Содержание

Введение **3**

1 Теоретическая часть **4**

1.1 Общие сведения о собственных векторах и собственных значениях и вычислении их на компьютере 4

1.2 Этапы вычисления 6

1.3 Первый этап. Преобразование к матрице Хессенберга с помощью редукции Хаусхолдера 7

1.4 QR разложение 9

1.5 Второй этап. Разложение Шура 11

1.6 Нахождение собственных векторов верхней треугольной матрицы 14

1.7 Матрицы основных элементарных преобразований на плоскости 15

2 Практическая часть **16**

2.1 Операции с матрицами 16

2.2 Матрица 19

2.3 Нахождение собственных векторов и собственных значений . 22

2.4 Матрицы основных элементарных преобразований на плоскости 24

Заключение **26**

Список литературы **27**

Приложение **28**

Введение

Анализ и изучение матриц и их свойств играют важную роль в различных областях науки, включая математику, физику, инженерию и компьютерные науки. Собственные векторы и собственные значения матриц являются основными характеристиками, которые позволяют нам понять структуру и свойства матриц.

Собственные векторы - это такие векторы, которые при умножении на матрицу остаются коллинеарными с исходным вектором, меняясь только в масштабе. Собственные значения, с другой стороны, являются коэффициентами масштабирования для собственных векторов.

Основная задача данной курсовой работы - создание библиотеки для работы с матрицами, изучение и реализация алгоритмов позволяющих находить все собственные вектора с их собственными значениями. Также необходимо реализовать функции, генерирующие матрицы элементарных преобразований на плоскости (поворот, отражение, проекция).

1. Теоретическая часть

1.1. Общие сведения о собственных векторах и собственных значениях и вычислении их на компьютере

Если вектор $0 \neq v \in \mathbb{C}^m$ матрицы $A \in \mathbb{C}^{m \times m}$ удовлетворяет

$$Av = \lambda v, \lambda \in \mathbb{C},$$

то такой вектор называется *собственным*, а число λ называется *собственным значением*.

Данное уравнение эквивалентно

$$Av - \lambda v = 0 \iff (A - \lambda I)v = 0.$$

Последнее равенство выполняется тогда и только тогда когда характеристический многочлен $\det(A - \lambda I) = 0$. Данный факт позволяет узнать все собственные значения матрицы найдя корни ее характеристического многочлена. У такого подхода есть две проблемы. Во-первых, вычисление корней полинома является сложной и непопулярной задачей в сфере вычислений на компьютере [1, с. 190]. Во вторых, такой подход не дает никакой дополнительной информации о собственных векторах матрицы.

Чтобы найти более перспективный подход к вычислению собственных векторов и собственных значений необходимо рассмотреть понятие *эквивалентных матриц*. Матрица $A \in \mathbb{C}^{m \times m}$ эквивалентна некоторой матрице $B \in \mathbb{C}^{m \times m}$ если выполняется

$$A = XBX^{-1},$$

где X - невырожденная матрица

У эквивалентных матриц есть свойство, что их собственные значения совпадают и если v - собственный вектор A, соответствующий собственному значению λ, то $X^{-1}v$ - собственный вектор B соответствующий тому же собственному значению [2]. Это свойство вместе с тем фактом что у треугольной матрицы все собственные значения находятся на диагонали означает, что если мы найдем такое X, что B - треугольная матрица, то узнаем все собственные значения A. Кроме того, ввиду треугольной структуры матрицы B, можно очень просто найти ее собственные вектора и, как следствие, собственные вектора A. Существование такого X гарантировано *теоремой о разложении Шура* [3, с. 351]. Данная теорема утверждает что для любой матрицы $A \in \mathbb{C}^{m\times m}$ существует невырожденная $Q \in \mathbb{C}^{m\times m}$, такая что $A = QTQ^{-1}$, причем Q унитарная матрица. Последнее позволяет переписать разложение как $A = QTQ^*$ (Q^* - *Эрмитова транспозиция* Q).

1.2. Этапы вычисления

Из соображений эффективности, преобразование матрицы осуществляется в два этапа [1, с. 193-194]. Первый этап преобразовывает матрицу в эквивалентную ей матрицу с нулями ниже первого элемента ниже главной диагонали, такая матрица называется *матрицей Хессенберга*. После этого выполняется разложение Шура. Решение о двух этапах мотивировано алгоритмом вычисляющим разложение Шура который требует намного меньшее количество операций если работает с матрицей Хессенберга.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \end{bmatrix}$$

Рис. 1.1. Этапы преобразования матрицы к треугольному виду, $\times$ - необязательно нулевой элемент

1.3. Первый этап. Преобразование к матрице Хессенберга с помощью редукции Хаусхолдера

Идея алгоритма редукции Хаусхолдера состоит в последовательном введении нулей в столбцы матрицы [1, с. 197-198]. Для этого к исходной матрице A последовательно применяются эквивалентные преобразования. Эту последовательность можно представить как

$$A \to Q_1^* A Q_1 \to Q_2^* Q_1^* A Q_1 Q_2 \to ... \to Q_{m-2}^* ... Q_2^* Q_1^* A Q_1 Q_2 ... Q_{m-2} = H,$$

где H - матрица Хессенберга.

Допустим $x \in \mathbb{C}^{(m-k)}$ - последние $m-k$ элементов k-ого столбца матрицы. Необходимо найти такую матрицу P, что

$$Px = \begin{bmatrix} \times \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Этому требованию соответствует *преобразование Хаусхолдера* [1, с. 70-73]

$$P = 2\frac{vv^*}{v^*v},$$

где $\| \dots \|$ - 2-норма вектора, а *вектор Хаусхолдера* $v = x \pm \|x\| e_1$ в действительном случае и $v = x \pm \alpha \|x\| e_1, \alpha = \frac{x_1}{|x_1|}$ в комплексном [3, с. 234-243]. С математической точки зрения в векторе v допустим любой выбор знака, но для предотвращения ошибок округления при вычислении предпочтительнее выбирать v с большей нормой. В действительном случае это $v = sign(x_1)\|x\|e_1 + x$, где $sign(x_1) = 1$ при $x_1 >= 0$ и $sign(x_1) = -1$ в остальных случаях. В комплексном это $v = \alpha\|x\|e_1 + x$.

Так как на k-ом этапе необходимо ввести изменения только в $m - k$ последних строках, не нарушив при этом ранее введенные нули, матрица Q_k^* имеет вид

$$Q_k^* = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix},$$

где $I \in \mathbb{C}^{k\times k}$, $P \in \mathbb{C}^{(m-k)\times(m-k)}$ [1, с. 70].

Такая матрица изменяет только $m-k$ последних строк при умножении на нее слева. Первые $k - 1$ столбцов имеют нули в $m - k$ последних элементах, соответственно матрица P их не изменет, эффективно подействовав лишь на нужный k-ый столбец. Q_k при умножении справа изменяет $m - k$ последних столбцов что также не разрушает ранее построенные нули. Тем что Q_k^* изменяет не все строчки можно воспользоваться при разработки кода как показано на рисунке 1.3.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix} \begin{bmatrix} \times & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \times & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ 0 & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ 0 & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ 0 & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}$$

Рис. 1.2. Шаги алгоритма редукции Хаусхолдера. Жирные элементы - те что подверглись изменению на данном шаге

```
for k=1 to m-2:
    x = A_{k+1:m,k} #нужная в данной итерации часть k-го столбца
    v^(k) = householder_vector(x) #вычисление вектора Хаусхолдера
    v^(k) = v^(k)/||v^(k)||
     #умножение на Q^(k)* слева
    A_{k+1:m,k:m} = A_{k+1:m,k:m} - 2v^(k)(v^(k)* A_{k+1:m,k:m})
     #умножение на Q^(k) справа
    A_{1:m,k+1:m} = A_{1:m,k+1:m} - 2(A_{1:m,k+1:m} v^(k) v^(k)*)
```

Рис. 1.3. Алгоритм редукции Хаусхолдера

1.4. QR разложение

Перед рассмотрением следующего шага необходимо рассказать о важнейшей состовляющей алгоритма, исполняющего второй этап вычислений, о *QR-разложении*.

Если $A = QR$, где Q - унитарная матрица, а R - верхняя триугольная, то произведение QR называется QR-разложением матрицы A. Такое разложение вычисляется похожим на редукцию Хаусхолдера способом - последовательным введением нулей в столбцы матрицы. Однако, ввиду того что мы на втором этапе работаем уже с матрицей Хессенберга, будет эффективнее вводить нули не с помощью преобразования Хаусхолдера, а с помощью *поворота Гивенса* [4, с. 67-68].

Поворот Гивенса это матрица $G \in \mathbb{C}$ вида как на рисунке 1.3, где c и $\overline{s}$ находятся на позиции (i,i) и (j,j) соответственно, $c = \cos v$, $s = \sin v$.

$$G(i,j,v) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\overline{s} & \cdots & \overline{s} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

Рис. 1.4. Поворот Гинеса

Обозначим за x_i $(i < m)$ i-ый диагональный элемент некоторой матрицы $A \in \mathbb{C}^{m \times m}$, а за x_j элемент под x_i. Если подобрать такое v, что

$$c = \frac{x_i}{\sqrt{|x_i|^2 + |x_j|^2}}, s = \frac{-\overline{x_j}}{\sqrt{|x_i|^2 + |x_j|^2}},$$

то при умножении матрицы $G(i,j,v)*$ на матрицу A слева элемент x_j станет нулем.

Поворот Гивенса, как легко видеть, является унитарной матрицей. Последовательное умножение на нужный поворот Гивенса слева преобразует матрицу Хессенберга в верхнюю треугольную как показано на рисунке 1.5. В результате получится

$$G(m-1, m, v_{m-1})^* \dots G(2,3,v_2)^* G(1,2,v_1)^* A = R \iff$$

$$\iff Q^* A = R \iff A = QR,$$

где R - верхняя треугольная матрица.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{G(1,2,v_1)^*} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

$$\xrightarrow{G(2,3,v_2)^*} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{G(3,4,v_3)^*} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

Рис. 1.5. Преобразование матрицы Хессенберга в вернхнюю треугольную путем последовательного умножения на поворот Гивенса слева
[4, с. 68]

Учитывая вид матрицы Гивеса, имеем алгоритм QR разложения, представленный на рисунке 1.6 [4, с. 70].

```
for k = 1,2, ... ,n-1:
    #вычисляем матрицу Гивенса по коэффициентам c и s
    G^(k) = givens(H_{k,k}, H_{k-1,k})
    H_{k:k+1,k:m} = G^(k)_{k:k+1,k:k+1}^* H_{k:k+1,k:n}
```

Рис. 1.6. Алгоритм QR разложения матрицы Хессенберга с помощью поворота Гивенса

1.5. Второй этап. Разложение Шура

Классическим алгоритмом для выполнения разложения Шура является *qr-алгоритм*. Базовый вариант алгоритма представлен на рисунке 1.7 [1, с. 211]. Легко заметить что H_k эквивалентна H_{k-1}:

$$H_k = R_k Q_k \iff H_k = Q_k^* H_{k-1} Q_k \iff H_{k-1} = Q_k H_k Q_k^*.$$

```
H_0 = H
for k = 1,2, ... :
    Q_k R_k = H_{k-1} #QR разложение A_k - 1
    H_k = R_k Q_k
```

Рис. 1.7. Базовый QR алгоритм

Главная проблема базового QR алгоритма это работа исключительно с действительными числами если $A \in \mathbb{R}^{m \times m}$. Треугольная матрица подразумевает наличие всех ее собственных значений на главной диагонали. Если матрица A имеет комплексные собственные значения, то действительная треугольную матрица эквивалентная ей не существует. Проблема решается введением в QR алгоритм комплексного *сдвига* σ, получаем алгоритм из рисунке 1.8. Эквивалентность H_k и H_{k-1} сохраняется.

$$H_k = R_k Q_k + \sigma I = (Q_k^* H_{k-1} - \sigma Q_k^*) Q_k + \sigma I = Q_k^* H_{k-1} Q_k \iff$$

$$\iff H_{k-1} = Q_k H_k Q_k^*.$$

```
H_0 = H
for k = 1,2, … :
    Q_k R_k = H_{k-1} − σI #QR разложение A_k − 1
    H_k = R_k Q_k + σI.
```

$$
\begin{aligned}
& H_0 = H \\
& \texttt{for k = 1,2, ... :} \\
& \quad Q_k R_k = H_{k-1} - \sigma I \ \texttt{\#QR разложение} \ A_k - 1 \\
& \quad H_k = R_k Q_k + \sigma I.
\end{aligned}
$$

Рис. 1.8. QR алгоритм со сдвигом

Сдвиг можно взять любой, но чем ближе он окажется к собственному значению матрицы, находящемуся на n-ой позиции треугольной матрицы эквивалентной H, тем быстрее матрица $H^{(k)}$ примет триугольны вид [3, с. 386-387]. Эвристические наблюдения показывают, что хорошим выбором является *сдвиг Уилкинсона* [4, с. 76], представляющий собой собственное значение части матрицы, изображенной на рисунке 1.9. Вычислим собственные значение матрицы W через нахождение корней характерестического уравнения.

$$\det(W - \lambda I) = 0 \iff (h_{n-1,n-1} - \lambda)(h_{n,n} - \lambda) - h_{n-1,n}h_{n,n-1} = 0 \iff$$

$$\iff \lambda^2 - \lambda(h_{n-1,n-1} + h_{n,n}) + (h_{n-1,n-1}h_{n,n} - h_{n-1,n}h_{n,n-1}) = 0 \iff$$

$$\iff \lambda^2 - \lambda \mathrm{Tr}(W) + \det(W) = 0 \iff \lambda_{1,2} = \frac{\mathrm{Tr}(W) \pm \sqrt{\mathrm{Tr}(W)^2 - 4\det(W)}}{2}.$$

$$W = \begin{bmatrix} h_{n-1,n-1} & h_{n-1,n} \\ h_{n,n-1} & h_{n,n} \end{bmatrix}$$

Рис. 1.9. Подматрица, используемая для вычисления сдвига Уилкинсона

Таким образом, итоговый алгоритм для вычисления разложения Шура представлен на рисунке 1.10. В практическом алгоритме обоснованно выбрано условие окончания цикла и введен механизм дефляции [4, с. 75].

```
H^(0) = H
k = 0
for p = n,n-1, … 2:
    k = k + 1
    do:
        #нахождение сдвига Уилкинсона по формуле выведеной выше
        σ = wilkinson_shift(H_{p-1:p,p-1:p})
        Q^(k)R^(k) = H^(k-1) - σI #QR разложение H^{k-1}
        H^(k) = R^(k)Q^(k) + σI
    until |h^(k)_{p,p-1}| is close to zero
```

Рис. 1.10. Практический QR алгоритм с применением сдвига Уилкинсона и дефляции

1.6. Нахождение собственных векторов верхней треугольной матрицы

В верхней треугольной матрицы все ее собственные значения находятся на главной диаганали. Следовательно все ее собственные вектора можно найти решив для каждого собственного значения уравнение

$$(A-\lambda I)v = 0 \iff \begin{bmatrix} a_{11}-\lambda & a_{12} & \cdots & a_{1n} \\ 0 & a_{22}-\lambda & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn}-\lambda \end{bmatrix} v = 0.$$

Такое уравнение порождает систему

$$\begin{cases} (a_{11}-\lambda)x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ (a_{22}-\lambda)x_2 + \cdots + a_{2n}x_n = 0 \\ \ldots \\ (a_{nn}-\lambda)x_n = 0 \end{cases}.$$

Решая систему сверху вниз относительно $v = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^*$, получим формулу для x_i:

$$x_i = -\frac{a_{i,i+1}x_{i+1} + \cdots + a_{in}x_n}{a_{ii}-\lambda}.$$

1.7. Матрицы основных элементарных преобразований на плоскости

Поворот вектора, принадлежащего $\mathbb{R}^2$ на θ по часовой стрелке можно реализовать как частный случай поворота Гивенса как на рисунке 1.11.

$$rot(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Рис. 1.11. Матрица поворота на θ радиан против часовой стрелки

Матрица проекции на прямую, направленую вектором d, представляет собой dd^* [1, с. 46].

Матрицу P отражения относительно прямой, направленую вектором d, выведем через ее собственные вектора. $d_1 = \frac{d}{\|d\|}$ - собственный вектор P с собственным значением $\lambda_1 = 1$. Вектор n_1, орторгональный d_1, равен

$$n_1 = rot(\frac{\pi}{2})d_1 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} d_1 = \begin{bmatrix} 0 \\ (d_1)_1 \end{bmatrix} + \begin{bmatrix} -(d_1)_2 \\ 0 \end{bmatrix} = \begin{bmatrix} -(d_1)_2 \\ (d_1)_1 \end{bmatrix}.$$

Вектор n_1 является собственным с собственным значением $\lambda_2 = -1$.

Пусть $Q = \left[\begin{smallmatrix} d_1 & n_1 \end{smallmatrix}\right]$, тогда Q - ортогональная матрица и

$$PQ = \begin{bmatrix} Pd_1 & Pn_1 \end{bmatrix} = \begin{bmatrix} \lambda_1 d_1 & \lambda_2 n_1 \end{bmatrix} = \begin{bmatrix} d_1 & -n_1 \end{bmatrix} \implies$$

$$\implies P = \begin{bmatrix} d_1 & -n_1 \end{bmatrix} \begin{bmatrix} d_1^* \\ n_1^* \end{bmatrix} = d_1 d_1^* - n_1 n_1^*.$$

2. Практическая часть

2.1. Операции с матрицами

Самым очевидным способом реализации базовых операция с матрицами является создание класса матриц и функций возвращающих матрицу полученную после выполнения операции, но такой подход нельзя назвать оптимизированным. Допустим необходимо выполнить несколько операций как показано на листинге 2.1. Сначала вычислиться матрица temp = m1 + m2 и только потом temp - m3. В ходе операции в памяти будет храниться совершенно ненужная матрица temp, кроме того такое вычисление потребует два цикла, хотя вычислить результат можно за один проход.

Вместо прямолинейного подхода мы будем использовать метод под названием *Expression template* [5]. Суть метода заключается в возвращении вместо матрицы объекта, представляющего операцию и хранящего информацию о том над какими объектами производится операция. Для больших матриц Expression template показывает более высокую производительность чем прямолинейный подход.

```
m1 + m2 - m3
```

Листинг 2.1. пример операий с матрицами

Самый базовый класс библиотеки матриц, представляющий собой абстрактную операцию над объектами, является класс *MatrixExpression*. Шаблон класса принимает тип T элементов матрицы, получаемой в результате вычисления выражения, и тип E своего сабкласса для делегации исполнения методов. Статическое поле *has_data* принимает значение *true* если класс содержит данные, необходимые для исполнения операции и *false* в обратном случае.

```cpp
template<typename T, typename E>
class MatrixExpression {
public:
    using value_type = T;

    static constexpr bool has_data = false;

    T operator[](std::size_t i, std::size_t j) const;

    std::size_t n() const;
    std::size_t m() const;
};
```

Листинг 2.2. Декларация класса MatrixExpression

Операции взятие обратного, сложения, отрицания, умножения, скалярного умножения и скалярного деления представляются в виде соответствующих подклассов. Разберем их структуру на примере класса сложения *Summation*.

Шаблон класса принимает тип элемента матрицы получаемой в результате исполнения сложения и типы объектов которые необходимо сложить. Конструктор класса принимает на вход два объекта класса *MatrixExpression* и, в зависимости от того содержит класс внутри себя данные или нет, сохраняет в полях копии объектов или ссылки на объект. Метод *operator[]* возвращает копию элемента матрицы, получаемой вычислением суммы *first* и *second*, который находится в i-ой строке j-го столбца. Методы *n* и *m* возвращают количество строк и столбцов соответственно.

```cpp
template<typename T, typename E1, typename E2>
class Summation : public MatrixExpression<T, Summation<T, E1, E2>> {
```

```
protected:
    std::conditional_t<E1::has_data, const E1&, E1> first;
    std::conditional_t<E2::has_data, const E2&, E2> second;
public:
    T operator[](std::size_t i, std::size_t j) const;

    std::size_t n() const;
    std::size_t m() const;

    template<typename T1, typename T2>
    Summation(
    const MatrixExpression<T1, E1> &first,
    const MatrixExpression<T2, E2> &second);
};
```

Листинг 2.3. Декларация класса *Summation*

Для каждого класса операции существует соответствующая функция. Разберем их структуру на примере функции для *Summation*.

Функция принимает на вход два объекта класса операции и возвращает объект класса *Summation* с информацией о складываемых объектах.

```
template<typename T, typename E1, typename E2>
Summation<T, E1, E2> operator+(
const MatrixExpression<T, E1> &first,
const MatrixExpression<T, E2> &second) {
    return Summation<T, E1, E2>(first, second);
}
```

Листинг 2.4. Декларация и реализация функции *operator+*

2.2. Матрица

Матрица в библиотеке представляется в виде класса *Matrix*. Класс хранит поля с количеством строк и столбцов и содержимым матрицы ввиде вложенных объектов класса *std::vector*. Методы класса позволяют получать доступ к элементам и целым подмасивам. Доступ к подмасивам осуществляется путем возврата объекта класса *Submatrix*, хранящего ссылку на исходный массив и границы определяющие часть массива. Методы этого класса позволяют изменять содержимое подматрицы путем сложения, вычитания с другой подматрицей, умножения на скаляр, деления на скаляр. Для задания границ подмассива определяется структура *Slice*, хранящяя начало среза и его конец. Значение статического поля *has_data* в классе *Matrix* является *true*. Это указывает на то, что при передачи объекта этого класса необходимо хранить ссылку на него, а не копию объекта.

```
template<typename T1>
class Matrix : public MatrixExpression<T1, Matrix<T1>> {
private:
    std::size_t N;
    std::size_t M;

protected:
    std::vector<std::vector<T1>> data;

public:
    static constexpr bool has_data = true;

    T1 operator[](std::size_t i, std::size_t j) const;

    T1& operator[](std::size_t i, std::size_t j);
```

```
    ConstSubmatrix<T1> operator[](std::size_t i) const;
    Submatrix<T1> operator[](std::size_t i);

    ConstSubmatrix<T1> operator[](Slice n_slice) const;
    Submatrix<T1> operator[](Slice n_slice);

    ConstSubmatrix<T1> operator[](Slice n_slice, std::size_t m) const;
    Submatrix<T1> operator[](Slice n_slice, std::size_t m);

    ConstSubmatrix<T1> operator[](std::size_t i, Slice m_slice) const;
    Submatrix<T1> operator[](std::size_t i, Slice m_slice);

    ConstSubmatrix<T1> operator[](Slice n_slice, Slice m_slice) const;
    Submatrix<T1> operator[](Slice n_slice, Slice m_slice);
    ...
};
```

Листинг 2.5. Часть декларации класса *Matrix*

```
template<typename T1>
class Submatrix :
public AbstractSubmatrix<std::vector<std::vector<T1>>, Submatrix<T1>> {
    using AbstractSubmatrix<
    std::vector<std::vector<T1>>,
    Submatrix<T1>>::AbstractSubmatrix;
public:
    T1& operator[](std::size_t i, std::size_t j);

```

```
    template<typename T2, typename E2>
    Submatrix operator=(const MatrixExpression<T2, E2> &other);

    template<typename T2, typename E2>
    Submatrix operator+=(const MatrixExpression<T2, E2> &other);

    template<typename T2, typename E2>
    Submatrix operator-=(const MatrixExpression<T2, E2> &other);

    Submatrix operator*=(T1 val);

    Submatrix operator/=(T1 val);
};
```

Листинг 2.6. Декларация класса *Submatrix*

```
struct Slice {
    std::size_t start;
    std::size_t end;

    Slice();
    Slice(std::size_t start, std::size_t end);
};
```

Листинг 2.7. Декларация структуры *Slice*

2.3. Нахождение собственных векторов и собственных значений

Все собственные вектора и их собственные значения находятся с помощью функции *eigenpairs* которая возвращает их ввиде массива пар типа "число, вектор". Функция прямо реализует алгоритм нахождения, описанный в первой главе. Код вместе с примечаниями находится в листинге 2.8.

```
template<typename T>
std::vector<std::pair<std::complex<T>, Matrix<std::complex<T>>>>
eigenpairs(Matrix<std::complex<T>> matrix) {
    if (matrix.n() != matrix.m()) {
        throw std::invalid_argument("only square matrix is allowed");
    }
    // Заменяет матрицу эквивалентной матрицей Хессенберга,
    // находит унитарную матрицу, преобразовывающую исходную
    auto Q1 = hessenberg(matrix);
    // Заменяет матрицу эквивалентной верхней треугольной,
    // находит унитарную матрицу, преобразовывающую исходную
    auto Q2 = complex_schur(matrix);
    // единичная матрица размера n
    auto id = identity<std::complex<T>>(matrix.n());
    auto result = std::vector<std::pair<std::complex<T>,
    Matrix<std::complex<T>>>>(matrix.n());
    for (int i = 0; i < matrix.n(); ++i) {
        // собственное значение матрицы, расположенное на i-ом элементе
        // главной диаганали эквивалентной ей треугольной
        auto eigenvalue = matrix[i, i];
        // находит собственный вектор треугольной матрицы и,
        // с помощью унитарных преобразований,
```

```
        //находит соответствующий ему собственный вектор исходной
        Matrix<std::complex<T>> eigenvector =
        Q1 * (Q2 * schur_eigenvector(matrix, i));
        result[i] = (std::make_pair(eigenvalue, eigenvector));
    }
    return result;
}
```

Листинг 2.8. Функция *eigenpairs*

2.4. Матрицы основных элементарных преобразований на плоскости

Матрицы элементарных пробразований на плоскости (поворот, проекция, отражение) реализованы в виде функций точно в соответствии с параграфом 1.7.

```
template<typename T>
Matrix<T> rotation_matrix(T angle) {
    auto result = Matrix<T>(2);
    result[0,0] = std::cos(angle);
    result[0,1] = -std::sin(angle);
    result[1,0] = std::sin(angle);
    result[1,1] = std::cos(angle);
    return result;
}

template<typename T>
Matrix<T> projection_matrix(Matrix<T> direction) {
    direction /= T(norm(direction));
    return direction * conj(direction);
}

template<typename T>
Matrix<T> reflection_matrix(Matrix<T> direction) {
    direction = direction / T(norm(direction));
    auto orthogonal = Matrix<T>(2,1);
    orthogonal[0,0] = -direction[1,0];
    orthogonal[1,0] = direction[0,0];
```

```
    return direction * conj(direction) - orthogonal * conj(orthogonal);
}
```

Листинг 2.9. функции генерирующие матрицы элементарных преобразований на плоскости

Заключение

В рамках курсовой работы были реализованы библиотека для работы с матрицами, функции генерирующие матрицы элементарных преобразований на плоскости. Реализован один из алгоритмов нахождения всех собственных значений и собственных векторов, изучена и разобрана его математическая база.

Список литературы

1. *Trefethen, L.N.* Numerical Linear Algebra / L.N. Trefethen, D. Bau — Society for Industrial and Applied Mathematics — 1997.

2. *Watkins, David S.* The QR algorithm revisited / David S. Watkins — 2008 — Pp. 2-3.

3. *Golub, G.H.* Matrix Computations / G.H. Golub, C.F. Van Loan — Johns Hopkins University Press — 2013.

4. *Arbenz, Peter*. Lecture notes on solving large scale eigenvalue problems / Peter Arbenz, Daniel Kressner, DME Zürich — 2012.

5. *Veldhuizen, Todd*. Expression templates / Todd Veldhuizen // *C++ Report* — 1995 — Pp. 10-11.

Приложение

```
// файл matrix/matrix-expression/matrix-expression.h

#ifndef MATRIX_CALCULATOR_MATRIXEXPRESSION_H
#define MATRIX_CALCULATOR_MATRIXEXPRESSION_H

// abstract class of expression with matrices
// T - type of elements of matrix obtained by evaluating expression
// E - subclass of MatrixExpression
template<typename T, typename E>
class MatrixExpression {
public:
    using value_type = T;

    // has_data equals true if class contains data (not reference to data)
↪   about elements of matrix
    static constexpr bool has_data = false;

    // returns copy of element on i-th row and j-th column of matrix
↪   obtained by evaluating expression
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating expression
    std::size_t n() const;
    std::size_t m() const;
};
```

```
// throw exception if matrices are not match by row count, column count or
↪  column-row count respectively
template<typename T1, typename E1, typename T2, typename E2>
void check_n(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second);

template<typename T1, typename E1, typename T2, typename E2>
void check_m(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second);

template<typename T1, typename E1, typename T2, typename E2>
void check_mn(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second);

// class of negation operation
// T - type of elements of matrix obtained by evaluating negation of the
↪  expression
// E - type of expression for negation
template<typename T, typename E>
class Negation : public MatrixExpression<T, Negation<T, E>> {
protected:
    std::conditional_t<E::has_data, const E&, E> expression;

public:
    // returns copy of element on i-th row and j-th column of matrix
↪  obtained by evaluating negation of the expression
    T operator[](std::size_t i, std::size_t j) const;
```

```
    // return size of matrix obtained by evaluating negation of the
    ↪ expression
    std::size_t n() const;
    std::size_t m() const;

    explicit Negation(const MatrixExpression<T, E>& expression);
};

// class of summation operation
// T - type of elements of matrix obtained by evaluating summation of the
↪ expressions
// E1, E2 - types of expressions for summation
template<typename T, typename E1, typename E2>
class Summation : public MatrixExpression<T, Summation<T, E1, E2>> {
protected:
    std::conditional_t<E1::has_data, const E1&, E1> first;
    std::conditional_t<E2::has_data, const E2&, E2> second;
public:
    // returns copy of element on i-th row and j-th column of matrix
    ↪ obtained by evaluating summation of the expressions
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating summation of the
    ↪ expressions
    std::size_t n() const;
    std::size_t m() const;

    template<typename T1, typename T2>
```

```
    Summation(const MatrixExpression<T1, E1> &first, const
↪   MatrixExpression<T2, E2> &second);
};

// class of subtraction operation
// T - type of elements of matrix obtained by evaluating subtraction of the
↪   expressions
// E1, E2 - types of expressions for subtraction
template<typename T, typename E1, typename E2>
class Subtraction : public MatrixExpression<T, Subtraction<T, E1, E2>> {
protected:
    std::conditional_t<E1::has_data, const E1&, E1> first;
    std::conditional_t<E2::has_data, const E2&, E2> second;

public:
    // returns copy of element on i-th row and j-th column of matrix
↪   obtained by evaluating subtraction of the expressions
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating subtraction of the
↪   expressions
    std::size_t n() const;
    std::size_t m() const;

    template<typename T1, typename T2>
    Subtraction(const MatrixExpression<T1, E1> &first, const
↪   MatrixExpression<T2, E2> &second);
};
```

```
// class of product operation
// T - type of elements of matrix obtained by evaluating product of the
↪  expressions
// E1, E2 - types of expressions for product
template<typename T, typename E1, typename E2>
class Product : public MatrixExpression<T, Product<T, E1, E2>> {
protected:
    std::conditional_t<E1::has_data, const E1&, E1> first;
    std::conditional_t<E2::has_data, const E2&, E2> second;

public:
    // returns copy of element on i-th row and j-th column of matrix
↪  obtained by evaluating product of the expressions
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating product of the
↪  expressions
    std::size_t n() const;
    std::size_t m() const;

    template<typename T1, typename T2>
    Product(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second);
};

// class of product of matrix and scalar
```

```
// T - type of elements of matrix obtained by evaluating product of the
↪  expression and scalar
// E - type of expression for product
// V - type of scalar for product
template<typename T, typename E, typename V>
class ScalarProduct : public MatrixExpression<T, ScalarProduct<T, E, V>> {
protected:
    std::conditional_t<E::has_data, const E&, E> expression;
    V val;
public:
    // returns copy of element on i-th row and j-th column of matrix
↪  obtained by evaluating product of the expression and scalar
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating product of the
↪  expression and scalar
    std::size_t n() const;
    std::size_t m() const;

    ScalarProduct(const MatrixExpression<T, E> &expression, V val);
};

// class of division of matrix by scalar
// T - type of elements of matrix obtained by evaluating division of the
↪  expression by scalar
// E - type of expression for division
// V - type of scalar for division
template<typename T, typename E, typename V>
```

```
class ScalarDivision : public MatrixExpression<T, ScalarDivision<T, E, V>> {
private:
    std::conditional_t<E::has_data, const E&, E> expression;
    V val;

public:
    // returns copy of element on i-th row and j-th column of matrix
↪   obtained by evaluating division of the expression by scalar
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating division of the
↪   expression by scalar
    std::size_t n() const;
    std::size_t m() const;

    ScalarDivision(const MatrixExpression<T, E> &expression, V val);
};

// expression surface-operations functions //
// NOTE: for reasons of possibility of deduction of return type of
↪   surface-operations,
// those functions can be used only with expressions/scalars with a same
↪   value_type

template<typename T, typename E>
Negation<T, E> operator-(const MatrixExpression<T, E> &expression);

template<typename T, typename E1, typename E2>
```

```
Summation<T, E1, E2> operator+(const MatrixExpression<T, E1> &first, const
↪ MatrixExpression<T, E2> &second);

template<typename T, typename E1, typename E2>
Subtraction<T, E1, E2> operator-(const MatrixExpression<T, E1> &first, const
↪ MatrixExpression<T, E2> &second);

template<typename T, typename E1, typename E2>
Product<T, E1, E2> operator*(const MatrixExpression<T, E1> &first, const
↪ MatrixExpression<T, E2> &second);

template<typename T, typename E>
ScalarProduct<T, E, T> operator*(const MatrixExpression<T, E> &expression, T
↪ val);

template<typename T, typename E>
ScalarProduct<T, E, T> operator*(T val, const MatrixExpression<T, E>
↪ &expression);

template<typename T, typename E>
ScalarDivision<T, E, T> operator/(const MatrixExpression<T, E> &expression,
↪ T val);

// output function //
template<typename T, typename E>
std::ostream& operator<<(std::ostream &ostream, const MatrixExpression<T, E>
↪ &expression);
```

```
#include "matrix-expression.tpp"

#endif //MATRIX_CALCULATOR_MATRIXEXPRESSION_H


// файл matrix/matrix-expression/matrix-expression.tpp


#include <iomanip>


// MatrixExpression implementation //


template<typename T, typename E>

T MatrixExpression<T, E>::operator[](std::size_t i, std::size_t j) const {

    return static_cast<const E&>(*this)[i, j];

}


template<typename T, typename E>

std::size_t MatrixExpression<T, E>::n() const {

    return static_cast<const E&>(*this).n();

}


template<typename T, typename E>

std::size_t MatrixExpression<T, E>::m() const {

    return static_cast<const E&>(*this).m();

}


// matrices compatibility functions //


template<typename T1, typename E1, typename T2, typename E2>
```

```
void check_n(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second) {
    if (first.n() != second.n()) {
        throw std::invalid_argument("matrices rows don't match");
    }
}

template<typename T1, typename E1, typename T2, typename E2>
void check_m(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second) {
    if (first.m() != second.m()) {
        throw std::invalid_argument("matrices columns don't match");
    }
}

template<typename T1, typename E1, typename T2, typename E2>
void check_mn(const MatrixExpression<T1, E1> &first, const
↪  MatrixExpression<T2, E2> &second) {
    if (first.m() != second.n() || first.m() == 0) {
        throw std::invalid_argument("matrices columns and rows don't
↪  match");
    }
}

// Negation implementation //

template<typename T, typename E>
T Negation<T, E>::operator[](std::size_t i, std::size_t j) const {
```

```
    return -expression[i, j];
}

template<typename T, typename E>
std::size_t Negation<T, E>::n() const {
    return expression.n();
}

template<typename T, typename E>
std::size_t Negation<T, E>::m() const {
    return expression.m();
}

template<typename T, typename E>
Negation<T, E>::Negation(const MatrixExpression<T, E>& expression_) :
 ↪  expression(static_cast<const E&>(expression_)) {}

// Summation implementation //

template<typename T, typename E1, typename E2>
T Summation<T, E1, E2>::operator[](std::size_t i, std::size_t j) const {
    return first[i, j] + second[i, j];
}

template<typename T, typename E1, typename E2>
std::size_t Summation<T, E1, E2>::n() const {
    return first.n();
}
```

```
template<typename T, typename E1, typename E2>
std::size_t Summation<T, E1, E2>::m() const {
    return second.m();
}

template<typename T, typename E1, typename E2>
template<typename T1, typename T2>
Summation<T, E1, E2>::Summation(const MatrixExpression<T1, E1> &first_,
↪  const MatrixExpression<T2, E2> &second_) :
first(static_cast<const E1&>(first_)), second(static_cast<const
↪  E2&>(second_)) {
    check_n(first, second);
    check_m(first, second);
}

// Subtraction implementation //

template<typename T, typename E1, typename E2>
T Subtraction<T, E1, E2>::operator[](std::size_t i, std::size_t j) const {
    return first[i, j] - second[i, j];
}

template<typename T, typename E1, typename E2>
std::size_t Subtraction<T, E1, E2>::n() const {
    return first.n();
}
```

```
template<typename T, typename E1, typename E2>
std::size_t Subtraction<T, E1, E2>::m() const {
    return second.m();
}

template<typename T, typename E1, typename E2>
template<typename T1, typename T2>
Subtraction<T, E1, E2>::Subtraction(const MatrixExpression<T1, E1> &first_,
↪   const MatrixExpression<T2, E2> &second_) :
first(static_cast<const E1&>(first_)), second(static_cast<const
↪   E2&>(second_)) {
    check_n(first, second);
    check_m(first, second);
}

// Product implementation //

template<typename T, typename E1, typename E2>
T Product<T, E1, E2>::operator[](std::size_t i, std::size_t j) const {
    T result = T(0);
    for (int k = 0; k < first.m(); ++k) {
        result += first[i, k] * second[k, j];
    }
    return result;
}

template<typename T, typename E1, typename E2>
std::size_t Product<T, E1, E2>::n() const {
```

```
    return first.n();
}

template<typename T, typename E1, typename E2>
std::size_t Product<T, E1, E2>::m() const {
    return second.m();
}

template<typename T, typename E1, typename E2>
template<typename T1, typename T2>
Product<T, E1, E2>::Product(const MatrixExpression<T1, E1> &first_, const
↪  MatrixExpression<T2, E2> &second_) :
first(static_cast<const E1&>(first_)), second(static_cast<const
↪  E2&>(second_)) {
    check_mn(first, second);
}

// ScalarProduct implementation//

template<typename T, typename E, typename V>
T ScalarProduct<T, E, V>::operator[](std::size_t i, std::size_t j) const {
    return expression[i, j] * val;
}

template<typename T, typename E, typename V>
std::size_t ScalarProduct<T, E, V>::n() const {
    return expression.n();
}
```

```cpp
template<typename T, typename E, typename V>
std::size_t ScalarProduct<T, E, V>::m() const {
    return expression.m();
}

template<typename T, typename E, typename V>
ScalarProduct<T, E, V>::ScalarProduct(const MatrixExpression<T, E>
↪   &expression_, V val_) :
expression(static_cast<const E&>(expression_)), val(val_) {}

// ScalarDivision implementation //

template<typename T, typename E, typename V>
T ScalarDivision<T, E, V>::operator[](std::size_t i, std::size_t j) const {
    return expression[i, j] / val;
}

template<typename T, typename E, typename V>
std::size_t ScalarDivision<T, E, V>::n() const {
    return expression.n();
}

template<typename T, typename E, typename V>
std::size_t ScalarDivision<T, E, V>::m() const {
    return expression.m();
}
```

```
template<typename T, typename E, typename V>
ScalarDivision<T, E, V>::ScalarDivision(const MatrixExpression<T, E>
 ↪  &expression_, V val_) :
expression(static_cast<const E&>(expression_)), val(val_) {}

// expression surface-operations functions implementation //

template<typename T, typename E>
Negation<T, E> operator-(const MatrixExpression<T, E> &expression) {
    return Negation<T, E>(expression);
}

template<typename T, typename E1, typename E2>
Summation<T, E1, E2> operator+(const MatrixExpression<T, E1> &first, const
 ↪  MatrixExpression<T, E2> &second) {
    return Summation<T, E1, E2>(first, second);
}

template<typename T, typename E1, typename E2>
Subtraction<T, E1, E2> operator-(const MatrixExpression<T, E1> &first, const
 ↪  MatrixExpression<T, E2> &second) {
    return Subtraction<T, E1, E2>(first, second);
}

template<typename T, typename E1, typename E2>
Product<T, E1, E2> operator*(const MatrixExpression<T, E1> &first, const
 ↪  MatrixExpression<T, E2> &second) {
    return Product<T, E1, E2>(first, second);
```

```
}

template<typename T, typename E>
ScalarProduct<T, E, T> operator*(const MatrixExpression<T, E> &expression, T
↪  val) {
    return ScalarProduct<T, E, T>(expression, val);
}

template<typename T, typename E>
ScalarProduct<T, E, T> operator*(T val, const MatrixExpression<T, E>
↪  &expression) {
    return expression * val;
}

template<typename T, typename E>
ScalarDivision<T, E, T> operator/(const MatrixExpression<T, E> &expression,
↪  T val) {
    return ScalarDivision<T, E, T>(expression, val);
}

// output function implementation //

template<typename T>
std::size_t number_length(T num) {
    std::stringstream ss;
    ss << num;
    return ss.str().length();
}
```

```cpp
template<typename T, typename E>
std::ostream& operator<<(std::ostream &ostream, const MatrixExpression<T, E>
↪   &expression) {
    if (expression.n() > 0) {
        std::vector<std::size_t> columns_lengths =
↪   std::vector<std::size_t>(expression.m());
        for (int j = 0; j < expression.m(); ++j) {
            for (int i = 0; i < expression.n(); ++i) {
                std::size_t element_length = number_length(expression[i,
↪   j]);
                if (element_length > columns_lengths[j]) {
                    columns_lengths[j] = element_length;
                }
            }
        }

        for (int j = 0; j < expression.m(); ++j) {
            ostream << std::setw(columns_lengths[j] + 3) << std::left <<
↪   expression[0, j];
        }
        for (int i = 1; i < expression.n(); ++i) {
            ostream << '\n';
            for (int j = 0; j < expression.m(); ++j) {
                ostream << std::setw(columns_lengths[j] + 3) << std::left <<
↪   expression[i, j];
            }
        }
```

```
    }
    return ostream;
}

// файл matrix/matrix.h

#ifndef MATRIX_CALCULATOR_MATRIX_H
#define MATRIX_CALCULATOR_MATRIX_H

#include <vector>
#include "matrix-expression/matrix-expression.h"

// structure of slice
struct Slice {
    std::size_t start;
    std::size_t end;

    Slice();
    Slice(std::size_t start, std::size_t end);
};

// abstract class of submatrix associated with matrix
// Cont - type of nested container for inner representation of matrix data
// E - subclass of AbstractSubmatrix
template<typename Cont, typename E>
class AbstractSubmatrix : public MatrixExpression<typename
 ↪  Cont::value_type::value_type, AbstractSubmatrix<Cont, E>> {
private:
```

```
    Slice n_slice; // horizontal bound of submatrix
    Slice m_slice; // vertical bound of submatrix

protected:
    Cont &data; // reference to data of a matrix

public:
    // type of elements of matrix
    using value_type = Cont::value_type::value_type;

    // returns copy of element on i-th row and j-th column of submatrix
    value_type operator[](std::size_t i, std::size_t j) const;

    // substitute elements of matrix with elements of "other"
    AbstractSubmatrix operator=(const AbstractSubmatrix &other);

    // return size of submatrix
    std::size_t n() const;
    std::size_t m() const;

    // return bounds of submatrix
    std::size_t n_start() const;
    std::size_t n_end() const;
    std::size_t m_start() const;
    std::size_t m_end() const;

    explicit AbstractSubmatrix(Cont &data);
    AbstractSubmatrix(Cont &data, Slice n_slice);
```

```
    AbstractSubmatrix(Cont &data, Slice n_slice, Slice m_slice);
};

// Submatrix with no possibility of changing elements of matrix
// T - type of elements of matrix
template<typename T>
class ConstSubmatrix : public AbstractSubmatrix<const
↪   std::vector<std::vector<T>>, ConstSubmatrix<T>> {
    // inheriting all constructors
    using AbstractSubmatrix<const std::vector<std::vector<T>>,
↪   ConstSubmatrix<T>>::AbstractSubmatrix;
};

// Submatrix with possibility of changing elements of matrix
// T - type of elements of original matrix
template<typename T1>
class Submatrix : public AbstractSubmatrix<std::vector<std::vector<T1>>,
↪   Submatrix<T1>> {
    // inheriting all constructors
    using AbstractSubmatrix<std::vector<std::vector<T1>>,
↪   Submatrix<T1>>::AbstractSubmatrix;
public:
    // returns reference to element of matrix on i-th row and j-th column
    T1& operator[](std::size_t i, std::size_t j);

    // substitute elements of matrix with elements of "other"
    template<typename T2, typename E2>
    Submatrix operator=(const MatrixExpression<T2, E2> &other);
```

```
    // surface-operations which change matrix
    template<typename T2, typename E2>
    Submatrix operator+=(const MatrixExpression<T2, E2> &other);

    template<typename T2, typename E2>
    Submatrix operator-=(const MatrixExpression<T2, E2> &other);

    Submatrix operator*=(T1 val);

    Submatrix operator/=(T1 val);
};

// class of matrix
// T - type of elements of matrix
template<typename T1>
class Matrix : public MatrixExpression<T1, Matrix<T1>> {
private:
    std::size_t N;
    std::size_t M;

protected:
    std::vector<std::vector<T1>> data;

public:
    // class Matrix contains data
    static constexpr bool has_data = true;
```

```
// returns copy of element on i-th row and j-th column
T1 operator[](std::size_t i, std::size_t j) const;

// returns reference to element on i-th row and j-th column
T1& operator[](std::size_t i, std::size_t j);

// return i-th row
ConstSubmatrix<T1> operator[](std::size_t i) const;
Submatrix<T1> operator[](std::size_t i);

// return submatrix with rows included in "n_slice"
ConstSubmatrix<T1> operator[](Slice n_slice) const;
Submatrix<T1> operator[](Slice n_slice);

// return submatrix with part of m-th column with elements included in
↪ "n_slice"
ConstSubmatrix<T1> operator[](Slice n_slice, std::size_t m) const;
Submatrix<T1> operator[](Slice n_slice, std::size_t m);

// return submatrix with part of i-th row with elements included in
↪ "m_slice"
ConstSubmatrix<T1> operator[](std::size_t i, Slice m_slice) const;
Submatrix<T1> operator[](std::size_t i, Slice m_slice);

// return submatrix
ConstSubmatrix<T1> operator[](Slice n_slice, Slice m_slice) const;
Submatrix<T1> operator[](Slice n_slice, Slice m_slice);
```

```
// return size of matrix
std::size_t n() const;
std::size_t m() const;

// substitute elements of matrix with elements of "other"
template<typename T2, typename E2>
Matrix& operator=(const MatrixExpression<T2, E2> &other);

// surface-operations which change matrix
template<typename T2, typename E2>
Matrix& operator+=(const MatrixExpression<T2, E2> &other);

template<typename T2, typename E2>
Matrix& operator-=(const MatrixExpression<T2, E2> &other);

Matrix& operator*=(T1 val);

Matrix& operator/=(T1 val);

Matrix();
explicit Matrix(std::size_t size);
Matrix(std::size_t N, std::size_t M);
Matrix(const std::vector<std::vector<T1>> &data);

template<typename T2, typename E2>
Matrix(const MatrixExpression<T2, E2> &expression);

// output function
```

```
    template<typename T>
    friend std::istream& operator>>(std::istream &istream, Matrix<T1>
↪   &matrix);
};

#include "matrix.tpp"

#endif //MATRIX_CALCULATOR_MATRIX_H

// файл matrix/matrix.tpp

// Slice implementation //

Slice::Slice() : start(0), end(0) {}
Slice::Slice(std::size_t start_, std::size_t end_) :
start(start_), end(end_) {}

// AbstractSubmatrix implementation //

template<typename Cont, typename E>
AbstractSubmatrix<Cont, E>::value_type AbstractSubmatrix<Cont,
↪   E>::operator[](std::size_t i, std::size_t j) const {
    if (i >= n() || j >= m()) {
        throw std::out_of_range("attempt to access outside of the bounds");
    }
    return data[i + n_start()][j + m_start()];
}
```

```
template<typename Cont, typename E>
AbstractSubmatrix<Cont, E> AbstractSubmatrix<Cont, E>::operator=(const
↪   AbstractSubmatrix<Cont, E> &other) {
    return (static_cast<E&>(*this) = other);
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::n() const {
    return n_slice.end - n_slice.start;
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::m() const {
    return m_slice.end - m_slice.start;
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::n_start() const {
    return n_slice.start;
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::n_end() const {
    return n_slice.end;
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::m_start() const {
```

```
    return m_slice.start;
}

template<typename Cont, typename E>
std::size_t AbstractSubmatrix<Cont, E>::m_end() const {
    return m_slice.end;
}

template<typename Cont, typename E>
AbstractSubmatrix<Cont, E>::AbstractSubmatrix(Cont &data_) :
↪   AbstractSubmatrix(data_, Slice(0, data_.size())) {}

template<typename Cont, typename E>
AbstractSubmatrix<Cont, E>::AbstractSubmatrix(Cont &data_, Slice n_slice_) :
data(data_), n_slice(n_slice_) {
    if (data.size() < n_end()) {
        throw std::invalid_argument("row slice out of bounds of matri");
    }
    else if (data.empty()) {
        n_slice = Slice();
        m_slice = Slice();
    }
    else {
        m_slice = Slice(0, data[0].size());
    }
}

template<typename Cont, typename E>
```

```
AbstractSubmatrix<Cont, E>::AbstractSubmatrix(Cont &data_, Slice n_slice_,
↪  Slice m_slice_) :
data(data_), n_slice(n_slice_), m_slice(m_slice_) {
    if (data.size() < n_end() || (!data.empty() && data[0].size() <
↪  m_end())) {
        throw std::invalid_argument("bounds of submatrix inappropriate for
↪  this data");
    }
}

// ConstSubmatrix constructors deduction guide //

template<typename Cont>
ConstSubmatrix(Cont &data) -> ConstSubmatrix<typename
↪  Cont::value_type::value_type>;

template<typename Cont>
ConstSubmatrix(Cont &data, Slice n_slice) -> ConstSubmatrix<typename
↪  Cont::value_type::value_type>;

template<typename Cont>
ConstSubmatrix(Cont &data, Slice n_slice, Slice m_slice) ->
↪  ConstSubmatrix<typename Cont::value_type::value_type>;

// Submatrix implementation //

template<typename T1>
T1& Submatrix<T1>::operator[](std::size_t i, std::size_t j) {
```

```
    if (i >= this->n() || j >= this->m()) {
        throw std::out_of_range("attempt to access outside of the bounds");
    }
    return this->data[i + this->n_start()][j + this->m_start()];
}


template<typename T1>
template<typename T2, typename E2>
Submatrix<T1> Submatrix<T1>::operator=(const MatrixExpression<T2, E2>
↪   &other) {
    check_n(*this, other);
    check_m(*this, other);

    std::vector<std::vector<T1>> result =
↪   std::vector<std::vector<T1>>(this->n(), std::vector<T1>(this->m()));
    for (int i = 0; i < this->n(); ++i) {
        for (int j = 0; j < this->m(); ++j) {
            result[i][j] = other[i,j];
        }
    }

    for (int i = 0; i < this->n(); ++i) {
        for (int j = 0; j < this->m(); ++j) {
            (*this)[i,j] = result[i][j];
        }
    }
    return *this;
}
```

```
template<typename T1>
template<typename T2, typename E2>
Submatrix<T1> Submatrix<T1>::operator+=(const MatrixExpression<T2, E2>
↪ &other) {
    *this = Summation<T1, AbstractSubmatrix<std::vector<std::vector<T1>>,
↪ Submatrix<T1>>, E2>(*this, other);
    return *this;
}

template<typename T1>
template<typename T2, typename E2>
Submatrix<T1> Submatrix<T1>::operator-=(const MatrixExpression<T2, E2>
↪ &other) {
    *this = Subtraction<T1, AbstractSubmatrix<std::vector<std::vector<T1>>,
↪ Submatrix<T1>>, E2>(*this, other);
    return *this;
}

template<typename T1>
Submatrix<T1> Submatrix<T1>::operator*=(T1 val) {
    *this = *this * val;
    return *this;
}

template<typename T1>
Submatrix<T1> Submatrix<T1>::operator/=(T1 val) {
    *this = *this / val;
```

```cpp
    return *this;
}

// Submatrix constructions deduction guide //

template<typename Cont>
Submatrix(Cont &data) -> Submatrix<typename Cont::value_type::value_type>;

template<typename Cont>
Submatrix(Cont &data, Slice n_slice) -> Submatrix<typename Cont::value_type::value_type>;

template<typename Cont>
Submatrix(Cont &data, Slice n_slice, Slice m_slice) -> Submatrix<typename Cont::value_type::value_type>;

// Matrix implementation //

template<typename T1>
T1 Matrix<T1>::operator[](std::size_t i, std::size_t j) const {
    return data[i][j];
}

template<typename T1>
T1& Matrix<T1>::operator[](std::size_t i, std::size_t j) {
    return data[i][j];
}
```

```
template<typename T1>
ConstSubmatrix<T1> Matrix<T1>::operator[](std::size_t i) const {
    return (*this)[Slice(i, i+1), Slice(0, m())];
}

template<typename T1>
Submatrix<T1> Matrix<T1>::operator[](std::size_t i) {
    return (*this)[Slice(i, i+1), Slice(0, m())];
}

template<typename T1>
ConstSubmatrix<T1> Matrix<T1>::operator[](Slice n_slice) const {
    return (*this)[n_slice, Slice(0, m())];
}

template<typename T1>
Submatrix<T1> Matrix<T1>::operator[](Slice n_slice) {
    return (*this)[n_slice, Slice(0, m())];
}

template<typename T1>
ConstSubmatrix<T1> Matrix<T1>::operator[](Slice n_slice, std::size_t m)
↪   const {
    return (*this)[n_slice, Slice(m, m+1)];
}

template<typename T1>
Submatrix<T1> Matrix<T1>::operator[](Slice n_slice, std::size_t m) {
```

```
    return (*this)[n_slice, Slice(m, m+1)];
}

template<typename T1>
ConstSubmatrix<T1> Matrix<T1>::operator[](std::size_t n, Slice m_slice)
↪   const {
    return (*this)[Slice(n, n+1), m_slice];
}

template<typename T1>
Submatrix<T1> Matrix<T1>::operator[](std::size_t n, Slice m_slice) {
    return (*this)[Slice(n, n+1), m_slice];
}

template<typename T1>
ConstSubmatrix<T1> Matrix<T1>::operator[](Slice n_slice, Slice m_slice)
↪   const {
    return ConstSubmatrix<T1>(data, n_slice, m_slice);
}

template<typename T1>
Submatrix<T1> Matrix<T1>::operator[](Slice n_slice, Slice m_slice) {
    return Submatrix<T1>(data, n_slice, m_slice);
}

template<typename T1>
std::size_t Matrix<T1>::n() const {
    return N;
```

```
}

template<typename T1>
std::size_t Matrix<T1>::m() const {
    return M;
}

template<typename T1>
template<typename T2, typename E2>
Matrix<T1>& Matrix<T1>::operator=(const MatrixExpression<T2, E2> &other) {
    *this = Matrix<T1>(other);
    return *this;
}

template<typename T1>
template<typename T2, typename E2>
Matrix<T1>& Matrix<T1>::operator+=(const MatrixExpression<T2, E2> &other) {
    *this = Summation<T1, Matrix<T1>, E2>(*this, other);
    return *this;
}

template<typename T1>
template<typename T2, typename E2>
Matrix<T1>& Matrix<T1>::operator-=(const MatrixExpression<T2, E2> &other) {
    *this = Subtraction<T1, Matrix<T1>, E2>(*this, other);
    return *this;
}
```

```cpp
template<typename T1>
Matrix<T1>& Matrix<T1>::operator*=(T1 val) {
    *this = *this * val;
    return *this;
}

template<typename T1>
Matrix<T1>& Matrix<T1>::operator/=(T1 val) {
    *this = *this / val;
    return *this;
}

template<typename T1>
Matrix<T1>::Matrix() : Matrix(0, 0) {}

template<typename T1>
Matrix<T1>::Matrix(std::size_t size) : Matrix(size, size) {}

template<typename T1>
Matrix<T1>::Matrix(std::size_t N_, std::size_t M_)
: N(N_), M(M_), data(std::vector<std::vector<T1>>(n(), std::vector<T1>(m()))) {}

template<typename T1>
Matrix<T1>::Matrix(const std::vector<std::vector<T1>> &data_) : N(data.size()), data(data_) {
    M = (data.empty() ? 0 : data[0].size());
}
```

```
template<typename T1>
template<typename T2, typename E2>
Matrix<T1>::Matrix(const MatrixExpression<T2, E2> &expression) :
    Matrix(expression.n(), expression.m()) {
    (*this)[Slice(0, n()), Slice(0, m())] = expression;
}

template<typename Cont, typename E>
Matrix(AbstractSubmatrix<Cont, E>) -> Matrix<typename
    AbstractSubmatrix<Cont, E>::value_type>;

// input function //

template<typename T>
std::istream& operator>>(std::istream &istream, Matrix<T> &matrix) {
    std::size_t n, m;
    istream >> n >> m;

    matrix = Matrix<T>(n, m);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            istream >> matrix[i, j];
        }
    }
    return istream;
}
```

```
// файл eigenpairs-finder/eigenpairs-finder.h

#ifndef MATRIX_CALCULATOR_EIGENPAIRS_FINDER_H
#define MATRIX_CALCULATOR_EIGENPAIRS_FINDER_H

#include <complex>
#include "../matrix/matrix.h"

// class of conjugation operation
// T - type of elements of matrix obtained by evaluating conjugation of the
↪  expression
// E - type of expression for conjugation
template<typename T, typename E>
class Conjugation : public MatrixExpression<T, Conjugation<T, E>> {
private:
    std::conditional_t<E::has_data, const E&, E> expression;

public:
    // returns copy of element on i-th row and j-th column of matrix
↪   obtained by evaluating conjugation of the expression
    T operator[](std::size_t i, std::size_t j) const;

    // return size of matrix obtained by evaluating conjugation of the
↪   expression
    std::size_t n() const;
    std::size_t m() const;
```

```
    explicit Conjugation(const MatrixExpression<T, E> &expression);
};

// conjugation functions //

double conj(double val);

template<typename T, typename E>
Conjugation<T, E> conj(const MatrixExpression<T, E> & expression);

// 2-norm of vector
template<typename T, typename E>
double norm(const MatrixExpression<T, E> &vector);

// returns identity matrix of size "size"
template<typename T>
Matrix<T> identity(std::size_t size);

// Hessenberg decomposition //

// return householder vector which used in householder projection
template<typename E>
Matrix<double> householder_vector(const MatrixExpression<double, E>
 ↪  &expression);

template<typename T, typename E>
Matrix<std::complex<T>> householder_vector(const
 ↪  MatrixExpression<std::complex<T>, E> &expression);
```

```
// functions which apply householder reflector with householder vector "v"
↪ to "matrix" //

// apply householder projection on left of "submatrix"
template<typename T1, typename T2, typename E2>
void left_h_transformation(Submatrix<T1> submatrix, const
↪ MatrixExpression<T2, E2> &v);

// apply householder projection on right of "submatrix"
template<typename T1, typename T2, typename E2>
void right_h_transformation(Submatrix<T1> submatrix, const
↪ MatrixExpression<T2, E2> &v);

// decomposes matrix A. A=QHQ* where Q - unitary matrix, H - upper
↪ Hessenberg matrix.
// H overwrites matrix, Q returned;
template<typename T>
Matrix<T> hessenberg(Matrix<T> &matrix);

// returns 2x2 givens matrix which transform vector (x, y)* to (0, z)
template<typename T>
Matrix<T> givens(T x, T y);

// performs QR step of hessenberg matrix by overriding matrix, returns Q
↪ matrix
template<typename T>
Matrix<T> QR_step(Matrix<T> &matrix);
```

```
// decomposes upper heisenberg matrix to QTQ* where T is upper triangular
↪  and Q is unitary.
// Overwrites matrix with T and returns Q
template<typename T>
Matrix<std::complex<T>> complex_schur(Matrix<std::complex<T>> &matrix);

// returns eigenvector associated with eigenvalue on value_index'th diagonal
↪  element of triangular matrix
template<typename T>
Matrix<std::complex<T>> schur_eigenvector(Matrix<std::complex<T>> matrix,
↪  std::size_t value_index);

// returns vector of eigenpairs of matrix
template<typename T>
std::vector<std::pair<std::complex<T>, Matrix<std::complex<T>>>>
↪  eigenpairs(Matrix<std::complex<T>> matrix);

#include "eigenpairs-finder.tpp"

#endif //MATRIX_CALCULATOR_EIGENPAIRS_FINDER_H

// файл eigenpairs-finder/eigenpairs-finder.tpp

#include <cmath>
#include <complex>
#include "../matrix/matrix.h"
```

```
// all numbers less than ZERO are considered 0
const double ZERO = 1e-15;

// Conjugation implementation //

template<typename T, typename E>
T Conjugation<T, E>::operator[](std::size_t i, std::size_t j) const {
    return conj(expression[j,i]);
}

template<typename T, typename E>
std::size_t Conjugation<T, E>::n() const {
    return expression.m();
}

template<typename T, typename E>
std::size_t Conjugation<T, E>::m() const {
    return expression.n();
}

template<typename T, typename E>
Conjugation<T, E>::Conjugation(const MatrixExpression<T, E> &expression_) :
expression(static_cast<const E&>(expression_)) {}

// conjugation functions implementation //

double conj(double val) { return val; }
```

```cpp
template<typename T, typename E>
Conjugation<T, E> conj(const MatrixExpression<T, E> & expression) {
    return Conjugation<T, E>(expression);
}

template<typename T, typename E>
double norm(const MatrixExpression<T, E>& vector) {
    if (vector.m() != 1) {
        throw std::invalid_argument("2-norm can be calculated for vectors only");
    }

    double square_sum = 0;
    for (int i = 0; i < vector.n(); ++i) {
        square_sum += std::abs<double>(vector[i,0]) * std::abs<double>(vector[i,0]);
    }
    return std::sqrt(square_sum);
}

template<typename T>
Matrix<T> identity(std::size_t size) {
    Matrix<T> result = Matrix<T>(size);
    for (int i = 0; i < size; ++i) {
        result[i,i] = 1;
    }
    return result;
}
```

```
// Hessenberg decomposition implementation //

template<typename E>
Matrix<double> householder_vector(const MatrixExpression<double, E> &expression) {
    if (expression.m() != 1) {
        throw std::invalid_argument("householder vector can be obtained only from another vector");
    }

    // both signs are appropriate, choose which one makes the longest hessenberg vector to improve stability
    int sign = (expression[0,0] >= 0 ? 1 : -1);
    Matrix<double> vector = expression;
    vector[0,0] += sign * norm(vector);
    return vector / norm(vector);
}

template<typename T, typename E>
Matrix<std::complex<T>> householder_vector(const MatrixExpression<std::complex<T>, E> &expression) {
    if (expression.m() != 1) {
        throw std::invalid_argument("householder vector can be obtained only from another vector");
    }

    Matrix<std::complex<T>> vector = expression;
```

```
    vector[0,0] += (vector[0,0] * norm(vector)) / std::abs(vector[0, 0]);
    return vector / std::complex<T>(norm(vector));
}

template<typename T1, typename T2, typename E2>
void left_h_transformation(Submatrix<T1> submatrix, const
↪   MatrixExpression<T2, E2> &v) {
    submatrix -= T2(2) * v * (conj(v) * submatrix);
}

template<typename T1, typename T2, typename E2>
void right_h_transformation(Submatrix<T1> submatrix, const
↪   MatrixExpression<T2, E2> &v) {
    submatrix -= T2(2) * (submatrix * v) * conj(v);
}

template<typename T>
Matrix<T> hessenberg(Matrix<T> &matrix) {
    if (matrix.n() != matrix.m()) {
        throw std::invalid_argument("only square matrices have Hessenberg
↪   decomposition");
    }

    auto Q = identity<T>(matrix.n());

    for (int k = 0; k < matrix.n()-2; ++k) {
        // creates householder vector
        auto x = matrix[Slice(k+1, matrix.n()), k];
```

```
        auto v = householder_vector(x);

        // apply householder transformation to both sides of the matrix in order to reach similarity
        left_h_transformation(matrix[Slice(k+1, matrix.n()), Slice(k, matrix.n())], v);

        right_h_transformation(matrix[Slice(0, matrix.n()), Slice(k+1, matrix.n())], v);

        auto Q_submatrix = Q[Slice(1, matrix.n()), Slice(k+1, matrix.n())];

        // updates matrix Q of hessenberg decomposition
        if (k == 0) { Q_submatrix -= T(2) * v * conj(v); }
        else { right_h_transformation(Q_submatrix, v); }
    }
    return Q;
}

template<typename T>
Matrix<T> givens(T x, T y) {
    auto result = Matrix<T>(2);

    double abs_x = std::abs(x);
    double abs_y = std::abs(y);

    result[0,0] = x / std::sqrt(abs_x * abs_x + abs_y * abs_y);
    result[0,1] = conj(-y) / std::sqrt(abs_x * abs_x + abs_y * abs_y);
```

```
    result[1,0] = conj(-result[0,1]);
    result[1,1] = conj(result[0,0]);
    return result;
}

template<typename T>
Matrix<T> QR_step(Matrix<T> &matrix) {
    if (matrix.n() != matrix.m()) {
        throw std::invalid_argument("QR step can be applied only to square matrix");
    }

    auto Q = identity<T>(matrix.n());

    // performs QR decomposition via multiplying by givens matrix on left
    auto givenses = std::vector<Matrix<T>>(matrix.n()-1);
    for (int k = 0; k < matrix.n()-1; ++k) {
        if (std::abs(matrix[k+1, k]) > ZERO) {
            givenses[k] = givens(matrix[k, k], matrix[k + 1, k]);
            auto submatrix = matrix[Slice(k, k+2), Slice(k, matrix.n())];
            submatrix = conj(givenses[k]) * submatrix;

            // updates matrix Q of decomposition
            auto Q_submatrix = Q[Slice(0, k+2), Slice(k, k + 2)];
            Q_submatrix = Q_submatrix * givenses[k];
        }
    }
```

```
    // multiplying the matrix by Q on right
    for (int k = 0; k < matrix.n()-1; ++k) {
        if (givenses[k].n() != 0) {
            auto submatrix = matrix[Slice(0,k+2), Slice(k,k+2)];
            submatrix = submatrix * givenses[k];
        }
    }
    return Q;
}

template<typename T>
Matrix<std::complex<T>> complex_schur(Matrix<std::complex<T>> &matrix) {
    auto Q = identity<std::complex<T>>(matrix.n());
    auto id = Q;
    for (int n = matrix.n(); n >= 2; --n) {
        do {
            auto tr = matrix[n-2,n-2] + matrix[n-1,n-1];
            auto det = matrix[n-2,n-2] * matrix[n-1,n-1] - matrix[n-2,n-1] *
    matrix[n-1,n-2];

            // computes Wilkinson shift
            std::complex<T> shift = (tr + std::sqrt(tr * tr -
    std::complex<T>(4) * det)) / std::complex<T>(2);

            // performs QR step with shift on the matrix
            matrix -= shift * id;
            Q = Q * QR_step(matrix);
            matrix += shift * id;
```

```
        } while (std::abs(matrix[n-1, n-2]) > ZERO);
        // after making element on the left of n-th diagonal element
↪   sufficiently small
        // deflates matrix and chooses different Wilkinson shift
    }
    return Q;
}

template<typename T>
Matrix<std::complex<T>> schur_eigenvector(Matrix<std::complex<T>> matrix,
↪   std::size_t value_index) {
    matrix -= matrix[value_index, value_index] *
↪   identity<std::complex<T>>(matrix.n());
    Matrix<std::complex<T>> result = Matrix<T>(matrix.n(), 1);
    result[value_index,0] = 1;

    // solves triangular system which finds eigenvalue of the desired value
    for (std::size_t i = value_index - 1; i < matrix.n(); --i) {
        for (std::size_t j = i+1; j < matrix.n(); ++j) {
            result[i,0] -= matrix[i, j] * result[j, 0];
        }
        result[i,0] /= matrix[i, i];
    }
    return result / std::complex<T>(norm(result));
}

template<typename T>
```

```cpp
std::vector<std::pair<std::complex<T>, Matrix<std::complex<T>>>>
    eigenpairs(Matrix<std::complex<T>> matrix) {
    if (matrix.n() != matrix.m()) {
        throw std::invalid_argument("only square matrix is allowed");
    }

    auto Q1 = hessenberg(matrix);
    auto Q2 = complex_schur(matrix);

    auto id = identity<std::complex<T>>(matrix.n());

    auto result = std::vector<std::pair<std::complex<T>,
    Matrix<std::complex<T>>>>(matrix.n());
    for (int i = 0; i < matrix.n(); ++i) {
        auto eigenvalue = matrix[i, i];
        Matrix<std::complex<T>> eigenvector = Q1 * (Q2 *
    schur_eigenvector(matrix, i));
        result[i] = (std::make_pair(eigenvalue, eigenvector));
    }

    return result;
}

// extra function. Isn't required to find eigenpairs but cost me a lot of
    time to implement :(
// decomposes real matrix to QTQ* where T is block-upper triangular and Q is
    unitary
template<typename T>
```

```
Matrix<T> real_schur(Matrix<T> &matrix) {
    Matrix<T> Q = identity<T>(matrix.n());

    std::size_t p = matrix.n();
    while (p > 2) {
        int count = 1;

        T s = matrix[p-2,p-2] + matrix[p-1,p-1];
        T t = matrix[p-2,p-2] * matrix[p-1,p-1] - matrix[p-2,p-1] * matrix[p-1,p-2];

        Matrix<T> column = Matrix<T>(3,1);
        column[0,0] = matrix[0,0] * matrix[0,0] + matrix[0,1] * matrix[1,0] - s * matrix[0,0] + t;
        column[1,0] = matrix[1,0] * (matrix[0,0] + matrix[1,1] - s);
        column[2,0] = matrix[1,0] * matrix[2,1];

        auto v = householder_vector(column);
        left_h_transformation(matrix[Slice(0,3), Slice(0, matrix.n())],v);
        std::size_t r = (p > 3 ? 4 : 3);
        right_h_transformation(matrix[Slice(0,r), Slice(0,3)], v);

        right_h_transformation(Q[Slice(0,matrix.n()), Slice(0,3)], v);

        for (std::size_t k = 0; k < p-3; ++k) {
            ++count;

            column[0,0] = matrix[k+1,k];
```

```
            column[1,0] = matrix[k+2,k];
            column[2,0] = matrix[k+3,k];

            v = householder_vector(column);
            left_h_transformation(matrix[Slice(k+1,k+4), Slice(k, matrix.n())], v);
            r = (k+5 < p ? k+5 : p);
            right_h_transformation(matrix[Slice(0,r), Slice(k+1,k+4)], v);

            right_h_transformation(Q[Slice(0,matrix.n()), Slice(k+1,k+4)], v);
        }

        ++count;

        column = Matrix<T>(2,1);
        column[0,0] = matrix[p-2,p-3];
        column[1,0] = matrix[p-1,p-3];

        v = householder_vector(column);
        left_h_transformation(matrix[Slice(p-2,p), Slice(p-3,matrix.n())], v);
        right_h_transformation(matrix[Slice(0, matrix.n()), Slice(p-2,p)], v);

        right_h_transformation(Q[Slice(0, matrix.n()), Slice(p-2,p)], v);

        if (std::abs(matrix[p-1,p-2]) < ZERO) { p -= 1; }
```

```
        else if (std::abs(matrix[p-2,p-3]) < ZERO) { p -= 2; }
    }
    return Q;
}

// файл surface-operations/operations.h

#ifndef MATRIX_CALCULATOR_OPERATIONS_H
#define MATRIX_CALCULATOR_OPERATIONS_H

#include<cmath>
#include "../matrix/matrix.h"

// returns matrix which rotates vector of size 2 by an "angle"
template<typename T>
Matrix<T> rotation_matrix(T angle);

// returns matrix which projects vector on the line directed by "direction"
↪  vector
template<typename T>
Matrix<T> projection_matrix(Matrix<T> direction);

// returns matrix which reflects vector along the line directed by
↪  "direction" vector
template<typename T>
Matrix<T> reflection_matrix(Matrix<T> direction);

#include"operations.tpp"
```

```
#endif //MATRIX_CALCULATOR_OPERATIONS_H

// файл surface-operations/operations.tpp

#include <cmath>
#include "../matrix/matrix.h"
#include "../eigenpairs-finder/eigenpairs-finder.h"

template<typename T>
Matrix<T> rotation_matrix(T angle) {
    auto result = Matrix<T>(2);
    result[0,0] = std::cos(angle);
    result[0,1] = -std::sin(angle);
    result[1,0] = std::sin(angle);
    result[1,1] = std::cos(angle);
    return result;
}

template<typename T>
Matrix<T> projection_matrix(Matrix<T> direction) {
    direction /= T(norm(direction));
    return direction * conj(direction);
}

template<typename T>
Matrix<T> reflection_matrix(Matrix<T> direction) {
    direction = direction / T(norm(direction));
```

```
    auto orthogonal = Matrix<T>(2,1);
    orthogonal[0,0] = -direction[1,0];
    orthogonal[1,0] = direction[0,0];

    return direction * conj(direction) - orthogonal * conj(orthogonal);
}
```