

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 7-13: Smooth integrand by transformation (ArcCos)

Integrals with non-smooth integrands can *sometimes* be treated by a suitable transformation, which converts the integrand into a smooth function, see [Lecture → Rem. 7.4.3.10]. In this problem we study an example.

You should be familiar with the transformation of quadrature rules [Lecture → Rem. 7.2.0.4] and Gaussian-Legendre quadrature formulas [Lecture → Section 7.4], [Lecture → Section 7.4.2]. You must be able to perform integration by substitution.

Given a smooth function $f : [-1, 1] \rightarrow \mathbb{R}$ our task is to come up with a numerical approximation of

$$I(f) := \int_{-1}^1 \arccos(t) f(t) dt. \quad (7.13.1)$$

We want to approximate this integral using global Gauss-Legendre quadrature as introduced in [Lecture → Section 7.4.2].

The function $t \mapsto \arccos t \triangleright$
($\hat{=}$ the inverse of $\cos$ on $[0, \pi]$)

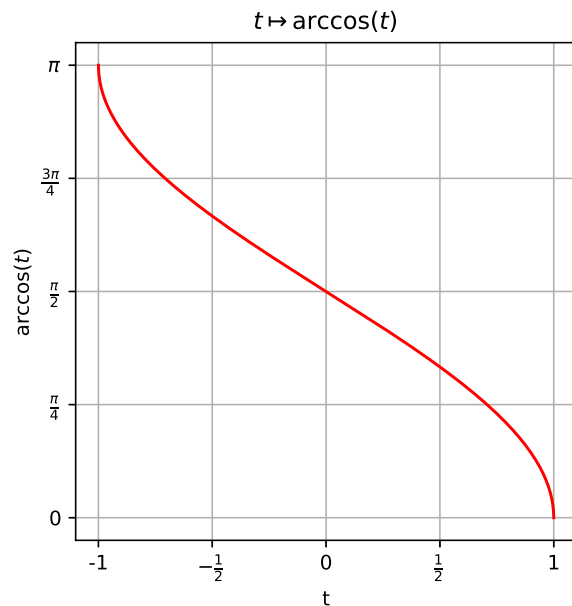


Fig. 78

The nodes and the weights of the n -point Gauss-Legendre quadrature rule on $[-1, 1]$ can be computed for $n \leq 256$ using the provided C++ function

```
QuadRule gaussquad(const unsigned n);
```

which initializes a data structure describing a quadrature rule

```
struct QuadRule {
    QuadRule() = default;
    explicit QuadRule(unsigned int n) : nodes_(n), weights_(n) {}
    Eigen::VectorXd nodes_;
    Eigen::VectorXd weights_;
};
```

The names of the data members tell their function. The function `gaussquad()` is declared in `gaussquad.hpp`.

(7-13.a) 🕒 (15 min.) In the file `arccosquad.hpp` implement a C++ function

```
void testConvGaussQuad();
```

that prints a table that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \rightarrow \infty$) of the quadrature error of n -point Gauss-Legendre numerical quadrature, when directly applied to (7.13.1) for $f(t) = \frac{1}{1+e^t}$.

Based on that table describe the observed empiric convergence of the quadrature error as a function of the number of quadrature nodes qualitatively and quantitatively. Explain, how you arrive at your

conclusions.

HINT 1 for (7-13.a): The function $t \mapsto \arccos t$ is available in C++ as `std::acos()`, if the header `cmath` is included. ┘

HINT 2 for (7-13.a): The integral cannot be evaluated in closed form. The idea is to use an approximate value provided by “overkill numerical quadrature” with many more quadrature nodes than used for the smallest error taken into account for gauging the convergence behavior.

Such a *reference value* is $I = 1.7576137811123187$. ┘

SOLUTION of (7-13.a):

Types of convergence have been introduced in [Lecture → Def. 6.2.2.7] for approximation errors. They apply to quadrature errors as well.

Due to the singularity of $t \mapsto \arccos t$ at $t = \pm 1$, we cannot expect exponential convergence in the number of quadrature nodes. So, taking the cue from [Lecture → § 6.2.2.9], we first try to detect algebraic convergence:

$$\epsilon_i \approx C n_i^{-p} \Leftrightarrow \log \epsilon_i \approx \log C - p \log n_i, \quad (7.13.2)$$

where $(\epsilon_i)_{i=1}^L$ is a sequence of quadrature errors, $(n_i)_{i=1}^L \in \mathbb{N}$ is the corresponding sequence of numbers of quadrature nodes and $p > 0$ is the *rate* of algebraic convergence. It is convenient to opt for a geometric growth of n_i , for instance, $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, because in this case

$$\log \epsilon_i \approx \log C - p \log m_0 - p i \log 2, \quad i = 1, \dots, L.$$

Then an *estimate* for the rate p can be obtained as

$$p \approx \frac{\log \epsilon_{i-1} - \log \epsilon_i}{\log 2}. \quad (7.13.3)$$

This is another quantity, one should tabulate.

Alternatively, using $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, the error behavior like (7.13.2) also manifests itself through

$$\frac{\epsilon_i}{\epsilon_{i+1}} \approx \frac{C n_i^{-p}}{C n_{i+1}^{-p}} = 2^p.$$

Hence, one may increase the number of quadrature points by a constant factor in a sequence of tests and then watch the ratio of two successive errors. If that ratio attains about the same value throughout the sequence, we can also conclude algebraic convergence. Thus the values $\epsilon_{i-1} : \epsilon_i$ could also be included in the table.

C++ code 7.13.4: Convenience struct `cvgdata`

```

2 struct cvgdata {
3     long n_{0};           // number of quadrature points
4     double val_{0.0};     // approximate value
5     double err_{-1.0};    // quadrature error
6     double rate_{-1.0};   // estimate for rate of algebraic convergence
7 };

```

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

C++ code 7.13.5: Code for `testConvGaussQuad()`

```

2 void testConvGaussQuad() {
3     // Table header
4     std::cout << std::setw(5) << "n" << std::setw(20) << " I_n " << std::setw(30)
5         << " Error " << std::setw(10) << (" rate\n " + std::string(65, '-'))
6         << std::endl;
7
8     // Value obtained by overkill quadrature as reference value
9     constexpr double ref_val = 1.7576137811123187;
10
11     // TO DO: Print a table that allows you to predict the asymptotic
12     // behaviour of Gauss-Legendre numerical quadrature when
13     // approximating I(f).
14     // START
15     // The integrand as lambda function
16     auto integrand = [](double t) { return std::acos(t) * 1.0 / (1.0 + std::exp(t)); };
17
18     // For recording errors etc.
19     std::vector<cvdata> data;
20     // We suspect algebraic convergence. Accordingly use doubling of
21     // number of
22     // quadrature nodes
23     constexpr int L = 8; // Use at most 2^L quadrature nodes
24     int n = 2;
25     for (int l = 1; l <= L; ++l, n *= 2) {
26         const QuadRule qr{gaussquad(n)};
27         // No transformation required: the integral is on [-1,1] anyway.
28         double I = 0.0; // Summation variable for quadrature formula
29         for (int j = 0; j < n; ++j) {
30             I += qr.weights_[j] * integrand(qr.nodes_[j]);
31         }
32         data.push_back({qr.weights_.size(), I, 0.0, 0.0});
33     }
34     for (int l = 0; l < data.size(); ++l) {
35         data[l].err_ = std::abs(data[l].val_ - ref_val);
36         if (l > 0) {
37             // Estimate for rate of algebraic convergence
38             data[l].rate_ =
39                 (std::log(data[l-1].err_) - std::log(data[l].err_)) / std::log(2.0);
40         }
41     }
42
43     // Output table
44     for (int l = 0; l < data.size(); ++l) {
45         const cvdata& item{data[l]};
46         std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
47             << std::setprecision(16) << item.val_ << std::setw(30)
48             << std::scientific << std::setprecision(16) << item.err_
49             << std::setw(10) << std::fixed << std::setprecision(2)
50             << item.rate_ << std::endl;
51     }
52     // END
53 }

```

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

The code produces the following output:

	n	I_n	Error	rate
1				
2				
3	2	1.7436937792198774	1.3920001892441247e-02	0.00
4	4	1.7553382175123289	2.2755635999898161e-03	2.61
5	8	1.7572962271476178	3.1755396470090069e-04	2.84
6	16	1.7575711396897615	4.2641422557165853e-05	2.90
7	32	1.7576082279627416	5.5531495770644312e-06	2.94
8	64	1.7576130715900262	7.0952229247467358e-07	2.97
9	128	1.7576136914136691	8.9698649619052162e-08	2.98
10	256	1.7576137698376213	1.1274697397922750e-08	2.99
11	512	1.7576137797013049	1.4110137502854059e-09	3.00
12	1024	1.7576137809381109	1.7420775932919241e-10	3.02
13	2048	1.7576137810929557	1.9362955683277505e-11	3.17

The data provide clear evidence for algebraic convergence with rate ≈ 3 .



(7-13.b) (10 min.) With a suitable change of variables transform the integral (7.13.1) into another integral whose integrand will be C^∞ -smooth on the *closed* interval of integration, if $f \in C^\infty([-1, 1])$.

SOLUTION of (7-13.b):

We use the transformation

$$s = \arccos t \Leftrightarrow t = \cos s,$$

for which

$$dt = -\sin s \, ds.$$

The new integration bounds will be $\pi = \arccos(-1)$ and $0 = \arccos(1)$. Thus, we arrive at

$$I(f) = - \int_{\pi}^0 s f(\cos s) \sin s \, ds. \quad (7.13.6)$$

Since $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, they do not affect the smoothness of the integrand:

$$f \in C^\infty([-1, 1]) \Rightarrow s \mapsto s f(\cos s) \sin s \in C^\infty([0, \pi]).$$



(7-13.c) (10 min.) [depends on Sub-problem (7-13.b)]

In the file `arccosquad.hpp` complete the code of the C++ function

```
template <typename FUNCTION>
double arccosWeightedQuad(FUNCTION &&f,
                          unsigned int n);
```

that

- expects the functor argument f to pass a function $f : [-1, 1] \rightarrow \mathbb{R}$ and to provide an evaluation operator **operator double** $()$ **(double) const**,
- uses n evaluations of the function f to compute an approximation $I_n(f)$ of $I(f)$ from (7.13.1),
- that achieves exponential asymptotic convergence $I_n(f) \rightarrow I(f)$ for $n \rightarrow \infty$, if the function f possesses an analytic extension beyond $[-1, 1]$.

SOLUTION of (7-13.c):

Of course, we now apply Gauss-Legendre quadrature to the transformed integral (7.13.6). If f is analytic in a neighborhood of $[-1, 1]$, then $s \mapsto sf(\cos s) \sin s$ will be analytic in a neighborhood of $[0, \pi]$, because $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, that is, analytic on all of $\mathbb{C}$.

Note that the weights and nodes of the Gauss-Legendre quadrature rules have to be transformed to the interval $[0, \pi]$ as explained in [Lecture $\rightarrow$ Rem. 7.2.0.4].

C++ code 7.13.7: Code for `arccosWeightedQuad()`

```

2  template <typename FUNCTION>
3  double arccosWeightedQuad(FUNCTION &&f, unsigned int n) {
4      double l = 0.0; // For accumulating quadrature result
5      // TO DO: Approximate I(f) with exponential convergence in n.
6      // START
7      // Obtain Gauss-Legendre quadrature rule
8      QuadRule qr{gaussquad(n)};
9      // The transformed integral covers the interval [0, π], which entails
10     // a transformation of the quadrature nodes and weights, see [Lecture
11         → Rem. 7.2.0.4]
12     qr.weights_ *= 0.5*M_PI;
13     qr.nodes_ = 0.5*M_PI*(1.0+qr.nodes_.array());
14     // Straightforward implementation of quadrature formula
15     for (int j = 0; j < n; ++j) {
16         const double c = qr.nodes_[j];
17         const double fval = f(std::cos(c));
18         l += qr.weights_[j] * fval * std::sin(c) * c;
19     }
20     // END
21     return l;
22 }
```

Get it on  GitLab (`arccosquad.hpp`).



(7-13.d)  (15 min.) [depends on Sub-problem (7-13.a) and Sub-problem (7-13.c)]

Analogous to Sub-problem (7-13.a) in the file `arccosquad.hpp` write a C++ function

```
void testConvTrfGaussQuad();
```

that outputs (in a suitable table) information that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \rightarrow \infty$) of the approximation error of your implementation of `arccosWeightedQuad()` for $f(t) = \frac{1}{1+e^t}$.

Characterize the observed empiric convergence of the quadrature error as a function of n qualitatively and quantitatively. Justify your conclusions.

SOLUTION of (7-13.d):

Note that $t \mapsto \frac{1}{1+\exp(t)}$ is analytic in a neighborhood of $[-1, 1]$, because its domain of analyticity is

$$\mathbb{C} \setminus (2\mathbb{Z} + 1)\pi.$$

Thus, this time we expect exponential convergence, that is, using the notation from the solution of Sub-problem (7-13.a) we expect

$$\epsilon_i \approx C \exp(-qn_i) \quad \text{for some } q > 0. \quad (7.13.8)$$

In this case it is advantageous to let n_i increase linearly, for instance, $n_i = i$, because this implies

$$\frac{\epsilon_{i+1}}{\epsilon_i} \approx \exp(-q) < 1.$$

In the case of exponential convergence we should observe a decay of the approximation error by a constant factor when raising n by 1. Thus tabulating the ratio of consecutive errors is highly recommended.

C++ code 7.13.9: Code for `testConvTrfGaussQuad()`

```

2 void testConvTrfGaussQuad() {
3     // Table header
4     std::cout << std::setw(5) << "n" << std::setw(20) << "l_n" << std::setw(30)
5         << "Error" << std::setw(10)
6         << ("decay ratio\n" + std::string(65, '-')) << std::endl;
7
8     // Value obtained by overkill quadrature as reference value
9     const double ref_val = 1.7576137811123187;
10
11     // TO DO: Print a table that allows you to predict the asymptotic
12     // behaviour of arccosWeightedQuad().
13     // START
14     // The integrand (/arccos(t)) as lambda funtion
15     auto f = [](double t) { return 1.0 / (1.0 + std::exp(t)); };
16
17     // For recording errors etc.
18     std::vector<cvdata> data;
19     // We expect exponential convergence. Increment number of
20     // quadrature nodes by
21     // one in each step
22     constexpr int L = 12; // Use a most L quadrature nodes
23     for (int n = 2; n <= L; ++n) {
24         data.push_back({n, arccosWeightedQuad(f, n), 0.0, 0.0});
25     }
26     for (int l = 0; l < data.size(); ++l) {
27         data[l].err_ = std::abs(data[l].val_ - ref_val);
28         if (l > 0) {
29             // Quotient of successive error values
30             data[l].rate_ = data[l].err_ / data[l - 1].err_;
31         }
32     }
33
34     // Output table
35     for (int l = 0; l < data.size(); ++l) {
36         const cvdata& item{data[l]};
37         std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
38             << std::setprecision(16) << item.val_ << std::setw(30)
39             << std::scientific << std::setprecision(16) << item.err_

```

```

39         << std::setw(10) << std::fixed << std::setprecision(2)
40         << item.rate_ << std::endl;
41     }
42     // END
43 }

```

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

The code produces the following output:

	n	I _n	Error	decay ratio
1				
2				
3	2	1.8492385346839764	9.1624753571657669e-02	0.00
4	3	1.7329033484157947	2.4710432696523954e-02	0.27
5	4	1.7605214393475945	2.9076582352758340e-03	0.12
6	5	1.7572524700920542	3.6131102026448758e-04	0.12
7	6	1.7576617761797750	4.7995067456341189e-05	0.13
8	7	1.7576074242880906	6.3568242281153431e-06	0.13
9	8	1.7576146283917027	8.4727938398643232e-07	0.13
10	9	1.7576136678083938	1.1330392490904728e-07	0.13
11	10	1.7576137962618741	1.5149555387949931e-08	0.13
12	11	1.7576137790901427	2.0221759822192098e-09	0.13
13	12	1.7576137813858141	2.7349544851063001e-10	0.14

The ratio of successive approximation errors clearly stabilizes at a value $\approx 0.14 < 1$, which hints at exponential convergence :

$$|I(f) - \text{arccosWeightedQuad}(f, n)| = O(0.14^n) \quad \text{for } n \rightarrow \infty.$$



End Problem 7-13 , 50 min.