source/main.cpp

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



int main() {
    using namespace utl;

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

    constexpr double      Nvar    = 1;
    constexpr double      epsilon = 1e-6; // 1e-1, 1e-3, 1e-6
    constexpr double      c       = Nvar / (Nvar + 1.) * epsilon;
    constexpr std::size_t N       = 10;

    // 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));

```

```cpp
    // Compute 'H' from Hessenberg decomposition 'A = P H P^*'
    Matrix H_hessenberg = hessenberg_reduce(A);

    // Compute numeric eigenvalues
    const auto [ T_shur, lambda_iteration_counts ] = qr_algorithm(H_hessenberg);

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

    // Compute numeric eigenvecs
    Matrix                   z(N, N);
    std::vector<std::size_t> z_iteration_counts(N);
    for (Idx k = 0; k < z0.cols(); ++k) {
        const auto res        = reverse_iteration(A, lambda(k));
        lambda(k)             = std::get<0>(res);
        z.col(k)              = std::get<1>(res); // Eigen doesn't like '
std::tie()'
        z_iteration_counts[k] = std::get<2>(res);
    }

    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));
    log::println("z       (analythic eigenvecs) -> ", stringify_matrix(z));

    table::set_latex_mode(true); // generate tables in export format

    table::create({4, 20, 25, 20, 25, 20});
    table::hline();
    table::cell(" j ", "lambda_j", " |lambda_j^0 - lambda_j| ", "Reduction
iterations", " ||z0_j - z-j||_2 ", "Reverse iterations");
    table::hline();
    for (std::size_t j = 0; j < N; ++j) {
        table::cell(j + 1, lambda(j), std::abs(lambda0(j) - lambda(j)),
lambda_iteration_counts[j], (z0.col(j) - z.col(j)).norm(), z_iteration_counts[j])
;
    }
    table::hline();

    return 0;
}
```