

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 7-4: Nested numerical quadrature

This problem addresses numerical quadrature over a two-dimensional domain. It turns out that this problem can be reduced to two one-dimensional integrals, which are amenable to the techniques covered in [Lecture → Chapter 7].

Implementation in C++ requested connected with [Lecture → Chapter 7].

A laser beam has intensity

$$I(x, y) := \exp(-\alpha((x - p)^2 + (y - q)^2)), \quad x, y \in \mathbb{R}, \quad \alpha > 0, \quad (7.4.1)$$

on the plane orthogonal to the direction of the beam. The beam is directed orthogonal to the $x - y$ -plane.

Note that the radiant power absorbed by a surface is the integral of the intensity over the surface.

(7-4.a) 🕒 (10 min.) Write down a formula involving only 1D integrals and giving the radiant power absorbed by the triangle

$$\Delta := \{(x, y)^T \in \mathbb{R}^2 \mid x \geq 0, y \geq 0, x + y \leq 1\}.$$

as a double integral.

SOLUTION of (7-4.a):

The radiant power absorbed by Δ can be written as:

$$\int_{\Delta} I(x, y) \, dx \, dy = \int_0^1 \int_0^{1-y} I(x, y) \, dx \, dy.$$

This is only one way to express the power. You may also change the order of integration or perform some other transformation of the integral.



(7-4.b) 🕒 (15 min.) Write a C++ function

```
template <class Function>
double evalquad(const double a, const double b,
                Function &&f, const QuadRule & Q);
```

that evaluates an n -point quadrature formula as introduced in [Lecture → Section 7.2] for an integrand passed in `f` on domain $[a, b]$. It should rely on the quadrature rule on the reference interval $[-1, 1]$ that is supplied through an object of type **QuadRule**:

```
struct QuadRule { Eigen::VectorXd nodes, weights; };
```

(The vectors `weights` and `nodes` denote the weights and nodes, respectively, of a quadrature rule on the reference interval $[-1, 1]$, see [Lecture → Def. 7.2.0.1].)

SOLUTION of (7-4.b):

We transform the quadrature rule according to the formulas derived in [Lecture → Rem. 7.2.0.4].

C++-code 7.4.2: Implementation of `evalquad()`.

```

2  template <class Function>
3  double evalquad(const double a, const double b, Function &&f,
4                  const QuadRule &Q) {
5      double I = 0.;
6      // TODO: (7-4.b) Use Q to approximate the integral of f over [a,b]
7      // START
8      // Loop over all nodes/weights pairs
9      for (unsigned int i = 0; i < Q.weights.size(); ++i) {
10         I += f((Q.nodes(i) + 1.) * (b - a) / 2. + a) * Q.weights(i);
11     }
12     I = I * (b - a) / 2.;
13     // END
14     return I;
15 }

```

**(7-4.c)** 🕒 (20 min.) [depends on Sub-problem (7-4.a)]

Write a C++ function

```

template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N)

```

for the computation of the integral

$$\int_{\Delta} f(x, y) \, dx \, dy \quad (7.4.3)$$

using nested n -point, 1D **Gauss quadrature** [Lecture → Section 7.4] in combination with the function `evalquad()` of Sub-problem (7-4.b). Use the function `gaussquad()` from `gaussquad.hpp` to obtain weights and nodes of Gauss quadrature formulas on $[-1, 1]$.

HINT 1 for (7-4.c): Write (7.4.3) explicitly as a double integral as you were asked to do in Sub-problem (7-4.a). Take special care to correctly find the intervals of integration. ┘

HINT 2 for (7-4.c): C++ lambda functions are very useful for this kind of implementation, see [Lecture → Section 0.3.3]. ┘

SOLUTION of (7-4.c):

As we have seen in Sub-problem (7-4.a), the integral can be written as

$$\int_{\Delta} f(x, y) \, dx \, dy = \int_0^1 \int_0^{1-y} f(x, y) \, dx \, dy$$

In the C++ implementation, we define the auxiliary (lambda) function f_y :

$$\forall y \in [0, 1], f_y : [0, 1 - y] \rightarrow \mathbb{R}, x \mapsto f_y(x) := f(x, y)$$

We also define the (lambda) approximated integrand:

$$g(y) := \int_0^{1-y} f_y(x) \, dx \approx \frac{1-y}{2} \sum_{i=0}^n w_i f_y\left(\frac{1-y}{2}(x_i + 1)\right) =: \mathcal{I}(y), \quad (7.4.4)$$

the integral of which can be approximated, using a nested Gauss quadrature:

$$\int_{\Delta} f(x, y) \, dx \, dy = \int_0^1 \int_0^{1-y} f_y(x) \, dx \, dy = \int_0^1 g(y) \, dy \approx \frac{1}{2} \sum_{j=1}^n w_j \mathcal{I}\left(\frac{y_j + 1}{2}\right). \quad (7.4.5)$$

C++-code 7.4.6: Implementation of `gaussquadtriangle()`.

```

2  template <class Function>
3  double gaussquadtriangle(const Function &f, const unsigned N) {
4      double l = 0.;
5      // TODO: (7-4.c) Use N-node gaussquad() to integrate f over
6      // the triangle 0<=x,y, x+y<=1.
7      // START
8      // Get nodes/weights for integral over dx and dy
9      QuadRule Q;
10     gaussquad(N, Q);
11
12     // We define the function g as the function of y
13     // g(y) := ∫₀¹⁻ʸ f(x,y)dx.
14     auto g = [&f, &Q](double y) {
15         return evalquad(
16             0, 1 - y, [&f, &Q](double x) { return f(x, y); }, Q);
17     };
18     // We integrate the function g over y from 0 to 1 to get
19     // ∫₀¹ g(y)dy := ∫₀¹ ∫₀¹⁻ʸ f(x,y)dxdy.
20     l = evalquad(0, 1, g, Q);
21     // END
22     return l;
23 }
```

This problem can also be solved using loops (so no nested lambda function). This more straightforward approach is listed next.

C++-code 7.4.7: Loop-based implementation of `gaussquadtriangle_loop()`.

```

2  template <class Function>
3  double gaussquadtriangle_loop(const Function &f, const unsigned N) {
4      // Get nodes/weights for integral over dx and dy
5      QuadRule Q;
6      gaussquad(N, Q);
7      // Integration over y from 0 to 1 of g(y) := ∫₀¹⁻ʸ I(x,y)dx
8      double l = 0;
9      constexpr double a = 0., b = 1.;
10     for (unsigned int i = 0; i < Q.weights.size(); ++i) {
11         // Find out the y at which we are
12         const double y = (Q.nodes(i) + 1) * (b - a) / 2 + a;
13         // Define f_y(x) (y is fixed and f_y is a function of x)
14         auto f_y = [&f, &y](double x) { return f(x, y); };
15         // Compute g(y) as ∫₀¹⁻ʸ I(x,y)dx
16         l += evalquad(0, 1 - y, f_y, Q) * Q.weights(i);
17     }
18     // Rescale interval
19     return l * (b - a) / 2.;
20 }
```

If you choose that implementation, also have a look at the nested-lambda version listed in Code 7.4.6

and try to understand how it works.



(7-4.d)  Write a C++ function

```
void convtest2DQuad(unsigned int nmax = 20);
```

that applies the function `gaussquadtriangle()` from Sub-problem (7-4.c) to approximate $\int_{\Delta} I(x,y) dx dy$ using the parameters $\alpha = 1$, $p = 0$, $q = 0$. Tabulate and plot (using `MATPLOTLIBCPP`, storing the figure in `convergence.png`) the quadrature error w.r.t. to the number of nodes $n = 1, 2, 3, \dots, 20$ using the “exact” value of the integral $I \approx 0.366046550000405$. What kind of convergence do you observe and why?

HINT 1 for (7-4.d): [Lecture $\rightarrow$ § 6.2.2.9] explains how to distinguish different kinds of convergence from error plots. 

SOLUTION of (7-4.d):

N	I_{approx}	error
1	0.365808	0.000238736
2	0.367346	0.00129915
3	0.365986	6.05452e-05
4	0.366048	1.5884e-06
5	0.366047	3.11447e-08
6	0.366047	5.07046e-10
7	0.366047	7.22744e-12
8	0.366047	9.32032e-14
9	0.366047	6.10623e-16
10	0.366047	9.4369e-16
11	0.366047	0
12	0.366047	4.44089e-16
13	0.366047	5.55112e-16
14	0.366047	6.66134e-16
15	0.366047	0
16	0.366047	6.10623e-16
17	0.366047	1.11022e-15
18	0.366047	1.66533e-16
19	0.366047	7.21645e-16
20	0.366047	1.11022e-16

The table shows the corresponding approximated value of I and the error w.r.t. number of quadrature nodes N .

Why does the decrease of the quadrature error stall at $N = 9$? Well, by $N = 9$ we have reached an error of about the same size as the machine precision $\text{EPS} \approx 10^{-16}$, see [Lecture $\rightarrow$ Ex. 1.5.3.12]. Then roundoff error interfere with any further improvement of accuracy. Thus, what we see as errors for $N > 9$ is essentially “random” roundoff errors.

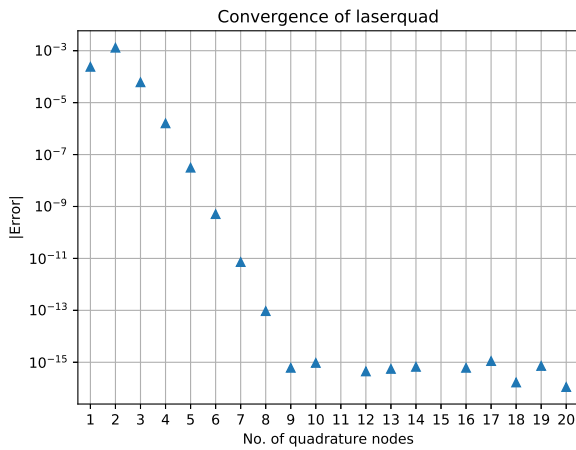


Fig. 62

We observe that for $N < 9$ the error points approximately lie on a straight line in the linear-logarithmic error plot, before the impact of roundoff errors becomes visible at very low error levels.

Thus we conclude that the implemented quadrature enjoys **exponential convergence**, recall [Lecture → Def. 6.2.2.7] and [Lecture → § 6.2.2.9].

We expect from theoretical considerations that the convergence is exponential, because we deal with an perfectly smooth (analytic) integrand, see [Lecture → § 7.4.3.2] and [Lecture → Ex. 7.4.3.9].

C++-code 7.4.8: Implementation of `convtest2DQuad()`.

```

2 void convtest2DQuad(unsigned int nmax = 20) {
3     // "Exact" integral
4     constexpr double I_ex = 0.366046550000405;
5     plt::figure();
6     // TODO: (7-4.d) Tabulate the error of gaussquadtriangle() for
7     // a laser beam intensity, using n=1,2,3,...,nmax nodes.
8     // START
9     // Parameters
10    constexpr double alpha = 1., p = 0., q = 0.;
11    // Laser beam intensity
12    auto I = [](double x, double y) {
13        return std::exp(-alpha * ((x - p) * (x - p) + (y - q) * (y - q)));
14    };
15
16    // memory allocation for plot
17    std::vector<double> err(nmax);
18    std::vector<int> num_pts(nmax);
19
20    std::cout << std::setw(3) << "N" << std::setw(15) << "I_approx"
21              << std::setw(15) << "error" << std::endl;
22    for (unsigned int n = 1; n <= nmax; n++) {
23        const double I_approx = gaussquadtriangle(I, n);
24
25        err[n - 1] = std::abs(I_ex - I_approx);
26        num_pts[n - 1] = n;
27
28        std::cout << std::setw(3) << n << std::setw(15) << I_approx << std::setw(15)
29                  << err[n - 1] << std::endl;
30    }
31    // Error plot rendered by matplotlibcpp
32    plt::semilogy(num_pts, err, "^", {"label", "error"});
33    plt::xlim(0.5, nmax + 0.5);
34    plt::xticks(num_pts);
35    plt::xlabel("No. of quadrature nodes");
36    plt::ylabel("|Error|");
37    plt::title("Convergence of laserquad");
38    plt::grid("True");
39    // END
40    plt::savefig("./cx_out/convergence.png");
41 }

```



End Problem 7-4 , 45 min.