

Evaluation

Pieter P

poly.hpp

```
1  #pragma once
2
3  #include <Eigen/Dense>
4  #include <algorithm>
5  #include <utility>
6
7  namespace poly {
8
9  template <class T = double>
10 using coef_t = Eigen::VectorX<T>;
11 using index_t = Eigen::Index;
12
13 template <class T, class BasisTag>
14 struct GenericPolynomial {
15     GenericPolynomial() = default;
16     GenericPolynomial(coef_t<T> coefficients)
17         : coefficients({std::move(coefficients)}) {}
18     explicit GenericPolynomial(index_t degree)
19         : coefficients {coef_t<T>::Zeros(degree + 1)} {}
20     explicit GenericPolynomial(std::initializer_list<T> coefficients)
21         : coefficients {coefficients.size()} {
22         std::copy(std::begin(coefficients), std::end(coefficients),
23             std::begin(this->coefficients));
24     }
25     coef_t<T> coefficients;
26 };
27
28 struct MonomialBasis_t {
29 } inline constexpr MonomialBasis;
30 struct ChebyshevBasis_t {
31 } inline constexpr ChebyshevBasis;
32
33 template <class T = double>
34 using Polynomial = GenericPolynomial<T, MonomialBasis_t>;
35 template <class T = double>
36 using ChebyshevPolynomial = GenericPolynomial<T, ChebyshevBasis_t>;
37
38 } // namespace poly
```

poly_eval.hpp

```

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