Project No ML - 23: Low Rank Approximation with Alternate Optimization

Master Degree in Computer Science, University of Pisa
Computational Mathematics for Learning and Data Analysis
Academic Year 2024/2025

Giacomo Aru - g.aru@studenti.unipi.it - 597700
Simone Marzeddu - s.marzeddu@studenti.unipi.it - 597134

Abstract

This project addresses the low-rank matrix approximation problem, aiming to approximate a high-dimensional matrix $A \in \mathbb{R}^{m \times n}$ by the product of two lower-dimensional matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$, with $k < \min(m, n)$. This dimensionality reduction enhances computational efficiency and interpretability. To achieve this, an alternating minimization strategy is used, iteratively optimizing U and V until convergence. The project thus outlines a systematic approach for constructing efficient low-rank approximations with robust algorithmic stability.

1 Introduction

The goal of this project is to solve a low-rank matrix approximation problem. Given a matrix $A \in \mathbb{R}^{m \times n}$, the objective is to find two matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$, where $k < \min(m, n)$, such that the product UV^T closely approximates A. Mathematically, this is expressed as the following optimization problem:

$$\min_{U \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{n \times k}} \|A - UV^T\|_F$$

where $\| \cdot \|_F$ denotes the Frobenius norm, which measures the error in terms of the sum of squared differences between A and its approximation UV^T.

The aim of the project is to represent a high-dimensional dataset in a lower-dimensional form while preserving significant structure or patterns within the data. By constraining the rank of UV^T to be k, we achieve a dimensionality reduction that can lead to advantages like computational efficiency or improved interpretability.

1.1 Solution Approach - Projectual Requirement

To solve this problem, we employ an alternating minimization strategy. Starting with an initial guess for V, we alternate between optimizing U and V iteratively:

1. Fix V: For a given V, we solve for U as:

$$U_1 = \arg\min_U \|A - UV^T\|_F$$

2. Fix U: Using the updated U, we then solve for V as:

$$V_1 = \arg\min_V \|A - UV^T\|_F$$

These steps are repeated, updating U and V iteratively until (hopefully) convergence is achieved. Each optimization step reduces the overall error until a locally optimal low-rank approximation is obtained.

To solve each subproblem (i.e., optimizing U or V), we will employ QR factorization, a matrix decomposition technique that expresses a matrix as the product of an orthogonal matrix Q and an upper triangular matrix R.

1.2 Algorithmic Choices

Each step of the iterative minimization consists in solving two linear systems in which we have (alternatively) the matricial variable V or U. This two sub-problems are very similar and thus will be solved in similar ways, this is clear if we analise the computation of V^T respect to U:

$$U_{t+1} = \arg\min_{U} \|A - U(QR)^T\|_F$$

$$V_{t+1} = \arg\min_{V^T} \|A^T - V(QR)^T\|_F$$

From now on, we will therefore only refer to the steps related to one of these sub-problems, assuming that the same reasoning is applicable to the other.

Our approach to the solution of each subproblem consists of three steps:

- Apply QR Factorization to the constant (fixed) matrix.
- Manipulate the equation so to simplify it's form and consequently be able to exploit Backward Substitution.
- Apply Backward Substitution to solve the linear system.

1.2.1 QR Factorisation

With regard to the algorithm for the computation of the QR factorisation, our design has opted for a Housholder transformation approach.

Each step of the algorithm functions independently as a self-contained module, allowing flexibility in selecting the specific algorithm variant for factoring. At the beginning of each step, all necessary inputs for factoring are already known, and our goal is to obtain all outputs in a single round, avoiding intermediate results. For this reason, we choose the Householder transformation, as it is the most stable variant of the algorithm and enables computational savings in the $Q \times A$ product (as will be detailed in Section 1.2.2) by leveraging the properties of Householder vectors. Specifically, we can perform a Thin QR factorization by returning only R and the Householder vectors as output, which we then use to accelerate the multiplication process.

In particular the acceleration in the multiplication is given by the following property:

$$HA = A - 2u_i(u_i^T A)$$

Applying this property, thanks to to fact that each vector shrinks as i grows, less values of A are influenced, thus reducing the number of components relevant to our computation.

Leveraging these techniques we get an agorithmic complexity for each step of:

$$O(nk^2 + mk^2 + mnk)$$

This cost will be further discussed in chapter 2.

1.2.2 Equation Manipulation

Considering now to have a complete QR Factorization, we would find our system in this state:

$$U_{t+1} = \arg\min_{U} \|A - U(QR)^T\|_F$$

Now we want to mantain only the upper triangular matrix R as the variable coefficient in the system. To do so we exploit the property of the Frobenius Norm to be invariant respect to the multiplication of the argument with orthogonal matrices.

$$\|BA\|_F = \|A\|_F$$

where:

- A is any $m \times n$ matrix.
- B is an $m \times m$ orthogonal matrix ($B^T B = I$).

- $\|\cdot\|_F$ denotes the Frobenius norm, defined as $\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2} = \sqrt{\mathrm{tr}(A^T A)}$.

We know that this property holds for the 2-Norm but we have to demonstrate that this holds also for the F-Norm:

$$\|BA\|_F = \sqrt{\mathrm{tr}((BA)^T BA)} = \sqrt{\mathrm{tr}(A^T B^T BA)} = \sqrt{\mathrm{tr}(A^T A)} = \|A\|_F$$

With this knowledge we are now able to perform the desired manipulation:

$$\arg\min_U \|A - U(QR)^T\|_F = \arg\min_U \|A - UR^T Q^T\|_F = \arg\min_U \|(A - UR^T Q^T)Q\|_F = \arg\min_U \|AQ - UR^T\|_F$$

Its important to notice that the R matrix is divided in two blocks R1 and R0 with R0 completely filled with zeros. We can exploit this fact to perform a block matrix operation so to highlite the structure of the final matrix argument of the Frobenius Norm.

$$AQ - UR^T = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} [Q_1 Q_2] - \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \left[R_1^T \mid R_0^T\right] = \begin{bmatrix} A_1Q_1 & A_1Q_2 \\ A_2Q_1 & A_2Q_2 \end{bmatrix} - \begin{bmatrix} U_1R_1^T & U_1R_0^T \\ U_2R_1^T & U_2R_0^T \end{bmatrix} =$$

$$= \begin{bmatrix} A_1Q_1 - U_1R_1^T & A_1Q_2 \\ A_2Q_1 - U_2R_1^T & A_2Q_2 \end{bmatrix}$$

As we can see from the formula at the previous step, all the system's variable are located in the first block column and so in our optimization process we can minimize only these elements (as the others remain constant, constituting a sort of residual inside the norm). This leads us to some important conclusions:

- We can focus only on the first term and so perform a thin QR Factorization in our algorithm. That is because the minimum of the norm is reached when all the elements of the first term cancels out.

$$\arg\min_U \|AQ - UR^T\|_F = \arg\min_U \|AQ_1 - UR_1^T\|_F$$

- In the final matrix that we find as argument of the Frobenius Norm, rows are mutually independent and this consents us to divide the problem in k resolutions of simpler linear systems in the form $xA = y$, where A is a lower triangual matrix and x and y are vectors.
- Since the R matrix is an upper triangular matrix, we are able to apply Backward Substitution to solve each of the previously mentioned simpler linear systems.

1.2.3 Backward Substitution

Backward Substitution solves the system iteratively from the last to the first component of the solution vector:

$$x_i = \frac{1}{r_{ii}} \left(y_i - \sum_{j=n}^{i+1} r_{ij} x_j \right), \quad \text{for } i = n, \ldots, 1.$$

This approach is specifically designed to solve systems of linear equations where the coefficient matrix is triangular. This method is a natural fit for our problem due to the following reasons:

- The optimization process discussed in the previous sections leads a linear system where the coefficient matrix R upper triangular. By transposing the system to fit the form $xR^T = y$, we obtain R^T, a lower triangular matrix, which allows us to leverage Backward Substitution.
- Backward Substitution offers two key advantages over more general algorithms: it is straightforward and computationally efficient. The reason for this lies in the fact that it leverages and naturally suits the already analysed structure of the problem, which allows it to be more efficient.
- Although Backward Substitution is not inherently stable, if the upper triangular matrix R is well conditioned it is less likely to introduce substantial numerical instabilities. It is possible to demonstrate that if V (constant matrix in the optimization step) is well conditioned, this property is inherited by R in its factorization. Therefore, studying V we can predict the precision of our solution.

1.2.4 Parallelisation

The use of parallelisation techniques represents a state-of-the-art approach providing essential performance gains for the algorithm. In this section, we discuss some possible parallelisation strategies which may be implemented in our architecture, bearing in mind, however, that these applications are beyond the scope of this project and will not be further explored or exploited.

As explained in Section 1.2.1, QR factorisation is the core algorithmic process in our application. Since this is an iterative algorithm, it is not possible to effectively parallelise it in its outermost loop. Despite is feasible to calculate the Householder vectors independently, the projections, are necessarily iterative as they globally modify the matrix A. Nevertheless, it is possible to calculate the individual projections in blocks and thus in parallel. This approach becomes particularly effective when m, n and k grow considerably.

After QR factorisation, the system undergoes algebraic manipulation, as discussed in Section 1.2.2. If the algorithm remains sequential, it is possible to save operations by exploiting the properties of Householder vectors, applying the reflections iteratively. However, it is also possible to compute Q and then perform the matrix multiplication in blocks, which can be easily parallelized.

Finally, the system obtained, as already noted in Section 1.2.3, can be solved independently for each row. Parallelization naturally follows, as it is possible to solve k different linear systems with the lower triangular matrix R independently.

2 Algorithm Expectation

2.1 Convergence Properties

In the case of our algorithm, the solution set S is the following:

$$S = \{x = U \cdot V \,|\, U \in \mathbb{R}^{m\times k},\, V \in \mathbb{R}^{k\times n},\, f(x) \leq f(x_0)\}$$

It is possible to demonstrate that this set is compact since $f(x)$ is based on the Frobenious Norm, which is coercive: $f(x) \to \infty \implies \|x\| \to \infty$.

Therefore, the condition $f(x) \leq f(x_0)$ implies that x must belong to a closed ball of finite radius (determined by $f(x_0)$) in the metric induced by the norm $f(x)$, centered in A. Hence, x is bounded, which implies that S is bounded. The solution set S is also closed because $f(x)$ is continuous and the inequality is non-strict. Moreover, any limit of a sequence $\{x_i\} \in S$ with $x_i \to x$ satisfies $f(x) \leq f(x_0)$, so $x \in S$.

Since S is a closed and bounded subset of the Euclidean space $\mathbb{R}^{m\times n}$ it follows by the Heine-Borel theorem [2] that S is compact.

The compactness of the solution set S and the property of the sequence output of our optimization process of being weakly decreasing, guarantee the convergence of our algorithm. By applying the Corollary 2 in [3] the convergence point found is granted to be a critical point of the problem.

A critical point is a point where is impossible to improve the objective function in any direction, given the problem's constraints. A point of this kind satisfies the definition:

$$\nabla f(\hat{x})^\top (y - \hat{x}) \geq 0 \quad \text{for every } y \in X,$$

Practically, this means that the gradient of the objective function is zero, or the function can't be further minimized or maximized while respecting the problem's restrictions.

Proving the convergence of U and V is more complex, and the demonstrations provided so far do not imply it. Indeed it is possible to create a counterexample where given a sequence of x and U, where the norm of x decrease while the norm of U increase, is always possible to compute V that satisfies $U \times V = x$. In this case, the norm of V will necessarily be decreasing to maintain balance in the product.

It is still possible to demonstrate that if at least one between V or U converge to an optimal value, the other would also converge (Proposition 2 in [3]).

2.1.1 Conditional Convergence

If our problem had certain specific properties and if certain conditions were met, it would be possible to demonstrate more accurate and relevant convergence qualities.

If we could show that during the process of minimising the product norm, the norms of U and V also decrease at each step, then we could establish with certainty that the algorithm converges to stationary points of the problem. If an iteration did not lead to an improvement in the minimisation process, this would univocally imply that the gradient of the explored surface is 0, thus defining convergence at a stationary point.

Other than the already demonstrated properties, if our problem satisfies the hypothesis of having bounded S_u and S_v, where $U \in S_u$ and $V \in S_v$, the cluster points of the sequence generated by the algorithm would be stationary points of the problem, as proved in [4] in Theorem 2.1,. This would also imply that U and V necessarily converge. Since the algorithm would converge to a stationary point, it would not be possible to obtain any further improvement from that execution.

2.2 Complexity Analysis

The complexity of each iterative minimization step can be calculated as the summation of the individual complexities of its constituting three sub-steps: Thin QR Factorization, the algebraic manipulation and Backward Substitution.

2.2.1 Complexity of Thin QR Factorization

This topic has been widely analysed in the lectures and is therefore glossed over in our discussion. It is relevant to recall that, given $A \in \mathbb{R}^{n\times k}$, the complexity of this step is $O(k^3)$ if $k \approx n$ and is $O(n \cdot k^2)$ if $k << n$, like in our case.

2.2.2 Complexity of the algebraic manipulation

As mentioned, after QR Factorization each step requires the solution of $A \times Q$ product. This can be decomposed in $A \times Q_1 \times ... \times Q_k$ where each Q_i is in the form:

$$Q_i = \begin{bmatrix} I_{i-1} & 0 \\ 0 & H_i \end{bmatrix}$$

Follows that for each intermediate product $A_i \times Q_i$, where A_i is the matrix obtained as a result of all the previous matrix multiplications, can be rewritten as;

$$A_i \times Q_i = \begin{bmatrix} A_{i1} & A_{i2} \\ A_{i3} & A_{i4} \end{bmatrix} \times \begin{bmatrix} I_{i-1} & 0 \\ 0 & H_i \end{bmatrix} = \begin{bmatrix} A_{i1} & A_{i2}H_i \\ A_{i3} & A_{i4}H_i \end{bmatrix}$$

Where the left part of A_i remains unchanged and only the right part needs to be multiplied by H_i.

We can achieve computational efficiency in each of these products by leveraging the following property of Householder vectors:

$$AH = A - 2(A \cdot u)u^T$$

Instead of performing a matrix product between $A \in \mathbb{R}^{m \times i}$ and $H \in \mathbb{R}^{i \times i}$, whose complexity is $O(m \cdot n^2)$, we are able to perform only one matrix-vector multiplication, an outer product and a matrix subtraction. This leads to the improved complexity of $O(m \cdot i + m \cdot i + m \cdot i) = O(m \cdot i)$.

Summing for each sub-product, we achieve a final complexity of:

$$\text{Total Complexity} = 2 \cdot \sum_{i=n}^{n-k+1} O(m \cdot i) = 2 \cdot O\left(m \cdot \frac{k(2n-k+1)}{2}\right) = O(m \cdot k \cdot n)$$

2.2.3 Complexity of Backward Substitution

The linear system generated by the second sub-step mentioned above is $AQ - UR^T = Y - UR^T$ where $Y \in \mathbb{R}^{m \times k}$, $U \in \mathbb{R}^{m \times k}$ and $R \in \mathbb{R}^{k \times k}$. We can apply Backward Substitution 1.2.3 for each row of U independently, according to the formula:

$$x_i = \frac{1}{r_{ii}} \left(y_i - \sum_{j=k}^{i+1} r_{ij} x_j \right), \quad \text{for } i = k, \ldots, 1.$$

This formula imply that each component of the solution vector depends on the already calculated ones. Therefore, the computational complexity depends quadratically on the solution vector's length k.

We need to consider that this computational weight needs to be multiplyed by the number of system solved with Forward Substitution (one for each row of the linear system).

Follows that we achieve a final complexity of:

$$\text{Total Complexity} = \sum_{l=1}^{m} \sum_{i=k}^{1} O(i) = \sum_{l=1}^{m} O\left(\frac{k(k+1)}{2}\right) = O(m \cdot k^2)$$

2.2.4 Total Complexity

As mentioned at the beginning of this section, the final complexity is obtained by adding up the complexity of each sub-step, and it must also be remembered that the minimisation steps are always paired two by two, with the minimisation step respect to U having both the variable and constant matrices inverted respect to to the minimisation step respect to V. According to this, the dimensionality of the rows and columns will be also inverted between each minimisation step:

$$\text{Total Complexity} = O(\text{minimization of } U) + O(\text{minimization of } V) =$$
$$= (O(n \cdot k^2) + O(m \cdot k \cdot n) + O(m \cdot k^2)) + (O(m \cdot k^2) + O(n \cdot k \cdot m) + O(n \cdot k^2)) =$$
$$= O(n \cdot k^2 + m \cdot k^2 + n \cdot m \cdot k)$$

3 Experimental Setup

This chapter outlines the experimental setup designed to measure and analyze the performance and characteristics of our code. The testing process we devised is divided into two sequential phases. The first phase focused on conducting a general test of the algorithm to ensure its correct functionality, while also examining the influence of initialization and termination parameters across different types of matrices.

Based on the general insights gained regarding the impact of these parameters, the second phase of testing was designed to include extensive yet targeted test cases. These were aimed at addressing questions that arose during the first phase and evaluating the algorithm's performance, while avoiding computationally disadvantageous or uninteresting initialization and parameter configurations for the relevant use cases.

Among the matrices tested are realistic cases, such as photographs of varying sizes depicting horses, low-resolution video game sprites, and the MNIST benchmark. Additionally, artificially generated matrices with specific desired properties were included.

The details and reasons behind each test will be discussed in the following sections.

3.1 Initialisation and Termination Methods

The code offers control over the initialisation and termination of the algorithm. Since in our implementation the matrix U is the first to be computed in a minimisation step, the matrix V can be initialised according to the following seven approaches:

1. **Random Uniform Distribution (`rand_u`)**
   - **Initialization**: Elements of V are drawn from a uniform distribution over the range $[-1, 1]$.
   - **Use Case**: Provides a simple and computationally cheap way to initialize V. It ensures that no particular structure or bias is introduced into the optimization process. Suitable as a baseline to understand how pure randomness without scaling influences convergence. May lead to slower convergence due to poor alignment with the data distribution in A.
2. **Random Normal Distribution (`rand_n`)**
   - **Initialization**: Elements of V are drawn from a normal (Gaussian) distribution with mean 0 and standard deviation 1.
   - **Use Case**: Also provides randomness, but draws values from a Gaussian distribution. This is often closer to the natural distribution of many datasets. Helps in analyzing whether Gaussian initialization aligns better with typical data properties compared to uniform distribution. It serves as a baseline but may have more consistent behavior for datasets with a normal structure.
3. **Scaled Uniform Distribution (`scaled_u`)**
   - **Initialization**: Elements of V are drawn from a scaled uniform distribution where:
     $$\text{Range} = \left[-\sqrt{\frac{3\|A\|_F}{nk}}, \sqrt{\frac{3\|A\|_F}{nk}} \right]$$
     where $\|A\|_F$ is the Frobenius norm of A.
   - **Use Case**: Ensures that the magnitude of $\|V\|$ matches the magnitude of $\|U\|$ knowing that the initialization needs to align with the Frobenius norm of A. Prevents the alternating minimization process from being dominated by scaling mismatches early on.
4. **Scaled Normal Distribution (`scaled_n`)**
   - **Initialization**: Elements of V are drawn from a normal (Gaussian) distribution with mean 0 and standard deviation:
     $$\gamma = \sqrt{\frac{\|A\|_F}{nk}}$$

- **Use Case**: Similar to the previous, but uses Gaussian values, which might better approximate natural data distributions. Useful for studying the interplay between random Gaussian initialization and scaling, especially for structured data. These last two initialisations may offer insights into whether the choice of uniform vs. normal scaling affects the optimization trajectory.

5. **Sketching with Gaussian Distribution (`sketching_g`)**

   - **Initialization**:
     (a) Compute $V = \text{transpose}(G \times A)$, where G is a matrix of Gaussian random variables.
     (b) Scale V to ensure $\|V\|_F \approx \sqrt{\|A\|_F \sqrt{k}}$.
   - **Use Case**: Uses a projection of A onto a Gaussian random subspace, effectively extracting a low-dimensional representation informed by A. The Gaussian distribution ensures that the entries of G are drawn from a continuous and symmetric distribution with zero mean. This provides a data-driven initialization that might reduce the number of iterations needed for convergence. It is also scaled to ensure similar norms between U and V at the beginning of the iterative process.

6. **Sketching with Bernoulli Distribution (`sketching_b`)**

   - **Initialization**:
     (a) Compute $V = \text{transpose}(B \times A)$, where B is a matrix of random values drawn from $\{-1, 1\}$ with equal probability.
     (b) Scale V to ensure $\|V\|_F \approx \sqrt{\|A\|_F \sqrt{k}}$.
   - **Use Case**: Similar to `sketching_g`, this method employs a Bernoulli distribution for the projection matrix, where the entries are drawn from a discrete and symmetric distribution. This results in a different randomization structure compared to the continuous Gaussian case, which can influence the behavior of the algorithm depending on the properties of A. While both methods achieve a similar but negligible computational cost, the choice of distribution reflects a trade-off in the statistical properties of the random projections rather than efficiency considerations.

7. **Semi-Orthogonal Initialization (`semi-orthogonal`)**

   - **Initialization**:
     (a) Generate a random $n \times k$ matrix from a Gaussian distribution.
     (b) Perform a thin QR factorization to ensure semi-orthogonality of V.
   - **Use Case**: Produces a matrix V with good structural properties, such as orthonormal columns, which can serve as a robust starting point for an iterative process. This initialization provides a well-conditioned basis that can improve numerical behavior and stability during the optimization process. The semi-orthogonal structure ensures that V spans a subspace with balanced contributions, which may help guide the iterations toward a low-rank approximation of A.

The termination of the algorithm is governed by a tolerance parameter ϵ, which represents the minimum accepted difference in the objective function's value between two consecutive iterations, where one iteration includes both minimization steps for U and V. This approach was chosen because it is impractical to predict the final minimum value of the objective function achievable with this technique a priori (except with significant computational overhead that would yield only an estimate). Instead, the convergence of the algorithm is monitored by checking whether the matrices U and V change between iterations, and this change is also reflected in the objective function. For this reason, evaluating the objective function provides a reliable indicator of convergence. Furthermore, ϵ is defined as a relative value, scaled by a contribution derived from A, as using an absolute threshold—independent of A would lead to results that are meaningful only in some cases, but inconsistent or unreliable in others.

3.2 Results Storage

For each execution of the algorithm, all relevant values are stored to allow both the identification of the specific execution and test case, as well as the analysis of the algorithm's behavior and performance. Specifically, all input parameters of the function, machine-specific information, the matrix A, the initial V_0, and the final solution matrices U and V are saved. Additionally, a CSV file is generated containing detailed data for each half-iteration, including the objective function value, the norm of U, the norm of V, the execution times of the three phases for each minimization step, and the iteration number.

3.3 First Experimental Phase

For this phase, we design around 3500 algorithm executions to perform tests varying with respect to the matrix to be approximated A, the initialization method for the matrix V, and the tolerance ϵ to control termination precision.

In this first phase, we study a few samples from different groups of A initializations:

- **Artificial Matrices**: Artificially created matrices with specific properties are used to test the algorithm's behavior. Specifically, random matrices are generated from Uniform and Gaussian distributions to create full-rank matrices of high complexity. Diagonal matrices are included to test the algorithm in edge cases involving very sparse and regular matrices, while symmetric matrices are used to evaluate its performance on visually simple yet mathematically distinct cases. The last two types of matrices are created based on their eigenvalues: diagonal matrices are generated by directly assigning values along the diagonal, while symmetric matrices are constructed by multiplying a diagonal matrix (containing the eigenvalues) by an orthogonal matrix on the left and its transpose on the right, resulting in symmetric matrices.

- **MNIST Database**: The MNIST (Modified National Institute of Standards and Technology) database is a well-known dataset in machine learning and computer vision, widely used for training and testing image processing systems. The database contains $70,000$ grayscale images of handwritten digits (0 through 9). In addition to being a real-life case study, the images in this dataset present a strong contrast between white pixels (pixels representing the hand-drawn digit) and black background pixels, representing instances of low-rank matrices. Furthermore, to emphasize this aspect and generate matrices with a rank lower than half the size of their dimensions, we scale these images to larger sizes 250x250 using bilinear interpolation. This method preserves the rank, a fact we specifically test and confirm for our use case.

- **Weizmann Horse Database**: This dataset is originally created for semantic image segmentation problems using machine learning approaches. Specifically, it is divided into images containing the subjects to be segmented (horses) and manually created masks representing the ground truth segmentation. For our tests, we choose to use this dataset without the masks, as it provides a substantial number (327) of images with significant variation in size, color, quality, and noise levels. Additionally, since these are real-world images, they are good representatives of a realistic level of complexity.

Samples are selected from these three groups of matrices to meet our requirements for variety while keeping the total execution time low. This approach is necessary because the total number of executions is significantly impacted by the different values of other hyperparameters to be tested.

Specifically, from the first group, four random matrices are selected: two generated from a uniform distribution and two from a normal distribution, with dimensions of 100x100 and 250x250, respectively. Additionally, six diagonal matrices and six symmetric matrices are chosen, created from three combinations of eigenvalues, using the same two dimensions (100x100 and 250x250). The eigenvalue combinations are designed to produce well-conditioned matrices (condition number = 1) with eigenvalues within the ranges $[-10, 10]$ or $[-12.5, 12.5]$, and poorly-conditioned matrices (condition number = 1000 or 800 depending on the size and smallest eigenvalue). The identity matrix is also selected as a special case of a diagonal matrix.

Finally, from the Weizmann Horse Database, six different images are selected, with sizes ranging from 205x218 to 600x800, a limited number due to longer execution time. From the MNIST dataset, given the smaller size, a total of 40 images are chosen, evenly distributed across different digits (0 to 9) and varying dimensions. Half of them retain the original size of 27x27, while the remaining images are scaled to 250x250.

For each of these configurations, the tests focus on analyzing the k parameter, where k belongs to the set {1, 2, 5, 10, 20, 50, 100}. For each case, only k values smaller or equal to half of the minimum between m and n (the dimensions of matrix A) are considered, ensuring that the selected values are always valid and reasonable for the given matrix size.

Two other hyperparameters are tested thoroughly and with variation: ϵ and the initialization method. For the first, it is decided to test only two values, which, while limited, are highly significant and key. These values are close to the machine precision in 32-bit and 64-bit arithmetic. The first value tested is $1e-08$, slightly lower than the machine precision in 32-bit, which is $1.92e-07$. This value is chosen because it is ideal for the precision of a solution found by an iterative algorithm of this type, as it is often sufficient, avoiding long convergence times (a common characteristic of similar algorithms) which tend to slow down as they approach the final solution. Additionally, it is just below the 32-bit machine precision, but since the algorithm operates with 64-bit numbers, there are no issues with numerical approximations. The second value tested is $1e-15$, slightly greater than the machine precision in 64-bit, which is approximately $2.22e-16$. This represents the highest precision currently sustainable by the implementation, and it is actually larger than the threshold because a smaller value could lead to unpredictable behaviors due to improper handling of operations at excessively high precision.

The initialization methods, on the other hand, are tested in their entirety, as each is created with specific characteristics aimed at solving a potential problem or at distinctly influencing the initialization point.

3.4 Second Experimental Phase

The second experimental phase involves the creation and execution of 8 distinct test cases, each designed to study a specific aspect of the algorithm. These test cases vary in terms of the nature of A and hyperparameter values.

Here is a detailed list of every testcase created:

- **Identity Test**: From the execution of the initial phase of testing, what initially appears to be a problem is subsequently identified as a characteristic inherent to the algorithm, which is discussed in greater detail later. The observed issue is the algorithm's strange convergence behaviour to a solution when A is the identity matrix. In particular we observed that the algorithm execution lasts for just one iteration. The experiment described here examines whether this behavior persists when A is perturbed and how it is influenced by variations in k. Specifically, the algorithm applies to a 100x100 identity matrix, to which a Gaussian perturbation is added, with ϵ ranging from 1 to $1e-15$, and k taking values from the set $\{1, 5, 10, 20\}$. Furthermore, the value of ϵ is fixed at $1e-15$ to eliminate any potential errors or uncertainties arising from numerical approximations or premature termination of the algorithm. Additionally, the initialization method is set to rand_n, as it provides a "neutral" starting point with generated values that are comparable with the identity matrix.

- **Low Rank Test**: This test is specifically designed to investigate the relationship between the algorithm's parameter k and the rank of the matrix A to be approximated. To achieve this, all matrices A are generated using the formulation $A = B \cdot C^T$, where B and C are randomly generated matrices with columns equal to the desired rank of A. While this approach does not guarantee that the rank of A exactly matches the number of columns in B and C, it makes this outcome highly improbable due to the fully random nature of both matrices. For the test, rank and k values are chosen in the range of [20, 30], with the size of matrix A fixed at 100x100. The value of ϵ is set to $1e-15$ again to avoid any influence from this parameter or premature termination. The initialization method is set to sketching_b, as it proves to be one of the best methods, particularly for keeping the norms of U and V close.

- **Norm Test**: This test investigates the behavior of the algorithm in situations where U and V have significantly different norms. Special attention is given to the initial norms of U and V, as very distant values, generated from matrices with large differences in their magnitudes, may lead to undesirable behaviors. The objective of this test is to explore this scenario. A custom method generates V_0 with a specified norm, and this value varies between 1 and 10^{16}. Matrix A is fixed as a randomly generated 100x100 matrix drawn from a normal distribution, and k varies across the set $\{1, 5, 10, 20\}$. The value of ϵ is fixed at $1e-15$ for the same previously mentioned reasons.

- **Structural Properties Test**: This test studies the behavior of the algorithm when applied to approximate matrices A with specific structural characteristics. The goal is to understand whether the execution is influenced, in terms of speed or approximation quality, by matrix properties such as diagonality, symmetry, orthogonality, upper triangularity, or the condition number of the matrix. For this test, five methods for generating the matrix A are specifically created. Each method takes as input the matrix size, as well as the mean and variance of the normal distribution from which samples are drawn, and returns a matrix with the desired properties according to that distribution.

  The first method generates a diagonal matrix with random values along the diagonal. The second method creates an orthogonal matrix by extracting a random matrix and applying the QR decomposition to obtain an orthogonal Q. The third method generates a random matrix, multiplies it by its transpose, and then divides by two to produce a symmetric matrix. The fourth method draws values from the distribution to create an upper triangular matrix. Lastly, the fifth method generates a matrix from specified eigenvalues. To exclude other desirable properties, this method first creates a random matrix, applies the QR factorization to obtain an orthogonal Q, constructs the diagonal matrix Λ with the chosen eigenvalues, and then computes the final matrix A as $Q \cdot \Lambda \cdot Q^T$.

  For the tests related to the condition number, matrices are generated such that their condition number varies from 1 to 1,000,000, with intermediate exponential steps. For all executions, the size of A, generated using these methods, is fixed at 100x100 and 200x200. The value of k varies across the set $\{1, 5, 20, 50\}$, and the initialization method is set to sketching_g because it has proven in the previous phase to be a well-rounded initialization method. Finally, ϵ is fixed at $1e-08$ to reduce the total execution time, as no significant influence of this value on the results is expected.

- **Performance Test**: This test is specifically designed to understand the algorithm's performance as k, the dimensions, and the nature of A vary. For this particular test, an additional dataset is used to increase the variety of matrices, providing samples whose complexity and rank are positioned between the Weizmann Horse Database and the MNIST Database. The dataset consists of 80x80 sprites representing 50 different "Pokémon" (short for "Pocket Monsters", these are fantastic creatures characteristic of the famous global franchise), sourced from the "Pokémon Diamond and Pearl" videogame. These images are selected to have a rank close to the matrix dimensions. For each image in the Weizmann Horse Database, one corresponding image is selected from both the MNIST and the new Sprite dataset, with the addition of a randomly generated image from a uniform distribution. These three images are scaled or generated to match the dimensions of the first one, enabling comparisons between the different datasets while maintaining the same matrix size. The scaling preserves the complexity. k varies over the set $\{1, 2, 4, 8, 16, 32, 64\}$, the initialization method is set to sketching_g due to its proven excellence in all aspects of comparison with other methods, and ϵ is set to $1e-08$ to avoid long execution times.

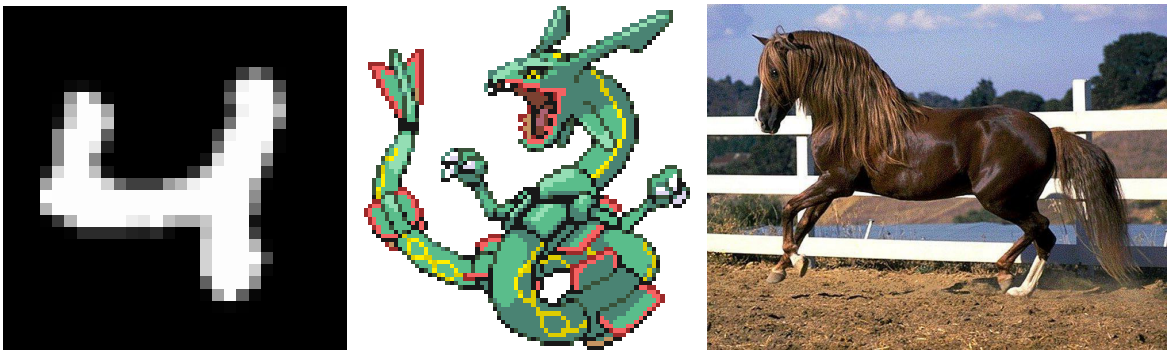


Figure 1: Samples extracted from our three datasets

- **OTS Test**: This test measures and compares the performance and capabilities of the algorithm we develop with those of other off-the-shelf algorithms. In particular, the two most relevant steps for comparisons with off-the-shelf approaches are thin QR factorisation and Backward Substitution. QR factorisation is implemented directly using **Numpy** methods, in particular `numpy.linalg.qr`. For the Backward Substitution, we use the `scipy.linalg` module, as this implementation offers a collection of functions to efficiently solve linear algebra problems. Among them, `solve_triangular` is commonly used to solve systems of linear equations in the form $Ax = b$ where A is a triangular

matrix (upper or lower). We perform all tests on the Weizmann Horse Database and Pokémon images using the off-the-shelf methods just described, so that we can compare the results with those obtained by our algorithm with the same input. Also in this case, images are scaled to match the dimensions of the Weizmann Horse Database one, still preserving the complexity of the scaled matrices. The parameter k varies over the set $\{1, 2, 4, 8, 16, 32, 64\}$, the initialization method is set to sketching_g due to its proven excellence in all aspects of comparison with other methods, and ϵ is set to $1e-08$ to avoid long execution times.

- **Convergence Test**:

  To perform this test, the algorithm runs again on the Weizmann Horse Database images. The goal of this task is to check if the convergence point reached by the executions matches or is similar to previous runs. Additionally, it aims to determine whether the algorithm reaches a local or global minimum.

  The test comprises two main analyses. The first focuses on examining whether multiple executions of the algorithm on identical configurations produce consistent outcomes. The second analysis involves a case-by-case comparison of the solutions generated by our algorithm with the global minimum. To compute the global minimum, we utilize truncated singular value decomposition (SVD), specifically leveraging the Eckart-Young theorem. According to this theorem, for a given matrix A and rank k, the optimal low-rank approximation of A with respect to the Frobenius norm is obtained by truncating the k principal singular values in the SVD decomposition.

4 Results

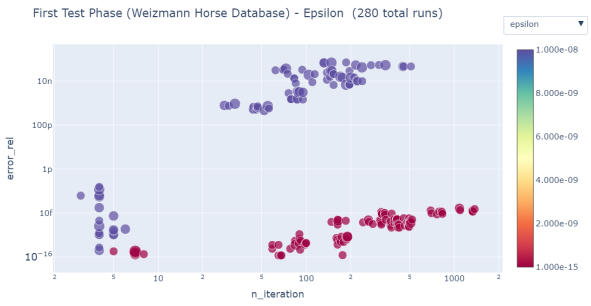
This document outlines the results of two experimental phases designed to evaluate the performance, behavior, and scalability of the algorithm under various configurations and conditions. The first phase focuses on foundational analyses, including the effects of the tolerance parameter ϵ, initialization methods, and structural characteristics of the input matrices. The second phase extends this exploration, introducing targeted test cases to analyze the algorithm's performance across diverse scenarios, including comparisons with off-the-shelf implementations, convergence behavior, and the impact of structural matrix properties. Together, these experiments provide a comprehensive understanding of the algorithm's strengths and limitations, guiding further refinements and applications.

The primary metrics of interest are as follows:

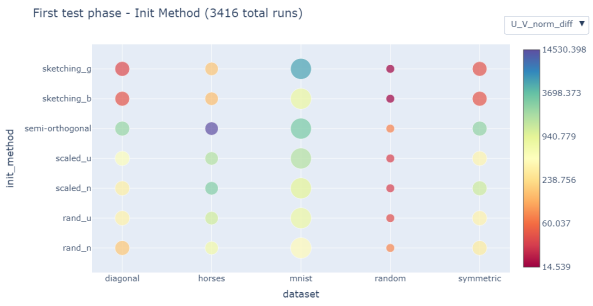
- Execution times, which can be categorized based on different operations (e.g., QR factorization, algebraic manipulation, backward substitution) or accumulated into mean execution time per algorithm step or total execution time of the whole algorithm.
- The number of iterations required by the algorithm to converge.
- The relative error between the optimal solution obtained using truncated SVD and the solution produced by our algorithm computed as: `error_rel` $= \frac{|f(U_k,V_k)-f(U_*,V_*)|}{|f(U_*,V_*)|}$, where f is the objective function.

All the plots created and used to support these analyses, along with additional interactive visualizations not included directly in the project, are accessible at [1].

4.1 First Experimental Phase



https://giacomoaru.github.io/CM_Project_Results/first_test_phase_4.html



https://giacomoaru.github.io/CM_Project_Results/first_test_phase_5.html

Figure 2: The left plot illustrates the relationship between the relative error (`error_rel`) and the number of iterations (`n_iteration`), with each point color-coded according to the parameter ϵ. The plot on the right shows the dataset (`dataset`) used on the x-axis and the initialization method (`init_method`) for V_0 on the y-axis. The color of each point represents the difference in norms between the solutions U and V (`U_V_norm_diff`).

4.1.1 Analysis on the Parameter ϵ

The value of ϵ, which represents the minimum tolerance on the difference in the objective function's value between two consecutive iterations, significantly influences the behavior of the tests. In particular, the number of iterations increases noticeably, with a mean value that ranges from 140 to 360 as ϵ decreases from 1e-08 to 1e-15. This observation, based on a weighted average calculated to account for the numerous executions of the MNIST dataset, is also reflected in the left plot of Figure 2, which shows four groups of points, well separated by the diagonal of the graph, with the points marked by smaller ϵ achieving, on average, more iterations and higher precision.

The mean step execution time remain almost identical and do not exhibit any clear or discernible trend by varying ϵ. Furthermore, the value of the objective function can be considered unaffected in magnitude by changes in ϵ. The only real difference being that for smaller ϵ the solution is naturally

more refined and the value of the objective function closer to the real solution of interest. What happens in the last iterations of the algorithm (roughly in the last 150) is in fact that it approaches the solution more and more slowly as convergence is approached.

A peculiar observation arises in the case of MNIST: the distance between the two norms of the matrices U and V is influenced by ϵ. However, this effect does not appear to offer any significant advantage or disadvantage.

Based on these findings, each subsequent test will focus on a single specific value of ϵ. Tests designed to evaluate precision will use ϵ = 1e-15, while others will fix ϵ = 1e-08. The latter choice balances efficiency, as it reduces computation time and resources while facilitating quicker identification of non-convergence issues. Although higher precision values for ϵ could slightly increase the number of iterations, the marginal benefit is outweighed by the added difficulty in detecting potential problems.

4.1.2 Analysis on Initialization Methods

The method used to initialize the matrices U and V exhibits very little, almost negligible, influence on the number of iterations required by our algorithm. It is worth noting, however, that the minimum number of iterations is achieved by the `sketching_g` method, while the maximum number is reached by the `rand_u` and `scaled_u` methods. Interestingly, these maximum values are consistently approximately 10 iterations higher (out of a total of around 160-170) compared to all other methods.

Execution time is similarly only slightly affected by the initialization method, and, as always, it primarily depends on the matrix A and the parameter k. Nevertheless, we observed that the `semi-orth` initialization method exhibits unstable performance, performing the best for the MNIST dataset but the worst for the images in the Weizmann Horse Database. By analyzing the times of individual phases, it becomes evident that this instability arises from the QR factorization and the Backward Substitution processes. Furthermore, we notice that among the other initialization methods for V_0, `rand_n` and `sketching_g` show slightly better performance.

The value of the objective function remains fairly consistent regardless of the selected initialization methods.

When studying the norms of U, V, and their differences, we observe that the methods produce highly variable results as shown in the plot on the right of Figure 2. The `semi-orth` method stands out as the one yielding the largest difference between the norms of U and V. Conversely, `sketching_b` (employing a Bernoulli distribution instead of a Gaussian one) presents a significantly smaller difference between the two norms compared to other methods, followed by `rand_n`. In the second phase of testing, we plan to investigate how the norms of U and V, as well as their difference, impact the performance of our algorithm.

4.1.3 General Insights and Goals of the Next Phase

The observations derived from the first phase of testing have raised interesting questions and doubts that we aim to explore further through an additional suite of tests. Consequently, the analysis of the results from the second testing phase will follow in the next section. Specifically, the aspects we intend to investigate in greater detail are as follows:

- Firstly, we seek to understand how the parameters k, m, and n influence both the number of iterations and the execution time of each step in our algorithm, considering both standard and extreme cases. Secondly, we aim to study whether having `norm_U` $\approx$ `norm_V` confers any advantages, and we intend to push this condition to its extremes to analyze its consequences.

- Another objective is to conduct tests on matrices A of large dimensions. Additionally, we plan to compare our algorithm with existing implementations and off-the-shelf methods to evaluate its relative performance.

- Having observed unusual behaviors in the convergence of the algorithm when A is the identity matrix, we wish to investigate the underlying reasons for this phenomenon and determine whether such behavior changes when A is perturbed.

- Finally, we will study the convergence of the algorithm concerning the global minimum. We also intend to verify whether multiple iterations of the algorithm consistently reach the same local minimum or not.

4.2 Second Experimental Phase

The second experimental phase involves the creation and execution of 7 distinct test cases, each designed to study a specific aspect of the algorithm. These test cases vary in terms of the nature of A and hyperparameter values.

4.2.1 Performance Test

This test is specifically designed to analyze the algorithm's performance as k, the dimensions, and the nature of the matrix A vary. The type of image or initialization of the matrix A strongly influences the number of iterations required by the algorithm. In Figure 3 is possible to observe how the four datasets used are divided into four distinct clusters, which shift to the right along the x-axis based on the characteristic complexity (in this case, the rank of matrix A) of the dataset. It should also be noted the presence of distinct "columns" of points located on the left side of the plot, generated by executions that terminated with small and peculiar numbers of iterations, such as 1, 3, and 4.

https://giacomoaru.github.io/CM_Project_Results/performance_test_1.html

Figure 3: The plot shows how the mean execution time for the optimization step and the total number of iterations required by the algorithm to converge vary with respect to the class of tests. The color values represent the different possible initializations for the matrix A, in particular **blue** for MNIST, **red** for Horses, **violet** for Pokemon and **green** for Normal Distribution Initialization.

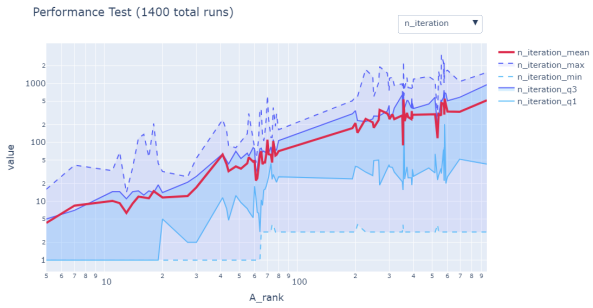
The plot in Figure 4 shows the linear relationship between the rank of the matrix and the number of iterations that emerged from this test, which demonstrates how low-rank matrices require significantly fewer iterations than full-rank matrices, with reductions on the order of two magnitudes, as shown in Table 1. This behavior was observed independently of the dimensions of matrix A, which, in contrast, have a smaller impact on the number of iterations compared to the other parameters.

Dataset	Mean Iteration Number
Pokémon	53
Gaussian	511
MNIST	11
Weizmann Horse Database	96

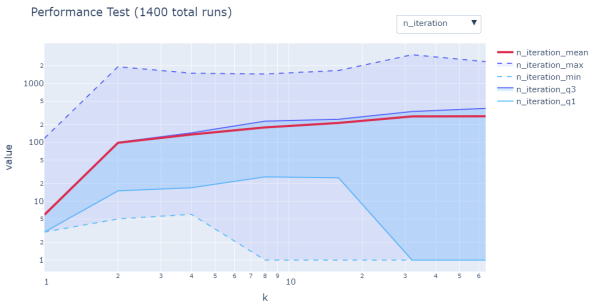
Table 1: Average iteration counts recorded for each dataset. For fair comparison, all dataset images were scaled to have the same dimension as the Weizmann Horse Database.

From the second plot in Figure 4, we can observe the relationship between the size of matrix A and the execution time, which serves as proof of the computational complexity theorized in the previous chapters. Additionally, by studying each phase of the execution step individually, we observed the same linear influence on the execution time.

The number of iterations was also found to depend significantly on the parameter k. Specifically, for $k = 64$, the average number of iterations was 277, while for $k = 8$, the average dropped to 179, and for $k = 1$, the average decreased to just 6 iterations. This relationship is shown in detail by the second plot in Figure 4, which demonstrates a likely sublinear relationship between the two quantities. In fact, although the number of iterations increases significantly during the initial phase of the graph, it tends to stabilize in the final phase. Finally, from the same plot, we can observe that for high values of k,

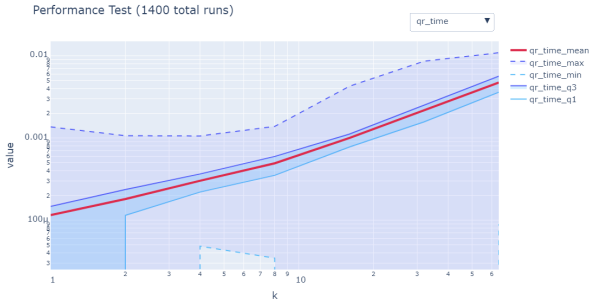


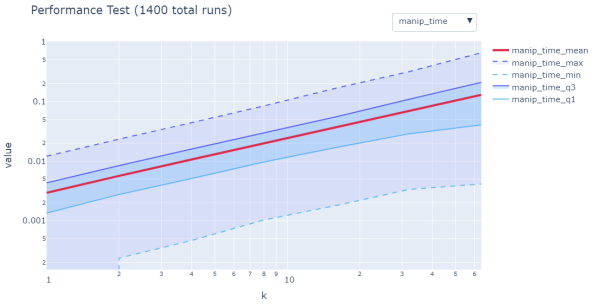
https://giacomoaru.github.io/CM_Project_Results/performance_test_7.html



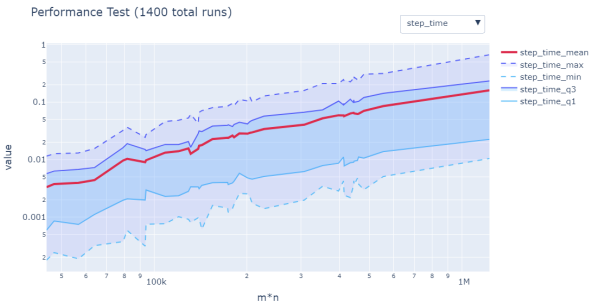
https://giacomoaru.github.io/CM_Project_Results/performance_test_6.html

Figure 4: The plot on the left shows how the number of iteration of the algorithm depends on the rank of the matrix A. The plot on the right shows how the number of iteration of the algorithm depends on the parameter k.





https://giacomoaru.github.io/CM_Project_Results/performance_test_6.html



https://giacomoaru.github.io/CM_Project_Results/performance_test_5.html

Figure 5: The first three plots illustrate how the execution time of the three iteration phases varies with respect to the parameter k. The first plot shows the average time of the initial phase, corresponding to the QR factorization. The second plot represents the average time of the second phase, which involves manipulation, while the third plot displays the average time for the final phase of backward substitution. Finally, the fourth plot shows how the average execution time of a complete iteration changes as the size of the A matrix changes.

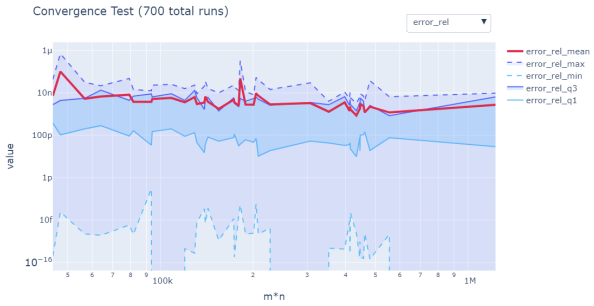
although the average number of iterations is high, many executions terminated with a small number of iterations, which explains the very low first quartile line. This is due to cases where k is very close to the rank of the matrix, as will become more evident in the subsequent tests.

By examining the effect of the parameter k in relation with the execution time, we observe behaviors that align perfectly with the theoretical analysis of our algorithm's time complexity discussed in previous chapters. Specifically, the execution times for QR Factorization and Backward Substitution exhibit a quadratic scaling with respect to k, while the algebraic manipulations show a linear dependency on the same parameter. In the plots reported in Figure 5 we can weakly see the mentioned quadratic dependencies, however the impact of this relationship would become increasingly pronounced as the parameter values grow larger. This result confirms that the parameter k, as indicated in the algorithmic complexity formula, has a greater influence on complexity compared to m and n.

No particular observations were made regarding the dependency between the type of matrix and the execution time of a single step of our algorithm. Instead, the dimensions of matrix A and the value of k emerged as the most influential parameters in determining this characteristic.

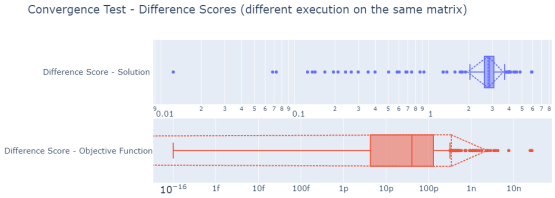
4.2.2 Convergence Test

As described in the previous section, the test involves two analyses: checking the algorithm's consistency across identical runs and comparing its solutions to the global minimum, calculated using truncated SVD via the Eckart-Young theorem. This test was designed to be executed on the images from the Weizmann Horse Database.



https://giacomoaru.github.io/CM_Project_Results/convergence_test_4.html

https://giacomoaru.github.io/CM_Project_Results/convergence_test_5.html



https://giacomoaru.github.io/CM_Project_Results/convergence_test_2.html

Figure 6: The first two plots show the behavior of the relative error as two different quantities vary: the size of the matrix $\mathbf{A}$ and the parameter k. The third plot examines the Difference Scores across different executions on the same matrix. The upper subplot shows the difference score between solutions, computed as $\frac{\|U_1-U_2\|}{\|U_2\|} + \frac{\|V_1-V_2\|}{\|V_2\|}$. while the lower subplot shows the difference score between objective functions, computed as $\frac{|f(U_1,V_1)-f(U_2,V_2)|}{|f(U_2,V_2)|}$.

Our study demonstrated that different executions of the algorithm, starting from the same initial configuration — that is, varying exclusively the initialization of the solution V_0 before computing U in the first optimisation step — still lead to convergence at different critical points. These critical points, while generating very similar values of the objective function, are not themselves similar, with a difference score between solutions that never approaches zero as shown in the plot on the right of Figure 6. When comparing the solution with the global minimum, we observe that our algorithm almost always converges to within a margin smaller than the epsilon set in the test ($\epsilon = 10^{-8}$). This can be observed by analyzing

the curve corresponding to the third quartile. Hence, we can state that, for almost all executions, the algorithm found the global minimum within the precision allowed by epsilon.

By further analyzing the second plot in Figure 6, it can be observed that the relative error increases as the value of k increases. This behavior, while noteworthy, is not surprising, as an increase in k leads to a more complex solution, resulting in a less accurate approximation by the algorithm.

Finally, as can be observed from the graph in Figure 7 that shows the convergence of the algorithm, the curves exhibit an initial phase where the convergence rate varies significantly, ranging from sublinear to superlinear, depending on the specific matrix being approximated. However, after this highly unstable initial phase, all curves eventually settle into a regime of linear convergence at varying speeds. This behavior is likely due to an initial approximation phase, during which the solution remains far from the convergence minimum, resulting in instability. Subsequently, once the solution enters the "attraction basin" of the local minimum, the convergence becomes progressively more stable.

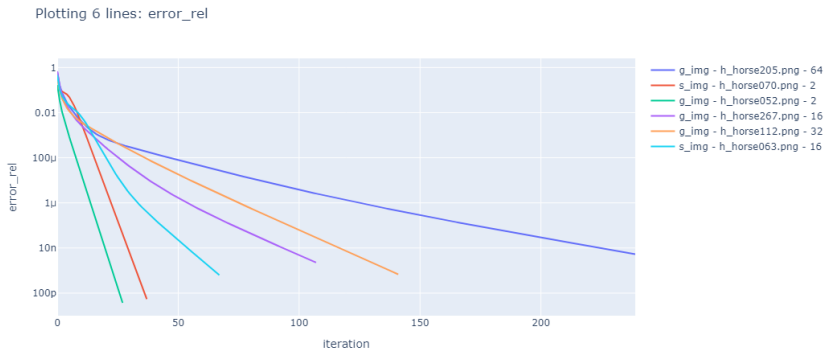


Figure 7: This plot displays six distinct curves representing the relative error of the solution at execution time, sampled from the convergence test. In the legend, the test class, the approximated matrix, and the value of k are shown.

4.2.3 OTS Test

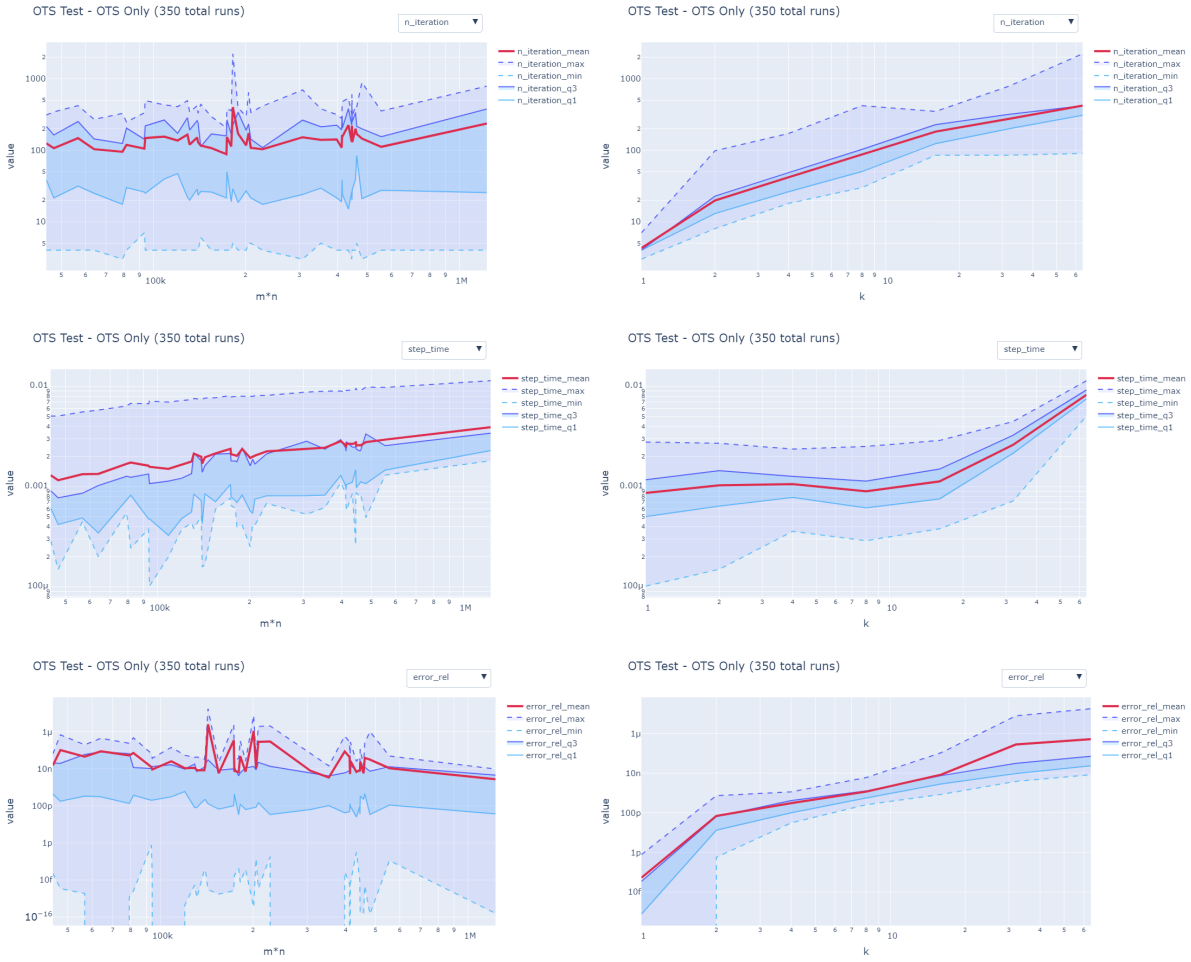
By studying the number of iterations required by our algorithm compared to the off-the-shelf (OTS) implementation, we observed that the average number of iterations for OTS is 157, while our algorithm proves faster in this context, with an average of 75 iterations. In both implementations, the iterations amount depend on k, but while the number of iterations for low k values is comparable, a significant difference emerges at $k = 64$. In this case, our implementation performs better, with an average of only 163 iterations compared to 420 for the OTS implementation. It is also interesting to note that the number of iterations appears to be practically unaffected by either the rank of the matrix or its size, as can be observed in the first plot of Figure 8.

When examining the average execution time per step, an inverse trend is evident: the OTS algorithm achieves an average of 0.0021 seconds per step, whereas our implementation requires an average of 0.038 seconds per step. Execution time per step in both implementations depends on k and the dimensions of matrix A as shown in Figure 8. However, OTS methods show a weaker scaling compared to our implementation. In the worst case, OTS methods recorded a mean `step_time` of 0.011 seconds, while our implementation registered 0.63 seconds, a difference of almost two orders of magnitude.

The operation that most significantly impacts step execution time in our algorithm is algebraic manipulation, which is far less efficient in our implementation due to the way it is computed. While, in a RAM model (which does not account for the hierarchical nature of memory systems or multiple execution streams), the complexity of matrix multiplication would surpass that of algebraic manipulation using Householder vector properties, but in real-world scenarios, where machine vectorization and parallelism are fully leveraged for matrix multiplication, the latter achieves a more advantageous execution time than the former.

QR factorization, on the other hand, demonstrates relatively similar performance in both implementations, with our algorithm even proving slightly more efficient than the OTS implementation.

Therefore, the total execution time exhibits scaling that increases negatively as k and the dimensions of matrix A increase. This effect, which results in longer execution times, is more pronounced in our



https://giacomoaru.github.io/CM_Project_Results/ots_test_4.html

https://giacomoaru.github.io/CM_Project_Results/ots_test_5.html

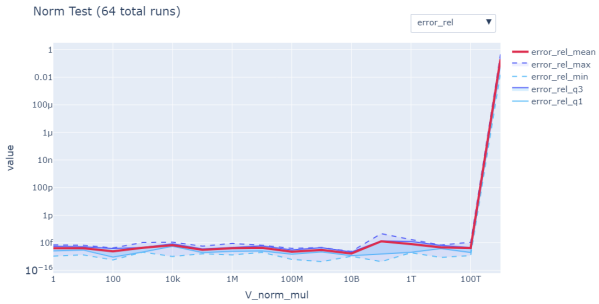
Figure 8: The plots show the relationship between the dimensions of matrix A and the inner dimension of the solutions k, with respect to `error_rel`, number of iterations, and average step time in the off-the-shelf implementation. Notably, there is no clear relationship between $m \times n$ and `error_rel` or the number of iterations. However, $m \times n$ exhibits a linear dependence on step time. The parameter k, on the other hand, shows a sublinear relationship with the number of iterations and `error_rel`, and a superlinear relationship with step time.

implementation than in the OTS one, as can be seen by comparing the plots in Figure 8 with those in Figure 5.

During this test, we also identified a characteristic that warrants further study and investigation. Analysing the relative error respect to the optimal minimum (obtained using truncated SVD), we observed that the OTS implementation occasionally produces poor results, with an `error_rel` mean of 1.361e-07, significantly worse than those for the same case respect to our implementation. Although our algorithm also shows a noticeable performance degradation for higher k, the overall effect is considerably less significant in fact our algorithm achieves a much lower `error_rel` mean of 7.293e-09. Such observations can be made by comparing the last plot in Figure 8 with the corresponding one in Figure 6.

4.2.4 Norm Test

in this test we observed that when V_0 is initialised with a particularly high norm value, the algorithm computes the free matrix with a proportionally smaller norm. Conversely, when V_0 is initialised with a particularly low norm value, the free matrix tends to have a proportionally larger norm as can be clearly seen from Figure 10. Throughout the execution, the norms remain relatively stable, with only

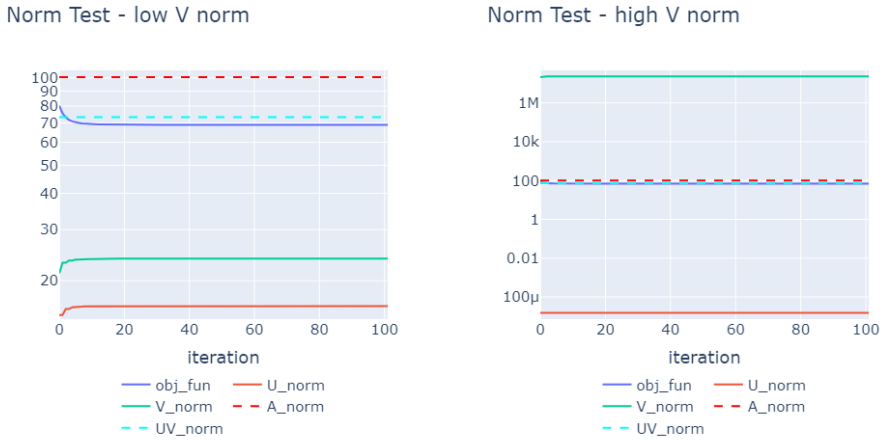


https://giacomoaru.github.io/CM_Project_Results/norm_test_2.html

Figure 9: The plot shows the `error_rel` as a function of the parameter `V_norm_mul`, which represents the coefficient applied to the matrix V to adjust its norm during initialization. This plot highlights that for extreme values of the matrix V norm, the algorithm's becomes inapplicable. The value plotted is equal to one as a consequence of wrong values caused by failure.

minor deviations from their initial values after a brief initial adjustment phase. This behavior reflects the value of the objective function, which changes rapidly during the initial phase of the algorithm and then stabilises as the process progresses. This is because U and V, during the execution, "adjust" to approximate A, and in this process, the norm of $U \cdot V$ approaches the norm of A.

Furthermore, this test demonstrated that the relative error of the solution obtained through the optimization process is independent of the difference between the norms of U and V as can be observed in the plot in Figure 9. This finding holds true with respect to both the number of iterations and the execution time required for each step of the algorithm.



https://giacomoaru.github.io/CM_Project_Results/norm_test_3.html

https://giacomoaru.github.io/CM_Project_Results/norm_test_4.html

Figure 10: In the two plots, we observe the curves generated by the execution of the algorithm with a fixed A under two different conditions: in the first case, where V_0 is initialized with a low norm, and in the second, where V_0 is initialized with a high norm. In both cases, the norm of U adjusts to balance the norm of V, though this behavior is more evident in the second case.

We also observed that in every execution of the algorithm, both solution matrices U and V tend to converge toward optimal values of U and V. Therefore, it might be worth investigating the assumption mentioned earlier regarding the convergence of the individual norms of U and V, in addition to the norm of their product.

As expected, there are extreme cases where increasing the disparity between the norms of U and V beyond sixteen orders of magnitude causes the algorithm to halt after the first iteration, yielding an undefined result. This failure is due to the imbalance being so severe that one matrix effectively becomes null relative to the other (once machine precision is considered), rendering the algorithm inapplicable.

4.2.5 Identity Test

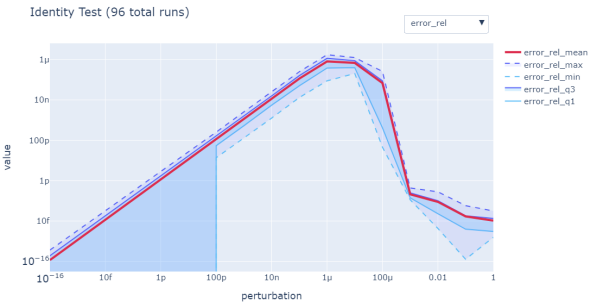
The issue analyzed in this test concerns the algorithm's unusual convergence behavior when A is the identity matrix. In particular, we observed that in such cases, the algorithm completes its execution in just one iteration.

Observing the results of this test shown in Figure 11, we can conclude that the number of iterations required by the algorithm is strongly influenced by the magnitude of the perturbation applied to the identity matrix A. When the matrix A closely resembles the identity, i.e., when the perturbation is smaller than approximately 10^{-8} (depending on k), the algorithm completes in a single iteration, similar to the behavior observed with the unperturbed identity matrix. Conversely, when the error is small but not sufficiently so to fall within the aforementioned case, the algorithm reaches the maximum permitted number of iterations, exhibiting extremely slow convergence toward the minimum. As the perturbation increases and the matrix diverges further from the identity, the number of iterations gradually decreases again, eventually converging to the behavior characteristic of a standard execution. All of these statements remain valid regardless of the value of k.

Additionally, we found that the relative approximation error increase as the magnitude of the perturbation applied to the identity matrix increases, suggesting that as the matrix becomes closer to the identity, the algorithm's accuracy in approximation increases. This behavior changes drastically when the matrix, due to the perturbation, no longer resembles an identity matrix, and the approximation error returns to the expected levels of a normal execution of the algorithm.

We can thus assert that the identity matrix is very easy to approximate using our algorithm. Nevertheless, matrices that are very similar to the identity can lead to such slow convergence that, even after reaching the maximum number of iterations, the quality of the approximation remains poor. This test confirmed that the quality of the approximation depends on the matrix type, with the identity matrix presenting a special case.

https://giacomoaru.github.io/CM_Project_Results/identity_test_1.html



https://giacomoaru.github.io/CM_Project_Results/identity_test_2.html

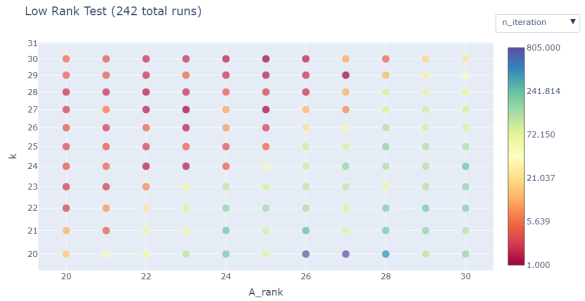
Figure 11: In the plot on the left, the parameter k is shown on the y-axis, and the `perturbation` amount, in particular the σ value representing the variance of the normal distribution exploited in the noising process, is displayed on the x-axis. The color values display how the number of iterations required by the algorithm to reach convergence depend on those values. The plot on the right shows how the `error_rel` changes depending on the `perturbation` amount.

4.2.6 Low Rank Test

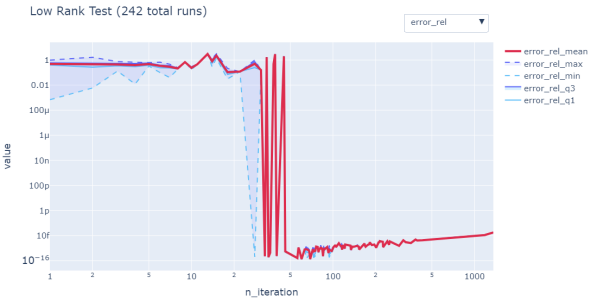
The primary objective of this test is to understand the relationship between the value of k and the quality of the approximation obtained by the algorithm when matrix A is low rank.

We demonstrated that the algorithm completes in a minimal number of iterations (essentially fewer than 5) for all cases where k is greater than the rank of matrix A as shown in Figure 12. Conversely, the average number of iterations required by the algorithm tends to increase as k becomes smaller than the rank of matrix A. In the case where k is exactly equal to the rank of matrix A, we observe both behaviors described above. This dual behavior is likely dependent on the initialization of matrix V.

It is also important to note that the accuracy of the solution, assessed with respect to the global minimum of the problem, remains consistently excellent when the algorithm progresses through a typical number of iterations. In contrast, when the number of iterations is small, the quality of the approximation appears to be poor. This behavior is clearly observable in the plot on the right in Figure 12 and



https://giacomoaru.github.io/CM_Project_Results/low_rank_test_1.html



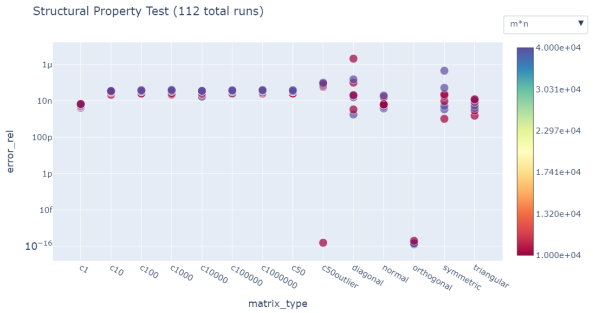
https://giacomoaru.github.io/CM_Project_Results/low_rank_test_2.html

Figure 12: In the plot on the left, the relationship between the parameter k and the rank of the matrix A is illustrated in relation to the number of iterations required for the algorithm to converge (represented by the color values). It is evident that there is a significant variation in the number of iterations, which divides the points into two distinct groups along the diagonal. The plot on the right shows the relation between the relative error and the number of iteration, demonstrating how the solution found by the algorithm, when performing few iterations generates an error value close to one, as the solution achieved by our algorithm reaches a higher precision than the global minimum computed via SVD.

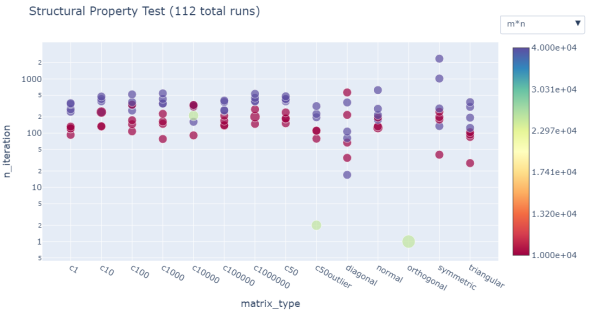
is particularly interesting, especially given that the high error values are very close to 1. A closer examination of the relative error formula reveals that when the approximation provided by our algorithm is more accurate than the one generated by the SVD method, the ratio tends to approach 1 in magnitude. We have observed that the SVD method can at most approximate the solution with a minimum tolerance value of around $1e-13$. Therefore, this behavior can be fully attributed to limitations related to machine precision and the fact that in thi cases our algorithm, with a very low ϵ value, is able to achieve higher precision than the SVD-based method.

4.2.7 Structural Properties Test

Regarding the executions on orthogonal matrix cases, the algorithm demonstrated its ability to achieve a particularly accurate solution in just one iteration, highlighting how the orthogonality of the matrix simplifies its approximation for this approach, is worth to mention that this level of effectiveness was not observed in any other case.



https://giacomoaru.github.io/CM_Project_Results/structural_property_test_1.html



https://giacomoaru.github.io/CM_Project_Results/structural_property_test_2.html

Figure 13: The left plot illustrates the `error_rel` across all conducted tests and matrix type, where each point represents a single test case. The points are color-coded according to the parameter $m \times n$, which denotes the matrix dimensions for the test. The right plot illustrates the same plot but with the number of iteration in the y-axis.

Triangular matrices have demonstrated performance and accuracy levels that, while not identical, are very similar and comparable to those obtained by executing the algorithm on matrices with no interesting properties.

Symmetric and diagonal matrices, on the other hand, exhibited significant instability in both solution accuracy and the number of iterations, displaying a much wider range of values compared to other cases. The distribution of `error_rel` and `n_iteration`, in relation to matrix size, also behaved unexpectedly, with values corresponding to matrices of different sizes not being well-separated. This behavior suggests a strong dependence of the algorithm's performance on the initialization of V_0 or the specific A matrix.

Finally, matrices initialized to have a chosen condition number exhibited very similar performance across the board, except for the matrix with a condition number equal to one, which was easily approximated in a few iterations, and the matrix labeled `c50outlier`. This matrix was uniquely initialized to have a single very large eigenvalue, very far from the others, which were clustered around one. This matrix demonstrated a higher `error_rel` compared to the others, with some executions terminating after a low number of iterations, suggesting a suspicious behavior that certainly warrants further investigation.

The results of this test are well described by the two plots in Figure 13.

References

[1] Giacomo Aru and Simone Marzeddu. *CM Project Results*. Accessed: January 10, 2025. 2025. URL: https://giacomoaru.github.io/CM_Project_Results.

[2] Wikipedia contributors. *Heine–Borel theorem*. Accessed: 2024-12-10. URL: https://en.wikipedia.org/wiki/Heine%E2%80%93Borel_theorem.

[3] Luigi Grippo and Marco Sciandrone. "On the convergence of the block nonlinear Gauss–Seidel method under convex constraints". In: *Operations research letters* 26.3 (2000), pp. 127–136.

[4] Yanjun Zhang and Hanyu Li. *Convergence directions of the randomized Gauss–Seidel method and its extension*. 2021. arXiv: 2105.10615 [math.NA]. URL: https://arxiv.org/abs/2105.10615.