Optimizing Sparse Tensor Train Decomposition to Solve Linear Systems of Equations

Csanád Bakos

Optimizing Sparse Tensor Train Decomposition to Solve Linear Systems of Equations

by

Csanád Bakos

Student Number: 4892364

Thesis supervisors (TUDelft):	Prof. Dr. ir. Robert Kooij, Dr. ir. Huijuan Wang
Thesis supervisors (Deltares):	Dr. ir. Xiaohan Li, Dr. ir. Didrik Meijer
Project Duration:	November, 2023 - August, 2024
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science
University:	Delft University of Technology

Cover: Estuaries near the coast of Guinea–Bissau branch out like a network of roots from a plant, under CC0 License.

Preface

This thesis is the culmination of my Master of Science studies in the field of computer science at the Delft University of Technology. My interest in the research area of this thesis was sparked by a course on Tensor Networks taught by Borbála Hunyadi and Kim Batselier. It has been a rewarding journey to explore this topic in depth with practical applications in mind.

I want to express my deepest gratitude to my thesis supervisors at Deltares, Xiaohan Li and Didrik Meijer, whose guidance, patience, and insights were invaluable throughout this project. I am also thankful to my academic advisors at the TUDelft, Robert Kooij and Huijuan Wang, for their encouragement and constructive feedback.

I would also like to acknowledge the assistance of Kim Batselier and Jan Noort, who have helped greatly with refining the project's direction through our repeated discussions. My sincere thanks go to Deltares for welcoming me as an intern for the duration of this thesis.

Special thanks go to my family and friends, who have been a constant source of support and motivation during this challenging journey.

Throughout this process, I encountered numerous challenges, from analysing the time complexity of highly complex algorithms to debugging advanced tensor library functions. Overcoming these challenges has taught me valuable lessons in perseverance, problem-solving, and the practical application of theoretical concepts.

As I conclude this thesis, I hope that my research will contribute to the ongoing advancements relating to low-rank tensor approximations. Especially, improving the scalability of solving linear systems of equations and inspiring further exploration in this area.

Thank you to everyone who has been part of this journey.

Csanád Bakos
Delft, August 2024

Summary

This thesis investigates the potential of solving large-scale linear systems of equations (LSEs) in the Tensor Train (TT) format. First, we study when this approach can outperform traditional solvers like Conjugate Gradient (CG) and Gaussian Elimination (GE). In turn, we examine the time complexity of the TT-solve technique when applied to ill-conditioned, sparse, and symmetric positive definite (SPD) matrices. This enables us to assess the scenarios where TT-solve may offer significant advantages.

During this exploration, we have also identified a handful of research gaps that could further optimize the TT-solve process. These include challenges related to poorly factorizing matrix sizes, as well as the need for a deeper understanding of the trade-offs between TT-matrix (TTM) rank and maximal mode size during the decomposition process. Additionally, the study examined the potential for altering the coefficient matrix through techniques like variable reordering and partial Gaussian elimination (PG) to achieve lower TTM-ranks without affecting the solution of the LSE.

The research led to several important insights. Methods like padding and PG effectively addressed issues with matrices that have sizes with poor prime factorizations, although they were less effective in runtime reductions for matrices with well-factored dimensions. Balancing mode size and TTM-rank was crucial, with the largest prime factor generally providing good results, though alternative factors occasionally performed better. The impact of matrix structure on TT-decomposition was significant; techniques such as variable reordering and the reverse Cuthill-McKee (RCM) algorithm could improve factorizations, though results varied depending on the matrix. The improved TT-solve approach showed potential to be competitive with traditional solvers like Conjugate Gradient (CG) and Gaussian Elimination (GE), though further refinements are necessary for consistent advantage.

A practical method for assessing TT-solve suitability through runtime estimates and minimal padding checks was highlighted, offering a way to predict potential performance improvements. However, TT-solve has limitations and cannot yet replace existing solvers for all types of sparse, ill-conditioned, symmetric, positive definite systems, that we have studied.

The study's limitations include reliance on theoretical runtime assessments that may not fully capture practical factors like hardware optimizations. The dataset was limited to symmetric, positive definite matrices from computational fluid dynamics, which constrains the generalizability of the findings. The analysis also assumed rapid convergence of TT-solve, which may not apply in all cases. Furthermore, strong assumptions were made regarding the TT-ranks of the right-hand side and solution vectors in the LSEs. Comparative analysis with traditional solvers was based on theoretical estimates, potentially overlooking existing improvements such as preconditioners.

Future research should focus on expanding the dataset to include a wider variety of matrices and incorporate statistical methods to better isolate the effects of different enhancements. Investigating the impact of the proposed strategies on TT-ranks of the right-hand side and solution vectors could provide additional insights. Exploring iterative solvers like GMRES with appropriate preconditioning for TT-subproblems and alternative matrix reorganization methods are expected to further enhance efficiency.

The thesis has made significant strides in optimizing TT-decompositions for solving large-scale sparse LSEs. The proposed methods, including padding, PG, RCM and tile size optimization, have shown promise in improving TT-solve efficiency. These advancements address practical challenges in computational problem-solving and offer valuable insights for further research and application in scientific and engineering fields that aim to utilise TT for solving LSEs.

Contents

1 Introduction

In many fields such as economics, climate science, finance, and engineering, mathematical models play a crucial role in understanding complex systems and predicting future outcomes. These models often involve partial differential equations (PDEs), which, when discretized, lead to large linear systems of equations (LSEs). The accurate and timely solution of these LSEs is essential for making informed decisions, be it predicting floods or implementing economic policies. Naturally, we aim to uncover the mechanics of more and more complex systems, run simulations at larger scales and increase the accuracy of our predictions. This means that the computational burden also increases, making existing methods often too slow to keep up with the changing requirements.

A significant portion of these computational challenges can be traced back to the need to solve large LSEs, as elaborated in Appendix A in the context of hydrodynamic simulations. To address these, various strategies have been developed to accelerate computations. One common approach is to reduce redundancies in the data, such as using sparse matrix representations instead of dense ones. Here, instead of storing and working with each entry explicitly, we only use the nonzero entries for the computations. This method leverages the fact that many entries in these matrices are zero, allowing for fewer operations and reduced storage requirements, which in turn lead to faster computations that scale better with the problem size.

Beyond simply exploiting sparsity, another promising direction is to compress the nonzero entries themselves by identifying and utilizing additional structure within the data. One such approach is low-rank approximation, where a matrix can be represented with significantly fewer entries as a product of smaller matrices, without losing (much) accuracy. We illustrate the idea in Figure 1.1.

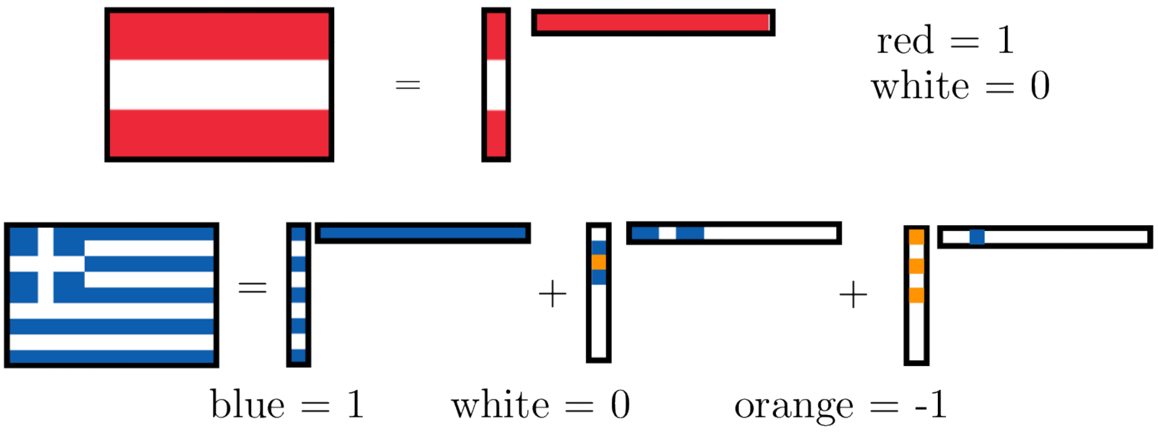


Figure 1.1: The flags of Austria and Greece as matrices, illustrating low rank decompositions. Image source: [1].

In turns out that many matrices that occur in practice can have low numerical rank [1, 2]. There are various arguments why this happens. It may occur due to the well-separated nature of variables [1]. For instance, considering the weather, given enough distance in space or time, factors of interest will have little to no influence on each other. Therefore, one has no need to encode the connections between every single element in a model. On the other hand, our world often appears to be 'smooth'. I.e. phenomena that are 'close' to each other, e.g. in space or time, can be often described by slowly varying functions. This means that we can represent a whole set of data points with a polynomial function that has only a handful of parameters, yielding small ranks [2]. In the context of water flows, for example, the velocity tends to be close to how it was a few seconds ago or how it is a few meters along a river bed. Of course, it is not hard to find exceptions to these examples - think of a waterfall for instance - however, these should help with gaining some intuition why low rank structures appear in our models. Lastly, we note that there are more arguments for the common appearance of low-rank structures, including correlations and the so called Sylvester equation [1].

Generally speaking, flattening higher dimensional data to a matrix breaks the patterns that would allow for low rank approximations. For a lower dimensional illustration, consider Figure 1.2. In such scenarios, tensor decompositions can offer value over matrix factorizations.

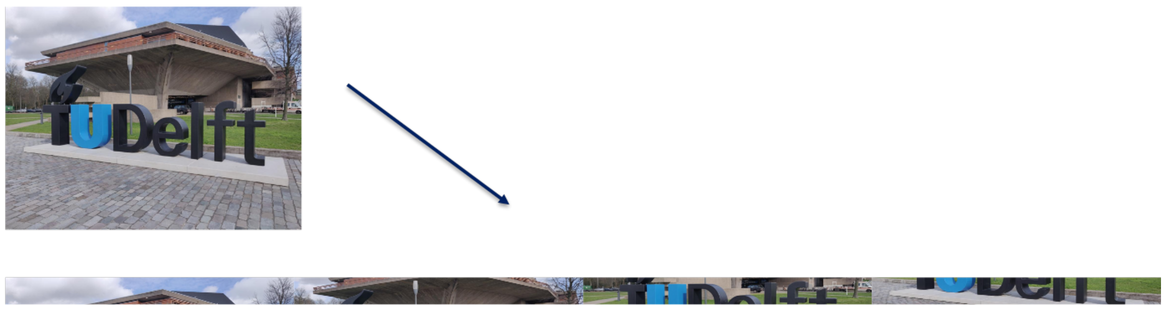


Figure 1.2: Information loss due to data flattening. Image source: [3]. In the original image, the neighboring pixels - meaning the adjacent entries in the matrix - are placed next to each other. Therefore, the patterns such as the TUDelft sign are clearly identifiable. When we slice up the image and flatten the data, the entries that previously showed clear arrangements together, are now separated. This breaks the structure present in the starting picture. We note that for a true mode size reduction, we would have to slice up the mode-2 image so that each slice only contains a single row of pixels. Concatenating these would yield a single (very long) row of pixels. For the purposes of this illustration, it is sufficient that more than one pixel row per slice was taken.

Having a low-rank factorization of the LSE is not the end of the story. A challenge remains: the system still needs to be solved efficiently after the decomposition. This is where formats like the Tensor Train (TT) representation become useful. TT not only allows for a more compact representation but also supports a range of operations that can be performed cheaply, including solving LSEs [4, 5, 6, 7, 8].

Despite the potential of the TT format, one of the key challenges has been the time required for the conversion and solution in the TT format. Considering the time it takes to convert a LSE to the TT-format can already be prohibitively expensive. The classical conversion algorithm, called TT-SVD, for example, is too slow to be practical for converting large systems [9, 10]. However, recent advances in TT research, particularly in sparse decomposition methods for matrices and tensors, offer promising directions. For instance, works such as [9] and [10] have developed faster algorithms that could make the TT approach more viable for real-world applications. While these new algorithms, represent significant progress, there remain open questions.

In this thesis, we uncover and propose strategies to address the following two research gaps. First, the size of a matrix can largely hinder the possibility of achieving a low-rank TT-decomposition, despite the matrix possessing low-rank structure across its entries. Second, there are tradeoffs to be made between the parameters of sparse TT-decompositions to minimize the time needed to find the LSE solution. Additionally, we revisit a previous inquiry in [10] that made alterations to the matrix, leading to better TT-decompositions, while leaving the solution to the linear system unchanged.

The rest of this study is organized as follows. In Chapter 2, we give the necessary background on a number of topics that are crucial for this exploration. First, we look at ways to assess runtime complexities of algorithms, followed by a recap of existing solvers for LSEs. Then, relevant matrix decompositions, and tensors are introduced. Additionally, we build up to the TT-decomposition and introduce the available algorithms to solve LSEs in this format. We close the chapter by looking at a few matrices and their different types of ranks. The next part, Chapter 3 presents our methodology, and our insights regarding the time complexity analysis of the TT-solve approach. This assessment leads us to uncover a number of research gaps, which we develop strategies for to tackle. We close this chapter by detailing our experiment setup. Afterwards, Chapter 4 illuminates our experimental results for the proposed strategies in isolation and when they are combined. Adding to this, we assess how TT-solve compares to the state of the art solvers for LSEs, with and without our improvements. As we come to the end of our study, in Chapter 5 we discuss the limitations of our approach, point out the key takeaways and give suggestions for future research. Lastly, Chapter 6 concludes our inquiry to this subject.

2 Background

In this chapter, we introduce the relevant literature that constitutes as the background of our study. First, we look at how runtimes of algorithms can be analysed. Next, we give details regarding the standard solvers for linear systems of equations (LSEs), with a focus on their time complexity. This is followed by revising a handful of matrix decomposition approaches that serve as building blocks for other methods we consider. The next topic is tensors, and algorithms to find low-rank approximations of these higher dimensional generalizations of matrices. We concentrate on the Tensor Train (TT) decomposition and how LSEs can be solved in this format. Lastly, we highlight some observations regarding the different types of ranks that matrices can have.

2.1. Runtime analysis approaches

The runtime of algorithms is a central theme in this thesis, therefore it is important to understand the framework available for its analysis. In this section, we explore various time complexity analysis approaches, including their limitations.

Empirical analysis This involves measuring the actual runtime of algorithms on various inputs using experimental techniques. By conducting experiments on real-world data sets or synthetic test cases, empirical analysis provides concrete insights into algorithm performance under specific conditions. Its simplicity comes with various drawbacks such as high computational resource needs, limited generalizability, sensitivity to the software and hardware environments and challenges with interpretation. Nevertheless, this approach is valuable when theoretical analysis is impractical or when considering factors not captured by theoretical models.

Counting primitive operations In order to enable a more generalizable runtime assessment, one can count the number of *primitive operations* of an algorithm as a function of the input size and other relevant parameters. This provides a direct measure of computational workload. These primitive operations include, for instance, assigning a value to a variable, performing an arithmetic operation, comparing two numbers or calling a method [11].

In numerical algorithms and scientific computing applications so called floating point operations (FLOPs) dominate, which are a subset of the primitive operations. Due to their prevalence, in such applications other type of primitive operations are often ignored for simplicity and since they don't have significant effect on the time complexity.

An important assumption here is that each primitive operation can be done in constant amount of time as memory access is unrestricted. In certain cases, memory bandwidth can dominate runtime, rendering this type of analysis ill-posed. Another limitation is how different primitive operations may be much faster than others thanks to dedicated hardware circuits [12]. A good example is using a GPU to perform fast matrix-matrix multiplications.

Finally, it should be noted that counting all such operations can be very tedious, especially for complex algorithms. Therefore, in practice, we often only consider the ”most time consuming parts”. This is formalized using the big-$\mathcal{O}$ notation which is described next.

Big-$\mathcal{O}$ notation This technique provides an asymptotic upper bound on an algorithm's time complexity, indicating how its runtime grows with the size of the input. It abstracts away constant factors and lower-order terms, focusing on the dominant behavior as the input size approaches infinity [13]. While widely used for its simplicity and generality, it has clear limitations. By neglecting constant factors and smaller order terms it often cannot capture the nuances of performance for smaller inputs. Also, it only gives an upper bound, therefore in many algorithms, depending on the structure of the input, the runtime may be significantly reduced.

Average-case analysis Instead of looking at the worst-case, one can consider the expected performance of an algorithm over all possible inputs. This provides insights into the typical behavior of the algorithm. By accounting for input distributions, average-case analysis can offer a more realistic assessment of algorithmic efficiency. In practice, it is often difficult to assess the distribution of inputs which hinders the applicability of this approach.

2.2. Solvers for linear systems of equations

Linear systems of equations (LSEs) are fundamental mathematical problems encountered across various fields including physics, engineering, computer science, and economics. As elaborated in Appendix A, they also arise when discretizing PDEs via FDM or FVM, as it is the case in hydrodynamic simulations. In this approach, solving LSEs remains computationally the most intensive part of the process [14].

Given a set of linear equations, the goal is to find a solution vector that satisfies all equations simultaneously. There are two main approaches to solving such systems: direct and iterative methods. Let $Ax = b$ be a linear system of equations, where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$. While in most cases, the linear systems in this study have $n = m$, in certain subproblems we work with rectangular matrices where $m \geq n$. We use z to denote the number of nonzero entries in A.

2.2.1. Direct solvers

Direct solvers aim to find the exact solution to the linear system in a finite number of steps, typically through a series of algebraic manipulations. These methods provide accurate solutions but may become computationally expensive for large-scale problems due to their high memory requirements and computational complexity. In this subsection, we introduce the main idea and time complexity of the standard algorithms of this approach.

Gaussian elimination (GE)

Gaussian elimination is one of the most widely used direct methods for solving linear systems. It involves transforming the system of equations into an equivalent upper triangular form through a sequence of row operations, such as row additions and row swaps. Once the system is in triangular form, back substitution is used to find the solution. The time complexity of this approach in terms of flop count, is $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n + \mathcal{O}(1)$ [15, 16]. We note that for a matrix with a bandwidth of w (as defined in Appendix C.8) and size n, we get a runtime of $\mathcal{O}(w^2 n)$ [17]. This can be much faster than $\mathcal{O}(n^3)$ when $w \ll n$.

LU decomposition

LU decomposition is another popular direct method that decomposes the coefficient matrix of the linear system into the product of a lower triangular matrix (L) and an upper triangular matrix (U). This decomposition allows for efficient solving of multiple linear systems with the same coefficient matrix by solving two sets of triangular systems. Using partial pivoting, LU decomposition takes $\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n + \mathcal{O}(1)$ flops [18] to obtain the decomposition.

When the LU factorization is used to solve $Ax = b$, after the decomposition, we also need to compute the forward and backward substitutions. These take an additional $n^2 - n$ and n^2 flops, respectively [19, 20]. Therefore, the LU solver requires in total $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n + \mathcal{O}(1)$ flops to solve $Ax = b$.

Cholesky decomposition

Cholesky decomposition is a specialized form of LU decomposition applicable to symmetric and positive definite matrices. It decomposes the matrix into the product of a lower triangular matrix and its conjugate transpose, resulting in faster and more memory-efficient solutions compared to general LU decomposition. This takes $\frac{1}{3}n^3 + \mathcal{O}(n^2)$ flops to compute, making it about twice as efficient as LU factorization when applicable [21, 22].

2.2.2. Iterative solvers

Iterative solvers, unlike direct methods, generate an approximate solution through iterative refinement. These methods are often favored for large, sparse linear systems where direct methods may be impractical due to their $\mathcal{O}(n^3)$ computational complexity. Iterative solvers start with an initial guess for the solution and iteratively refine it until convergence criteria are met.

In this work, we mainly focus on symmetric, positive definite matrices, however a number of subproblems do not meet this requirement. To this end, we introduce both the Conjugate Gradient method (CG) [23] and the more widely applicable Generalized Minimal Residual (GMRES) [24] approach.

Conjugate Gradient (CG) method

The conjugate gradient method is an iterative solver specifically designed for symmetric positive definite matrices. It minimizes the error in the solution space by iteratively generating conjugate directions. The conjugate gradient method is particularly efficient for large sparse systems. For a detailed explanation about the method, see [25].

When considering the CG method, the dominating operation at each iteration is computing the matrix-vector product. This can be done in $\mathcal{O}(z)$ time [25]. Furthermore, in many application areas A is sparse enough to have $z \in \mathcal{O}(n)$ [25]. To quantify its overall runtime, we denote the *condition number* of the matrix as κ (see Appendix C.9 for the definition). Then, it can be shown that the number of iterations needed to satisfy an error bound ϵ is upper bounded by $\sqrt{\kappa}$. Thus, we have $\mathcal{O}(z\sqrt{\kappa})$ time complexity and $\mathcal{O}(z)$ space complexity for CG [25].

Generalized Minimal Residual method

The GMRES algorithm is the recommended algorithm in Matlab when solving LSEs formed by square matrices that don't have the positive definite property [26, 27]. While in certain cases, the related biconjugate gradient algorithms (e.g. BICG [28], BICGSTAB [29], CGS [30]) yield solutions quicker, their convergence behaviour is not well understood in general, making GMRES better suited to try first [26].

GMRES relies on computing a sequence of orthogonal vectors, combining them using a least-squares solution and update. It needs to store this sequence, yielding excessive storage requirements. This is mitigated in practice by the restarted GMRES(k) method, where k is the number of iterations before the algorithm is restarted [24, 27]. As described in Appendix B.2, assuming $\mathcal{O}(1)$ restarts are enough, the time complexity of the method is $\mathcal{O}(k^2n + kn^2)$ for the dense case. When faced with a sparse system, i.e. $z \in \mathcal{O}(n)$, this reduces to $\mathcal{O}(k^2n)$ flops.

Choosing k is problem dependent and no general guidelines are available. Another notable result is that, assuming exact arithmetic, the algorithm converges in at most n iterations [27]. However, this has little practical use as that means a runtime of $\mathcal{O}(n^3)$ which is prohibitive for large LSEs.

2.2.3. Preconditioners

The spectral composition of the coefficient matrix largely influences the convergence rate of iterative methods. To this end, we are often interested in transforming the LSE to an equivalent system with spectral properties that yield faster convergence. A *preconditioner* is a matrix that performs such an adjustment [27]. As an illustration, consider a matrix M that approximates the coefficient matrix A in some way. Then the system $M^{-1}Ax = M^{-1}b$ has the same solution as $Ax = b$ and may have favourable spectral properties.

Depending on the iterative solver in use, the preconditioner should have different characteristics. In the case of CG, the goal is to have $\kappa(M^{-1}A) < \kappa(A)$ as the convergence of the algorithm directly relates to the condition number. On the other hand, for GMRES one aims to choose one so that the eigenvalues of $M^{-1}A$ are better clustered [31].

Using preconditioners is not free, it takes time to construct a suitable M^{-1} and also at each iteration we need to apply it. Therefore, there is a tradeoff to be made between the time saved and the extra computational time spent [27]. While constructing universal preconditioners is easy, their effectiveness is highly problem dependent [32].

2.3. Matrix decompositions

In this section, we introduce the most prominent matrix factorization approach: Singular Value Decomposition (SVD) with a focus on its time complexity. Additionally, we look at the QR factorization. Both of them are key components of the Tensor Train algorithms we utilise in the rest of this study. Lastly, we make a note of other rank revealing factorizations that can serve as alternatives to SVD.

2.3.1. Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) is a widely used matrix factorization technique that decomposes a real or complex matrix into three matrices. An orthogonal matrix U, a diagonal matrix Σ containing the singular values, and the transpose of another orthogonal matrix V. Mathematically, for an $m \times n$ matrix A, the SVD is represented as $A = U\Sigma V^T$, where U is an $m \times m$ orthogonal matrix, Σ is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal (singular values), and V^T is the transpose of an $n \times n$ orthogonal matrix V [33]. The singular values in Σ represent the importance or strength of the corresponding singular vectors (columns of U and V), enabling low-rank approximations and noise reduction by discarding small singular values and their associated vectors [33, 34, 35]. Notably, it is proven to give the best low-rank approximation of a matrix [34].

Several approaches exist to compute the SVD. Depending on which method is used and what parts of the decomposition are needed, the time complexity changes up to a constant factor. Computing it is of order $\mathcal{O}(mn^2 + n^3)$ for an $m \times n$ matrix ($m > n$) or $\mathcal{O}(n^3)$ in the square case. The constant factor of the cubic term, hidden by the big-O notation, is generally in the range of $4 - 30$ [36].

Truncated SVD tSVD is a variant of the SVD method, where only the k most significant singular values and their corresponding singular vectors are computed. In applications where k is known in advance, this approach can be much faster, especially when $k \ll \min(m, n)$, since classical algorithms give results in $\mathcal{O}(mnk)$ time for dense matrices. Randomized variants reduce this to $\mathcal{O}(mn\log(k))$ flops. For sparse inputs the time complexity is retained, however, randomized methods are often more robust and can exploit multiprocessor architectures better than classical, Krylov subspace-based ones [37].

2.3.2. QR decomposition

The QR decomposition is a matrix factorization technique that takes an $m \times n$ matrix A and decomposes it into the product of an orthogonal $m \times m$ matrix Q and an upper triangular $m \times n$ matrix R, such that $A = QR$. To avoid unnecessary computations when $m > n$, in practice, often only a $m \times n$ matrix Q and an upper triangular $n \times n$ matrix R are computed [36].

One of the widely used algorithms to compute it is based on the numerically stable Householder transformations. Constructing the $m \times n$ Q and $n \times n$ R each take $2n^2m - \frac{2}{3}n^3$ flops. This yields a total of $4mn^2 - \frac{4}{3}n^3$ flops. In case, one needs the full $m \times m$ Q matrix, its computation takes $4m^2n - 4mn^2 + \frac{4}{3}n^3$. The $m \times n$ version of R takes the same flops to compute as earlier, therefore, we have $4m^2n$ flop count for this case [21, 36].

2.3.3. Rank revealing factorizations

SVD is the most well-known rank revealing factorization, however, there are various alternatives. There are rank revealing versions of the QR (RRQR) [38, 39], LU (RRLU) [40, 41] and Cholesky (RRCh) decompositions [42]. While the algorithms for these alternatives come with better theoretical time complexity (up to a constant factor), in practice, the lack of optimized LAPACK routines mean slower runtimes for RRLU and LLCh [43]. There exists better optimized versions of RRQR decomposition implementations that outperform SVD in practice [44]. The main reason why SVD remains preferable in most applications, is that these alternatives are not guaranteed to give the optimal ranks. I.e. if we want to find a low rank approximation with error tolerance ϵ, then SVD is guaranteed to give an approximation with the smallest possible rank that meets this criteria. On the other hand, the alternative methods tend to need higher ranks to meet the specified error tolerance [44, 45]. Additionally, for instance, the RRQR algorithms can fail to give a result on specific inputs, however, for most cases in practice this is not a concern [36, 46, 47]. Finally, it's worth mentioning that this is an active research area. Various algorithms have been proposed with different tradeoffs on computation time and guarantees on how good ranks they give [45].

Low-rank matrix approximations are a very powerful technique. However, as we have seen in Figure 1.2, when flattening higher dimensional data, we can lose essential patterns. This motivates the introduction of decomposition algorithms that can operate directly on higher dimensional arrays, also known as tensors. We look at these approaches next.

2.4. Tensors

Tensors are the higher dimensional generalization of matrices as illustrated by Figure 2.1. An Nth-order tensor is denoted as $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$. A specific entry indexed by $i_1, i_2, \ldots i_d$ is given by $x_{i_1, i_2, \ldots, i_d} = \underline{\mathbf{X}}(i_1, i_2, \ldots, i_d) \in \mathbb{R}$. We refer to the *order* of the tensor as the number of "modes", "dimensions" or "ways" it has. *Size* refers to the number values an index can take within a chosen mode [48].

Curse of Dimensionality First coined in [49], the term mostly refers to the exponentially increasing number of parameters needed to capture a high dimensional system's behaviour. In the case of tensors, this refers to the exponential number of entries with respect to the order of the tensor. I.e. for a tensor of order d, where each mode has size I, we need I^d entries to describe it exactly [48].

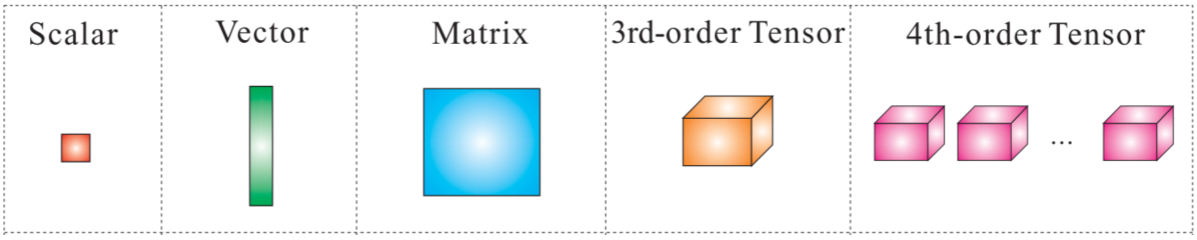


Figure 2.1: Graphical illustration of tensors (multi-way arrays), with increasing complexity. Source: [48].

2.5. Tensor decompositions

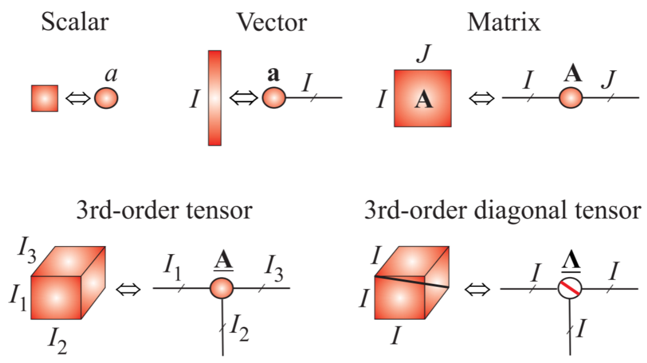
In certain cases, tensor decompositions allow one to alleviate the curse of dimensionality for tensors. This is achieved by exploiting inherent dependencies between the modes to find low-rank approximations of the data. While this comes at a cost of some accuracy, large amount of compression can be achieved [48].

One of the most well-known approach is the *Canonical Polyadic Decomposition (CPD)* that factorizes a tensor into a sum of rank-one tensors [50]. While CPD can reduce the exponential data size I^d to dIR where R is the number of summands in the factorization, it has a number of drawbacks. The minimal R needed to represent a tensor in this format is called the *tensor rank*, however, its computation is NP-Hard [4, 51]. When approximating with a fixed rank, computing CPD can be ill-posed [52] leading to failed convergence of numerical algorithms or getting stuck in local optima [4, 48].

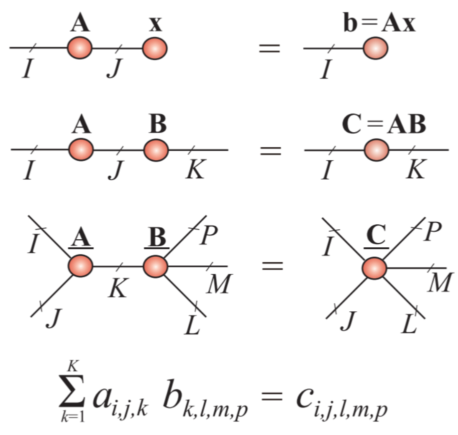
Another famous approach is the *Tucker format* which does not suffer from such numerical instabilities, however, it does not scale well with the number of modes of the tensor as it has an exponential dependence on it with its rank [4, 53, 54].

2.5.1. Tensor Train (TT) decomposition

The TT method, also known as the Matrix Product State (MPS) in the quantum physics community, occupies a middle ground between the CPD and Tucker formats. While it doesn't exhibit an inher-



(a) Tensor network diagram building blocks.



(b) Matrix-vector multiplication (top), matrix-matrix multiplication (middle) and tensor contraction (bottom) illustrated by tensor diagrams.

Figure 2.2: Graphical representation of tensors and the contraction operation. Source: [48].

ent exponential relationship with dimensionality (d), it maintains stability. This stability means that a high-quality approximation, with bounded TT-ranks, is always attainable. Achieving this quasi-optimal approximation (i.e. approximation error is bounded by a constant ϵ) generally involves a series of SVDs on auxiliary matrices [4, 5, 55].

It can be written in a handful of equivalent ways [4, 48]. In this work, we adopt the following notation. Let $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ be an order-d tensor. Then, the TT-decomposition may be written, using multilinear products of TT-cores, as $\underline{\mathbf{X}} \cong \underline{\mathbf{G}}^{(1)} \times^1 \cdots \times^1 \underline{\mathbf{G}}^{(d)}$. Here, $\underline{\mathbf{G}}^{(k)} \in \mathbb{R}^{s_k \times I_k \times s_{k+1}}$ is the k^{th} TT-core in the TT, $k \in \{1, 2, \ldots, d\}$ and $s_1 = s_{d+1} = 1$. We refer to the largest rank as *TT-rank*: $s = \max_k s_k$.

The Tensor Train Matrix (TTM), also known as the Matrix Product Operator (MPO), is a convenient way to apply the TT method for matrices. In particular, this approach, firstly represents a huge scale, structured matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$, as an order-$2d$ tensor, $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \ldots I_d \times J_d}$, where $n = I_1 I_2 \cdots I_d$ and $m = J_1 J_2 \cdots J_d$. Then the tensor can be transformed into a TTM with order-4 TT-cores. Thus, we have $\underline{\mathbf{X}} \cong \underline{\mathbf{G}}^{(1)} \times^1 \cdots \times^1 \underline{\mathbf{G}}^{(d)} \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \ldots I_d \times J_d}$, where $\underline{\mathbf{G}}^{(k)} \in \mathbb{R}^{r_k \times I_k \times J_k \times r_{k+1}}$ is the k^{th} TT-core and $r_1 = r_{d+1} = 1$ as for TT [48]. We will denote the maximal mode size of the TT-cores as $I = \max_k I_k$ and $J = \max_k J_k$. We call the TT-rank of a TTM, the TTM-rank, which is given by $r = \max_k r_k$.

Besides the aforementioned advantages, it is possible to perform various linear algebra operations in TT-format without exiting the TT-structure. These include the efficient computation of addition, matrix-by-vector multiplication and elementwise multiplication. Unfortunately, after such operations, the TT-ranks begin to grow. TT-ranks largely determine the storage needs of the TT representation and the time-complexity of these operations, therefore large TT-ranks must be mitigated. This is achieved by the TT-round procedure which can be used as a post-processing step after mathematical operations [4, 5, 55].

In the TTM approach, instead of employing conventional global low-rank matrix approximations, compression of large matrices involves low-rank approximations of block-matrices, organized hierarchically. However, to achieve a low-rank TTM and thereby achieve effective compression, the ranks of all corresponding unfolding matrices within a specific structured data tensor must be low, This means that their singular values must diminish quickly. Although this holds for numerous structured matrices, regrettably, it is not generally the case [48].

2.5.2. TT-decomposition algorithms

Given a higher order tensor or a matrix in a tensor format, a key question is how its TT representation can be computed. The classical algorithm for this purpose is the TT-SVD [4]. The key idea of the method is to compute successive truncated SVDs on the auxiliary matrices that originate from the reshaped input tensor. In this process, we have control over the TT-ranks or the error (ϵ) tolerance of

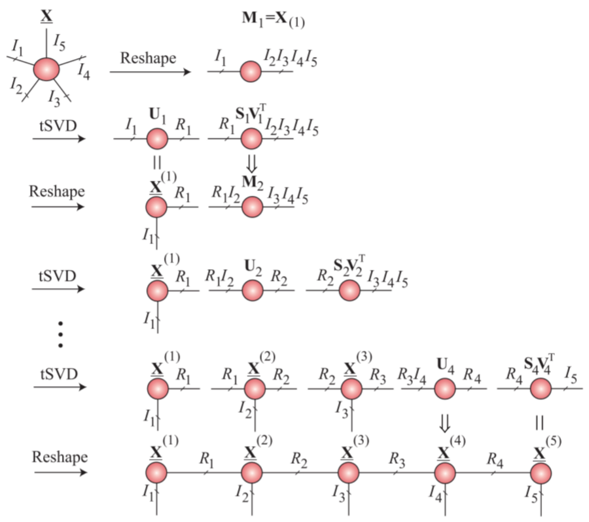


Figure 2.3: The TT-SVD algorithm for a tensor with 5 modes. Image source: [48]. Here tSVD is the algorithm from Section 2.3.1 when the TT-ranks are known in advance. Otherwise, it refers to the full SVD computation followed by a truncation. The top panel shows that the tensor $\underline{\mathbf{X}}$ with size $I_1 \times I_2 \times \cdots \times I_5$ is reshaped to an auxiliary, long matrix $\mathbf{M_1}$ of size $I_1 \times I_2 \cdots I_5$. In the 2nd stage, tSVD is applied to produce a low-rank matrix decomposition with $\mathbf{U_1}$ and $\mathbf{S_1 V_1^T}$ as factor matrices, such that $\mathbf{M_1} \cong \mathbf{U_1 S_1 V_1^T}$. The third panel illustrates that $\mathbf{U_1}$ becomes the first TT-core $\underline{\mathbf{X}}^{(1)}$ and $\mathbf{S_1 V_1^T}$ is reshaped to be $\mathbf{M_2}$. In the next stage, tSVD is applied again, now to $\mathbf{M_2}$. This procedure is repeated until each mode has been processed and as the bottom panels illustrate, the last TT-core is obtained from $\mathbf{S_4 V_4^T}$. We note that in our notation we use s_k for the kth TT-rank, this diagram refers to it as R_k.

the approximation. When the TT-ranks can be determined in advance the tSVD algorithm mentioned in Section 2.3.1 leads to faster runtimes. In most cases, however, we want to specify the error tolerance ϵ, i.e. the TT-ranks must be determined adaptively. In such scenarios, the full SVD must be computed followed by a truncation. Figure 2.3 illustrates the procedure. We refer the reader to [4] for an in depth understanding of the algorithm.

Besides the TT-SVD, there is the TT-Cross algorithm [56], which can be significantly faster, however in various scenarios it performs slower [9]. Due to its speed inconsistency, we don't consider it here in more detail. Additionally, one can replace the SVD algorithm in TT-SVD with other rank-revealing factorizations for better speed. Using randomized SVD yields the rTTSVD algorithm which is consistently faster than TT-SVD [9, 57]. Additionally, the constrained Tucker-2 decomposition is a possibility too [48, 58]. All these algorithms boil down to either computing low-rank matrix factorizations (LRMF) or other decompositions such as the Eigenvalue decomposition (EVD) on auxiliary matrices to construct the TT-decomposition [48]. These methods do not utilize sparsity patterns, leaving much value on the table when the nonzero entries in the data only account for a smaller portion of the input.

More recent approaches, aim to find a TT-decomposition when the data is sparse [9, 10]. One of these algorithms is the *matrix2mpo* method which partitions the input matrix into blocks so that each nonzero block is converted into a rank-1 TTM that are added together to form the final TTM. This can be followed by parallel-vector rounding and the TT-round procedures to reduce the TT-ranks of the resulting TTM [10]. *FastTT* [9] works by a similar principle, however it starts with a tensor as an input as opposed to a matrix. For both of these methods, their runtime bottleneck lies in the TT-round procedure which may be skipped in certain cases. Notably, even with the TT-round procedure, their runtime can be several to several ten times faster than TT-SVD for sparse inputs [9, 10].

The *matrix2mpo* algorithm has a central place in this study, therefore we illustrate its core idea in more detail next. Suppose we have a block matrix $A \in \mathbb{R}^{n \times n}$ with nonzero blocks A_{12}, A_{23} and $A_{31} \in \mathbf{R}^{I \times I}$ so that $n = I \cdot 3$. Let us denote a 3×3 zero matrix with a 1 at its ith row and jth column coordinate pair as E_{ij}. Then, using the Kronecker product, we can write A as shown in Eq. (2.1).

$$A = \begin{pmatrix} 0 & A_{12} & 0 \\ 0 & 0 & A_{23} \\ A_{31} & 0 & 0 \end{pmatrix} = E_{12} \otimes A_{12} + E_{23} \otimes A_{23} + E_{31} \otimes A_{31} \tag{2.1}$$

The *matrix2mpo* uses this principle to construct a TTM from the input matrix. Each summand corresponds to a rank-1 TTM with two TT-cores: A_{ij} and E_{ij}. After identifying the nonzero blocks, the algorithm determines the correct nonzero coordinates in the E_{ij} matrices. Finally, we add the rank-1 TTMs together, which is achieved by essentially merging the corresponding TT-cores [4]. In this case the final TTM has two TT-cores, the first one is formed by stacking A_{12}, A_{23} and A_{31}, while the second one is formed by merging E_{12}, E_{23} and E_{31}.

2.6. TT-solve methods

Let us consider a linear system of equations (LSEs) $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^n$. The linear system can be converted to the TT format by computing the TTM of $\mathbf{A}$ and the TT of $\mathbf{b}$. Afterwards, different TT-solve algorithms may be utilized to find $\mathbf{x}$ as a TT, which can be easily converted back to the solution vector of the LSE.

2.6.1. Quadratic form optimization

To solve the LSE in the TT-format, we can reformulate $\mathbf{Ax} = \mathbf{b}$ as a minimization problem. Suppose $\mathbf{A}$ is positive definite. Then the vector-valued function $f(\mathbf{x}) = \frac{1}{2}\langle \mathbf{x}, \mathbf{Ax}\rangle - \langle \mathbf{x}, \mathbf{b}\rangle$ has a global minimum at $\mathbf{x_m}$ that satisfies $\mathbf{Ax_m} = \mathbf{b}$ [59]. Using this optimization formulation, the question remains how we can find the minimum of this objective function when $\mathbf{A}, \mathbf{b}$ and $\mathbf{x}$ are in the TT-format.

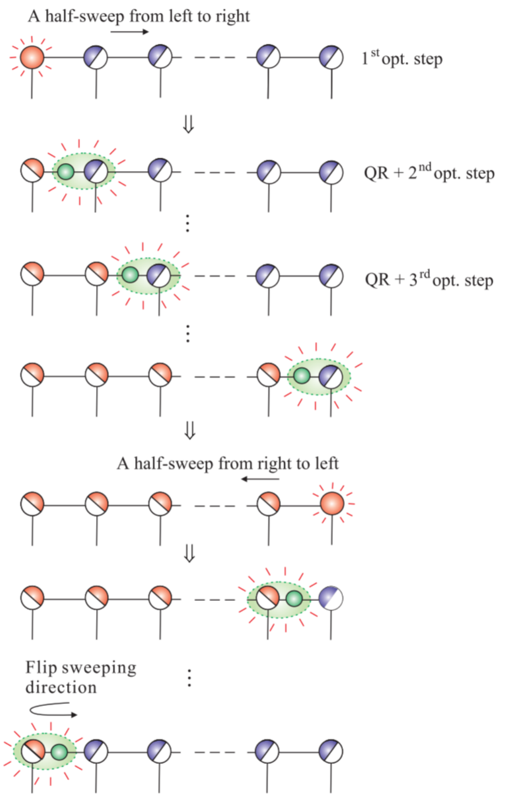
Consider the task of minimizing the function, $\mathbf{J}(\underline{\mathbf{X}})$, which depends on an N-th order tensor variable represented in the TT-format. The goal is to find a solution in the form of a tensor train. However, minimizing over all tensor cores $X^{(n)}$ simultaneously is typically too complex and nonlinear. To make the problem feasible, this process is replaced by a sequence of optimizations, each focusing on one core at a time [60].

The so called **Alternating Least Squares** (**ALS**) method (also known as Alternating Linear Scheme or one-site Density Matrix Renormalization Group (DMRG) / DMRG1) operates by updating one core tensor $\underline{\mathbf{X}}^{(n)}$ at a time during each local optimization (micro-iteration step), while keeping all other cores fixed. Starting with an initial guess for all cores, the method first updates core $\underline{\mathbf{X}}^{(1)}$ while keeping cores $\underline{\mathbf{X}}^{(2)}, \ldots, \underline{\mathbf{X}}^{(N)}$ fixed. Next, it updates $\underline{\mathbf{X}}^{(2)}$ while keeping $\underline{\mathbf{X}}^{(1)}, \underline{\mathbf{X}}^{(3)}, \ldots, \underline{\mathbf{X}}^{(N)}$ fixed, and so on, until $\underline{\mathbf{X}}^{(N)}$ is optimized. After this forward pass (half-sweep), the algorithm proceeds backward, updating cores from N to 1 in the backward half-sweep. A complete sequence of forward and backward passes constitutes one full sweep (global iteration). These iterations continue until a stopping criterion is met [6, 60].

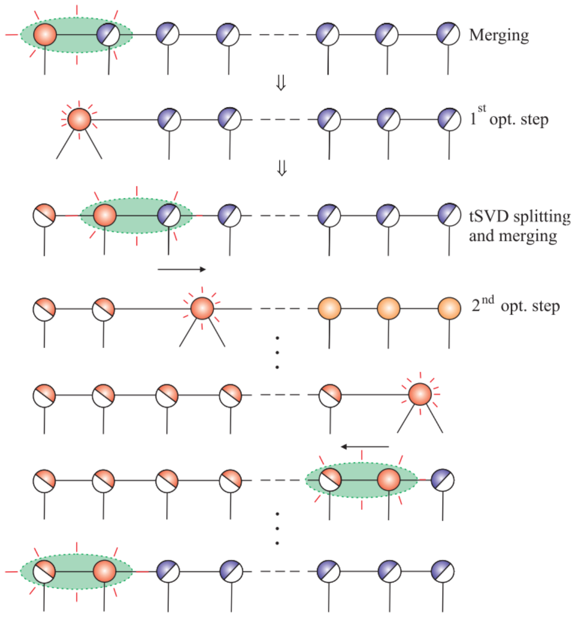
A challenge arising in this method is that the desired TT-rank is unknown, so standard ALS must rely on an initial guess for the TT-rank. This can lead to a very slow iteration process depending on the initial conditions. To mitigate these problems, the **modified ALS** (**MALS**, two-site DMRG / DMRG2) scheme merges two neighboring TT-cores (blocks) into a single ”super-node” (also referred to as a ”super-core” or ”super-block”). This combined entity is optimized, and then it is split back into separate factors using low-rank matrix factorizations, usually through computing SVD. This allows for determining the TT-ranks adaptively [6, 60].

For both ALS and MALS, it has been shown that, the cost function will only decrease with each iteration [6]. However, for a general $\mathbf{J}(\underline{\mathbf{X}})$, there is no guarantee of reaching a global minimum [6, 60]. In fact, a global minimum might not even exists [52, 61, 62]. Thankfully, practical experience supports the finding that getting stuck in local minima is a rare occurrence [6]. Also, there has been some work on showing the local and global convergence of these methods, especially when $\mathbf{J}(\underline{\mathbf{X}})$ is a convex functional. However, the conditions to be satisfied are often highly technical and not straightforward to verify in practice [61, 63, 64].

The MALS algorithm is further extended by a number of tricks in [5] to make it into a black-box solver. Its implementation can be found in [65]. It is worth mentioning that this extended version of MALS is



(a) A full sweep of the ALS algorithm on a TT. The key idea is to optimize one core tensor at a time by minimizing an appropriate cost function, while keeping the other cores fixed. After optimizing each core tensor, an orthogonalization step is performed using QR or LQ decompositions. The resulting factor matrices R are then absorbed into the next core tensor. The small green circle represents a triangular matrix R or L, which is combined with a neighboring TT core (green shaded ellipsoid). Source: [60]



(b) A full sweep of the MALS algorithm on a TT. This involves merging two neighboring cores during each optimization step to form a "super-core," which is then optimized. Afterward, tSVD or other low-rank matrix factorizations are used to divide the optimized super-core back into two separate cores. Although each optimization sub-problem is more computationally demanding than in the standard ALS, the MALS algorithm can considerably accelerate convergence and adaptively estimate TT ranks throughout the iteration process. Source: [60]

Figure 2.4: Graphical representation of the ALS and MALS sweeps.

often referred to as the DMRG algorithm in the TT literature [5, 7, 8].

Another method called the **Alternating Minimum Energy** (**AMEn**) operates on single TT-cores at a time (like ALS), making its time complexity per sweep comparable to the ALS approach. Simultaneously, it is capable of choosing TT-ranks adaptively for a specified error tolerance ϵ like the MALS method. This is achieved by using inexact gradient directions and the theory of steepest descent algorithms. For the details, we refer the reader to [8, 65].

2.6.2. Pseudo-inverse optimization

Another formulation involves solving the LSE $\mathbf{Ax} = \mathbf{b}$ in the TT format by finding the Moore-Penrose pseudo-inverse matrix $\mathbf{P}^T$ of $\mathbf{A}$. Once obtained, the LSE can be solved by computing $\mathbf{x} = \mathbf{P}^T\mathbf{b}$. In order to find $\mathbf{P}$, the following optimization problem should be solved (in the TT format): $\min_{\mathbf{P}} \|\mathbf{I} - \mathbf{P}^T\mathbf{A}\|_F^2$. Rewritten as a cost function, we need to minimize $F(\mathbf{P}) =$ trace$(\mathbf{P}^T\mathbf{A}\mathbf{A}^T\mathbf{P} - \mathbf{2}\mathbf{P}^T\mathbf{A})$. After different techniques to rewrite this expression in the TT-format to make it tractable, we can use MALS, for instance, to find $\mathbf{P}$ and then in turn $\mathbf{x}$. See [66, 67] for more details on this methodology.

2.6.3. TT-GMRES

A completely different method to solve the $\mathbf{Ax} = \mathbf{b}$ LSE is by following the GMRES algorithm (see section 2.2.2) but in the TT-format. In particular, TT-GMRES operates by approximating the solution vector and all intermediate Krylov subspace vectors in the TT-format, thereby reducing both storage and computational complexity when such low-rank approximation is possible. This approach involves performing the TT-rounding operation and TT-matrix-vector products within the iterative GMRES framework. By maintaining these vectors in a compressed TT-form, it can achieve a significant reduction in the dimensionality and size of the data being processed, facilitating the efficient handling of high-dimensional problems [68].

2.7. Low-rank approximable matrix types

An important question to tackle is which matrices can be approximated well as a TTM. I.e. which matrices have a low TTM-rank, while accurately representing the input, up to a small error (e.g. $\epsilon = 1e^{-7}$). Intuitively, one might turn to matrix ranks for guidance. Lower (numerical) ranks often yield smaller TTM-ranks, as in the case of the Hilbert matrix in Table 2.1. However, TTM-ranks can be much smaller than the (numerical) rank as we can observe for the other matrices in Table 2.1. Also, as we can see in the case of the Vandermonde matrix, TTM-ranks may be higher than the (numerical) rank of the matrix. These observations are in line with our reasoning around Figure 1.2, where we have illustrated how flattening higher dimensional data can lead to pattern breaking. That is, tensor decompositions can find a low-rank approximation, while matrix-based approaches cannot.

To the best of our knowledge, there is no methodology to assess whether a general (sparse) matrix has a small TTM-rank without actually computing its TT-decomposition. However, advantageously, TT-decomposition techniques like the *matrix2mpo* algorithm [10] can be computed very fast to assess the suitability of TT-methods for a given sparse LSE. In the case of dense matrices, randomized-SVD based techniques can help, however, their runtime still highly exceeds the *matrix2mpo* approach [10].

Matrix name	n	Rank	Numerical rank		TTM-rank		
			$\epsilon = 1e^{-3}$	$\epsilon = 1e^{-10}$	$\epsilon = 0$	$\epsilon = 1e^{-3}$	$\epsilon = 1e^{-10}$
ex3	1821	1821	1821	1821	9	7	7
ex10	2410	2410	2404	2410	100	28	28
ex10hs	2548	2548	2537	2548	784	169	169
ex13	2568	1434	2193	2568	576	80	82
Hilbert	2000	25	9	22	625	4	9
Vandermonde		45	20	43	625	119	251
Hankel		2000	2000	2000	625	3	3
Toeplitz		2000	2000	2000	625	3	3

Table 2.1: This table shows the size (n) and the different ranks of a number of matrices. The first four matrices come from [69] and are described in more detail in Section 3.5.1. The other four matrices are special matrices that are often used as examples in this context [1, 2, 60]. We use *numpy* [70] and *scipy* [71] to generate them. For the Hankel and Toeplitz matrices, we use an input vector of $[1, 2, \ldots, n]$. On the other hand, for the Vandermonde matrix we use a random vector with values between 0 and 1 to avoid numerical issues, resulting from too large scalars. By rank we mean the rank of a matrix in the linear algebra sense. The rank values shown are taken from [69]. To compute the numerical ranks, we use the *linalg.matrix_rank* method from *numpy* [70] which is based on SVD-truncation. In order to obtain the TTM-ranks, TT-SVD is run on the matrix, after it is reshaped to a tensor with mode sizes as the prime factors of its size, n. For this part, we have utilized the *Scikit-TT* library [72].

3 Approach

In this chapter, first, we elaborate on our methodology that we use to compare TT-solve against the Conjugate Gradient (CG) and Gaussian elimination (GE) solvers. Afterwards, we summarize the key insights of this comparison. Our inquiry leads us to identify a handful of research gaps regarding the sparse TTM-decomposition process. We propose a number of strategies to address these. Finally, we design an experimental configuration to examine the suggested techniques.

3.1. Methodology

Our primary objective is to assess the competitiveness of the TT-solve approach against traditional methods such as CG and GE. In particular, by TT-solve we mean the TT-MALS approach. We use this as opposed to TT-ALS because it has control over the TT-ranks, i.e. the approximation error while solving the system. As we are interested in (near) exact solutions of LSEs, this is a must have in our setting. The AMEn and TT-GMRES solvers reviewed in Section 2.6 could have been selected as well. However, these methods are significantly more complex and require a deeper understanding for time complexity assessments. In our initial experiments using implementations from the TT-toolbox [65], we found that these solvers were either slower or failed to converge when TT-MALS succeeded. Furthermore, the existing literature on AMEn and TT-GMRES is much more limited, suggesting that pursuing these directions may not be as fruitful.

CG and GE have been extensively optimized over the years, benefiting from highly specialized code for specific hardware setups. This could result in an uneven comparison when introducing a newer algorithm like TT-solve. To address this disparity, we propose a theoretical assessment methodology designed to be performed quickly, facilitating rapid evaluations and comparisons. By not assuming a particular hardware setup, the results remain broadly applicable across different computing environments. Additionally, the methodology minimizes the impact of code optimizations available to more mature approaches, ensuring a fair comparison based on algorithmic principles rather than implementation details. Finally, it is crucial to demonstrate how the TT-solve algorithm scales with the size of the input, providing insights into its performance for varying problem sizes.

To achieve these goals, our theoretical time assessment incorporates several simplifications. While not capturing every nuance of practical execution, it still offers a robust estimate of expected runtime. These simplifications allow us to focus on the core computational complexities and scaling behavior of the TT-solve algorithm.

When possible, we rely on existing literature for theoretical runtimes of the algorithms. In certain cases, the available research is insufficient for our analysis, prompting our own assessment. The detailed time complexity derivations are given in Appendix B. Next we elaborate on our insights derived from this investigation.

3.2. Insights from TT-solve time complexity analysis

To solve a linear system of equations (LSE) in the TT-format, we need to perform three blocks of computations:

1. Convert LSE to TT-format
2. Find solution in TT-format
3. Convert the solution back to vectorized form

As we will see, besides the matrix size n, the number of TT-cores d, the maximal mode size I, the TTM-rank r and the TT-rank s of the solution vector and right-hand side vector are all needed to describe the time complexity of the TT-solve approach. Therefore, let us first examine these individually and the relations between them.

3.2.1. TTM-rank and TT-rank

As we have seen in Table 2.1, the TTM-rank of a (sparse) matrix is different from its rank and numerical rank. While lower (numerical) rank often leads lower TTM-ranks, they are not strictly linked. Depending on the method used to find the TT-decomposition of a matrix, we get different TTM-ranks. For example, using TT-SVD is guaranteed to yield the best possible TTM-ranks for a chosen error tolerance [4]. On the other hand, when dealing with sparse matrices, a much faster method, the *matrix2mpo* algorithm can be utilised [10]. This approach, does not come with the same guarantee as TT-SVD. However, the resulting TTM-rank corresponds to the number of submatrices needed to cover all nonzero entries in the matrix as we illustrated in Eq. (2.1). We also refer to this submatrix dimension as tile size. The size of these submatrices must be constant, and their multiple must add up to the matrix size n. Importantly, this tile size corresponds to the mode size of the first TT-core, I_1. Therefore we have that $I_1 \cdot k = n$, $k \in \mathbb{N}$.

The TT-rank, s, in our context refers to the rank of the TT-decomposition of the solution vector and the vector on the right-hand side of the LSE. To solve the LSE via TT-solve, besides converting the matrix, we also need to compute the right-hand side vector to the TT-format. As this vector is rarely sparse, using a TT-SVD type method is a reasonable choice for this task.

3.2.2. Mode sizes

When decomposing the matrix, besides the TTM-rank, we also need to keep an eye on the size of the modes, where the maximum mode size is I. This is important, as this parameter also frequently appears in the time complexity bounds. Thus, we aim to have I as small as possible. Due to the structure of TT-decompositions, we know that $I_1 \cdot I_2 \cdots I_d = n$, thus $I \geq \sqrt[d]{n} = n^{1/d}$. This means that the bigger d is, the smaller I can be. A practical lower bound is $I \geq 2$ because $I \in \mathbb{N}$ and having smaller sized modes is pointless.

When aiming to compute the TT-decomposition of a LSE, we need to reshape our matrix and right-hand side vector as tensors in order to use TT-SVD. To achieve this, we need to find positive integers $I_1, I_2, \ldots, I_d$ s.t. $I_1 \cdot I_2 \cdots I_d = n$. Then, we can reshape a matrix of size $n \times n$ to a tensor of size $I_1 \times I_1 \times I_2 \times I_2 \times \cdots \times I_d \times I_d$. Similarly, a vector of size $n \times 1$ can be reshaped as a tensor of size $I_1 \times I_2 \times \cdots \times I_d$. In the case of *matrix2mpo*, having a matrix input is assumed. Nevertheless, we have to find $I_1, \ldots I_d$ first in this setting too. This is necessary, as the algorithm takes these as input besides the matrix. Additionally, a choice has to be made regarding which integer factor will be I_1. This determines the tile size used to capture the nonzero entries.

On top of these observations, it is important to realize that there is a close relationship between the mode sizes and TT(M)-ranks. For example, as stated in Theorem 3.2 of [10], the product of mode sizes can be used to bound the TT(M)-ranks. Moreover, when working with the *matrix2mpo* approach, the mode size of the first TT-core determines the submatrix size used to capture nonzero entries. Since the number of tiles needed to capture all nonzero entries will be the TTM-rank, the link between the two is apparent.

The time complexity of TT-operations are determined by the largest mode size I. Thus keeping this small is desirable. On the other hand, choosing I_1 to be one of the larger factors means a (generally) smaller TTM-rank, as it is much easier to capture all nonzeros when the used submatrices are bigger.

To this end, our suspicion is that the authors in [10], chose to use the largest prime factor of n as I_1, as the largest mode size cannot be smaller. Additionally, choosing a smaller tile size (i.e. a smaller factor as I_1) would most likely just increase the TTM-rank.

Method	Use	Time complexity
TT-SVD	Convert tensor to TT format	$\mathcal{O}(s^3 In)$
TTM-SVD	Convert tensor to TTM format	$\mathcal{O}(r^3 I^2 n^2)$
modeSize2Rank	Find achievable TTM-rank of sparse matrix via matrix2mpo	$\mathcal{O}(z)$
matrix2mpo	Convert sparse matrix to TTM format	$\mathcal{O}(z + dI^2 r^2)$
TT-MALS (direct solver)	Solve LSE in TT(M)-format	$\mathcal{O}\big(cd(s^3 r^2 I^2 + s^3 r I^3 + s^6 I^6)\big)$
TT-MALS (iterative solver)	Solve LSE in TT(M)-format	$\mathcal{O}(cds^3 r^2 I^3)$
TT-sol2vector	Convert solution in TT-format to a vector	$\mathcal{O}(dns^4)$
CG	Solve SPD sparse LSE	$\mathcal{O}(z\sqrt{\kappa})$

Table 3.1: This Table summarizes the time complexity of the TT-algorithms relevant for solving LSEs in the TT-format. The derivations can be found in Appendix B.1. The variables meaning is as follows. r is the TTM-rank, s is the TT-rank of the right-hand side and the solution vector. I is the maximum mode size of the TTM. d is the number of TT-cores and is known to equal $\log(n)$ approximately. κ is the condition number of the matrix, z is the number of nonzero entries and finally c is the number of half-sweeps TT-MALS needs to converge. SPD stands for symmetric, positive definite.

3.2.3. The number of TT-cores

Taking the above reasoning further gives the upper bound $d \leq \log_2(n)$ which can be derived as follows. We use that $n \geq 2$ and $d \geq 1$.

$$I_1 \cdot I_2 \cdots I_d = n \qquad \text{(relation between } I \text{ and } n \text{ in TT format)} \tag{3.1}$$
$$2^d = n \qquad \text{(choose smallest possible value } \forall I\text{)} \tag{3.2}$$
$$\log_2(2^d) = \log_2(n) \qquad \text{(take logarithm of both sides)} \tag{3.3}$$
$$d = \log_2(n) \qquad \text{(use logarithm rules)} \tag{3.4}$$

To find a lower bound for d, we solve the following inequality:

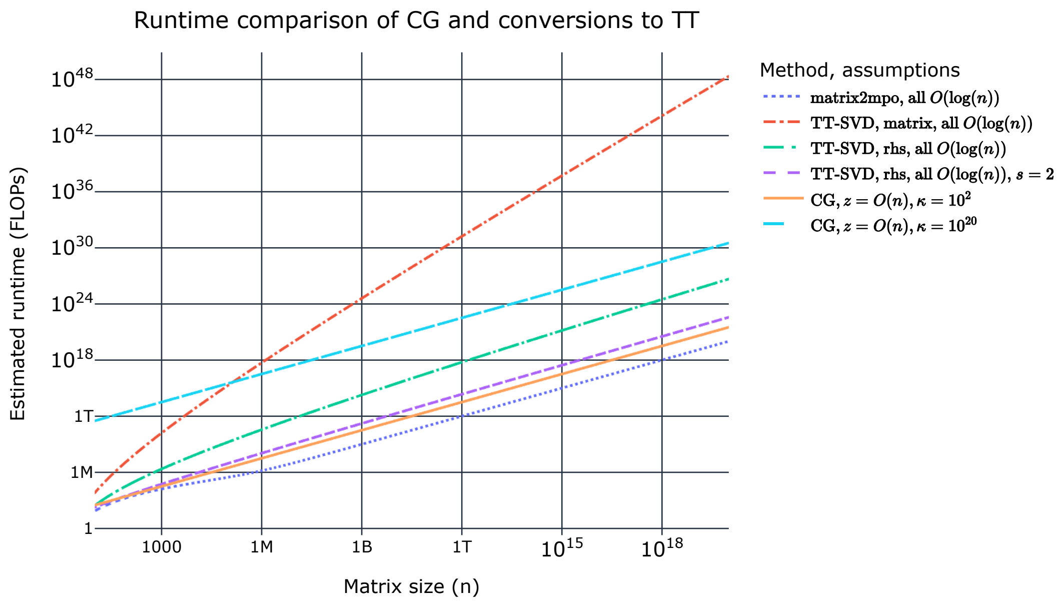
$$I \geq n^{1/d} \qquad \text{(relation between } I, d \text{ and } n \text{ in TT format)} \tag{3.5}$$
$$\log(I) \geq \frac{1}{d}\log(n) \qquad \text{(take logarithm of both sides)} \tag{3.6}$$
$$d \geq \frac{\log(n)}{\log(I)} \qquad \text{(reorder terms)} \tag{3.7}$$
$$d \geq \log_I(n) \qquad \text{(use logarithm rule)} \tag{3.8}$$

We conclude that $d = \Theta(\log(n))$. Therefore, we cannot influence the TT-decomposition by changing the parameter d when starting with a fixed (e.g. 2) dimensional problem. Finally, we note that the TT-decomposition with fixed mode sizes $I_1 = I_2 = \cdots = I_d = 2$ is termed the Quantized Tensor Train (QTT) format [5, 73].

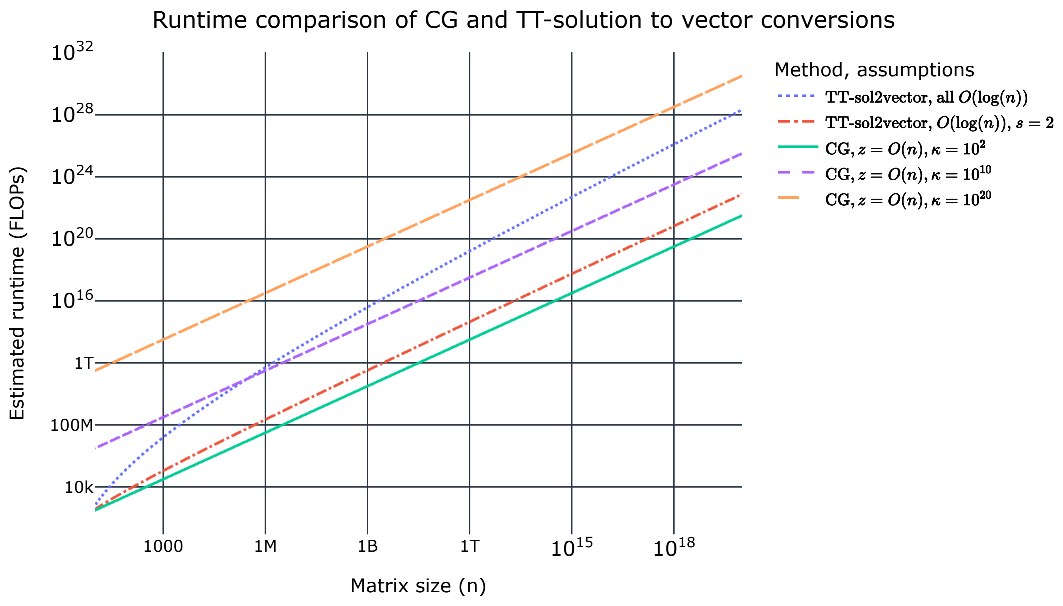
3.2.4. Runtime comparison of TT-solve and CG

In this section, we look at how the theoretical time complexities of the CG solve and TT-approach compare. The time complexity derivations of the TT-algorithms relevant for solving LSEs are in Appendix B.1. We summarize the relevant algorithms' time complexity in Table 3.1.

In order to be able to compare the runtime of CG with the TT-solve approach, we need to make a number of assumptions. Firstly, let us suppose that $z = \mathcal{O}(n)$ since we work with sparse matrices. As we have seen in Section 3.2.3, we can safely write that $d = \mathcal{O}(\log(n))$. CG's runtime is linear in the number of nonzero entries z and thus the matrix size n. If any of the TT-parameters like r, s, I become $\mathcal{O}(n)$, CG will be faster as these terms appear with higher exponents than 1. Thus, we can conclude

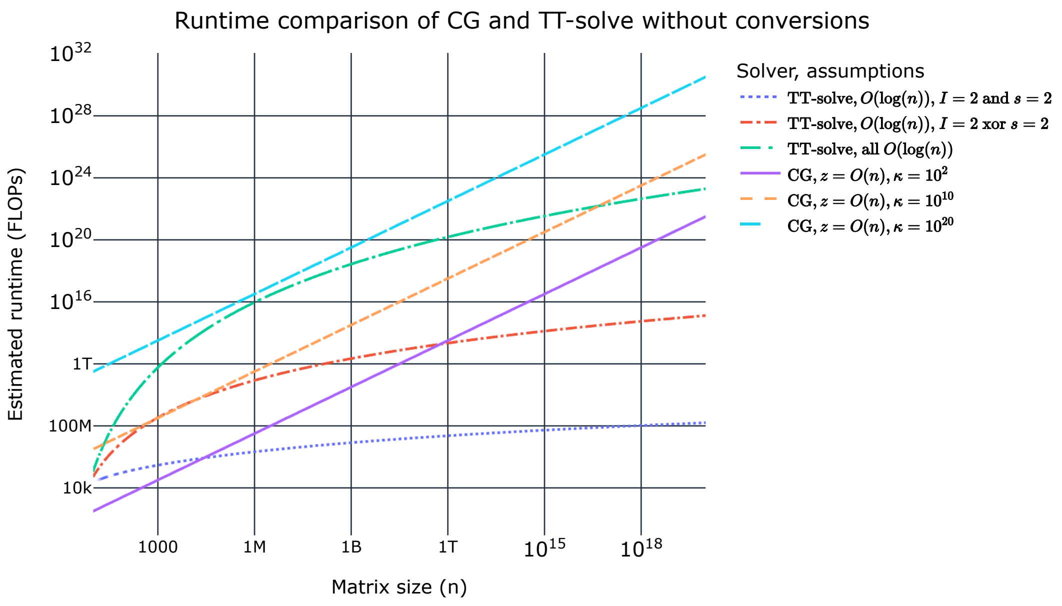


(a) This Figure shows a comparison between the CG solver runtime and the time it takes to convert the LSE to TT-format. In particular, we can observe the big difference between the *matrix2mpo* approach and TT-SVD for converting the coefficient matrix. Additionally, we can see how the TT-SVD runtime for converting the vector on the right-hand side (rhs) scales.

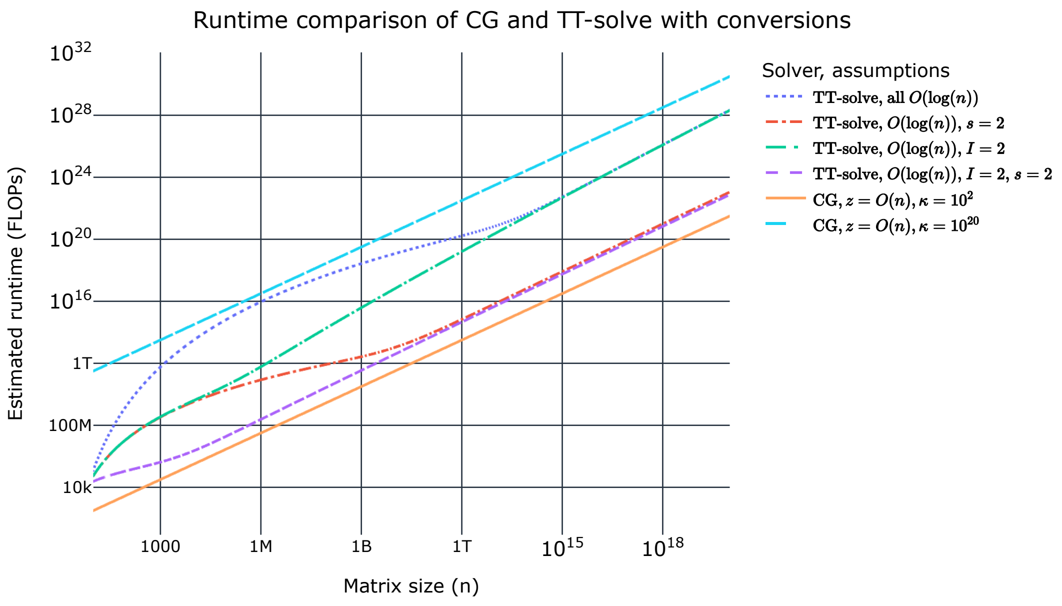


(b) This Figure compares the runtime of the CG solver with the method (*TT-sol2vector*) used to convert the TT-solution back to a vector.

Figure 3.1: This Figure illustrates the theoretical time complexity of converting a sparse LSE to the TT-format and then converting the solution back to a vector. These runtime complexities are contrasted with the runtime of the CG solver, assuming a sparse matrix ($z = \mathcal{O}(n)$) with different condition numbers κ. The unit of runtime is a floating point operation (FLOP), which is plotted as a function of the matrix size n. For all TT-parameters such as d, r, s, I we assume that they are $\mathcal{O}(\log(n))$, unless specified otherwise.



(a) This Figure illustrates how the TT-solve approach compares to the time complexity of the CG solver. For this plot, the TT-solve runtime does not include the time needed for converting to and from the TT-format. I.e., here it is assumed that the LSE is given in the TT-format from the start and we desire the solution in this format as well.



(b) This Figure illustrates how the TT-solve approach compares to the time complexity of the CG solver. For this plot, the TT-solve runtime includes the time needed for converting to and from the TT-format. In particular, we assume the use of *matrix2mpo* for converting the coefficient matrix, TT-SVD for converting the right-hand side (rhs) and the *TT-sol2vec* procedure to convert the solution back to a vector.

Figure 3.2: This Figure illustrates the theoretical time complexity of solving a LSE via CG and in the TT-format. We show the TT-solve time complexities both with and without conversions to and from the TT-format. The use of TT-MALS is assumed with a direct solver for local LSEs. Additionally, the input matrix is presumed to be sparse, i.e $z = \mathcal{O}(n)$ and we show runtimes for different condition numbers κ. The unit of runtime is a floating point operation (FLOP), which is plotted as a function of the matrix size n. For all TT-parameters such as d, r, s, I and the number of half-sweeps needed for convergence, we assume that they are all $\mathcal{O}(\log(n))$, unless specified otherwise.

that these parameters should be of lower order than $\mathcal{O}(n)$. In particular, smaller than $\mathcal{O}(\sqrt[6]{n})$, as the largest factors have exponents of 6. As there are many terms to consider in the TT-approach, we go with the assumption that r, s, I, c are at most $\mathcal{O}(\log(n))$, to be on the safe side. With these assumptions, now we are ready to measure the TT-approach runtimes as a function of n.

Let us first look at the time complexity of converting the LSE to the TT-format and then the solution back to a vector, in Figure 3.1. We can notice in Figure 3.1a, how using the classical TT-SVD approach for matrix inputs is impractical in practice due to its runtime. On the other hand, using it for converting the right-hand side vector is more feasible, especially when its TT-ranks can be kept really small. This plot also shows the superior runtime of the *matrix2mpo* algorithm used to convert a sparse coefficient matrix into the TTM-format. From the plot it is evident that these assumptions on keeping TT-parameters $\mathcal{O}(\log(n))$ or smaller is not enough. We can only hope to achieve any speedups, if the matrix is ill-conditioned, rendering the CG runtime higher than usual. Similar conclusions hold for Figure 3.1b, where we illustrate the runtime of converting the solution in the TT-format to a vector. In addition, we can see how the TT-rank (s) of this vector can be a bottleneck when it is too large.

Next, in Figure 3.2, we reflect on the runtime of TT-solve with and without the conversions showcased in Figure 3.1. Again, we contrast this with the runtime of CG, assuming differently conditioned matrices. Firstly, Figure 3.2a displays the runtime of TT-MALS with direct solvers for the local problems. As also suggested by Table 3.1, the bottleneck variables are s and I which is evident in the plot as well.

Figure 3.2b combines the TT-conversions with TT-MALS. In particular, we use the *matrix2mpo* for converting the matrix into TTM-format, while TT-SVD is utilized for transforming the right-hand side. Additionally, after solving the LSE via TT-MALS, we also add the runtime of transferring the solution back to a vector. We can see that keeping the TT-rank s constant is more important than the mode size I as this is a bottleneck both for the TT-MALS and the TT-sol2vector procedures. However, keeping either of these as a constant, while assuring that the others are $\mathcal{O}(\log(n))$, gives the TT-solve approach a considerable edge over CG for ill-conditioned matrices.

3.3. Research gaps

In this section, we look at some challenges that one faces when attempting to decompose a LSE to the TT-format.

3.3.1. Achievable TT-decompositions

Matrices come in various sizes. Depending on this size, n, the prime factorization largely determines the possible TT-decompositions that are possible, both via TT-SVD and *matrix2mpo*. In the worst case, the matrix size n is a prime number, making its (meaningful) reshaping to a tensor impossible. In such a situation, the TT-decomposition has just one TT-core that is the original matrix, with TTM-ranks of 1 on each end. Notably, the matrix size does not need to be prime to make the TT-decomposition problematic. For instance, if the largest prime factor is relatively large, then, this will lead to a bottleneck when attempting to solve the LSE in the TT-format. This is because of the I^6 term in the time complexity of TT-MALS, as seen in Table 3.1. To the best of our knowledge, this challenge has been left unaddressed in existing literature.

3.3.2. Tradeoff between TTM-rank and mode size for matrix2mpo

In this section, we explore another untapped possibility to improve the runtime of TT-solve. We do this by utilizing the inherent link between the maximum mode size I and the TTM-rank r which we have seen in Section 3.2.2.

Given a sparse $n \times n$ matrix $\mathbf{A}$ with z nonzero entries, we know the following of its TT decomposition when the $matrix2mpo$ algorithm is used. The dimensions of $\underline{\mathbf{A}}^{(1)}$ are $r_1 \times I_1 \times I_1 \times r_2$, where $r_1 = 1$. I_1 defines the size of the submatrices which are used to partition $\mathbf{A}$. r_2 equals the number of submatrices in the partitioning that contain a nonzero entry. Therefore, it is crucial to create a partitioning with minimal number of partitions that capture all nonzero entries. The trivial partition with minimal rank $r = 1$ is achieved when we choose $I_1 = n$. Choosing I to be small is also important while minimizing r. Therefore, a tradeoff has to be made when using this decomposition approach. Finding an optimal pair of r and I depends on various factors that we uncover in this section.

We assume that a TT-MALS type solver will be used. From a time complexity point of view, there are two cases one needs to consider separately. The TT-solve method can use either a direct or an iterative solver to find solutions to the local optimization problems. As pointed out earlier, this is the computational bottleneck of the whole process. Also, depending on which local solver is used, the time complexity depends on I and r differently. For this analysis, we look at the direct solver case. We note that an analogous assessment can be made when iterative solvers are applied. However, due to the unpredictability of iterative solver convergence, such an analysis is less robust.

As noted in Appendix B.1.3, the direct solver complexity does not depend on the rank of TTM, only the TT-ranks of the solution and right-hand side vectors. As the right-hand side vector is generally not sparse, its decomposition is best done via a TT-SVD type algorithm. Additionally, the ranks of the solution vector in the TT format can be directly controlled by the TT-MALS algorithm. It is important to note that the TT-MALS algorithm still has lower order dependencies on the TTM-ranks. Namely, we need $\mathcal{O}(r^2I^2)$ for micro-iteration preparations per TT-core and $\mathcal{O}(rI^3)$ for contractions in between.

While the time complexity is mainly dependent on the chosen mode sizes ($\mathcal{O}(I^6)$), we also need to be careful with the TTM-ranks. For instance, we might be able to achieve $I = 2$, i.e. I is $\mathcal{O}(1)$ but getting TT-ranks of order $\mathcal{O}(n)$, then $\mathcal{O}(r^2) = \mathcal{O}(n^2)$ will dominate the time complexity leading to infeasible computation times.

This means that minimizing I is more important than r, however, we cannot fully neglect the TTM-ranks either. This analysis motivates the development of techniques that optimize this tradeoff between I and r in the context of the matrix2mpo algorithm. Following the above analysis, we propose the following objective function to evaluate different pairs of r and I:

$$f(r, I) = I^6 + rI^3 + r^2I^2 \tag{3.9}$$

3.4. Strategies to optimize sparse TTM decompositions for TT-solve

In this section, we propose a number of strategies to improve the estimated TT-MALS runtime by addressing the research gaps that we have uncovered in Section 3.3.

3.4.1. Reducing TTM-ranks

The resulting ranks directly depend on how well the *matrix2mpo* algorithm can capture the nonzero entries. That is, the fewer tiles it needs, the better. In this part, we look at two different approaches that aim to help with this.

Choosing tile size from factors

The mode sizes of the TT-cores produced by the matrix2mpo algorithm are determined by the integer factors of the size of the given matrix (n). For instance, if we have a matrix of size 2410×2410, then since $2410 = 241 \cdot 5 \cdot 2$ is the prime factorization of n, the modes of the TTM will be $I_1 = 241$, $I_2 = 5$ and $I_3 = 2$.

When using the *matrix2mpo* algorithm, any of these modes can be chosen as the tile size for the submatrices used to capture nonzero elements. Choosing the biggest mode to size the tiles, often yields the smallest TTM-rank, because it is easier to cover all nonzero entries with bigger blocks. However, smaller submatrices can also prevail because they tend to capture the nonzero patterns better. Additionally, we can choose to combine factors to form a differently sized submatrix for nonzero capture. In the above example, we could combined $I_2 = 5$ and $I_3 = 2$ as a new factor $I_2' = I_2 \cdot I_3 = 10$. It should be noted that combining factors to increase the mode sizes directly increases the TT-solve runtime. Therefore, it is important to keep in mind the tradeoff between TT-ranks and mode size at this step.

Variable ordering

An existing method to achieve smaller TTM-ranks by the matrix2mpo algorithm is to reorder the variables in the matrix [10]. In particular, they use the Cuthill–Mckee permutation [74] to minimize the bandwidth (as defined in Appendix C.8) of the input matrix. Its runtime can be bounded by $\mathcal{O}(\log(d_{max})z)$, where d_{max} is the maximum number of nonzero entries in a column [75].

This strategy can be an effective way to reduce the TTM-rank of the matrix2mpo decomposition because reducing the bandwidth often means concentrating the nonzero entries close to each other, near the diagonal. Therefore, fewer number of submatrix blocks are needed to cover the nonzero entries. Its impact is largely dependent on the original structure of the matrix, however, it can often reduce the TTM-ranks by around 50% on various examples [10].

3.4.2. Better partitioning for mode size reduction

While the previous strategies are helpful to optimize TTM-ranks, as concluded earlier, minimizing the mode sizes is even more crucial. To this end, next, we uncover ways to reduce them too.

The maximum mode size cannot be lowered by the previous strategies. In order to allow a reduction, we need to change the factorization of n. We can do this in two ways, either by increasing n or by reducing it. The strategy, we call *padding* corresponds to the former, while doing partial Gaussian-elimination is linked to the latter. We describe these proposed strategies in the next sections.

Padding

The idea of padding is simple: add zero rows and columns after the last row and column, respectively. Importantly, if we add zero entries extending on the diagonal as well, then we get a singular matrix. This can cause problems for the solvers, therefore, we add ones on the diagonal extension. This allows us to keep the matrix invertible and to create different factorizations, thus choosing different mode sizes for the TT-decomposition. For example, changing $n = 2410 = 241 \cdot 5 \cdot 2$ to $n' = 2412$ allows us to have $2412 = 67 \cdot 3^2 \cdot 2^2$. This reduces the maximum mode size, I, greatly. On the other hand, it may slightly increase the TTM-rank due to the added nonzero entries on the diagonal extension.

It is important to keep in mind that we are zero-padding the coefficient matrix of a LSE. Therefore, to ensure that the system is still solvable, we also have to add zero padding to the right-hand side. Additionally, the solution will also contain variables that correspond to the padding. When everything works as expected, these variables become zero and the solution of the original system can be obtained by ignoring these at the end. As all padding entries are zeros, except the ones on the diagonal, we do not alter the solution of the original system.

When using this strategy, we need to tackle two important questions.

1. What is the maximal padding that we should even consider?
2. What level of padding should we choose within this range?

Adding zero padding increases n, and z. Our matrices are stored in sparse format and the matrix2mpo algorithm's time complexity does not depend on n. Also, the TT-decomposition it produces does not suffer from any drawbacks when n is increased. The story is different for z, the nonzero count changes linearly with the level of padding. Therefore adding a lot of padding will also lead to increased TTM-ranks, as additional nonzero tiles will be probably necessary to cover them in the *matrix2mpo* approach. Additionally, we need to consider that for each row-column pad, computing the integer factorization of n' and checking how "good" it is, is necessary. In the best case scenario, we can do this in $\mathcal{O}(1)$ time for each n'. To keep the time complexity of this method feasible in comparison to using a solver like CG, we prefer to have sub-linear padding configurations to check. This also bounds the maximal TTM-rank increase. Thus, we aim to have $\mathcal{O}(1)$ paddings or at most $\mathcal{O}(\log(n))$ to assess in practice.

As a side note, we point out that polynomial time algorithms are not known for integer factorization on classical (non-quantum) computers [76]. However, for $n < 2^{16} = 65536$ we can easily store and then look up the prime factorizations in $\mathcal{O}(1)$ time. In case, $2^{16} \leq n < 2^{32} = 4294967296$ a list of precomputed primes is advisable with a fast divisibility check [77].

The straightforward way of choosing the best padding within this range can be done as follows. We can check the TTM-rank produced by each possible tile size for a given padding. Possible tile sizes are all the different products we can make from the factors of n', not necessarily using all of them. After these are computed, we can check which combination of I and TTM-rank would minimize the runtime complexity of the TT-solve method in use. Additionally, we can check if the best possible mode size and TTM-rank combination makes it desirable to use the TT-solve approach or it is better to opt for an iterative solver like CG.

Partial Gaussian (PG) elimination

In order to change the integer factorization of n, we can also reduce the size of the matrix. In this case, ensuring that we leave the solution of the LSE unchanged is less trivial. In particular, we cannot just remove rows and columns of the coefficient matrix as that would clearly lead to different solutions.

Inspired by the Gauss-CG solver method [78, 79, 80] which is used at Deltares for the company's Delft3DFM product [81], we can use partial Gaussian elimination in this context too, to get a smaller subproblem. This means that we row-reduce the system for the first $k < n$ variables, yielding a subsystem of size $n - k$. Once this subsystem is solved (e.g. via CG or TT-solve), we need to apply backsubstitution for the initial k variables to get the complete solution.

A challenge with this approach is that row reduction creates fill-in, that is, additional nonzero entries in the remaining subsystem [82]. This can be mitigated to a large extent by variable permutation. Notably, finding a variable ordering that minimizes fill in is an NP-complete problem [83]. To this end, various heuristic algorithms have been developed to ensure a tractable runtime, see for instance [84, 85, 86]. One of the most popular approaches is the so called approximate minimum degree (AMD) algorithm. This estimates the number of nonzero entries in each column and reorders the variables so that the variables corresponding to the most sparse columns are eliminated first [87, 88, 89]. The runtime of the minimum degree approach $\mathcal{O}(n^2 z)$ is too expensive, however its approximate version is often used in practice with its $\mathcal{O}(nz)$ time complexity [90]. It is important to consider whether, for our application domain, this time complexity is prohibitively expensive. As concluded earlier, in Section 3.2.4, TT-solve is competitive when the condition number of the matrix, κ, is quite large. In such a situation the runtime of CG $\mathcal{O}(z\sqrt{\kappa})$ will be considerably higher than this. Nevertheless, this is just a small part of solving the LSE in the TT-format. Thus its effectiveness must be investigated for its use to be justified.

Suppose, we are given a choice of k. Then, we need to evaluate how $n - k$ factorizes and in turn the possible mode sizes. Then, similarly to the padding approach, we can choose a pair of tile size and TTM-rank that minimizes the estimated TT-solve runtime. Using this estimate, we can decide to use the CG solver or the TT-solve method depending on the computed runtime estimate.

The next important question to tackle is how we can bound the value of k, so that we do not have to assess every possible situation. Firstly, empirical evidence suggests that having $k \approx \frac{n}{2}$ tends to give the best runtime when using the Gauss-CG solve approach [91]. Unfortunately, this does not necessarily mean that the overall runtime of Gauss - TT-solve will be worse than Gauss-CG when $k > \frac{n}{2}$. Using a more careful analysis of the Gauss-CG solver time complexity, one could bound k. However, we leave this for future investigations as it is not a crucial detail for our research.

Thankfully, choosing the best k is more straightforward. This is made possible by accessing the intermediate steps of the AMD algorithm. These intermediate steps contain the (estimated) nonzero pattern of the partially row-reduced system, therefore allowing the assessment of different tile size choices' impact on TTM-ranks.

3.4.3. Coupled approaches

Besides assessing the proposed strategies in isolation, we also want to see their combined effects. Here, we discuss sensible ways to join the methods, including their ordering.

Firstly, AMD should come before partial Gauss (PG) to reduce fill-in during the elimination process. Choosing the best tile size should come last. If the added padding is what PG reduces, then, clearly it is better to start with PG. The other scenario we do not explore here, but it may be interesting to try. Therefore, for our experiments, we put padding after AMD and PG. Finally, RCM can be reasonably placed in different places. In one interpretation, we can put it after padding and before choosing the best tile size, in order to concentrate nonzero entries before that last step. In our experiments, we follow this idea, however, it can be also worth to explore placing it before padding. The thought behind this recipe is to first condense the nonzero entries, then find what tile size choice would be the best for the result. Hence, adding padding after RCM to allow for better factorizations and thus better tile size choices at the end.

In this work, we aim to get a better understanding of the effectiveness of these strategies in isolation and together. Therefore, we will look at all possible combinations, while keeping the above described pecking order.

3.5. Experiment design

In this section, we outline our experimental setup to assess the previously presented techniques. First, we describe our choices regarding test data, followed by the evaluation metrics and our baseline that we compare against. After outlining our methodology to evaluate the effectiveness of the proposed techniques, we give details regarding the implementation.

3.5.1. Test data

In our experiments, we focus on sparse matrices that are openly available at [69]. Additionally, we only experiment with symmetric and positive definite matrices that come from computational fluid dynamics (CFD) problems. In particular, we focused on problems that are related to the flow of water or at least liquids. This decision was made because the original motivation for the project comes from the hydrodynamics domain. Linked to this reason, we found that the matrices of interest at Deltares are symmetric and diagonally dominant [91], thus symmetric positive definite (SPD) (see Appendix C). While TT-MALS often converges for non-SPD matrices as well, mathematically it is unclear why it should happen [5]. Additionally, to keep experiment runtimes feasible on a personal computer, we decided to focus on smaller matrices with $n < 15000$ and $z < 800000$. The majority of the matrices satisfying these requirements were initially created using the *Fluid Dynamics Analysis Program* (FIDAP) [92]. We have excluded one matrix (FIDAP/ex5) which is an outlier in terms of its sparsity $1 - \frac{z}{n^2} = 0.62$ (as defined in Appendix C.10).

This procedure has left us with 7 matrices. Five of these, are from the FIDAP group. These are the *ex3, ex10, ex10hs, ex13* and *ex15* matrices. These 5 have a banded structure. The other two, *Pres_Poisson* and *bcsstk13* are not banded. For the experiments that involve Gaussian elimination over many variables, the two largest matrices (*ex15, Pres_Poisson*) have been excluded due to infeasible runtimes. Additionally, we note that all test matrices are relatively ill-conditioned with $1e{+}10 < \kappa < 1e{+}16$, except for *Pres_Poisson* that has $\kappa = 2.04e + 6$.

3.5.2. Evaluation metrics

In order to assess our proposed strategies, we utilize a few different metrics. Our main objective is to minimize the theoretical TT-solve runtime. That is, the time it takes to convert the LSE to TT-format, find the solution via TT-MALS and then transform the solution back to a vector.

Based on our discussion in Section 3.3, Eq. (3.9) is helpful to assess the key tradeoff between r and I. This captures well the compromise to be made between the two variables in the context of the TT-MALS bottleneck. Additionally, its formula is simple, keeping interpretation easier. Therefore, we use it throughout Section 4.1 to look at the effects of our techniques. However, when considering the overall TT-solve time complexity - including the conversion times - means that slightly different pairs of r and I are optimal. To this end, when such runtimes are in focus, such as in Figure 4.9 and Section 4.2, we use the complete runtime complexity as the objective function. Based on Table 3.1, we can write this as follows.

$$\begin{aligned}
g(c,r,s,I,d,n,z) &= s^3In && \text{convert right-hand side with TT-SVD} \quad (3.10)\\
&+ z + dI^2r^2 && \text{convert matrix using matrix2mpo} \quad (3.11)\\
&+ cd(s^3r^2I^2 + s^3rI^3 + s^6I^6) && \text{TT-MALS (direct local solver)} \quad (3.12)\\
&+ dns^4 && \text{TT-sol2vector} \quad (3.13)
\end{aligned}$$

Let us also propose another metric for more insight. However, first, we shall illustrate where it comes from. In the context of the *matrix2mpo* algorithm, for a chosen tile size I_1, all submatrices that have a nonzero entry are selected. For clarity, recall from Eq. (2.1) that these submatrices give a partitioning of the (block) matrix. Stacking these blocks to create a mode-3 tensor gives the first TTM-core of the decomposition. The fewer submatrices are needed to capture all nonzero entries, the lower TTM-rank r we get for a chosen I_1. More precisely, r, the TTM-rank is exactly the number of nonzero blocks in the partitioning. I.e. all the blocks that contain at least one nonzero entry. Assuming all entries of interest are nonzero, a perfect partitioning means that for a chosen tile size, each block contains no zero entries. In order to evaluate how close we are to this ideal situation we propose the following metric. We divide the number of nonzero entries in the matrix by the number of entries (both zero and nonzero) the nonzero blocks capture in the partitioning. We note that this number of nonzero entries may be different from the count in the original matrix as some of our methods alter it. Therefore, we denote this potentially modified count as z'. Now we are ready to give the formula for this criterion.

$$h(r, I_1, z') = \frac{z'}{rI_1^2} \tag{3.14}$$

3.5.3. A baseline

For what comes next, it is important to establish a baseline, we can compare our approaches against. In the *matrix2mpo* algorithm paper, the authors utilize the largest prime factor from the matrix size's integer factorization as the tile length [10]. Therefore, we will regard this choice as our baseline.

3.5.4. Assessing the effectiveness of our methods

As we are investigating the effects of different strategies we proposed, it could be helpful to test for statistical significance. In this section, we detail, why we have concluded that this approach is not well-suited for our study and describe our chosen assessment methodology.

As a starting point, our data sample consists only of a handful of matrices. To mitigate this, relaxing the CFD and fluid flow constraints is the easiest approach. Also, experimenting with larger matrices could be achieved by utilizing non-local compute clusters. Assuming, we had the time and resources for such adjustments, there are other challenges that remain. The dependent variable is the TT-solve runtime estimate, while our predictor variables should be the chosen tile size, level of partial Gauss, padding and whether we used AMD or RCM reordering. Checking the individual effects is already problematic. The original matrix size n largely determines the possible tile sizes, which in turn have a large effect on what is possible. Since matrices come in different sizes, the same level of padding or PG will have a big influence on some matrices, while non at all on others. Therefore, statistical assessments of particular predictor variables such as level-k padding, cannot be assessed reliably. Of course, there are ways to mitigate this. For instance, using synthetic matrices of the same size would counter this effect. However, generating realistic matrices of this type goes beyond the scope of this project. Another possibility is to group experiment samples together that come from the same (reduced) matrix sizes. In any case, the added complexity is significant. On top of this, it is important to assess the effects when our strategies are combined together. While standard statistical tests assume the independence of these predictor variables, this condition is not met in several cases for us. Consider tile size choices together with padding or PG. Clearly, possible submatrix sizes directly depend on the level of padding or PG that is applied. Naturally, there are ways to counter these. However, such adjustments, together with the previously discussed challenges were deemed out of reach, when considering the resources available for this study.

As our analysis is of an exploratory nature, not testing for statistical significance but focusing on understanding the relationships between our variables and uncovering patterns, are of value on their own. To this end, we concentrate on gaining such insights by looking at our test matrices one by one and visualizing the general trends our methods promote. Additionally, we assess how much each of our proposals can improve the metrics described in Section 3.5.2. Finally, we use the best results to contrast the TT-solve runtime estimates with CG and Gaussian elimination, besides the original TT-solve time complexity.

3.5.5. Implementation details

In this section, we outline our software setup for the experiments.

Firstly, a choice had to be made regarding the programming language to be used. While several TT-related algorithms come implemented in Matlab [10, 65], Python has its fair share of available libraries for this domain too [72, 93]. Matlab comes with faster execution times, however, as we look at theoretical time complexities, this is not a primary concern. An important factor is the availability of efficient implementations of RCM and AMD variable ordering for sparse matrices besides partial Gaussian elimination. We found that both languages come with AMD and RCM implementations. For Python, these are available via the *cvxopt* [94] and *scipy* [71] libraries. However, we have not found readily available partial row reduction algorithms in either language. As these aspects did not tilt the scale to either side, the choice was made to use Python because of the following. It is an open-source, easy to read and widely used programming language for data analysis. Having to analyse our experiments and ensuring easy reproducibility, these characteristics aided the decision.

We have made custom implementations for padding, partial Gaussian elimination and the first stage of *matrix2mpo*, the *modeSize2rank* algorithm. This latter is used for computing the TTM-rank of a given sparse matrix for a chosen tile size. These implementations come complemented with extensive test suits. In order to run our experiments in parallel and track the results, the *wandb* package [95] was utilized. When experimenting with PG, we had to take extra precautions. As we work with ill-conditioned matrices, after reducing only a handful of variables, floating point errors lead to entries turning nonzero which should have remained zeros. For our use case this is particularly problematic. To mitigate this, we have added rounding to PG. The correct level of rounding was calibrated as follows. We used the LU decomposition implementation from *scipy* and counted the nonzero entries in the resulting U matrix. This U matrix corresponds to the row echelon form of the original matrix after (partial) Gaussian elimination is applied on all variables [96]. We choose the rounding level so that when we run our PG implementation on all variables, the nonzero pattern and count matches with U. The highest level of truncation threshold that we had to apply is $1e-3$.

On top of these, we implemented an extra measure for cases when the nonzero count explodes during PG. As we aim to optimize the TTM-decomposition process via *matrix2mpo*, working with dense matrices is pointless. To this end, when the sparsity ratio drops below 80%, we do not eliminate any more variables in the PG approach.

Lastly, let us highlight our data analysis practices. We make use of Jupyter Notebooks [97], the *pandas* library [98] for data manipulation and Plotly [99] for visualizations. The code repository of the project, together with all the test data, is available at `https://github.com/cbakos/sparse_tt_decomp_opt`.

4 Results

In this chapter, we detail our experimental results. We do this in two parts. First, we report how much our proposed strategies can reduce the theoretical runtime of the TT-solve approach. This part also includes an outlook on how much space is left for further improvements. Second, we compare the original and improved TT-solve approach with classical solvers such as CG and GE in terms of time complexity.

4.1. Sparse TTM decomposition optimization for TT-solve

In this section, we present the results of our experiments to assess the runtime reductions for TT-solve. Each proposed module's effects are examined individually. This is followed by the analysis of the combined approaches.

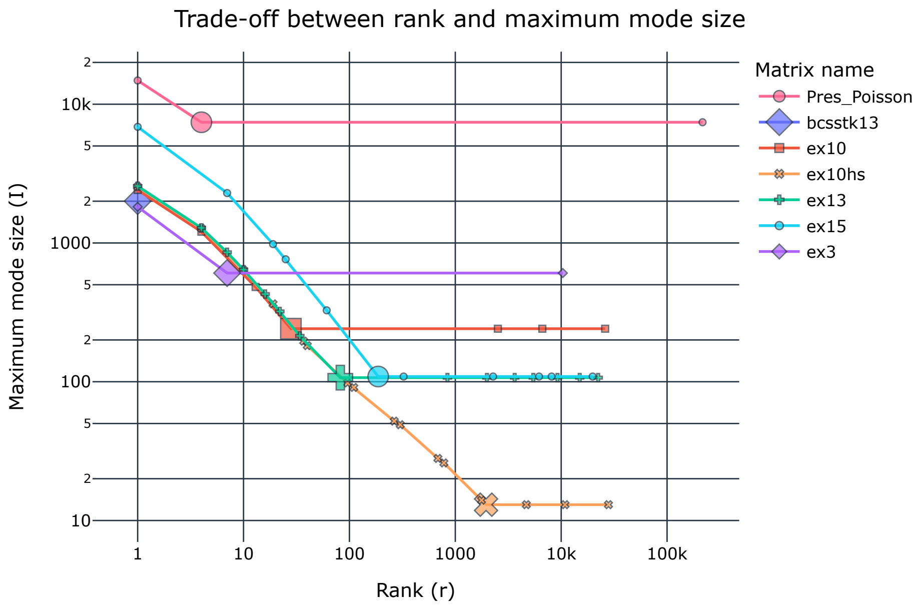
4.1.1. Tile selection

In this part, we show how choosing different tile sizes impacts the TT-MALS runtime bottleneck. The baseline tile size choices are indicated by the enlarged markers. That is, when the tile size choice corresponds to the largest prime factor of the matrix size n.

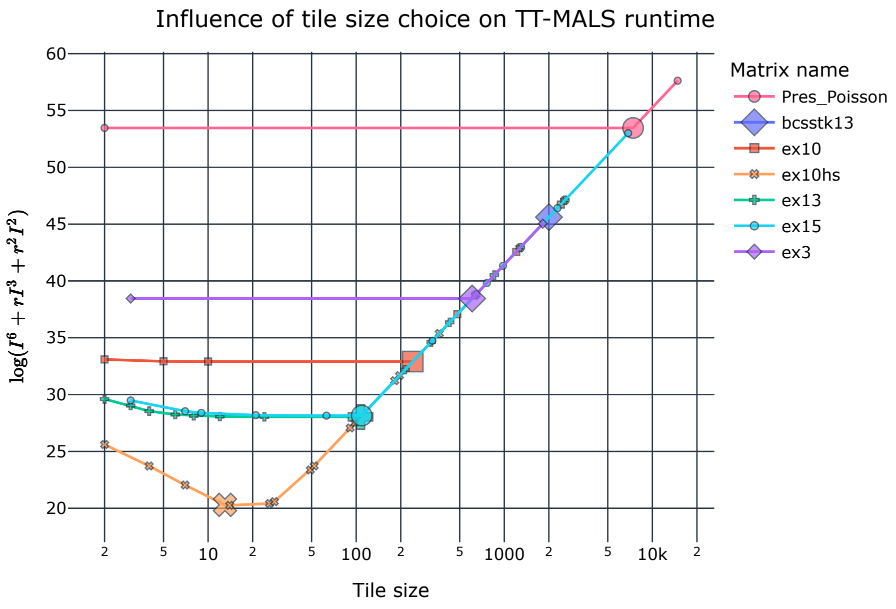
Figure 4.1a illustrates the tradeoff between the TTM-rank r and the maximum mode size I. Notably, as we choose tile sizes that yield higher ranks, the maximum mode size does not decrease but flattens out. This can be explained by the fact that choosing tile sizes smaller than the largest prime factor does not decrease the maximum mode size. However, it does tend to increase the rank as we generally need more tiles to cover all the nonzero entries in the matrix.

Figure 4.1b shows the TT-MALS runtime bottleneck as a function of the tile size for different matrices. We note how increasing the tile size eventually leads to a sharp rise in the runtime estimates for all matrices. This phenomena is caused by the I^6 term in Eq. (3.9). It blows up the time complexity as the maximum mode size becomes larger than the TTM-rank, thanks to the tile size increase.

Let us consider how different tile size choices compare to the default setting, using the largest prime factor. Based on Figure 4.1a, it appears that the largest prime factor indicates the breaking point in the tradeoff curve between maximum mode size and rank. This makes sense because choosing lower factors as tile sizes does not decrease the maximum mode size. In terms of runtime, because of the dominating I^6 term in Eq. (3.9), in Figure 4.1b the baseline tile choices mostly appear on the right, just before the runtime explodes due to bigger tile size choices. Interestingly, for *ex10hs* this is not the case. For this matrix, thanks to the many integer factors of its size n, it is possible to combine these to get slightly larger tile sizes than the baseline. These in turn yield smaller ranks, however, overall the runtimes appear similar to the baseline. If the objective function was less dominated by the maximum mode size, these newly constructed, slightly larger tile size choices likely lead to bigger reductions. Also, for other matrices, with the same objective function, it is possible, that slightly larger mode sizes - constructed from smaller prime factors - can reduce the TTM-rank enough to make that the optimal tile size choice. Therefore, we can conclude that choosing the largest prime factor as tile size is a very effective heuristic. However, depending on the objective function being minimized, it may be less likely to yield a result close to the optimal one.

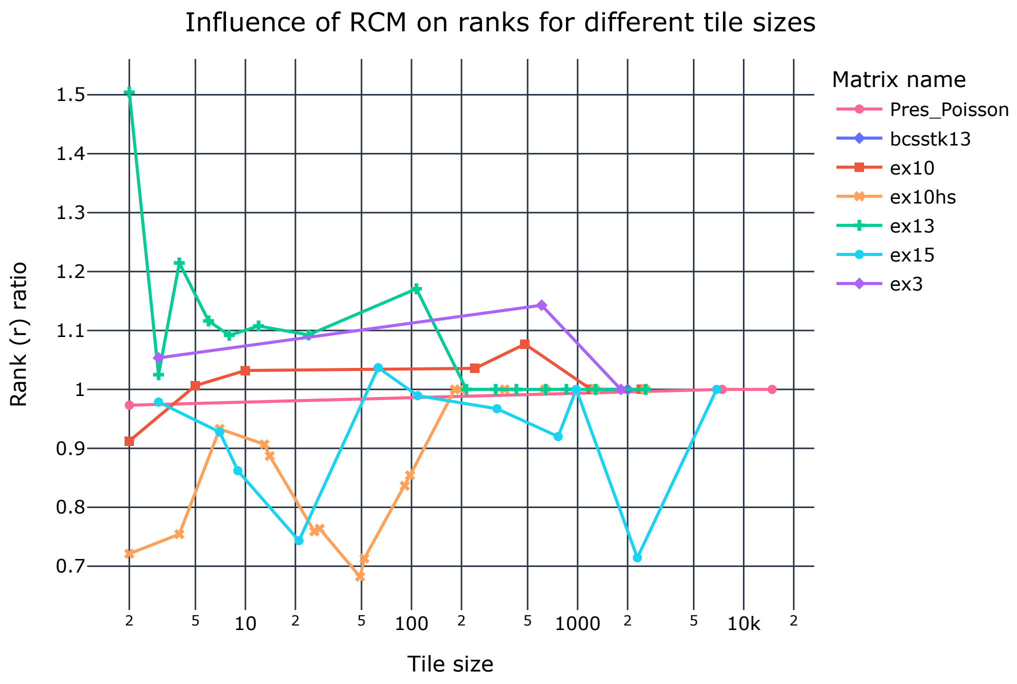


(a) This plot shows the relationship between the maximum mode size and the TT-rank for our test matrices. The markers indicate feasible tile sizes for the matrices, which in turn determine the TT-rank and maximum mode size of the TT-decomposition. While some matrices allow many different tile size choices (e.g. *ex10hs*), others like *bcsstk13* have only one possible combination as its size $n = 2003$ is a prime number.

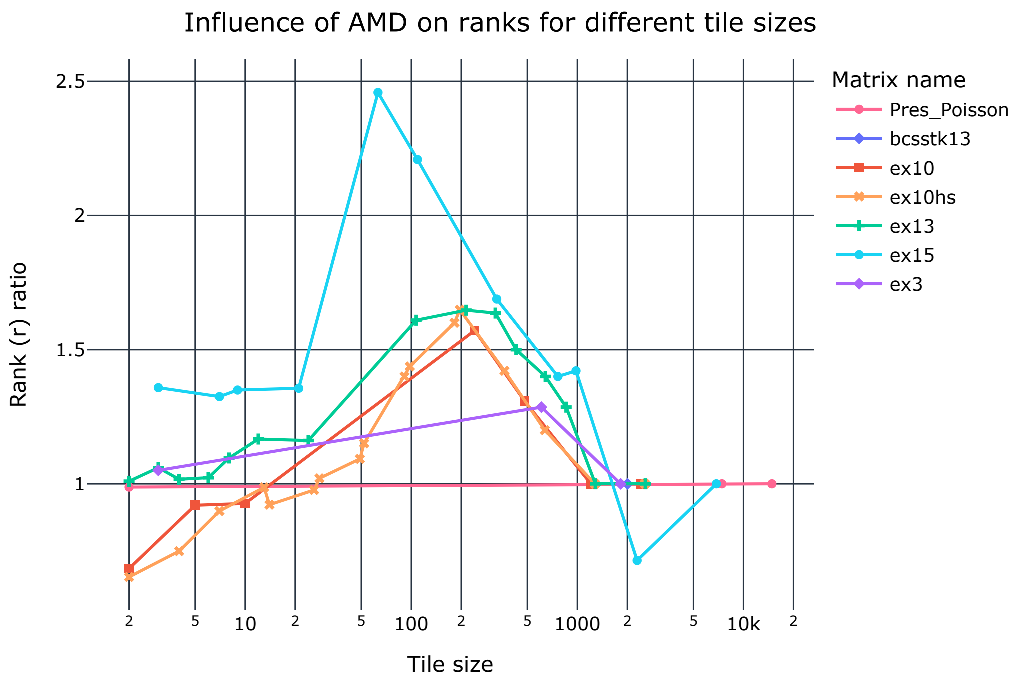


(b) This plot illustrates how choosing different factors as a tile size influences the runtime of the TT-MALS bottleneck. $\mathcal{O}(I^6 + rI^3 + r^2I^2)$ is the estimated time complexity when taking the TTM-rank r and maximum mode size I as variables. As the figure shows, the choice of possible tile sizes is largely constrained by the size of the matrix. Take for instance *ex3*, where $n = 1821 = 3 \cdot 607$ gives only 3 possible choices: $3, 607$ and 1821.

Figure 4.1: Illustration of the TT-decompositions that are possible on our test matrices when choosing different possible tile sizes for *matrix2mpo*. The enlarged markers indicate when the tile size choice corresponds to the largest prime factor of n, i.e. the baseline setting.



(a) Reverse Cuthill-McKee (RCM) ordering.



(b) Approximate Minimum Degree (AMD) ordering.

Figure 4.2: This visual showcases how the rank is effected by RCM and AMD variable reordering for each test matrix, choosing different tile sizes. The y-axis measures the ratio of the rank with respect to the original rank for a given matrix and tile size choice. For example if the rank was 100 without variable ordering but now its rank ratio is 0.7, then the new rank is $0.7 \cdot 100 = 70$ using the chosen variable ordering algorithm.

4.1.2. RCM variable ordering

We show the effects of the reverse Cuthill-McKee (RCM) variable ordering on our test matrices' rank in Figure 4.2a. We observe that for certain test matrices (*ex10hs, ex15*), when the tile size is under 100 the ranks are noticeably reduced. Assuming the prime factorization of the matrix size allows it, the best runtime estimates often occur for tiles sizes between 10 and 100, (see Figure 4.1b). Therefore, for some matrices, we can conclude the usefulness of RCM in this context. While useful, unfortunately, given our objective function to estimate the bottleneck of TT-MALS (Eq. (3.9)), the overall runtime is not significantly affected, as this approach leaves the maximum mode size unchanged.

In Figure 4.2a it is hard to tell whether RCM can be more effective for non-banded matrices (*Pres_Poisson, bcsstk13*) than the banded ones. This is caused by the limited integer factorization of the non-banded matrices' size. To this end, we postpone this assessment, until Section 4.1.6, where we consider RCM together with other modules that enable more tile size choices.

4.1.3. AMD variable ordering

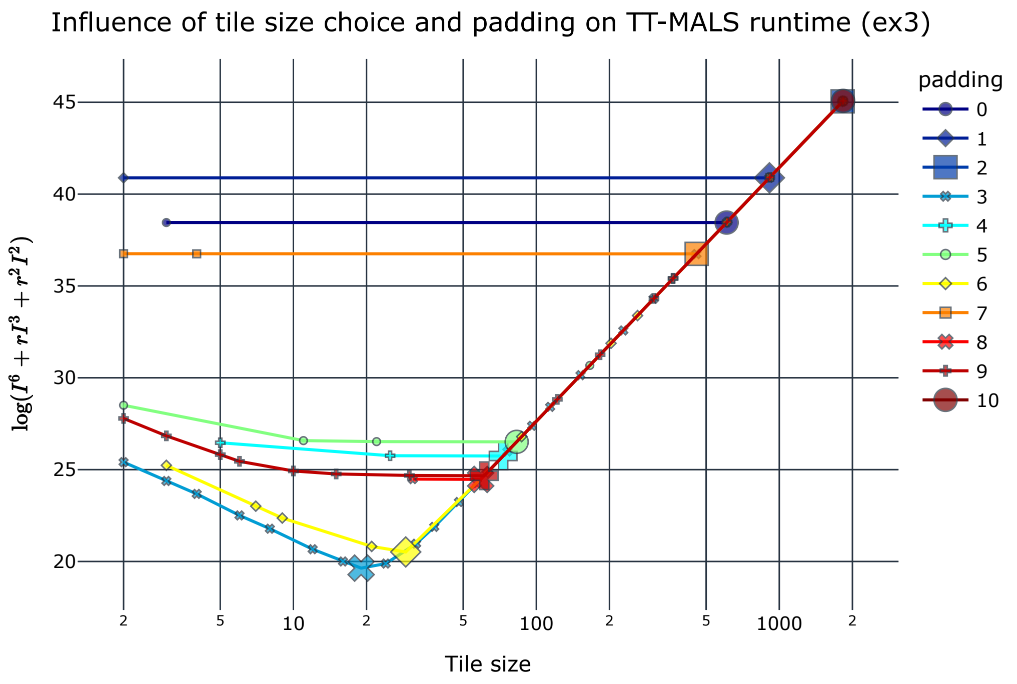
The Approximate Minimum Degree (AMD) variable ordering algorithm's main aim is to reduce fill-in when computing an LU decomposition or doing Gaussian elimination [84]. Therefore, it is not expected to enable better capture of the nonzero entries with the tiles used in *matrix2mpo* as a standalone module. We include it here for completeness. Later, we assess whether it can have a positive effect when used together with partial Gaussian elimination. Figure 4.2b confirms our suspicion that it does not improve the ranks of the TT-decomposition, in fact, for most tile sizes of interest (above 10 for our test matrices), it makes things worse.

4.1.4. Padding

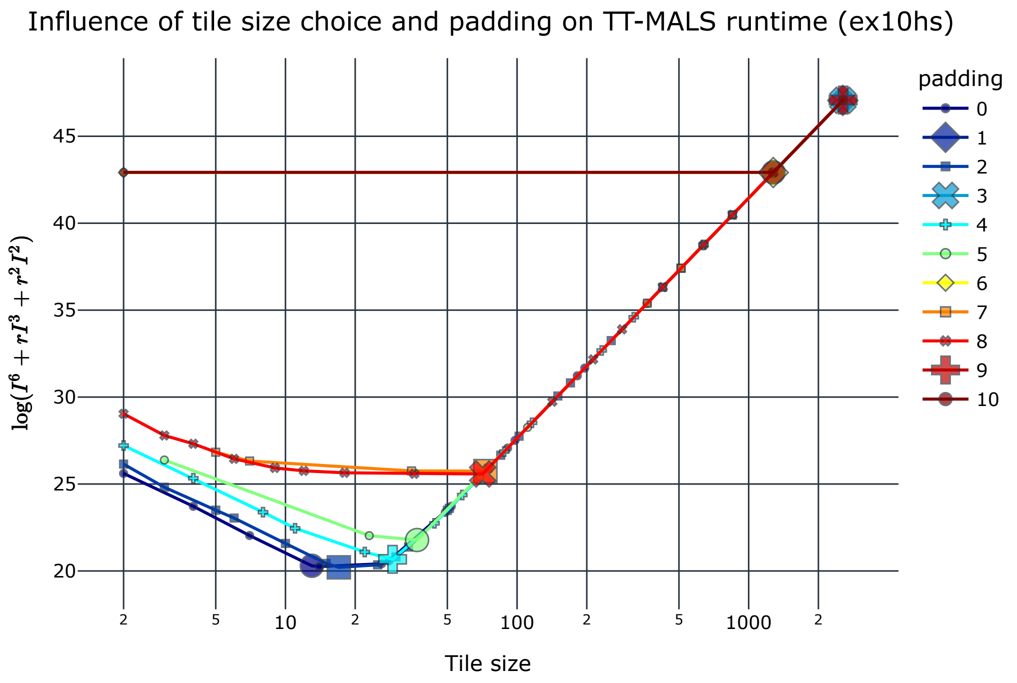
In this segment, we look at the results when padding is applied to our test matrices. First, the case of *ex3* in Figure 4.3a is analysed. For this matrix, the initial factorization (0-padding) makes the choice of possible tile sizes very constrained. This is because $n = 1821 = 3 \cdot 607$ when doing integer factorization. This means 3 possible tile size options: $3, 607$ and 1821. Adding padding mitigates this by allowing better factorizations. For instance, adding 3 levels of padding provides the best result in this case. This means that the new size of the matrix becomes $n' = 1824 = 2^5 \cdot 3 \cdot 19$, which clearly opens up many more tile size options.

To quantify the improvement, we can calculate the speedup ratio between the level-0 padding runtime estimate and the level-3 padding. On the plot (Figure 4.3a) we can see that the level-0 padding has around 38 on the y-axis in $\log$-scale. On the other hand, level-3 padding gives a runtime of around 20 on the $\log$-scale. Therefore, we get a speedup ratio of $\frac{e^{38}}{e^{20}} \approx 65.66 \cdot 10^6$.

For the *ex10hs* matrix, the original factorization (level-0 padding) is $n = 2548 = 2 \cdot 2 \cdot 7 \cdot 7 \cdot 13$. This allows for several different tile sizes choices: $2, 4, 7, 13, 14, 26, 28, 49, 52, 91, 98, 182, 196, 364, 637,$ 1274 and 2548. As Figure 4.3b illustrates, padding cannot improve the TT-MALS bottleneck estimate here because the original factors already provide sufficient coverage. Interestingly, when adding level-8 padding to the matrix, we can see that we have more tile size choices under 100 compared to the original setting. Nevertheless, level-0 padding gives the best result. This observation points out that having more factors does not necessarily translate to better TT-MALS runtimes. The explanation for this stems from the nature of padding. As we increase the size of the matrix, we tend to get slightly different tile sizes that perform the best. Depending on the exact location of the nonzero entries, having these slightly bigger or smaller tiles can mean that we need or do not need an extra block to capture the same set of entries. In cases when such extra blocks are needed, the rank is higher, leading to worse runtimes.

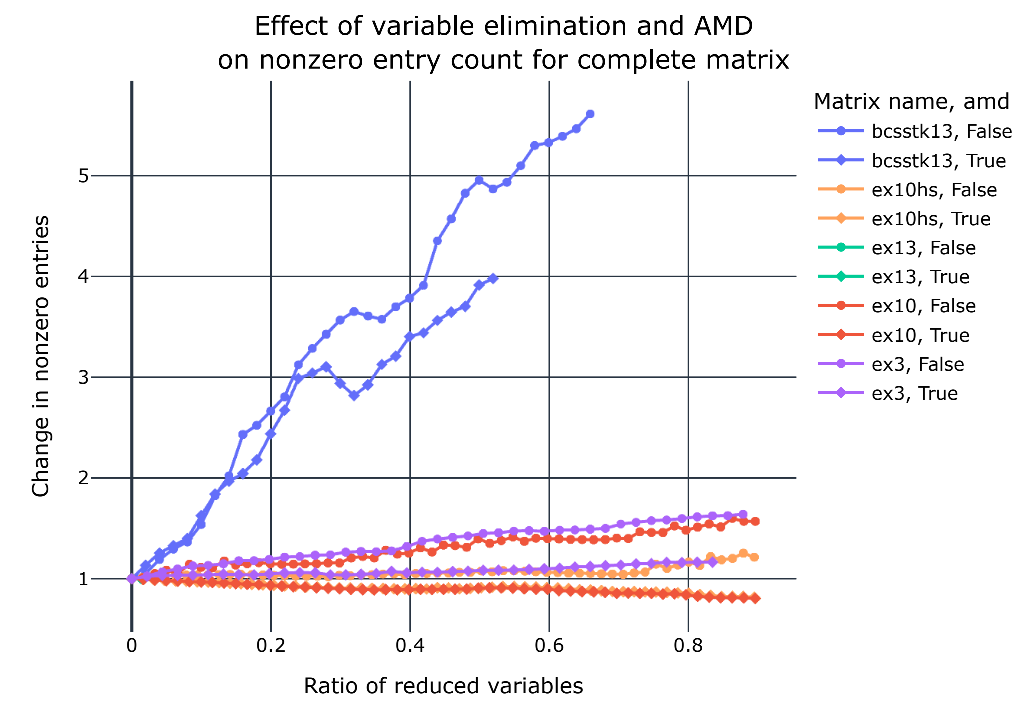


(a) This plot illustrates how different levels of padding for the *ex3* matrix allows for various tile size choices (x-axis), which in turn reduce the expected runtime bottleneck of TT-MALS (y-axis).

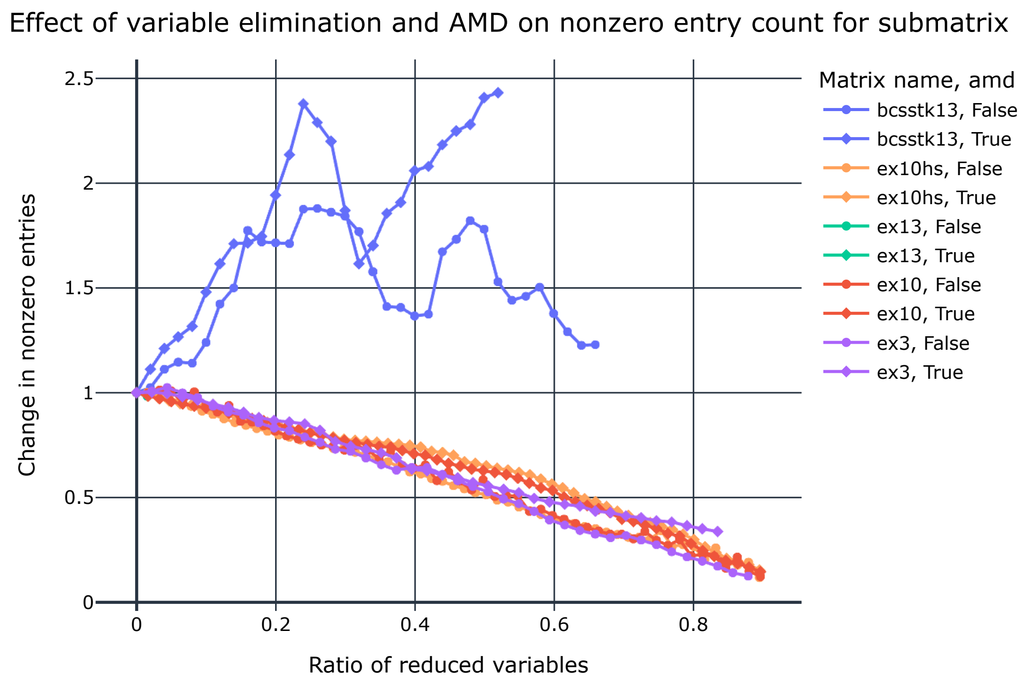


(b) This plot illustrates how different levels of padding for the *ex10hs* matrix allows for new tile size choices (x-axis), yet, it cannot reduce the expected runtime bottleneck of TT-MALS (y-axis).

Figure 4.3: Comparison of the effects of zero-padding on two different matrices. The enlarged markers indicate when the tile size choice corresponds to the largest prime factor of n, i.e. the baseline setting.



(a) This plot illustrates the change in nonzero entry count for the entire matrix.



(b) This plot shows the change in nonzero entry count for the submatrix that remains after the partial Gaussian elimination is finished.

Figure 4.4: These figures illustrate how the nonzero entry count changes when partial Gauss is applied with and without AMD to our test matrices that have $n < 3000$. The x-axis measures the ratio of reduced variables. For instance, if the matrix has size $n = 2410$, then in the x dimension, 0.2 indicates the result when $0.2 \cdot 2410 = 482$ variables are reduced. On the other hand, the y-axis measures the nonzero entry count change. The value 1 is the original one, e.g. 52685 for *ex3* and a value of 2 means that the original rate has doubled. When the output is below 1, then the nonzero count is reduced. For example, 0.2 means that only 20% remains of the original count. We note that the data points for *ex13* are available only for small ratios of reduced variables and hence hidden by the other results on this plot. As mentioned in Section 3.5.5 we terminate experiments where the sparsity drops below 80%. This was the case for this test matrix after a handful of eliminated variables, hence the absence of complete results.

4.1.5. Partial Gaussian elimination

In this part, we detail how partial Gaussian elimination affects the remaining submatrix which can be decomposed via the *matrix2mpo* algorithm to allow solving it in the TT-format. First, we study how the nonzero entry count is affected due to fill-in. This is important because having extra nonzero entries may require more tiles to capture them, leading to higher ranks when using the *matrix2mpo* algorithm. This is followed by checking the influence of Partial Gauss on the TT-MALS bottleneck estimate.

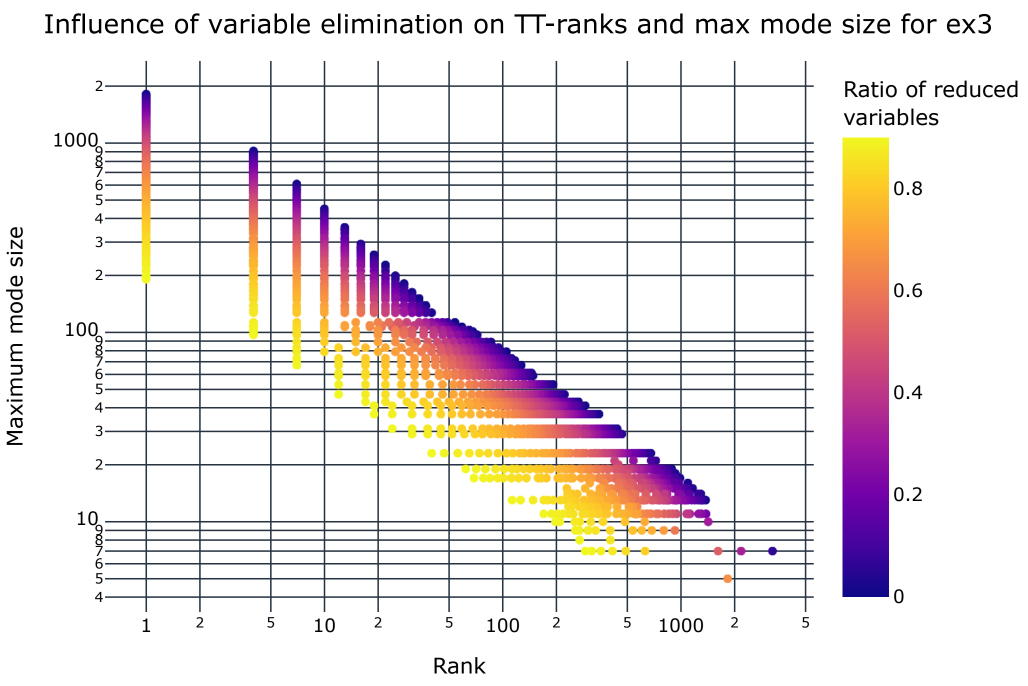
When looking at nonzero counts, we can do this in two different ways: either consider the complete matrix or only the remaining submatrix that is solved in the TT-format for our investigation or via CG in other studies [78, 80]. The former is in line with the nonzero entry count that AMD aims to minimize by reordering the variables. Depending on the implementation of Gaussian elimination, the change in nonzero count outside the submatrix may or may not be relevant. If a sparse-Gauss approach is used, fill-in will be very relevant for the runtime for this part. However, if a dense-Gauss reduction is applied, the runtime will not be affected as each entry will be iterated over outside of the submatrix anyhow. Consulting the literature [78, 80, 91] that used the combined Gauss-CG solver, we found no specification regarding these details. To this end, we look at both cases to capture the complete picture. Figure 4.4 shows these effects with and without AMD variable ordering. For this section, we focus on the case, when no AMD is applied.

There is a clear trend difference between the banded and non-banded (*bcsstk13*) matrices. Figure 4.4a shows that for the banded matrices the nonzero entry count for the full matrix does not grow over more than $1.6 - 1.7$ times the original number, when reducing up to 90% of the variables. In the meantime, for the non-banded matrix, the nonzero entry count explodes to over 5 times the original count, already when 60% of the variables are reduced.

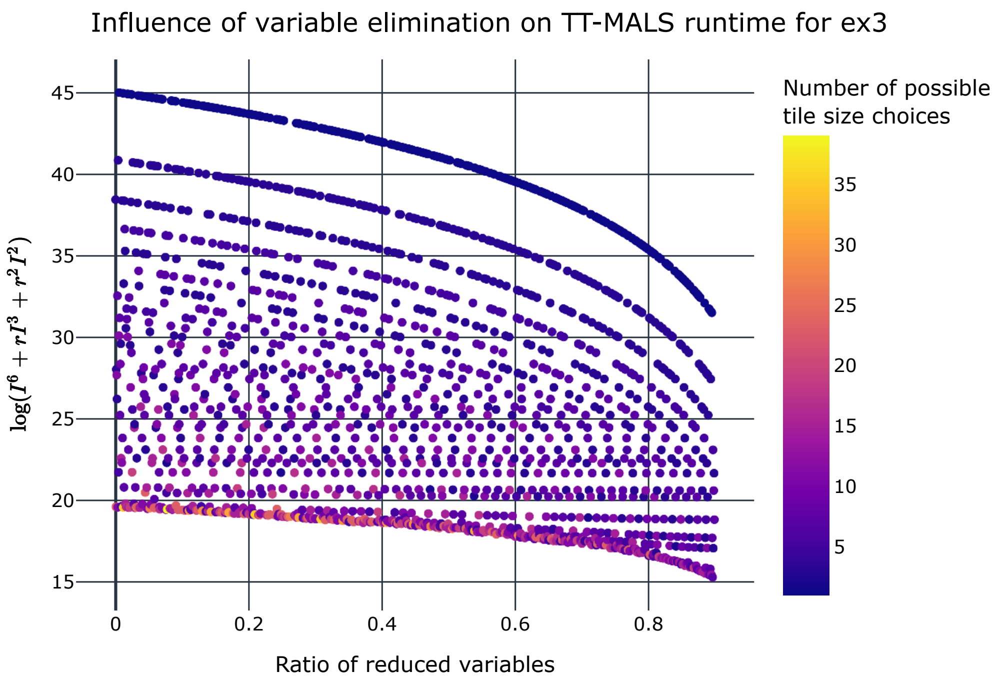
In Figure 4.4b we can see what happens for the submatrix. For the banded matrices, we can observe a relatively consistent, linear reduction in the nonzero count as more and more variables are eliminated. However, the non-banded case largely deviates from this tendency. The nonzero count is around 1.5-times the original value in the submatrix, when at least around 15% of variables are eliminated.

Based on these results showcased by Figure 4.4, we can make the following inferences. Firstly, for non-banded matrices, we can expect to run into trouble with the nonzero count increase when partial-Gauss is applied without any strategies to mitigate this. Secondly, for banded matrices, the nonzero count in the submatrix approximately reduces by the same ratio as the fraction of variables that are eliminated. In the meantime, the nonzero entry change for the complete matrix can slightly increase, however, it won't be as drastic as for the non-banded case.

Next, we look at Figure 4.5 to see how Partial Gauss influences the TT-MALS runtime bottleneck. Here, we only use the data points with the best tile size choice at each variable elimination step. A tile size choice becomes *best* by minimizing Eq. (3.9). Figure 4.5a illustrates how partial Gauss leads to better maximum mode size and rank combinations. On the other hand, Figure 4.5b aims to show the change in runtime estimates as more and more variables are eliminated. Additionally, it also unveils the large fluctuation in runtime estimates originating from the factorization possibilities at different steps. The coloring reveals that the more tile size choices are available, the better our chances to get the lowest runtimes.



(a) This figure shows how different scales of variable elimination affect the TT-rank and maximum mode size.



(b) This figure shows how different scales of variable elimination and the (lack of) tile size choices affect the estimated TT-MALS runtime.

Figure 4.5: This figure illustrates the effects of partial Gauss on *ex3*, which gives representative insights for the other test matrices as well. For these plots, only those data points are kept that minimize the TT-MALS runtime estimate by varying the tile size while keeping the other variables fixed.

4.1.6. Combined approaches

Here, we examine our experimental results for different combinations of the previously introduced approaches.

Padding then RCM

Once padding is applied, we can attempt to concentrate the nonzero entries using RCM. In Figure 4.6 we show the result of this approach. Firstly, we can observe that in the overwhelming majority of cases, the estimated runtime does not change (see the huge peak at 0 on the x-axis). In those few cases, when RCM does have an effect, depending on the matrix, the effect tendency is either positive or negative. In particular, for *Pres_Poisson, ex15, ex10hs* RCM leads to reductions while for *ex13, bcsstk13* we see the opposite. Since the non-banded matrices, *Pres_Poisson* and *bcsstk13* are split between the two classes, we can only conclude that RCM's effectiveness in this context is matrix dependent. I.e. it does not necessarily lead to improvements, just because the matrix has a non-banded structure.

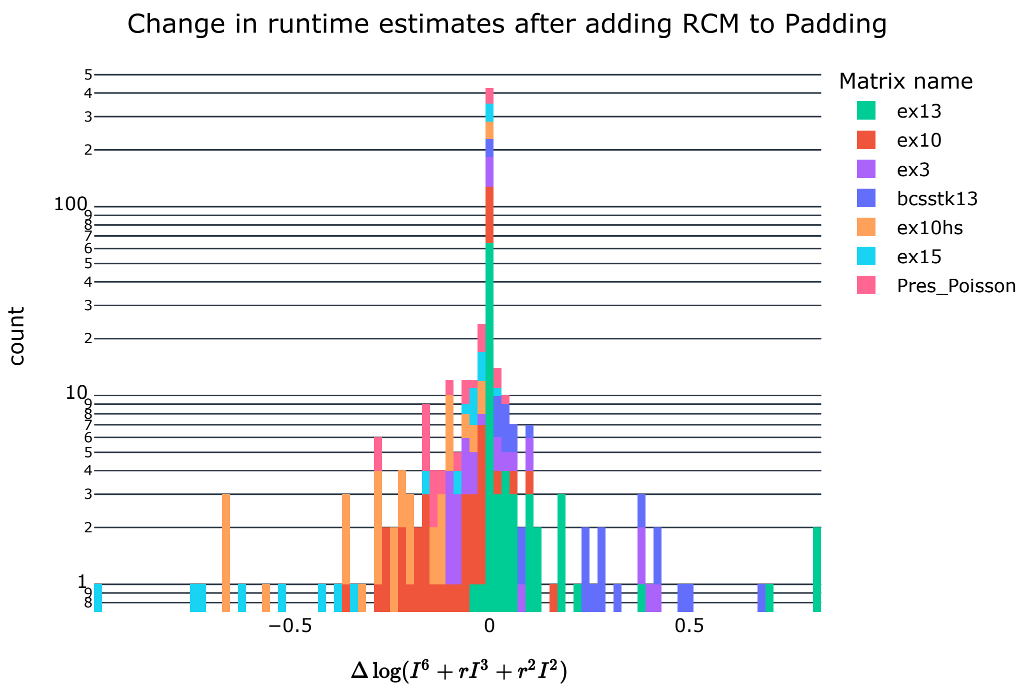
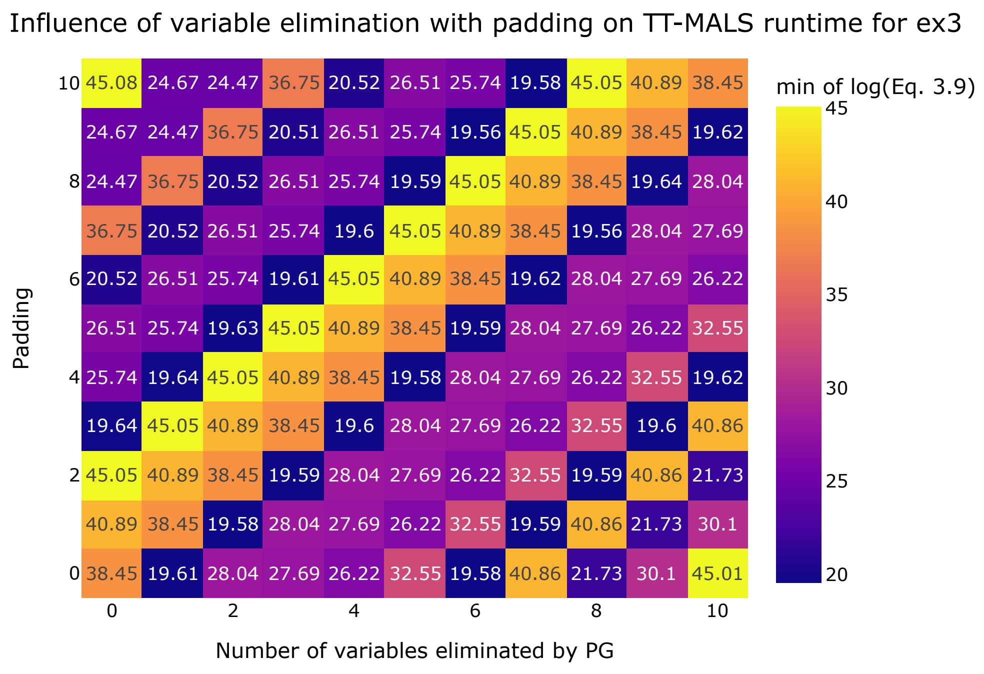


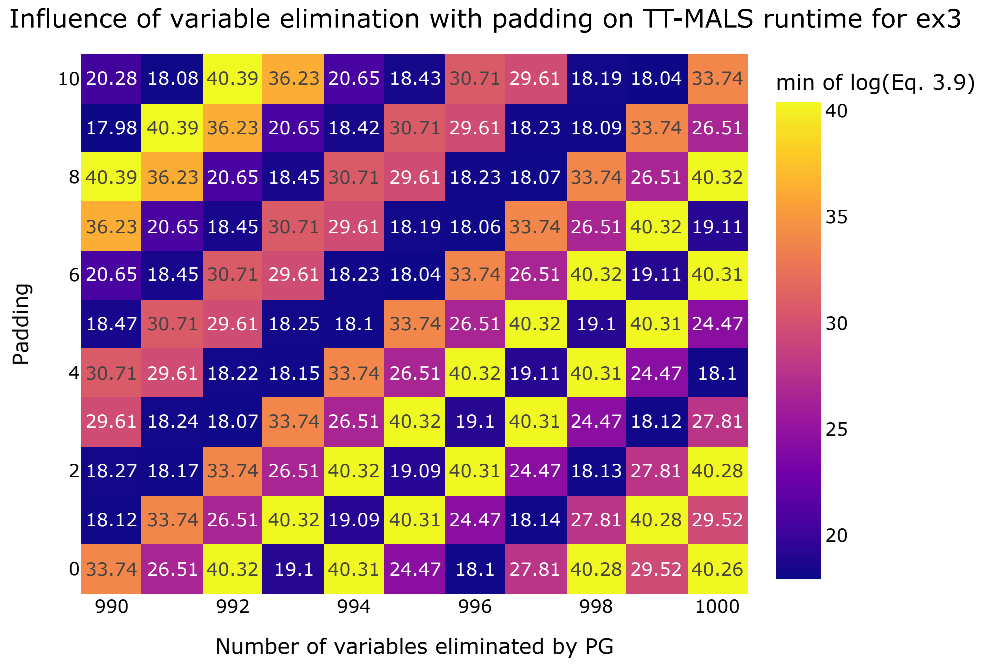
Figure 4.6: This histogram illustrates the change in the TT-MALS bottleneck estimate when we add RCM variable reordering after padding. All test matrices and samples are used to make the plot. Note that the y-axis, count, is in log-scale to make the smaller values visible too.

PG then Padding

After partial Gauss we can also apply padding. In Figure 4.7 we show what effect this has on the TT-solve runtime bottleneck for the *ex3* test matrix. A clearly visible pattern is how the runtime estimates remain (nearly) constant on the skew-diagonals. This phenomena can be explained as follows. Each skew-diagonal corresponds to a constant value when we take the difference between padding and PG levels. Taking the difference between the level of PG and padding equals the absolute change to the size of the matrix. As seen before, different matrix sizes allow for different integer factorization and in turn different tile size choices. Therefore, if we want to observe the effects of PG and padding without the effects of different factorizations, we need to compare entries within skew-diagonals. In most cases, these changes are under 0.01 which corresponds to a factor of $e^{0.01} \approx 1.01$. Meaning the shift in the runtime estimate remains under 1%. However a few cases also exhibit an alteration of 0.05 on the skew-diagonal. This latter situation corresponds to a factor of $e^{0.05} \approx 1.051$, that means a shift of 5.1% in the runtime.



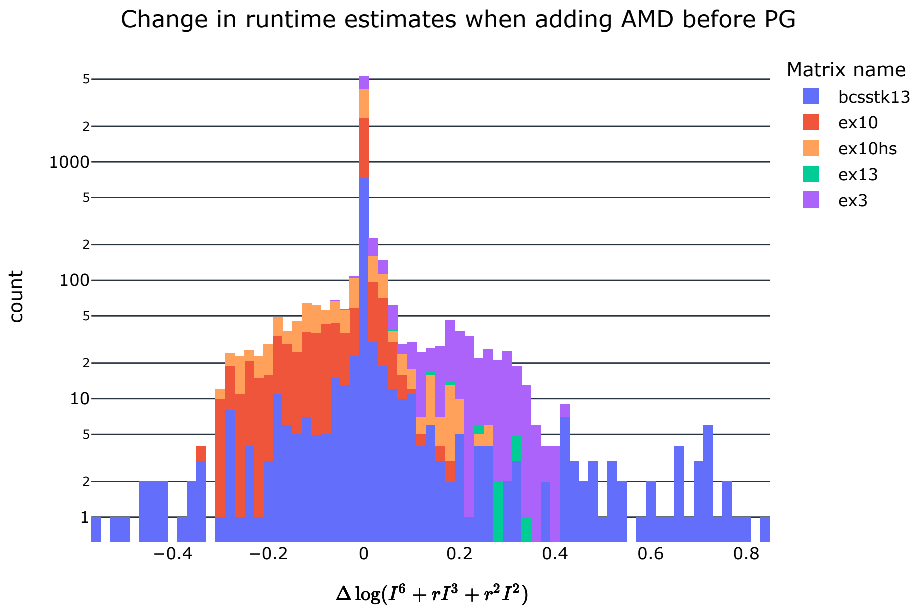
(a) 0-10 eliminated variables



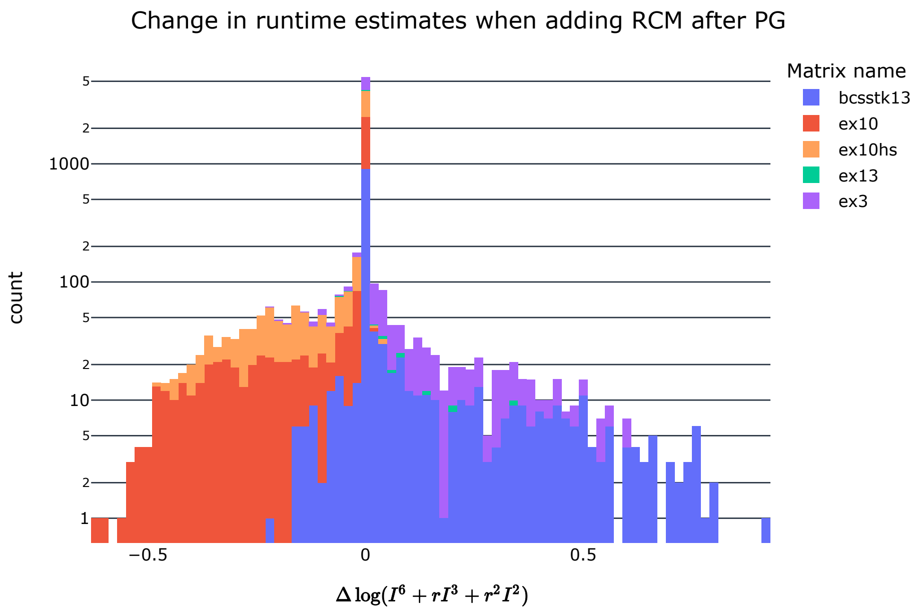
(b) 990-1000 eliminated variables

Figure 4.7: These heatmaps quantify the TT-MALS bottleneck estimate, when combining padding and partial Gauss for *ex3*. Only the best tile size choice data points are kept, minimizing the log of Eq. (3.9). We rounded the values to 2 decimal places for readability. Each column shows how different levels of padding alters the best possible runtime for a given level of partial Gauss. Rows show the opposite; for a fixed padding level, how PG effects the runtime.

In addition to these observations, as we compare Figures 4.7a and 4.7b, we can see the skew-diagonal pattern remains unchanged as more variables are eliminated by PG. However, in agreement with the observations in Figure 4.5b, we can notice a general reduction of runtimes as the number of eliminated variables grows.



(a) This histogram illustrates the case when we add AMD variable ordering in addition to partial Gauss.



(b) This histogram illustrates the case when we add RCM variable reordering after partial Gauss.

Figure 4.8: These stacked histograms show how two different approaches change the TT-MALS runtime bottleneck for different matrices. Note that the y-axis, count, is in log-scale to make the smaller values visible too. We only plot the data points that correspond to the best tile size choices. The two largest matrices are excluded here because of the computational infeasibility when doing PG.

AMD followed by PG

We can apply AMD variable reordering before doing partial variable elimination to minimize fill-in. We illustrate how this addition to PG affects the runtime estimates in Figure 4.8a. Similarly to the case of Figure 4.6, the overwhelming majority of cases show no change in runtime. For the *ex3* and *ex13* matrices we can observe a consistent runtime increase in those few cases when there is a shift, while *ex10* shows the opposite behaviour.

We also note an important observation regarding Figure 4.4 that shows how AMD and PG affect the nonzero count in the matrices. When looking at Figure 4.4a, we can see that AMD reduces the generated fill-in. However, for *bcsstk13*, the non-banded matrix, the nonzero count still explodes. On the other hand, in Figure 4.4b, AMD has a negative effect as the nonzero count is in fact increased in almost all data points shown. These observations, motivate the exploration of alternative variable ordering algorithms, in the hope of better fill-in reduction for the submatrix case as well.

RCM after PG

We can utilize RCM variable reordering after doing partial variable elimination to attempt to concentrate the nonzero entries around the diagonal. We demonstrate how this addition to PG affects the runtime estimates in Figure 4.8b. Similarly to the cases of Figure 4.6 and 4.8a, the overwhelming majority of cases show no change in runtime. We can again see that *ex3* and *ex13* matrices consistently have a runtime increase in those few cases when there is a shift, while *ex10* shows the opposite behaviour. Compared to 4.8a, here for *ex10hs*, RCM also appears to invariably lead to a lower runtime in a number of instances.

Higher level interactions

We do not consider in detail the cases when more than two methods are applied together. This decision was made because of the relatively small improvements of higher level interactions. As we will show in Figure 4.9, for our test matrices, we have less than a factor of 1 improvement compared to the best runtimes within level-two combinations.

4.1.7. Best performing methods

In this section, we assess which combination of our proposed approaches yield the best results. First, we look at the speedup gains using Eq. (3.10)-(3.13), then our focus shifts to the nonzero ratio of captured tiles by utilising Eq. (3.14). The latter analysis is particularly insightful in determining the possibility of future improvements.

Speedup gains

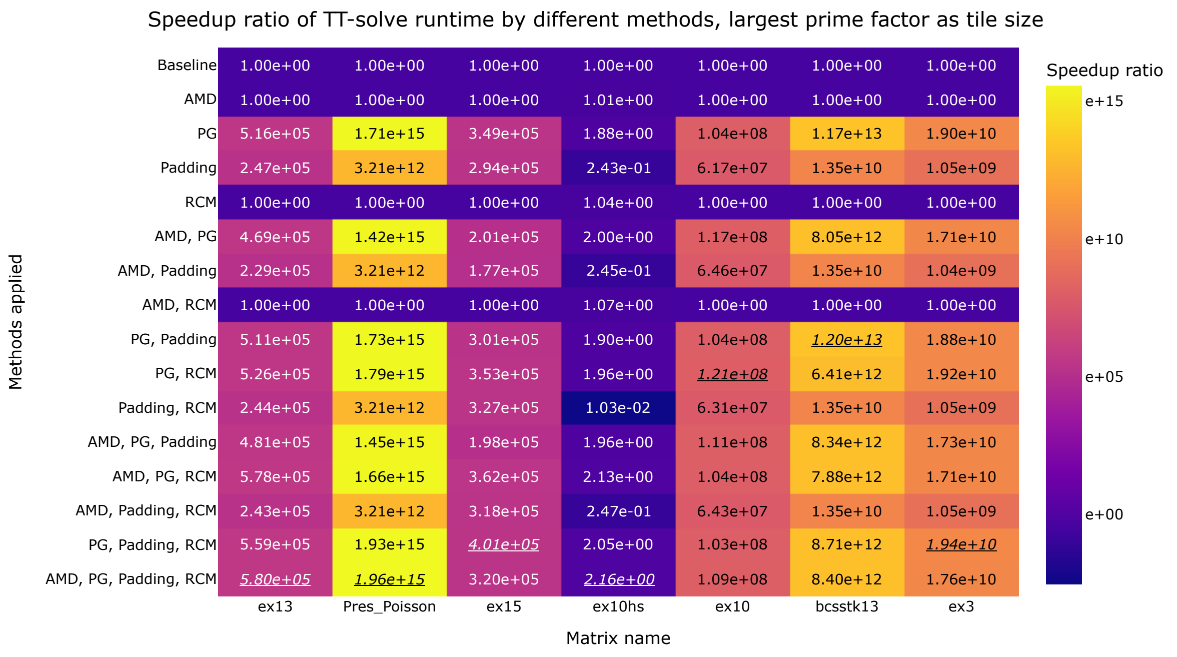
In Figure 4.9 we showcase the performance of each proposed method, both individually and in a combined fashion. We look at two variants, firstly, Figure 4.9a shows the results when no tile size selection is performed, i.e. the tile size equals the largest prime in the integer factorization of the matrix size n. In Figure 4.9b we show how different methods, together with choosing the best possible tile size, affect the estimated runtimes. In both cases, the *Baseline* method uses the largest prime factor as tile size.

There are a number of observations we can make in Figure 4.9. Firstly, we note that the main contributors to the speedups are padding and PG for all matrices. Looking at the *ex10hs* matrix, the effectiveness of these methods is limited. This can be traced back to the observations made in Figure 4.3b. As mentioned earlier, the size of *ex10hs* factorizes well, giving several tile size options to choose from. Therefore, the new factorizations enabled by padding and PG do not lead to significant changes.

While variable ordering techniques do not lead to speedups of several orders of magnitude, they play a crucial role in achieving the best possible speedups. This is supported by the recognition that for all matrices, except *bcsstk13*, RCM, sometimes together with AMD achieve the best speedup when added to PG and padding.

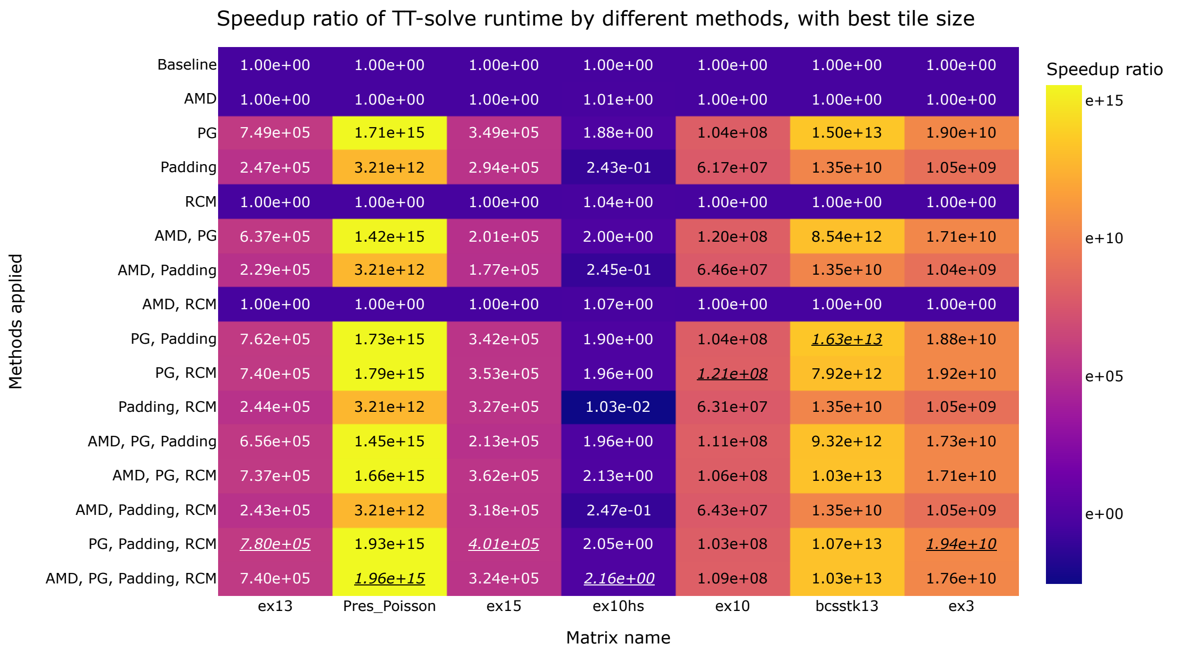
We can also see that the methods' effectiveness, in terms of speedup, is largely dependent on the matrix at hand. As our matrix sample is really small, we cannot reasonably draw any conclusions for these type of matrices in general.

Comparing Figures 4.9a and 4.9b we can also spot that the two heatmaps are very similar to each other. This points us to notice the small effect enabling tile size choice has on the speedups for this objective function. This is in line with our previous results in Section 4.1.1. These improvements are of similar magnitude to what AMD or RCM achieve on top of padding and PG. In 5 of of 7 cases, when looking at the best performing strategy combinations, choosing the largest prime factor as tile size already yields the best result.



Methods applied	ex13	Pres_Poisson	ex15	ex10hs	ex10	bcsstk13	ex3
Baseline	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00
AMD	1.00e+00	1.00e+00	1.00e+00	1.01e+00	1.00e+00	1.00e+00	1.00e+00
PG	5.16e+05	1.71e+15	3.49e+05	1.88e+00	1.04e+08	1.17e+13	1.90e+10
Padding	2.47e+05	3.21e+12	2.94e+05	2.43e-01	6.17e+07	1.35e+10	1.05e+09
RCM	1.00e+00	1.00e+00	1.00e+00	1.04e+00	1.00e+00	1.00e+00	1.00e+00
AMD, PG	4.69e+05	1.42e+15	2.01e+05	2.00e+00	1.17e+08	8.05e+12	1.71e+10
AMD, Padding	2.29e+05	3.21e+12	1.77e+05	2.45e-01	6.46e+07	1.35e+10	1.04e+09
AMD, RCM	1.00e+00	1.00e+00	1.00e+00	1.07e+00	1.00e+00	1.00e+00	1.00e+00
PG, Padding	5.11e+05	1.73e+15	3.01e+05	1.90e+00	1.04e+08	*1.20e+13*	1.88e+10
PG, RCM	5.26e+05	1.79e+15	3.53e+05	1.96e+00	*1.21e+08*	6.41e+12	1.92e+10
Padding, RCM	2.44e+05	3.21e+12	3.27e+05	1.03e-02	6.31e+07	1.35e+10	1.05e+09
AMD, PG, Padding	4.81e+05	1.45e+15	1.98e+05	1.96e+00	1.11e+08	8.34e+12	1.73e+10
AMD, PG, RCM	5.78e+05	1.66e+15	3.62e+05	2.13e+00	1.04e+08	7.88e+12	1.71e+10
AMD, Padding, RCM	2.43e+05	3.21e+12	3.18e+05	2.47e-01	6.43e+07	1.35e+10	1.05e+09
PG, Padding, RCM	5.59e+05	1.93e+15	*4.01e+05*	2.05e+00	1.03e+08	8.71e+12	*1.94e+10*
AMD, PG, Padding, RCM	*5.80e+05*	*1.96e+15*	3.20e+05	*2.16e+00*	1.09e+08	8.40e+12	1.76e+10

(a) In this visual, all methods use the largest prime factor of the matrix size factorization as the tile size.



Methods applied	ex13	Pres_Poisson	ex15	ex10hs	ex10	bcsstk13	ex3
Baseline	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00
AMD	1.00e+00	1.00e+00	1.00e+00	1.01e+00	1.00e+00	1.00e+00	1.00e+00
PG	7.49e+05	1.71e+15	3.49e+05	1.88e+00	1.04e+08	1.50e+13	1.90e+10
Padding	2.47e+05	3.21e+12	2.94e+05	2.43e-01	6.17e+07	1.35e+10	1.05e+09
RCM	1.00e+00	1.00e+00	1.00e+00	1.04e+00	1.00e+00	1.00e+00	1.00e+00
AMD, PG	6.37e+05	1.42e+15	2.01e+05	2.00e+00	1.20e+08	8.54e+12	1.71e+10
AMD, Padding	2.29e+05	3.21e+12	1.77e+05	2.45e-01	6.46e+07	1.35e+10	1.04e+09
AMD, RCM	1.00e+00	1.00e+00	1.00e+00	1.07e+00	1.00e+00	1.00e+00	1.00e+00
PG, Padding	7.62e+05	1.73e+15	3.42e+05	1.90e+00	1.04e+08	*1.63e+13*	1.88e+10
PG, RCM	7.40e+05	1.79e+15	3.53e+05	1.96e+00	*1.21e+08*	7.92e+12	1.92e+10
Padding, RCM	2.44e+05	3.21e+12	3.27e+05	1.03e-02	6.31e+07	1.35e+10	1.05e+09
AMD, PG, Padding	6.56e+05	1.45e+15	2.13e+05	1.96e+00	1.11e+08	9.32e+12	1.73e+10
AMD, PG, RCM	7.37e+05	1.66e+15	3.62e+05	2.13e+00	1.06e+08	1.03e+13	1.71e+10
AMD, Padding, RCM	2.43e+05	3.21e+12	3.18e+05	2.47e-01	6.43e+07	1.35e+10	1.05e+09
PG, Padding, RCM	*7.80e+05*	1.93e+15	*4.01e+05*	2.05e+00	1.03e+08	1.07e+13	*1.94e+10*
AMD, PG, Padding, RCM	7.40e+05	*1.96e+15*	3.24e+05	*2.16e+00*	1.09e+08	1.03e+13	1.76e+10

(b) In this visual, all methods (except the Baseline) utilise the possibility of choosing the best tile size, constructed from the integer factorization of the matrix size.

Figure 4.9: These heatmaps illustrate the speedup ratio on the estimated TT-solve runtimes when different improvements are applied per matrix. The best results per matrix are underlined and are shown in *italic* font. Here we apply at most 10 levels of padding and/or PG. This is to ensure that the additional time complexity of PG does not ruin the overall runtime.

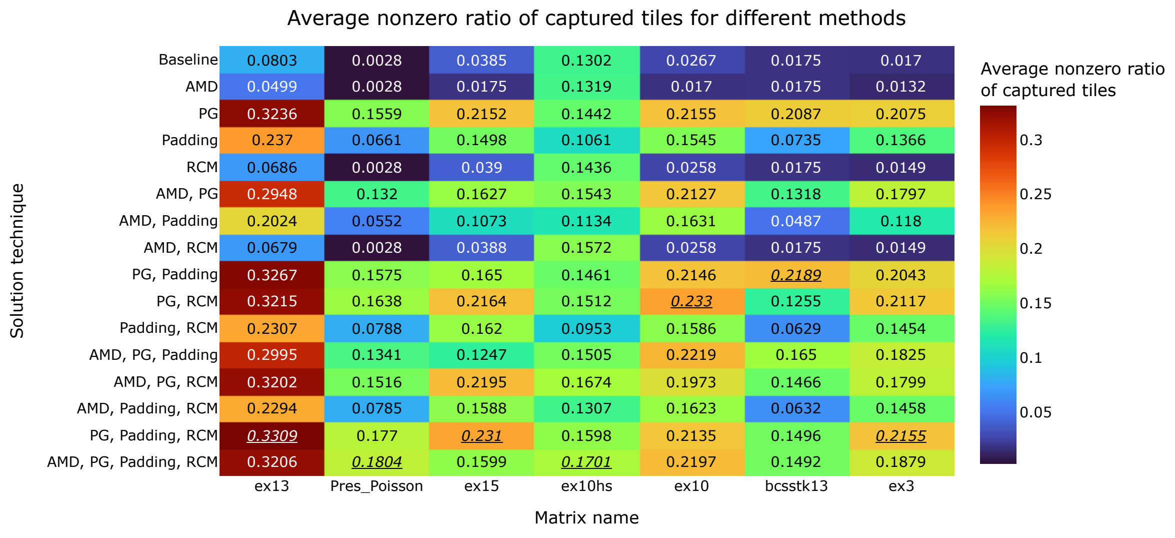
Overall padding and PG appear as the most effective methods, however to achieve the maximal speedup gains, combined approaches are needed. This is evident, as the best results are obtained when all 4 methods are applied for 2 out of the 7 matrices. 3 cases need the combination of 3 methods and lastly 2 matrices get the best speedups when 2 approaches are combined. When the tile size choice is restricted, the best speedup for *ex13* needs all 4 methods combined, instead of 3.

Nonzero ratio of captured tiles

In addition to evaluating the speedup gains, we also look at the nonzero ratio of the captured tiles for *matrix2mpo*. The metric, we use to quantify this, is given by Eq. (3.14). In particular, in Figure 4.10 we illustrate the results for this criterion for each matrix and proposed improvement.

Firstly, when comparing it to Figure 4.9, we can see high correlation between the (best) speedup and the (highest) nonzero ratio for the tiles. I.e., having a higher nonzero ratio for a particular tile size choice means a lower rank. This happens because we capture all nonzero entries with fewer tiles. Therefore, having a higher ratio for a given tile size leads to better runtimes.

Figure 4.10 is especially helpful in understanding the limited speedup gains in the case of *ex10hs*. As we can see, for every other test matrix, we start with a very small nonzero ratio (below 9%) in the captured tiles. After our improvements, the best results correspond to the cases when the ratio has significantly increased. In particular, these advancements mean a minimum absolute value increase of 0.18, while for *ex10hs* this change is only 0.04. In combination with the high baseline ratio of *ex10hs* of 0.13, this indicates the limitations of our proposed strategies. It appears that our methods are more effective in cases when this nonzero ratio is lower at the start.



Solution technique	ex13	Pres_Poisson	ex15	ex10hs	ex10	bcsstk13	ex3
Baseline	0.0803	0.0028	0.0385	0.1302	0.0267	0.0175	0.017
AMD	0.0499	0.0028	0.0175	0.1319	0.017	0.0175	0.0132
PG	0.3236	0.1559	0.2152	0.1442	0.2155	0.2087	0.2075
Padding	0.237	0.0661	0.1498	0.1061	0.1545	0.0735	0.1366
RCM	0.0686	0.0028	0.039	0.1436	0.0258	0.0175	0.0149
AMD, PG	0.2948	0.132	0.1627	0.1543	0.2127	0.1318	0.1797
AMD, Padding	0.2024	0.0552	0.1073	0.1134	0.1631	0.0487	0.118
AMD, RCM	0.0679	0.0028	0.0388	0.1572	0.0258	0.0175	0.0149
PG, Padding	0.3267	0.1575	0.165	0.1461	0.2146	*0.2189*	0.2043
PG, RCM	0.3215	0.1638	0.2164	0.1512	*0.233*	0.1255	0.2117
Padding, RCM	0.2307	0.0788	0.162	0.0953	0.1586	0.0629	0.1454
AMD, PG, Padding	0.2995	0.1341	0.1247	0.1505	0.2219	0.165	0.1825
AMD, PG, RCM	0.3202	0.1516	0.2195	0.1674	0.1973	0.1466	0.1799
AMD, Padding, RCM	0.2294	0.0785	0.1588	0.1307	0.1623	0.0632	0.1458
PG, Padding, RCM	*0.3309*	0.177	*0.231*	0.1598	0.2135	0.1496	*0.2155*
AMD, PG, Padding, RCM	0.3206	*0.1804*	0.1599	*0.1701*	0.2197	0.1492	0.1879

Figure 4.10: This heatmap shows the average nonzero ratio of the tiles used in the *matrix2mpo* algorithm for each strategy and matrix. The baseline uses the largest prime factor as tile size, while the other ones utilize the best possible tile choice based on the TT-solve runtime. This ratio corresponds to the number of nonzero entries in the (modified) matrix, divided by the number of entries the nonzero submatrices capture. I.e. the divisor is the TTM-rank multiplied by the tile size squared. We report the average value because of two reasons. Firstly, the submatrices have different ratios across the matrix. As the variance or distribution is not of interest at this time, we focus on the average only. Additionally, this mean value is the same as the nonzero ratio of the first TTM-core in the TTM-decomposition.

4.2. TT-solve applicability for matrices

In this section, we look at how traditional techniques like Gaussian elimination and Conjugate Gradient compare against the runtime of TT-solve, with and without the proposed improvements. Figure 4.11 shows the comparison per matrix across the different approaches. For this analysis, we include the LSE conversion times to and from the TT-format. I.e. the runtime criterion is given by Eq. (3.10)-(3.13).

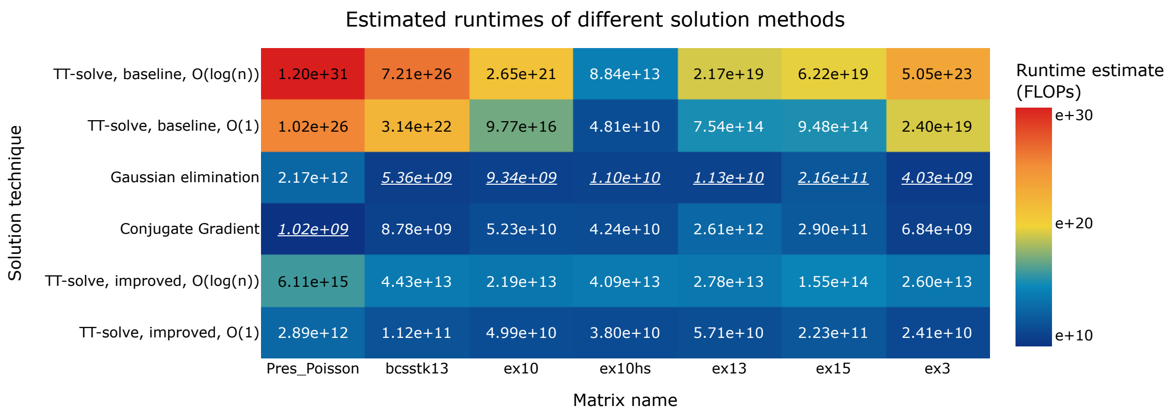


Figure 4.11: This heatmap illustrates the estimated runtimes of solving linear systems of equations by different methods. We use the runtime estimates as stated in Table 3.1. The baseline TT-solve refers to the TT-MALS runtime without AMD, PG, Padding, RCM or choosing the best tile size. The improved TT-solve indicates the best runtime achieved by a combination of our proposed techniques. In both cases, the runtime estimates include the time needed to convert the LSE to the TT(M)-format. Additionally, they also include the time needed to convert the solution back to a vector from the TT-format. We differentiate between two different scenarios. The $\mathcal{O}(1)$ one happens when we assume that the number of iterations needed for TT-MALS to converge is constant ($c = 1$). Additionally, the solution vector can be well-approximated with a TT-rank of $\mathcal{O}(1)$, w.r.t. the matrix size n, in particular we take $s = 2$. In the $\mathcal{O}(\log(n))$ scenario, we take $c = \log(n)$ and $s = \log(n)$. In both situations, $d = \log(n)$, in line with our derivation in Section 3.2.3. In the baseline case I is given by the largest prime factor of the matrix size. This in turn determines the value of r. When looking at the improved situation, the r and I pair is chosen across our evaluated strategies so that they minimize the TT-solve runtime, including conversions. Regarding the traditional solver approaches we note the following. The Gaussian elimination runtime estimate does not utilize the potential gains which can be achieved on banded matrices. Similarly, we show CG runtime estimates here without any preconditioning. This decision was made because their effectiveness is largely problem dependent and the number of possible preconditioner techniques to try are countless.

The best theoretical runtimes are achieved by Gaussian elimination and CG for all our test matrices. As we have discussed earlier, our strategies have limited effects for *ex10hs*, which can be observed in this plot as well. However, for the other test matrices, we can see that the significantly higher baseline runtimes of TT-solve are reduced to the same order of magnitude as CG and GE, under the $\mathcal{O}(1)$ assumption scenario. An exception to this is the *Pres_Poisson* matrix which comes with a lower condition number, thus computing its solution is more efficient via CG. In the case of *ex10hs*, we can see that the baseline TT-solve runtime is already comparable to GE and CG under the $\mathcal{O}(1)$ premise.

Overall, we can conclude that our proposed methods are able to bring down the TT-solve time to comparable FLOP counts to what CG and GE require. This is assuming that the expected time complexity is not already comparable as in the case of *ex10hs*. Notably, we still rely on strong assumptions regarding the TT-MALS convergence and TT-rank of the right-hand side and solution vector. On the flip side, our analysis is done on general sparse, CFD matrices which have no special structure to allow for very low numerical or TTM-ranks as seen in Table 2.1. Recalling our observations regarding Figure 4.10, we know that our methods were able to increase the nonzero ratio in the captured tiles to at most around 33%. This means, that for a given tile size I, the lowest possible TTM-rank is only at around $\frac{1}{3}$ of its theoretical limit. It remains a challenge to achieve better nonzero entry concentrations, however, our analysis shows the possibility. Therefore, we can conclude that the door is opened for future research to achieve better TT-solve runtimes than CG and GE, for general, sparse and ill-conditioned linear systems of equations.

5 Discussion

In this chapter, we highlight the limitations of this study, then discuss our main results, and finally give pointers for future investigations in this subject.

5.1. Limitations

In this section, we review the restrictions of this research project. First, we recap the benefits and drawbacks of our runtime assessment methodology. This is followed by the implications of our test dataset used in the experiments. Lastly, we address the assumptions that were made regarding the TT-solve approach and the runtime of the CG and GE solvers.

5.1.1. Runtime assessment methodology

The way we have assessed the runtimes of algorithms is highly theoretical. While this has several advantages, there are obvious drawbacks to this approach too. This method of counting primitive operations or in certain cases just FLOPs, often combined with the big-$\mathcal{O}$ notation has been the backbone of this project. Its ease of use contributed to the assessment of the time complexity of various algorithms under a short period of time. This approach, does not just give reasonable estimates on how these scale, depending on the size of the input, but also it does not suffer from runtime dependencies on our particular hardware setup.

This, however, also means that we neglect several details that influence the true runtime of the algorithms in question. For example, for Conjugate Gradient (CG) and Gaussian elimination (GE), there are highly optimized code bases available [100, 101] that have been developed over several decades. In the meantime, while TT-methods can partially reuse these libraries, overall, their codebases are not yet optimized to this extent [65]. Another observation is that certain primitive operations are simply faster than others in practice [12], although we assume that they all take the same amount of time. While our analysis has taken into account lower order terms, we have neglected several constant factors for the runtime assessment of the TT-methods. Therefore, our assessment shows the right trend for large inputs, but for smaller ones, these constant factors could still dominate, yielding results different from what our findings suggest.

5.1.2. Test data

Our study's reliance on a limited set of test matrices imposes constraints on the generalizability of our findings. First, the matrices used are exclusively drawn from the sparse matrix collection available at [69]. Our selection is further narrowed to symmetric and positive definite matrices originating from computational fluid dynamics (CFD) problems related to the flow of water or other liquids. This focused selection was motivated by the domain of interest at Deltares. We note however, that our test matrices do not follow any special structure such as Toeplitz, Hankel or Cauchy matrices that are known to have low numerical rank [2]. Therefore, we do expect our general findings to hold for similar, ill-conditioned, sparse matrices, coming from CFD problems and potentially from other application domains.

Our choice to work with smaller matrices (with $n < 15000$ and $z < 800000$) was driven by the practical consideration of keeping experiment runtimes feasible on personal computing resources. We have also excluded one matrix (FIDAP/ex5) due to its outlier status in terms of sparsity, further narrowing our dataset. The resulting small sample size of just seven matrices, constrains the statistical power of our analyses and limits our ability to generalize the results to larger or differently structured matrices.

Finally, the nature of our exploratory analysis, which aims to understand relationships between variables and identify patterns, precludes the use of statistical significance testing. The complexity of interactions between predictor variables, such as tile size, level of partial Gaussian elimination (PG), padding, and reordering strategies (AMD or RCM), combined with the dependency on matrix size, makes it challenging to isolate the effects of individual factors. Although methods like using synthetic matrices or grouping matrices by size could mitigate some of these issues, such approaches were beyond the scope of our study.

In summary, while our findings provide valuable insights into the performance of the tested methods, the limitations in matrix selection, sample size, and the exploratory nature of our analysis restrict the broader applicability of our conclusions. Future work could address these limitations by expanding the matrix dataset, including larger and more varied matrices, and employing more robust statistical methods to evaluate the effects of individual and combined factors.

5.1.3. TT-solve assumptions

Our results stated in Figure 4.11 rely on strong low-rank assumptions which, if relaxed, would lead to increased runtimes for the TT-solve approach. Specifically, the TT-rank of the solution vector, right-hand side and the number of iterations required for the convergence of TT-MALS are assumed to be $\mathcal{O}(1)$ or $\mathcal{O}(\log(n))$, depending on the scenario. As we saw, the former allowed us to reach the same order of time complexity for several of our test matrices as traditional solvers, using our proposed improvements. However, when only the weaker assumption was satisfied, it has been clear that the TT-solve algorithm would be still considerably slower than Conjugate Gradient (CG) or Gaussian elimination. This highlights the importance of the TT-ranks of the vectors involved in the linear system too. This, we did not explore in detail, as our focus has been on improving the TT-decomposition of sparse matrices. This primary objective was chosen as the main runtime bottleneck of TT-solve lies here. Additionally, by itself, the *matrix2mpo* conversion method does not impact the TT-ranks of the vectors.

We have not looked at the effects of our improvements on the TT-ranks of the right-hand side or the solution vector. We suspect that our strategies can also alter the ranks of these. For instance, in the case of padding or PG, we change the mode sizes of the right-hand side vector as well, when decomposed. Therefore, having an effect on the TT-rank, due to the link between them mentioned in Section 3.2.2. Assessing the magnitude of such effects is left for future investigations into this subject.

For practical considerations, it is an important question to ask whether our low-rank and convergence assumptions are reasonable or not. Based on the existing TT-literature [6, 7, 8, 61, 63], we could not find definitive answers on the convergence of TT-MALS or low-rank TT-approximability of vectors of LSEs. This is expected, as in general these properties are highly problem dependent. The good news is, that existing TT-algorithms come with parameters to specify the maximum number of iterations and maximal ranks [65]. Therefore, one can easily restrict these to allow reasonable runtimes and then check whether the result is good enough by checking the introduced error.

5.1.4. Solver runtimes

When we compare the TT-solve runtime to CG and GE solvers in Figure 4.11, we use their runtime estimates from Sections 2.2.1 and 2.2.2.

For GE, these estimates consider constant factors and lower order terms, giving a very accurate result. However, in case of banded matrices, one can use other GE variants that run more efficiently. In particular, for a matrix with bandwidth of w and size n, we get a runtime of $\mathcal{O}(w^2 n)$ [17], which is much faster than $\mathcal{O}(n^3)$ when $w \ll n$. While some of our test matrices have the banded structure, not all of them do. Therefore, to be able to present consistent results, we used the generally applicable GE time complexity estimates.

In the case of CG, the runtime estimate is known to be an upper bound [25], meaning runtimes can be faster than this. This is due to the iterative nature of the algorithm. On the other hand, constant factors and terms are not given for this estimate. More importantly, when attempting to utilise CG for ill-conditioned matrices, one usually applies a preconditioner to improve the convergence properties of the system. While this incurs additional compute cost, it can lead to significant runtime savings overall [27]. In our analysis, we considered the vanilla CG, without any preconditioning. This choice was made because choosing an effective preconditioner technique is highly problem dependent and its effectiveness varies from one matrix to the other greatly [27].

Lastly, we would like to point out a limitation related to the TT-solve runtime examination. As customary when using the big-$\mathcal{O}$ approach, we have considered the worst case scenarios. While this aided the relatively straightforward assessment, in certain cases, this was an unnecessarily pessimistic premise. In particular, when considering the TT-ranks and mode size of each TT-core, currently, we take the maximal value of these across all TT-cores. However, in most TT-cores, the mode sizes will likely be lower. Additionally, most of the TT-ranks can be cheaply reduced without loss of accuracy, using the parallel-rounding algorithm [10, 102].

5.2. Key takeaways

After a throughout understanding of our limitations, discussed in the previous section, we are ready to recount the most important outcomes of this thesis.

5.2.1. Solving TTM-decomposition problems caused by poor matrix size factors

The proposed methods - padding and Partial Gaussian Elimination (PG) - were developed to tackle the challenges associated with Tensor Train (TT) decomposition, particularly when dealing with matrices that have poorly factored sizes. The effectiveness of these methods was evaluated across various matrices, and the results offer valuable insights into their impact on achievable TT-decompositions and therefore, attainable speedup gains when solving linear systems of equations (LSEs) in the TT-format.

The padding strategy, which involves extending the matrix dimensions to make them more amenable to factorization, proved highly effective in cases where the original matrix size had limited factorization options. The results demonstrated that padding can significantly enhance the TT-decomposition process by increasing the number of possible tile sizes, leading to better decomposition outcomes. For example, in the case of the *ex3* matrix, the original size $n = 1821$ had only three possible tile sizes due to its prime factorization. By applying three levels of padding, the matrix size increased to $n' = 1824$, which has a much richer factorization. This not only provided more options for tile sizes but also resulted in a substantial speedup in runtime - approximately 65.66×10^6 times faster compared to the unpadded case. This clearly indicates that padding can dramatically reduce computational time, particularly for matrices with limited initial factorization options. However, the effectiveness of padding is not universal. For the *ex10hs* matrix, which already had a well-factored size, additional padding did not lead to significant performance improvements. This highlights a key limitation: padding is most beneficial for matrices with poor initial factorizations, but its impact diminishes when the original matrix size already offers ample tile size options.

The PG strategy works by a similar principle to padding, however, it alters the nonzero entry structure more significantly, by partially solving the system. The results indicate that PG is also a practical way to allow for better factorizations. The analysis revealed that PG's effectiveness varies depending on the matrix structure. For banded matrices, PG resulted in a relatively modest increase in nonzero entries - no more than 1.6 to 1.7 times the original count - even when a large percentage of variables were eliminated. This contrasts with non-banded matrices, like *bcsstk13*, where the nonzero entry count increased significantly, even with AMD variable ordering, making the decomposition process more challenging. Despite this, PG proved invaluable in enhancing the performance of the *matrix2mpo* algorithm. This is because its largest effect comes from enabling better integer factorizations of the matrix size, which requires only reducing a handful of variables.

Overall, our proposed strategies address this research gap. As we have seen the results in Figure 4.11, we can bring down the time complexity of TT-solve for badly factorized matrices to the same level as well-factorized ones.

5.2.2. Optimizing the tradeoff between maximum mode size and TTM-rank

The balance between the maximum mode size and the TTM-rank plays a critical role in determining the efficiency of TT-computations. Optimizing this tradeoff is essential for ensuring that the computational resources are utilized effectively. In the case of solving LSEs in the TT-format, our extensive theoretical time complexity analysis of TT-solve, provides the framework needed to find the best pairs of these variables.

Using the largest prime factor as tile size has been shown to be an effective heuristic to minimize the runtime of the TT-solve approach. However, based on our insights in Figure 4.11, choosing other factors for the *matrix2mpo* algorithm can sometimes lead to better results. Additionally, when for instance iterative solvers are used for the local problems in TT-MALS, the balance between r and I is largely altered. This motivates the need for evaluating alternative tile size choices, besides the largest prime factor.

5.2.3. Altering the matrix for better TT-decomposition

In the process of TT-decomposition, the structure of a (sparse) matrix can significantly influence the attainable ranks. Altering the matrix through techniques such as reordering variables or modifying the matrix in others ways, can lead to better arrangement of the matrix values, allowing lower-rank factorizations. In particular, for sparse matrices, altering the positioning of nonzero elements can lead to better decompositions via the *matrix2mpo* approach [10].

We have introduced the metric *h* in Eq. (3.14), to measure the ratio of nonzero elements within the nonzero blocks of *matrix2mpo.* For starters, this metric helps with identifying cases, when alternative matrix size factorizations - enabled by padding or PG - can be beneficial. In our experience, such matrices have really low *h*-values, less than 5-10%. Also, as we have seen in Figure 4.10, it suggests that there is much room for further optimization by reorganizing the matrix to better cluster nonzero elements.

When contrasting the findings in [10] and our test matrices, the effectiveness of the reverse Cuthill-McKee (RCM) strategy to concentrate the nonzero entries and thus reducing TTM-ranks is largely matrix dependent. Nevertheless, the prospect of variable ordering algorithms should not be dismissed. As we have highlighted, there is potential for enhancement by better positioning and concentration of nonzero entries.

Besides variable ordering, we have also turned to the PG strategy to alter the matrix. When disregarding its positive effects enabled by better integer factorizations of the matrix size, we saw its relatively limited impact on reducing the time complexity of TT-solve (Figure 4.5). Thus, while PG does reduce the matrix size, and the count of remaining variables to be solved, the TT-decomposition of the residual submatrix has not shown large reductions in attainable TTM-ranks. Linked to this, we experimented with utilizing AMD to reduce fill-in while running PG. Our results reveal that AMD can perform poorly in certain cases, sometimes even worsening the situation by increasing fill-in, especially in the submatrix that is subject to the TT-decomposition (see Figure 4.4). This outcome is particularly pronounced in non-banded matrices, where the structure is less predictable, and the reordering can inadvertently lead to a more dispersed distribution of nonzero elements, complicating the decomposition. The underperformance of AMD in reducing fill-in for non-banded matrices highlights the need to explore alternative techniques. In domains where PG is of interest, particularly those that are non-banded, relying on AMD might not be the best approach. Instead, it would be beneficial to investigate other reordering strategies in the future.

Given the limitations of the tried methods to alter the entry patterns in the matrices, there is a clear need to explore alternative techniques that could more effectively reorganize the matrix for TT-decomposition. These alternatives might include heuristic approaches that can dynamically adapt to the structure of the matrix, potentially offering better nonzero concentration without the drawbacks of increased fill-in. For example, heuristic-based reordering methods could be developed to prioritize the clustering of nonzeros within tiles, even if it means sacrificing some other properties like minimal bandwidth. Most likely, exact algorithms for this purpose are too expensive to compute, however, they can provide insights for designing these heuristics. Understanding the underlying principles of successful reordering strategies might lead to the development of more robust methods that can consistently improve TT-decomposition across a wider range of matrix types.

5.2.4. Comparing theoretical runtimes of TT-solve, CG and Gauss

We have contrasted the theoretical runtime of vanilla CG, Gaussian elimination and TT-MALS under some assumptions for sparse matrices. When considering TT-solve, we also included the time needed to convert the LSE to TT-format and the time necessary to convert the solution back to a vector from its TT-representation. As shown in Figure 4.11 our improvements make the TT-solve approach in several cases comparable to CG and Gaussian elimination. Of course, this is only possible under the assumptions that we have discussed previously. Notably, to make the TT-solve approach faster than existing solvers, additional enhancements will be necessary for the type of matrices we have worked with in this study.

5.2.5. Testing TT-solve applicability via runtime estimates

Our work shows a way to efficiently assess sparse LSEs' suitability for the TT-solve approach. While theoretical runtime estimates have their limitations, their computation can be done quickly, allowing such assessments. For example, one can easily evaluate the expected TT-solve runtime of any LSE using the time complexity derivations from this study. Additionally, checking a few, i.e. $\mathcal{O}(1)$ levels of padding, the corresponding factorizations and tile sizes give a cheap but good estimate on the expected speedup ratio achievable by our improvements. These quick checks are enabled by running the *modeSize2rank* method which has a time complexity of $\mathcal{O}(z)$ as shown in Appendix B.1.2.

5.2.6. Usability of TT-solve in practice

While our improvements show the promise of TT-solve for sparse, ill-conditioned, symmetric, positive definite linear systems, several challenges and limitations remain as we have discussed. Therefore, until these are addressed, we cannot recommend TT-solve to replace existing solvers.

5.3. Future research directions

In this section, we elaborate on related research ideas that we deem worth investigating based on this inquiry. Additionally, we encourage future explorations to address the limitations that we have raised previously.

5.3.1. Iterative solvers for TT-subproblems

The TT-solve method needs to solve smaller subproblems of LSEs which are the bottleneck of the computation. In this study, we have assumed that these subproblems are solved by GE. This choice was made because of the better runtime predictability GE as opposed to using iterative solvers whose convergence highly depends on the problem at hand [27]. It is possible to replace GE by an iterative solver like GMRES here, which is expected to have a better runtime as seen in Appendix B.1.3.

It would be worth investigating the properties of the linear systems in these subproblems. In turn, knowing which iterative solver with what configuration is suited for different scenarios could lead to significant improvements. Adding to this, identifying appropriate preconditioners and assessing the corresponding runtime enhancements would be valuable.

5.3.2. Altering the nonzero entry structure

Exploring alternative ways to reorganize the locations of the nonzero entries presents a promising avenue for future research. We have seen the limited but positive effect of RCM on a few of our test matrices which aims to reduce the bandwidth. Also, in [10], RCM was able to cut the maximal rank, nearly by half for the AMD-G2-circuit matrix from [69] for various tile sizes. RCM is not the only algorithm for this task [103, 104, 105, 106], therefore we recommend future research to evaluate alternative variable ordering approaches. Adding to this, conceptually, an ideal reorganization of nonzero entries does not have to concentrate them around the diagonal. Instead, gathering them in dense blocks around the matrix would be already helpful. Designing variable ordering algorithms with this aim, remains largely unexplored to the best of our knowledge.

Another improvement we have started to explore was AMD, designed to minimize fill-in when doing Gaussian elimination. As in the case of RCM, there are various alternative algorithms [86, 107] that future research could focus on in this context. We note that these algorithms are developed for minimizing overall fill-in when reducing the complete system, i.e. computing the LU or Cholesky decomposition. Therefore, variable ordering algorithms specifically designed to minimize fill-in within submatrices remaining during partial Gaussian elimination could be beneficial.

Often, sparse linear systems arise from discretizing PDEs. During this process, there are several choices to be made, which in turn influence the final positioning of the nonzero entries. For instance, one has control over whether to use FDM or FVM for discretization. Another choice is how the spatial grid is constructed [91]. Therefore, it could be intriguing to see how these choices effect the effectiveness of TT-decomposition via *matrix2mpo*.

5.3.3. Applications beyond TT-solve: Extremal Eigenvalue problems and more

As we have seen in Section 2.6, the TT-MALS solution method for linear systems of equations comes down to the following. At its core, an alternating least squares type of optimization is applied to a tensor-train-valued objective function. This means that one can replace this objective function with differently constructed tensor trains that correspond to different problems. There are various applications discussed in [60] that could potentially make us of our TT-decomposition optimization, assuming the data is sparse. Here, we highlight two. Firstly, in large scale, structured tensor data when a large proportion of the entries are missing or noisy, we can attempt to reconstruct the problematic entries [60, 108, 109]. Another one is when extremal (minimal or maximal) Eigenvalues and the corresponding Eigenvectors need to be computed for large matrices ($n > 10^{15}$). In such cases, a TT-based approach can help [60, 110, 111, 112]. We suspect that the techniques proposed by us to improve the TT-decomposition, could help with reducing the runtime in these settings as well.

6 Conclusion

In this thesis, we have delved into the optimization of Tensor Train (TT) decompositions for sparse linear systems of equations (LSEs), with a focus on improving TT-solve runtime efficiency. Our research has sought to address key challenges associated with TT-decomposition, including poor matrix size factorizations and optimizing the tradeoff between the maximal mode size and TTM-rank. Additionally, we have looked into the possibility of altering the coefficient matrix by variable ordering and partial Gaussian elimination (PG) to get TTM-decompositions that reduce the TT-solve runtime. The practical motivation behind this work has been to enhance the speed and scalability of solving sparse, large-scale LSEs, which is crucial for various applications across science and engineering. Our investigation has yielded several noteworthy findings and contributions to the field.

6.1. Key findings

The key findings from this thesis can be summarized as follows:

1. The proposed methods, including padding and PG, were effective in addressing TT-decomposition challenges related to matrices with poorly factored sizes. However, their effectiveness in runtime reductions diminished for well-factored matrices.

2. The study emphasized the importance of balancing maximum mode size and TTM-rank for efficient TT computations. While using the largest prime factor as a tile size was generally effective, alternative factors sometimes offered better results.

3. Matrix structure significantly influences TT-decomposition, and altering it through variable reordering or other modifications can lead to lower rank factorizations. Techniques like RCM were effective in some cases but highly matrix-dependent. PG had limited success in reducing time complexity when disregarding its effect due to enabling new matrix size factorizations. Reordering methods like AMD sometimes worsened decomposition.

4. The theoretical runtime analysis showed that the improved TT-solve approach could be comparable to CG and GE under certain assumptions, though further enhancements are needed to gain advantage over traditional solvers for the studied matrices.

5. The study provided a method for quickly assessing the suitability of TT-solve for sparse LSEs through runtime estimates. These estimates, combined with minimal padding checks, offer a practical way to gauge potential speedup gains.

6. Despite the promising improvements, the thesis acknowledges that TT-solve has limitations and cannot yet replace existing solvers for solving sparse, ill-conditioned, symmetric, positive definite linear systems.

6.2. Limitations

This study presents several limitations that may affect the generalizability of our findings. First, the runtime assessment methodology primarily relies on theoretical analysis, using operation counts and big-$\mathcal{O}$ notation. While this approach offers insights into scalability, it overlooks practical factors such as hardware optimizations and constant factors, potentially skewing results for smaller matrices.

Our test dataset is limited to a small number of symmetric, positive definite matrices from computational fluid dynamics, restricting the generalizability of our conclusions. On the other hand, our matrices lack special structures like Toeplitz or Hankel, which are known to exhibit low numerical ranks.

Assumptions made in the TT-solve runtime analysis, such as low TT-rank and rapid convergence of TT-MALS, may not hold in all cases. We focused primarily on matrix decomposition improvements, neglecting the potential impact on TT-ranks of the right-hand side or solution vectors.

Finally, our comparative analysis of TT-solve, CG and GE runtimes is based on theoretical estimates, which may not fully capture the complexities of real-world applications, especially regarding preconditioners and different matrix structures.

6.3. Future work

Future research should address these limitations by expanding the matrix dataset to include larger, more varied matrices and by incorporating statistical methods to better isolate the effects of individual factors. Investigating the impact of our strategies on TT-ranks of right-hand side and solution vectors could provide a more comprehensive understanding of their efficacy. Further exploration into using iterative solvers for TT-subproblems, such as GMRES, could lead to significant runtime improvements. Additionally, alternative methods for reorganizing nonzero matrix entries, beyond the RCM and AMD algorithms used here, may further optimize TT-decomposition and reduce runtimes. Finally, examining how discretization choices in PDEs influence the effectiveness of TT-decomposition could open new avenues for optimizing TT-solve for specific applications.

6.4. Final thoughts

In summary, this thesis has made significant strides in optimizing TT-decompositions for solving sparse linear systems of equations. The proposed methods such as padding, PG, RCM and tile size optimization have demonstrated substantial potential for improving TT-solve efficiency. These advancements are crucial for addressing the practical challenges of solving large-scale LSEs, which have broad applications across various scientific and engineering domains.

The insights gained from this research not only enhance the theoretical understanding of TT-decomposition but also provide practical solutions that can be applied to real-world problems. As we continue to refine these techniques and explore new avenues of research, we can further improve the efficiency and scalability of LSE solvers, ultimately contributing to more effective and timely solutions in complex computational fields.

A

Hydrodynamic simulations

This appendix chapter explores the field of hydrodynamic simulations, as it has served as the original motivation for this thesis.

Hydrodynamic simulations play a crucial role in understanding and predicting the behavior of water flow in various natural and engineered systems. These simulations involve the application of mathematical models to simulate the movement of water, taking into account factors such as fluid dynamics, topography, and boundary conditions. In this section, we will delve into the fundamental concepts underlying hydrodynamic simulations. Furthermore, we will discuss one of the widely applied methods used to solve these equations over time and space. In particular, we will touch upon the finite difference and finite volume methods, which ultimately lead to the formulation of large sparse linear systems of equations.

Shallow Water Equations (SWE) and PDE Solving The shallow water equations are a set of hyperbolic partial differential equations that describe the flow of shallow water over varying topography. These equations are derived from the conservation principles of mass and momentum and are widely used in hydrodynamic modeling due to their ability to accurately capture the behavior of water flows in many practical scenarios [113, 114]. In such settings, analytical solutions are lacking for the SWE, therefore they are solved numerically [114]. This requires discretizing them over both time and space.

Discretization over space and time Discretization is the process of approximating the continuous equations using a discrete set of values. In the context of hydrodynamic simulations, this can be done in various ways [114]. The general idea is to divide the spatial domain into a grid and discretize the equations over both time steps and grid cells [91, 114].

1D/2D grid modeling approach Hydrodynamic simulations often adopt a grid-based approach to represent the spatial domain [91]. In one dimension (1D), the domain is discretized along a single axis. This is often used for modelling the topology of the (underground) sewer system and open channels with clear flow paths [115, 116, 117]. On the other hand, discretizing space over two dimensions (2D), allows for better modelling of the flow over flat surfaces, also it makes the model setup in practice much easier. The downside here is the increased computational load because of the higher dimensionality [116, 117, 118]. In current practice, the two approaches are combined to model different parts of drainage systems to get the best of both worlds [91]. See [116] for a detailed overview of this in the context of urban pluvial flood modelling.

Finite Difference and Finite Volume approach Besides choosing the dimensionality of the spatial discretization, different approaches exist to determine how the flow is computed from one grid point to the other. In the case of the shallow water equations, the finite difference and finite volume methods are the most commonly used [119].

Finite Difference Method (FDM): In this approach, derivatives in the partial differential equations are approximated using finite differences. The equations are discretized on a grid, and finite difference approximations are applied to compute the spatial and temporal derivatives. This results in a system of algebraic equations that can be solved iteratively to obtain the solution at each time step [120].

Finite Volume Method (FVM): The finite volume method focuses on the conservation of mass and momentum within control volumes or cells of the grid. The equations are integrated over each cell, leading to a set of discrete equations that represent the conservation laws. These equations are then solved iteratively to obtain the solution over the entire domain [121].

Large sparse linear systems of equations The discretization of the shallow water equations using finite difference or finite volume methods leads to the formation of large sparse linear systems of equations. These systems arise due to the discretization of the spatial domain and can be solved using iterative methods such as the Gauss-Seidel method, the conjugate gradient method, or other sparse matrix solvers [91].

B

Time complexity derivations

B.1. TT-algorithms

In this section we give the details of the time complexity derivation of relevant TT-algorithms.

Let us start by introducing some notation. Suppose $\mathbf{b} \in \mathbb{R}^m$. Then this vector can be reshaped into a tensor $\underline{\mathbf{B}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ which has order d with $I_1 I_2 \cdots I_d = m$. Additionally, let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix. Then, we can reshape it as a $2d$th-order tensor, $\underline{\mathbf{A}} \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \ldots I_d \times J_d}$, where $m = I_1 I_2 \cdots I_d$ and $n = J_1 J_2 \cdots J_d$. Generally, unless stated otherwise, we only look at the case where $n = m$.

Now, consider the TT-decomposition of $\underline{\mathbf{A}}$ as a TT matrix (TTM) with d TT-cores. Then, its kth core is a 4-dimensional tensor, $\underline{\mathbf{A}}^{(k)} \in \mathbb{R}^{r_k \times i_k \times j_k \times r_{k+1}}$. The TT-cores in the TT-decomposition of $\mathbf{b}$ are given as $\underline{\mathbf{B}}^{(k)} \in \mathbb{R}^{s_k \times i_k \times s_{k+1}}$ for $k \in \{1, 2, ..., d\}$. Analogously, the TT-cores of a tensorized vector $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ in the TT-format are given as $\underline{\mathbf{X}}^{(k)} \in \mathbb{R}^{t_k \times j_k \times t_{k+1}}$ for $k \in \{1, 2, ..., d\}$. Let $I = \max_k i_k$, $J = \max_k j_k$, $r = \max_k r_k$, $s = \max_k s_k$ and $t = \max_k t_k$, where $k \in K = \{1, 2, ..., d\}$ for i_k and j_k. And $k \in K \cup \{d+1\}$ for s_k, t_k and r_k. Note that under the assumption $n = m$, we also have that $n = \prod_{k=1}^{d} i_k = \prod_{k=1}^{d} j_k$. When it is clear from context, which TT(M) we refer to, we just use r to denote the maximal TT-rank. Additionally, mostly, we do not distinguish between the maximal rank of $\underline{\mathbf{X}}$ and $\underline{\mathbf{B}}$, thus we just denote $s = \max(s, t)$.

B.1.1. Rounding algorithms

TT-round

The time complexity of the TT-round is $\mathcal{O}(dIr^3)$ for TT [4]. In case of an TTM, however, this is higher. The QR and SVD passes through the TT-cores in this case operate on $I^2 r \times r$ matrices, instead of $Ir \times r$ matrices. Therefore the time complexity becomes $\mathcal{O}(dI^2 r^3)$. This is in line with the TT-round implementation in [65].

B.1.2. TT-decomposition

TT-SVD

Examining Algorithm 1 in [4], we can determine the runtime of TT-SVD both for the TT and TTM cases. We assume that the ranks have to be determined on the go, that is, only the error tolerance ϵ is known in advance.

Let us first consider the TT case, for simplicity we have $I_1 = I_2 = \cdots = I_d = I \geq 2$. The most expensive part of the algorithm is clearly the successive SVD computations on the input tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ that is reshaped. When constructing the first TT-core, the SVD algorithm is applied to a $rI_1 \times I_2 I_3 \cdots I_d$ sized matrix ($r = 1$ for the first TT-core). Therefore the time complexity of this step is $\mathcal{O}(I_2 I_3 \cdots I_d \cdot I_1^2 r^2 + I_1^3 r^3) = \mathcal{O}(nI_1 r^3) = \mathcal{O}(I^{d+1} r^3)$. When computing the kth TT-core, SVD operates on a matrix of size $rI_k \times I_{k+1} \cdots I_d$. Therefore, the time complexity for the kth step is $\mathcal{O}(I_{k+1} \cdots I_d \cdot r^2 + I_k^3 r^3) = \mathcal{O}(I^{d-k} r^3)$. The last computation is when $k = d-1$. At this step we operate on a matrix of size $rI_{d-1} \times I_d$, yielding a time complexity of $\mathcal{O}(rI_{d-1} I_d^2 + I_d^3) = \mathcal{O}(rI^3)$. While in general, it does not have to hold, in this manuscript, we assume that $r \leq I$ for our theoretical time complexity assessments. Therefore, at each TT-core, the SVD computation will have a r^3 term, except the last one. To make the remaining analysis easier, let us bound the last SVD computation by $\mathcal{O}(r^3 I^3)$ yielding a small overestimation. Adding this up for $k = 1, \dots, d-1$, we get $\mathcal{O}(r^3(I^{d+1} + \dots + I^{d-k} + \dots + I^3))$. Now if we let S represent this sum, we have:

$$
\begin{aligned}
S &= I^3 + \cdots I^{d+1} && && \text{(B.1)}\\
IS &= I^4 + \cdots I^{d+2} && \text{(multiply by } I\text{)} && \text{(B.2)}\\
IS + I^3 &= I^3 + I^4 + \cdots I^{d+2} && \text{(add } I^3\text{)} && \text{(B.3)}\\
IS + I^3 &= S + I^{d+2} && \text{(use definition of } S\text{)} && \text{(B.4)}\\
IS - S &= I^{d+2} - I^3 && \text{(reorder terms)} && \text{(B.5)}\\
S(I-1) &= I^{d+2} - I^3 && && \text{(B.6)}\\
S &= \frac{I^{d+2} - I^3}{I-1} && && \text{(B.7)}\\
S &= \frac{I^{d+2} - I^3}{I} && \text{(ignore } -1 \text{ in big-}\mathcal{O}\text{)} && \text{(B.8)}\\
S &= I^{d+1} - I^2 && && \text{(B.9)}
\end{aligned}
$$

Therefore, we have that the SVD computations in TT-SVD cost $\mathcal{O}(r^3(I^{d+1} - I^2)) = \mathcal{O}(r^3 I^{d+1}) = \mathcal{O}(r^3 In)$, assuming each mode has more or less the same size and that $r \leq I$.

Additional costs include multiplying the diagonal matrix of singular values with the matrix V^T after each SVD computation. V has a size of $I^{k+1} \cdots I^{d-1} \times r$ after truncation for the kth iteration of the loop. The multiplication only needs to happen with the top r singular values, therefore this operation costs $\mathcal{O}(I^{d-k} r)$ per iteration. Taking the sum for $k = 1, \dots, d-1$, we have $\mathcal{O}(r(I + I^2 + \dots + I^{d-1}))$ which can be shown to be approximately equal to $\mathcal{O}(rn)$, using the same logic that we used for the SVD computations time.

The initial Frobenius norm to determine the truncation parameter, can be computed by vectorizing the input tensor (of size $n \times 1 = I^d \times 1$), taking the inner product with itself (n multiplications, $n-1$ additions), followed by computing the square root (1 flop) of the resulting scalar. Therefore this computation takes $\mathcal{O}(I^d) = \mathcal{O}(n)$ time. Overall, for the TT case, TT-SVD takes $\mathcal{O}(r^3 In + rn + n) = \mathcal{O}(r^3 In)$ time.

When forming a TTM using TT-SVD, the input tensor has a size of $I^d \times I^d = n \times n$. We can treat this as a TT where each mode has size I^2 instead of I, thus the TT-SVD time complexity becomes $\mathcal{O}(r^3 I^2 n^2 + rn^2 + n^2) = \mathcal{O}(r^3 I^2 n^2)$ for TTMs.

modeSize2rank

We term the first part of the matrix2mpo method (Algorithm 4.1) in [10] as *modeSize2rank*. This is because it determines the TTM-rank of the decomposition which can be constructed from a given sparse matrix and a chosen tile size. Let us examine its runtime in this section.

Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ with z nonzero entries and let k be the chosen tile size s.t. $n \mod k \equiv 0$. Additionally, we denote the number of tiles needed to cover n in one dimension as b. Thus, we have $n = k \cdot b$, i.e. assuming k equals the maximal mode size I, we have $b = \frac{n}{I}$.

The algorithm has a *for loop* over each column block which takes $\mathcal{O}(b)$. Inside each loop, firstly, all rows with a nonzero entry are identified. Considering the current implementation with dense representation, this can take $\mathcal{O}(nk)$ time when we need to check each entry in the block-column. The number of rows in the column is n and the width of the column is k. This is followed by $\mathcal{O}(1)$ manipulations on the indices of these nonzero row indices, which can be bounded by $\mathcal{O}(n)$. Therefore, the overall runtime of this method is $\mathcal{O}(bnk) = \mathcal{O}(\frac{nnI}{I}) = \mathcal{O}(n^2)$ using the current implementation.

We can optimize this further for sparse inputs. Let us assume that the sparse matrix is in Coordinate (COO) format for ease of interpretation. That is, each nonzero entry is described by a triplet: its row index, its column index and its value. Let us denote this by (x, y, v) We can determine the nonzero submatrices by a single pass through the nonzero entries in $\mathcal{O}(z)$ time as follows. We need to assign the block coordinates (x', y') to each triplet. We can compute these by diving the row and column indices, (x, y) by the tile size b, followed by rounding down (zero-indexing assumed). To compute the TTM-rank, we need to determine the number of unique blocks where nonzero entries are located. One way to achieve this is to have one more pass over the nonzero entries and count the unique block coordinate pairs. We can conclude that the *modeSize2rank* algorithm can be implemented to run in $\mathcal{O}(z)$ time for sparse matrices.

matrix2mpo

After the *modeSize2rank* part, the second segment is about initializing the TT. This can take at most the time needed to pass through every entry in the TT. We note that sparse representations are not used for the TT-representation, as most TT-operations would break the sparsity anyways. Therefore the size of the TTM corresponds to the runtime of this part, which is $\mathcal{O}(dI^2r^2)$.

Using the dense representation, the third part contains a nested for loop which iterates over each tile that has a nonzero entry, this takes $\mathcal{O}(r)$ as the number of nonzero blocks corresponds to the resulting TTM-rank. At each nonzero block, the algorithm needs to determine the corresponding nonzero entries in the Kronecker product matrices. The *lin2ten* subroutine and going through each TT-core when assigning the indices, both require $\mathcal{O}(d)$ time. Therefore, overall this third segment needs $\mathcal{O}(rd)$ time. Lastly each TT-core is reshaped, which requires $\mathcal{O}(d)$ time.

When the first part is done by the sparse approach, the third procedure described above needs some adjustments. For the first TT-core, we just need to assign the nonzero entries into the correct locations which can be done in $\mathcal{O}(z)$ time. For each remaining TT-core, to determine the nonzero entries, we need to convert the nonzero block indices into tensor indices via the *lin2ten* subroutine. This procedure takes $\mathcal{O}(rd)$ similarly to the dense case.

We conclude that the overall runtime of *matrix2mpo* can be bounded by $\mathcal{O}(n^2 + dI^2r^2)$ for the dense implementation, while it takes $\mathcal{O}(z + dI^2r^2)$ for the sparse one.

B.1.3. TT-solve

TT-ALS

Preparing each micro-iteration per sweep takes $\mathcal{O}(s^3r^2I^2d)$. Additionally, some in between contractions take $\mathcal{O}(rs^3I^2d)$. Solving the micro LSE arising at each TT-core needs $\mathcal{O}(s^3r^2I^2d)$ when using iterative solvers and $\mathcal{O}(s^6I^3d)$ for direct solvers like Gaussian elimination. The QR factorizations need $\mathcal{O}(s^3Id)$ time per sweep [6].

TT-MALS

Similarly to ALS, preparing each micro-iteration per sweep takes $\mathcal{O}(s^3r^2I^2d)$. On the other hand, the in between contractions take $\mathcal{O}(s^3rI^3d)$ for this approach. Solving the micro LSE arising at each TT-core needs $\mathcal{O}(s^3r^2I^3d)$ when using iterative solvers and $\mathcal{O}(s^6I^6d)$ for direct solvers like Gaussian elimination. For MALS, instead of QR, we need SVD factorizations, that need $\mathcal{O}(s^3I^3d)$ time per sweep [6].

Let us denote the number of half sweeps needed to converge as c. Then, the overall runtime estimate can be approximated as $\mathcal{O}\big(c \cdot d \cdot (s^3r^2I^2 + s^3rI^3 + s^6I^6 + s^3I^3)\big) = \mathcal{O}\big(c \cdot d \cdot (s^3r^2I^2 + s^3rI^3 + s^6I^6)\big)$ when using a direct solver for local problems.

As a side note, we give the runtime estimate when an iterative solver like GMRES is used in the local problems. Here the $\mathcal{O}(s^6I^6d)$ term is replaced by $(s^3r^2I^3d)$ according to [6]. Thus we have $\mathcal{O}\big(c \cdot d \cdot (s^3r^2I^2 + s^3rI^3 + s^3r^2I^3 + s^3I^3)\big) = \mathcal{O}\big(cds^3r^2I^3\big)$. In what comes next, we assume the use of the direct solver approach for local problems.

To get a more interpretable runtime estimate, let us assume that $c = s = r = I = \mathcal{O}(\log(n))$. Additionally, let us use that $d \approx \log(n)$. Then, we have $\mathcal{O}(\log(n)^{14})$.

Suppose, we have our TTM-decomposition, therefore we know the values I and r. Then, we only need to make assumptions about c and s to get a good estimate. First, let us consider the case when

$c = s = \mathcal{O}(1)$. Then, the overall runtime can be approximated as $\mathcal{O}\Big(\log(n) \cdot (r^2I^2 + rI^3 + I^6)\Big)$. Another case is when we have $c = s = \mathcal{O}(\log(n))$. Then, we have $\mathcal{O}\big(\log(n)^5 \cdot (r^2I^2 + rI^3 + \log(n)^3 I^6)\big)$.

B.1.4. TT to Vector conversion

In order to reconstruct $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ from $\underline{\mathbf{X}}^{(1)} \times^1 \underline{\mathbf{X}}^{(2)} \times^1 \cdots \times^1 \underline{\mathbf{X}}^{(d)}$, where $\underline{\mathbf{X}}^{(k)} \in \mathbb{R}^{s \times I_k \times s}$, we have perform $d-1$ contractions between the TT-cores. This can be done in $\mathcal{O}(I^2s^4 + I^3s^4 + \cdots + I^{d-1}s^4 + I^ds^2) = \mathcal{O}(dI^{d-1}s^4 + I^ds^2) = \mathcal{O}(dns^4)$ time [67].

B.2. GMRES(k)

Let us investigate the flop count of the the GMRES(k) method, where k is the number of iterations before the algorithm is restarted. According to the original paper [24], its time complexity for a matrix of size $n \times n$ is composed of the following parts. Running GMRES for k steps requires $k(k+2)n + kz = k^2n + 2kn + kz$ multiplications. Out of this, kz comes from k matrix-vector multiplications. In practice k is chosen so that $k \ll n$ to keep the runtime feasible. If this assumption holds, then the above gives a good runtime approximation [24]. Assuming, we need $\mathcal{O}(1)$ restarts, we have the following overall time complexity in terms of flops. For the dense case, $z = n^2$ thus we have a runtime of $\mathcal{O}(k^2n + kn^2)$. However, in the sparse case, when $z \in \mathcal{O}(n)$ (where z is the number of nonzero entries), this reduces to $\mathcal{O}(k^2n + kn) = \mathcal{O}(k^2n)$ time.

C

Definitions, Theorems and Proofs

Let $x, y \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$, then, we have the following definitions and results.

Definition C.1 (Symmetric matrix)**.** A is symmetric if and only if it is equal to its transpose. Formally, $A = A^T$.

Definition C.2 (Orthogonal vectors)**.** x and y are orthogonal if $x \cdot y = 0$. We say that a set of vectors $\{x_1, x_2, ..., x_n\}$ are (mutually) orthogonal if every pair of vectors is orthogonal, i.e. $x_i \cdot x_j = 0, \forall i \neq j$.

Definition C.3 (Orthonormal vectors)**.** We say that a set of vectors $\{x_1, x_2, ..., x_n\}$ are (mutually) orthonormal if every pair of vectors is orthogonal and each vector is a unit vector, i.e. $\|x\| = 1$, alternatively $x = \frac{x}{\|x\|}$.

Definition C.4 (Orthogonal matrix)**.** A square matrix A is orthogonal if $A^{-1} = A^T$. Equivalently, A is orthogonal if $AA^T = A^TA = I$, where I is the identity matrix.

Definition C.5 (Positive definite matrix)**.** A is said to be positive definite if and only if $x^TAx > 0, \forall x \in \mathbb{R}^n \backslash \{0\}$. Alternatively, it can be defined in terms of its eigenvalues; A is positive definite if and only if all of its eigenvalues are positive.

Definition C.6 (Positive semi-definite matrix)**.** A is said to be positive semi-definite if and only if $x^TAx \geq 0, \forall x \in \mathbb{R}^n$. Alternatively, A is positive semi-definite if and only if all of its eigenvalues are non-negative.

Definition C.7 (Diagonally dominant matrix)**.** A is diagonally dominant if $|a_{ii}| \geq \sum_{i\neq j} |a_{ij}|, \forall i \in \{1, 2, ..., n\}$, where a_{ij} denotes the entry in row i and column j of matrix A. Note that this is sometimes referred to as *weak diagonal dominance*. When strict inequality is used, then we talk about *strict diagonal dominance*. In this manuscript, when we say diagonal dominance, we mean weak diagonal dominance.

Definition C.8 (Banded matrix)**.** A banded matrix is a sparse, not necessarily square matrix, where non-zero entries are limited to the main diagonal and potentially extend to zero or more diagonals on either side. For this definition, let $A \in \mathbb{R}^{m\times n}$. Then, we say that A has *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j + p$ and *upper bandwidth* q if $a_{ij} = 0$ whenever $j > i + q$ [122]. The *bandwidth* of the matrix is the maximum of p and q, i.e. it is the number w s.t. $a_{ij} = 0$ if $|i - j| > w$ [123].

Theorem C.1. A (real) symmetric, weakly diagonally dominant matrix with non-negative diagonal entries is (symmetric) positive semi-definite (SPSD).

Theorem C.2. A (real) symmetric, strongly diagonally dominant matrix with positive diagonal entries is (symmetric) positive definite (SPD).

Definition C.9 (Condition number)**.** Let A be a square matrix, then its condition number $\kappa(A)$ is defined as: $\kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \frac{\sigma_{max}}{\sigma_{min}}$, where σ_{max} and σ_{min} are the maximal and minimal singular values, respectively. If A is singular, then $\kappa(A) = \infty$. When A is symmetric, we have that $\kappa(A) = \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|}$, where λ_{max} and λ_{min} are the largest and smallest eigenvalues of A, respectively [124, 125].

Definition C.10 (Sparsity of a matrix)**.** Let A be a $n \times m$ matrix with z nonzero entries. Then its sparsity is given by: $1 - \frac{\text{number of nonzero entries in } A}{\text{total number of entries in } A} = 1 - \frac{z}{n \cdot m}$.

Lemma C.3 (Gershgorin circle theorem)**.** The eigenvalues of $A \in \mathbb{C}^{n \times n}$ lie in the union of the n discs in the complex plane $D_i = \left\{ z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{j \neq i} |a_{ij}| \right\}, i \in \{1, ..., n\}$.

Lemma C.4. A Hermitian matrix has real eigenvalues.

Proof. As described in [126], this can be proven as follows. Let A be a complex matrix such that $A^* = A$. Let λ be one of A's eigenvalues and $u \neq 0$ a corresponding eigenvector. We know, by the definition of conjugate transpose, that $Au = \lambda u \implies u^* A^* = \lambda^* u^*$. Hence

$$\lambda^* u^* u = u^* A^* u = u^* A u = \lambda u^* u$$

and therefore $\lambda^* = \lambda$. Therefore λ is real.

Since real symmetric matrices are Hermitian, their eigenvalues are real. □

Proof of Theorem C.1. As described in [127], this can be proven as follows. Let A be a Hermitian diagonally dominant matrix with real non-negative diagonal entries. By Lemma C.4, its eigenvalues are real. Furthermore, by Gershgorin's circle theorem, for each eigenvalue λ an index i exists such that: $\lambda \in [a_{ii} - \sum_{i \neq j} |a_{ij}|, a_{ii} + \sum_{i \neq j} |a_{ij}|]$. Then, by assumption we have that $a_{ii} \geq 0, \forall i$ and by definition C.7 of diagonal dominance, we have that $a_{ii} \geq \sum_{i \neq j} |a_{ij}|$, therefore $\lambda \geq 0$. □

Proof of Theorem C.2. This proof is analogous to the proof of Theorem C.1 with strict inequalities. □

References

[1] Alex Townsend. *Why are so many matrices of low rank in computational math?* https://pi.math.cornell.edu/~ajt/presentations/LowRankMatrices.pdf. [Online; accessed 02-August-2024]. 2017.

[2] Ethan N. Epperly. *Big Ideas in Applied Math: Low-rank Matrices*. https://www.ethanepperly.com/index.php/2021/10/26/big-ideas-in-applied-math-low-rank-matrices/. [Online; accessed 30-July-2024]. 2021.

[3] Borbala Hunyadi and Kim Batselier. "Loss of information due to flatenning". Lecture slides, EE4750 Tensor Networks for Green AI and Signal Processing Lecture 1, TUDelft, September 17, 2023. Unpublished. 2023.

[4] Ivan V Oseledets. "Tensor-train decomposition". In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2295–2317.

[5] Ivan V Oseledets and Sergey V Dolgov. "Solution of linear systems and matrix inversion in the TT-format". In: *SIAM Journal on Scientific Computing* 34.5 (2012), A2718–A2739.

[6] Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. "The alternating linear scheme for tensor optimization in the tensor train format". In: *SIAM Journal on Scientific Computing* 34.2 (2012), A683–A713.

[7] Sergey V Dolgov, Boris N Khoromskij, and Ivan V Oseledets. "Fast solution of parabolic problems in the tensor train/quantized tensor train format with initial application to the Fokker–Planck equation". In: *SIAM Journal on Scientific Computing* 34.6 (2012), A3016–A3038.

[8] Sergey V Dolgov and Dmitry V Savostyanov. "Alternating minimal energy methods for linear systems in higher dimensions". In: *SIAM Journal on Scientific Computing* 36.5 (2014), A2248–A2271.

[9] Lingjie Li, Wenjian Yu, and Kim Batselier. "Faster tensor train decomposition for sparse data". In: *Journal of Computational and Applied Mathematics* 405 (2022), p. 113972.

[10] Kim Batselier et al. "Computing low-rank approximations of large-scale matrices with the tensor network randomized SVD". In: *SIAM Journal on Matrix Analysis and Applications* 39.3 (2018), pp. 1221–1244.

[11] Thomas H Cormen et al. *Introduction to Algorithms*. MIT press, 2022.

[12] Vincent Hindriksen. *How expensive is an operation on a CPU?* https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/. [Online; accessed 30-July-2024]. 2012.

[13] J Kleinberg and E Tardos. *Algorithm Design*. Vol. 92. Pearson Education, 2006.

[14] Catalin Trenchea and Kim Wong. *Course notes MATH2071, Lab 6: Solving linear systems*. https://sites.pitt.edu/~kimwong/lab06/index.html. [Online; accessed 23-April-2024]. 2019.

[15] Richard William Farebrother. *Linear least squares computations*. Routledge, 2018.

[16] Binan Gu. *MA3257 lecture notes: Gaussian Elimination - Exact Operation Counts*. https://users.wpi.edu/~bgu/sp23/ma3257/lecture_notes/MA3257_L10.pdf. [Online; accessed 23-April-2024]. 2023.

[17] Masayuki Yano et al. *(Numerical) Linear Algebra 2 - Solution of Linear Systems, Unit 5, Section 27.3 Gaussian Elimination and Back Substitution*. https://eng.libretexts.org/Bookshelves/Mechanical_Engineering/Math_Numerics_and_Programming_(for_Mechanical_Engineers). [Online; accessed 30-July-2024]. 2024.

[18] Jens Saak and Dipl.-Math Martin Kohler. *Scientific Computing 1 Handout-Tutorial 10: Exact Flop-Count for the LU Decomposition*. https://cscproxy.mpi-magdeburg.mpg.de/mpcsc/lehre/2016_WS_SC/handouts/handout_LU_counting.pdf. [Online; accessed 23-April-2024]. 2017.

[19] Frolov Alexey Vyacheslavovich. *Backward substitution*. https://algowiki-project.org/en/Backward_substitution. [Online; accessed 24-April-2024]. 2022.

[20] Frolov Alexey Vyacheslavovich. *Forward substitution*. https://algowiki-project.org/en/Forward_substitution. [Online; accessed 24-April-2024]. 2022.

[21] Nicholas J. Higham. *Functions of Matrices*. Society for Industrial and Applied Mathematics, 2008. DOI: 10.1137/1.9780898717778. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717778.

[22] Stephen Boyd and Mert Pilanci. *EE364b lecture notes: Numerical Linear Algebra background*. https://stanford.edu/class/ee364b/lectures/num-lin-alg2.pdf. [Online; accessed 23-April-2024]. 2023.

[23] Magnus Rudolph Hestenes, Eduard Stiefel, et al. *Methods of conjugate gradients for solving linear systems*. Vol. 49. 1. NBS Washington, DC, 1952.

[24] Youcef Saad and Martin H Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems". In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869.

[25] Jonathan R Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. USA: Carnegie Mellon University, 1994.

[26] MathWorks ®. *Iterative Methods for Linear Systems*. https://nl.mathworks.com/help/matlab/math/iterative-methods-for-linear-systems.html. [Online; accessed 23-April-2024]. 2023.

[27] Richard Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. DOI: 10.1137/1.9781611971538. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611971538.

[28] Roger Fletcher. "Conjugate gradient methods for indefinite systems". In: *Numerical Analysis: Proceedings of the Dundee Conference on Numerical Analysis, 1975*. Springer. 2006, pp. 73–89.

[29] Henk A Van der Vorst. "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems". In: *SIAM Journal on scientific and Statistical Computing* 13.2 (1992), pp. 631–644.

[30] Peter Sonneveld. "CGS, a fast Lanczos-type solver for nonsymmetric linear systems". In: *SIAM journal on scientific and statistical computing* 10.1 (1989), pp. 36–52.

[31] Greg Fasshauer. *Preconditioning, Chapter 16 of Class Notes, 477/577 Numerical Linear Algebra/Computational Mathematics I, Illnois Institute of Technology*. http://www.math.iit.edu/~fass/477_577_handouts.html. [Online; accessed 14-March-2024]. 2006.

[32] Chung-Han Chou et al. "On the preconditioner of conjugate gradient method - A power grid simulation perspective". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2011, pp. 494–497.

[33] Virginia Klema and Alan Laub. "The singular value decomposition: Its computation and some applications". In: *IEEE Transactions on automatic control* 25.2 (1980), pp. 164–176.

[34] N Kishore Kumar and Jan Schneider. "Literature survey on low rank approximation of matrices". In: *Linear and Multilinear Algebra* 65.11 (2017), pp. 2212–2244.

[35] Biswa Nath Datta. *Numerical Linear Algebra and Applications, 2nd Edition*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2010. DOI: 10.1137/1.9780898717655. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717655.

[36] Gene H. Golub and Charles F. Van Loan. *Matrix Computations - 4th Edition*. Philadelphia, PA: Johns Hopkins University Press, 2013. DOI: 10.1137/1.9781421407944. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781421407944.

[37] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions”. In: *SIAM review* 53.2 (2011), pp. 217–288.

[38] Tony F Chan. “Rank revealing QR factorizations”. In: *Linear Algebra and its Applications* 88 (1987), pp. 67–82.

[39] Ming Gu and Stanley C Eisenstat. “Efficient algorithms for computing a strong rank-revealing QR factorization”. In: *SIAM Journal on Scientific Computing* 17.4 (1996), pp. 848–869.

[40] C-T Pan. “On the existence and computation of rank-revealing LU factorizations”. In: *Linear Algebra and its Applications* 316.1-3 (2000), pp. 199–222.

[41] L Miranian and Ming Gu. “Strong rank revealing LU factorizations”. In: *Linear Algebra and its Applications* 367 (2003), pp. 1–16.

[42] Ming Gu and Luiza Miranian. “Strong rank revealing Cholesky factorization”. In: *Electronic Transactions on Numerical Analysis* 17 (2004), pp. 76–92.

[43] DirkT. *Alternatives to SVD for rank factorization*. Computer Science Stack Exchange. [Online; accessed 1-May-2024]. eprint: `https://cs.stackexchange.com/q/162228`.

[44] Esragul Korkmaz et al. “Reaching the Quality of SVD for Low-Rank Compression Through QR Variants”. PhD thesis. Inria Bordeaux-Sud Ouest, 2022.

[45] Nathan D Heavner. “Building rank-revealing factorizations with randomization”. PhD thesis. University of Colorado at Boulder, 2019.

[46] Christian H Bischof and Gregorio Quintana-Ortí. “Computing rank-revealing QR factorizations of dense matrices”. In: *ACM Transactions on Mathematical Software (TOMS)* 24.2 (1998), pp. 226–253.

[47] Shivkumar Chandrasekaran and Ilse CF Ipsen. “On rank-revealing factorisations”. In: *SIAM Journal on Matrix Analysis and Applications* 15.2 (1994), pp. 592–622.

[48] Andrzej Cichocki et al. “Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions”. In: *Foundations and Trends® in Machine Learning* 9.4-5 (2016), pp. 249–429.

[49] Richard E. Bellman. *A Guided Tour*. Princeton: Princeton University Press, 1961. ISBN: 9781400874668. DOI: `doi:10.1515/9781400874668`. URL: `https://doi.org/10.1515/9781400874668`.

[50] Tamara G Kolda and Brett W Bader. “Tensor decompositions and applications”. In: *SIAM review* 51.3 (2009), pp. 455–500.

[51] Johan Håstad. “Tensor rank is NP-complete”. In: *Journal of Algorithms* 11.4 (1990), pp. 644–654.

[52] Vin De Silva and Lek-Heng Lim. “Tensor rank and the ill-posedness of the best low-rank approximation problem”. In: *SIAM Journal on Matrix Analysis and Applications* 30.3 (2008), pp. 1084–1127.

[53] Ledyard R Tucker. “Some mathematical notes on three-mode factor analysis”. In: *Psychometrika* 31.3 (1966), pp. 279–311.

[54] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. “A multilinear singular value decomposition”. In: *SIAM journal on Matrix Analysis and Applications* 21.4 (2000), pp. 1253–1278.

[55] Ivan Oseledets. “Compact matrix form of the d-dimensional tensor decomposition”. In: *IEICE Proceedings Series* 43.B2L-C2 (2009).

[56] Ivan Oseledets and Eugene Tyrtyshnikov. “TT-cross approximation for multidimensional arrays”. In: *Linear Algebra and its Applications* 432.1 (2010), pp. 70–88.

[57] Benjamin Huber, Reinhold Schneider, and Sebastian Wolf. “A randomized tensor train singular value decomposition”. In: *Compressed Sensing and its Applications: Second International MATHEON Conference 2015*. Springer. 2017, pp. 261–290.

[58] Anh-Huy Phan et al. “Tensor networks for latent variable analysis. Part I: Algorithms for tensor train decomposition”. In: *arXiv preprint arXiv:1609.09230* (2016).

[59] Rodica D. Costin. *Spectral Properties of Self-adjoint Matrices*. https://people.math.osu.edu/costin.10/5101/Selfadjointness.pdf. [Online; accessed 19-May-2024]. 2013.

[60] Andrzej Cichocki et al. “Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives”. In: *Foundations and Trends® in Machine Learning* 9.6 (2017), pp. 431–673.

[61] Mike Espig, Wolfgang Hackbusch, and Aram Khachatryan. “On the convergence of alternating least squares optimisation in tensor format representations”. In: *arXiv preprint arXiv:1506.00062* (2015).

[62] Joseph M Landsberg, Yang Qi, and Ke Ye. “On the geometry of tensor network states”. In: *arXiv preprint arXiv:1105.4449* (2011).

[63] Thorsten Rohwedder and André Uschmajew. “On local convergence of alternating schemes for optimization of convex problems in the tensor train format”. In: *SIAM Journal on Numerical Analysis* 51.2 (2013), pp. 1134–1162.

[64] Ivan V Oseledets, Maxim V Rakhuba, and André Uschmajew. “Alternating least squares as moving subspace correction”. In: *SIAM Journal on Numerical Analysis* 56.6 (2018), pp. 3459–3479.

[65] Ivan Oseledets et al. *TT-Toolbox*. https://github.com/oseledets/TT-Toolbox. 2024.

[66] Namgil Lee and Andrzej Cichocki. “Regularized computation of approximate pseudoinverse of large matrices using low-rank tensor train decompositions”. In: *SIAM Journal on Matrix Analysis and Applications* 37.2 (2016), pp. 598–623.

[67] Hengnu Chen et al. “Tensor train decomposition for solving large-scale linear equations”. In: *Neurocomputing* 464 (2021), pp. 203–217.

[68] Sergey V Dolgov. “TT-GMRES: solution to a linear system in the structured tensor format”. In: *Russian Journal of Numerical Analysis and Mathematical Modelling* 28.2 (2013), pp. 149–172.

[69] Timothy A Davis and Yifan Hu. “The University of Florida sparse matrix collection”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[70] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[71] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[72] Dr. Patrick Gelß et al. *Scikit-TT*. https://github.com/PGelss/scikit_tt. 2024.

[73] Vladimir A. Kazeev and Boris N. Khoromskij. “Low-Rank Explicit QTT Representation of the Laplace Operator and Its Inverse”. In: *SIAM Journal on Matrix Analysis and Applications* 33.3 (2012), pp. 742–758. DOI: 10.1137/100820479. eprint: https://doi.org/10.1137/100820479.

[74] Elizabeth Cuthill and James McKee. “Reducing the bandwidth of sparse symmetric matrices”. In: *Proceedings of the 1969 24th national conference*. 1969, pp. 157–172.

[75] Jeremy Siek. *C++ Boost Cuthill-McKee Ordering*. https://cs.brown.edu/~jwicks/boost/libs/graph/doc/cuthill_mckee_ordering.html. [Online; accessed 06-June-2024].

[76] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[77] Sergey Slotin. *Integer Factorization*. https://en.algorithmica.org/hpc/algorithms/factorization/. [Online; accessed 05-June-2024].

[78] Herman WJ Kernkamp et al. “Efficient scheme for the shallow water equations on unstructured grids with application to the Continental Shelf”. In: *Ocean Dynamics* 61 (2011), pp. 1175–1188.

[79] David G Harris, Francis Sullivan, and Isabel Beichl. “Linear Algebra and Sequential Importance Sampling for Network Reliability”. In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*. IEEE. 2011, pp. 3339–3347.

[80] Sepehr Eslami et al. “Flow division dynamics in the mekong delta: application of a 1D-2D coupled model”. In: *Water* 11.4 (2019), p. 837.

[81] Deltares. *Delft3D FM Suite 2D3D*. https://www.deltares.nl/en/software-and-data/products/delft3d-flexible-mesh-suite. [Online; accessed 18-March-2024]. 2024.

[82] Jennifer Scott and Miroslav Tuma. *Algorithms for sparse linear systems*. Springer Nature, 2023.

[83] Mihalis Yannakakis. "Computing the minimum fill-in is NP-complete". In: *SIAM Journal on Algebraic Discrete Methods* 2.1 (1981), pp. 77–79.

[84] Esmond G Ng and Padma Raghavan. "Performance of greedy ordering heuristics for sparse Cholesky factorization". In: *SIAM Journal on Matrix Analysis and Applications* 20.4 (1999), pp. 902–914.

[85] Anshul Gupta. "Fast and effective algorithms for graph partitioning and sparse-matrix ordering". In: *IBM Journal of Research and Development* 41.1.2 (1997), pp. 171–183.

[86] Ajit Agrawal, Philip Klein, and Ramamurthy Ravi. "Cutting down on fill using nested dissection: provably good elimination orderings". In: *Graph Theory and Sparse Matrix Computation*. Springer, 1993, pp. 31–55.

[87] Matlab©. *Approximate minimum degree permutation*. https://nl.mathworks.com/help/matlab/ref/amd.html. [Online; accessed 06-June-2024].

[88] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. "An approximate minimum degree ordering algorithm". In: *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905.

[89] Matlab©. *Sparse Matrix Operations*. https://nl.mathworks.com/help/matlab/math/sparse-matrix-operations.html. [Online; accessed 06-June-2024].

[90] Stephen Ingram. *Minimum Degree Reordering Algorithms: A Tutorial*. http://sfingram.net/cs517_final.pdf. [Online; accessed 06-June-2024].

[91] Deltares. *Delft3D FM Suite 2D3D Technical Reference Manual*. https://content.oss.deltares.nl/delft3dfm2d3d/D-Flow_FM_Technical_Reference_Manual.pdf. [Online; accessed 18-March-2024]. 2024.

[92] Michael S Engelman. "FIDAP (A Fluid Dynamics Analysis Program)". In: *Advances in Engineering Software (1978)* 4.4 (1982), pp. 163–166.

[93] Ion Gabriel. *torchTT*. https://github.com/ion-g-ion/torchTT. 2024.

[94] Martin Andersen, Joachim Dahl, and Lieven Vandenberghe. *cvxopt*. https://github.com/cvxopt/cvxopt. 2024.

[95] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[96] Martin Licht. *Gaussian elimination and LU decomposition*. https://mathweb.ucsd.edu/~mlicht/wina2021/pdf/lecture06.pdf. [Online; accessed 10-August-2024]. 2021.

[97] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[98] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[99] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: https://plot.ly.

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

[101] Satish Balay et al. *PETSc/TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.21. Argonne National Laboratory, 2024. DOI: 10.2172/2205494.

[102] C Hubig, IP McCulloch, and Ulrich Schollwöck. "Generic construction of efficient matrix product operators". In: *Physical Review B* 95.3 (2017), p. 035129.

[103] SL Gonzaga de Oliveira and Guilherme Oliveira Chagas. "A systematic review of heuristics for symmetric-matrix bandwidth reduction: methods not based on metaheuristics". In: *The XLVII Brazilian Symposium of Operational Research (SBPO)*. Vol. 6. 2015.

[104] Petrică Pop, Oliviu Matei, and Călin-Adrian Comes. “Reducing the bandwidth of a sparse matrix with a genetic algorithm”. In: *Optimization* 63.12 (2014), pp. 1851–1876.

[105] Guilherme Oliveira Chagas and Sanderson L Gonzaga de Oliveira. “Metaheuristic-based heuristics for symmetric-matrix bandwidth reduction: a systematic review”. In: *Procedia Computer Science* 51 (2015), pp. 211–220.

[106] Norman E Gibbs, William G Poole Jr, and Paul K Stockmeyer. “A comparison of several bandwidth and profile reduction algorithms”. In: *ACM Transactions on Mathematical Software (TOMS)* 2.4 (1976), pp. 322–330.

[107] Edward Rothberg and Stanley C Eisenstat. “Node selection strategies for bottom-up sparse matrix ordering”. In: *SIAM Journal on Matrix Analysis and Applications* 19.3 (1998), pp. 682–695.

[108] Lars Grasedyck, Melanie Kluge, and Sebastian Kramer. “Variants of alternating least squares tensor completion in the tensor train format”. In: *SIAM Journal on Scientific Computing* 37.5 (2015), A2424–A2450.

[109] Michael Maximilian Steinlechner. *Riemannian optimization for solving high-dimensional problems with low-rank tensor structure*. Tech. rep. EPFL, 2016.

[110] Sergey V Dolgov et al. “Computation of extreme eigenvalues in higher dimensions using block tensor train format”. In: *Computer Physics Communications* 185.4 (2014), pp. 1207–1216.

[111] Daniel Kressner, Michael Steinlechner, and André Uschmajew. “Low-rank tensor methods with subspace correction for symmetric eigenvalue problems”. In: *SIAM Journal on Scientific Computing* 36.5 (2014), A2346–A2368.

[112] Namgil Lee and Andrzej Cichocki. “Estimating a few extreme singular values and vectors for large-scale matrices in tensor train format”. In: *SIAM Journal on Matrix Analysis and Applications* 36.3 (2015), pp. 994–1014.

[113] Shreedhar Maskey. *Catchment hydrological modelling: The science and art*. Elsevier, 2022.

[114] Cornelis Boudewijn Vreugdenhil. *Numerical methods for shallow-water flow*. Vol. 13. Springer Science & Business Media, 2013.

[115] Ole Mark et al. “Potential and limitations of 1D modelling of urban flooding”. In: *Journal of hydrology* 299.3-4 (2004), pp. 284–299.

[116] Susana Ochoa-Rodríguez et al. *Urban Pluvial Flood Modelling: Current Theory and Practice*. `http://www.raingain.eu/sites/default/files/wp3_review_document.pdf`. [Online; accessed 20-April-2024]. 2013.

[117] Jorge Leandro et al. “Comparison of 1D/1D and 1D/2D coupled (sewer/surface) hydraulic models for urban flood simulation”. In: *Journal of hydraulic engineering* 135.6 (2009), pp. 495–504.

[118] Richard Allitt et al. “Investigations into 1D-1D and 1D-2D urban flood modelling”. In: *WaPUG Autumn Conference*. Vol. 25. 2009, pp. 1–12.

[119] Yulong Xing. “Numerical methods for the nonlinear shallow water equations”. In: *Handbook of Numerical Analysis*. Vol. 18. Elsevier, 2017, pp. 361–384.

[120] James William Thomas. *Numerical partial differential equations: finite difference methods*. Vol. 22. Springer Science & Business Media, 2013.

[121] Fadl Moukalled et al. *The finite volume method*. Springer, 2016.

[122] Charles F Van Loan and G Golub. *Matrix Computations - 3rd Edition*. Vol. 5. Johns Hopkins University Press, 1996.

[123] Kendall Atkinson. *An Introduction to Numerical Analysis*. John Wiley & Sons, 1991.

[124] Encyclopedia of Mathematics. *Condition number*. `https://encyclopediaofmath.org/wiki/Condition_number`. [Online; accessed 29-April-2024]. 2020.

[125] David A Belsley, Edwin Kuh, and Roy E Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*. John Wiley & Sons, 2005.

[126] anonymous67. *A question about the proof of a symmetric matrix has real eigenvalues*. Mathematics Stack Exchange. [Online; accessed 29-April-2024]. eprint: `https://math.stackexchange.com/q/899175`.

[127] Andrea Ambrosio. *properties of diagonally dominant matrix*. `https://planetmath.org/propertiesofdiagonallydominantmatrix`. [Online; accessed 29-April-2024]. 2013.