ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Численные методы линейной алгебры

Выполнил:
Студент группы БПМИ2210 _____ Д.И.Тимижев
Подпись И.О.Фамилия

25.04.2024
Дата

Принял:
Руководитель проекта Никита Сергеевич Лукьяненко
Имя, Отчество, Фамилия
старший преподаватель
Должность, ученое звание
ФКН НИУ ВШЭ
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки _____ 2024 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2024

Содержание

Аннотация

Главной задачей проекта является изучение с теоретической стороны вычислительно устойчивых алгоритмов матричных разложений с последующей их имплементацией на языке C++. Для удобства взаимодействия код предлагается оформить в виде библиотеки, а также провести тестирование всего реализованного функционала.

1 Введение

Линейная алгебра играет важную роль в таких областях, как анализ данных, машинное обучение, квантовые вычисления, компьютерное моделирование и любая другая область с большими объёмами данных. Работа с матрицами связана с решением систем линейных уравнений и анализом информации при помощи матричных разложений, из чего вытекает потребность в устойчивых и быстрых алгоритмах. Математики и в наше время продолжают исследовать различные алгоритмы матричных разложений для снижения вычислительной сложности и количества операций. В рамках данного проекта предлагается изучить работы известных учёных, посвящённые теории вычислительно устойчивых алгоритмов, и имплементировать изученные алгоритмы на языке C++.

В рамках библиотеки требуется написать класс для представления матриц с реализацией арифметических операций с матрицами и скалярными числами, а также с полезными методами для работы с матрицами. На основе реализуемого матричного класса требуется написать матричные алгоритмы для поиска различных разложений. Для всего реализуемого функционала требуется написать тесты на корректность и устойчивость.

Теоретические аспекты работы матричных алгоритмов основаны на источниках, указанных в конце отчёта. При детальном разборе конкретных алгоритмов цитируются конкретные книги, откуда были взяты идеи.

Результатом данного проекта является готовая библиотека с названием `LinearKit`, которая включает в себя класс матриц `Matrix` с дополнительными типами для оптимизации, а также алгоритмы, такие как:

- QR разложения через отражения Хаусхолдера и вращения Гивенса.
- Приведение к форме Хессенберга.
- Приведение к бидиагональной форме.
- QR алгоритмы для симметричных и бидиагональных матриц.
- Сингулярное разложение матрицы (SVD).

Подробнее с каждым алгоритмом можно ознакомиться в разделе Теоретическая часть 3. Технические детали работы алгоритмов описаны в разделе Архитектура библиотеки 4. Для основных типов и алгоритмов написаны тесты на корректность и устойчивость. Для таких разложений, как QR и SVD, написаны тесты с замерами времени. Подробнее о тестах можно узнать в разделе Тестирование 5.

Реализованная библиотека и документация представлены в формате репозитория на `Github`, с которым можно ознакомиться по ссылке [6].

2 Описание функциональных и нефункциональных требований к программному проекту

2.1 Функциональные требования

1. Устойчивая вещественная арифметика, включающая сравнение вещественных чисел по модулю с заданными значениями малого ε для каждого типа, округления и использование общепринятых практик на основе представления типов `float`, `double` и `long double` в языке C++.

2. Класс для работы с матрицами `Matrix` над полем $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$. Класс должен предоставлять полезные методы для работы с матрицами. В его основе должна лежать матричная арифметика, транспонирование и эрмитово сопряжение, методы для конструирования матриц, рассмотрение столбцов/строк и подматриц с последующим применением арифметики к ним.

3. Классы `MatrixView` и `ConstMatrixView` для рассмотрения подматриц существующей матрицы. Используются для избежания копирования в том случае, когда стоит задача применения некоторого преобразования к конкретной части матрицы, но не ко всей.

4. Вычислительно устойчивые алгоритмы:

   (a) *QR разложение* - представление матрицы как произведение матриц Q и R. Широко используется для решения систем линейных уравнений, а также для поиска спектра матрицы. Реализация разложения на основе отражений Хаусхолдера и вращений Гивенса.

   (b) *Форма Хессенберга* - представление матрицы в форме Хессенберга, когда у матрицы ниже второй диагонали стоят нули. Алгоритм реализуется аналогично QR разложению.

   (c) *QR алгоритм* - алгоритм для поиска собственных значений матрицы, в основе которого лежит вычисление QR разложения. Использует приведение матрицы к форме Хессенберга для улучшения сходимости.

   (d) *Бидиагонализация* - представление матрицы как произведение трёх матриц U, B, V^*, причём матрица B является двудиагональной.

   (e) *SVD* - представление матрицы как произведение матриц U, Σ и V^*, причём матрица Σ является диагональной. Данное разложение является одним из самых полезных в линейной алгебре и широко используется в различных технических направлениях.

5. Тестирование библиотеки: предоставление тестов для типов и основных реализованных алгоритмов.

2.2 Нефункциональные требования

1. Реализация библиотеки на языке C++ со стандартом `C++20` без использования сторонних библиотек (за исключением библиотеки для тестирования).

2. Компилятор `clang-16` с поддержкой 20 стандарта языка C++.

3. Система сборки `CMake` версии не ниже 3.26. с поддержкой санитайзера для выявления ошибок и неопределённого поведения.

4. Форматирования кода с помощью `clang-format` на основе стиля LLVM.

5. Статического анализа кода с помощью `clang-tidy`.

6. Библиотека `GoogleTest` [3] для написания тестов.

7. Система контроля версий `Git` с удалённым репозиторием на `Github` [6].

8. Интегрированная среда разработки `CLion`.

9. Минимальные требования по памяти в компьютере: 4 ГБ для ОЗУ.

3 Теоретическая часть

3.1 Базовые определения

Определение 1. *Матрицей* размера $m \times n$ будем называть прямоугольную таблицу высоты m и n с элементами из $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad a_{ij} \in \mathbb{F}$$

Множество всех матриц размера $m \times n$ с коэффициентами из $\mathbb{F}$ будем обозначать как $\mathbb{F}^{m \times n}$.

Определение 2. *Вектором* будем называть матрицу, состоящую из одного столбца или одной строчки:

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix}$$

Под словом *вектор* будем подразумевать вектор-столбец. Множество всех векторов размера n с коэффициентами из $\mathbb{F}$ будем обозначать как $\mathbb{F}^n = \mathbb{F}^{n \times 1}$.

Определение 3. *Эрмитово сопряженная матрица* - матрица A^* для $A \in \mathbb{C}^{m \times n}$, полученная путем транспонирования и комплексного сопряжения каждого элемента, т.е. $A^* = \overline{A}^T$

Определение 4. Матрица A является *невырожденной*, если её определитель не равен нулю.

Определение 5. Матрицы $A, B \in \mathbb{F}^{n \times n}$ являются *подобными*, если существует невырожденная $C \in \mathbb{F}^{n \times n}$, такая что $B = C^{-1}AC$.

Определение 6. Матрица $A \in \mathbb{C}^{n \times n}$ является *унитарной*, если $A^* = A^{-1}$, то есть $A^*A = AA^* = I$, где I - единичная матрица. Столбцы (строки) унитарной матрицы образуют ортонормированный базис в унитарном пространстве.

Определение 7. Матрица $A \in \mathbb{R}^{n \times n}$ является *ортогональной*, если $A^T = A^{-1}$, где A^T - транспонированная матрица A, то есть $A^TA = AA^T = I$, где I - единичная матрица. Столбцы (строки) ортогональной матрицы образуют ортонормированный базис в вещественном пространстве. Ортогональная матрица - аналог унитарной матрицы, но над полем $\mathbb{R}$.

Определение 8. *Скалярным произведением* векторов $a, b \in \mathbb{R}^n$, где $\mathbb{R}^n$ - вещественное евклидовое пространство, называется величина $\langle a, b \rangle = b^T a = \sum_{i=1}^{n} a_i b_i$. В случае комплескного евклидового пространства $\mathbb{C}^n$ скалярное произведение векторов $u, v \in \mathbb{C}^n$ задаем как: $\langle u, v \rangle = v^* u = \sum_{i=1}^{n} u_i \overline{v_i}$.

Определение 9. *Евклидовой нормой* для вектора $u = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \in \mathbb{F}^n$ называется число $\|u\|_2 := \sqrt{\sum_{i=1}^{n} |u_i|^2}$.

Нормой $\|u\|$ стандартно будем называть евклидову норму для вектора u, если не обговорена иная норма.

Определение 10. Собственным значением матрицы $A \in \mathbb{F}^{n \times n}$ будем называть такое $\lambda \in \mathbb{F}$, что для него существует некоторый вектор $v \in \mathbb{F}^n$, такой что $Av = \lambda v$. Множество собственных значений матрицы A будем обозначать как $\lambda(A)$.

Определение 11. Матрица $A \in \mathbb{F}^{m \times n}$ называется *верхнетреугольной*, если её элементы, расположенные ниже диагонали, равны нулю:

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

Определение 12. Матрица $A \in \mathbb{F}^{m\times n}$ называется *диагональной*, если её элементы, расположенные вне диагонали, равны нулю:

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} & \cdots & 0 \end{bmatrix}$$

Определение 13. Матрица $A \in \mathbb{F}^{m\times n}$ называется *двудиагональной (бидиагональной)*, если её элементы, расположенные ниже диагонали и выше поддиагонали, равны нулю:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & a_{22} & a_{23} & \cdots & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{mn} & \cdots & 0 \end{bmatrix}$$

Определение 14. Матрица $A \in \mathbb{F}^{n\times n}$ называется *тридиагональной*, если её элементы, расположенные ниже поддиагонали и выше поддиагонали, равны нулю:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ 0 & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Замечание 15. Обозначим A^i как i-ый столбец матрицы $A \in \mathbb{F}^{m\times n}$, а также $A^{i:j}$ - подматрица A с i-го по j-ый столбец. Аналогично A_i для i-ой строки и $A_{i:j}$ - подматрица с i-ой по j-ую строку.

Замечание 16. Договоримся в общем случае использовать в формулах эрмитово сопряжение для матриц над всяким полем $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$. Если конкретное поле не задано, то эрмитово сопряжение в случае $\mathbb{F} = \mathbb{R}$ понимаем как транспонирование.

Замечание 17. Договоримся в общем случае называть матрицу $A \in \mathbb{F}^{n\times n}$ унитарной, если она соответсвует определению унитарности. В случае $\mathbb{F} = \mathbb{R}$ полагаем, что матрица является ортогональной.

3.2 Вычислительные алгоритмы

3.2.1 Отражение Хаусхолдера

Определение 18. *Отражение Хаусхолдера* представляет собой линейное преобразование векторного пространства, отражающее его относительно гиперплоскости, проходящей через начало координат. Это ортогональное преобразование находит широкое применение в различных численных методах, например для QR разложения и бидиагонализация матрицы. Идея алгоритма взята из этой книги [4].

Это отражение достигается путем умножения исходного вектора на унитарную матрицу. Уникальное свойство данного произведения заключается в том, что можно повернуть вектор так, чтобы он лежал на оси координат, то есть все его координаты ниже первой будут равны нулю. Это позволяет применять линейные отражения для приведения матрицы к верхнетреугольному виду, последовательно обнуляя все элементы ниже главной диагонали. Матрицу Хаусхолдера можно вычислить по такой формуле:

$$H = I - \frac{2}{\|v\|} vv^*, \text{ где } v = \operatorname{sign}(x_1)\|x\|e_1 + x \text{ для обнуления координат ниже первой}$$

Причём данная матрица является унитарной. Если $x_1 \in \mathbb{R}$, в таком случае $\operatorname{sign}(x_1)$ возвращает знак числа, при этом $\operatorname{sign}(0) = 1$. Если же $x_1 \in \mathbb{C}$, тогда $\operatorname{sign}(x_1)$ нормирует комплексное число, т.е. $\operatorname{sign}(x_1) = \frac{x}{|x|}$.

Пусть $A \in \mathbb{F}^{m\times n}$, вычислим вектор v на основе A^1, тогда при помощи отражения Хаусхолдера мы можем обнулить все элементы ниже первого в столбце путём произведения слева:

$$HA = A - \frac{2}{\|v\|} v(v^*A)$$

Пример отражения Хаусхолдера для вектора $x \in \mathbb{F}^n$:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \xrightarrow{H} \quad Hx = \begin{bmatrix} \operatorname{sign}(x_1)\|x\| \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

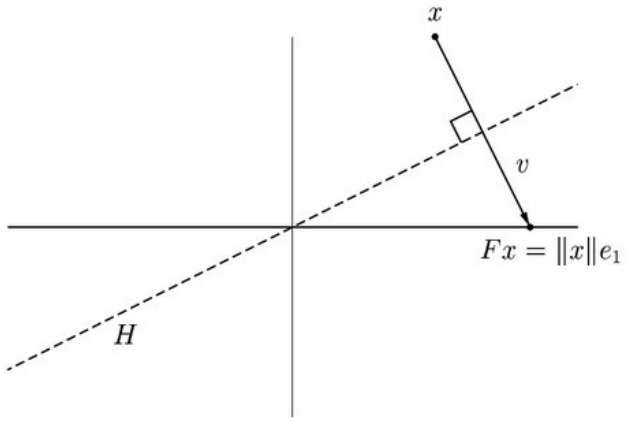


Рис. 1: Геометрическое представление отражения Хаусхолдера для вектора x

Можно выделить левое и правое отражение Хаусхолдера. Левое отражение работает со столбцом, а правое со строкой. Подробнее коснемся этой темы при обсуждении бидиагонализации матрицы.

Algorithm 1 Левое отражение Хаусхолдера

```
function HouseholderLeftReflection(A ∈ F^{m×n}, x ∈ F^m)
    v = x + sign(x_1)‖x‖e_1
    v = v · 1/‖v‖
    HA = A − 2v(v*A)
    return HA
```

Algorithm 2 Правое отражение Хаусхолдера

function HOUSEHOLDERRIGHTREFLECTION($A \in \mathbb{F}^{m \times n}, x^* \in \mathbb{F}^n$)
    $v = x + \text{sign}(x_1)\|x\|e_1$
    $v = v \cdot \frac{1}{\|v\|}$
    $AH = A - 2(Av^*)v$
    return AH

3.2.2 Вращение Гивенса

Определение 19. *Вращение Гивенса* - это ортогональный оператор, который поворачивает вектор на некоторый угол ϕ. Более подробная информация находится в этой книге [2].

Для выполнения поворота на угол конструируется матрица поворота G_{ij}, которую называют матрицей Гивенса, поворот которой осуществляется относительно i-го и j-го элемента вектора. В общем виде ортогональный оператор имеет вид:

$$G_{ij} = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\phi & \cdots & -\sin\phi & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin\phi & \cdots & \cos\phi & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

Данная матрица отличается от единичной лишь подматрицей, которая является матрицей поворота в $\mathbb{F}^2$:

$$G_{ij}[[i:j],[i:j]] = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

Вращение Гивенса обладает таким же свойством, как и отражение Хаусхолдера, а именно - при помощи вращений можно обнулить все координаты вектора ниже первой. Особенность лишь в том, что данное вращение обнуляет лишь 1 координату за одно применение оператора:

$$a = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix},\ \cos\phi = \frac{a_1}{\|a\|}, \sin\phi = \frac{a_2}{\|a\|} \implies G_{12}a = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \times \\ 0 \end{bmatrix}$$

Таким образом, можно выполнить последовательность вращений Гивенса для вектора и обнулить его все координаты ниже первой:

$$\begin{bmatrix} \times \\ \times \\ \times \\ \times \end{bmatrix} \xrightarrow{G_{34}} \begin{bmatrix} \times \\ \times \\ \times \\ 0 \end{bmatrix} \xrightarrow{G_{23}} \begin{bmatrix} \times \\ \times \\ 0 \\ 0 \end{bmatrix} \xrightarrow{G_{12}} \begin{bmatrix} \times \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

По аналогии с отражениями Хаусхолдера, можно выделить левый и правый повороты Гивенса. В таком случае правый поворот работает с вектором-строкой. Причём матрица Гивенса для правого поворота имеет такой же вид, как для левого, но транспонированная:

$$a^T G_{12} = \begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} \times & 0 \end{bmatrix}$$

$$\begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \xrightarrow{G_{34}} \begin{bmatrix} \times & \times & \times & 0 \end{bmatrix} \xrightarrow{G_{23}} \begin{bmatrix} \times & \times & 0 & 0 \end{bmatrix} \xrightarrow{G_{12}} \begin{bmatrix} \times & 0 & 0 & 0 \end{bmatrix}$$

Algorithm 3 Коэффициенты Гивенса

function GIVENSCOEFFICIENTS($a, b \in \mathbb{F}$)
    $cos = a\ /\sqrt{|a|^2 + |b|^2}$
    $sin = -b\ /\sqrt{|a|^2 + |b|^2}$
    return cos, sin

Algorithm 4 Левый поворот Гивенса

function LeftGivensRotation($a \in \mathbb{F}^2$)
  $cos, sin = \text{GivensCoefficients}(a_1, a_2)$
  $a'_1 = \overline{cos} \cdot a_1 - \overline{sin} \cdot a_2$
  $a'_2 = sin \cdot a_1 + cos \cdot a_2$
  $a_1 = a'_1$
  $a_2 = a'_2$

Algorithm 5 Правый поворот Гивенса

function RightGivensRotation($a^T \in \mathbb{F}^2$)
  $cos, sin = \text{GivensCoefficients}(a_1, a_2)$
  $a'_1 = \overline{cos} \cdot a_1 - \overline{sin} \cdot a_2$
  $a'_2 = sin \cdot a_1 + cos \cdot a_2$
  $a_1 = a'_1$
  $a_2 = a'_2$

3.2.3 QR разложение

Определение 20. *QR разложение* матрицы - представление матрицы $A \in \mathbb{F}^{m \times n}$ в виде произведения матриц Q и R, где $Q \in \mathbb{F}^{m \times m}$ - унитарная, а $R \in \mathbb{F}^{mtimesn}$ - верхнетреугольная.

Получение желаемого разложения основано на использовании преобразований Хаусхолдера для каждого столбца матрицы A, причём преобразование применяем ко всей матрице. Аналогичным способом можно получить QR разложение через вращения Гивенса, поэтому рассмотрим только случай Хаусхолдера.

Как было описано выше, вся суть алгоритма в последовательном применении преобразований Хаусхолдера к столбцам матрицы A поочередно, при этом на каждой итерацией берём столбец меньшего размера, чтобы привести матрицу к верхнетреугольной R. Заметим, что формула для R имеет такой вид:

$$R = H_n H_{n-1} \dots H_1 A, \text{ где } H_i \text{ - матрица Хаусхолдера}$$

Так как матрица Хаусхолдера является унитарной и эрмитовой, в таком случае $H_i^{-1} = H_i^* = H_i$. Поэтому матрицу Q соберём так, чтобы при произведении обнулялись все преобразования, применённые к матрице A:

$$Q = H_1 H_2 \dots H_n \implies QR = H_1 H_2 \dots H_n H_n \dots H_2 H_1 A = A$$

Рассмотрим, как будет меняться матрица A в процессе вычисления QR разложения:

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{H_2} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{H_3} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix} = R, \quad Q = H_1 H_2 H_3$$

Algorithm 6 QR разложение через отражения Хаусхолдера

function HouseholderQR(Matrix $A \in \mathbb{F}^{m \times n}$)
  $Q = I_m, \ R = A$

  for $k = 1 \dots \min(m, n)$ **do**
    $A_{sub} = A_{k:n}^{k:m}$
    $a = A_{sub}^1$ - первый столбец
    $H_k A_{sub} = \text{HouseholderLeftReflection}(A_{sub}, a)$
    $H_k Q = \text{HouseholderLeftReflection}(Q^{k:m}, a)$

    $R_{k:n}^{k:m} = H_k A_{sub}$
    $Q^{k:m} = Q H_k$
  $Q = Q^*$
  return Q, R

3.2.4 Форма Хессенберга

Определение 21. Говорят, что матрица H имеет *форму Хессенберга*, если она удовлетворяет такому виду:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2n} \\ 0 & h_{32} & h_{33} & \cdots & h_{3n} \\ 0 & 0 & h_{43} & \cdots & h_{4n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & h_{mn} \end{bmatrix}$$

То есть она напоминает верхнетреугольный вид за исключением того факта, что на нижней поддиагонали стоят не нули. Такая матрица хорошо сходится к своему разложению Шура в QR алгоритме, который будет описан далее. Также стоит заметить, что достаточно легко вычислить QR разложение для такой матрицы - требуется лишь пройти всю диагональ с вращениями Гивенса, чтобы занулить поддиагональ, из-за чего в некоторых источниках для вычисления QR разложения перед началом проверяют, не имеет ли матрица форму Хессенберга.

Вычисление формы Хессенберга напоминает вычисление QR разложения через отражения Хаусхолдера, лишь требуется занулять элементы ниже поддиагонали. При этом, далее выполняется правое отражение Хаусхолдера уже для строки, в связи с чем данный алгоритм в случае симметричной матрицы приводит её к тридиагональной форме, которая сходится к диагональной в QR алгоритме. Подробнее об этом рассказано в этой книге [2]. По итогу из матрицы A получаем форму Хессенберга H и унитарную матрицу Q, причём $A = QHQ^*$, из чего следует, что A и H являются подобными.

Псевдокод для алгоритма приведения к форме Хессенберга отличается от псевдокода QR разложения взятием подматрицы на размер меньше и применением правого отражения после левого к первой строке, потому псевдокод не будет приведён.

3.2.5 QR алгоритм

Определение 22. *QR Алгоритм* представляет собой численный метод для нахождения собственных значений матрицы. Суть метода заключается в итерационном приведении матрицы A к некоторой унитарно подобной ей матрице A_k при помощи QR разложения, причем матрица A_k сходится к форме Шура (или же к диагональной в симметричном случае). В силу подобия матриц A и A_k их наборы собственных значений совпадают. Таким образом задача поиска собственных значений матрицы A сводится к задаче выведения матрицы A_k и поиска собственных значений для нее, что является более тривиальной задачей.

Для улучшения сходимости алгоритма используют такие методы, как приведение к форме Хессенберга перед итерациями, а также использование сдвигов. Опишем как наивную версию алгоритма, так и с использованием сдвигов, так как они позволяют существенно сократить количество итераций для сходимости. Наивный алгоритм имеет такой вид:

Algorithm 7 Наивный QR алгоритм

```
function GetEigenvaluesNative(A ∈ F^{n×n}, ItCount)
    A_1 = A
    for k = 1 ... ItCount do
        Q_k R_k = A_k
        A_{k+1} = R_k Q_k
    return A_ItCount
```

Далее опишем алгоритм с формой Хессенберга и на основе сдвигов. В его работе для каждой итерации используется некоторый параметр λ_k, который выбирается из собственных значений правого нижнего угла размера 2×2 матрицы A_k. Стандартно за λ_k берут сдвиг Уилкинсона, об этом подробнее можно узнать в этой книге [2]:

Algorithm 8 QR алгоритм со сдвигами

function GetEigenvalues($A \in \mathbb{F}^{n\times n}$, ItCount)
    $H_1 = \text{ToHessenbergForm(A)}$
    for $k = 1 \dots$ ItCount **do**
        $\lambda_k = \text{GetWilkinsonShift}(H_k)$
        $Q_k R_k = H_k - \lambda_k I$
        $H_{k+1} = R_k Q_k + \lambda_k I$
    return $H_{ItCount}$

3.2.6 Бидиагонализация

Определение 23. *Бидиагонализация* матрицы - представление матрицы $A \in \mathbb{F}^{m\times n}$ в виде произведения матриц U, B, V^*, где $U \in \mathbb{F}^{m\times m}$ и $V \in \mathbb{F}^{n\times n}$ - унитарные, а $B \in \mathbb{F}^{m\times n}$ - бидиагональная.

Алгоритм бидиагонализации основан на поочередном применении отражений Хаусхолдера, сначала для столбца левое отражение, а потом правое для строки. Все левые отражения заносим в матрицу U, а правые - в матрицу V, аналогично QR разложению. Процесс бидиагонализации выглядит так:

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{V_1} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{U_2}$$

$$\xrightarrow{U_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{V_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = B$$

Алгоритм бидиагонализации матрицы широко используется при вычислении сингулярного разложения, так как бидиагональная матрица сходится к диагональной матрице с сингулярными числами. Сингулярное разложение подробнее разберём далее.

Algorithm 9 Бидиагонализация матрицы

function Bidiagonalization($A \in \mathbb{F}^{m\times n}$)
    $B = A,\ U = I_m,\ V = I_n$

    for $k = 1 \dots \min(m, n) - 1$ **do**
        $A_{sub} = A^{k:m}_{k:n}$
        $a = A^1_{sub}$ - первый столбец
        $H_k A_{sub} = \text{HouseholderLeftReflection}(A_{sub}, a)$
        $H_k U = \text{HouseholderLeftReflection}(U^{k:m}, a)$

        $B^{k:m}_{k:n} = H_k A_{sub}$
        $U^{k:m} = H_k U$

        if $k + 1 < n$ **then**
            $B_{sub} = B^{k:m}_{k+1:n}$
            $b = (B_{sub})_1$ - первая строка
            $BH_k = \text{HouseholderRightReflection}(B_{sub}, b)$
            $VH_k = \text{HouseholderRightReflection}(V, b)$

            $B^{k:m}_{k+1:n} = BH_k$
            $V_{k+1:n} = VH_k$

    $U = U^*$
    $V = V^*$
    return U, B, V

3.2.7 QR алгоритм для бидиагональной матрицы

Определение 24. *QR алгоритм для бидиагональной матрицы* отличается от стандартного QR алгоритма как минимум тем фактом, что стандартный принимает квадратную матрицу. Ключевая суть алгоритма является аналогичной - итеративное сведение бидиагональной матрицы B к диагональной матрице Σ, что пригодится для вычисления SVD далее.

Определим матрицу $T = B^T B$ (матрица B является вещественной, об этом в следующем пункте). Тогда эта матрица является симметричной тридиагональной и сходится к диагональной матрице в QR алгоритме.

Можно вычислить собственное значение нижнего правого 2×2 блока матрицы T - сдвиг Уилкинсона, который обозначим за λ. Тогда если применить левый поворот Гивенса к матрице B относительно вектора $\begin{bmatrix} T_{0,0} - \lambda & T_{0,1} \end{bmatrix}^T = \begin{bmatrix} B_{0,0}^2 - \lambda & B_{0,1} \cdot B_{0,0} \end{bmatrix}^T$, на позиции $(1, 0)$ матрицы B появится некоторое значение, которое можно итеративно вывести за матрицу. Данный процесс является итерацией QR алгоритма для бидиагональной матрицы. На примере:

G' - такая, что $G'[[1:2],[1:2]] \begin{bmatrix} T_{0,0} - \lambda \\ T_{0,1} \end{bmatrix} = \begin{bmatrix} \cos & -\sin \\ \sin & \cos \end{bmatrix} \begin{bmatrix} T_{0,0} - \lambda \\ T_{0,1} \end{bmatrix} = \begin{bmatrix} \times \\ 0 \end{bmatrix}$, тогда итерация QR алгоритма:

$$G'B = G' \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = \begin{bmatrix} \times & \times & 0 & 0 \\ + & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{12}} \begin{bmatrix} \times & \times & + & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{23}^T}$$

$$\xrightarrow{G_{23}^T} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & + & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{23}} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & + \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{34}^T} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & + & \times \end{bmatrix} \xrightarrow{G_{34}} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

Итеративно повторяя данные шаги QR алгоритма бидиагональная матрица B сойдётся к диагональной матрице Σ.

Отметим тот факт, что если какое-то число на верхней поддиагонали матрицы B близко к нулю, то можно разбить матрицу на 2 бидиагональные матрицы и выполнять QR алгоритма для по отдельности:

$$B = \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & 0 & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}$$

Если какое-то число на диагонали близко к нулю, тогда при помощи правых поворотов Гивенса можем избавиться от числа на поддиагонали рядом с нулевым, по итогу сможем разбить матрицу B на две бидиагональные:

$$B = \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & 0 & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{23}} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{G_{24}} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}$$

Данный процесс называется *Cancellation*. Подробнее ознакомиться с процессом разбиения и Cancellation можно в данной презентации [1], а с алгоритмом можно в книге [2] от Gene H. Golub, который и описал данную реализацию. В случае разбиения матрицы и Cancellation остаётся непонятным, с какого момента число считать близким к нулю. Автор рекомендует сравнивать относительно значения $\max_i \varepsilon_m (B_{i,i} + B_{i,i+1})$. Детальные объяснения работы итераций алгоритма изложены в разделе 11.2.4 данного источника [5].

3.2.8 SVD

Определение 25. *Singular Value Decomposition (SVD)* - разложение матрицы A в произведение трёх матриц U, Σ и V^*, причём $U \in \mathbb{F}^{m \times m}$ и $V \in \mathbb{F}^{n \times n}$ - унитарные, а Σ - диагональная с положительными вещественными элементами, которые называют сингулярными числами. Причём, сингулярные числа в Σ отсортированы, поэтому за $\sigma_1(A)$ можно обозначить старшее сингулярное число матрицы A и $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_r > 0$.

Так как любую матрицу можно разложить в сумму матриц ранга 1, то пусть $r = \mathrm{rk}(A)$, тогда матрицу A можно представить в виде суммы r матриц:

$$A = U\Sigma V^* = U^1 \sigma_1 V_1^* + \ldots + U^r \sigma_r V_r^*$$

Левыми сингулярными векторами будем называть столбцы матрицы U, соответсвенно правые сингулярные векторы - столбцы матрицы V.

Существуют несколько видов сингулярных разложений, к примеру: полное, компактное, тонкое и усечённое. В данной работе рассматривается только полное сингулярное разложение, которое задано матрицами $U \in \mathbb{F}^{m\times m}$, $V \in \mathbb{F}^{n\times n}$ и $\Sigma^{m\times n}$, поэтому остальные виды оставим без комментариев. Полное сингулярное разложение можно представить в блочном виде:

$$A = U\Sigma V^* = \begin{bmatrix} U_r & U_r^\perp \end{bmatrix} \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_r & V_r^\perp \end{bmatrix}^*$$

где $U_r \in \mathbb{F}^{m\times r}$ - r левых сингулярных векторов, $V_r \in \mathbb{F}^{n\times r}$ - r правых сингулярных векторов, $\Sigma_r \in \mathbb{F}^{r\times r}$ - невырожденная диагональная матрица с сингулярными значениями, а $U_r^\perp$ и $V_r^\perp$ - ортогональные дополнения матриц U_r и V_r, которые добавлены для сохранения размеров матриц U, V и их унитарности. Причём векторы из ортогональных дополнений обнуляются об матрицу Σ, поэтому они не так важны.

Вычисление SVD - достаточно трудоемкая задача. Наивный алгоритм через нахождение A^*A и её диагонализацией является неустойчивым, что доставляет много трудностей, потому Gene H. Golub в своей работе [2] предложил алгоритм для нахождения сингулярного разложения путём бидиагонализации матрицы $A = U_1 B V_1^*$ и использования QR алгоритма для матрицы B, из чего получим $B = U_2\Sigma V_2^*$, тогда $U = U_1U_2$ и $V^* = V_2^*V_1^*$, и искомое разложение $A = U\Sigma V^*$.

Algorithm 10 Golub-Kahan SVD Algorithm

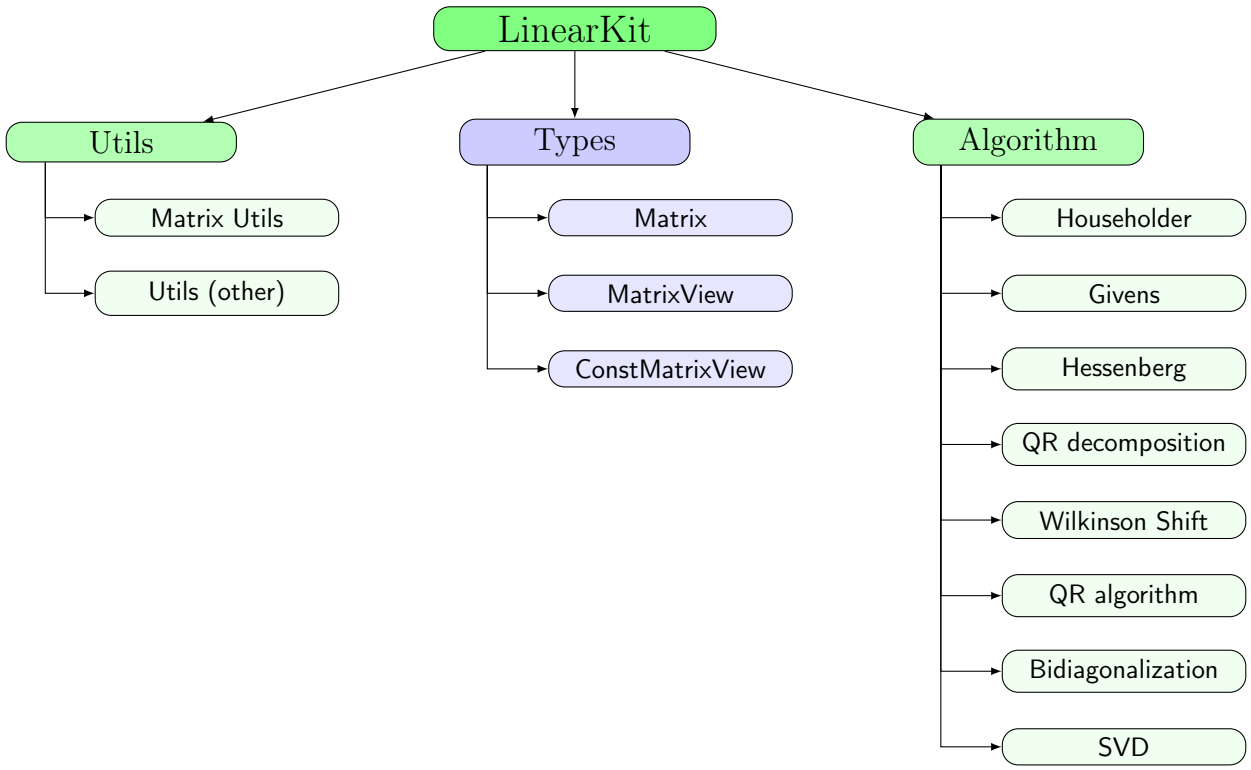
```
1:  function SVD(A ∈ F^{n×n}, ItCount)
2:      U_1, B, V_1^* = Bidiagonalization(A)
3:      U_2, Σ, V_2^* = BidiagAlgorithmQR(B, ItCount)
4:
5:      ToPositiveSingularValues(Σ, V_2^*)
6:      SortSingularValues(U_2, Σ, V_2^*)
7:
8:      U = U_1 U_2
9:      V^* = V_2^* V_1^*
10:     return U, Σ, V^*
```

В результате работы бидиагонального QR алгоритма могут получиться отрицательные сингулярные числа, поэтому требуется умножить их на -1, а этот коэффициент занести в матрицу V^*. Также они могут быть не отсортированы по неубыванию, в связи с чем требуется переставить местами векторы в матрицах U, V и сингулярные значения в Σ.

Стоит отметить, что в случае $A \in \mathbb{C}^{m\times n}$ бидиагональная матрица B, полученная в результате функции `Bidiagonalization`, тоже может иметь комплексные коэффициенты, из-за чего может выйти некорректное SVD с комплексными сингулярными числами. В таком случае требуется на этапе бидиагонализации после каждого отражения Хаусхолдера домножать элементы B на сопряженные, чтобы по итогу получить вещественную бидиагональную матрицу. Соответственно, для корректного разложения матрицы A потребуется на коэффициенты также домножать определённые строки/столбцы матриц U и V.

4 Архитектура библиотеки

Библиотека разбита на несколько пространств и типов для соблюдения семантики и легкой ориентации внутри проекта. Структура библиотеки в форме диаграммы:



Рассмотрим подробнее каждое пространство.

4.1 Utils

- `Utils` - в данном пространстве собраны вспомогательные функции, структуры и концепты:

  1. Концепт `Utils::FloatOrComplex` для создания структур исключительно на основе вещественных и комплексных типов на основе вещественных (например, `std::complex<float>`).
  2. Значение ε для каждого типа с функциями для сравнения вещественных (комплексных) чисел. На основе функций сравнения реализовано сравнение матриц.
  3. Функция `Sign` для вещественных и комплексных чисел. Для вещественных чисел возвращает 1 или -1 в зависимости от знака, а для комплексных - нормированное комплексное число. Используется для отражений Хаусхолдера и сдвигов Уилкинсона.

- `MatrixUtils` - в данном пространстве собраны вспомогательные функции, структуры и концепты для матриц:

  1. Концепты `MatrixUtils::MatrixType` и `MatrixUtils::MutableMatrixType`, которые реализованы в целях удобства и мобильности. Благодаря ним можно вызывать алгоритмы как для `Matrix`, так и для следящих `MatrixView` и `ConstMatrixView`.
  2. Множество булевых функций для матриц. Можно сравнивать матрицы относительно некоторого ε, а также определять вид матрицы при помощи функций `IsSquare`, `IsUnitary`, `IsDiagonal` и так далее. Широко используется в алгоритмах и тестах.
  3. Функция для смены типа матрицы `CastMatrix`, которая используется в SVD, чтобы, к примеру, перейти от комплексных матриц к вещественным во время вычисления.
  4. Функции `Split` и `Join`, которые могут разбить матрицу на 2 или же соединить 2 матрицы в одну. Используются в QR алгоритме для бидиагональных матриц.

4.2 Types

- `Matrix` - базовый класс для представления матрицы. Данные хранятся в буфере на основе `STL` контейнера `std::vector<T>` с некоторым типом `T`, который может быть вещественным или комплексным `std::complex<T>` на основе вещественного за счёт концепта `Utils::FloatOrComplex`. Предоставляет методы для работы с элементами матрицы, со строками, столбцами и диагональю. Матрицу можно транспонировать, эрмитово сопрячь или нормировать (в случае вектора). При этом, матрицу можно создать на основе данных со стандартного входного потока и в красивом формате вывести в консоль.

- `MatrixView` - класс, который имитирует поведение матрицы, но следит за какой-то существующей матрицей (или её подматрицей). За счёт этого можно изменять подматрицу (или столбец/строку) существующей матрицы без лишних копирований. К примеру, взять подматрицу и умножить на другую квадратную матрицу.

- `ConstMatrixView` - класс, который работает аналогично `MatrixView` за тем исключением, что подматрицу, за которой следит объект, нельзя изменять. На основе данного класса реализована арифметика, которая влечёт за собой создание новой матрицы, так как исходные не меняются.

- За счёт концептов `MatrixUtils::MatrixType` и `MatrixUtils::MutableMatrixType` реализованы шаблонные арифметические и условные операторы, поэтому в рамках библиотеки доступна любая арифметика между различными матрицами. Можно спокойно складывать, к примеру, объекты классов `Matrix` и `MatrixView`.

Пример работы с матрицами:

```
1     LinearKit::Matrix<> m = {{1, 2, 3}, {4, 5, 6}};
2     LinearKit::MatrixView<> v = m.GetSubmatrix({0, 2}, {0, 2}); // [[1, 2], [4, 5]]
3
4     LinearKit::ConstMatrixView<> cv = v.ConstView();
5     v += cv; // [[2, 4], [8, 10]]
6
7     std::cout << m << std::endl; // [[2, 4, 3], [8, 10, 6]]
```

4.3 Algorithm

- `Householder` - функции для применения отражений Хаусхолдера для матриц. В эту тему входят функции `HouseholderReduction` для подготовки вектора к отражению на ось, `HouseholderLeftReflection` и `HouseholderRightReflection` для отражения матрицы. Внутри себя рассматривает желаемую подматрицу и для неё выполняет произведение на матрицу Хаусхолдера без лишних копирований.

- `Givens` - функции для применения вращений Гивенса для матриц. В эту тему входят такие функции, как `GetGivensCoefficient` для вычисления $\cos\phi$ и $\sin\phi$ для матрицы Гивенса, а также `GivensLeftRotation` и `GivensRightRotation`, которые относительно двух номеров строк (столбцов) и переданных элементов применяют поворот ко всей матрице.

- `Hessenberg` - функция `GetHessenbergForm`, которая возвращает для матрицы пару, состоящую из матриц H - форма Хессенберга исходной матрицы и Q - унитарная, такая что $A = QHQ^*$ для некоторой матрицы A. Сам алгоритм реализован на основе левых и правых отражений хаусхолдера из темы `Householder`.

- `QR decomposition` - алгоритм для получения QR разложения реализован в виде нескольких функций, которые в возвращают пару из матриц Q и R, удовлетворяющих определению:

  1. `HouseholderQR` - реализация алгоритма на основе отражений Хаусхолдера. Проходим каждый столбец, выбираем его и применяем отражение как к исходной матрице, так и к матрице Q.
  2. `GivensQR` - реализация алгоритма на основе вращений Гивенса. Принцип тот же - для каждого столбца обнуляем его до нужной координаты, применяем к матрицам R и Q, которые перед началом итераций равны A и I соответственно для некоторой A.
  3. `HessenbergQR` - особый случай, который вызывается в предыдущих двух алгоритмах, если матрица имеет форму Хессенберга. Простой проход по всей диагонали с обнулением элемента под диагональю поворотами Гивенса.

- `Wilkinson shift` - реализована функция для вычисления сдвига `GetWilkinsonShift`. Относительно переданного индекса и матрицы вычисляет сдвиг устойчивой формулой. Также существует функция `GetBidiagWilkinsonShift`, которая используется в рамках QR алгоритма бидиагональной матрицы, так как формирует подматрицу матрицы $T = B^T B$ из описания алгоритма и для неё возвращает сдвиг Уилкинсона.

- `QR algorithm` - в отдельных файлах представлены как реализация QR алгоритма для симметричных матриц `GetSpecDecomposition`, так и QR алгоритм для бидиагональных матриц `BidiagAlgorithmQR`. Обе функции возвращают диагональную матрицу вместе с унитарной матрицей перехода (бидиагональный алгоритм возвращает 2 унитарные матрицы)

- `Bidiagonalization` - функция, которая возвращает бидиагональную матрицу с двумя унитарными матрицами перехода. Сама форма вычисляется при помощи левых и правых отражений Хаусхолдера. В комплексном случае домножает элементы на коэффициенты так, чтобы бидиагональная форма матрицы стала вещественной.

- `SVD` - функция, которая возвращает сингулярное разложение для матрицы. Если $m < n$ для $A \in \mathbb{F}^{m \times n}$, алгоритм эрмитово сопрягает матрицу и вычисляет сингулярное разложение уже для A^*, а позднее сопрягает обратно. Внутри себя функция приводит матрицу к бидиагональному виду, после чего переводит её в тип `long double` и запускает `BidiagAlgorithmQR`, после которого возвращает тип обратно, сортирует сингулярные значения и возвращает вектор-строку с сингулярными числами вместе с матрицами левых и правых сингулярных векторов.

Пример работы с алгоритмами:

```
using namespace std;

LinearKit::Matrix<double> m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
auto [U, S, VT] = LinearKit::Algorithm::SVD(m);

auto S_diag = LinearKit::Matrix<double>::Diagonal(S, U.Columns(), VT.Rows());
auto [Q, R] = LinearKit::Algorithm::HouseholderQR(S_diag);

assert(LinearKit::MatrixUtils::IsUnitary(Q) && "QR is not QR.");

auto new_U = U * Q;
auto new_VT = R * VT;

cout << "New decomposition:" << endl;
cout << new_U << endl;
cout << new_VT << endl;
cout << LinearKit::MatrixUtils::AreEqualMatrices(m, new_U * new_VT) << endl;
```

5 Тестирование

В целях организации юнит-тестов в сборку подключена сторонняя библиотека `GoogleTest` [3]. Для удобного тестирования был написан генератор случайных матриц `RandomMatrixGenerator<T>`, который создаёт случайные прямоугольные или квадратные матрицы желаемых размеров со случайными коэффициентами от -100 до 100. Также генератор может сгенерировать случайную матрицу случайных размеров от 1 до 100, что используется в стресс тестах.

Запустить все тесты можно с помощью таргета `test_all`. Помимо этого существуют отдельные таргеты тестирования, которые будут описаны далее.

5.1 Тестирование Matrix

Для тестирования типов, предоставляемых библиотекой, реализованы 2 тестовые сборки: `test_matrix` и `test_matrix_view`. Тесты сборок реализованы в виде юнит-тестов, проверяющих корректность работы методов, реализованных для каждого типа. Для отдельного типа `ConstMatrixView` не предусмотрены тесты, так как большинство методов первых двух типов внутри себя используют `ConstMatrixView`.

Тесты для типов написаны со следующей семантикой:

- *Корректность* - для каждого функционала предусмотрен тест, который проверяет корректное поведение для заданных явно матриц. Каждый тест включает в себя как матрицу на вход, так и желаемую матрицу, которые и сравниваются после исполнения некоторого функционала.
- *Устойчивость* - для каждого типа отдельно написан тестовый блок со стресс тестом, проверяющим устойчивость арифметики. На каждой итерации генерируются 2 случайные матрицы случайных размеров, для которых применяются операторы. Для данного теста корректность не важна, оценивается лишь живучесть реализованного функционала.

Все тесты для данных типов проходят успешно. Ознакомиться с ними можно в папке `tests` репозитория проекта [6]. Названия таргетов для тестирования соотвествуют названию файлов.

5.2 Тестирование Algorithm

Предусмотрены тесты для основных алгоритмов, таких как:

- *QR разложение* с таргетом `test_qr_decomposition`. Тестируются 3 функции: `HouseholderQR`, `GivensQR` и `HessenberQR`.
- *Форма Хессенберга* с таргетом `test_hessenberg`. Тестируется только функция `GetHessenbergForm`.
- *Бидиагонализация* с таргетом `test_bidiag`. Тестируется только функция `Bidiagonalize`.
- *QR алгоритм* с таргетом `test_spectral`. Тестируется только функция `GetSpecDecomposition`.
- *SVD* с таргетом `test_svd`. Тестируется только функция `SVD`.

Отдельно не написаны тесты для отражений Хаусхолдера, вращений Гивенса, QR алгоритма для бидиагональной и сдвигов Уилкинсона, так как тестируемые алгоритмы внутри себя используют перечисленный функционал. Все тесты для алгоритмов написаны по 1 шаблону:

- Тест для пустой матрицы.
- Тест для квадратной матрицы.
- Тест для прямоугольной матрицы *(по возможности)*.
- Тест для `MatrixView` матрицы.
- Тест для комплексной матрицы *(по возможности)*.
- Стресс тест с комплексными *(вещественными)* матрицами.

Для каждого алгоритма реализованы отдельные функции для проверки корректности. При этом, стресс тест в том числе проверяет корректность, так как некоторые алгоритмы могут иметь проблемы со сходимостью. Все тесты проходят успешно. Ознакомиться с ними можно в папке `tests` репозитория проекта [6].

5.3 Производительность

Для таких функций, как `HouseholderQR` и `SVD`, реализованы тесты с замерами времени исполнения для квадратных комплексных матриц с размерами от 1×1 до 100×100. Таргеты можно исполнить с названиями `test_performance_qr` и `test_performance_svd`. Данные тесты итеративно генерируют случайную матрицу размера $n \times n$, где n - номер итерации, замеряют время работы алгоритма и выводят в консоль время исполнения в милисекундах. Для каждого размера запускается алгоритм с различными матрицами 10 раз и вычисляется среднее время.

5.3.1 QR разложение

Ознакомиться со временем исполнения алгоритма в зависимости от размера матрицы можно на графике:

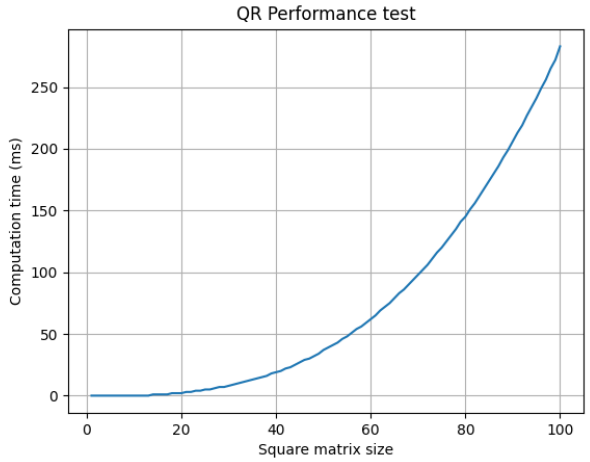


Рис. 2: Тест на производительность QR разложения.

5.3.2 SVD

Ознакомиться со временем исполнения алгоритма в зависимости от размера матрицы можно на графике:

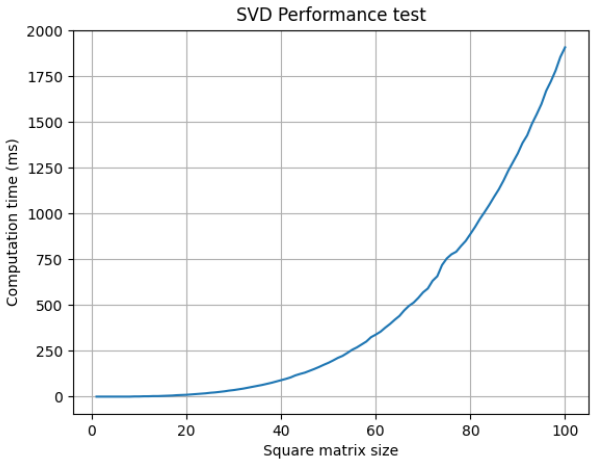


Рис. 3: Тест на производительность SVD.

Текущие замеры получены с компьютера конфигурации: AMD Ryzen™ 5 4600H, Radeon™ Graphics × 12, RAM 16 GB. Результаты замеров могут отличаться от замеров на других компьютерах.

Список литературы

[1] Walter Gander. The first algorithms to compute the svd, 2022. URL: https://people.inf.ethz.ch/gander/talks/Vortrag2022.pdf.

[2] Charles F. Van Loan Gene H. Golub. *Matrix Computations, 4Th Edn.* The Johns Hopkins University Press, 2013. URL: https://books.google.ru/books/about/Matrix_Computations.html?hl=ru&id=X5YfsuCWpxMC&redir_esc=y.

[3] Google. Googletest. URL: https://github.com/google/googletest.

[4] David Bau III Lloyd N. Trefethen. *Numerical Linear Algebra.* SIAM: Society for Industrial and Applied Mathematics, 1997. URL: https://disk.yandex.ru/i/h0Jokrx_qiLO2g.

[5] Margaret Myers Robert van de Geijn. *Advanced Linear Algebra: Foundations to Fronties.* The University of Texas at Austin, 2023. URL: https://www.cs.utexas.edu/users/flame/laff/alaff/ALAFF.html.

[6] Daniil Timizhev. Программный проект: Linearkit - библиотека для работы с матрицами. URL: https://github.com/dahbka-lis/cp-linear-kit.