

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 4-5: Implementing Discrete Convolution

[Lecture → Section 4.2.2] discussed in detail the implementation of discrete convolutions of signals stored in vectors by means of the discrete Fourier transform (DFT). This problem revisits these techniques and adds a new twist to implementation.

This problem assumes solid knowledge of [Lecture → Section 4.2.2].

From [Lecture → Section 4.1.3] we recall the definitions

Definition [Lecture → Def. 4.1.3.3]. Discrete convolution

Given $\mathbf{x} = [x_0, \dots, x_{m-1}]^\top \in \mathbb{K}^m$, $\mathbf{h} = [h_0, \dots, h_{n-1}]^\top \in \mathbb{K}^n$ their **discrete convolution** (DCONV) $\mathbf{h} * \mathbf{x}$ is the vector $\mathbf{y} = [y_0, \dots, y_{m+n-2}]^\top \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j, \quad k = 0, \dots, m+n-2, \quad [\text{Lecture} \rightarrow \text{Eq. (4.1.3.4)}]$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

and

Definition cf. [Lecture → Def. 4.1.4.7]. Discrete periodic convolution (of vectors)

The **discrete periodic convolution** $\mathbf{p} *_n \mathbf{x}$ of two vectors $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$ is the vector $\mathbf{y} \in \mathbb{K}^n$ with components (C++ indexing)

$$(\mathbf{y})_k := \sum_{j=0}^{n-1} (\mathbf{p})_{(k-j) \bmod n} (\mathbf{x})_j, \quad k \in \{0, \dots, n-1\}.$$

$((k-j) \bmod n \in \{0, \dots, n-1\} \triangleq (k-j) \% n$ in C++ syntax.)

(4-5.a) ☞ (15 min.) For given $\mathbf{h} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^m$, specify the vectors $\tilde{\mathbf{h}}, \tilde{\mathbf{x}}$, the *minimal* vector length $N \in \mathbb{N}$, and the missing index range $?:?$ in the formula

$$\mathbf{h} * \mathbf{x} = \left(\tilde{\mathbf{h}} *_N \tilde{\mathbf{x}} \right)_{?:?}.$$

C++ indexing has to be used and the notational convention is $(\mathbf{v})_{r:s} = [(\mathbf{v})_r, \dots, (\mathbf{v})_s] \in \mathbb{K}^{s-r+1}$ for $r, s \in \mathbb{N}$, $r \leq s$.

SOLUTION of (4-5.a):

We can rewrite the discrete convolution of two vectors [Lecture → Eq. (4.1.3.4)] as (C++ indexing used)

$$(\mathbf{h} * \mathbf{x})_k = \sum_{j=\max\{0, k-n+1\}}^{\min\{m-1, k\}} (\mathbf{x})_j (\mathbf{h})_{k-j}, \quad k = 0, \dots, m+n-2. \quad (4.5.1)$$

We take the cue from [Lecture → Rem. 4.1.4.15] and borrow the crucial trick of **zero padding**. For $N \geq \max\{m, n\}$ we introduce the **zero-padded** vectors

$$\tilde{\mathbf{x}} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N, \quad \tilde{\mathbf{h}} := \begin{bmatrix} \mathbf{h} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N.$$

Next we closely examine the periodic convolution of these two vectors, exploiting our knowledge about zero components of $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{h}}$:

$$\begin{aligned} (\tilde{\mathbf{h}} *_N \tilde{\mathbf{x}})_k &= \sum_{j=0}^{N-1} (\tilde{\mathbf{x}})_j (\tilde{\mathbf{h}})_{(k-j) \bmod N} = \sum_{j=0}^{m-1} (\mathbf{x})_j (\tilde{\mathbf{h}})_{(k-j) \bmod N} \\ &= \sum_{j=0}^{\min\{m-1, k\}} (\mathbf{x})_j (\tilde{\mathbf{h}})_{k-j} + \sum_{j=\min\{m, k+1\}}^{m-1} (\mathbf{x})_j (\tilde{\mathbf{h}})_{k+N-j}. \end{aligned}$$

❶ Case $k \in \{m-1, \dots, N-1\}$: In this case the second sum becomes void and we find

$$(\tilde{\mathbf{h}} *_N \tilde{\mathbf{x}})_k = \sum_{j=0}^{m-1} (\mathbf{x})_j (\tilde{\mathbf{h}})_{k-j} = \sum_{j=\max\{0, k-n+1\}}^{\min\{m-1, k\}} (\mathbf{x})_j (\mathbf{h})_{k-j} = (\mathbf{h} * \mathbf{x})_k,$$

that is, for these vector components both discrete convolutions produce the same result for any $N \geq \max\{m, n\}$.

❷ Case $k \in \{0, \dots, m-2\}$: The corresponding components of the discrete periodic convolution are

$$\begin{aligned} (\tilde{\mathbf{h}} *_N \tilde{\mathbf{x}})_k &= \sum_{j=0}^k (\mathbf{x})_j (\tilde{\mathbf{h}})_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j (\tilde{\mathbf{h}})_{k+N-j} \\ &= \sum_{j=\max\{0, k-n+1\}}^k (\mathbf{x})_j (\mathbf{h})_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j (\tilde{\mathbf{h}})_{k+N-j}. \end{aligned}$$

Clearly, the second sum should evaluate to zero if both discrete convolutions are to yield the same vector components. Thus, we have to demand

$$\begin{aligned} &(\tilde{\mathbf{h}})_{k+N-j} = 0 \quad \forall j \in \{k+1, \dots, m-1\}, \quad \forall k \in \{0, \dots, m-2\} \\ \Leftrightarrow &k+N-j \geq n \quad \forall j \in \{k+1, \dots, m-1\}, \quad \forall k \in \{0, \dots, m-2\} \\ \Leftrightarrow &N - (m-1) \geq n \\ \Leftrightarrow &N \geq n + m - 1. \end{aligned}$$

Hence, $N \geq n + m - 1$ is a necessary and sufficient condition for the agreement of the vector components with indices $k \in \{0, \dots, m-2\}$.

Summing up we have found $N = n + m - 1$ and

$$\mathbf{h} * \mathbf{x} = \left(\tilde{\mathbf{h}} *_{m+n-1} \tilde{\mathbf{x}} \right)_{0:n+m-2}.$$

In fact, the range specification for the vector components could be dropped, because for $N = n + m - 1$ all components of the vector produced by the discrete periodic convolution $*_{m+n-1}$ are relevant.

The following code demonstrates the implementation of this idea:

C++ code 4.5.2: Discrete convolution by periodic convolution

```
2 Eigen::VectorXd dconv_p(const Eigen::VectorXd &h, const Eigen::VectorXd &x) {
3   const int n = h.size(); // length of vector h
4   const int m = x.size(); // length of vector x
5   const int N = m + n - 1; // Minimal length of periodic convolution
6   // Zero-padded vectors of length m+n-1
```

```

7 | Eigen::VectorXd ht = Eigen::VectorXd::Zero(N);
8 | Eigen::VectorXd xt = Eigen::VectorXd::Zero(N);
9 | ht.head(n) = h;
10 | xt.head(m) = x;
11 | // Discrete periodic convolution, see [Lecture → Code 4.2.2.1]
12 | return pconv(ht, xt);
13 | }

```

Get it on  GitLab (discreteconvolution.hpp).

A basic loop-based implementation of `pconv()` is shown in [Lecture → Code 4.2.2.1].



The discrete Fourier transform in EIGEN relies on the library `KissFFT`, which offers a very efficient implementation of DFT with asymptotic computational cost $O(n \log n)$ for vectors of length $n := 2^\ell$, $\ell \in \mathbb{N}$, but may not be fast for vectors of other lengths. To avoid degraded performance of a numerical code, the following rule is imposed throughout this problem:

EIGEN's built-in DFT implementations may be invoked only for vectors of length 2^ℓ for some $\ell \in \mathbb{N}$!

(4-5.b)  (40 min.)

Based on EIGEN, in the file `discreteconvolution.hpp` implement an *efficient* function

```

Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p,
                           const Eigen::VectorXd &x);

```

that computes the **discrete periodic convolution** $\mathbf{p} *_n \mathbf{x}$ as defined in [Lecture → Def. 4.1.4.7] of two real vectors $\mathbf{p}, \mathbf{x} \in \mathbb{R}^n$ of the same length.

HINT 1 for (4-5.b): Please note that EIGEN's FFT implementation can only deal with vectors with complex-valued entries, that is, the type `Eigen::VectorXcd`. In EIGEN you have to use the method `cast<std::complex<double>>()` when using both real-valued and complex-valued vectors in an expression.

SOLUTION of (4-5.b):

We elaborate how to realize a general periodic discrete convolution $\mathbf{p} *_n \mathbf{x}$, $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$, $n \in \mathbb{N}$, by means of a discrete periodic convolution of vectors of length $N > n$. We assume $N \geq 2n$!

We first rewrite the formula for the discrete periodic convolution of $\mathbf{p} = [p_0, \dots, p_{n-1}]^\top \in \mathbb{K}^n$, $\mathbf{x} = [x_0, \dots, x_{n-1}]^\top \in \mathbb{K}^n$ from [Lecture → Def. 4.1.4.7], see also [Lecture → Code 4.2.2.1]:

$$(\mathbf{p} *_n \mathbf{x})_k = \sum_{j=0}^{n-1} x_j p_{(k-j) \bmod n} = \sum_{j=0}^k x_j p_{k-j} + \sum_{j=k+1}^{n-1} x_j p_{n+k-j}, \quad k = 0, \dots, n-1. \quad (4.5.3)$$

Then we introduce auxiliary vectors by **zero padding** and **periodic padding**, respectively:

$$\tilde{\mathbf{x}} = \begin{bmatrix} \tilde{x}_0 \\ \vdots \\ \tilde{x}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N, \quad \tilde{\mathbf{p}} = \begin{bmatrix} \tilde{p}_0 \\ \vdots \\ \tilde{p}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{p} \\ \mathbf{0} \\ \mathbf{p} \end{bmatrix} \in \mathbb{K}^N.$$

Written in components this amounts to

$$(\tilde{\mathbf{x}})_j := \begin{cases} x_j & \text{for } j = 0, \dots, n-1, \\ 0 & \text{for } j = n, \dots, N-1 \end{cases}, \quad (\tilde{\mathbf{p}})_j := \begin{cases} p_j & \text{for } j = 0, \dots, n-1, \\ 0 & \text{for } j = n+1, \dots, N-n-1, \\ p_{j+(n-N)} & \text{for } j = n-N, \dots, N-1. \end{cases}$$

In particular, this means

$$\tilde{p}_j = p_j \quad \text{for } j = 0, \dots, n-1, \quad \tilde{p}_{N-\ell} = p_{n-\ell} \quad \text{for } \ell = 1, \dots, n. \quad (4.5.4)$$

Based on (4.5.3), we examine the size- N discrete periodic convolution of $\tilde{\mathbf{p}}$ and $\tilde{\mathbf{x}}$:

$$(\tilde{\mathbf{p}} *_N \tilde{\mathbf{x}})_k = \sum_{j=0}^k \tilde{x}_j \tilde{p}_{k-j} + \sum_{k=k+1}^{N-1} \tilde{x}_j \tilde{p}_{N+k-j}.$$

First we take into account that $\tilde{x}_j = 0$ for $j \geq n$:

$$\blacktriangleright \quad (\tilde{\mathbf{p}} *_N \tilde{\mathbf{x}})_k = \sum_{j=0}^k x_j \tilde{p}_{k-j} + \sum_{j=k+1}^{n-1} x_j \tilde{p}_{N+k-j}. \quad (4.5.5)$$

Sums whose lower bound is larger than the upper bound are meant to be $= 0$.

Now we restrict ourselves to $k \in \{0, \dots, n-1\}$, which implies

- that in the first sum of (4.5.5) that $k-j \in \{0, \dots, n-1\}$,
- and that $j-k \in \{1, \dots, n-1\}$ in the second sum of (4.5.5), which means $\tilde{p}_{N+k-j} = p_{n+k-j}$ thanks to (4.5.4).

$$\blacktriangleright \quad (\tilde{\mathbf{p}} *_N \tilde{\mathbf{x}})_k = \sum_{j=0}^k x_j p_{k-j} + \sum_{k=k+1}^{n-1} x_j p_{n+k-j} = (\mathbf{p} *_n \mathbf{x})_k, \quad k \in \{0, \dots, n-1\}.$$

This is exploited in the following C++ function, to which the parameter N passes N :

C++ code 4.5.6: Resizing periodic convolution

```

2  Eigen::VectorX<T> pconvN(const Eigen::VectorX<T> &p, const Eigen::VectorX<T> &x,
3                          unsigned int N) {
4      const int n = x.size();
5      assert(n == p.size());
6      // Requirement on N to avoid overlap in periodic padding
7      assert(N >= 2 * n);
8      // Periodic padding of vector p
9      Eigen::VectorX<T> pt{Eigen::VectorX<T>::Zero(N)};
10     pt.head(n) = p;
11     pt.tail(n) = p;
12     // Zero padding of vector x
13     Eigen::VectorX<T> xt{Eigen::VectorX<T>::Zero(N)};
14     xt.head(n) = x;
15     // Periodic convolution of length N
16     const Eigen::VectorX<T> y = pconv(pt, xt);
17     return y.head(n);
18 }

```

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

Of course, any efficient implementation must be based on DFTs by the FFT algorithm. Thus, we have to replace the call to `pconv()` in Code 4.5.6 by the DFT-based implementation given in [Lecture → Code 4.2.2.1]. We have to comply with the requirement that only DFTs of length $N = 2^\ell$, $\ell \in \mathbb{N}$ are admissible, and also meet the condition $N \geq 2n$. Thus, N has to be chosen as the smallest power of 2 which is $\geq 2n$. The following code chooses that N and merges Code 4.5.6 and [Lecture → Code 4.2.2.1].

C++ code 4.5.7: Fast discrete periodic convolution implemented by FFT-based DFT of length $N = 2^\ell$

```

2  Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p, const Eigen::VectorXd &x) {
3      const int n = x.size();
4      assert(n == p.size());
5      Eigen::VectorXd z = Eigen::VectorXd::Zero(n);
6
7      // TO DO: Compute the discrete periodic convolution of p with x
8      // in an efficient manner.
9      // START
10
11     // N >= 2*n is required to avoid overlap in periodic padding
12     int N = 1;
13     while (N < 2 * n) {
14         N *= 2;
15     }
16     // Note that Eigen's FFT routines expect complex vectors
17     // Periodic padding of vector p
18     Eigen::VectorXcd pt{Eigen::VectorXcd::Zero(N)};
19     pt.head(n) = p.template cast<std::complex<double>>();
20     pt.tail(n) = p.template cast<std::complex<double>>();
21     // Zero padding of vector x
22     Eigen::VectorXcd xt{Eigen::VectorXcd::Zero(N)};
23     xt.head(n) = x.template cast<std::complex<double>>();
24     // Periodic convolution of length N = 2^l realized by FFT
25     Eigen::FFT<double> fft;
26     z = (fft.inv(((fft.fwd(pt)).cwiseProduct(fft.fwd(xt))).eval()))
27         .real()
28         .head(n);
29
30     // END
31     return z;
32 }

```

Get it on  GitLab (discreteconvolution.hpp).

Note that this choice of N is important for efficiency, because choosing a larger power of 2 will incur significant extra work for large n .

Remark. An even more efficient implementation could also exploit the fact that p, x are *real* vectors, see [Lecture → Section 4.2.4]. However, this further optimization is not expected here.



End Problem 4-5 , 55 min.