

NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "HSE and University of London Double Degree Programme in Data
Science and Business Analytics"

Programming project report

on the topic Numerical Algorithms of Linear Algebra

Student:
group БПАД232



Sign

З. Ю. Зинкин

Name, Patronymic, Surname

03.06.2025

Date

Supervisor:

Дмитрий Витальевич Трушин

Name, Patronymic, Surname

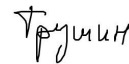
доцент / ФКН НИУ ВШЭ

Position / Company

03.06.2025

Date

Grade



Sign

Contents

1	Introduction	3
1.1	Motivation	3
2	Description of functional and non-functional requirements for the program project	4
2.1	Functional requirements	4
2.2	Non-functional requirements	4
3	Theoretical part	5
3.1	Main definitions	5
3.2	QR Decomposition	7
3.2.1	Householder reflection	7
3.2.2	Givens rotations	8
3.2.3	QR Decomposition algorithm	9
3.3	Schur Decomposition and QR Algorithm	11
3.3.1	Hessenberg Form and Hessenberg QR Decomposition	11
3.3.2	Wilkinson Shifts	12
3.3.3	Real Schur Decomposition	12
3.4	Singular Value Decomposition	13
3.4.1	Bidiagonalization	13
3.4.2	SVD Algorithm	14
4	Library LinAlgTools	15
4.1	Types	15
4.2	Core	16
4.3	Algorithms	16
5	Testing	18
5.1	Validation tests	18
5.2	Benchmark tests	18
5.2.1	Householder vs. Givens performance	19
5.2.2	Real Schur Performance	21
5.2.3	Naive SVD Performance	21
6	Conclusion	22

Abstract

The main focus of the programming project is to study matrix decomposition algorithms from a theoretical point of view with its subsequent implementation in the C++ language.

1 Introduction

Matrix decomposition algorithms serve as fundamental tools in Numerical Linear Algebra and Computer Science, enabling efficient storage of large matrices, eigenvalue computation, solution of linear systems and least squares problems, among other applications. This extensive utility requires both awareness of real-world applications and a profound understanding of the theory and implementation details behind the algorithms.

This programming project studies modern Numerical Linear Algebra algorithms, with particular emphasis on matrix decompositions, which represent a matrix in a certain way, as a product of other matrices, for further computation. In order to have a profound understanding of each algorithm, it is necessary to implement a C++ library from scratch. Thus, the project aims to achieve the following goals:

- Present theoretical material in a concise and accessible manner.
- Develop C++ library with the following functionality:
 - Matrix arithmetic
 - QR Decomposition
 - Hessenberg Form
 - Schur decomposition
 - Bidiagonalization
 - Singular Value Decomposition (SVD)

C++ is the language of choice for the project, since its performance and memory control, along with the specialization in object-oriented design and template system make it suitable for working with matrices.

The core of the implementation revolves around a matrix class that handles matrix arithmetic operations and other methods. All decomposition algorithms build upon this fundamental class. The entire library is tested using GoogleTest[4].

The theoretical part (3) provides mathematical background for each algorithm. Implementation details are documented in the Library LinAlgTools (4) section. Validation and performance tests appear in the Testing (5) section.

The information for the theoretical part of the project is taken from the books, which are cited at the end of the paper. Citation of exact chapters and theorems is used where necessary.

1.1 Motivation

Before proceeding with theoretical and technical details, it is essential to highlight the importance of this work and compare it to existing solutions. While the motivation for learning decomposition algorithms has already been outlined, the motivation behind the technical implementation must also be addressed. Let us first discuss existing libraries with similar functionality.

- *Eigen*[5] and *Intel MKL*[7] - high-performance libraries, supporting dense and sparse matrices, solvers and many decomposition algorithms. However, due to prioritization of performance over readability, the code becomes unreadable. Moreover, the syntax of the libraries can be overwhelming.
- *Armadillo*[10] - a library, aiming towards a good balance between speed and ease of use. Decomposition algorithms are provided through integration with LAPACK[2], which relies on highly optimized implementation. Therefore, the library's implementation may not be clear for an inexperienced user.
- *Dlib*[8] - a machine learning library with readable C++11 code and thorough documentation. However, its age makes it less appealing than Eigen (which updates regularly) for professional use. Due to older C++ standard the library does not utilize powerful additions to the language, such as concepts, to reduce the size of code. More importantly, the source code is flooded with inexpressive variable names, which make it difficult to follow.

As demonstrated, neither of the libraries are capable of providing clear and comprehensive implementation of the algorithms we need. They either prioritize performance at the expense of clarity or suffer from obsolete design choices. Therefore, `LinAlgTools` is developed as a tool, which can be used to properly understand each algorithms' implementation by any C++ amateur.

2 Description of functional and non-functional requirements for the program project

2.1 Functional requirements

- A *Matrix* class, which allows users to perform matrix arithmetic. The class must be able to support both real and complex elements. Moreover, the class must have a clear and intuitive interface.
- A *SubMatrix* and *ConstSubMatrix* classes for handling a particular part of an existing matrix without copying data. These classes must fully replicate the interface of the base *Matrix* class to ensure complete compatibility. Furthermore, cross-type operations must be implemented to perform arithmetic between objects of *Matrix* and *SubMatrix* classes of appropriate sizes.
- Additional utility functions are to be added. They must perform the following functionality: calculate the sign of a real or complex number, distinguish between real and complex types, verify whether a given number is close 0, check if two matrices are equal with a possible error of a pre-determined ε , and more.
- Implementation of numerically stable algorithms:
 - QR Decomposition - representation of a matrix as a product of Q and R matrices, where Q is unitary and R is upper-triangular. The decomposition must be implemented using both Householder reflections and Givens rotations.
 - Hessenberg Form - representation of a matrix as a product of U , H and U^* matrices, where U is unitary and H is upper Hessenberg matrix.
 - Schur Decomposition - representation of a matrix as a product of U , T and U^* , where U is unitary and T is upper-triangular.
 - Bidiagonalization - representation of a matrix as a product of U , B and V^* matrices, where U , V are unitary and B is bidiagonal.
 - SVD Decomposition - representation of a matrix as a product of U , Σ and V^* matrices, where U , V are unitary and Σ is a diagonal matrix.
- Comprehensive testing is essential to ensure the library's reliability. This includes validation on both square and rectangular matrices with real and complex entries, along with tests on potential edge cases.
- Randomized benchmark testing should be performed, generating multiple matrices of specific sizes per iteration. Each test should measure decomposition execution times, recording the mean, minimum, and maximum time in a dedicated file for further analysis.

2.2 Non-functional requirements

- Implementation of the library in the C++ language without third-party libraries, except for Googletest.
- *Git*, a version control system, and a remote repository on *GitHub* [13].
- *Clang* compiler with C++23 standard support.
- Code formatter *clang-format* to automatically format code based on the Microsoft style with additional changes.
- Static code analyzer *clang-tidy* to identify potential errors and to stick to uniform naming of variables, functions, etc.
- A third-party library *Googletest* [4] for writing tests.
- A text editor with IDE capabilities *Neovim*.
- Minimum RAM requirement: 4GB.

3 Theoretical part

Theoretical part is divided into two parts: *Definitions* and *Algorithms*. The first part establishes rigorous framework for the algorithm discussed later. The second part discusses implemented algorithms from a theoretical point of view by providing descriptions, mathematical statements with proofs and pseudocode. Special emphasis is made on proofs, as they not only mathematically validate the algorithms, but often reveal their structure. Presented proofs are adapted versions of proofs from [3] and [9]. In case when the proof does not pose significant value for the algorithm (e.g. only proves existence), it will only be cited. Pseudocode sections deliberately omit edge-cases to focus on core ideas and maintain clarity.

The paper assumes familiarity with Linear Algebra concepts, allowing for the omission of elementary explanations.

3.1 Main definitions

Remark 1. A field \mathbb{F} is denoted by either \mathbb{R} or \mathbb{C} .

Remark 2. A set of all $m \times n$ matrices over a field \mathbb{F} is denoted by $\mathbb{F}^{m \times n}$.

Remark 3. An $n \times n$ Identity matrix is denoted by I_n .

Remark 4. A set of all $n \times 1$ or $1 \times n$ vectors over \mathbb{F} is denoted by \mathbb{F}^n .

Remark 5. The i -th standard basis vector of a vector space \mathbb{F}^m is denoted by $e_i = [0, \dots, \underset{i}{1}, \dots, 0]$

Definition 1. The subdiagonal of a matrix is the set of elements directly below the elements comprising the diagonal.

Definition 2. A matrix $A \in \mathbb{F}^{m \times n}$ is called *upper Hessenberg* if all its elements below the subdiagonal are 0:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1k} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2k} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3k} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{mk} & \cdots & a_{mn} \end{bmatrix}$$

Definition 3. The superdiagonal of a matrix is the set of elements directly above the elements comprising the diagonal.

Definition 4. A matrix $A \in \mathbb{F}^{m \times n}$ is called *upper Bidiagonal* if all its elements below the diagonal and above the superdiagonal are 0:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & a_{23} & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{mk} & a_{mk+1} & \cdots & 0 \end{bmatrix}$$

Definition 5. A *Hermitian conjugate* of a matrix $A \in \mathbb{C}^{m \times n}$ is an $n \times m$ matrix, obtained by transposing A and applying complex conjugate to each of entry: $(A^*)_{ij} = \overline{A_{ji}}$.

Remark. For real matrices *Hermitian conjugate* coincides with transpose of the matrix: $A^* = A^T$.

Definition 6. A matrix $A \in \mathbb{C}^{m \times n}$ is called *Normal* if $A^*A = AA^*$.

Definition 7. A matrix $A \in \mathbb{C}^{m \times n}$ is called *Hermitian* if $A^* = A$.

Definition 8. A matrix $A \in \mathbb{R}^{m \times n}$ is called *Orthogonal* if $A^T = A^{-1}$. As a consequence, $A^T A = AA^T = I$.

Definition 9. A matrix $A \in \mathbb{C}^{m \times n}$ is called *Unitary* if $A^* = A^{-1}$. As a consequence, $A^*A = AA^* = I$.

Statement 1. A product of unitary (orthogonal) matrices is also unitary (orthogonal).

Proof. Let U and V be unitary:

$$(UV)^*UV = V^*U^*UV = V^*V = I$$

□

Definition 10. Matrices $A \in \mathbb{F}^{n \times n}$ and $B \in \mathbb{F}^{n \times n}$ are called *similar* if exists a matrix $C \in \mathbb{F}^{n \times n}$ such that $A = C^{-1}BC$.

Definition 11. A *scalar product* is a positive definite Hermitian form $\langle \cdot, \cdot \rangle : \mathbb{F}^n \times \mathbb{F}^n \rightarrow \mathbb{F}$. The scalar product of two vector $v, w \in \mathbb{C}^n$ is defined in the following way:

$$\langle v, w \rangle = w^* v = \sum_{i=1}^n v_i \overline{w_i}$$

In Euclidean spaces such as \mathbb{R}^m or \mathbb{C}^n we will use the definition above.

Definition 12. A *2-norm*, or *Euclidean norm*, on an Euclidean space is a function: $\|\cdot\|_2 : \mathbb{F}^n \rightarrow \mathbb{R}_+$.

Let $v \in \mathbb{F}^n$:

$$\|v\|_2 = \sqrt{\langle v, v \rangle} = \sqrt{\sum_{i=1}^n |v_i|^2}$$

Remark. Geometrically, a 2-norm returns the distance from the origin to the point in the space.

Definition 13. A matrix is called *sparse* if the majority of its elements are 0.

Definition 14. A matrix is called *dense* if the majority of its elements are non-zero.

Remark. There is no strict threshold to classify a matrix as sparse or dense, but a common convention considers a matrix sparse if the number of non-zero elements is approximately equal to the number of rows or columns.

3.2 QR Decomposition

3.2.1 Householder reflection

Definition 15. A *Householder reflection* is a linear transformation which reflects a vector about a hyperplane[3].

Given a vector $v \in \mathbb{F}^n : \|v\| = 1$, an $n \times n$ matrix P (or more commonly $H(v)$) of the form

$$P = H(v) = I - 2vv^*$$

is called a *Householder reflector* or *Householder matrix*. The matrix allows us to reflect vector in the hyperplane $\text{span}\{v\}^\perp$. Moreover, a Householder matrix is unitary and hermitian ($H^{-1} = H^* = H$), which turns out to be key in the *QR* decomposition algorithm.

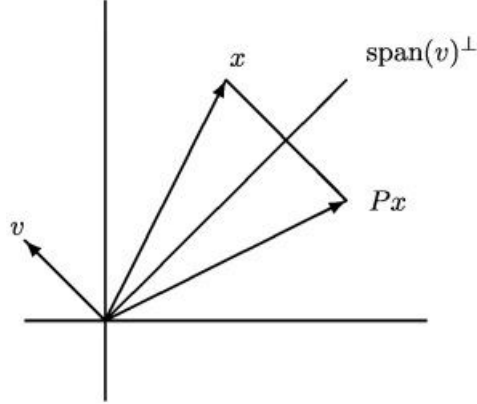


Figure 1: Householder reflection applied to a vector x

Statement 2. $\forall a, b \in \mathbb{F}^n : \|a\|_2 = \|b\|_2, \exists \gamma \in \mathbb{F} : |\gamma| = 1$ and $\exists v \in \mathbb{F}^n : \|v\|_2 = 1$ such that $H(v)a = \gamma b$.

Proof. By direct computation we can find the required γ and v :

$$H(v) = I - 2vv^*$$

$$H(v)a = a - 2vv^*a = \gamma b$$

$$2(v^*a)v = a - \gamma b \tag{1}$$

From here it follows that v must be of the form:

$$v = \mu \frac{a - \gamma b}{\|a - \gamma b\|_2}$$

Since we are proving the existence, we can set μ to 1. Therefore

$$v = \frac{a - \gamma b}{\|a - \gamma b\|_2}$$

Plugging it into (1):

$$\begin{aligned} 2 \frac{(a - \gamma b)^*}{\|a - \gamma b\|_2} a \frac{a - \gamma b}{\|a - \gamma b\|_2} &= a - \gamma b \\ 2(a - \gamma b)^*a &= \|a - \gamma b\|_2^2 \\ 2(a - \gamma b)^*a &= (a - \gamma b)^*(a - \gamma b) \\ 2a^*a - 2\bar{\gamma}b^*a &= a^*a + b^*b - \gamma a^*b - \bar{\gamma}b^*a \\ \bar{\gamma}b^*a &= \text{Re}(\bar{\gamma}b^*a) \\ \gamma &= \pm \frac{b^*a}{|b^*a|} \end{aligned}$$

If $a \perp b$, γ can be any number such that $|\gamma| = 1$. If $a \parallel b$, then we should carefully choose the sign to guarantee numeric stability of the transformation. \square

Corollary 1. $\forall x \in \mathbb{F}^n \exists v \in \mathbb{F}^n : \|v\|_2 = 1$ and

$$H(v)x = \gamma \begin{bmatrix} \|x\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$v = \frac{x - \gamma \|x\|_2 e_1}{\|x - \gamma \|x\|_2 e_1\|_2}, \gamma = -\frac{x_1}{\|x\|_2}$$

The proof and corollary yield an algorithm for vector transformation. To apply this transformation to a matrix from the left (or right), we use:

Algorithm 1 Householder Left Reflection

```

1: function HOUSEHOLDERLEFTREFLECTION( $A \in \mathbb{F}^{m \times n}$ ,  $x \in \mathbb{F}^m$ )
2:    $v = x$ 
3:    $v_1 = x_1 - \text{sign}(x_1) \cdot \|x\|_2$ 
4:    $v = v \cdot \frac{1}{\|v\|}$ 
5:    $M = A - (2v) \cdot (v^* A)$ 
6:   return M
7: end function

```

3.2.2 Givens rotations

Definition 16. A *Givens rotation* is a linear unitary transformation which amounts to a counterclockwise rotation by an angle of θ [3].

In case we need to zero specific elements of the matrix, *Givens rotations* are our tool of choice. The matrix of a *Givens rotation* is of the form ($c = \cos \theta$, $s = \sin \theta$):

$$G(i, j, \theta, \phi) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -se^{-i\phi} & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & se^{i\phi} & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ \end{matrix}$$

Statement 3. $\forall v \in \mathbb{F}^m, i, j \in \mathbb{N}_+ \exists \theta \in \mathbb{F} :$

$$G(i, j, \theta, \phi) \begin{bmatrix} \vdots \\ a \\ \vdots \\ b \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ r \\ \vdots \\ 0 \\ \vdots \end{bmatrix} \begin{matrix} i \\ j \end{matrix}$$

Proof. If $x \in \mathbb{F}^m$ and

$$y = G(i, j, \theta, \phi)x$$

Then:

$$y_j = \begin{cases} cx_i - se^{-i\phi}x_k, & j = i \\ se^{i\phi}x_i + cx_k, & j = k \\ x_j, & j \neq i, k \end{cases}$$

From here we can deduct that $y_k = 0$ if:

$$c = \frac{|x_i|}{\sqrt{|x_i|^2 + |x_k|^2}}, \quad s = \frac{|x_k|}{\sqrt{|x_i|^2 + |x_k|^2}}, \quad e^{i\phi} = \frac{x_k/|x_k|}{x_i/|x_i|}$$

□

Notice that G is almost the identity matrix. We can exploit this structure to achieve faster computation by applying the transformation directly to the rows which are affected by G .

Algorithm 2 Givens Left Rotation

```

1: function GIVENSLEFTROTATION( $A \in \mathbb{F}^{m \times n}$ ,  $i \in \mathbb{N}_+$ ,  $j \in \mathbb{N}_+$ ,  $a \in \mathbb{F}$ ,  $b \in \mathbb{F}$ )
2:    $r = \sqrt{|a|^2 + |b|^2}$ 
3:    $c = a/r$ 
4:    $s = -b/r$ 
5:    $M = A$ 
6:   for  $k = 1$  to  $n$  do
7:      $M_{ik} = \bar{c} \cdot A_{ik} - \bar{s} \cdot A_{jk}$ 
8:      $M_{jk} = s \cdot A_{ik} + c \cdot A_{jk}$ 
9:   end for
10:  return  $M$ 
11: end function

```

3.2.3 QR Decomposition algorithm

The *QR Decomposition* serves as a fundamental tool for solving linear least squares problems and computing matrix eigenvalues. The primary motivation for the implementation of this algorithm is to enable the *QR algorithm* (discussed later). The decomposition's numerical stability and usefulness across algorithms make it irreplaceable for us.

Theorem 1. For any matrix $A \in \mathbb{F}^{m \times n}$ there exists a unitary matrix $Q \in \mathbb{F}^{m \times m}$ and an upper-triangular matrix $R \in \mathbb{F}^{m \times n}$ such that $A = QR$.

Proof. Using Householder reflections (for convenience $A \in \mathbb{F}^{4 \times 3}$):

$$A = \begin{bmatrix} \star & \star & \star \\ \star & \star & \star \\ \star & \star & \star \\ \star & \star & \star \end{bmatrix} \xrightarrow{H(v_1)} \begin{bmatrix} \star & \star & \star \\ 0 & \star & \star \\ 0 & \star & \star \\ 0 & \star & \star \end{bmatrix} \xrightarrow{H(v_2)} \begin{bmatrix} \star & \star & \star \\ 0 & \star & \star \\ 0 & 0 & \star \\ 0 & 0 & \star \end{bmatrix} \xrightarrow{H(v_3)} \begin{bmatrix} \star & \star & \star \\ 0 & \star & \star \\ 0 & 0 & \star \\ 0 & 0 & 0 \end{bmatrix} = R$$

$$H_3 H_2 H_1 A = R$$

$$A = H_1^{-1} H_2^{-1} H_3^{-1} R = H_1 H_2 H_3 R = QR$$

□

Algorithm 3 Householder QR Decomposition

```

1: function HOUSEHOLDERQR( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(4mn^2 - \frac{4}{3}n^3)$ 
2:    $Q = I_m$ 
3:    $R = A$ 
4:   for  $k = 1$  to  $\min(m, n)$  do
5:      $v = R[k : m, k]$ 
6:      $R[k : m, k : n] = \text{HouseholderLeftReflection}(R[k : m, k : n], v)$ 
7:      $Q[k : m, 1 : m] = \text{HouseholderLeftReflection}(Q[k : m, 1 : m], v)$ 
8:   end for
9:    $Q = Q^*$ 
10:  return  $Q, R$ 
11: end function

```

Proof. Using Givens rotations (for convenience $A \in \mathbb{F}^{3 \times 2}$):

$$A = \begin{bmatrix} \star & \star \\ \star & \star \\ \star & \star \end{bmatrix} \xrightarrow{G_{13}} \begin{bmatrix} \star & \star \\ \star & \star \\ 0 & \star \end{bmatrix} \xrightarrow{G_{12}} \begin{bmatrix} \star & \star \\ 0 & \star \\ 0 & \star \end{bmatrix} \xrightarrow{G_{23}} \begin{bmatrix} \star & \star \\ 0 & \star \\ 0 & 0 \end{bmatrix} = R$$

$$G_{23}G_{12}G_{13}A = R$$

$$A = G_{13}^*G_{12}^*G_{23}^*R = QR$$

□

Algorithm 4 Givens QR Decomposition

```

1: function GIVENSQR( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(6mn^2 - \frac{5}{2}n^3)$ 
2:    $Q = I_m$ 
3:    $R = A$ 
4:   for  $k = 1$  to  $\min(m, n)$  do
5:     for  $i = m$  downto  $k + 1$  do
6:        $a = R[i - 1, k]$ 
7:        $b = R[i, k]$ 
8:       if  $b \neq 0$  then
9:          $R[1 : m, k : n] = \text{GivensLeftRotation}(R[1 : m, k : n], i - 1, i, a, b)$ 
10:         $Q[1 : m, 1 : m] = \text{GivensRightRotation}(Q[1 : m, 1 : m], i - 1, i, a, b)$ 
11:       end if
12:     end for
13:   end for
14:   return  $Q, R$ 
15: end function

```

While Householder reflections appear superior to Givens rotations due to better computational complexity, it is not always the case. Givens rotations differ by zeroing specific elements of the matrix, making them particularly effective for sparse matrices or matrices nearly upper-triangular (see [1]). Conversely, Householder reflections demonstrate better results on dense matrices through efficient full-column reductions.

Consequently, each approach has its own distinct advantages and limitations that must be taken into account when working with structured matrices.

3.3 Schur Decomposition and QR Algorithm

We now turn to the *Eigenvalue Problem* - the computation of a matrix's eigenvalues and corresponding eigenvectors.

Theorem 2. (*Schur*) $\forall A \in \mathbb{C}^{n \times n}$ there exists a unitary $U \in \mathbb{C}^{n \times n}$ and an upper-triangular $T \in \mathbb{C}^{n \times n}$ such that $A = UTU^*$.

Proof. Theorem 7.1.3 [3]. □

However, the proof does not provide an algorithm, making it necessary to introduce a computational method, which enables this decomposition and subsequent SVD. The eigenvalue problem is either computationally inefficient or numerically unstable. These problems suggest to shift our attention to iterative algorithms and examine their convergence. In the following references, convergence will be understood in the sense of Frobenius norm convergence.

Algorithm 5 QR Algorithm

```

1: function ALGORITHMQR( $A \in \mathbb{F}^{n \times n}$ )
   Complexity:  $\mathcal{O}(n^4)$ 
2:    $A_0 = A$ 
3:   for  $k = 1, 2 \dots$  do:
4:      $A_{k-1} = Q_k R_k$ 
5:      $A_k = R_k Q_k$ 
6:   end for
7:   return  $A_k$ 
8: end function

```

Definition 17. The *QR Algorithm*, when applied to a square real symmetric or normal matrix A , converges to an upper-triangular matrix T from the Schur decomposition.

Proof. The general convergence proof is quite complicated (see [12]), but a simpler version for normal matrices can be found in [11]. □

While the above pseudocode represents the algorithm's simplest form, its convergence is often slow or may fail entirely. Two particularly effective improvements are *matrix preprocessing* and introduction of *shifts*. The latter ensures convergence with at worst cubic complexity $\mathcal{O}(n^3)$, where n denotes the size of the matrix.

3.3.1 Hessenberg Form and Hessenberg QR Decomposition

As previously mentioned, matrix preprocessing can dramatically improve the algorithm's computational efficiency. In modern practice, transforming the matrix to *Hessenberg form* goes before applying the QR algorithm.

Statement 4. $\forall A \in \mathbb{F}^{n \times n}$ there exists a unitary $U \in \mathbb{F}^{n \times n}$ and an upper Hessenberg $H \in \mathbb{F}^{n \times n}$ such that $A = UHU^*$.

Proof. The proof is identical to the the Householder QR, but we zero out the elements below the subdiagonal. □

Algorithm 6 Hessenberg form

```

1: function HESSENBERGFORM( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(2mn^2 - \frac{2n^3}{3})$ 
2:    $U = I_n, H = A$ 
3:   for  $k = 1$  to  $n$  do
4:      $v = H[k+1 : n, k]$ 
5:      $H[k+1 : n, k : n] = \text{HouseholderLeftReflection}(H[k+1 : n, k : n], v)$ 
6:      $H[1 : n, k+1 : n] = \text{HouseholderRightReflection}(H[1 : n, k+1 : n], v)$ 
7:      $U[k+1 : n, 1 : n] = \text{HouseholderLeftReflection}(U[k+1 : n, 1 : n], v)$ 
8:   end for
9:    $U = U^*$ 
10: end function

```

Since A and H are similar, their eigenvalues are identical, allowing for the QR algorithm to operate on the Hessenberg structure with improved stability and convergence. Additionally, the Hessenberg form reduces the QR Decomposition complexity to $\mathcal{O}(n^2)$ by limiting elimination to elements below the subdiagonal.

Algorithm 7 Hessenberg QR Decomposition

```

1: function HESSENBERGQR( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(3n^2)$ 
2:    $Q = I_m$ 
3:    $R = A$ 
4:   for  $k = 1$  to  $\min(m, n)$  do
5:      $a = R[i, j]$ 
6:      $b = R[i + 1, j]$ 
7:      $R[1 : m, k : n] = \text{GivensLeftRotation}(R[1 : m, k : n], i, i + 1, a, b)$ 
8:      $Q[1 : m, 1 : m] = \text{GivensRightRotation}(Q[1 : m, 1 : m], i, i + 1, a, b)$ 
9:   end for
10:  return  $Q, R$ 
11: end function

```

3.3.2 Wilkinson Shifts

An alternative approach for improving convergence, and in some cases enabling convergence to exact solutions, is the use of *Wilkinson Shifts*. These shifts choose near-diagonal eigenvalue estimates, which avoid stagnation by breaking eigenvalue clusters for non-symmetric matrices. An example of such acceleration is algorithm 8, discussed below. More information about their effectiveness can be found the QR algorithm's convergence proof [12].

Definition 18. Given a matrix $A \in \mathbb{R}^{n \times n}$:

$$A = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdots & a & b \\ \cdots & c & d \end{bmatrix} \quad \delta = \frac{a + d}{2}$$

A Wilkinson shift is a number:

$$\mu = d - \frac{\text{sign}(\delta)b^2}{|\delta| + \sqrt{\delta^2 + b^2}}$$

3.3.3 Real Schur Decomposition

Theorem 3. $\forall A \in \mathbb{R}^{n \times n}$ there exists an orthogonal $Q \in \mathbb{R}^{n \times n}$ such that:

$$Q^T A Q = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ 0 & R_{21} & \cdots & R_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{mm} \end{bmatrix}$$

where each R_{ii} is either a 1×1 or a 2×2 matrix having complex conjugate eigenvalues.

Proof. Theorem 7.4.1 [3] □

We may therefore regard the Real Schur Decomposition as an *Eigenvalue-Revealing Decomposition*, since the real and imaginary parts of the eigenvalues can be extracted directly from the diagonal blocks R_{ii} .

Algorithm 8 Real Schur Decomposition

```

1: function REALSCHUR( $A \in \mathbb{R}^{n \times n}$ )
   Complexity:  $\mathcal{O}(n^3)$ 
2:    $U_0, S_0 = \text{HessenbergForm}(A)$ 
3:   for  $k = 1, 2 \dots$  do:
4:      $\alpha = \text{GetWilkinsonShift}(S_{k-1})$ 
5:      $Q_k, R_k = \text{HouseholderQR}(S_{k-1} - \alpha I_n)$  ▷ Utilizes HessenbergQR if structure is detected
6:      $S_k = R_k Q_k + \alpha * I_n$ 
7:      $U_k = U_k Q_k$ 
8:   end for
9:   return  $U_k, S_k$ 
10: end function

```

3.4 Singular Value Decomposition

Theorem 4. $\forall A \in \mathbb{F}^{m \times n}$ there exists a unitary $U \in \mathbb{F}^{m \times m}$, $V \in \mathbb{F}^{n \times n}$ and a diagonal $\Sigma \in \mathbb{R}^{m \times n}$ such that $A = U\Sigma V^*$.

Proof. The proof of existence appears in Lecture 3 of [9]. □

The decomposition above is called *Singular Value Decomposition*. The columns of U are called *left singular vectors* and the columns of V are called *right singular vectors*. Σ always has real values on its diagonal, and they are called *singular values*. Moreover, if A has rank r , then singular values satisfy: $\sigma_1 \geq \sigma_2 \geq \dots \sigma_r > 0$ and:

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix}$$

The naive algorithm calculates $A^T A$ and finds its eigenvalues. However, that procedure is unstable and impractical. Let us discuss ways to improve it.

3.4.1 Bidiagonalization

Statement 5. $\forall A \in \mathbb{F}^{m \times n}$ there exists a unitary $U \in \mathbb{F}^{m \times m}$, $V \in \mathbb{F}^{n \times n}$ and a bidiagonal $B \in \mathbb{F}^{m \times n}$ such that $A = UB V^*$

Proof. For convenience, $A \in \mathbb{F}^{5 \times 4}$ (U_i and V_i are Householder Left Reflections and Right Reflections accordingly):

$$\begin{aligned}
& \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{V_1} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \\
& \xrightarrow{U_2} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \xrightarrow{V_2} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & * \end{bmatrix} \xrightarrow{U_4} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \end{bmatrix}
\end{aligned}$$

□

Algorithm 9 Bidiagonalization

```
1: function BIDIAGONALIZATION( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(4mn^2 - \frac{4n^3}{3})$ 
2:    $U = I_m, V = I_n, B = A$ 
3:   for  $k = 1$  to  $\min(m, n)$  do
4:      $v_1 = B[k : m, k]$ 
5:      $B[k : m, k : n] = \text{HouseholderLeftReflection}(B[k : m, k : n], v_1)$ 
6:      $U[k : m, 1 : m] = \text{HouseholderLeftReflection}(U[k : m, 1 : m], v_1)$ 
7:     if  $k < n - 1$  then
8:        $v_2 = B[k, k + 1 : n]$ 
9:        $B[k : m, k + 1 : n] = \text{HouseholderRightReflection}(B[k : m, k + 1 : n])$ 
10:       $V[1 : n, k + 1 : n] = \text{HouseholderRightReflection}(V[1 : n, k + 1 : n])$ 
11:     end if
12:   end for
13:    $U = U^*, V = V^*$ 
14:   return  $U, B, V$ 
15: end function
```

3.4.2 SVD Algorithm

Implementation of the SVD utilizes two aforementioned techniques: preprocessing by bidiagonalization and alternating QR Algorithm. The former transforms a matrix A into bidiagonal matrix B , which is then exploited by applying QR Decomposition first to B , then takes the factor from the decomposition R and applies another QR Decomposition to R^T . This procedure converges to a diagonal matrix Σ from the SVD.

Algorithm 10 Naive SVD

```
1: function NAIVESVD( $A \in \mathbb{F}^{m \times n}$ )
   Complexity:  $\mathcal{O}(14mn^2 + 10n^3)$ 
2:    $U, S, V = \text{Bidiagonalization}(A)$ 
3:    $V = V^*$ 
4:   for  $k = 1, 2 \dots$  do
5:      $Q_1, R_1 = \text{HouseholderQR}(S)$ 
6:      $Q_2, R_2 = \text{HouseholderQR}(R_1^*)$ 
7:      $S = R_2^*$ 
8:      $U = UQ_1$ 
9:      $V = VQ_2$ 
10:  end for
11:   $V = V^*$ 
12:   $\text{SortSingularValues}(U, S, V)$ 
13:  return  $U, S, V$ 
14: end function
```

Algorithm presented above is numerically stable, since it utilizes Bidiagonalization and Householder QR Decomposition. Moreover, it does not explicitly construct $A^T A$. However, this simplicity comes at a cost of convergence, which will be illustrated in the Performance section.

An improved SVD algorithm utilizes such techniques as *Bidiagonal QR Algorithm*, *shifts* and *splitting the matrix*. These complications appear in *Golub-Kahan SVD Algorithm* (see [3]). This implementation, numerically stable and computationally fast, is used in libraries like Numpy[6] and Eigen[5].

4 Library LinAlgTools

The library is designed with an extensible architecture to efficiently handle decomposition algorithms. It has the following architecture:

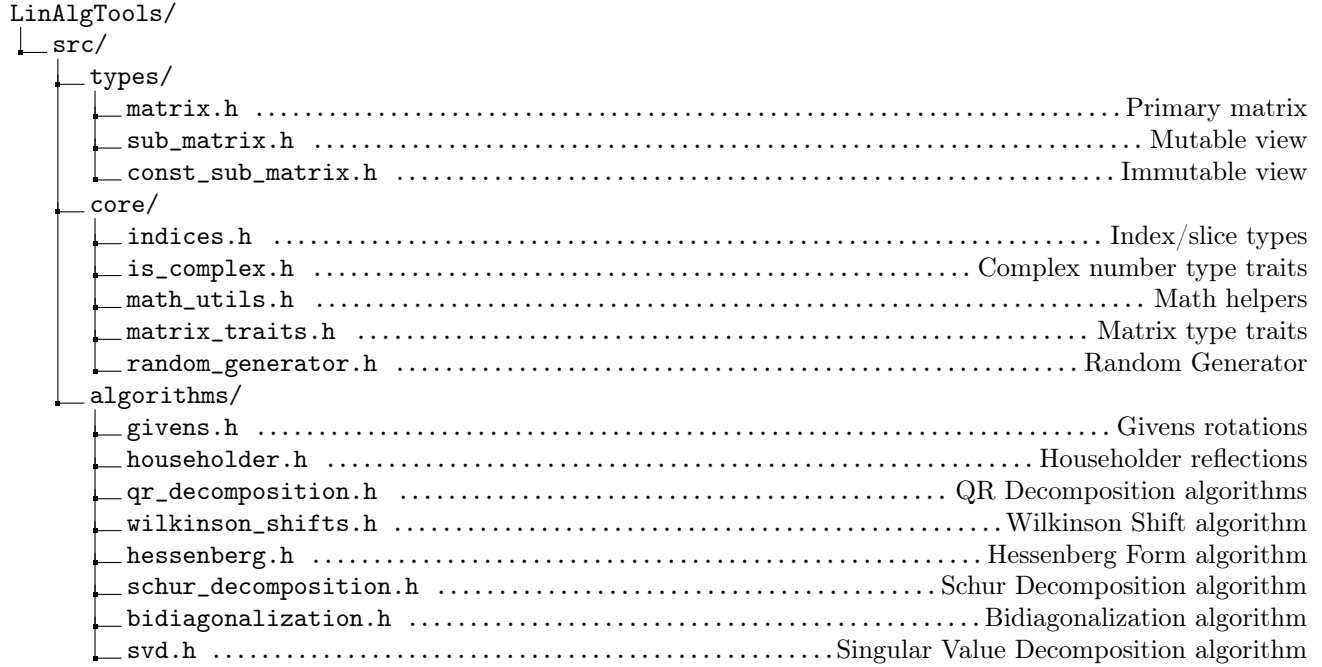


Figure 2: Directory structure of the LinAlgTools library

4.1 Types

This part of the library contains the `Matrix` class and its derivatives: `SubMatrix` and `ConstSubMatrix` classes.

- The `Matrix` class makes it possible to construct matrices with various sizes and element types (C++ fundamental types or `std::complex`), supporting full matrix arithmetic. It implements all required methods for algorithm functionality along with additional utilities for basic linear algebra operations. Furthermore, through `concepts`, the class supports cross-type arithmetic operations, allowing for multiplication/addition/-subtraction between `matrix` and `submatrix` objects.

```
1  LinAlgTools::Matrix<int> matrix = {{1, 2}, {3, 4}};
2  std::cout << matrix; //[1, 2], [3, 4]
3
4  LinAlgTools::Matrix<int> sq_matrix = {2};
5  std::cout << sq_matrix; //[0, 0], [0, 0]
6
7  LinAlgTools::Matrix<int> identity = LinAlgTools::Matrix<int>::Identity(2);
8  std::cout << identity; //[1, 0], [0, 1]
```

- The `SubMatrix` and `ConstSubMatrix` classes allow to perform operations with a particular part of the matrix without copying data. This proves to be particularly useful in our case, enabling application of transformations to a submatrix, greatly reducing the number of unnecessary computations. The classes also allow to create an object of a `matrix` class.

```
1  LinAlgTools::Matrix<int> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
2  LinAlgTools::SubMatrix<int> submatrix = {matrix, {0, 1}, {0, 1}};
3  std::cout << submatrix; //[1, 2], [4, 5]]1
4
5  submatrix *= 5;
6  std::cout << matrix; //[5, 10, 3], [20, 25, 6], [7, 8, 9]];
```

4.2 Core

The entirety of this part of the library is dedicated towards providing useful functions or other functionality, which is used throughout the project.

- **Math_utils** provide the ability to calculate the sign of a real or a complex number, check whether a matrix is diagonal, orthogonal and much more. **Indices** store uniform types of matrix entry index, row and column slices. This allows to easily change the required type if needed, and the changes will take place throughout the whole library.

```
1 LinAlgTools::Matrix<double> matrix = {{1e-16, 1e-16}};  
2 LinAlgTools::Matrix<double> zero_matrix = {{0, 0}};  
3 std::cout << LinAlgTools::AreEqualMatrices(matrix, zero_matrix); //true
```

- **Matrix_traits** and **is_complex** are used to handle different matrix types and complex number respectively. Concepts, mentioned earlier, were introduced in C++20 and are a tool, allowing for uniform templates for any matrix class.

```
1 template<Core::MatrixType F, Core::MatrixType S>  
2 Matrix<typename F::ElementType> operator+(const F& lhs, const S& rhs) {  
3     \*Implementation*\  
4 }
```

- **Random_generator** provides the ability to generate random values of all types (fundamental C++ types and `std::complex`). Moreover, it also allows to generate dense and sparse matrices.

```
1 LinAlgTools::Core::RandomGenerator rnd_gen(20);  
2 int rnd_int = rnd_gen.GetRandomTypeValue<int>(-5, 5);  
3 LinAlgTools::Matrix<double> = rnd_gen.GetRandomDenseMatrix<double>(5, 5, 1e-10, 1e+10);
```

4.3 Algorithms

This part of the library contains the implementation of the algorithms, which were extensively discussed above. Majority of algorithms' subroutines are implemented as separate functions and can be used as linear transformations. Moreover, the implementation of the algorithms exactly matches its pseudocode counterpart for preserving cohesion and level of abstraction.

- **Householder** contains **HouseholderVectorReduction** for transforming a vector, zeroing out each elements below the first first one. **HouseholderLeftReflection** and **HouseholderRightReflection**, provided a vector, are used to apply a reflection to a matrix.
- **Givens** contains **GetGivensPair**, which calculates coefficients for further rotation. **GivensLeftRotation** and **GivensRightRotation**, provided a vector, are used to apply a rotation to a matrix.
- **QR_Decomposition** contains two implementations of the decomposition: one based on Householder reflections and another based on Givens rotations. The former utilizes **HouseholderVectorReduction** and **HouseholderLeftReflection** to transform the matrix to the required form. Reflections are calculated in-place and to a particular submatrix to avoid unnecessary computations and copies. The latter uses **GetGivensPair**, **GivensLeftRotation** and **GivensRightRotation** to calculate matrices Q and R . Left rotations are applied to submatrices of R , while right rotations are applied to submatrices of Q .

A third implementation, **HessenbergQR**, is also provided. Given its conceptual similarity to **GivensQR** and the comparison already presented in the theoretical section, we omit further detailed discussion here.

```
1 LinAlgTools::Matrix<double> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
2 auto [Q_1, R_1] = LinAlgTools::Algorithm::HouseholderQR(matrix);  
3 auto [Q_2, R_2] = LinAlgTools::Algorithm::GivensQR(matrix);  
4 std::cout << LinAlgTools::Core::AreEqualMatrices(Q_1 * R_1, matrix); //true
```

- **HessenbergForm** contains the implementation of the eponymous algorithm. Though similar to Householder QR Decomposition, it applies Householder Reflections starting with the second row. Moreover, **HouseholderRightReflections** are also applied to preserve similarity of the matrix.


```

1  LinAlgTools::Matrix<double> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
2  auto [U, H] = LinAlgTools::Algorithm::HessenbergForm(matrix);
3  auto reconstructed = U * H * U.ConjugateTranspose();
4  std::cout << LinAlgTools::Core::AreEqualMatrices(reconstructed, matrix); //true

```

- **Wilkinson_Shift** contains the implementation of the Wilkinson Shifts. Functions **GetWilkinsonShift**, given a matrix, calculates the shift.
- **Schur_Decomposition** contains the implementation of the **RealSchur** decomposition algorithm. The decomposition utilizes **HessenbergForm** to preprocess the matrix, transforming it into Hessenberg Form, which in return significantly increases computational speed. **GetWilkinsonShift** is used on every iteration to accelerate convergence and allow for convergence in specific cases.

The loop of the algorithm breaks when either S is upper-triangular or when the number of iterations exceeds a pre-determined value to avoid infinite loops.

```

1  LinAlgTools::Matrix<double> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
2  auto [U, S] = LinAlgTools::Algorithm::RealSchur(matrix);
3  auto reconstructed = U * S * U.ConjugateTranspose();
4  std::cout << LinAlgTools::Core::AreEqualMatrices(reconstructed, matrix); //true

```

- **Bidiagonalization** contains the implementation of eponymous algorithm. Implementation of **Bidiagonalization** reminds of **HouseholderQR**, which is not a coincidence given conceptual similarity. **HouseholderLeftReflection** are used to eliminate entries below the diagonal of matrix B , and transformations are then applied to matrix U . **HouseholderRightReflection** are used to eliminate entries above the superdiagonal of B , transformations are also applied to matrix V .

```

1  LinAlgTools::Matrix<double> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
2  auto [U, B, VT] = LinAlgTools::Algorithm::Bidiagonalization(matrix);
3  auto reconstructed = U * B * VT;
4  std::cout << LinAlgTools::Core::AreEqualMatrices(reconstructed, matrix); //true

```

- **SVD** contains the implementation of Singular Value Decomposition. Function **NaiveSVD** preprocesses the matrix by bidiagonalizing it. Afterwards, alternating **HouseholderQR** is applied to matrix S , then to its transpose from QR Decomposition, R . All transformations are also applied to matrices U and VT . Moreover, at the end singular values are sorted in non-descending order.

The loop of the algorithm breaks when either S is diagonal or when the number of iterations exceeds a pre-determined value to avoid infinite loops.

```

1  LinAlgTools::Matrix<double> matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
2  auto [U, B, VT] = LinAlgTools::Algorithm::NaiveSVD(matrix);
3  auto reconstructed = U * B * VT;
4  std::cout << LinAlgTools::Core::AreEqualMatrices(reconstructed, matrix); //true

```

5 Testing

The library utilizes a testing framework to validate correctness and measure performance. The test architecture is organized as follows:

```
LinAlgTools/  
├── tests/  
│   ├── benchmark_tests/  
│   │   ├── benchmark_utils/  
│   │   │   └── algorithm_benchmark.h  
│   │   └── decomposition_benchmark_tests.cpp  
│   └── validation_tests/  
│       ├── matrix_tests.cpp  
│       ├── sub_matrix_tests.cpp  
│       ├── const_sub_matrix_tests.cpp  
│       ├── qr_decomposition_tests.cpp  
│       ├── hessenberg_form_tests.cpp  
│       ├── schur_decomposition_tests.cpp  
│       ├── bidiagonalization_tests.cpp  
│       └── svd_tests.cpp
```

Figure 3: Directory structure of the **Tests** section

5.1 Validation tests

The majority of implemented functions, classes, and algorithms in the library are tested across multiple test cases. All methods of all classes have been tested on hand-crafted tests. Other certain functions that form part of a larger algorithm are not individually tested, as the correct of the algorithm inherently verifies their functionality. The reliability of the library is further validated using Googletest [4].

All tests related to matrices follow the general structure:

- Test on square matrices.
- Test on rectangular matrices.
- Test on matrices with complex coefficients.
- Test on matrices with ε (small) coefficients.
- Test on potential edge-cases.

These tests verify the correctness of the implementation, ensuring that all functions work properly with square or rectangular matrices with real or complex coefficients.

In addition to these tests, decomposition algorithms are additionally tested using:

- Comparison of the reconstructed matrix to the original, ensuring that the error stays within the range of machine error bounds.
- Verification of decomposition correctness by confirming matrices exhibit the expected structure.

The final validation step is typically implemented as a dedicated function, named `Check{AlgorithmName}`.

5.2 Benchmark tests

All decomposition algorithms are tested using randomly generated matrices. For each matrix size within the range (from minimum to maximum), every algorithm performs the decomposition on 10 dense complex (if possible) matrices. The following performance metrics are recorded in a dedicated CSV file (`algorithm_performance.csv`):

- Matrix size.
- Mean execution time.
- Minimum execution time.
- Maximum execution time.
- Standard deviation of timings.

The benchmarking hyper-parameters are fully configurable:

- Minimum and maximum matrix sizes.
- Number of test matrices per size iteration.
- Size increment step between tests.

Remark. The benchmark tests have been conducted on a MacBook Pro with Apple M2 Pro chip and 32GB of RAM.

5.2.1 Householder vs. Givens performance

The same seed is used in the random number generator to ensure correctness of the comparisons.

As previously stated, this configuration is perfect for Householder implementation, and the empirical data supports this claim. Figure 4 demonstrates that as matrix size grows, the mean execution time of the Givens implementation becomes greater than that of the Householder method, which is an expected result given its higher computational complexity. Figure 5 strengthens this observation by illustrating the ratio of Givens mean execution time to Householder for each matrix size. However, the performance gap narrows with greater matrix size, though the limited sample size prevents us from drawing conclusions about whether this trend persists for larger matrices.

The theoretical part also highlighted Givens advantage for sparse matrices, but empirical testing failed to validate this claim, so the corresponding data is not presented. The current Givens implementation does not fully exploit the structure of a sparse matrix, requiring further optimizations, such as those found in *Numpy* [6]. Since such optimizations were beyond the scope of this project, they were not implemented into LinAlgTools. Nevertheless, the project's scalable design allows for future enhancements to better handle structured matrices.

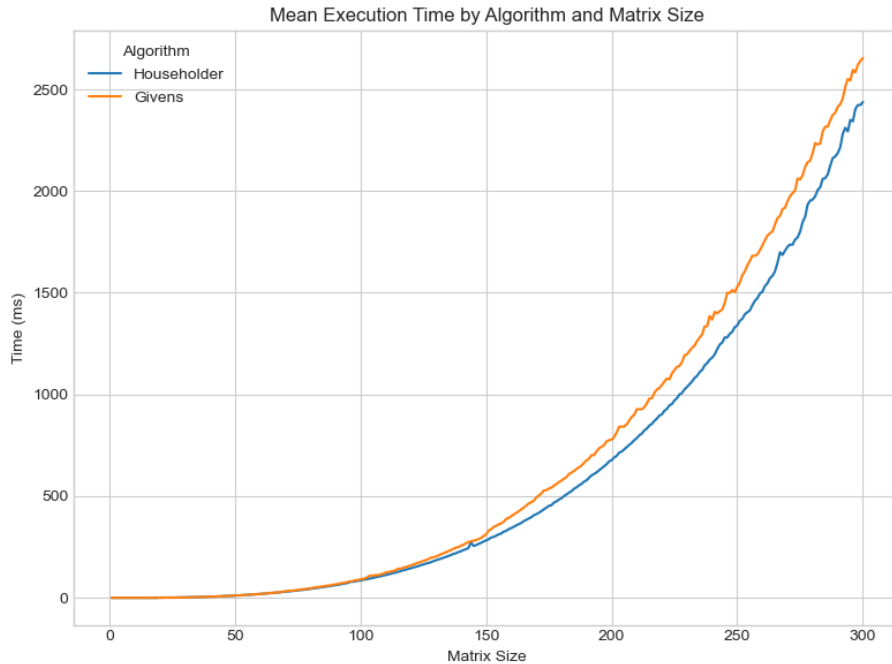


Figure 4: Mean execution time by algorithm and matrix size

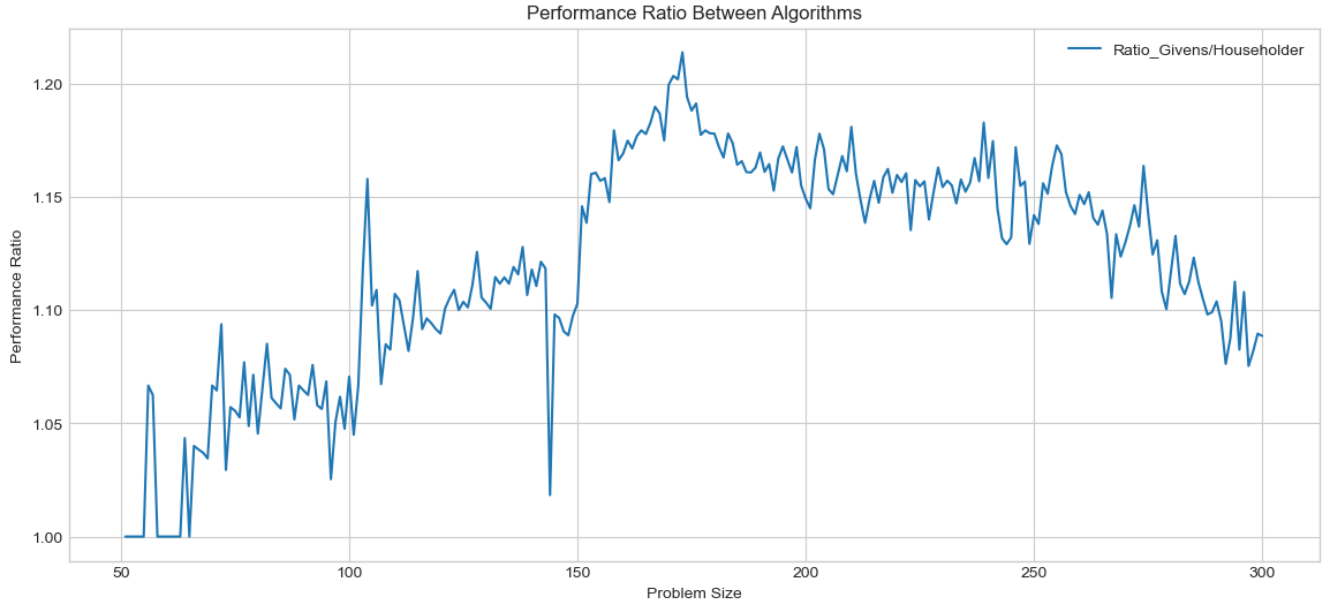


Figure 5: Performance ratio between GivensQR and HouseholderQR

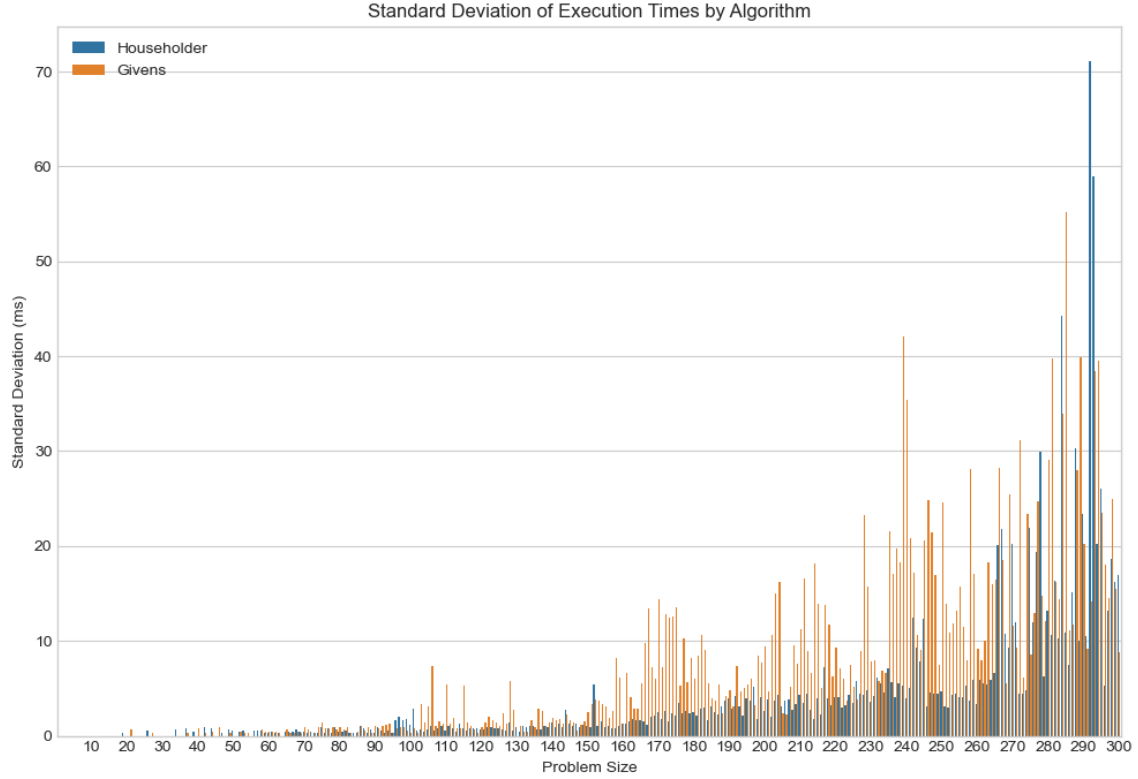


Figure 6: Standard deviation of execution times by algorithm

5.2.2 Real Schur Performance

The figure below illustrates the performance of the Real Schur Decomposition Algorithm.

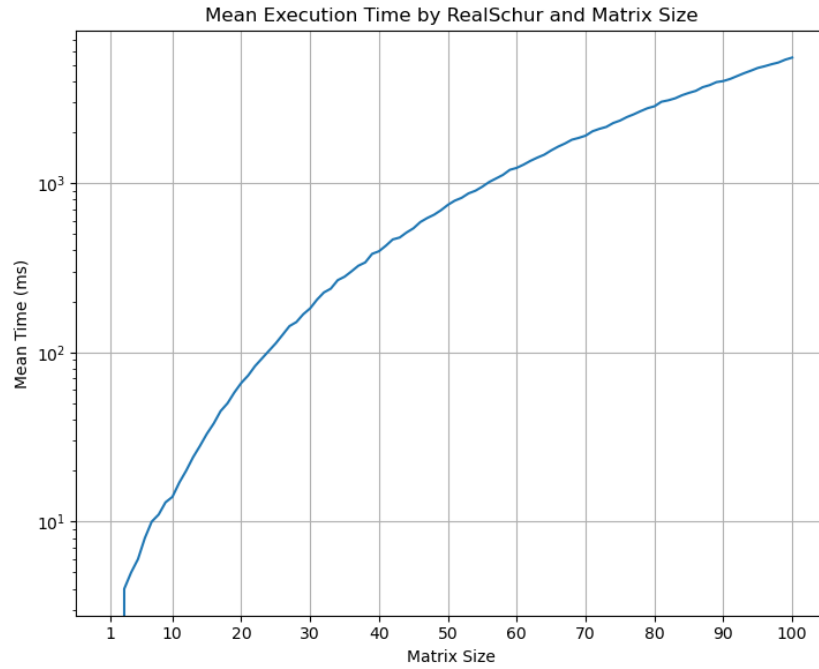


Figure 7: Mean execution time by RealSchur and matrix size

5.2.3 Naive SVD Performance

The figure below illustrates the performance of the SVD Algorithm. The execution time varies significantly due to the loop breaking condition. Hence, standard deviation increases with matrix size.

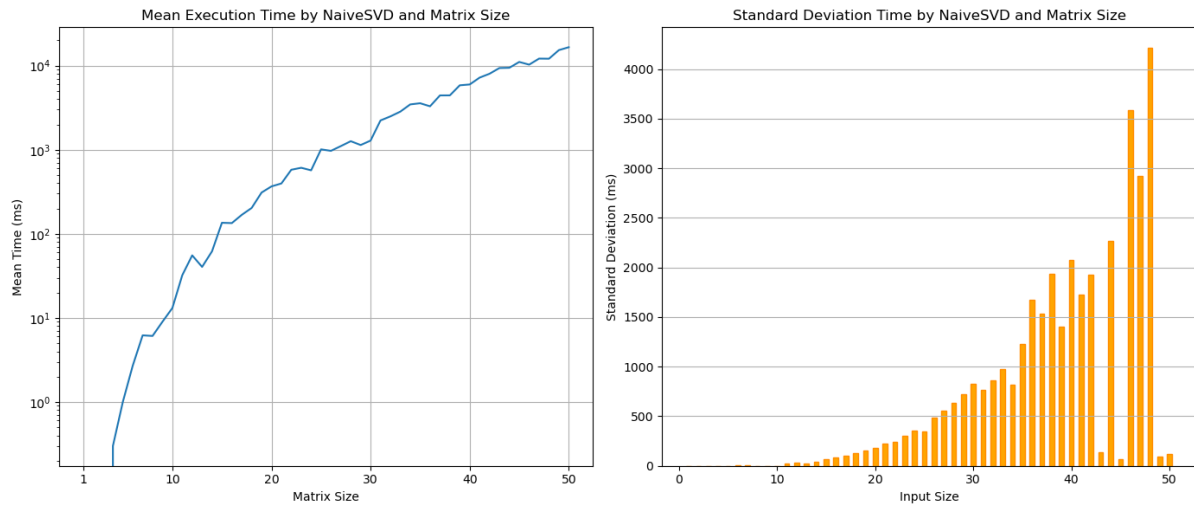


Figure 8: Mean execution time and StdDev by NaiveSVD and matrix size

6 Conclusion

In conclusion, all objectives of the programming project have been successfully achieved. These include:

- Profound theoretical understanding of mathematical ideas and technical implementation.
- Theoretical presentation of the material in a concise and accessible manner.
- Developed C++ library from scratch with all planned functionality.

The result of the accomplished work is `LinAlgTools`[\[13\]](#), a C++ library, offering matrix arithmetic and decomposition algorithms. Rigorous testing and performance benchmarking confirm its reliability, while the intentionally clean, well-structured source, code, publicly available on GitHub, makes it accessible to anyone with some knowledge of C++.

References

- [1] Joseph W. H. Liu Alan George. Householder reflections versus givens rotations in sparse orthogonal decomposition. URL: <https://www.sciencedirect.com/science/article/pii/002437958790111X>, 2002.
- [2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [3] Charles F. Van Loan Gene H. Golub. Matrix computations, 4th edition. URL: <https://doi.org/10.56021/9781421407944>, 2013.
- [4] Google. Googletest. URL: <https://github.com/google/googletest>, 1.15.2.
- [5] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. URL: <http://eigen.tuxfamily.org>, 2010.
- [6] Millman K.J. van der Walt S.J. et al Harris, C.R. Numpy. URL: <https://github.com/numpy/numpy>, 2025.
- [7] Intel Corporation. Intel math kernel library (intel mkl). URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html>, 2025.
- [8] Davis E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.
- [9] Maxim Rakhuba. Fundamentals of matrix computations. URL: <https://tinyurl.com/p5tf6ddw>, 2025.
- [10] Conrad Sanderson and Ryan Curtin. Armadillo. URL: https://arma.sourceforge.net/armadillo_iccae_2025.pdf, 2025.
- [11] Michelle Schatzman. A simple proof of convergence of the QR algorithm for normal matrices without shifts. <https://conservancy.umn.edu/server/api/core/bitstreams/ec1ba98c-42ec-4f42-8fa9-f3494af09c2d/content>, 1990.
- [12] Tai-Lin Wang and William B. Gragg. Convergence of the shifted QR algorithm for unitary hessenberg matrices. <https://www.ams.org/journals/mcom/2002-71-240/S0025-5718-01-01387-4/S0025-5718-01-01387-4.pdf>, 2001.
- [13] Zinkin Zakhar. LinAlgTools. URL: <https://github.com/Avgustineiw/LinAlgTools>, 2025.