IWR, Heidelberg University — Summer Semester 2017

Exercise Sheet 7 — 30. 05. 2017

Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Dr. Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 13. 06. 2017 before the lecture

Exercise 1 Interfaces: Integration

In the lecture the concept of interfaces was introduced using virtual methods, and numerical integration was presented as an example. We now want to revisit and expand this example.

The goal is to write a code library that allows, via numerical integration, to determine the integral of an arbitrary function $f(t)$ on a given interval $t \in [A, B]$ ($A, B \in \mathbb{R}$). The interval $[A, B]$ is divided in N subintervals of uniform length Δt.

$$\Delta t = \frac{B-A}{N}, \quad t_i = A + \Delta t \cdot i, \quad i = 0, \dots, N$$

On each of the subintervals the integral can be estimated using an appropriate quadrature rule:

$$\int\limits_A^B f(t)dt = \sum_{i=0}^{N-1} \left(Q_{t_i}^{t_{i+1}}(f) + E_{t_i}^{t_{i+1}}(f) \right)$$

Here Q_a^b refers to the result of the quadrature rule on the interval $[a, b]$ and E_a^b to the error. Quadrature rules are normally based on polynomial interpolation. This means it is possible to specify the maximum degree of polynomials that are integrated exactly and the order of the error in the case of functions that can't be integrated exactly.

In this exercise, we want to limit ourselves to two quadrature rules:

- The trapezoidal rule:
$$Q_{\text{Trapez}}{}_a^b(f) = \frac{b-a}{2}(f(a) + f(b))$$
is exact for polynomials up to first order, the error is $O(\Delta t^2)$.

- The Simpson rule:
$$Q_{\text{Simpson}}{}_a^b(f) = \frac{b-a}{6} \cdot \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$
is exact for polynomials up to degree two, the error is $O(\Delta t^4)$.

A good test for such a numerical integration is to study the order of convergence. For a problem with a known solution, calculate the error E_N for N subintervals and for $2N$ subintervals. Doubling the number of subintervals cuts the interval width Δt in half, reducing the error. The quotient

$$EOC = \frac{\log(E_N/E_{2N})}{\log(2)}$$

is an indication of the order of convergence of the error (for the trapezoidal rule you should observe order 2).

1. Design interfaces for the different concepts. To this end, proceed as follows:
   - First, identify the abstract concepts in our problem.
     - We want to evaluate integrals of various functions; so we have the function as an abstract concept.
     - An integral is represented as a sum of integrals of subintervals.
     - On each subinterval a suitable quadrature formula is applied.
     - The quadrature formula evaluates the function to be integrated at certain points in order to determine an approximate solution.
   - How are these abstract concepts mapped to interfaces? Usually, one introduces a base class as an interface description for each concept.
     - What is a suitable interface for a function? Maybe you already know one from the lecture.
     - A quadrature rule evaluates a function on a given interval, how would a suitable virtual method look like?
     - For each quadrature rule it is also possible to specify the degree of exactly integrable polynomials, and specify the order of convergence of the integration error.
     - The actual integration (composite quadrature rule) is parameterized with a quadrature rule and a function. It evaluates the integral of the function on an interval that has to be specified, dividing the interval into N subintervals. In addition, the quadrature formula that is to be used must be specified.
     - Apart from the equidistant composite quadrature used here, there may be other approaches (see second exercise). In anticipation of extensions, also introduce the abstract concept of a composite quadrature rule, although right now there is only one actual implementation.
2. Describe your interfaces and explain the design decisions. What are the reasons for your specific choice of interfaces?
3. Implement the interfaces with dynamic polymorphism. Write implementations for all of the above variants of abstract concepts. Your functions and classes should be as general as possible, i.e. use interface classes for arguments and return values. Make use of the best practices introduced in the lecture.
4. Test your implementation with the integration of $2t^2 + 5$ on the interval $[-3, 13]$ and $\frac{t}{\pi}\sin(t)$ on the interval $[0, 2\pi]$. To do this, introduce test functions as a subset of the functions above. Each test function should have two additional methods:
   - `void integrationInterval(double& l, double& r) const;` to define the limits that should be used when the function is used for testing.
   - `double exactIntegral() const;` to obtain the exact solution when using the test interval.

   Use dynamic polymorphism in your implementation of the tests. Write a free function that determines the order of convergence for a given test problem and a given quadrature rule. It should also be possible to choose the kind of composite quadrature, although there is only one choice in the context of this exercise.

   In an improved variant, this function can additionally compare the expected order of convergence of the quadrature rule with the observed order of convergence. Note that the specified convergence order is valid only asymptotically.

20 Points

EXERCISE 2 ADAPTIVE INTEGRATION

An improvement over the integration using equidistant intervals is adaptive integration. Here one tries to only use small intervals Δt where the error is large otherwise.

If you are in need of additional points or not challenged enough `;-)`, you may write a class for adaptive integration as complement for the equidistant one from the previous exercise. This new version should use the same interface for functions and quadrature rules and fit into the test infrastructure you have written.

One can estimate the error in one of the subintervals by comparing the result of quadrature with the result of an interval that has been refined once, i.e. the local error is estimated through hierarchical refinement. The goal is to have the "error density", i.e. error per unit length of interval, to be about the same in all subintervals. Subintervals with large errors are divided again iteratively or recursively. One starts with a single interval or a small number of subintervals and repeats the error estimation and refinement until the desired number of subintervals is reached.

To avoid having to constantly reevaluate the integrals of the individual subintervals, it is helpful to store the subintervals, integrals and errors in a separate data structure each. For this purpose doubly linked lists may be useful, they are available in the C++ Standard Library under the name `deque`. You are free to design more or less efficient algorithms for the local error estimation and for deciding which intervals should be bisected and in which order. Your primary concern is a program that produces good approximations w.r.t. the number of intervals used, thoughts about runtime efficiency are a good idea but not mandatory.

The points of this exercise don't count towards the total number of points that can be achieved during the semester. Consequently, you can do this exercise instead of another one later on or in addition to the other exercises to achieve a higher point percentage.

10 Bonus Points

As always: Put comments into your code, and explain what its purpose is.