source/eigenvalues.hpp

```cpp
#pragma once

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

#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.
//
// Returns { T_shur, numer_of_iteration_for_each_eigenvalue }
```

```cpp
//
std::pair<Matrix, std::vector<std::size_t>> qr_algorithm(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;
    std::vector<std::size_t> iteration_counts;
    iteration_counts.reserve(N);

    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()) {          // O(1)
            --N;
            if (iteration_counts.empty()) iteration_counts.push_back(iteration);
            else iteration_counts.push_back(iteration - iteration_counts.back());
        }
    }

    return {T_schur, iteration_counts};
}

// Reverse iteration for computing eigenvectors and (possible) imroving eigenvalues.
//
// Returns { eigenvalue, eigenvector, number_of_iterations }
//
std::tuple<double, Vector, std::size_t> reverse_iteration(const Matrix& A, double lambda_0) {
    assert(A.rows() == A.cols());

    const std::size_t max_iterations = 1 * A.rows();
    std::size_t       iteration      = 0;
    const Idx         N              = A.rows();

    double lambda;
    Vector x = Vector::Ones(N) / N; // ||x0||_2 should be 1

    while (iteration++ < max_iterations) {
        x      = partial_piv_gaussian_elimination(A - lambda_0 * Matrix::Identity(N, N), x).normalized();
        lambda = x.transpose() * A * x;
        if (std::abs(lambda - lambda_0) < 1e-12) break;
        lambda_0 = lambda;
    }

    if (x(0) < 0) x *= -1; // "standardize" eigenvec signs
    return {lambda, x, iteration};
}
```