

ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Chapter 5

Data Interpolation and Data Fitting in 1D

Problem 5-1: Evaluating the derivatives of interpolating polynomials

In [Lecture → Ex. 5.1.0.8] we learned about the importance of data interpolation for obtaining functor representations of constitutive relationships $t \mapsto f(t)$. Numerical methods like Newton's method [Lecture → Code 8.4.2.2] often require information about the derivative f' as well. Therefore, we need efficient ways to evaluate the derivatives of interpolants. In this problem we discuss this issue for polynomial interpolation for (i) monomial representation and (ii) “update-friendly” point evaluation. We generalize the Horner scheme [Lecture → § 5.2.1.5] and the Aitken-Neville algorithm [Lecture → § 5.2.3.8].

This problem is connected with [Lecture → Section 5.2.3], simple coding in C++/EIGEN is requested.

We first deal with polynomials in monomial representation [Lecture → Rem. 5.2.1.4].

(5-1.a) (20 min.) Using the Horner scheme [Lecture → § 5.2.1.5], in the file `eval_deriv.hpp` write an efficient C++ implementation of a template function which returns the pair $(p(x), p'(x))$, where p is a polynomial with coefficients in `c`:

```
std::pair<double, double> evaldp(const CoeffVec &c, double x);
```

Vector `c` contains the coefficient of the polynomial in the monomial basis, following the PYTHON/MATLAB convention (leading coefficient in `c[0]`), that is,

$$p(t) = c[0]t^k + c[1]t^{k-1} + \dots + c[k-1]t + c[k], \quad t \in \mathbb{R}. \quad (5.1.1)$$

As indicated above, the type `CoeffVec` must provide component access via `operator [] (int) const` and a `size()` method.

SOLUTION of (5-1.a):

The idea is to apply the Horner scheme [Lecture → Eq. (5.2.1.6)] to

$$p'(t) = kc[0]t^{k-1} + (k-1)c[1]t^{k-2} + \dots + 2c[k-2]t + c[k-1], \quad t \in \mathbb{R}. \quad (5.1.2)$$

The corresponding extension of [Lecture → Code 5.2.1.7] is straightforward.

C++-code 5.1.3: Implementation of `evaldp()`

```

2  std::pair<double, double> evaldp(const Eigen::VectorXd &c, const double x) {
3      double px, dpx;
4      const int s = c.size(); // degree + 1
5
6      // TODO (5-1.a): Implement the function evaldp().
7
8      // START
9      // Standard Horner scheme
10     px = c(0);
11     for (int i = 1; i < s; ++i) {
12         px = x * px + c(i);
13     }
14
15     // Horner scheme for derivative
16     dpx = (s - 1) * c(0);
17     for (int i = 1; i < s - 1; ++i) {
18         dpx = x * dpx + (s - i - 1) * c(i);
19     }
20     // END
21
22     return {px, dpx};
23 }

```

Get it on  [GitLab \(eval_deriv.hpp\)](#).



(5-1.b)  (15 min.) For the sake of testing, write a naive C++ implementation of the evaluation requested in (5-1.a)

`std::pair<double, double> evaldp_naive(const CoeffVec &c, double x);`

which returns the same pair $(p(x), p'(x))$. However, this time $p(x)$ and $p'(x)$ should be calculated by means of simply summing the contributions of the monomials $t \mapsto t^k$.

SOLUTION of (5-1.b):

The following code uses `std::pow` to evaluate x^k .

C++-code 5.1.4: Solution of (5-1.b): `evaldp_naive()`

```

2  std::pair<double, double> evaldp_naive(const Eigen::VectorXd &c,
3                                          const double x) {
4      double px, dpx;
5      const int n = c.size();
6
7      // TODO (5-1.b): Implement the function evaldp_naive().
8
9      // START
10     px = c[0] * std::pow(x, n - 1);
11     for (int i = 1; i < n; ++i) {
12         px = px + c[i] * std::pow(x, n - i - 1);
13     }
14
15     dpx = (n - 1) * c[0] * std::pow(x, n - 2);

```

```

16   for (int i = 1; i < n - 1; ++i) {
17       dpx = dpx + (n - i - 1) * c[i] * std::pow(x, n - i - 2);
18   }
19   // END
20
21   return {px, dpx};
22 }

```

Get it on  GitLab (eval_deriv.hpp).



(5-1.c)  (5 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b)]

What are the asymptotic complexities of the two functions you implemented in (5-1.a) and (5-1.b) in terms of raising the polynomial degree?

SOLUTION of (5-1.c):

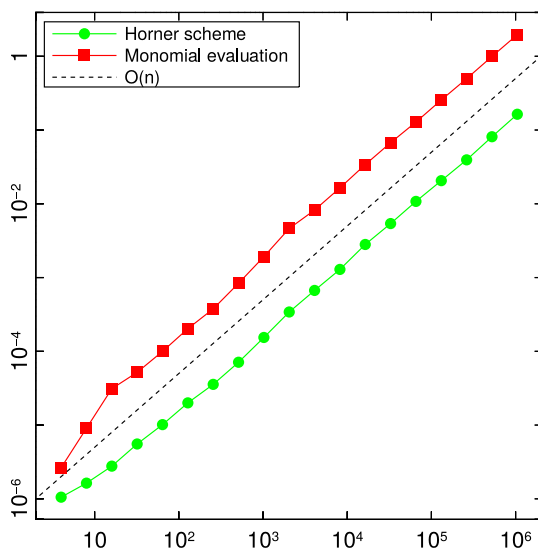


Fig. 30

The answer depends on your implementation. In any case, the code for `evaldp()` from Sub-problem (5-1.a) should require $\approx n$ multiplications and additions, and the asymptotic complexity is therefore $\mathcal{O}(n)$ for $n \rightarrow \infty$, where n is the degree of the polynomial.

Also the algorithm from Code 5.1.4 requires $\approx n$ multiplications and additions. However, the implementation also calls the `pow()` function, which may be costly. Beside in Fig. 30 we report comparative runtime measurements for the two implementation. They reveal that `std::pow()` probably incurs only constant computational cost!



(5-1.d)  (15 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b)]

Implement a function

```
bool polyTestTime(const unsigned int d);
```

that

- validates your implementation of the two functions `evaldp()/evaldp_naive()` from (5-1.a)/(5-1.b) by comparing their results for $x = 1.23$ and a polynomial of degree n with monomial coefficients $c_0 = 1, c_1 = 2, \dots, c_n = n$ and $n = 2, 4, 8, \dots, 2^d$. Return **false** in case of a discrepancy.
- measures the runtimes of the two functions for every degree and tabulates them. To that end use the **Timer** class defined in `timer.h`:

```

Timer t;
t.start(); /* do something */ t.stop();
double time = t.duration();

```

HINT 1 for (5-1.d): Please recall from [Lecture → Rem. 1.5.3.15] the advice on comparing numerically computed values

SOLUTION of (5-1.d):

C++-code 5.1.5: Solution of Sub-problem (5-1.d)

```

2  bool polyTestTime(const unsigned int d) {
3      bool ret = false; // Return value
4      double x = 0.123;
5      std::pair<double, double> p, p_naive;
6      const int repeats = 10;
7
8      std::cout
9          << std::setw(10) << "n" << std::setw(25)
10         << "Horner scheme:" << std::setw(25) << "Monomial approach:"
11         << "\n"
12         << "===== \n";
13
14     // TODO (5-1.d): Compare the output of evaldp() and evaldp_naive() for
15     // x = 0.123 and tabulate their runtimes for n = 2, 4, ..., 2^d.
16
17     // START
18     const double TOL = 1E-8;
19     const int max_n = std::pow(2, d);
20     Eigen::VectorXd c = Eigen::VectorXd::LinSpaced(max_n + 1, 1, max_n + 1);
21
22     for (int n = 2; n <= max_n; n *= 2) {
23         Timer tm_slow, tm_fast;
24
25         for (int r = 0; r < repeats; r++) {
26             tm_fast.start();
27             p = evaldp(c.head(n + 1), x);
28             tm_fast.stop();
29
30             tm_slow.start();
31             p_naive = evaldp_naive(c.head(n + 1), x);
32             tm_slow.stop();
33         }
34
35         std::cout << std::setw(10) << n << std::setw(25) << tm_fast.mean()
36         << std::setw(25) << tm_slow.mean() << "\n";
37
38         // Check correctness
39         double err_p = std::abs(p.first - p_naive.first);
40         double err_dp = std::abs(p.second - p_naive.second);
41
42         if (err_p >= TOL || err_dp >= TOL) {
43             std::cout << "Discrepancy at n = " << n << std::endl;
44
45             return false;
46         }
47     }
48 }

```

```

49 |   ret = true;
50 |   // END
51 |
52 |   return ret;
53 | }

```

Get it on  GitLab (eval_deriv.hpp).



Now we revisit polynomial interpolation. In [Lecture → Section 5.2.3.2] we learned about an efficient and “update-friendly” scheme for evaluating Lagrange interpolants at a *single or a few* points. This so-called **Aitken-Neville algorithm**, see [Lecture → Code 5.2.3.10], can be extended to return the derivative value of the polynomial interpolant as well. This will be explored next.

(5-1.e)  (30 min.) In the file `eval_deriv.hpp` write an efficient C++ function

```

Eigen::VectorXd dipoleval(const Eigen::VectorXd & t,
                          const Eigen::VectorXd & y,
                          const Eigen::VectorXd & x);

```

that returns the row vector $[p'(x_1), \dots, p'(x_m)]^\top$, when the argument x passes $[x_1, \dots, x_m]$, $m \in \mathbb{N}$ small. Here, p' denotes the *derivative* of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i) , $i = 0, \dots, n$, for pairwise different $t_i \in \mathbb{R}$ and data values $y_i \in \mathbb{R}$.

To that end differentiate the recursion formula [Lecture → Eq. (5.2.3.9)] and devise an algorithm in the spirit of the Aitken-Neville algorithm implemented in [Lecture → Code 5.2.3.10].

HINT 1 for (5-1.e): You may EIGEN's **Array** facilities and componentwise operations to achieve a vectorized implementation. 

SOLUTION of (5-1.e):

Differentiating the recursion formula [Lecture → Eq. (5.2.3.9)] we obtain

$$\begin{aligned}
 p_i(t) &\equiv y_i, \quad p'_i(t) \equiv 0, \quad i = 0, \dots, n, \\
 p_{i_0, \dots, i_m}(t) &= \frac{(t - t_{i_0})p_{i_1, \dots, i_m}(t) - (t - t_{i_m})p_{i_0, \dots, i_{m-1}}(t)}{t_{i_m} - t_{i_0}}, \\
 p'_{i_0, \dots, i_m}(t) &= \frac{p_{i_1, \dots, i_m}(t) + (t - t_{i_0})p'_{i_1, \dots, i_m}(t) - p_{i_0, \dots, i_{m-1}}(t) - (t - t_{i_m})p'_{i_0, \dots, i_{m-1}}(t)}{t_{i_m} - t_{i_0}}.
 \end{aligned}$$

C++-code 5.1.6: Example code using Aitken-Neville scheme

```

2 | Eigen::VectorXd dipoleval(const Eigen::VectorXd &t, const Eigen::VectorXd &y,
3 |                          const Eigen::VectorXd &x) {
4 |     assert(t.size() == y.size() && "t and y must have same size!");
5 |     Eigen::VectorXd ret;
6 |     ret.resizeLike(x); // Return value
7 |
8 |     // TODO (5-1.e): Use the Aitken-Neville algorithm to evaluate at x
9 |     // the derivative of the interpolating polynomial of the data (t,y).
10 |
11 |     // START

```

```

12 // Non-vectorized version: loop over all evaluation points
13 for (int i = 0; i < x.size(); ++i) {
14     Eigen::VectorXd p(y);
15     Eigen::VectorXd dP = Eigen::VectorXd::Zero(y.size());
16
17     // Aitken-Neville outer loop
18     for (int im = 1; im < y.size(); ++im) {
19         for (int i0 = im - 1; i0 >= 0; --i0) {
20             dP(i0) = (p(i0 + 1) + (x(i) - t(i0)) * dP(i0 + 1) - p(i0) -
21                     (x(i) - t(im)) * dP(i0)) /
22                     (t(im) - t(i0)));
23
24             p(i0) = ((x(i) - t(i0)) * p(i0 + 1) - (x(i) - t(im)) * p(i0)) /
25                     (t(im) - t(i0)));
26         }
27     }
28
29     ret(i) = dP(0);
30 }
31 // END
32
33 return ret;
34 }

```

Get it on  GitLab (eval_deriv.hpp).



(5-1.f)  (30 min.) For validation purposes devise an alternative, less efficient, implementation of `dipoleval()`

```

Eigen::VectorXd dipoleval_alt(const Eigen::VectorXd &t,
                             const Eigen::VectorXd &y,
                             const Eigen::VectorXd &x);

```

based on the following algorithm:

1. Use the function

```

Eigen::VectorXd polyfit(const Eigen::VectorXd& t,
                       const Eigen::VectorXd& y,
                       unsigned int degree);

```

(supplied in `polyfit.hpp`) to compute the monomial coefficients of the Lagrange interpolant of the data points.

2. Compute the monomial coefficients of the derivative.
3. Use the function

```

void polyval(const Eigen::VectorXd& p,
            const Eigen::VectorXd& x,
            Eigen::VectorXd& y);

```

(defined in `polyval.hpp`) to evaluate the derivative at a number of points. This function evaluates the polynomial with monomial coefficients passed in `p` at all points in `x`, cf. [Lecture → Code 5.2.1.7].

Use `dipoleval_alt()` to verify the correctness of your implementation of `dipoleval()` by computing the derivative of the degree-10 interpolating polynomial for the values $(t_i, \sin(t_i))$, where t_i are equidistant points in $[0, 3]$. This test should be conducted within the function

```
bool testDipolEval(void);
```

Return **false** in case the test fails, **true** otherwise.

SOLUTION of (5-1.f):

C++-code 5.1.7: Implementation of validation function `dipoleval_alt()`

```
2  Eigen::VectorXd dipoleval_alt(const Eigen::VectorXd &t,
3                               const Eigen::VectorXd &y,
4                               const Eigen::VectorXd &x) {
5      assert(t.size() == y.size() && "t and y must have same size!");
6      Eigen::VectorXd ret; // return value
7      const int n = y.size();
8
9      // TODO (5-1.f): Use polyfit() and polyval() to evaluate at x
10     // the derivative of the interpolating polynomial of the data (t,y).
11
12     // START
13     Eigen::VectorXd P = polyfit(t, y, n - 1).head(n - 1);
14
15     for (int i = 0; i < P.size(); ++i) {
16         P(i) *= n - 1 - i;
17     }
18
19     ret = polyval(P, x);
20     // END
21
22     return ret;
23 }
```

Get it on  GitLab (eval_deriv.hpp).

C++-code 5.1.8: Testing framework

```
2  bool testDipolEval() {
3      bool ret = false;
4      Eigen::VectorXd dPx, dPx_alt;
5
6      // TODO (5-1.f): Compare dipoleval() and dipoleval_alt() for the data
7      // given on the problem sheet.
8
9      // START
10     double TOL = 1E-8;
11     const int n = 10; // polynomial degree
12     Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, -1, 4);
13     Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n + 1, 0, 3);
14     Eigen::VectorXd y = sin(t.array()).matrix();
15
16     dPx = dipoleval(t, y, x);
17     dPx_alt = dipoleval_alt(t, y, x);
18
19     double err = (dPx - dPx_alt).norm();
20
21     if (err < TOL) {
```

```

22     ret = true;
23 }
24 // END
25
26 return ret;
27 }

```

Get it on  GitLab (eval_deriv.hpp).



(5-1.g)  (20 min.) Based on your version of `dipoleval()` write a C++ function

```
void plotPolyInterpolant(const std::string &filename);
```

that saves plots the derivative of the polynomial interpolant for the test data as specified in Sub-problem (5-1.f) in the file `<filename>.png`. To that end evaluate the polynomial in 100 equidistant points in the interval $[0,3]$ and, based on the obtained value, use the `plot()` function of `MATPLOTLIBCPP` to draw the polynomials. Do not forget to add axis labels " t " and " $p'(t)$ ", and a title.

HINT 1 for (5-1.g): Axis labels can be added to the graphics object of `MATPLOTLIBCPP` by the functions `matplotlibcpp::xlabel()` and `matplotlibcpp::ylabel()`. 

SOLUTION of (5-1.g):

C++-code 5.1.9: Plotting of polynomial interpolant

```

2  void plotPolyInterpolant(const std::string &filename) {
3      plt::figure();
4      // TODO (5-1.g): Plot the derivative of the polynomial that
5      // interpolates the data from (5-1.f).
6      // Hint: matplotlibcpp (here = plt) implements Python's
7      // matplotlib functions such as figure(), plot(), xlabel(), title(),...
8      // You can use main.cpp of the LinearDataFit problem as a reference.
9
10     // START
11     Eigen::VectorXd dPx;
12     const int n = 10; // polynomial degree
13     Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(100, 0, 3);
14     Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n + 1, 0, 3);
15     Eigen::VectorXd y = sin(t.array()).matrix();
16
17     dPx = dipoleval(t, y, x);
18
19     plt::plot(x, dPx);
20     plt::title("Derivative of interpolating polynomial");
21     plt::xlabel("t");
22     plt::ylabel("p'(t)");
23     // END
24
25     plt::savefig("./cx_out/" + filename + ".png");
26 }

```

Get it on  GitLab (eval_deriv.hpp).

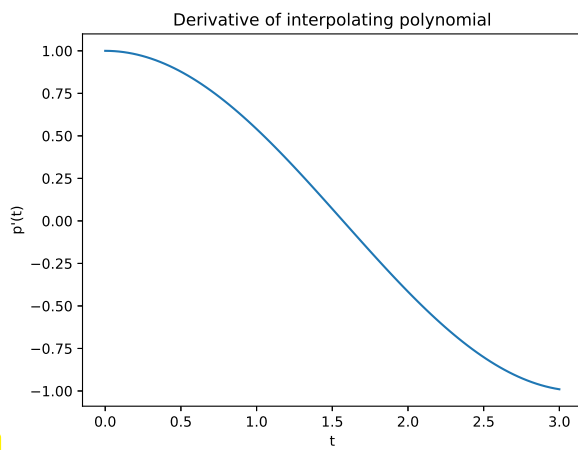


Fig. 31



End Problem 5-1 , 135 min.