Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ Фундаментальные науки

КАФЕДРА Прикладная математика

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ДОМАШНЕЙ РАБОТЕ

ПО КУРСУ:

"Методы численного решения задача линейной алгебры"

Студент

ФН2 – 31М
(Группа)

(Подпись, дата) Д. И. Богданов (И. О. Фамилия)

(Подпись, дата) А. С. Родин (И. О. Фамилия)

2024 г.

Содержание

1. Постановка задачи

Нужно сформировать матрицу размером 10×10 по следующему принципу. В качестве базовой матрицы , берется известная матрица, которая получается после дискретизации одномерного оператора Лапласа методом конечных разностей или методом конечных элементов. На равномерной сетке:

$$A_0 = \{a_{ij}\}_{i,j=\overline{1,n}}$$

где

$$a_{ij} = \begin{cases} 2, & i = j, \\ -1, & |i - j| = 1, \\ 0, & \text{else.} \end{cases}$$

Для данной матрицы известны аналитические формулы для собственных значений $(n = 10)$

$$\lambda_j^0 = 2(1 - \cos\frac{\pi j}{n+1}), \quad j = \overline{1, n}.$$

и компонент собственных векторов (вектора имеют 2-норму равную 1):

$$z_j^0(k) = \sqrt{\frac{2}{n+1}} \sin\frac{\pi j k}{n+1}, \quad k = \overline{1, n}.$$

Итоговая матрица получается по формулам:

$$A = A_0 + \delta A,$$

$$\delta A_{ij} = \begin{cases} \dfrac{c}{i+1}, & i \neq j, \\ 0, & i = j, \end{cases}$$

$$c = \frac{N_{var}}{N_{var} + 1}\varepsilon.$$

где N_{var} — номер варианта (совпадает с номером студента в списке в журнале группы), ε - параметр, значение которого задается далее.

Нужно выполнить следующие задания:

1.1. Задание 1

Взять матрицу для значения $\varepsilon = 0.1$, убрать последний столбец и сформировать из первых 9 столбцов матрицу $\hat{A}$ размера 10×9. Решить линейную задачу наименьших квадратов для вектора невязки

$$r = \hat{A}x - b,$$

где вектор b размерности 10×1 нужно получить по следующему алгоритму: выбрать вектор x_0, размерности 9×1 и для него вычислить $b = \hat{}_0$.

Для решения поставленной задачи использовать QR разложение: для вариантов с четным номером использовать соответствующий алгоритм, основанный на методе вращений Гивенса, для вариантов с нечетным номером - алгоритм, основанный на методе отражений Хаусхолдера.

После получения решения сделать оценку величины $||x - x_0||_2/||x_0||_2$.

1.2. Задание 2

Для матрицы найти все ее собственные значения ($\lambda_j, j = \overline{1, 10}$) и собственные вектора ($z_j, j = \overline{1, 10}$, с 2-нормой равной 1) с помощью неявного QR-алгоритма со сдвигом (с предварительным приведением матрицы к форме Хессенберга) для трех вариантов: $\varepsilon = 10^{-1}, 10^{-3}, 10^{-6}$.

По итогам расчетов нужно сделать сводную таблицу, в которой указать следующие величины: $\lambda_j - \lambda_j^0$ и $||z_j - z_j^0||_2$ для $j = \overline{1, 10}$.

2. Решение линейной задачи наименьших квадратов в с помощью QR-разложения методом отражений Хаусхолдера

В рамках данной задачи реализованы следующие алгоритмы:

- Отражение Хаусхолдера;
- Обычное QR-разложение;
- QR-разложение для метода LLS;
- Обратный ход метода Гаусса;
- Linear Least Squares с помощью QR-разложения.

Расчетная реализация всех алгоритмов выполнена на языке `C++` с использованием библиотеки `Eigen` для базовых матричных операций. Для отладки программы и проверки корректности результатов реализован вспомогательный скрипт на `Wolfram Mathematica`, использующий встроенные методы для получения необходимых разложений.

Изначальная форма алгоритма QR-разложения имеет следующий вид:

```
-----------------------------------------------------------------------
-   Q_wave = I;                                                       -
-   R_wave = A;                                                       -
-   for i = 1,min(M-1,N) {                                            -
-      ui               = House(R_wave[i:M, i])      // O(N)          -
-      pi_wave          = I - 2 ui ui^T              // O(N^2)        -
-      pi               = I                          //               -
-      pi[i:M, i:M]     = pi_wave                    //               -
-      R_wave[i:M, i:N] = pi_wave * R_wave[i:M, i:N] // O(N^3)        -
-      Q_wave[0:M, i:M] = Q_wave[0:M, i:M] * pi_wave // O(N^3)        -
-   }                                                                 -
-   return { Q[0:M, 0:N], R[0:N, 0:N] }                               -
-----------------------------------------------------------------------
```

В явном виде алгоритм имеет сложность $O(N^4)$, это решается если подставить матрицу p_i явно и расписать матричное умножение как 2 умножения матрицы на вектор. Приходим к следующему алгоритму:

```
----------------------------------------------------------------------
-   Q_wave = I;                                                      -
-   R_wave = A;                                                      -
-   for i = 1,min(M-1,N) {                                           -
-      ui                = House(R_wave[i:M, i])          // O(N)    -
-      R_wave[i:M, i:N] -= 2 ui (ui^T * R_wave[i:M, i:N]) // O(N^2)  -
-      Q_wave[0:M, i:M]  -= Q_wave[0:M, i:M] * 2 ui ui^T  // O(N^2)  -
-   }                                                                -
-   return { Q[0:M, 0:N], R[0:N, 0:N] }                              -
----------------------------------------------------------------------
```

Произведено тестовое QR-разложение матрицы $\hat{A}$, проверена ортогональность матрицы Q и приблизительное совпадение $QR \approx A$. Результаты сходятся с разложением с помощью встроенного метода `QRDecomposition[ ]` в пакете `Wolfram Mathematica`.

Модификация QR-разложени для метода LLS также позволяет вычислять правую часть $Q^T b$ сразу по ходу разложения с целью экономии вычислительных ресурсов. Алгоритм описан в листинге программы

В качестве тестового вектора выбран $x_0 : x_i = i^2$. Для него решена задача наименьших квадратов относительно невязки, получен вектор x_{LLS} имеющий следующую норму ошибки:

```
lls_error_estimate -> 3.990082657206211e-16
```

Результаты LLS также сверены с помощью скрипта.

3. Получение собственных значений матрицы с помощью QR-алгоритма со сдвигами

В рамках данной задачи реализованы следующие алгоритмы:

- Приведение матрицы к Хессенберговой форма;
- QR-алгоритм без сдвигов (для отладочных целей);
- $O(N^2)$ QR-разложение для верхне-хессенберговых матриц с вычислением RQ;
- QR-алгоритм со сдвигами и приведением изначальной матрицы к Хессенберговой форме;

В качестве значений сдвига в QR-алгоритме берется H_{nn}, итерации производятся до тех пор пока значение $|H_{n,n-1}|$ не станет меньше некоторого ε, после чего n-е значение на диагонали считаем найденым и редуцируем задачу к работе с блоком $(n-1) \times (n-1)$. Процедура повторяется до достижения максимального числа итераций или редукции n к 2-м (при нормальном ходе программы ожидается второй исход). Приходим к следующему алгоритму:

```
---------------------------------------------------------------------------
-   while (N >= 2 && iteration++ < max_iterations) {                      -
-      sigma = T_shur[N, N]                                   // O(1)     -
-      [ Q, R, RQ ] = qr_factorize_hessenberg(T_shur[1:N, 1:N]) // O(N^2) -
-      T_shur[0:N, 0:N] = RQ + sigma I                        // O(N^2)   -
-      if (|T_shur[N, N-1]| < eps) --N                        // O(1)     -
-   }                                                                     -
---------------------------------------------------------------------------
```

Таким образом, метод редуцируется к сложности $O(N^3)$, без учета хессенберговой структуры имели бы $O(N^4)$. Матрицы Q, R в данной реализации не нужны в явном виде, однако программно также определяются для отладочных целей.

В результате работы алгоритма получаем матрицу T из разложения Шура, значения на главной диагонали соответствуют собственным значениям изначальной матрицы.

Получены собственные значения матрицы A, результаты сравнения с аналитическими результатами приведены ниже для $\varepsilon = 10^{-1}, 10^{-3}, 10^{-6}$:

j	$\|\lambda_j^0 - \lambda_j\|$	$\|\|z_j^0 - z_j\|\|_2$
1	0.0383792	x
2	0.00164998	x
3	0.00198987	x
4	0.00445514	x
5	0.00474202	x
6	0.00674079	x
7	0.00682261	x
8	0.00718661	x
9	0.00664725	x
10	0.00542461	x

Таблица 1. Ошибки при $\varepsilon = 10^{-1}$

j	$\|\lambda_j^0 - \lambda_j\|$	$\|\|z_j^0 - z_j\|\|_2$
1	0.000395515	x
2	1.15997e-05	x
3	1.46055e-05	x
4	4.50412e-05	x
5	4.80620e-05	x
6	6.73953e-05	x
7	6.82741e-05	x
8	7.18550e-05	x
9	6.65622e-05	x
10	5.45308e-05	x

Таблица 2. Ошибки при $\varepsilon = 10^{-1}$

j	$\|\lambda_j^0 - \lambda_j\|$	$\|\|z_j^0 - z_j\|\|_2$
1	3.95623e-07	x
2	1.15569e-08	x
3	1.45556e-08	x
4	4.50458e-08	x
5	4.80682e-08	x
6	6.73952e-08	x
7	6.82745e-08	x
8	7.18549e-08	x
9	6.65631e-08	x
10	5.45337e-08	x

Таблица 3. Ошибки при $\varepsilon = 10^{-1}$

Пример сводки результатов расчета для малого n (в силу вербозности результата при $n = 10$) приведен в приложении.

4. Листинг расчетной программы на C++

source/utils.hpp

```cpp
#pragma once

#include "firstparty/proto_utils.hpp"
#include "thirdparty/Eigen/Dense"
#include "thirdparty/Eigen/src/Core/Matrix.h"
#include "thirdparty/Eigen/src/Core/util/Meta.h"
#include <limits>



using Matrix    = Eigen::MatrixXd;
using Vector    = Eigen::VectorXd;
using RowVector = Eigen::RowVectorXd;
using Idx       = Eigen::Index; // Eigen uses signed (!) indeces

constexpr bool collapse_small_values = false;

// Eigen has formatting options built-in, but I prefer the style of my own package.
// Eigen stores matrices as col-major so we do a matrix view into the CR memory layout.
inline std::string stringify_matrix(const Matrix& eigen_matrix) {
    using namespace utl;

    if constexpr (collapse_small_values) {
        utl::mvl::Matrix<double> mvl_matrix(
            eigen_matrix.rows(), eigen_matrix.cols(),
            [&](std::size_t i, std::size_t j) { return (std::abs(eigen_matrix(i, j)) < 1e-12) ? 0. : eigen_matrix(i, j); });
        return utl::mvl::format::as_matrix(mvl_matrix);
    } else {
        mvl::ConstMatrixView<double, mvl::Checking::BOUNDS, mvl::Layout::CR> view(
            eigen_matrix.rows(), eigen_matrix.cols(), eigen_matrix.data());
        return mvl::format::as_matrix(view);
    }
}
```

source/qr_factorization.hpp

```
#pragma once

#include "utils.hpp"



// Householder reflection operation. O(N) complexity.
//
// Template so we can take vectors/blocks/views as an argument and not force a
copy.
//
template <class VectorType>
Vector householder_reflect(const VectorType& x) {

    // u = { x[0] + sign(x[0]) * ||x||_2, x[1], x[2], ... , x[K] }
    Vector u = x;
    u(0) += utl::math::sign(u(0)) * u.norm();

    return u.normalized();
}

// QR factorization. O(N^3) complexity.
//
// Original alg would be:
// ---------------------------------------------------------------------------
// -   Q_wave = I;                                                           -
// -   R_wave = A;                                                           -
// -   for i = 1,min(M-1,N) {                                                -
// -      ui                 = House(R_wave[i:M, i])      // O(N)            -
// -      pi_wave            = I - 2 ui ui^T              // O(N^2)          -
// -      pi                 = I                          //                 -
// -      pi[i:M, i:M]       = pi_wave                    //                 -
// -      R_wave[i:M, i:N]   = pi_wave * R_wave[i:M, i:N] // O(N^3)          -
// -      Q_wave[0:M, i:M]   = Q_wave[0:M, i:M] * pi_wave // O(N^3)          -
// -   }                                                                     -
// -   return { Q[0:M, 0:N], R[0:N, 0:N] }                                   -
// ---------------------------------------------------------------------------
//
// After the algorithm we end up with a following decomposition:
// A = p1 * ... * pN * rcat[ R 0 ]
//     ^^^^^^^^^^^^^   ^^^^^^^^^^^
//     Q_wave          R_wave
// where Q_wave and R_wave are "extended" matrices Q and R, to get proper QR we
need to trim a few rows/cols at the end
//
// This alg also results in O(N^4). We can rewrite it by substituting 'pi_wave'
directly and doing
// 2 matrix*vector products instead of 1 matrix*matix, which brings complexity
down to O(N^3).
//
// ---------------------------------------------------------------------------
// -   Q_wave = I;                                                           -
// -   R_wave = A;                                                           -
// -   for i = 1,min(M-1,N) {                                                -
// -      ui                 = House(R_wave[i:M, i])          // O(N)        -
// -      R_wave[i:M, i:N] -= 2 ui (ui^T * R_wave[i:M, i:N]) // O(N^2)      -
// -      Q_wave[0:M, i:M]  -= Q_wave[0:M, i:M] * 2 ui ui^T  // O(N^2)      -
```

```
// -   }                                                               -
// -   return { Q[0:M, 0:N], R[0:N, 0:N] }                             -
// -----------------------------------------------------------------------
//
inline std::pair<Matrix, Matrix> qr_factorize(const Matrix& A) {
    const auto M = A.rows();
    const auto N = A.cols();

    Matrix Q_wave = Matrix::Identity(M, M);
    Matrix R_wave = A;

    for (Idx i = 0; i < std::min(M - 1, N); ++i) {
        const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1)); // O(N)
        R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() * R_wave.block(i, i, M - i, N - i)); // O(N^2)
        Q_wave.block(0, i, M, M - i) -= Q_wave.block(0, i, M, M - i) * 2. * ui * ui.transpose();           // O(N^2)
    }

    return {Q_wave.block(0, 0, M, N), R_wave.block(0, 0, N, N)};
}

// A variant of QR-decomposition used for linear least squares. O(N^3) complexity.
//
// Is is more efficient since in LSQ we don't need 'Q' explicitly,
// we can directly compute 'Q^T b'.
//
inline std::pair<Matrix, Matrix> qr_factorize_lls(const Matrix& A, const Vector& b) {
    const auto M = A.rows();
    const auto N = A.cols();

    Matrix R_wave = A;
    Vector QTb    = b;

    for (Idx i = 0; i < std::min(M - 1, N); ++i) {
        // ui                = House(R_wave[i:M, i])
        // R_wave[i:M, i:N] -= 2 ui ui^T * R_wave[i:M, i:N]
        const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1)); // O(N)
        R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() * R_wave.block(i, i, M - i, N - i)); // O(N^2)

        // gamma     = - 2 ui^T QTb[i:M]
        // QTb[i:M] += gamma * ui
        const Matrix gamma = -2. * ui * QTb.segment(i, M - i).transpose(); // O(N)
        QTb.segment(i, M - i) += gamma * ui;                               // O(N^2)
    }

    return {QTb.segment(0, N), R_wave.block(0, 0, N, N)};
}

// A variant of QR-decomposition used decomposing upper-hessenberg matrices in QR-iteration. O(N^2) complexity.
//
// Returns { Q, R, RQ }. Technically we only need RQ for for the QR-algorithm, but for testing purposes { Q, R}
```

```
// are left the same.
//
// Same algorithm as regular QR factorization, except instead of blocks of 'M - i' rows/cols we
// have blocks of '2' rows/cols, which reduces O(N^2) operations to O(N).
//
inline std::tuple<Matrix, Matrix, Matrix> qr_factorize_hessenberg(const Matrix& A) {
    assert(A.rows() == A.cols());

    const auto M = A.rows();

    Matrix Q = Matrix::Identity(M, M);
    Matrix R = A;
    Matrix V = Matrix::Zero(M, M);

    // Compute { Q, R } in O(N^2)
    for (Idx i = 0; i < M - 1; ++i) {
        const Vector ui = householder_reflect(R.block(i, i, 2, 1)); // O(N)
        R.block(i, 0, 2, M) -= 2. * ui * (ui.transpose() * R.block(i, 0, 2, M)); // O(N)
        Q.block(0, i, M, 2) -= Q.block(0, i, M, 2) * 2. * ui * ui.transpose(); // O(N)
        V.block(i, i, 2, 1) = ui; // O(N)
    }

    // Compute { RQ } in O(N^2)
    Matrix RQ = R;
    for (Idx i = 0; i < M - 1; ++i) {
        Vector v = V.block(i, i, 2, 1);
        RQ.block(0, i, M, 2) -= RQ.block(0, i, M, 2) * 2. * v * v.transpose(); // O(N)
    }

    return {Q, R, RQ};
}

// Hessenberg QHQ^T-factorization using householder reflections. O(N^3) complexity.
//
// Algorithm:
// -------------------------------------------------------------
// -   H = A;                                                  -
// -   for i = 1, M - 2 {                                      -
// -      ui = House(H[i+1:M, i])                     // O(N)    -
// -      H[i+1:M, i:M] -= 2 ui (ui^T * H[i+1:M, i:M]) // O(N^2)  -
// -      H[1:M, i+1:M] -= 2 (H[1:M, i+1:M] * ui) ui^T // O(N^2)  -
// -   }                                                       -
// -------------------------------------------------------------
//
inline Matrix hessenberg_reduce(const Matrix& A) {
    assert(A.rows() == A.cols());

    const Idx M = A.rows();

    Matrix H = A;

    for (Idx i = 0; i < M - 2; ++i) {
        const Vector ui = householder_reflect(H.block(i + 1, i, M - i - 1, 1));
```

```
        H.block(i + 1, i, M - i - 1, M - i) -= 2. * ui * (ui.transpose() * H.block(i + 1, i, M - i - 1, M - i));
        H.block(0, i + 1, M, M - i - 1) -= 2. * (H.block(0, i + 1, M, M - i - 1) * ui) * ui.transpose();
    }

    return H;
}
```

source/linear_least_squares.hpp

```cpp
#pragma once

#include "qr_factorization.hpp"

// Backwards gaussian elimination. O(N^2) complexity.
//
// Assumes 'R' to be upper-triangular matrix.
//
inline Vector backwards_gaussian_elimination(const Matrix& R, Vector rhs) {
    for (Idx i = R.rows() - 1; i >= 0; --i) {
        for (Idx j = i + 1; j < R.cols(); ++j) rhs(i) -= R(i, j) * rhs(j);
        rhs(i) /= R(i, i);
    }

    return rhs;
}

// Linear Least Squares problem. O(N^3) complexity.
//
// LLS has a following solusion:
//    x = A^+ b
//    where A^+ = R^-1 * Q^T
//
// We can rewrite it as a SLAE:
//    R x = Q^t b
//
// since 'R' is upper-triangular, we only need to do the backwards gaussian elimination, which is O(N^2).
//
Vector linear_least_squares(const Matrix& A, const Matrix& b) {
    // Computing QR the usual way
    // const auto [Q, R] = qr_factorize(A);
    // const auto x      = backwards_gaussian_elimination(R, Q.transpose() * b);

    // Computing QR with (Q^T * b) directly
    const auto [QTb, R] = qr_factorize_lls(A, b);
    const auto x        = backwards_gaussian_elimination(R, QTb);

    return x;
}
```

source/eigenvalues.hpp

```
#pragma once

#include "qr_factorization.hpp"
#include "thirdparty/Eigen/src/Core/util/Constants.h"
#include "utils.hpp"
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <limits>

#include "thirdparty/Eigen/Core"

// QR-method for eigenvalues with NO shift and NO Hessenberg form optimization.
//
// Used as a reference. O(N^3) single iteration complexity.
//
Matrix eigenvalues_prototype(const Matrix& A) {
    assert(A.rows() == A.cols());

    Matrix T_shur = A;

    for (Idx i = 0; i < A.rows() * 100; ++i) {
        const auto [Q, R] = qr_factorize(T_shur); // O(N^3)
        T_shur            = R * Q;                // O(N^3)
        // no stop condition, just do a ton of iterations
    }

    return T_shur;
}

// QR-method for eigenvalues with shifts and Hessenberg form optimization.
//
// Requires 'A' to be in upper-Hessenber form (!).
// Using Hessenberg form brings complexity down to O(N^2) per iteration.
//
// Algorithm:
// ----------------------------------------------------------------------
// -   while (N >= 2 && iteration++ < max_iterations) {                 -
// -      sigma = T_shur[N, N]                                 // O(1)   -
// -      [ Q, R, RQ ] = qr_factorize_hessenberg(T_shur[1:N, 1:N]) // O(N^2) -
// -      T_shur[0:N, 0:N] = RQ + sigma I                      // O(N^2) -
// -      if (|T_shur[N, N-1]| < eps) --N                      // O(1)   -
// -   }                                                                 -
// ----------------------------------------------------------------------
//
// Note that matrix multiplication here is O(N^2) because 'R' is tridiagonal.
//
// As a stop-condition for deflating the block we use last row element under the diagonal,
// as soon as it becomes "small enough" the block can deflate.
//
// 'Q' and 'R' matrices aren't directly used anywhere, but still computed for debugging purposes.
//
Matrix eigenvalues(const Matrix& A) {
    assert(A.rows() == A.cols());

```

```
    const std::size_t max_iterations = 500 * A.rows();
    std::size_t       iteration      = 0;
    Idx               N              = A.rows(); // mutable here since we shrink the working block (!)

    Matrix T_schur = A;

    while (N >= 2 && iteration++ < max_iterations) {
        const double sigma = T_schur(N - 1, N - 1); // O(1)
        [[maybe_unused]] const auto [Q, R, RQ] =
            qr_factorize_hessenberg(T_schur.block(0, 0, N, N) - sigma * Matrix::Identity(N, N)); // (N^2)
        T_schur.block(0, 0, N, N) = RQ + sigma * Matrix::Identity(N, N); // (N^2)
        if (std::abs(T_schur(N - 1, N - 2)) < std::numeric_limits<double>::epsilon()) --N; // O(1)
    }

    return T_schur;
}
```

source/main.cpp

```cpp
#include "eigenvalues.hpp"
#include "linear_least_squares.hpp"
#include "utils.hpp"
#include <cmath>



int main() {
    using namespace utl;

    // ===============
    // --- Problem ---
    // ===============

    constexpr double      Nvar    = 1;
    constexpr double      epsilon = 1e-6; // 0.1
    constexpr double      c       = Nvar / (Nvar + 1.) * epsilon;
    constexpr std::size_t N       = 4;

    // A0 = {  2, if (j == j)
    //      { -1, if (i == j - 1 || i == j + 1)
    //      {  0, else
    Matrix A0(N, N);
    for (Idx i = 0; i < A0.rows(); ++i)
        for (Idx j = 0; j < A0.cols(); ++j) A0(i, j) = (i == j) ? 2. : (std::abs(i - j) == 1) ? -1. : 0.;

    // deltaA = { c / (i + j), if (i != j)
    //          {           0, else
    Matrix deltaA(N, N);
    for (Idx i = 0; i < deltaA.rows(); ++i)
        for (Idx j = 0; j < deltaA.cols(); ++j) deltaA(i, j) = (i != j) ? c / (i + j + 2) : 0;

    // A      = A0 + deltaA
    const Matrix A = A0 + deltaA;

    // A_hat = <A without the last column>
    const Matrix A_hat = A.block(0, 0, A.rows(), A.cols() - 1);

    log::println("---------------");
    log::println("--- Problem ---");
    log::println("---------------");
    log::println();
    log::println("epsilon -> ", epsilon);
    log::println("N       -> ", N);
    log::println("A0      -> ", stringify_matrix(A0));
    log::println("deltaA  -> ", stringify_matrix(deltaA));
    log::println("A       -> ", stringify_matrix(A));
    log::println("A_hat   -> ", stringify_matrix(A_hat));

    // ==============
    // --- Task 1 ---
    // ==============
    //
    // Solving LLS (Linear Least Squares) with QR factorization method.
    //
```

```cpp

    // Try QR decomposition to verify that it works
    const auto [Q, R] = qr_factorize(A_hat);

    log::println("------------------------");
    log::println("--- QR factorization ---");
    log::println("------------------------");
    log::println();
    log::println("Q             -> ", stringify_matrix(Q));
    log::println("R             -> ", stringify_matrix(R));
    log::println("Verification:");
    log::println();
    log::println("Q^T * Q       -> ", stringify_matrix(Q.transpose() * Q));
    log::println("Q * R - A_hat -> ", stringify_matrix(Q * R - A_hat));

    // Generate some 'x0',
    // set b = A_hat * x0

    Vector x0(N - 1);
    for (Idx i = 0; i < x0.rows(); ++i) x0(i) = math::sqr(i + 1);
    const Vector b = A_hat * x0;

    // Solve LLS
    const Vector x_lls = linear_least_squares(A_hat, b);

    // Relative error estimate ||x_lls - x0||_2 / ||x0||_2
    const double lls_error_estimate = (x_lls - x0).norm() / x0.norm();

    log::println("-------------------------------------");
    log::println("--- Linear Least Squares solution ---");
    log::println("-------------------------------------");
    log::println();
    log::println("x0                 -> ", stringify_matrix(x0));
    log::println("b                  -> ", stringify_matrix(b));
    log::println("x_lls              -> ", stringify_matrix(x_lls));
    log::println("lls_error_estimate -> ", lls_error_estimate);
    log::println();

    // ==============
    // --- Task 2 ---
    // ==============
    //
    // Computing eigenvalues of the matrix using QR method with a shift.
    //

    // Compute analythical eigenvalues
    Vector lambda0(N);
    for (Idx j = 0; j < lambda0.size(); ++j) lambda0(j) = 2. * (1. - std::cos(math::PI * (j + 1) / (N + 1)));
    std::sort(lambda0.begin(), lambda0.end());

    // Compute analythical eigenvectors (columns of the matrix store vectors)
    Matrix z0(N, N);
    for (Idx k = 0; k < z0.cols(); ++k)
        for (Idx i = 0; i < z0.rows(); ++i)
            z0(i, k) = std::sqrt(2. / (N + 1)) * std::sin(math::PI * (i + 1) * (k + 1) / (N + 1));

    // Compute 'H' from Hessenberg decomposition 'A = P H P^*'
```

```cpp
    Matrix H_hessenberg = hessenberg_reduce(A);

    // Compute numeric eigenvalues
    const auto T_shur = eigenvalues(H_hessenberg);

    // Extract numeric eigenvalues as a sorted vector for comparison
    Vector lambda = T_shur.diagonal();
    std::sort(lambda.begin(), lambda.end());

    log::println("---------------------------");
    log::println("--- Eigenvalue solution ---");
    log::println("---------------------------");
    log::println();
    log::println("H_hessenberg                  -> ", stringify_matrix(H_hessenberg));
    log::println("T_shur                        -> ", stringify_matrix(T_shur));
    log::println("lambda0 (analythic eigenvals) -> ", stringify_matrix(lambda0));
    log::println("lambda    (numeric eigenvals) -> ", stringify_matrix(lambda));
    log::println("z0      (analythic eigenvecs) -> ", stringify_matrix(z0));

    table::create({4, 25, 25});
    table::hline();
    table::cell(" j ", " |lambda_j^0 - lambda_j| ", " ||z0_j - z-j||_2 ");
    table::hline();

    for (std::size_t j = 0; j < N; ++j) {
        table::cell(j + 1, std::abs(lambda0(j) - lambda(j)), "x");
    }

    return 0;
}
```

5. Листинг проверочного скрипта на Wolfram Mathematica

In[1234]:=

```
Nvar = 1;
ε = 0.1;
c = Nvar / (Nvar + 1) ε;
N = 4;

A0 = Table[Piecewise[{{2, i == j}, {-1, (i == j - 1) || (i == j + 1)}}, 0], {i, 1, N}, {j, 1, N}];
δA = Table[Piecewise[{{c / (i + j), i ≠ j}}, 0], {i, 1, N}, {j, 1, N}];
A = A0 + δA;
Ahat = A[[All, 1 ;; -2]];

{Q, R} = QRDecomposition[Ahat];
Q = Transpose@Q; (* for some reason Mathematica returns Q as transposed *)

x0 = Table[i * i, {i, 1, N - 1}];
b = Ahat.x0;

LLSx = Inverse[R].Transpose[Q].b;
LLSx2 = LeastSquares[Ahat, b]; (* Should give the same result as formula above *)

eigenvals = Reverse@Eigenvalues[A];
{ShurQ, ShurT} = SchurDecomposition[A];
(* Should give the same eigenvalues as method above *)
(* Eigenvalues will be stored on the main diagonal of 'T' *)
{HessP, HessH} = HessenbergDecomposition[A];

Framed@"Problem"
Row@{"A0 = ", A0 // MatrixForm}
Row@{"δA = ", δA // MatrixForm}
Row@{"A = ", A // MatrixForm}
Row@{"Â = ", Ahat // MatrixForm}
Framed@"QR decomposition"
Row@{"Q = ", Q // N // MatrixForm}
Row@{"R = ", R // N // MatrixForm}
Row@{"QTQ = ", Transpose[Q].Q // MatrixForm}
Row@{"QR = ", Q.R // MatrixForm}
Framed@"LLS"
Row@{"x0 = ", x0 // MatrixForm}
Row@{"b = ", b // MatrixForm}
Row@{"xLLS (formula) = ", LLSx // MatrixForm}
Row@{"xLLS (built-in) = ", LLSx2 // MatrixForm}
Framed@"Eigenvalue problem"
Row@{"{λi}Ni=1 = ", eigenvals // MatrixForm}
```

6. Приложение. Пример сводки результатов расчетной программы

```
---------------
--- Problem ---
---------------

epsilon -> 1e-06
N       -> 4
A0      -> Tensor [size = 16] (4 x 4):
  [  2 -1  0  0 ]
  [ -1  2 -1  0 ]
  [  0 -1  2 -1 ]
  [  0  0 -1  2 ]

deltaA  -> Tensor [size = 16] (4 x 4):
  [           0 1.66667e-07    1.25e-07        1e-07 ]
  [ 1.66667e-07           0       1e-07 8.33333e-08 ]
  [    1.25e-07       1e-07           0 7.14286e-08 ]
  [       1e-07 8.33333e-08 7.14286e-08           0 ]

A       -> Tensor [size = 16] (4 x 4):
  [        2          -1 1.25e-07       1e-07 ]
  [       -1           2       -1 8.33333e-08 ]
  [ 1.25e-07          -1        2          -1 ]
  [    1e-07 8.33333e-08       -1           2 ]

A_hat   -> Tensor [size = 12] (4 x 3):
  [        2          -1 1.25e-07 ]
  [       -1           2       -1 ]
  [ 1.25e-07          -1        2 ]
  [    1e-07 8.33333e-08       -1 ]

------------------------
--- QR factorization ---
------------------------

Q            -> Tensor [size = 12] (4 x 3):
  [    -0.894427    -0.358569 -0.19518 ]
```

```
  [     0.447214    -0.717137 -0.39036 ]
  [ -5.59017e-08     0.597614 -0.58554 ]
  [ -4.47214e-08 -9.76103e-08  0.68313 ]

R             -> Tensor [size = 9] (3 x 3):
  [     -2.23607      1.78885 -0.447214 ]
  [  2.22045e-16     -1.67332   1.91237 ]
  [ -2.64698e-23 -1.11022e-16  -1.46385 ]

Verification:

Q^T * Q       -> Tensor [size = 9] (3 x 3):
  [           1  2.77556e-16  8.32667e-17 ]
  [ 2.77556e-16            1 -2.22045e-16 ]
  [ 8.32667e-17 -2.22045e-16            1 ]

Q * R - A_hat -> Tensor [size = 12] (4 x 3):
  [  2.22045e-15 -1.77636e-15  6.83606e-16 ]
  [ -8.88178e-16 -8.88178e-16  1.33227e-15 ]
  [  1.32697e-16  3.33067e-16 -4.44089e-16 ]
  [  3.97047e-23 -7.58428e-17  2.22045e-16 ]

-------------------------------------
--- Linear Least Squares solution ---
-------------------------------------

x0                 -> Tensor [size = 3] (3 x 1):
  [ 1 ]
  [ 4 ]
  [ 9 ]

b                  -> Tensor [size = 4] (4 x 1):
  [ -2 ]
  [ -2 ]
  [ 14 ]
  [ -9 ]

x_lls              -> Tensor [size = 3] (3 x 1):
  [ 1 ]
  [ 4 ]
```

```
  [ 9 ]

lls_error_estimate -> 1.8496162997539822e-16

---------------------------
--- Eigenvalue solution ---
---------------------------

H_hessenberg                  -> Tensor [size = 16] (4 x 4):
  [           2            1 -2.64698e-23  1.32349e-23 ]
  [           1            2           -1 -2.64698e-23 ]
  [ 2.64698e-23           -1            2            1 ]
  [ 1.32349e-23 -2.64698e-23            1            2 ]

T_shur                        -> Tensor [size = 16] (4 x 4):
  [     3.61803 -1.46354e-16 -1.55722e-16 -1.88824e-16 ]
  [ 2.56137e-27     0.381966  4.50213e-16  3.42274e-16 ]
  [ 6.03553e-17  1.73417e-16      2.61803 -4.28483e-16 ]
  [ 1.15867e-17  1.13712e-16 -8.62537e-22      1.38197 ]

lambda0 (analythic eigenvals) -> Tensor [size = 4] (4 x 1):
  [ 0.381966 ]
  [  1.38197 ]
  [  2.61803 ]
  [  3.61803 ]

lambda    (numeric eigenvals) -> Tensor [size = 4] (4 x 1):
  [ 0.381966 ]
  [  1.38197 ]
  [  2.61803 ]
  [  3.61803 ]

z0      (analythic eigenvecs) -> Tensor [size = 16] (4 x 4):
  [ 0.371748  0.601501  0.601501  0.371748 ]
  [ 0.601501  0.371748 -0.371748 -0.601501 ]
  [ 0.601501 -0.371748 -0.371748  0.601501 ]
  [ 0.371748 -0.601501  0.601501 -0.371748 ]

|----|------------------------|------------------------|
|  j | |lambda_j^0 - lambda_j| |        ||z0_j - z-j||_2 |
```

```
|----|-------------------------|-------------------------|
|   1|             2.99649e-07|                        x|
|   2|             8.66901e-08|                        x|
|   3|             9.96489e-08|                        x|
|   4|             1.13310e-07|                        x|
```