Simulation and High-Performance Computing

University of Kiel, September 28th to Oktober 9th, 2020.

Exercises: Day Two

CSC Computational Science Center

General Remark

On day two, we will look at a few different ways of approaching a problem - such as differentiation or integration - by interpolating an underlying function with a polynomial and then solving that problem for the *much easier to handle* polynomial. This then gives us an *approximate* solution to the original problem. From an abstract viewpoint, our methods for solving ordinary differential equations also emanate from that approach. Usually, but not always, it holds that the higher order polynomial we use, the better the approximation is.

Main Exercises: Applications of Lecture 03

a) Approximating the First Derivative Through Fourth Order Interpolation

Work out a formula for approximating the derivative $f'(0)$ of a given function f by interpolating it with a fourth order polynomial p in the points -2, -1, 1, 2 and then evaluating $p'(0)$. **Implement** the application of your formula to the function $f(t) = e^t$ in the file `exercise_polynomial.c`. Do not forget to implement a calculation and printing of the error. **Compile** and **execute** `exercise_polynomial`.

Now, alter your formula by interpolating in the points $-2h$, $-h$, h, $2h$ for some small $h > 0$ instead. **Implement** - with decreasing sizes of h - the application of your new formula to the function $f(t) = e^t$ in `exercise_polynomial.c`. **Test** your algorithm and observe the order of convergence.

b) Approximating a Limit Through Third Order Interpolation

Similarly to a), work out a formula for approximating the limit $\lim_{t\to 0} f(t)$ of a given function f by interpolating it with a third order polynomial p in the points $\frac{1}{4}$, $\frac{1}{2}$, 1 and then evaluating $p(0)$. **Implement** the application of your formula to the function $f(t) = \frac{e^t-1}{t}$ in the file `exercise_polynomial.c`. Again, do not forget to implement a calculation and printing of the error. What is the exact value $\lim_{t\to 0} f(t)$ here? **Test** your algorithm.

Now, alter your formula by interpolating in the points $\frac{h}{4}$, $\frac{h}{2}$, h for some small $h > 0$ instead. **Implement** - with decreasing sizes of h - the application of your new formula to the function $f(t) = \frac{e^t-1}{t}$ in `exercise_polynomial.c`. **Test** your algorithm and observe the order of convergence.

c) Approximating an Integral through Quadrature and Composite Quadrature

Computing analytical solutions of integrals is sometimes hard and often even impossible. This is where *numerical integration* or *quadrature* come in handy. One of the simplest approaches to this task is the *trapezoidal rule*.

Implement the trapezoidal rule in the file `exercise_quadrature.c` and test your code for the function

$$f : \mathbb{R} \to \mathbb{R}, \quad x \mapsto \exp\left(-\frac{x^2}{2\sigma^2}\right),$$

by computing the integral

$$\int_{-0.5}^{0.5} f(x)\,dx.$$

The trapezoidal rule - as well as the Simpson quadrature rule that was also introduced in the lectures - have in common that there is no mechanism to control the error of the approximation. This drawback is fixed by *composite quadrature rules*.

Work out and **implement** a composite version of Simpson's rule. **Test** your implementation with the same integral as before. Which rate of convergence can you observe with respect to the number of subintervals?

Additional Exercises: Applications of Lecture 04

a) Classical Runge-Kutta Method and Error Calculation

All time-stepping methods we have seen so far fit into the framework of *Runge-Kutta methods*. Within the file `ode.c`, **implement** the *classical Runge-Kutta method* in the function `rk_classic_step`.

In order to better be able to examine a time-stepping method, we need to compute an error. Since for many problems, an analytical solution is hard to come by or even unknown, we take the approach of first calculating a *reference solution* by using a reliable and accurate method with a very small time step.

Within the file `exercise_mass_spring.c`, **implement** a solution of the mass-spring system using the classical Runge-Kutta method with a step size of $\delta = 10^{-5}$ and store the position value at the last point in time in some variable.

Below that - in the same file - then **implement** a solution of the mass-spring system using the classical Runge-Kutta method with varying but much larger step sizes. Use the stored position value to calculate an error. **Run** your programm and see which order of convergence you can observe.

b) Predator-Prey (also: Lotka-Volterra) Equations

Here we will look at another interesting example from the class of ordinary differential equations. It describes the dynamics of biological systems in which two species interact, one as a predator and the other as prey.

Implement the Lotka-Volterra equations - as they were introduced in lecture 01 - in the file `exercise_lotka_volterra.c`. Choose and **implement** a time-stepping method and a way of assessing the solution - such as visualization via gnuplot or calculating an error using a reference solution. **Test** your algorithm.

c) The Adams-Bashforth Methods

All previously introduced methods for the solution of ordinary differential equations had in common that they *only* used information from the numerical solution at the time step *exactly before the current one*. Multistep methods aim to store and use the information from the numerical solution at *multiple* previous time steps.

Some of the best-known multistep methods are the *Adams-Bashforth methods*. Note that these

methods are *explicit*. We want to work with the Adams-Bashforth method with $m = 3$ and equidistant time steps as it was described in the lectures. **Implement** it in the function `adams_bashforth_step` in the file `ode.c`.

Apply that Adams-Bashforth method to a problem of your choice - such as the mass-spring system or the Lotka-Volterra equations - and assess the solution - i.e., by visualization via gnuplot or computation of an error via a reference solution.

You have to pay special attention to the *starting values*. Usually, only starting values for one point in time are given. In multistep methods, however, one usually needs more values to begin the multistep procedure. Come up with a way to generate the missing values, **implement** everything and **test** your algorithm.

Please provide feedback about the format/difficulty/length of the exercises so that we can adjust accordingly. Contact us via Email or on the OpenOLAT forum.