

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 4-3: Multiplication and division of polynomials based on DFT

In [Lecture → § 4.1.3.6] we saw that the formula of discrete convolution [Lecture → Def. 4.1.3.3] also gives the coefficients of products of polynomials. Thus, in light of the realization of discrete convolutions by means of DFT [Lecture → Section 4.2.2], the Fast Fourier Transform (FFT) becomes relevant for efficiently multiplying polynomials of very high degree.

This problem requires knowledge about DFT and discrete convolutions, see [Lecture → Section 4.1.3] and [Lecture → Section 4.2.2]. For the connection of DFT, FFT, and polynomial multiplication also see the [YouTube Video](#).

Polynomials. We call a **polynomial** in x of **degree** $N \in \mathbb{N}$ a function $p : \mathbb{K} \rightarrow \mathbb{K}$ of the form

$$p(x) = \sum_{j=0}^N \gamma_j x^j \quad \text{with coefficients } \gamma_j \in \mathbb{K}, \quad (4.3.1)$$

also called **monomial representation**. Under pointwise addition and multiplication the polynomials of degree N form a vector space $\mathcal{P}_N$ of dimension $N+1$. Associating the vector $\mathbf{c} := [\gamma_0, \dots, \gamma_N] \in \mathbb{K}^{N+1}$ to every $p \in \mathcal{P}_N$ as given in (4.3.1) induces an isomorphism of the vector spaces $\mathcal{P}_N$ and $\mathbb{K}^{N+1}$. In other words, we can identify $p \in \mathcal{P}_N$ with its coefficient vector $\mathbf{c}$.

Thus, when we have to represent polynomials $\in \mathcal{P}_N$ in a C++ code we can do this via objects of type **Eigen::VectorXd** that contain their coefficient vectors. This convention is adopted throughout this problem.

Let two large numbers $m, n \gg 1$ and two polynomials of degrees $m-1$ and $n-1$, respectively, in monomial representation

$$u(x) = \sum_{j=0}^{m-1} \alpha_j x^j, \quad v(x) = \sum_{j=0}^{n-1} \beta_j x^j, \quad \alpha_j, \beta_j \in \mathbb{C}, \quad (4.3.2)$$

be given.

The tasks that we are going to tackle in this problem are the following.

1. **Polynomial multiplication:** Efficiently compute the coefficients of the polynomial $p = uv$ (point wise product, a polynomial of degree $\leq m+n-2$).
2. **Polynomial division:** Given the polynomials p and u in terms of their coefficients, the degree of p at least as big as that of u , find a polynomial v , if it exists, such that $p = uv$.

(4-3.a)  (25 min.) Write a C++ function

```
Eigen::VectorXd polyMult_naive(const Eigen::VectorXd & u, const
                               Eigen::VectorXd & v);
```

that “naively”, i.e. using a simple loop-based implementation, computes and returns the vector of coefficients of the polynomial $p = uv$. The coefficient vectors for the polynomials u and v are passed in u and v .

Also determine the asymptotic complexity of your algorithm for $m, n \rightarrow \infty$ (separately).

HINT 1 for (4-3.a): Consult [Lecture → § 4.1.3.6].

SOLUTION of (4-3.a):

The following code provides a straightforward double-loop implementation of the convolution of two vectors.

C++ code 4.3.3: Solution of (4-3.a)

```

2  Eigen::VectorXd polyMult_naive(const Eigen::VectorXd &u,
3                                const Eigen::VectorXd &v) {
4      // Fetch degrees of input polynomials
5      int degu = u.size() - 1;
6      int degv = v.size() - 1;
7
8      // Object for product polynomial p = uv
9      int degp = degu + degv;
10
11     Eigen::VectorXd uv(degp + 1);
12
13     // TODO: (4-3.a) Multiply polynomials u and v naively.
14     // START
15     for (int i = 0; i <= degp; ++i) {
16         int fst = std::max(0, i - degv);
17         int lst = std::min(degu, i);
18         for (int j = fst; j <= lst; ++j) {
19             uv(i) += u(j) * v(i - j);
20         }
21     }
22     // END
23
24     return uv;
25 }

```

Get it on  GitLab (polydiv.hpp).

Since a double loop is used, the complexity is $\mathcal{O}(nm)$.



(4-3.b) (30 min.) Write a C++ function

```

Eigen::VectorXd polyMult_fast(const Eigen::VectorXd &u, const
Eigen::VectorXd &v);

```

that *efficiently* computes the vector of monomial coefficients of the product polynomial $p = uv$ of the polynomials u and v , passed through their respective coefficient vectors. To that end use the DFT functionality of EIGEN, see [Lecture → Code 4.2.1.22].

HINT 1 for (4-3.b): As elaborated in [Lecture → § 4.1.3.6], the coefficient vector of $p = uv$ is the discrete convolution [Lecture → Def. 4.1.3.3] of the coefficient vectors of u and v . [Lecture → Code 4.2.2.5] demonstrates the efficient FFT-based discrete convolution of two vectors of the *same size*.



SOLUTION of (4-3.b):

C++11-code 4.3.4: Solution of (4-3.b)

```

2  Eigen::VectorXd polyMult_fast(const Eigen::VectorXd &u,

```

```

3      const Eigen::VectorXd &v) {
4      // Initialization
5      int m = u.size();
6      int n = v.size();
7
8      Eigen::VectorXd u_tmp(m + n - 1);
9      u_tmp.head(m) = u;
10     u_tmp.tail(n - 1).setZero();
11
12     Eigen::VectorXd v_tmp(m + n - 1);
13     v_tmp.head(n) = v;
14     v_tmp.tail(m - 1).setZero();
15
16     Eigen::VectorXd uv;
17
18     // TODO: (4-3.b) Multiply polynomials u and v efficiently.
19     // START
20     Eigen::FFT<double> fft;
21     Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);
22     Eigen::VectorXcd v_tmp_ = fft.fwd(v_tmp);
23     Eigen::VectorXcd tmp = u_tmp_.cwiseProduct(v_tmp_);
24     uv = fft.inv(tmp).real();
25     // END
26
27     return uv;
28 }

```

Get it on  GitLab (polydiv.hpp).

Compare this code with [Lecture → Code 4.2.2.5], which implements the discrete convolution of two vectors. The EIGEN method `conservativeResizeLike()` is used for **zero padding** here.

(4-3.c)  Determine the complexity of your implementation of algorithm in (4-3.b) making use of the fact that EIGEN's DFT relies on the FFT algorithm [Lecture → Section 4.3] and thus can be carried out with an asymptotic effort $O(n \log n)$ for vector length $n \rightarrow \infty$.

HINT 1 for (4-3.c): The asymptotic complexity of `polyMult_fast` is clearly inherited by that of a DFT.

SOLUTION of (4-3.c):

Assuming `Eigen::VectorXd u` and `Eigen::VectorXd v` represent polynomials of the degrees m and n , respectively, the complexity of the algorithm is $O((n + m) \log(n + m))$.

The greater efficiency of the FFT-based implementation is noticeable only $m, n \geq 10^4$. The runtimes reported in the plot beside were measured for $m = n$ and using EIGEN's built-in FFT implementation.

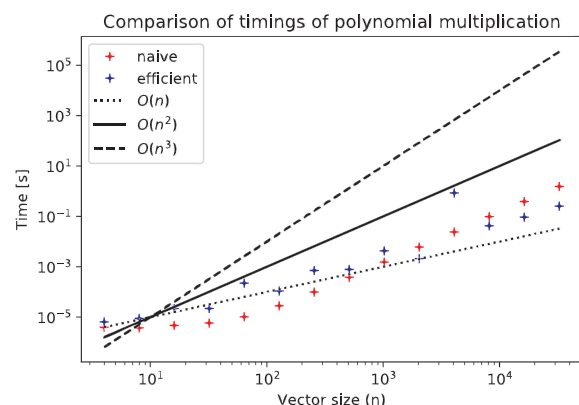


Fig. 27



(4-3.d) (20 min.) Give an interpretation of the entries of the vector $\text{DFT}_n \mathbf{u}$ obtained by applying a discrete Fourier transform to the coefficient vector $\mathbf{u}$ of a polynomial u of degree $n - 1$? What is an equivalent operation that can be performed for the polynomial u which leads to the same result?

HINT 1 for (4-3.d): When computing a DFT of $\mathbf{u} \in \mathbb{C}^n$, $\mathbf{c} := \text{DFT}_n \mathbf{u} = \mathbf{F}_n \mathbf{u}$, we evaluate the following expressions, (see [Lecture → Def. 4.2.1.18]):

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj}, \quad k = 0, \dots, n-1, \quad (4.3.5)$$

where ω_n is the n -th root of unity $\exp(-\frac{2\pi i}{n})$. For which arguments z do you have to evaluate the polynomial

$$p(z) := \sum_{j=0}^{n-1} (\mathbf{u})_j z^j, \quad z \in \mathbb{C},$$

to obtain the complex numbers c_j ? ┘

SOLUTION of (4-3.d):

(4.3.5) implies that each entry of the DFT vector $\mathbf{c} = \mathbf{F}_n \mathbf{u}$ is the evaluation of the polynomial with coefficients $(\mathbf{u})_j$ in the roots of unity $\omega_n^k = \exp(-\frac{2\pi i k}{n})$. Thus the DFT provides an alternative “root-of-unity representation” of a polynomial through “equidistant samples on the unit circle”. Notice that no information is lost when switching between these representation, as a polynomial with degree $n - 1$ is uniquely determined by its values in n points.

Remark. Hence, the polynomial multiplication carried out via DFT in (4-3.b) means that you first transform the polynomials u and v from the “coefficient” representation to a “unit root evaluation” representation. In this representation, each entry k of n -dimensional vectors $\mathcal{F}(\mathbf{u})$ and $\mathcal{F}(\mathbf{v})$ is the evaluation of polynomials u and v in one root of unity ω_n^k . In the unit root representation, multiplication of polynomials is straightforward. Indeed, the component-wise product of the value vectors is the root-of-unity representation of the product polynomial uv . The inverse Fourier transform in the convolution theorem used in (4-3.b) finally transforms back the product polynomial in the coefficient representation, thus solving the polynomial multiplication problem.



The following sub-problems deal with polynomial division. Polynomial division with remainder can be carried out using Euclid’s algorithm and is captured by the following theorem:

Theorem 4.3.6. Polynomial division with remainder

For every $p \in \mathcal{P}_N$ and $u \in \mathcal{P}_m$, $m \leq N$, there exist unique polynomials $q \in \mathcal{P}_{N-m}$ and $r \in \mathcal{P}_{m-1}$ such that

$$p = uq + r.$$

Using the notations of the theorem, we are interested in determining, whether the polynomial p can be divided by u exactly, that is, whether $r \equiv 0$. In this case we also want to compute the quotient polynomial q .

(4-3.e) 🕒 (45 min.) [depends on Sub-problem (4-3.b)]

Write a C++ function

```
Eigen::VectorXd polyDiv(const Eigen::VectorXd & p, const
Eigen::VectorXd & u);
```

that takes the coefficient vectors p and u of two polynomials p and u , respectively, as arguments, where the degree of p must be at least as large as that of u . The function is to return the coefficient vector of a q , such that $p = uv$, if it exists. If not, a vector of length zero should be returned.

HINT 1 for (4-3.e): If $p \in \mathcal{P}_N$, $u \in \mathcal{P}_m$, $p = uv$ for some $v \in \mathcal{P}_n$, $n := N - m$, then, according to [Lecture → § 4.1.3.6] and the considerations in [Lecture → Section 4.2.2],

$$p = \hat{u} *_{N+1} \hat{v}, \quad (4.3.7)$$

where $p \in \mathbb{C}^{N+1}$ is the coefficient vector for p , and $\hat{v}, \hat{u} \in \mathbb{C}^{N+1}$ are the coefficient vectors of u and v , extended to length $N + 1$ by **zero padding**. As in [Lecture → Section 4.1.4], $*_{N+1}$ denotes the **discrete periodic convolution** of two vectors of length $N + 1$. ┘

HINT 2 for (4-3.e): Tests whether results of floating-point computations vanish have to be implemented in a numerically safe way according to [Lecture → Rem. 1.5.3.15] ┘

SOLUTION of (4-3.e):

By the **convolution theorem** [Lecture → Thm. 4.2.2.2], (4.3.7) implies

$$\text{DFT}_{N+1} p = \text{DFT}_{N+1}(\hat{u}) \odot \text{DFT}_{N+1}(\hat{v}),$$

where $\odot$ stands for componentwise multiplication of vectors. Hence, formally we get

$$\hat{v} = \text{DFT}_{N+1}^{-1}(\text{DFT}_{N+1}(p) \oslash \text{DFT}_{N+1}(\hat{u})), \quad (4.3.8)$$

with componentwise division written as $\oslash$.

However, two complications arise and those have to be taken care of in a code.

- (i) If $(\text{DFT}_{N+1} \hat{u})_j = 0$ for some $j = 0, \dots, N$ (C++ indexing), the componentwise division will not be possible.
- (ii) The result vector $\hat{v}$ may not be a valid coefficient vector for a polynomial $v \in \mathcal{P}_n$ (notations as in the first hint), because $(\hat{v})_j = 0$ is not guaranteed for $j > n$.

To deal with (i), we have to check, whether $(\text{DFT}_{N+1} \hat{u})_j = 0$ for some $j = 0, \dots, N$. If also $(\text{DFT}_{N+1} p)_j = 0$, then set the result of the division to zero, otherwise u does not divide p . Complication (ii) can be detected by a simple test: If any $(\hat{v})_j$, $j > n$, does not vanish we conclude that u does not divide p .

C++ code 4.3.9: Solution of (4-3.e)

```
2 Eigen::VectorXd polyDiv(const Eigen::VectorXd &uv, const Eigen::VectorXd &u) {
3     // Initialization
4     int mn = uv.size();
5     int m = u.size();
6     // need well behaved input
7     if (mn < m) {
8         std::cerr << "uv can't be divided by u\n";
9         return Eigen::VectorXd(0);
10    }
```

```

10 }
11 int dim = mn;
12 int n = mn - m + 1;
13
14 Eigen::VectorXd v;
15
16 // TODO: (4-3.e) Divide polynomials uv and u efficiently.
17 // START
18
19 // zero padding
20 Eigen::VectorXd uv_tmp = uv;
21 uv_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
22 Eigen::VectorXd u_tmp = u;
23 u_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
24
25 Eigen::FFT<double> fft;
26 Eigen::VectorXcd uv_tmp_ = fft.fwd(uv_tmp);
27 Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);
28
29 // check divisibility: case(i)
30 for (int i = 0; i < dim; ++i) {
31     if (abs(uv_tmp_(i)) < 1e-13) {
32         if (abs(u_tmp_(i)) < 1e-13) {
33             // make cwiseQuotient at i-th position equal 0
34             uv_tmp_(i) = 0.0; // complex assignment (0., 0.)
35             u_tmp_(i) = 1.0; // complex assignment (1., 0.)
36         } else {
37             std::cerr << "uv can't be divided by u\n";
38             return Eigen::VectorXd(0);
39         }
40     }
41 }
42
43 Eigen::VectorXcd tmp = uv_tmp_.cwiseQuotient(u_tmp_);
44
45 v = fft.inv(tmp).real();
46 // check divisibility: case(ii)
47 for (int i = n; i < dim; ++i) {
48     if (abs(v(i)) > 1e-13) {
49         std::cerr << "uv can't be divided by u\n";
50         return Eigen::VectorXd(0);
51     }
52 }
53
54 // reshape v to a suitable size
55 v.conservativeResizeLike(Eigen::VectorXd::Zero(n));
56 // (mn-1) - (m-1) + 1
57 // END
58
59 return v;
60 }

```

Get it on  GitLab (polydiv.hpp).



End Problem 4-3, 120 min.