Exercise Sheet #8

Deadline: 10.06.2024, 12:00h

Problem 1 (*Butcher Tableaus*) (10 points)

The general form of an explicit Runge-Kutta method is given by

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i,$$

where

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_i &= f\left(t_n + c_i h,\ y_n + h \sum_{j=1}^{s} a_{ij} k_j\right), \qquad i = 2, \dots, s. \end{aligned}$$

To specify a particular method, one needs to provide the integer s (the number of stages), and the coefficients a_{ij} (for $1 \le j < i \le s$), b_i (for $i = 1, \dots, s$) and c_i (for $i = 1, \dots, s$), which is most conveniently done in a *Butcher tableau*

$$\begin{array}{c|cccc} 0 & & & & \\ c_2 & a_{21} & & & \\ \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

(a) What are the Butcher tableaus for Euler's method and a fourth-order Runge-Kutta method?

(③ points)

(b) Explicitly state the evolution equations for the Runge-Kutta method with Butcher tableau

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

and implement the method in `C++`.

(⑤ points)

(c) Use your implementation to solve a simple harmonic oscillator

$$\ddot{x}(t) + \omega_0^2 x(t) = 0.$$

(② points)

Problem 2 (*Numerical Solution of Differential Equations*) (10 points)

The Kepler problem, namely a point mass in a plane under a $-k/r$ potential, was introduced in the lecture (link) as an example for a differential equation that can be solved with different numerical methods.

(a) Implement solvers of the Kepler problem using

1. Euler's method,
2. fourth-order Runge-Kutta,
3. the leapfrog scheme
4. and the Adams-Bashforth scheme.

Since the trajectories lie on a plane, only consider two dimensions.

(⑥ points)

(b) For initial conditions with negative energy, the trajectories are bounded and closed. Plot the trajectories for the different algorithms. Are they bounded and closed? Also, plot the total energy and angular momentum over time to check whether they are conserved.

(③ points)

(c) What happens to the orbits with a potential $V(r) = -k/r^\alpha$ with $\alpha \neq 1$? Use $\alpha = 1/2,\ 3/2$.

(① point)

(d) **Optional:** Experiment with a harmonic potential, i.e. with $\alpha = -2$ and $k < 0$.

Problem 3 (Advanced: *Hash Table*) (10 points)

In many applications, the problem arises to store a pair of data, which we will from now on denote *key* and *value*. Consider for example a phone book software, where we want to associate a name (key) with a phone number (value). For this purpose, an often encountered and very useful data structure are *hash tables*[1]. The idea of behind hash tables is, to input the key into a function, called *hash function*, that calculates an index for the key and then stores the associated value at the respective index in an array underlying the hash table. The great thing about hash tables from a programmers point of view is that the lookup (i.e. finding the value for a given key or in our example the phone number for a given name) is very fast, since the index, at which the corresponding value is stored, can quickly be calculated from the key.[2] Note, that it is required that the keys are unique (on the other hand, the values don't have to be unique).
The goal of this problem is to implement a very simple (and mostly educational) hash table in `C++` that stores `int` values associated with `int` keys.

- We first build the bare-bones structure of our hash table and implement the functionality in the following parts. Write a class `HashTable` with two private member variables `int t_size` and `int* t_values` that are the size of the hash table, i.e. the number of values one can store, and a pointer to the array, where the values will be stored, respectively. Add a constructor, that takes an `int size` as an argument and initializes `t_size` with `size` and `t_values` with an integer array of size `size`. Furthermore, implement a destructor that deletes the `t_values` array.

- Next, we implement a hash function. In the calculation, that the hash function performs to produce an index for a given key, it can happen that two different keys produce the same index. This is referred to as a *hash collision* and the quality of a good hash function is to have as little collisions as possible. Usually, if a collision happens, two different values (with their associated keys) are stored at the same index. The terminology here is that we have *buckets* at every index. Then, at lookup, we still have to go through the bucket to find the correct key, but the number of items will be substantially smaller than the size of the hash table. Since we do not care about quality here, we just use a very simple hash function and only store a single value at a given

[1] Sometimes they are referred to by *hash maps* or (e.g. in Python) *dictionaries.*

[2] On the flip side, imagine you would store pairs of names and phone numbers in an ordinary array: Finding the phone number of John Doe would require going through (potentially the whole) array until we encounter John and his phone number!

index. If a collision does happen, we just output a warning to the user (we will implement that later). For your implementation we will use the following hash function:

```
int hash(int key) const {
    return key % t_size;
}
```

Implement that function as a private method in the `HashTable` class (yes, you just have to copy-paste that function into your program).

- Next, implement a function
  `void insert(int key, int value){...}`
  that inserts `value` into the hash table. Use `key` and the hash function to determine the index in `t_values`, where `value` must be stored. If you detect a hash collision, simply output a warning to the user and do absolutely nothing.
  Hint: To easily detect a hash collision, it helps to make sure that `t_values` is initialized with zeros.

- Further, implement a function
  `int get(int key)const {...}`
  that retrieves the value associated to `key` and returns it.

- Finally, test your implementation with a meaningful example. Try to produce a hash collision.