Simulation and High-Performance Computing
University of Kiel, September 28th to Oktober 9th, 2020.

Exercises: Day One

CSC Computational Science Center

General remarks

In this course, the exercises will be done in the C-Programming language. As already explained, we do not expect you to already know the ins and outs of C-Programming. Hence, you will find a set of preinstalled files to provide you with the basic framework needed to complete the exercises. Before we start with the first exercise, a short introduction into our basic datatypes is given:

```
typedef double real;
typedef double field;
```

The idea behind these definitions is the following: We want to have an abstract view of datatypes. Instead of directly using a `float` or a `double` variable, we should think about the role it is taking. Do we want to store some interpolation points, which are always real numbers, or do we want to store some matrix coefficients, which can be either from the field of real numbers $\mathbb{R}$ or from the field of complex numbers $\mathbb{C}$, for instance.

Therefore we introduce the type `real` and use it to store real numbers. Similarly, we introduce the type `field` , which can handle values from some field.

The number of bits we want to spend and the accuracy we want to achieve is something completely different and can vary from problem to problem. Our approach lets us easily adjust the specific data types we use by changing them at a central position in the code.

```
typedef unsigned int uint;
```

In 99 % of all cases we need integer variables for indexing. Therefore it makes sense to restrict to non-negative integers. In order to make the code shorter, we should use the type `uint` for this task.

Matrices and vectors

We will need matrices and vectors in a wide variety of applications. Therefore we define the following datastructures in the `linalg` module.

```
struct _vector {
   field *x;
   uint dim;
};

typedef struct _vector vector;
typedef vector *pvector;
```

We can use the datatype `vector` which consists of some integer `dim` denoting its dimension and some array of `field` values stored in `x`. For all datatypes we also define the type again with a `'p'` as prefix. This means we want to deal with a pointer to this type.

If we have some vector $v \in \mathbb{K}^n$, we can access its i-th coefficient by `v->x[i]`.

For the datatype of a matrix we have a similar structure, but another very important member, the *leading dimension* `ld`.

```
struct _matrix {
   field *x;
   uint rows;
   uint cols;
   uint ld;
};

typedef struct _matrix matrix;
typedef matrix *pmatrix;
```

For a coefficient at the position (i,j) of a matrix $A \in \mathbb{K}^{n \times m}$ we need to state `A->x[i + j * ld]`, where i is the row-index and j respectively the column-index. This means we want to store matrices as Fortran does, in a *column-major-order*.

Although in many cases the members `rows` and `ld` might coincide, please always use the latter one for indexing due to handling of submatrices.

Main Exercises: Applications of Lecture 01

a) Mass-Spring System Using the Explicit Euler Method

One of the simplest ordinary differential equations is the *mass-spring system* in 1D. An easy derivation yields the equation

$$y'(t) = \begin{pmatrix} x'(t) \\ v'(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ -\frac{c}{m}x(t) \end{pmatrix} = f(t, y(t)).$$

Implement the right-hand side function of the mass-spring system in the file `exercise_mass_spring.c`.

In order to solve the differential equation, also **implement** the function `euler_step` in the file `ode.c`. That function should perform a single step of the explicit Euler method.

After **compiling** and **executing** the file `exercise_mass_spring`, Euler's method is being used with different step sizes δ and the data is stored within the folder `data`.

Before we take a look at the visualization of said data, we implement a different method to solve the same problem. It is, in a way, more accurate, but also has a higher computational cost.

b) Mass-Spring System Using Runge's Method

In a similiar way, **implement** the function `runge_step` in the file `ode.c`. That function should perform a single step of Runge's method.

After **compiling** and **executing** the file `exercise_mass_spring`, both Euler's and Runge's method are now being used with different step sizes δ and the data is stored within the folder `data`.

Now change directory to the folder `gnuplot` and use gnuplot to generate graphs from the data.

```
$> cd gnuplot
$> gnuplot "plot_euler_vs_runge.plt"
```

c) Gravitational Problem

Now **implement** the right-hand-side of the 3-dimensional *gravitational problem* in `exercise_gravitation.c`.

Since here we consider a 3D problem, we have to discuss how to store the coefficients within the vector `yt`.

Let $n \in \mathbb{N}$ be the number of interacting bodies and let for $i \in [1:n]$ be $x_i \in \mathbb{R}^3$ their positions, $v_i \in \mathbb{R}^3$ their velocities and m_i their masses respectively. Then we will store all coefficients as follows:

$$y = \big((x_1)_1, \dots, (x_n)_1, (x_1)_2, \dots, (x_n)_2, (x_1)_3, \dots, (x_n)_3, (v_1)_1, \dots, (v_n)_1, \dots, (v_n)_3, m_1, \dots, m_n\big) \in \mathbb{R}^{7n}.$$

Choose one of the two methods that we used above to solve the problem. **Compile** and **execute** your program and see if the values converge for decreasing values of δ.

Additional Exercises: Applications of Lecture 02

a) The Leapfrog Method

In the first part of the lectures and exercises, we saw that Euler's method had a lower computational cost than Runge's method but also had a lower convergence rate. We now use the leapfrog method, which is as accurate as Runge's method and about as 'cheap' as Euler's.

In order to **implement** the leapfrog method, you have to choose the step size δ and the starting values correctly. Therefore, do not try to implement the method for the general case. You should rather stick to a particular problem by implementing the method `leapfrog_step` in the file `ode.c` but *also* by adjusting the step size and starting values in, i.e., `exercise_mass_spring.c` correctly.

Compile and **execute** your program, which will produce some data that can once again be visualized by gnuplot. When using gnuplot, think about where to store the data and how to adjust the gnuplot script accordingly.

b) The Crank-Nicolson Method

Another drawback of Euler's method is the conservation of energy. We have seen that this method is not capable of keeping the energy of a system at a constant level, which would be very much appreciated in many applications.

It can be shown that the *Crank-Nicolson method* on the other hand *is* capable of conserving the energy of a given system. In general, though, it is an implicit method. I.e we have to solve some set of equations for every time step we want to perform.

In the case of the mass-spring system this again can be rewritten as an explicit formula. **Implement** the Crank-Nicolson method in the file `ode.c` in the function `crank_nicolson_step` and **test** your algorithm.

Finally, come up with a way to measure conservation of energy for the mass-spring system numerically and **implement** it. Then compare different methods in regards to their conservation of energy. Can you observe a better energy conservation when using the Crank-Nicolson method?

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.