INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS (1)

Objectives:

- Integrate a first-order ordinary differential equation numerically;
- Implement the forward Euler method;
- Identify the order of convergence of an integration scheme by analysing how the global truncation error changes with the timestep;
- Implement the Runge-Kutta 4 method;
- Use the function `solve_ivp` of the sub-module `scipy.integrate` to integrate a first-order ordinary differential equation;
- Implement the predictor-corrector (Heun's) method.

No list manipulation is allowed in this tutorial!

I. Introduction

In this tutorial, we focus on the following first-order Ordinary Differential Equation (ODE):

$$\begin{cases} \dot{x}(t) = t\left[x(t)^2 + 1\right], \ t \in [0,\, 1] \\ x(0) = 1, \end{cases} \tag{1}$$

whose exact solution is

$$x(t) = \tan\left(\frac{t^2}{2} + \frac{\pi}{4}\right). \tag{2}$$

Question 1: Define a function `exact_solution(t)` which returns the exact solution to the above ODE given by Eq. (2).

Question 2: Plot the function (make a readable plot).
You should note that the function is growing fast, this is why this Cauchy problem is an ideal test for the different methods of integration presented in the lecture notes.

Question 3: Define a function `derivative_ode(t,x)` which returns the time derivative of the solution to the above ODE given by the right-hand side of Eq. (1).

II. Forward Euler method

We start the tutorial with the forward Euler method.

Question 4: Define a function `euler(h, xi=1., ti=0., tf=1.)` which takes as an input the timestep h, implements the forward Euler method and returns two arrays `t_arr` (array of time values) and `x_arr` (array of values of the solution). The other arguments represent the initial condition `xi`, the initial time `ti` and the final time `tf`.

Question 5: For $h = 0.01$, plot on the same graph the numerical solution obtained from the forward Euler method (with symbols) and the exact solution (with a straight line). The plot must be understood by anyone.

Question 6: We define the relative global error at the end of the integration procedure as

$$\epsilon = \left| \frac{x(1) - x_{\mathrm{E}}(1)}{x(1)} \right|, \tag{3}$$

where $x(t)$ is the exact solution and $x_{\mathrm{E}}(t)$ the solution obtained from the forward Euler method. What is the value of ϵ (in %) for $h = 0.01$? How should you change h to decrease the error by a factor of 10 approximately?

Question 7: For 13 values of h regularly spaced in logarithmic scale between 10^{-5} and 10^{-1} (look at the function `logspace` from the NumPy package), compute ϵ. Plot ϵ as a function of h in a loglog plot. Make a readable plot.

Question 8: We recall that the order of convergence p of the integration scheme is defined as the integer p such that $\epsilon \sim h^p$. Extract the order of convergence p_{E} of the forward Euler method by simple reading of the plot obtained at the previous question.

III. Runge-Kutta 4 method

We now turn to the Runge-Kutta 4 method.

Question 9: Define a function `rk4(h, xi=1., ti=0., tf=1.)` which takes as an input the timestep h, implements the Runge-Kutta 4 method, and returns two arrays `t_arr` (array of time values) and `x_arr` (array of values of the solution).

Question 10: For $h = 0.01$, plot on the same graph the numerical solution obtained from the forward Euler method and the Runge-Kutta 4 method (with symbols) and the exact solution (with a straight line). The plot must be understood by anyone. Which solution is the best approximate of the exact solution?

Question 11: For 13 values of h regularly spaced in logarithmic scale between 10^{-5} and 10^{-1}, compute ϵ. Plot ϵ as a function of h in a loglog plot and extract the order of convergence p_{R} of the Runge-Kutta 4 method.

IV. Using SciPy

We now take advantage of the SciPy module.

Question 12: The package SciPy provides a function called `solve_ivp` which implements different methods to integrate ODEs. Look at the documentation to understand how the function works: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html#scipy.integrate.solve_ivp. Define a function `python_solver(xi=1., ti=0., tf=1.)` which uses the function `solve_ivp` to implement the 8$^{\text{th}}$ order Runge-Kutta method ***with a high level of accuracy***, and returns two arrays `t_arr` (array of time values) and `x_arr` (array of values of the solution).

Question 13: Plot on the same graph the numerical solution obtained from the 8$^{\text{th}}$ order Runge-Kutta method (with symbols) and the exact solution (with a straight line). The plot must be understood by anyone.

Question 14: What do you see on the graph? Can you explain why?
Hint: look at how time values are distributed.

Question 15: What is the value of ϵ (in %) for the 8$^{\text{th}}$ order Runge-Kutta method? How does it compare with the two previous methods?

V. Bonus: Predictor-corrector (Heun's) method

We eventually turn to the predictor-correct method (or Heun's method).

Question 16: Define a function `heun(h, xi=1., ti=0., tf=1.)` which takes as an input the timestep h, implements the predictor-correct method, and returns two arrays `t_arr` (array of time values) and `x_arr` (array of values of the solution).

Question 17: For 13 values of h regularly spaced in logarithmic scale between 10^{-5} and 10^{-1}, compute ϵ. Plot ϵ as a function of h in a loglog plot and extract the order of convergence p_{H} of the predictor-corrector method.