steps (F_0 is thus a standard LRA of X in precision u_ℓ). The number next to each marker indicates the rank of F_i after recompression.

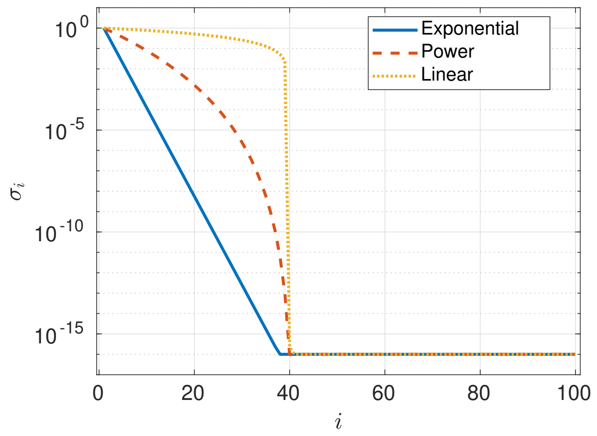


Fig. 5.1: Three types of singular value distributions used in the experiments.

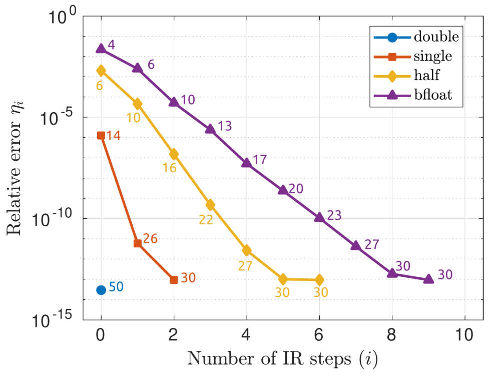
Figure 5.2 shows that for all three matrices and for both QRCP and randomized SVD, Algorithm 2.1 behaves as expected: the error η_i is roughly equal to $\varepsilon_\ell^{i+1} = (\theta u_\ell)^{i+1}$ after i refinement steps. Thus, using single precision as the low precision u_ℓ, we can achieve an accuracy close to double precision with only one refinement step. Naturally, since for QRCP $\theta = 2^{-3}$ is relatively small, $(u_\ell/\theta)^2$ is noticeably larger than the accuracy achieved by a standard double precision LRA; the refined factors are thus not completely as accurate as if computed directly in double precision. This gap can be filled by performing a second refinement step, although this would likely be unnecessary in most practical scenarios.

Similar results are obtained using half precision as the low precision format (fp16 or bfloat16 arithmetic). An error close to the single precision accuracy can be achieved in just one or a few steps. Moreover, we can even reach double precision accuracy if needed, which illustrates an attractive property of Algorithm 2.1: it can reach an accuracy which is only limited by the high precision, but not by the low precision, despite this low precision being used to perform most of its operations.

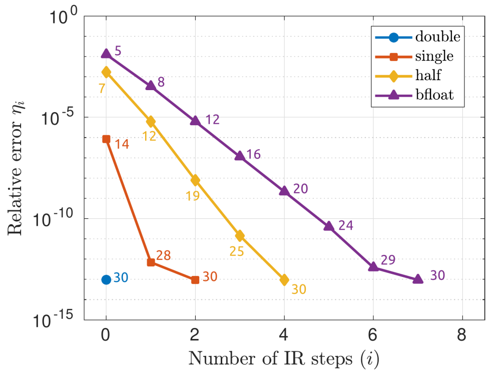
Finally, we discuss the rank behavior of F_i across IR steps. We see that it is roughly equal to $\textsc{Rank}(X, \eta_i)$, the numerical rank of X at accuracy η_i. Thus, for matrices with rapidly decaying singular values such as the exponential case in Figure 5.1, these ranks tend to be much smaller at the early steps in low precision. Reflecting on the cost of the algorithm, we can expect the use of low precision plus refinement to be particularly cost efficient for such matrices.

Figure 5.3 shows similar plots for the tensor decompositions (TTSVD, HOSVD, and HTSVD). The results for tensors follow the same trend, and lead to the same conclusion as for matrices. The main difference is that the rank of F_i (text labels) is now a vector instead.

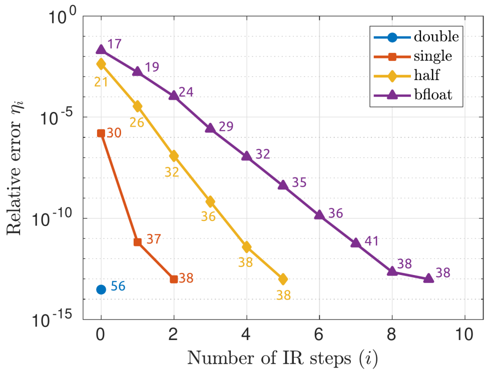
Overall, our experiments confirm that Algorithm 2.1 is able to rapidly converge to a high accuracy, while using low precision for the LRA kernel. This observation is valid for all types of matrices and tensors in our test set, and all five $\textsc{Lra}$ kernels that we tested, which shows that Algorithm 2.1 is robust and can work in a wide variety of settings.



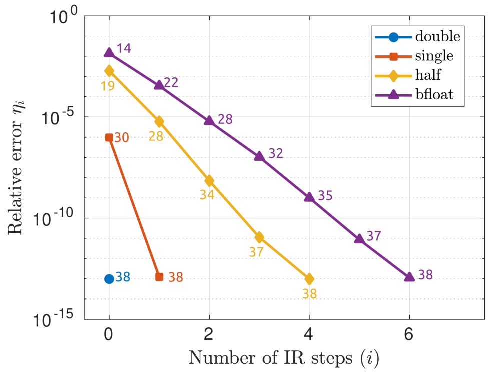
(a) QRCP (exponential).



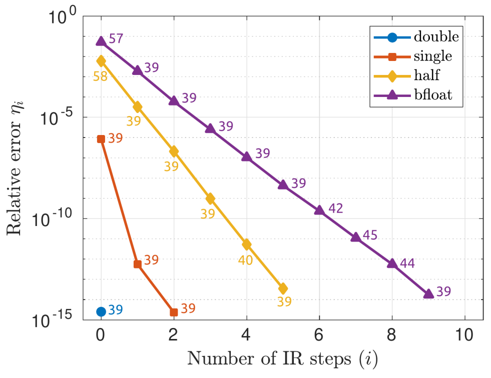
(b) Randomized SVD (exponential).



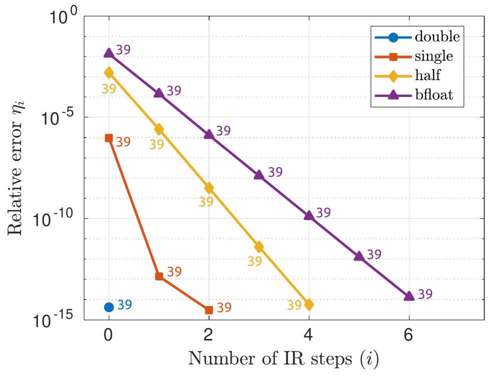
(c) QRCP (power).



(d) Randomized SVD (power).

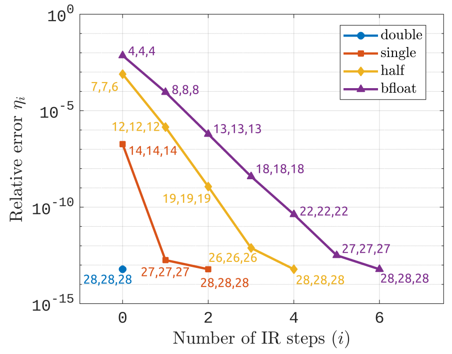


(e) QRCP (linear).

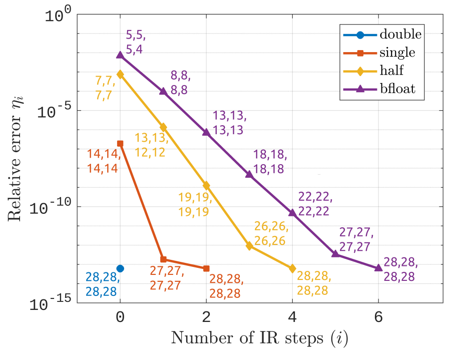


(f) Randomized SVD (linear).

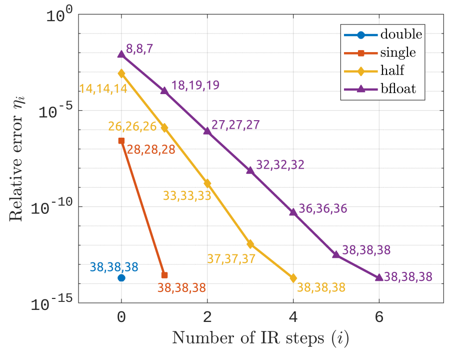
Fig. 5.2: Convergence of Algorithm 2.1 for three types of matrices (with different singular value distributions, see Figure 5.1) and for two different LRA kernels (QRCP or randomized SVD). The number next to each marker indicates the rank of F_i after recompression.



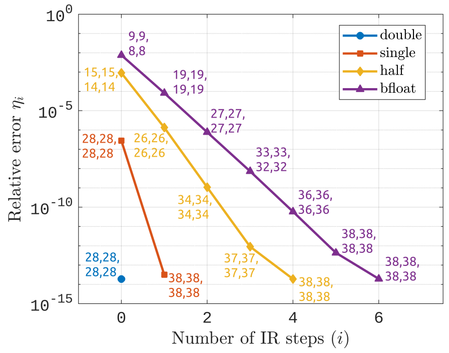
(a) TTSVD (exponential).



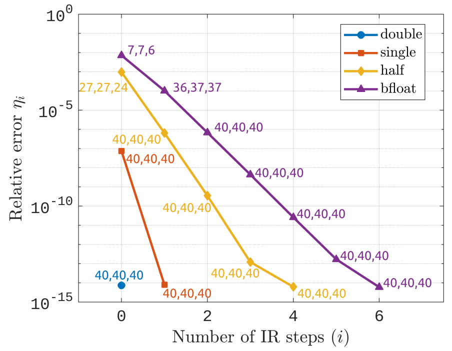
(b) HOSVD (exponential).



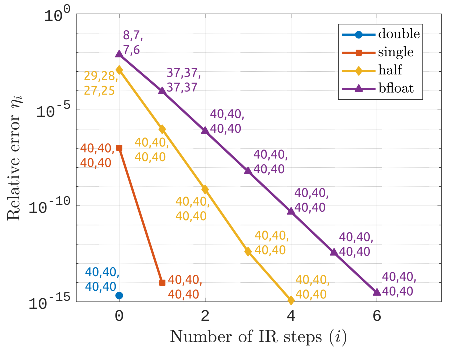
(c) TTSVD (power).



(d) HOSVD (power).

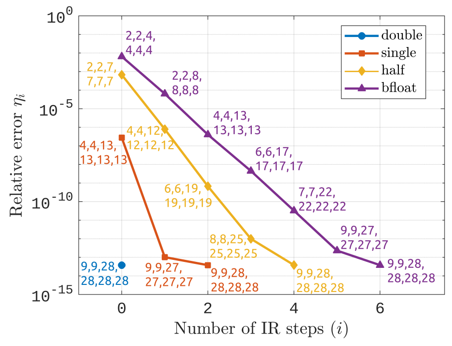


(e) TTSVD (linear).

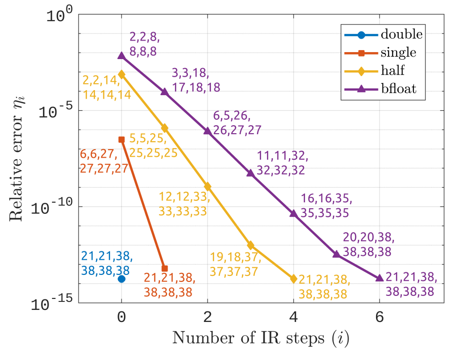


(f) HOSVD (linear).

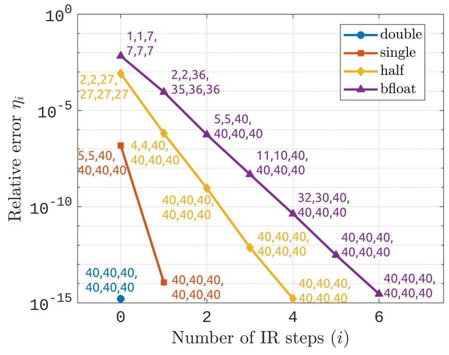
Fig. 5.3: Convergence of Algorithm 2.1 for three types of tensors (depending on the singular value distribution, see Figure 5.1) and with TTSVD or HOSVD as LRA kernel. The numbers next to each marker indicate the rank vector of F_i after recompression.



(a) HTSVD (exponential).



(b) HTSVD (power).



(c) HTSVD (linear).

Fig. 5.4: Convergence of Algorithm 2.1 for three types of tensors (depending on the singular value distribution, see Figure 5.1) and with HTSVD as LRA kernel. The numbers next to each marker indicate the rank vector of F_i after recompression.

5.3. Role of θ_ℓ. There is a tradeoff in choosing the scaling factor θ_ℓ: the larger it is, the faster the convergence (since the error is reduced by a factor roughly equal to $\theta_\ell u_\ell$ at each refinement step), but if it is too large, the rank will no longer be correctly detected and this will lead to a significant rank growth.

We illustrate this in Table 5.1 and Table 5.2, where we compare the convergence of Algorithm 2.1 for different values of θ_ℓ. Table 5.1 is for the TTSVD kernel and Table 5.2 is for the QRCP kernel; fp16 arithmetic is used as the low precision u_ℓ in both cases. The tables show that the method converges faster as θ_ℓ increases, as expected: the error η_i is smaller for larger values of θ_ℓ. For TTSVD (Table 5.1), this faster convergence is achieved without compromising the correct rank detection, which remains contained throughout the iterations and for all values of $\theta_\ell \leq 1$. Thus, in this case, a large value of θ_ℓ is recommended. The situation is different for QRCP (Table 5.2), for which a too large value of θ_ℓ (here $\theta_\ell \geq 2^{-2}$) prevents the rank from being correctly detected, thus leading to a rank explosion.

We therefore conclude that the optimal choice of θ_ℓ depends on the LRA algorithm, and more specifically, on its sensitivity to rounding errors when detecting the

Table 5.1: Relative error η_i and $\textsc{Rank}(F_i)$ at different steps i and for different values of θ_ℓ, for TTSVD (using fp16 as u_ℓ and exponential distribution of singular values).

	Relative error η_i					$\textsc{Rank}(F_i)$				
$i\|\theta_\ell$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	6e-04	8e-04	9e-04	2e-03	5e-03	23,39,16	7, 7, 6	6, 6, 6	5, 5, 5	4, 4, 4
1	3e-07	1e-06	4e-06	1e-05	9e-05	14,14,15	12,12,12	11,11,11	10,10,10	8, 8, 8
2	2e-10	1e-09	8e-09	7e-08	6e-07	21,21,21	19,19,19	17,17,17	15,15,15	13,13,13
3	9e-14	8e-13	2e-11	4e-10	4e-09	29,29,29	26,26,26	23,23,23	20,20,20	18,18,18
4		6e-14	6e-14	1e-12	4e-11		28,28,28	28,28,28	26,26,26	22,22,22
5				6e-14	3e-13				28,28,28	27,27,27
6					6e-14					28,28,28

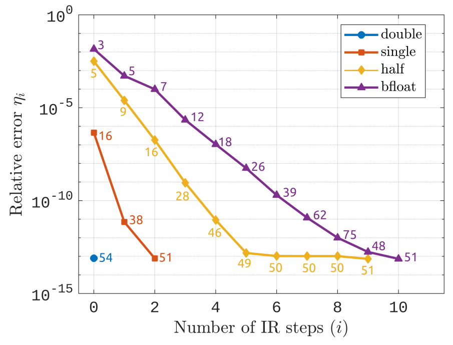
Table 5.2: Relative error η_i and $\textsc{Rank}(F_i)$ at different steps i and for different values of θ, for QRCP decomposition (using fp16 as u_ℓ and exponential distribution of singular values).

	Relative error η_i					$\textsc{Rank}(F_i)$				
$i\|\theta_\ell$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	2e-03	2e-03	2e-03	2e-03	3e-03	13	9	9	7	6
1	2e-05	1e-05	1e-05	5e-05	1e-04	25	18	13	10	9
2	5e-08	8e-08	7e-08	1e-07	1e-06	72	43	20	16	14
3	2e-10	2e-10	4e-10	8e-10	8e-09	95	86	36	21	19
4	2e-12	2e-12	3e-12	2e-12	4e-11	27	27	29	27	24
5	7e-13	7e-13	7e-13	7e-13	1e-12	28	28	28	28	27
6					7e-13					28

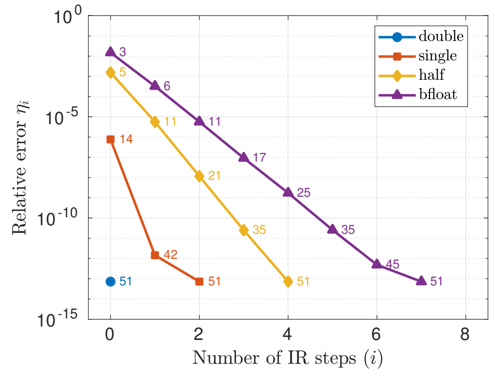
numerical rank. Empirically, we have observed QRCP to be the most sensitive of the LRA kernels, and to a lesser extent the randomized SVD kernel; the other kernels behaved well even for large θ_ℓ. For this reason, in our experiments we have set $\theta_\ell = 2^{-1}$ for all kernels except QRCP and randomized SVD, for which we have set $\theta_\ell = 2^{-3}$ and $\theta_\ell = 2^{-2}$, respectively. This setting allows to keep the ranks contained except for a few sporadic cases when using half precision. It must be noted that the sensitivity to rounding errors is also matrix (or tensor) dependent; however, we do not tune θ_ℓ individually for each matrix/tensor, because this would not be representative of a realistic scenario where it is unknown in advance how sensitive the data at hand actually is.

5.4. Results on real-life data. Finally, we also experiment our method on data sets (one matrix and one tensor) coming from real-life applications. The matrix corresponds to a subdomain in the discretization of the Poisson equation. It is of size 253×252 and its singular values decay relatively rapidly (not shown). The tensor comes from a quantum chemistry application and corresponds to an eigenfunction of the Hamiltonian operator for computing the vibrational spectrum of the H2CO molecule [30]. Here, the tensor is 6th-order ($d = 6$), of size $17 \times 17 \times 13 \times 13 \times 9 \times 9$, and obtained from an eigenvalue computation in which both the operator and vectors are compressed in the TT format throughout (and we obtain the full tensor with Decompress in high precision in the end), and the precision is set to 10^{-9} to prevent rank explosion.

We report the results for these two problems in Figure 5.5-Figure 5.6 with all the corresponding LRA kernels, using the same legend and methodology as for the random data experiments. Overall, the results lead to the same trends and conclusions as the previous experiments; ranks progressively increase with the IR steps, and our method manages to achieve high precision levels in all cases. In the TTSVD case, ranks stop increasing around the compute precision of the application (10^{-9}), whereas in HOSVD and HTSVD cases we still observe a significant rank increase beyond this threshold. This is mostly due to the fact that the underlying tensor decomposition of this tensor is TT by construction, and this topology mismatch results in higher ranks.



(a) QRCP (Poisson matrix).



(b) Randomized SVD (Poisson matrix).

Fig. 5.5: Convergence of Algorithm 2.1 for a real-life matrix (Poisson) of size 253×252, for different LRA kernels. The numbers next to each marker indicate the rank of F_i after recompression.

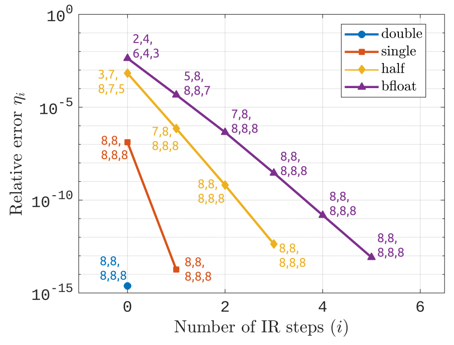
5.5. Estimation of the time cost and role of ω_ℓ.

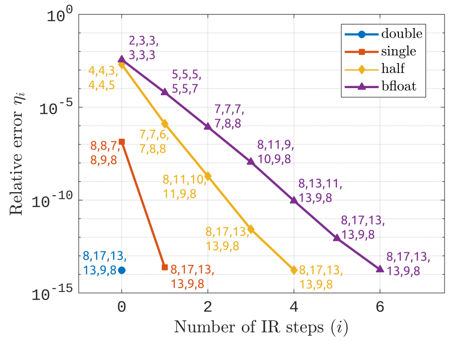
Based on the results obtained in the previous experiments, we can also obtain a rough estimation of the time cost of each method, using the formulas (2.31) (for classical hardware) or (2.34) (for hardware with block FMA units). This estimation depends on the number of iterations needed to converge to the desired accuracy, the ranks at each iteration, and the relative speed of the low precision arithmetic ω_ℓ.

We illustrate how this estimation can be obtained by using the case of randomized SVD with the Poisson matrix (Figure 5.5b) as an example. We consider two possible settings:

- Setting 1: classical (CPU) hardware, where the relative speed of each arithmetic is proportional to the number of bits. For this setting we target an accuracy of $\varepsilon = 10^{-12}$ and we use fp64 as the high precision and fp32 as the low precision (hence $\omega_\ell = 0.5$).
- Setting 2: GPU hardware with tensor core units (a type of block FMA). We target an accuracy of $\varepsilon = 10^{-5}$ and we use fp32 as the high precision and fp16 as the low precision. On most tensor core architectures (A100 and H100 in particular), fp16 arithmetic is 16 times faster than fp32, hence we take $\omega_\ell = \frac{1}{16} = 0.0625$.

In setting 1, our IR method (red curve in Figure 5.5b) converges in 2 iterations to an accuracy of $\varepsilon = 10^{-12}$. We use formula (2.31) with $\textsc{Rank}(X, 10^{-6}) = 14$,


(a) TTSVD (H2CO tensor).


(b) HOSVD (H2CO tensor).

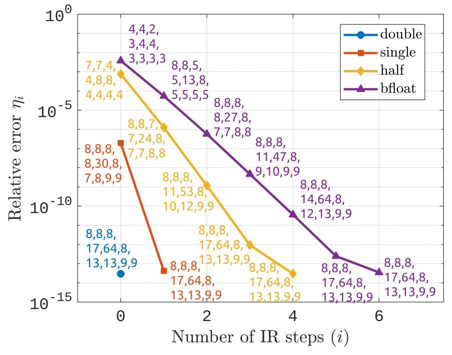

(c) HTSVD (H2CO tensor).

Fig. 5.6: Convergence of Algorithm 2.1 for a real-life tensor (H2CO) of size $17 \times 17 \times 13 \times 13 \times 9 \times 9$, for different LRA kernels. The numbers next to each marker indicate the rank of F_i after recompression.

$\textsc{Rank}(X, 10^{-12}) = 42$, $c_1 = 6$, and $c_2 = 2$ to obtain an estimated cost of

$$p \times \big(0.5 \times 6 \times (14 + 14 + 42) + 2 \times 14 + 3\big) = 241p.$$

This is to be compared with simply computing the LRA directly in fp64 arithmetic, for which formula (2.32) gives an estimated time cost of $6 \times 42 \times p = 252p$. Hence, we can expect a time reduction of about 4%. This example illustrates that even with classical hardware where the low precision is “only” twice as fast as the high precision, our IR method can still achieve moderate speedups in the case where the ranks decrease rapidly at low accuracies, such as for this Poisson matrix. This is however not the case for other matrices, including the random synthetic ones used previously.

In setting 2, our IR method (yellow curve in Figure 5.5b) converges in 2 iterations to an accuracy of $\varepsilon = 10^{-5}$. We use formula (2.34) with $\textsc{Rank}(X, 10^{-3}) = 5$, $\textsc{Rank}(X, 10^{-5}) = 11$, $c_1 = 6$, and $c_2 = 2$ to obtain an estimated cost of

$$p \times \Big(0.0625 \times \big(6 \times (5 + 5 + 11) + 2 \times 5\big) + 3\Big) = 11.5p.$$

This is to be compared with simply computing the LRA directly in fp32 arithmetic, for which formula (2.32) gives an estimated time cost of $6 \times 11 \times p = 66p$. Hence we can expect a significant time reduction by a factor of almost $6\times$. This example illustrates that our IR method is particularly attractive on hardware with very fast low precision arithmetic.

The above examples are only meant to illustrate the trends that one might expect on different types of hardware and for different target accuracies and low precisions. This analysis can be similarly applied to the other matrices, LRA algorithms, and to tensors.

6. Conclusion.

We have presented a new mixed precision iterative refinement algorithm for computing low-rank matrix and tensor approximations. The algorithm first computes a low-rank approximation in low precision, and then computes another low-rank approximation of the error term, also in low precision, to refine the accuracy of the approximation. The process can be repeated to further refine the accuracy, and we ensure the rank of the approximation remains bounded by using inexpensive recompression operations. We have performed the error analysis of this algorithm, which proves that the low precision determines the convergence speed, whereas the attainable accuracy only depends on the high precision. Therefore, high accuracy can be attained, even though most of the operations are performed in low precision. This makes the algorithm computationally attractive, and we have performed a complexity analysis to determine specific conditions under which we can expect it to be cheaper than simply computing a low-rank approximation directly in high precision. We have applied our algorithm to various matrix and tensor low-rank approximations algorithms, and performed MATLAB experiments that confirm its robustness and convergence in a wide range of settings.

This paper lays the theoretical and algorithmic foundations for a mixed precision iterative refinement framework for matrix and tensor low-rank approximations. In future work, we will develop high performance implementations of these algorithms to tackle the low-rank approximation of large scale matrices and tensors on modern parallel architectures, especially those equipped with GPU accelerators, which provide extremely fast low precision arithmetic. We have presented preliminary performance results in [3], which show that Algorithm 2.1 with randomized SVD, using fp16 as the low precision and fp32 as the high precision, can achieve up to a $2.2\times$ speedup with respect to randomized SVD in fp32, while achieving an accuracy that is equivalent or better.

Our error analysis is based on a general model that assumes the use of numerically stable implementations for the basic kernels (in particular, low-rank approximation and recompression). For tensors, such computations are experimentally observed to behave stably in practice, but formally proving their stability remains an open problem which provides an interesting perspective for future work.

Acknowledgements. This work was performed thanks to generous grants from ANR (JCJC SELESTE, ANR-20-CE46-0008-01) and Paris Ile-de-France Region (DIM RFSI RC-TENSOR No 2021-05). We thank Olivier Coulaud for providing the H2CO tensor data.

REFERENCES

[1] E. Agullo, O. Coulaud, L. Giraud, M. Iannacito, G. Marait, and N. Schenkels, *The backward stable variants of GMRES in variable accuracy*, Research Report RR-9483, Inria, Sept. 2022, https://inria.hal.science/hal-03776837.

[2] P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary, *Mixed precision low rank approximations and their application to block low rank LU factorization*, IMA J. Numer. Anal., (2022), https://doi.org/10.1093/imanum/drac037.

[3] M. Baboulin, S. Donfack, O. Kaya, T. Mary, and M. Robeyns, *Mixed precision randomized low-rank approximation with GPU tensor cores*, in Euro-Par 2024: Parallel Processing, Cham, 2024, Springer Nature Switzerland, pp. 31–44, https://doi.org/10.1007/978-3-031-69583-4_3.

[4] B. W. Bader and T. G. Kolda, *Algorithm 862: MATLAB tensor classes for fast algorithm prototyping*, ACM Trans. Math. Software, 32 (2006), pp. 635–653, https://doi.org/10.1145/1186785.

[5] C. Beltrán, V. Noferini, and N. Vannieuwenhoven, *When can forward stable algorithms be composed stably?*, IMA J. Numer. Anal., 44 (2024), pp. 886–919, https://doi.org/10.48550/arXiv.2109.10610.

[6] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, *Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores*, SIAM J. Sci. Comput., 42 (2020), pp. C124–C141, https://doi.org/10.1137/19M1289546.

[7] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic, *Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions*, Foundations and Trends® in Machine Learning, 9 (2016), pp. 249–429, https://doi.org/http://dx.doi.org/10.1561/2200000059.

[8] P. Comon, *Tensors: a brief introduction*, IEEE Signal Processing Magazine, 31 (2014), pp. 44–53, https://doi.org/10.1109/MSP.2014.2298533.

[9] M. P. Connolly, N. J. Higham, and S. Pranesh, *Randomized low rank matrix approximation: Rounding error analysis and a mixed precision algorithm*, MIMS EPrint 2022.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, July 2022, http://eprints.maths.manchester.ac.uk/2884/. Revised March 2023.

[10] O. Coulaud, L. Giraud, and M. Iannacito, *A robust GMRES algorithm in tensor train format*, arXiv preprint arXiv:2210.14533, (2022).

[11] L. De Lathauwer, B. De Moor, and J. Vandewalle, *A multilinear singular value decomposition*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1253–1278, https://doi.org/10.1137/S0895479896305696.

[12] C. Eckart and G. Young, *The approximation of one matrix by another of lower rank*, Psychometrika, 1 (1936), pp. 211–218, https://doi.org/10.1007/BF02288367.

[13] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis, *Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores*, SIAM J. Sci. Comput., 45 (2023), pp. C1–C19, https://doi.org/10.1137/21m1465032.

[14] G. H. Golub and C. F. Van Loan, *Matrix computations*, JHU press, 2013, https://doi.org/10.1137/1.9781421407944.

[15] L. Grasedyck, *Hierarchical singular value decomposition of tensors*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2029–2054, https://doi.org/10.1137/090764189.

[16] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitteilungen, 36 (2013), pp. 53–78, https://doi.org/10.1002/gamm.201310003.

[17] W. Hackbusch and S. Kühn, *A new scheme for the tensor representation*, Journal of Fourier analysis and applications, 15 (2009), pp. 706–722, https://doi.org/10.1007/s00041-009-9094-9.

[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002, https://doi.org/10.1137/1.9780898718027.

[19] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, Acta Numerica, 31 (2022), pp. 347–414, https://doi.org/10.1017/s0962492922000022.

[20] N. J. Higham and S. Pranesh, *Simulating low precision floating-point arithmetic*, SIAM J. Sci. Comput., 41 (2019), pp. C585–C602, https://doi.org/10.1137/19M1251308.

[21] N. Kishore Kumar and J. Schneider, *Literature survey on low rank approximation of matrices*, Linear Multilinear Algebra, 65 (2017), pp. 2212–2244, https://doi.org/10.1080/03081087.2016.1267104.

[22] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51

(2009), pp. 455–500, https://doi.org/10.1137/07070111X.

[23] D. Kressner and C. Tobler, *Algorithm 941: htucker—a MATLAB toolbox for tensors in hierarchical tucker format*, ACM Trans. Math. Softw., (2014), https://doi.org/10.1145/2538688.

[24] P.-G. Martinsson and S. Voronin, *A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices*, SIAM J. Sci. Comput., 38 (2016), pp. S485–S507, https://doi.org/10.1137/15M1026080.

[25] 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, HPCAsia '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 92–101, https://doi.org/10.1145/3368474.3368479.

[26] H. Ootomo and R. Yokota, *Recovering single precision accuracy from Tensor Cores while surpassing the fp32 theoretical peak performance*, Int. J. High Perform. Comput. Appl., 36 (2022), pp. 475–491, https://doi.org/10.1177/10943420221090256.

[27] H. Ootomo and R. Yokota, *Mixed-precision random projection for RandNLA on tensor cores*, 2023, https://doi.org/10.1145/3592979.3593413.

[28] I. Oseledets, *TT-Toolbox v2.2.2*, 2023, https://github.com/oseledets/TT-Toolbox.

[29] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317, https://doi.org/10.1137/090752286.

[30] M. Rakhuba and I. Oseledets, *Calculating vibrational spectra of molecules using tensor train decomposition*, The Journal of Chemical Physics, 145 (2016), p. 124101, https://doi.org/10.1063/1.4962420.

[31] L. R. Tucker, *Some mathematical notes on three-mode factor analysis*, Psychometrika, 31 (1966), pp. 279–311, https://doi.org/10.1007/BF02289464.

[32] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, *A new truncation strategy for the higher-order singular value decomposition*, SIAM J. Sci. Comput., 34 (2012), pp. A1027–A1052, https://doi.org/10.1137/110836067.

[33] Z. Yang, J. Shan, and Z. Zhang, *Hardware-efficient mixed-precision CP tensor decomposition*, arXiv preprint arXiv:2209.04003, (2022), https://doi.org/10.48550/arXiv.2209.04003.

Appendix A. Randomized SVD-based Recompress algorithm.

To recompress a (non-optimal) low-rank matrix $X = U_{in}\Sigma_{in}V_{in}^T$ to an optimal representation $U_{out}\Sigma_{out}V_{out}^T$ using randomized SVD, Algorithm 3.2 must be adapted as outlined in Algorithm A.1. The matrix products on Lines 6 and 8 of Algorithm A.1 can be efficiently performed by exploiting the low-rank structure of X. We stop the main loop when the norm of the low-rank matrix $Q_E R_E$ becomes smaller than the desired accuracy ε. This $Q_E R_E$ matrix is initially equal to the input matrix X and is incrementally deflated at each step by $Q_b B_b$. In order to measure its norm, we reorthonormalize Q_E at each step and measure the norm of R_E.

Algorithm A.1 RECOMPRESS algorithm using randomized SVD decomposition.

Input: a truncated SVD $U_{in}\Sigma_{in}V_{in}^T \in \mathbb{R}^{m\times n}$, the target accuracy ε, and a block size b.

Output: a recompressed SVD $U_{out}\Sigma_{out}V_{out}^T$ decomposition.

```
 1: Initialize $Q$ and $B$ to empty matrices
 2: Initialize $Q_E = U_{in}$ and $R_E = \Sigma_{in}V_{in}^T$.
 3: $n_X = \|\Sigma_{in}\|$
 4: repeat
 5:     Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times b}$.
 6:     $Y = U_{in}\Sigma_{in}(V_{in}^T\Omega)$
 7:     $Q_b = \mathtt{qr}(Y - Q(Q^T Y))$
 8:     $B_b = (Q_b^T U_{in})\Sigma_{in}V_{in}^T$
 9:     $Q \leftarrow [Q\ Q_b]$
10:     $B \leftarrow \begin{bmatrix} B \\ B_b \end{bmatrix}$
11:     $Q_E \leftarrow [Q_E\ Q_b]$
12:     $R_E \leftarrow \begin{bmatrix} R_E \\ -B_b \end{bmatrix}$
13:     $Q_E\overline{R} = \mathtt{qr}(Q_E)$
14:     $R_E \leftarrow \overline{R}R_E$
15: until $\|R_E\| \le \varepsilon n_X$
16: Compute the truncated SVD decomposition $B \approx \overline{U}\Sigma_{out}V_{out}^T$ at accuracy $\varepsilon$.
17: $U_{out} = Q\overline{U}$
```