

Evaluation

Pieter P

poly.hpp

```
1  #pragma once
2
3  #include <Eigen/Dense>
4  #include <algorithm>
5  #include <utility>
6  #include <vector>
7
8  namespace poly {
9
10 template <class T = double>
11 using vector_t = Eigen::VectorX<T>;
12 template <class T = double>
13 using vector_ref_t = Eigen::Ref<const vector_t<T>>;
14 template <class T = double>
15 using vector_mut_ref_t = Eigen::Ref<vector_t<T>>;
16 template <class T = double>
17 using coef_t = vector_t<T>;
18 using index_t = Eigen::Index;
19
20 template <class T, class BasisTag>
21 struct GenericPolynomial {
22     GenericPolynomial() = default;
23     GenericPolynomial(coef_t<T> coefficients)
24         : coefficients({std::move(coefficients)}) {}
25     explicit GenericPolynomial(index_t degree)
26         : coefficients {coef_t<T>::Zeros(degree + 1)} {}
27     explicit GenericPolynomial(std::initializer_list<T> coefficients)
28         : coefficients {coefficients.size()} {
29         std::copy(std::begin(coefficients), std::end(coefficients),
30                 std::begin(this->coefficients));
31     }
32     coef_t<T> coefficients;
33 };
34
35 struct MonomialBasis_t {
36 } inline constexpr MonomialBasis;
37 struct ChebyshevBasis_t {
38 } inline constexpr ChebyshevBasis;
39
40 template <class T = double>
41 using Polynomial = GenericPolynomial<T, MonomialBasis_t>;
42 template <class T = double>
43 using ChebyshevPolynomial = GenericPolynomial<T, ChebyshevBasis_t>;
44
45 namespace detail {
46 template <class Container>
47 vector_ref_t<typename Container::value_type>
48 vector_map_ref(const Container &x) {
49     return Eigen::Map<const vector_t<typename Container::value_type>>(x.data(),
50                                                                     x.size());
51 }
52 template <class Container>
53 vector_mut_ref_t<typename Container::value_type> vector_map_ref(Container &x) {
54     return Eigen::Map<vector_t<typename Container::value_type>>(x.data(),
55                                                                     x.size());
56 }
57 } // namespace detail
58
59 } // namespace poly
```

poly_eval.hpp

```
1  #pragma once
2
3  #include <poly.hpp>
4  #include <vector>
5
6  namespace poly {
7
8  namespace detail {
9
10     /// Tail-recursive implementation to allow C++11 constexpr.
11     /// @param x Point to evaluate at
12     /// @param p Polynomial coefficients
13     /// @param n Index of current coefficient (number of remaining iterations)
14     /// @param b Temporary value @f$ b_{n-1} = p_n + b_n x @f$
15     template <class T, class P>
16     constexpr T horner_impl(T x, const P &p, index_t n, T b) {
17         return n == 0 ? p[n] + x * b // base case
18             : horner_impl(x, p, n - 1, p[n] + x * b);
19     }
20
21     /// Evaluate a polynomial using [Horner's method](https://en.wikipedia.org/wiki/Horner%27s_method).
22     template <class T, class P>
23     constexpr T horner(T x, const P &p, index_t n) {
24         return n == 0 ? T{0} // empty polynomial
25             : n == 1 ? p[0] // constant
26                 : horner_impl(x, p, n - 2, p[n - 1]);
27     }
28
29     template <class T, size_t N>
30     constexpr T horner(T x, const T (&coef)[N]) {
31         return horner(x, &coef[0], N);
32     }
33
34     template <class T>
35     constexpr T horner(T x, const Polynomial<T> &poly) {
36         return horner(x, poly.coefficients.data(), poly.coefficients.size());
37     }
38
39 } // namespace detail
40
41 template <class T>
42 constexpr T evaluate(const Polynomial<T> &poly, T x) {
43     return detail::horner(x, poly);
44 }
45
46 namespace detail {
47
48     /// Tail-recursive implementation to allow C++11 constexpr.
49     /// @param x Point to evaluate at
50     /// @param c Polynomial coefficients
51     /// @param n Index of current coefficient (number of remaining iterations)
52     /// @param b1 Temporary value @f$ b^1_{n-1} = c_n + 2 b^1_n x - b^2_n @f$
53     /// @param b2 Temporary value @f$ b^2_{n-1} = b^1_n @f$
54     template <class T, class C>
55     constexpr T clenshaw_cheb_impl(T x, const C &c, size_t n, T b1, T b2) {
56         return n == 0 ? c[n] + x * b1 - b2 // base case
57             : clenshaw_cheb_impl(x, c, n - 1, c[n] + 2 * x * b1 - b2, b1);
58     }
59
60     /// Evaluate a Chebyshev polynomial using [Clenshaw's algorithm](https://en.wikipedia.org/wiki/Clenshaw_algorithm).
61     template <class T, class C>
62     constexpr T clenshaw_cheb(T x, const C &c, index_t n) {
63         return n == 0 ? T{0} // empty polynomial
64             : n == 1 ? c[0] // constant
65                 : n == 2 ? c[0] + x * c[1] // linear
66                     : clenshaw_cheb_impl(x, c, n - 3, c[n - 2] + 2 * x * c[n - 1],
67                         c[n - 1]);
68     }
69
70     template <class T, size_t N>
71     constexpr T clenshaw_cheb(T x, const T (&coef)[N]) {
72         return clenshaw_cheb(x, &coef[0], N);
73     }
74
75     template <class T>
76     constexpr T clenshaw_cheb(T x, const ChebyshevPolynomial<T> &poly) {
77         return clenshaw_cheb(x, poly.coefficients.data(), poly.coefficients.size());
78     }
79
80 } // namespace detail
81
82 template <class T>
83 constexpr T evaluate(const ChebyshevPolynomial<T> &poly, T x) {
84     return detail::clenshaw_cheb(x, poly);
85 }
86
87 template <class T, class Basis>
88 constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
89     vector_ref_t<T> x, vector_mut_ref_t<T> y) {
90     y = x.unaryExpr([&](double x) { return evaluate(poly, x); });
91 }
92
93 template <class T, class Basis>
```

```

94 constexpr vector_t<T> evaluate(const GenericPolynomial<T, Basis> &poly,
95                               vector_ref_t<T> x) {
96     return x.unaryExpr([&](double x) { return evaluate(poly, x); });
97 }
98
99 template <class T, class Basis>
100 constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
101                        const vector_t<T> &x, vector_t<T> &y) {
102     evaluate(poly, vector_ref_t<T> {x}, vector_mut_ref_t<T> {y});
103 }
104
105 template <class T, class Basis>
106 constexpr vector_t<T> evaluate(const GenericPolynomial<T, Basis> &poly,
107                               const vector_t<T> &x) {
108     return evaluate(poly, vector_ref_t<T> {x});
109 }
110
111 template <class T, class Basis>
112 constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
113                        const std::vector<T> &x, std::vector<T> &y) {
114     evaluate(poly, detail::vector_map_ref(x), detail::vector_map_ref(y));
115 }
116
117 template <class T, class Basis>
118 constexpr std::vector<T> evaluate(const GenericPolynomial<T, Basis> &poly,
119                                  const std::vector<T> &x) {
120     std::vector<T> y(x.size());
121     evaluate(poly, x, y);
122     return y;
123 }
124
125 } // namespace poly

```

poly_eval.cpp

```

1 #include <iostream>
2 #include <poly_eval.hpp>
3
4 int main() {
5     // Create the polynomial  $p(x) = 1 - 2x + 3x^2 - 4x^3 + 5x^4$ 
6     poly::Polynomial<> p {1, -2, 3, -4, 5};
7     // Evaluate the polynomial at  $x = 3$ 
8     std::cout << evaluate(p, 3.) << std::endl;
9 }

```