# Einführung in die Programmierung für Physiker

SoSe 2024 – Marc Wagner

Michael Eichberg: eichberg@itp.uni-frankfurt.de

# Project part I

## 1 Overview

In the lecture you have seen, how ordinary differential equations (ODEs) can be solved numerically using the Runge-Kutta method. An obvious example for such an ODE from classical mechanics is Newton's equation of motion for the harmonic oscillator, where the analytical solution is known. The anharmonic oscillator, on the other hand, is a problem, where such an analytical solution does not exist.

In the first part of the programming project, it is your task to numerically solve the equation of motion for the 1-dimensional anharmonic oscillator and study its trajectory. This is a preparatory step for the upcoming project parts, where you will solve a time-independent Schrödinger equation and compute the corresponding quantized energy levels.

## 2 Assignment

Consider the 1-dimensional anharmonic oscillator given by the Lagrangian

$$L(x,\dot{x}) = \frac{m}{2}\dot{x}^2 - m\alpha x^n\,, \tag{1}$$

where $\alpha > 0$ is a constant and $n \in 2, 4, 6, \ldots$ is the characteristic exponent of the potential. The corresponding equation of motion is

$$m\ddot{x} = -m\alpha n x^{n-1}\,. \tag{2}$$

Rewriting this as a system of first-order ODEs yields

$$\dot{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{y}(t), t)\,, \tag{3}$$

where

$$\boldsymbol{y}(t) = (x(t), v(t))\,, \tag{4}$$

$$\boldsymbol{f}(\boldsymbol{y}(t), t) = (v(t), -\alpha n x^{n-1}(t))\,. \tag{5}$$

(i) Eqs. (17) to (21) in the lecture notes provide formulae to perform a 4th order Runge-Kutta step with fixed step-size $\tau$. Write a C program, which uses the Runge-Kutta method with step-size $\tau$ and number of steps $N$ to numerically solve Eq. (5) for $\alpha = 0.5$ and $n = 2$ (this corresponds to the harmonic oscillator with $\omega^2 = 1$), and writes the results, i.e. a sequence of points $(t, x(t))$, to a file.

(ii) Compute $\boldsymbol{y}(t)$ in the interval $[0, 4\pi]$ with initial conditions $\boldsymbol{y}(0) = (1.0, 0.0)$ and step-sizes $\tau_k = \pi \cdot 2^{-k}$, $k = 1, 2, \ldots, 6$, and plot the results (e.g. using Gnuplot).

(iii) Now, consider the anharmonic oscillator with $\alpha = 1.0$, $n = 6$ and the same initial conditions $\boldsymbol{y}(0) = (1.0, 0.0)$ and use your code to determine the oscillation period $T$. You can do this by finding the first root of the trajectory for positive $t$, i.e. the point $t_0 = T/4 > 0$, given by $x(t_0) = 0$. A possibility to determine $t_0$ approximately is computing $x(t)$ using the Runge-Kutta method with step-size $\tau = 10^{-8}$ and define $t_0 = t + \tau/2$, where $x(t)x(t + \tau) < 0$.

Write a `README.txt` with instructions to compile your code and generate and discuss results for the above tasks.

Produced with the ExerciseHandler

## 3 Code design

When carrying out numerical investigations, it is strongly recommended to first think about the structure of your program in a preparatory step. Since this is probably the first big project in your career as a physicist, this section discusses, how you can and should design your code.

The assignment in section 2 requires the writing of code to carry out a specific task (solving the anharmonic oscillator in classical mechanics). Part of the assignment is, however, quite general, namely solving ODEs using the Runge-Kutta method, which will also be required in upcoming parts of the project. It is therefore advantageous, to implement the Runge-Kutta algorithm in a separate `.h` and `.c` file and compile the latter to a library (an `.o` file) such, that it can be reused later in an easy and flexible way. The minimal amount of files should, thus, be three, where

- `rk.h`, `rk.c` include general Runge-Kutta components, not related to the anharmonic oscillator,
- `oscillator.c` includes everything specific to tasks (i)-(iii) and the anharmonic oscillator, in particular the `main` function. Moreover, `oscillator.c` should include a function which computes $\boldsymbol{f}(y(t), t) \cdot \tau$, since this function contains the dynamics of the anharmonic oscillator. It can be declared as

  ```
  void compute_f_tau(double *f_tau, const Y_T *y_t, double tau);
  ```

  where the result is stored in an array pointed to by `*f_tau` and `*y_t` contains the value of $\boldsymbol{y}(t)$ at time $t$ (see the definition of the type `Y_T` below).

In the following, we provide several `struct`s and function declarations, which should be part of `rk.h` and `rk.c`, and which you should use in your project.

-
  ```
  #define __N__ 2
  ```

  This symbolic constant defines the size of the vector $\boldsymbol{y}(t)$. Setting `__N__` to the fixed value 2 restricts the Runge-Kutta library to problems, where $\boldsymbol{y}(t)$ has $n = 2$ components. A generalization to arbitrary $n$, which can be chosen during runtime, requires dynamic memory allocation and increases the programming effort and difficulty of this project.[1]

-
  ```
  struct y_t {
      double t;
      double y[__N__];
  };

  typedef struct y_t Y_T;
  ```

  This `struct` contains the function $\boldsymbol{y}(t)$, i.e. the solution of the ODE, at time $t$.

-
  ```
  typedef void (*COMPUTE_F_TAU_PTR)(double *, const Y_T *, double);
  ```

  The purpose of this `typedef` is to make the code more readable. `COMPUTE_F_TAU_PTR` defines a new type, which is a pointer to a `void` function taking one `double *`, one `const Y_T *` and one `double` variable as arguments. A pointer of this type can later be utilized to store the address of the function `compute_f_tau`, which computes $\boldsymbol{f}(\boldsymbol{y}(t), t) \cdot \tau$ (see above).

-
  ```
  // k1 = f(y, t) * tau
  void compute_k1(double *k1, const Y_T *y_t, double tau,
     COMPUTE_F_TAU_PTR compute_f_tau_ptr);
  // k2 = f(y + k1/2, t + tau/2) * tau
  void compute_k2(double *k2, const Y_T *y_t, const double *k1,
     double tau, COMPUTE_F_TAU_PTR compute_f_tau_ptr);
  // k3 = f(y + k2/2, t + tau/2) * tau
  ```

[1] If you are interested in a challenge, feel free to design a more powerful Runge-Kutta library, where the user can provide $n$ during runtime.

Produced with the ExerciseHandler

```
void compute_k3(double *k3, const Y_T *y_t, const double *k2,
   double tau, COMPUTE_F_TAU_PTR compute_f_tau_ptr);
// k4 = f(y + k3, t + tau) * tau
void compute_k4(double *k4, const Y_T *y_t, const double *k3,
   double tau, COMPUTE_F_TAU_PTR compute_f_tau_ptr);

// y(t+tau) = y(t) + (k1 + 2*k2 + 2*k3 + k4) / 6
void step_rk4(Y_T *y_t_tau, const Y_T *y_t, double tau,
   COMPUTE_F_TAU_PTR compute_f_tau_ptr);
```

These functions compute $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, $\boldsymbol{k}_3$, $\boldsymbol{k}_4$ and $\boldsymbol{y}(t+\tau)$, using information about $\boldsymbol{y}(t)$ stored in $*y_t$, and a pointer to a function representing $\boldsymbol{f}(\boldsymbol{y}(t), t)$. In `step_rk4`, arrays `double k1[__N__], k2[__N__],...` can be defined locally and passed to the function `compute_k1`, `compute_k2`, .... With the above function declarations, a Runge-Kutta step can be performed by calling

```
step_rk4(&y_t_tau, &y_t, tau, compute_f_tau);
```

When passed to `compute_k1`, `compute_k2`, ..., you can call `compute_f_tau` via

```
(*compute_f_tau_ptr)(...);
```

- `void write_y_t_to_file(FILE *file, const Y_T *y_t);`

  This function prints one line with $t$ and the components of $\boldsymbol{y}(t)$ to a file, i.e.

  ```
  <t> <y[0]> <y[1]>
  ```

  As you know from the lecture, `FILE` is defined in `stdlib.h` from the standard library.

## 4 Using a Makefile

Compiling projects consisting of several files manually can be tedious. Tools like *Makefile* are thus commonly used, to automate this process. Here we give a short example of a Makefile, which can be used to compile the project files `rk.c` (in this case, a library) and `oscillator.c` (containing the `main` function).

```
CC = gcc
CFLAGS = -std=c11 -Wall -Wextra -pedantic -O3
LIBS = -lm

all: oscillator

oscillator: oscillator.o rk.o
  $(CC) $(CFLAGS) -o oscillator oscillator.o rk.o $(LIBS)

oscillator.o: oscillator.c
  $(CC) $(CFLAGS) -c -o oscillator.o oscillator.c

rk.o: rk.c
  $(CC) $(CFLAGS) -c -o rk.o rk.c

clean:
  rm -f *.o oscillator
```

Makefiles are scripts with filename `Makefile` and typically placed in the root directory of the project. The script is run by executing `make [TARGET]` or simply `make` in the terminal (see `man make` for more information). In the above example executing `make` would run the `make`-target `all`, which consists of the target `oscillator`, which in turn consists of the targets `oscillator.o` and `rk.o`. `CC`, `CFLAGS` and

Produced with the ExerciseHandler

LIBS are variables, where CC is equivalent to the compiler command gcc, CFLAGS represents various compiler options, and LIBS includes the system's math library. Thus, when Makefile executes the lines $(CC) $(CFLAGS) ..., the gcc command is run to compile the specified source files in an appropriate way and ordering.

It is common to also include a target clean in a Makefile. The purpose of this target is to remove files previously generated via make [TARGET]. In the above example, executing make clean will delete all generated .o files, as well as the executable oscillator.

Produced with the ExerciseHandler