

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.

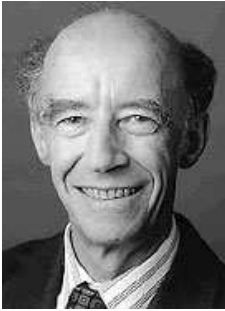
We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

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 7-12: Clenshaw-Curtis-Fejer Quadrature Formula

In [J. WALDVOGEL, *Fast construction of the Fejér and Clenshaw-Curtis quadrature rules*, BIT, 46 (2006), pp. 195–202.] Jörg Waldvogel, former professor at the Seminar of Applied Mathematics of ETH Zürich, elaborates formulas for the weights and nodes of so-called Clenshaw-Curtis-Fejer quadrature rules, cf. [Lecture → Rem. 7.3.0.8]. This problem is concerned with their efficient implementation.

This problem requires familiarity with the discrete Fourier transform (DFT) from [Lecture → Section 4.2] and the FFT algorithm [Lecture → Section 4.3] and its use in EIGEN. Knowledge about the concept and the order of quadrature formulas is taken for granted.

The following class provides the weights and nodes of a quadrature formula of the Clenshaw-Curtis-Fejer family on the interval $[-1, 1]$.

C++ code 7.12.1: Definition of quadrature class

```

2  class CCFQuadRule {
3  public:
4      explicit CCFQuadRule(unsigned int n);
5      ~CCFQuadRule() = default;
6
7      const Eigen::VectorXd &nodes() const { return nodes_; }
8      const Eigen::VectorXd &weights() const { return weights_; }
9
10 private:
11     Eigen::VectorXd nodes_;
12     Eigen::VectorXd weights_;
13 };

```

Get it on  GitLab (clenshawcurtisfejer.hpp).



Note that passing an integer n to the constructor builds an $n + 1$ -point quadrature formula!

The methods `nodes()` and `weights()` return vectors whose entries supply the nodes c_j and weights w_j , $j = 0, \dots, n$, of the quadrature formula (C++ indexing used!).

This is the listing of the implementation of the constructor:

C++ code 7.12.2: Constructor of CCFQuadRule

```

2  CCFQuadRule::CCFQuadRule(unsigned int n) : weights_(n + 1) {
3      assert(n > 0);
4      nodes_ = (Eigen::ArrayXd::LinSpaced(n + 1, 0, n) * M_PI / n).cos().matrix();
5      int m = n / 2; // Integer division!
6      const Eigen::ArrayXd idx = Eigen::ArrayXd::LinSpaced(m, 1.0, m);
7      const Eigen::ArrayXd cos_arg = 2.0 * M_PI * idx / n;
8      Eigen::ArrayXd fac = 2.0 / (4 * idx.pow(2) - 1.0);
9      if (n % 2 == 0)
10         fac[m - 1] /= 2.0;
11     weights_[0] = (1.0 - fac.sum()) / n;
12     for (int j = 1; j < n; ++j) {
13         weights_[j] = (1.0 - (fac * (j * cos_arg).cos()).sum()) * 2.0 / n;

```

```

14 | }
15 | weights_[n] = weights_[0];
16 | }

```

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

Recall that

- `Eigen::ArrayXd::LinSpaced(n, a, b)` builds a sequence of n values $\left(a + \frac{b-a}{n-1}j\right)_{j=0}^{n-1}$,
- the `cos()` method of `Eigen::ArrayXd` applies the cosine function to all entries of the sequence, and
- the call of `pow(2)` for `Eigen::ArrayXd` squares all members of the sequence,
- that the `sum()` method computes the sum of all entries of a vector/matrix in EIGEN.

(7-12.a)  (10 min.) Based on Code 7.12.2 give a mathematical expression for the nodes c_j , $j = 0, \dots, n$, of the $n + 1$ -point Clenshaw-Curtis-Fejer quadrature formula.

SOLUTION of (7-12.a):

From Line 4 of Code 7.12.2 we read off that the $n + 1$ quadrature nodes are

$$c_j = \cos\left(j\frac{\pi}{n}\right), \quad j = 0, \dots, n.$$

Remark. The nodes are the sites of the extrema of the Chebyshev polynomial of degree n .



(7-12.b)  (15 min.) Based on Code 7.12.2 write down (in mathematical notation) a formula for the quadrature weights w_j , $j = 0, \dots, n$, of the $n + 1$ -point Clenshaw-Curtis-Fejer quadrature formula implemented in the class `CCFQuadRule`.

SOLUTION of (7-12.b):

It is important to realize that $m = n/2$ is an integer division: $m = \lfloor \frac{n}{2} \rfloor$.

According to [WAV06] we have

$$w_j = \frac{a_j}{n} \left(1 - \sum_{\ell=1}^m \frac{b_\ell}{4\ell^2 - 1} \cos\left(2\ell j \frac{\pi}{n}\right) \right), \quad j = 0, \dots, n, \quad (7.12.3)$$

where

$$\begin{aligned}
 m &:= \begin{cases} n/2 & \text{for even } n, \\ n-1/2 & \text{for odd } n, \end{cases} \\
 a_j &:= \begin{cases} 1 & \text{for } j = 0, n, \\ 2 & \text{else,} \end{cases} \quad j = 0, \dots, n, \\
 b_\ell &:= \begin{cases} 1 & \text{for } \ell = \frac{n}{2}, \\ 2 & \text{else,} \end{cases} \quad \ell = 1, \dots, m.
 \end{aligned}$$

A collapsed sum is supposed to evaluate to zero. There are several ways to write this formula, for instance, one could separately state

$$w_0 = w_n = \frac{1}{n} \left(1 - \sum_{\ell=1}^m \frac{b_\ell}{4\ell^2 - 1} \right).$$

One could also distinguish the cases of even and odd n .



(7-12.c) (15 min.) The C++ function

```
unsigned int determineOrderCCF(unsigned int n);
```

is supposed to return the **order** [Lecture → Def. 7.4.1.1] of the quadrature formulas provided by an object of the class **CCFQuadRule** when constructed with a parameter n .

Fill in the missing parts of the code in the boxes (valid C++ syntax!):

```
unsigned int determineOrderCCF(unsigned int n) {
    const CCFQuadRule ccfr(n);
    int d = 0;           // Degree of monomial
    double quaderr;      // Quadrature error
    double I_ex;         // Exact integral value
    double I_qr;         // Value computed with CCF quadrature rule
    const double abstol = std::numeric_limits<double>::epsilon() * n
        * 2;
    const double reltol = std::numeric_limits<double>::epsilon() * n
        * 20;
    do {
        const Eigen::ArrayXd pdvals = ccfr. 
            .array() .pow(d);
        I_qr = ;
        I_ex = ;
        quaderr = std::abs(I_qr - I_ex);
        d = d + 1;
    } while ((quaderr <= abstol) || (quaderr <= reltol * I_ex));
    return ;
}
```

SOLUTION of (7-12.c):

The order of a quadrature rule is one plus the largest degree d of uni-variate polynomials that are still integrated exactly by the quadrature rule. Thanks to linearity, we need to check this for the elements of a basis of $\mathcal{P}_d$ only and we opt for the convenient basis of monomials $\{t \mapsto t^k, k = 0, \dots, d\}$, for which we find the exact integral values

$$\int_{-1}^1 t^k dt = \begin{cases} \frac{2}{k+1} & \text{for even } k, \\ 0 & \text{for odd } k, \end{cases} \quad k \in \mathbb{N}_0. \quad (7.12.4)$$

Applied to the monomials a generic $n + 1$ -point quadrature formula with nodes/weights c_j/w_j , $j = 0, \dots, n$ yields

$$Q(\{t \mapsto t^k\}) = \sum_{j=0}^n w_j c_j^k, \quad k \in \mathbb{N}_0. \quad (7.12.5)$$

This is the complete implementation of the function:

C++ code 7.12.6: Implementation of `determineOrderCCF()`

```

2  unsigned int determineOrderCCF(unsigned int n) {
3      const CCFQuadRule ccfr(n);
4      int d = 0;          // Degree of monomial, whose integral value is being
                          // tested
5      double quaderr;     // Quadrature error
6      double l_ex;        // Exact integral value
7      double l_qr;        // Value computed with CCF quadrature rule
8      // Predefined fixed tolerances
9      const double abstol = std::numeric_limits<double>::epsilon() * n * 2;
10     const double reltol = std::numeric_limits<double>::epsilon() * n * 20;
11     // Main quadrature loop
12     do {
13         // Compute powers of nodes:  $c_j^d$ 
14         const Eigen::ArrayXd pdvals = ccfr.nodes().array().pow(d);
15         // Evaluate quadrature formula: Implicit summation via dot
                          // product
16         l_qr = ccfr.weights().dot(pdvals.matrix());
17         // Exact integral value according to (7.12.4)
18         l_ex = (d % 2 == 0) ? (2.0 / (d + 1)) : 0.0;
19         quaderr = std::abs(l_qr - l_ex);
20         d = d + 1;
21     } while ((quaderr <= abstol) || (quaderr <= reltol * l_ex)); //
22     return d - 1;
23 }

```

Get it on  [GitLab \(clenshawcurtisfejer.hpp\)](https://gitlab.com/clenshawcurtisfejer).

Of course, testing `l_ex == l_qr` is pointless, because both values have been computed numerically and will certainly be affected by roundoff. This is why we have to test for absolute and relative closeness in Line 21 of the code.

Note that any other basis of $\mathcal{P}_k$ could also be used, for instance Legendre polynomials.



(7-12.d) (90 min.) [depends on Sub-problem (7-12.b)]

Your task is to devise a more efficient implementation of (the constructor of) the class **CCFQuadRule**. To that end you have created a copy, renamed it to **CCFQuadRule_Fast** and deleted parts of the constructor. Now you should re-implement the constructor in the file `clenshawcurtisfejer.hpp` so that the asymptotic effort for computing the nodes and weights of the Clenshaw-Curtis-Fejer quadrature rule becomes $O(n \log n)$ for $n \rightarrow \infty$.

SOLUTION of (7-12.d):

Of course, we aim to compute the weights w_j from (7.12.3) by means of the **discrete Fourier transform** (DFT) [Lecture $\rightarrow$ Def. 4.2.1.18]. To that end we rewrite the formula. Using the notations from Sub-

problem (7-12.b), we abbreviate

$$z_\ell := \frac{b_\ell}{4\ell^2 - 1} \in \mathbb{R} \quad \ell = 1, \dots, m, \quad z_0 := 1,$$

which permits us to express

$$w_j = \frac{a_j}{n}(1 - \tilde{w}_j), \quad \tilde{w}_j := \sum_{\ell=1}^m z_\ell \cos(2\ell j \frac{\pi}{n}), \quad j = 0, \dots, n-1. \quad (7.12.7)$$

Considering only $j \leq n-1$ is sufficient, since $w_n = w_0$. Thanks to Euler's formula $\cos x = \frac{1}{2}(e^{ix} + e^{-ix}) = \operatorname{Re}\{e^{-ix}\}$, and the fact that $z_\ell \in \mathbb{R}$, we can conclude

$$\tilde{w}_j = \operatorname{Re}\left\{\sum_{\ell=1}^m z_\ell e^{-\frac{2\pi i}{n}\ell j}\right\}, \quad j = 0, \dots, n-1. \quad (7.12.8)$$

Recall from [Lecture → Def. 4.2.1.18] that the DFT amounts to a matrix×vector multiplication with the **Fourier matrix**

$$\mathbf{F}_n := \left[\exp\left(-\frac{2\pi i}{n}\ell j\right) \right]_{\ell,j=0}^{n-1} \in \mathbb{C}^{n,n}. \quad [\text{Lecture} \rightarrow \text{Eq. (4.2.1.13)}]$$

Thus (7.12.8) means that

$$[\tilde{w}_j]_{j=0}^{n-1} = \operatorname{Re}\{\mathbf{F}_n \mathbf{z}\} = \operatorname{Re}\{\operatorname{DFT}_n(\mathbf{z})\} \quad \text{with} \quad \mathbf{z} := \begin{bmatrix} 0 \\ z_1 \\ \vdots \\ z_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n. \quad (7.12.9)$$

From the definition of the Fourier matrix it is clear that

$$\mathbf{F}_n \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

which means that (C++ indexing!)

$$[w_j]_{j=0}^{n-1} = \left[\frac{a_j}{n}(1 - \tilde{w}_j) \right]_{j=0}^{n-1} = \frac{1}{n} \operatorname{diag}\left([a_j]_{j=0}^{n-1}\right) (\operatorname{Re}\{\operatorname{DFT}_n(\mathbf{z}')\})_{0:n-1},$$

with $\mathbf{z}' = [1 \ -z_1 \ \dots \ -z_m \ 0 \ \dots \ 0]^\top \in \mathbb{R}^m$. This formula is implemented in the following code.

C++ code 7.12.10: FFT-based implementation of constructor of **CCFQuadRule_Fast**

```
2 CCFQuadRule_Fast::CCFQuadRule_Fast(unsigned int n) : weights_(n + 1) {
3   // TO DO: Provide an implementation with asymptotic complexity of
   //           O(n log(n))
4   // START
5   assert(n > 0);
```

```

6 // The nodes  $c_j = \cos\left(\frac{j\pi}{n}\right)$ ,  $j = 0, \dots, n$ 
7 nodes_ = (Eigen::ArrayXd::LinSpaced(n + 1, 0, n) * M_PI / n).cos().matrix();
8 // Length of sum:  $m = \frac{n}{2}$  for even  $n$ ,  $m = \frac{n-1}{2}$  for odd
9 //  $n$ 
10 int m = n / 2;
11 // Initialize vector  $\mathbf{z}$  to be multiplied with Fourier matrix
12 Eigen::VectorXcd z{Eigen::VectorXcd::Zero(n)};
13 z[0] = 1.0;
14 for (int l = 1; l <= m; ++l)
15     z[l] = -2.0 / (4.0 * l * l - 1.0);
16 if (n % 2 == 0)
17     z[m] /= 2.0;
18 // FFT-based multiplication with Fourier matrix
19 Eigen::FFT<double> fft;
20 // Eigen's built-in FFT crashes for vectors of length = 1
21 // This is a work around
22 if (n == 1)
23     weights_[0] = 2.0;
24 else
25     weights_.head(n) = 2.0 * (fft.fwd(z)).real() / n;
26 weights_[0] /= 2.0;
27 weights_[n] = weights_[0];
28 // END
29 }

```

Get it on 🐙 [GitLab \(clenshawcurtisfejer.hpp\)](https://gitlab.com/clenshawcurtisfejer).



End Problem 7-12 , 130 min.