

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

Problem 6-3: Adaptive polynomial interpolation

In [Lecture → Section 6.2.3] we have seen that the *a priori* placement of interpolation nodes is key to a good approximation by a polynomial interpolant. This problem deals with an *a posteriori* adaptive strategy that controls the placement of interpolation nodes depending on the interpolant. It employs a **greedy algorithm** to grow the node set based on an intermediate interpolant. This strategy has recently gained prominence for a variety of approximation problems, see [TEM08].

Considerable implementation in C++ is requested in this problem.

A description of the greedy algorithm for adaptive polynomial interpolation is as follows:

Given a function $f : [a, b] \mapsto \mathbb{R}$ one starts $\mathcal{T}_0 := \{\frac{1}{2}(b+a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a, b]$ of *sampling points* one augments the set of nodes according to

$$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \operatorname{argmax}_{t \in \mathcal{S}} |f(t) - I_{\mathcal{T}_n}(t)| \right\}, \quad (6.3.1)$$

where $I_{\mathcal{T}_n}$ is the polynomial interpolation operator (→ [Lecture → Cor. 5.2.2.8]) for the node set $\mathcal{T}_n$, until the relative interpolation error (in maximum norm) drops below a prescribed tolerance $\text{tol} > 0$:

$$\max_{t \in \mathcal{S}} |f(t) - I_{\mathcal{T}_n}(t)| \leq \text{tol} \cdot \max_{t \in \mathcal{S}} |f(t)|. \quad (6.3.2)$$

Throughout this exercise you may rely on a C++ function

```
// IN: t: vector of nodes t_0, ..., t_n
// y: vector of data y_0, ..., y_n
// x: vector of evaluation points x_1, ..., x_N
// OUT: p: interpolant evaluated at x
Eigen::VectorXd intpolyval(const Eigen::VectorXd& t,
                           const Eigen::VectorXd& y,
                           const Eigen::VectorXd& x);
```

that computes the values $p(x_i)$, $i = 1, \dots, N$, where $p \in \mathcal{P}_n$ is the global polynomial interpolant through the data points (t_j, y_j) , $j = 0, \dots, n$, see [Lecture → Code 5.2.3.7].

C++ code 6.3.3: Function `intpolyval()`

```
2 Eigen::VectorXd intpolyval(const Eigen::VectorXd& t, const Eigen::VectorXd& y,
3                           const Eigen::VectorXd& x) {
4     const unsigned int
5         n = t.size(), // no. of interpolation nodes = deg. of polynomial
6         N = x.size(); // no. of evaluation points
7
8     Eigen::VectorXd p = Eigen::VectorXd(N);
9
10    // Precompute the weights  $\lambda_i$  with effort  $O(n^2)$ 
11    Eigen::VectorXd lambda(n);
12    for (unsigned int k = 0; k < n; ++k) {
13        // little workaround: cannot subtract a vector from a scalar
14        // -> multiply scalar by vector of ones
15        lambda(k) =
```

```

16 |         1. /
17 |         ((t(k) * Eigen::VectorXd::Ones(k) - t.head(k)).prod() *
18 |          (t(k) * Eigen::VectorXd::Ones(n - k - 1) - t.tail(n - k - 1)).prod());
19 |     }
20 |     // Compute quotient of weighted sums of  $\frac{\lambda_i}{t-t_i}$ ,
21 |     // effort  $O(n)$ 
22 |     for (unsigned int i = 0; i < N; ++i) {
23 |         Eigen::VectorXd z = (x(i) * Eigen::VectorXd::Ones(n) - t);
24 |
25 |         // check if we want to evaluate at a node <-> avoid division by zero
26 |         double* ptr = std::find(z.data(), z.data() + n, 0.0);
27 |         if (ptr != z.data() + n) { // if ptr = z.data + n = z.end no zero was
28 |             found
                p(i) = y(ptr - z.data()); // ptr - z.data gives the position of the
                zero
29 |         } else {
30 |             Eigen::VectorXd mu = lambda.cwiseQuotient(z);
31 |             p(i) = (mu.cwiseProduct(y)).sum() / mu.sum();
32 |         }
33 |     }
34 |     return p;
35 | }

```

(6-3.a) 🕒 (60 min.) Write a C++ function

```

template <class Function>
Eigen::VectorXd adaptivepolyintp(Function&& f,
                                double a, double b,
                                double tol, int N);

```

that implements the algorithm described above.

The function arguments are: the functor object f , the interval bounds a, b , the relative tolerance tol , the number N of *equidistant* sampling points, that is

$$\mathcal{S} := \{a + (b - a)j/N : j = 0, \dots, N\}, \quad (6.3.4)$$

and t will be used as return parameter for interpolation nodes (i.e. the final set $\mathcal{T}_n$). The type **Function** defines a function and is supposed to provide an operator **double operator()** (**double**) **const**. A suitable lambda function can satisfy this requirement. The function should return the final set of interpolation nodes as proposed by the greedy algorithm.

HINT 1 for (6-3.a): You might need to convert between the types **std::vector** and **Eigen::VectorXd**. This can be done using EIGEN's **Map** class and the method **data()** of **Eigen::VectorXd**. See also 🐘 [EIGEN documentation](#). ┘

SOLUTION of (6-3.a):

See Code 6.3.6 below. For this part of the exercise you can ignore the argument `err_vs_step_no`.



(6-3.b) 🕒 (15 min.) [depends on Sub-problem (6-3.a)]

Extend the function `adaptivepolyintp()` from Sub-problem (6-3.a) to

```

template <class Function>
Eigen::VectorXd
adaptivepolyintp(Function &&f,
    double a, double b,
    double tol, int N,
    std::vector<double> *errortab = nullptr);

```

so that it also reports (and returns in `errortab`) the quantity

$$\epsilon_n := \max_{t \in \mathcal{S}} |f(t) - T_n(t)| \quad (6.3.5)$$

for each intermediate node set $\mathcal{T}_n$.

SOLUTION of (6-3.b):

C++ code 6.3.6: Implementation of function `adaptivepolyintp()`

```

2  template <class Function>
3  Eigen::VectorXd adaptivepolyintp(Function& f, double a, double b, double tol,
4                                     int N,
5                                     std::vector<double>* errortab = nullptr) {
6      // Generate sampling points and evaluate f there
7      Eigen::VectorXd sampling_points = Eigen::VectorXd::LinSpaced(N, a, b),
8          fvals_at_sampling_points = sampling_points.unaryExpr(f);
9
10     // TODO: (6-3.a) Implement the greedy algorithm for adaptive interpolation.
11     // Ignore the errortab part of this function for now.
12     // TODO: (6-3.b) save the error in errortab
13     // START
14     // Approximate max|f(x)|
15     const double maxf = fvals_at_sampling_points.cwiseAbs().maxCoeff();
16
17     // Adaptive mesh (initial node)
18     std::vector<double> t{(a + b) / 2.}, // Set of interpolation nodes
19         y{static_cast<double>(f((a + b) / 2.))}, // Values at nodes
20         errors; // Error at nodes
21
22     for (std::size_t i = 0; i < N; ++i) {
23         // *** Step 1: interpolate with current nodes
24         // need to convert std::vector to
25         // Eigen::VectorXd to use the function intpolyval
26         Eigen::Map<Eigen::VectorXd> te(t.data(), t.size());
27         Eigen::Map<Eigen::VectorXd> ye(y.data(), y.size());
28         Eigen::VectorXd intpolyvals_at_sampling_points =
29             intpolyval(te, ye, sampling_points);
30
31         // *** Step 2: find node where error is the largest
32         Eigen::VectorXd err =
33             (fvals_at_sampling_points - intpolyvals_at_sampling_points).cwiseAbs();
34
35         // We use an Eigen "Visitor"
36         //
37         // https://eigen.tuxfamily.org/dox/group__TutorialReductionsVisitorsBroadcasting.html
38         int idx = 0;
39         const double max = err.maxCoeff(&idx);
40
41         // Step 3: check termination criteria
42         if (max < tol * maxf) {

```

```

42     return te;
43 }
44 // Step 4: add this node to our set of nodes
45 t.push_back(sampling_points(idx));
46 y.push_back(fvals_at_sampling_points(idx));
47 // (7-3.b): save error
48 if (errortab != nullptr) {
49     errortab->push_back(max);
50 }
51 }
52 std::cerr << "Desired accuracy could not be reached." << std::endl;
53 // END
54 return sampling_points; // return all sampling points
55 }

```

Get it on  GitLab (adaptivepolyintp.hpp).



(6-3.c) (30 min.) [depends on Sub-problem (6-3.b)]

For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ write a C++ function

```
void plotInterpolationError() ;
```

that creates (employing `MATPLOTLIBCPP`) and stores (in `intperrplot.png`) a plot of ϵ_n versus n (the number of interpolation nodes). Choose axis scales that reveal the qualitative decay (types of convergence as given in [Lecture → Def. 6.2.2.7]) of this error norm as the number of interpolation nodes is increased. Use interval $[a, b] = [0, 1]$, $N=1000$ sampling points, tolerance `tol = 1e-6`.

SOLUTION of (6-3.c):

C++ code 6.3.7: Function `plotInterpolationError()`

```

2  void plotInterpolationError() {
3      // Declare test functions
4      auto f1 = [](double t) { return std::sin(std::exp(2 * t)); };
5      auto f2 = [](double t) { return std::sqrt(t) / (1 + 16 * t * t); };
6
7      // Test interval
8      constexpr double a = 0, b = 1;
9      // Get interpolation nodes
10     constexpr unsigned int N = 1000; // no. of sampling points
11     constexpr double tol = 1e-6;    // tolerance
12
13     plt::figure();
14
15     // TODO: (6-3.c) generate the plots of error vs number of nodes for f1, f2
16     // START
17     Eigen::VectorXd tf1, tf2; // nodes for f1 resp. f2
18     std::vector<double> ef1, ef2; // errors for f1 resp. f2
19
20     tf1 = adaptivepolyintp(f1, a, b, tol, N, &ef1);
21     tf2 = adaptivepolyintp(f2, a, b, tol, N, &ef2);
22     Eigen::VectorXd n1 = Eigen::VectorXd::LinSpaced(ef1.size(), 1, ef1.size());
23     Eigen::VectorXd n2 = Eigen::VectorXd::LinSpaced(ef2.size(), 1, ef2.size());

```

```

24 // Plot
25 plt::title("Error VS step");
26 plt::xlabel("No. of interpolation nodes");
27 plt::ylabel("max|f(t) - I_T f(t)|");
28 plt::semilogy(n1, ef1, "ro", {"label", "f_1(t) = sin(e^{2t})"});
29 plt::semilogy(n2, ef2, "bo", {"label", "f_2(t) = sqrt(t)/(1 + 16t^2)"});
30 plt::legend();
31 // END
32
33 plt::savefig("./cx_out/intperrplot.png");
34 }

```

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

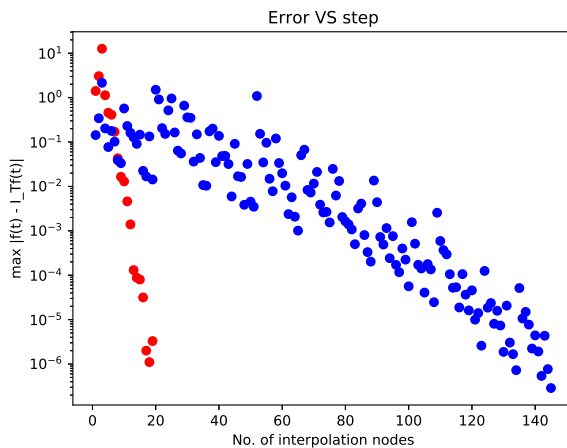


Fig. 56

$$\bullet \hat{=} f_1(t) := \sin(e^{2t})$$

$$\bullet \hat{=} f_2(t) = \frac{\sqrt{t}}{1+16t^2}$$

The error data points are approximately located on straight lines in the linear-logarithmic error plots.



Exponential convergence



End Problem 6-3 , 105 min.