Evaluation

Pieter P

poly.hpp

```cpp
#pragma once

#include <Eigen/Dense>
#include <algorithm>
#include <utility>
#include <vector>

namespace poly {

template <class T = double>
using vector_t = Eigen::VectorX<T>;
template <class T = double>
using vector_ref_t = Eigen::Ref<const vector_t<T>>;
template <class T = double>
using vector_mut_ref_t = Eigen::Ref<vector_t<T>>;
template <class T = double>
using coef_t = vector_t<T>;
using index_t = Eigen::Index;

template <class T, class BasisTag>
struct GenericPolynomial {
    GenericPolynomial() = default;
    GenericPolynomial(coef_t<T> coefficients)
        : coefficients({std::move(coefficients)}) {}
    explicit GenericPolynomial(index_t degree)
        : coefficients {coef_t<T>::Zeros(degree + 1)} {}
    explicit GenericPolynomial(std::initializer_list<T> coefficients)
        : coefficients {coefficients.size()} {
        std::copy(std::begin(coefficients), std::end(coefficients),
                  std::begin(this->coefficients));
    }
    coef_t<T> coefficients;
};

struct MonomialBasis_t {
} inline constexpr MonomialBasis;
struct ChebyshevBasis_t {
} inline constexpr ChebyshevBasis;

template <class T = double>
using Polynomial = GenericPolynomial<T, MonomialBasis_t>;
template <class T = double>
using ChebyshevPolynomial = GenericPolynomial<T, ChebyshevBasis_t>;

namespace detail {
template <class Container>
vector_ref_t<typename Container::value_type>
vector_map_ref(const Container &x) {
    return Eigen::Map<const vector_t<typename Container::value_type>>(x.data(),
                                                                      x.size());
}
template <class Container>
vector_mut_ref_t<typename Container::value_type> vector_map_ref(Container &x) {
    return Eigen::Map<vector_t<typename Container::value_type>>(x.data(),
                                                                x.size());
}
} // namespace detail

} // namespace poly
```

poly_eval.hpp

```cpp
#pragma once

#include <poly.hpp>
#include <vector>

namespace poly {

namespace detail {

/// Tail-recursive implementation to allow C++11 constexpr.
/// @param  x   Point to evaluate at
/// @param  p   Polynomial coefficients
/// @param  n   Index of current coefficient (number of remaining iterations)
/// @param  b   Temporary value @f$ b_{n-1} = p_n + b_n x @f$
template <class T, class P>
constexpr T horner_impl(T x, const P &p, index_t n, T b) {
    return n == 0 ? p[n] + x * b // base case
                  : horner_impl(x, p, n - 1, p[n] + x * b);
}

/// Evaluate a polynomial using [Horner's method](https://en.wikipedia.org/wiki/Horner%27s_method).
template <class T, class P>
constexpr T horner(T x, const P &p, index_t n) {
    return n == 0 ? T {0} // empty polynomial
         : n == 1 ? p[0]  // constant
                  : horner_impl(x, p, n - 2, p[n - 1]);
}

template <class T, size_t N>
constexpr T horner(T x, const T (&coef)[N]) {
    return horner(x, &coef[0], N);
}

template <class T>
constexpr T horner(T x, const Polynomial<T> &poly) {
    return horner(x, poly.coefficients.data(), poly.coefficients.size());
}

} // namespace detail

template <class T>
constexpr T evaluate(const Polynomial<T> &poly, T x) {
    return detail::horner(x, poly);
}

namespace detail {

/// Tail-recursive implementation to allow C++11 constexpr.
/// @param  x   Point to evaluate at
/// @param  c   Polynomial coefficients
/// @param  n   Index of current coefficient (number of remaining iterations)
/// @param  b1  Temporary value @f$ b^1_{n-1} = c_n + 2 b^1_n x - b^2_n @f$
/// @param  b2  Temporary value @f$ b^2_{n-1} = b^1_n @f$
template <class T, class C>
constexpr T clenshaw_cheb_impl(T x, const C &c, size_t n, T b1, T b2) {
    return n == 0 ? c[n] + x * b1 - b2 // base case
                  : clenshaw_cheb_impl(x, c, n - 1, c[n] + 2 * x * b1 - b2, b1);
}

/// Evaluate a Chebyshev polynomial using [Clenshaw's algorithm](https://en.wikipedia.org/wiki/Clenshaw_algorithm).
template <class T, class C>
constexpr T clenshaw_cheb(T x, const C &c, index_t n) {
    return n == 0 ? T {0}           // empty polynomial
         : n == 1 ? c[0]            // constant
         : n == 2 ? c[0] + x * c[1] // linear
                  : clenshaw_cheb_impl(x, c, n - 3, c[n - 2] + 2 * x * c[n - 1],
                                       c[n - 1]);
}

template <class T, size_t N>
constexpr T clenshaw_cheb(T x, const T (&coef)[N]) {
    return clenshaw_cheb(x, &coef[0], N);
}

template <class T>
constexpr T clenshaw_cheb(T x, const ChebyshevPolynomial<T> &poly) {
    return clenshaw_cheb(x, poly.coefficients.data(), poly.coefficients.size());
}

} // namespace detail

template <class T>
constexpr T evaluate(const ChebyshevPolynomial<T> &poly, T x) {
    return detail::clenshaw_cheb(x, poly);
}

template <class T, class Basis>
constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
                        vector_ref_t<T> x, vector_mut_ref_t<T> y) {
    y = x.unaryExpr([&](double x) { return evaluate(poly, x); });
}

template <class T, class Basis>
constexpr vector_t<T> evaluate(const GenericPolynomial<T, Basis> &poly,
```

```cpp
                                vector_ref_t<T> x) {
    return x.unaryExpr([&](double x) { return evaluate(poly, x); });
}

template <class T, class Basis>
constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
                        const vector_t<T> &x, vector_t<T> &y) {
    evaluate(poly, vector_ref_t<T> {x}, vector_mut_ref_t<T> {y});
}

template <class T, class Basis>
constexpr vector_t<T> evaluate(const GenericPolynomial<T, Basis> &poly,
                               const vector_t<T> &x) {
    return evaluate(poly, vector_ref_t<T> {x});
}

template <class T, class Basis>
constexpr void evaluate(const GenericPolynomial<T, Basis> &poly,
                        const std::vector<T> &x, std::vector<T> &y) {
    evaluate(poly, detail::vector_map_ref(x), detail::vector_map_ref(y));
}

template <class T, class Basis>
constexpr std::vector<T> evaluate(const GenericPolynomial<T, Basis> &poly,
                                  const std::vector<T> &x) {
    std::vector<T> y(x.size());
    evaluate(poly, x, y);
    return y;
}

} // namespace poly
```

poly_eval.cpp

```cpp
#include <iostream>
#include <poly_eval.hpp>

int main() {
    // Create the polynomial p(x) = 1 - 2x + 3x² - 4x³ + 5x⁴
    poly::Polynomial<> p {1, -2, 3, -4, 5};
    // Evaluate the polynomial at x = 3
    std::cout << evaluate(p, 3.) << std::endl;
}
```