CSCI 235, Programming Languages, Python 2

Deadline: 16.11.2018, 21.00

Goal of this exercise is that

1. you will use Python for scientific computing,
2. that you have seen Runge-Kutta methods for solving differential equations numerically,
3. that you understand in principle what the convergence order of a numerical method is,
4. that you have a feeling of the difference in speed between Python and C^{++}.

- **Read the complete task before starting!**
- **Submit the final answer into Moodle as a single file myname.py. Additional information must be put in Python comments**
- **Answers that show evidence of lazyness, not reading the complete task, incomplete testing, may result in loss of credit!**

The *catenary* is the form that a chain (or a rope without stiffness) assumes, when it is fixed at two points. It is defined by the differential equation

$$y''(x) = \mu.\sqrt{1 + (y'(x))^2}. \tag{1}$$

The differential equation has an exact solution, namely

$$y(x) = \frac{\cosh(\mu.x)}{\mu} = \frac{e^{\mu.x} + e^{-\mu.x}}{2\mu}.$$

Because we know the exact solution, it can be used for testing numerical methods. We can approximate a solution, using Runge-Kutta, and compare it to the exact solution given above.

Runge-Kutta methods are a well-known family of numerical methods that are used for solving differential equations. Runge-Kutte methods approximate differential equations of form

$$y' = F(y).$$

Equation 1 is second order, but this problem can be solved by vectorization. We first list a few Runge-Kutta methods:

Euler's Method Euler's method is the simplest Runge-Kutta method: It is given by

$$\begin{array}{c} k = F(y) \\ \hline y(x+h) = y(x) + h.k \end{array}$$

Heun's Method: Heun's method is also a Runge-Kutta method.

$$\begin{array}{l} k_1 = F(x) \\ k_2 = F(x + h.k_1) \\ \hline y(x+h) = \frac{h}{2}(k_1 + k_2) \end{array}$$

Standard Runge-Kutta:

$$\begin{array}{l} k_1 = F(y) \\ k_2 = F(y + \frac{h}{2}k_1) \\ k_3 = F(y + \frac{h}{2}k_2) \\ k_4 = F(y + hk_3) \\ \hline y(x+h) = \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{array}$$

More methods can be found in the C^{++} -program.
In order to obtain a first-order method, Equation 1 has to be vectorized by defining $y_0(x) = y(x),\ y_1(x) = y'(x)$. Then one can use

$$F((y_0, y_1)) = (\ y_1,\ \mu.\sqrt{1 + (y_1)^2}\ ).$$

The exact solution is defined by

$$y(x) = (\ \frac{\cosh(\mu.x)}{\mu},\ \sinh(\mu.x)\ ) =$$

$$y(x) = (\ \frac{e^{\mu.x} + e^{-\mu.x}}{2\mu},\ \frac{e^{\mu.x} - e^{-\mu.x}}{2}\ ).$$

1. Download the C^{++} program **chain.cpp**, and **chain.py**. Rename **chain.py** into **yourname.py**. (If your name is very long, you may shorten the filename.)

   Implement the Runge-Kutta methods which are present in the C^{++} program, in the Python program. In C^{++}, the Runge-Kutta methods are implemented as templates. In Python, there is no need to do that, because Python is dynamically typed. It is all explained in the slides.

2. Make sure that you understand how function **approx**(h) relates to **main.h**. In C^{++}, a class **pair** is defined, in Python, we use **scipy.array**.

3. Make sure to test the function **approx**(h) very carefully. This can be done by comparing the printed errors to the errors printed by the C^{++} -program for different values of h, and for different Runge-Kutta methods. If you program correct, they will agree on 10 decimals at least.

   Note that in **main** and in **approx**, the Runge-Kutta method is called twice, so make sure that you always call the same method.

4. Now we can do some systematic measurements of the errors. Create (and submit) a table for $h = 10^{-3},\ h = 2.10^{-3},\ h = 4.10^{-3}$, for the methods Euler, Heun and standard Runge-Kutta. (This gives 9 entries in the table.)

5. Since there are continuing rumours that C^{++} is faster than Python, we are going to measure the difference.

   For the C^{++} program, time can be measured by calling `time ./chain` . The program will run, and after that, print the execution time.

   In python, one can `import timeit`, and type `timeit( approx, number = N ) / N` Try some number `N` that gives a reasonable time.

   Make sure that that C^{++} and Python run with the same h and same Runge-Kutta method. Compare the results.

   Give measurements for at least 3 different settings. How does C^{++} compare to Python, what is the relative difference?