source/qr_factorization.hpp

```cpp
#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, i, 2, M - i) -= 2. * ui * (ui.transpose() * R.block(i, i, 2, M - i)); // 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 vi = V.block(i, i, 2, 1);
        RQ.block(0, i, M, 2) -= RQ.block(0, i, M, 2) * 2. * vi * vi.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;
}
```