Interpolation

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_interp.hpp

```cpp
#pragma once

#include <Eigen/LU>
#include <poly.hpp>
#include <vector>

namespace poly {

namespace detail {

template <class T, class F>
coef_t<T> interpolate(vector_ref_t<T> x, vector_ref_t<T> y, F &&vanderfun) {
    assert(x.size() == y.size());
    assert(x.size() > 0);
    // Construct Vandermonde matrix
    auto V = vanderfun(x, x.size() - 1);
    // Scale the system
    const vector_t<T> scaling = V.colwise().norm().cwiseInverse();
    V *= scaling.asDiagonal();
    // Solve the system
    vector_t<T> solution = V.fullPivLu().solve(y);
    solution.transpose() *= scaling.asDiagonal();
    return solution;
}

template <class T>
auto make_monomial_vandermonde_system(vector_ref_t<T> x, index_t degree) {
    assert(degree >= 0);
    const index_t N = x.size();
    Eigen::MatrixXd V(N, degree + 1);
    V.col(0) = Eigen::VectorXd::Ones(N);
    for (Eigen::Index i = 0; i < degree; ++i)
        V.col(i + 1) = V.col(i).cwiseProduct(x);
    return V;
}

} // namespace detail

template <class T>
Polynomial<T> interpolate(vector_ref_t<T> x, vector_ref_t<T> y,
                          MonomialBasis_t) {
    auto *vanderfun = detail::make_monomial_vandermonde_system<T>;
    auto coef = detail::interpolate(x, y, vanderfun);
    return {std::move(coef)};
}

namespace detail {

template <class T>
auto make_chebyshev_vandermonde_system(vector_ref_t<T> x, index_t degree) {
    assert(degree >= 0);
    const index_t N = x.size();
    Eigen::MatrixXd V(N, degree + 1);
    V.col(0) = Eigen::VectorXd::Ones(N);
    if (degree >= 1) {
        V.col(1) = x;
        for (Eigen::Index i = 0; i < degree - 1; ++i)
            V.col(i + 2) = 2 * V.col(i + 1).cwiseProduct(x) - V.col(i);
    }
    return V;
}

} // namespace detail

template <class T>
ChebyshevPolynomial<T> interpolate(vector_ref_t<T> x, vector_ref_t<T> y,
                                   ChebyshevBasis_t) {
    auto *vanderfun = detail::make_chebyshev_vandermonde_system<T>;
    auto coef = detail::interpolate(x, y, vanderfun);
    return {std::move(coef)};
}

template <class T, class Basis>
GenericPolynomial<T, Basis> interpolate(const vector_t<T> &x,
                                        const vector_t<T> &y, Basis basis) {
    return interpolate(vector_ref_t<T> {x}, vector_ref_t<T> {y}, basis);
}

template <class T, class Basis>
GenericPolynomial<T, Basis> interpolate(const std::vector<T> &x,
                                        const std::vector<T> &y, Basis basis) {
    return interpolate(detail::vector_map_ref(x), detail::vector_map_ref(y),
                       basis);
}

} // namespace poly
```

poly_interp.cpp

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

int main() {
    std::cout.precision(17);
    // Evaluate some points on this polynomial
    poly::Polynomial<> p {1, -2, 3, -4, 5};
    poly::vector_t<> x = poly::vector_t<>::LinSpaced(5, -1, 1);
    poly::vector_t<> y = evaluate(p, x);
    // Interpolate the data
    poly::Polynomial<> p_interp = interpolate(x, y, poly::MonomialBasis);
    std::cout << p_interp.coefficients.transpose() << std::endl;

    // Now do the same with std::vectors
    std::vector<double> vx {x.begin(), x.end()};
    std::vector<double> vy {y.begin(), y.end()};
    p_interp = interpolate(vx, vy, poly::MonomialBasis);
    std::cout << p_interp.coefficients.transpose() << std::endl;

    // Fit a Chebyshev polynomial through the same points
    poly::ChebyshevPolynomial<> p_cheb =
        interpolate(x, y, poly::ChebyshevBasis);
    std::cout << p_cheb.coefficients.transpose() << std::endl;
    // Evaluate the polynomial again to verify
    poly::vector_t<> yc = evaluate(p_cheb, x);
    std::cout << y.transpose() << std::endl;
    std::cout << yc.transpose() << std::endl;
}
```