

source/qr_factorization.hpp

```

1  #pragma once
2
3  #include "utils.hpp"
4
5
6
7  // Householder reflection operation.  $O(N)$  complexity.
8  //
9  // Template so we can take vectors/blocks/views as an argument and not force a
10 // copy.
11 //
12 template <class VectorType>
13 Vector householder_reflect(const VectorType& x) {
14     //  $u = \{ x[0] + \text{sign}(x[0]) * ||x||_2, x[1], x[2], \dots, x[K] \}$ 
15     Vector u = x;
16     u(0) += utl::math::sign(u(0)) * u.norm();
17
18     return u.normalized();
19 }
20
21 // QR factorization.  $O(N^3)$  complexity.
22 //
23 // Original alg would be:
24 // -----
25 // -   Q_wave = I;
26 // -   R_wave = A;
27 // -   for i = 1,min(M-1,N) {
28 // -       ui
29 // -         = House(R_wave[i:M, i])           //  $O(N)$ 
30 // -       pi_wave
31 // -         = I - 2 ui ui^T                    //  $O(N^2)$ 
32 // -       pi
33 // -         = I                               //
34 // -       pi[i:M, i:M]
35 // -         = pi_wave                         //
36 // -       R_wave[i:M, i:N]
37 // -         = pi_wave * R_wave[i:M, i:N]      //  $O(N^3)$ 
38 // -       Q_wave[0:M, i:M]
39 // -         = Q_wave[0:M, i:M] * pi_wave      //  $O(N^3)$ 
40 // -   }
41 // -   return { Q[0:M, 0:N], R[0:N, 0:N] }
42 // -----
43 //
44 // After the algorithm we end up with a following decomposition:
45 // A = p1 * ... * pN * rcat[ R 0 ]
46 //   ^^^^^^^^^^^^^   ^^^^^^^^^^^
47 //   Q_wave          R_wave
48 // where Q_wave and R_wave are "extended" matrices Q and R, to get proper QR we
49 // need to trim a few rows/cols at the end
50 //
51 // This alg also results in  $O(N^4)$ . We can rewrite it by substituting 'pi_wave'
52 // directly and doing
53 // 2 matrix*vector products instead of 1 matrix*matrix, which brings complexity
54 // down to  $O(N^3)$ .
55 //
56 // -----
57 // -   Q_wave = I;
58 // -   R_wave = A;
59 // -   for i = 1,min(M-1,N) {
60 // -       ui
61 // -         = House(R_wave[i:M, i])           //  $O(N)$ 
62 // -       R_wave[i:M, i:N]
63 // -         -= 2 ui (ui^T * R_wave[i:M, i:N]) //  $O(N^2)$ 
64 // -       Q_wave[0:M, i:M]
65 // -         -= Q_wave[0:M, i:M] * 2 ui ui^T   //  $O(N^2)$ 

```

```

54 // - } -
55 // - return { Q[0:M, 0:N], R[0:N, 0:N] } -
56 // -----
57 //
58 inline std::pair<Matrix, Matrix> qr_factorize(const Matrix& A) {
59     const auto M = A.rows();
60     const auto N = A.cols();
61
62     Matrix Q_wave = Matrix::Identity(M, M);
63     Matrix R_wave = A;
64
65     for (Idx i = 0; i < std::min(M - 1, N); ++i) {
66         const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1));
67         // O(N)
68         R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() *
69 R_wave.block(i, i, M - i, N - i)); // O(N^2)
70         Q_wave.block(0, i, M, M - i) -= Q_wave.block(0, i, M, M - i) * 2. * ui *
71 ui.transpose(); // O(N^2)
72     }
73
74     return {Q_wave.block(0, 0, M, N), R_wave.block(0, 0, N, N)};
75 }
76
77 // A variant of QR-decomposition used for linear least squares. O(N^3)
78 // complexity.
79 //
80 // Is is more efficient since in LSQ we don't need 'Q' explicitly,
81 // we can directly compute 'Q^T b'.
82 //
83 inline std::pair<Matrix, Matrix> qr_factorize_lls(const Matrix& A, const Vector&
84 b) {
85     const auto M = A.rows();
86     const auto N = A.cols();
87
88     Matrix R_wave = A;
89     Vector QTb = b;
90
91     for (Idx i = 0; i < std::min(M - 1, N); ++i) {
92         // ui = House(R_wave[i:M, i])
93         // R_wave[i:M, i:N] -= 2 ui ui^T * R_wave[i:M, i:N]
94         const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1));
95         // O(N)
96         R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() *
97 R_wave.block(i, i, M - i, N - i)); // O(N^2)
98
99         // gamma = - 2 ui^T QTb[i:M]
100         // QTb[i:M] += gamma * ui
101         const Matrix gamma = -2. * ui * QTb.segment(i, M - i).transpose(); //
102 O(N)
103 QTb.segment(i, M - i) += gamma * ui; //
104 O(N^2)
105     }
106
107     return {QTb.segment(0, N), R_wave.block(0, 0, N, N)};
108 }
109
110 // A variant of QR-decomposition used decomposing upper-hessenberg matrices in
111 // QR-iteration. O(N^2) complexity.
112 //
113 // Returns { Q, R, RQ }. Technically we only need RQ for for the QR-algorithm,
114 // but for testing purposes { Q, R}

```

```

104 // are left the same.
105 //
106 // Same algorithm as regular QR factorization, except instead of blocks of 'M -
107 // i' rows/cols we
108 // have blocks of '2' rows/cols, which reduces  $O(N^2)$  operations to  $O(N)$ .
109 //
110 inline std::tuple<Matrix, Matrix, Matrix> qr_factorize_hessenberg(const Matrix&
111 A) {
112     assert(A.rows() == A.cols());
113
114     const auto M = A.rows();
115
116     Matrix Q = Matrix::Identity(M, M);
117     Matrix R = A;
118     Matrix V = Matrix::Zero(M, M);
119
120     // Compute { Q, R } in  $O(N^2)$ 
121     for (Idx i = 0; i < M - 1; ++i) {
122         const Vector ui = householder_reflect(R.block(i, i, 2, 1));
123         //  $O(N)$ 
124         R.block(i, i, 2, M - i) -= 2. * ui * (ui.transpose() * R.block(i, i, 2, M
125 - i)); //  $O(N)$ 
126         Q.block(0, i, M, 2) -= Q.block(0, i, M, 2) * 2. * ui * ui.transpose();
127         //  $O(N)$ 
128         V.block(i, i, 2, 1) = ui;
129         //  $O(N)$ 
130     }
131
132     // Compute { RQ } in  $O(N^2)$ 
133     Matrix RQ = R;
134     for (Idx i = 0; i < M - 1; ++i) {
135         Vector vi = V.block(i, i, 2, 1);
136         RQ.block(0, i, M, 2) -= RQ.block(0, i, M, 2) * 2. * vi * vi.transpose();
137         //  $O(N)$ 
138     }
139
140     return {Q, R, RQ};
141 }
142
143 // Hessenberg QHQ^T-factorization using householder reflections.  $O(N^3)$ 
144 // complexity.
145 //
146 // Algorithm:
147 // -----
148 // - H = A;
149 // - for i = 1, M - 2 {
150 // -     ui = House(H[i+1:M, i]) //  $O(N)$ 
151 // -     H[i+1:M, i:M] -= 2 ui (ui^T * H[i+1:M, i:M]) //  $O(N^2)$ 
152 // -     H[1:M, i+1:M] -= 2 (H[1:M, i+1:M] * ui) ui^T //  $O(N^2)$ 
153 // - }
154 // -----
155 //
156 inline Matrix hessenberg_reduce(const Matrix& A) {
157     assert(A.rows() == A.cols());
158
159     const Idx M = A.rows();
160
161     Matrix H = A;
162
163     for (Idx i = 0; i < M - 2; ++i) {
164         const Vector ui = householder_reflect(H.block(i + 1, i, M - i - 1, 1));

```

```
157     H.block(i + 1, i, M - i - 1, M - i) -= 2. * ui * (ui.transpose() *  
H.block(i + 1, i, M - i - 1, M - i));  
158     H.block(0, i + 1, M, M - i - 1) -= 2. * (H.block(0, i + 1, M, M - i - 1)  
* ui) * ui.transpose();  
159     }  
160  
161     return H;  
162 }
```