Assignment 3 - The Butterfly Effect

Numerical Methods (6E5X0) — 2023/2024

1 Goals

After successfully completing this assignment, you will be able to:

- Set up ODE equations as Python functions
- Use the SciPy ODE solver to solve your equations
- Implement various first-order and higher-order ODE solvers
- Verify your implementations
- Show the order of convergence of various solvers
- Perform a numerical analysis on the double-pendulum

To summarize, the ultimate goal of this assignment is that you build, verify and characterize your own ODE solver(s), and use them to solve a system of coupled ODEs.

2 General advice

This assignment mainly focuses on the implementation of the methods for ODE's that were discussed during the lectures. If you cannot figure out how to make a solver for systems of ODE's, you can still proceed by making a higher-order ODE solver for single ODE's to get a partial score. Similarly, if you cannot make a higher-order ODE solver, you can still try to perform certain analysis using the `scipy.integrate.solve_ivp` ODE solver. Bottom line: the order of the questions should guide you through the process, but if you really get stuck you may still be able to (partially) solve the subsequent questions.

3 Questions

We consider a third-order chemical reaction performed in a batch reactor:

$$\mathrm{A} \xrightarrow{\mathrm{k_3}} \mathrm{B} \quad (\text{with } k_3 = 0.01\ \mathrm{L^2\,mol^{-2}\,s^{-1}})$$

1. Give the ODE that describes the concentration C_A over time, i.e. $\frac{dC_A}{dt}$. Create a Python function that uses t and C_A as input arguments, and returns the time-derivative as an output.

2. Determine the analytical solution of this ODE. Create a Python function that computes the analytical solution, using a vector t, an initial concentration c_0 and a reaction rate constant k_3 as input arguments.

3. Create a script that does the following, using the functions developed above:

   - Use the `scipy.integrate.solve_ivp` function to solve your ODE function for a reasonable time span.
   - Plot the results as a function of time, along with the analytical solution.
   - Show, as a function of time, the difference between the analytical and numerical solution.
   - Analyze your results.

We will now continue to create a function that behaves similarly to the `scipy.integrate.solve_ivp` function (or any other ODE solver for that matter), i.e. using a similar type of input and output, but which is using an explicit first-order Euler solution technique.

4. Create a new function: `first_order_euler(fun, tspan, y0)`, where `fun` is a function handle to the ODE function, `tspan` is the time span and `y0` the initial condition. Decide for yourself whether you want to supply more arguments. Verify your function by solving the ODE from Q1 with a small time step (e.g. $\Delta t = 0.001$ s), and compare with the analytical solution.

5. Characterize the implemented method by computing the error ε between the numerical and analytical solution using various amounts of time-steps N_t, and determine the maximum time step size that still allows a stable solution. Present the values of ε and N_t in a clear way. What is the rate of convergence?

You have now created your own ODE solver, and verified and characterized it's implementation. It is likely that you have set up this solver to work for a single ODE only, but often we are interested in multiple coupled ODEs. Let's make the reaction exothermic: solve additionally to the ODE of Q1 an energy balance, and use an Arrhenius-based reaction rate constant:

$$\begin{aligned}\frac{dC_A}{dt} &= \mathcal{R} = -k_3 C_A^3 \\ \rho C_p \frac{dT}{dt} &= -\mathcal{R}(-\Delta H_r) \\ k_3 &= A \exp\left(-\frac{E_A}{RT}\right)\end{aligned}$$

Use: $A = 3.1$, $\rho = 1.25\ \mathrm{kg\,m^{-3}}$, $C_p = 1200\ \mathrm{J\,kg^{-1}\,K^{-1}}$, $\Delta H_r = -184\ \mathrm{kJ\,mol^{-1}}$ and $E_A = 25.8\ \mathrm{kJ\,mol^{-1}}$. For initial conditions, use $c_0 = 10\ \mathrm{mol\,m^{-3}}$ and $T_0 = 313$ K.

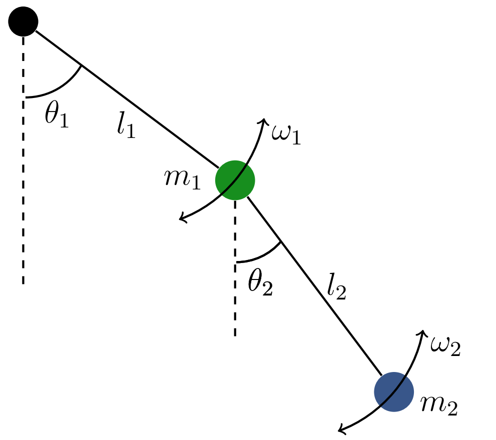
6. First, we want to make sure that our ODE function is implemented correctly. Create a Python function similar to Q1, implementing the concentration and energy balances as a system of ODEs, and use the generic `solve_ivp` function to solve for a reasonable time span.

7. Create a new first-order Euler solver that allows to solve multiple ODEs (`first_order_euler_system (fun, tspan, y0)`).

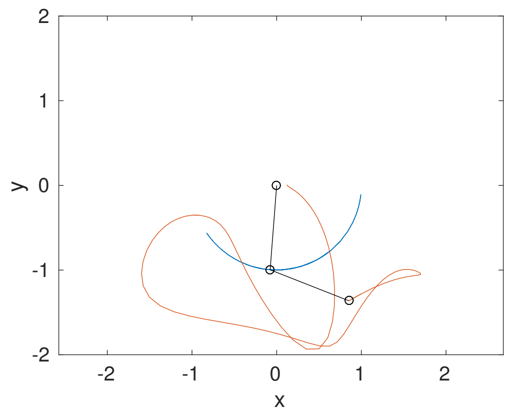
8. Building on `first_order_eulerSystem`, create 1 or more higher-order solvers for systems of ODEs. If you like a challenge: create a master function that allows selection of the solver via input parameter (or likewise), but this master function is not mandatory (you will earn respect and skills, not points.).

9. Characterize the higher-order solver using the *single ODE* and analytical solution as we did in question 1-3 and 5.

Now we have created our very own numerical solvers that are capable of solving systems of ODEs. Why don't we put them to the test by solving a more dynamic system: the double pendulum. The double pendulum (see Figure 1) is the classical example of a chaotic system, meaning that a tiny change in the initial conditions will ultimately result in completely different behaviour of the swinging motion of the pendula. The sensitivity of a system to its initial conditions is also known as *the butterfly effect*.



(a) Schematic overview of the double pendulum system



(b) Example trajectory for $\theta_1 = \frac{7}{180}\pi$ rad and $\theta_2 = \pi$ rad (other parameters unity)

Figure 1

The motion of the system, consisting of two masses m_1 and m_2 (in kg) connected by a rod of length l_1 and l_2 (in m) is described by a system of 4 ODEs:

$$\begin{aligned}
\frac{d\theta_1}{dt} &= \omega_1 \\
\frac{d\omega_1}{dt} &= \frac{m_2 l_1 \omega_1^2 \sin\Delta \cos\Delta + m_2 g \sin\theta_2 \cos\Delta + m_2 l_2 \omega_2^2 \sin\Delta - (m_1 + m_2) g \sin\theta_1}{(m_1 + m_2) l_1 - m_2 l_1 \cos^2\Delta} \\
\frac{d\theta_2}{dt} &= \omega_2 \\
\frac{d\omega_2}{dt} &= \frac{-m_2 l_2 \omega_2^2 \sin\Delta \cos\Delta + (m_1 + m_2)\left(g \sin\theta_1 \cos\Delta - l_1 \omega_1^2 \sin\Delta - g \sin\theta_2\right)}{(m_1 + m_2) l_2 - m_2 l_2 \cos^2\Delta}
\end{aligned}$$

Here, $\theta_{1,2}$ is the angle of the respective mass with the vertical axis in rad and $\omega_{1,2}$ is the angular momentum of the masses in rad s^{-1}. The parameter $\Delta = \theta_2 - \theta_1$ and $g = 9.81$ m s^{-2} represents the gravitational acceleration.

10. Set up the ODE system for the double pendulum and solve with your favorite solver(s). Analyze your observations.

11. Analyze the energy budget for this system: compute the kinetic energy and potential energy and check for conservation. Include in your analysis the results of different solvers, amounts of time steps, etc. Feel free to explore the system further and report your findings.

4 Hints

- Q4: Note that you will have to decide whether the `tspan` contains *all time steps* on which you want to compute the solution (e.g. `tspan = 0:0.01:10`), or you want `tspan` to be a vector with just t_0 and t_{end}. The first case requires you to compute a Δt in the solver, the second case would need an additional argument for either the time step Δt or the amount of time steps N_t.

- Q7: Make use of NumPy's vector-computations. Carefully assess which variables should become a vector instead of a scalar. This also makes the algorithm general, allowing larger systems of ODEs.

- Q11: The potential energy of the system and kinetic energy of the first bob are straightforward (how can you verify these whether you implemented them correctly?). The kinetic energy of the second bob is slightly more complicated. It is computed as:

$$E_{\text{kin},2} = \frac{1}{2} m_2 \left(\omega_1^2 l_1^2 + \omega_2^2 l_2^2 + 2\omega_1 l_1 \omega_2 l_2 \cos(\theta_1 - \theta_2)\right);$$

References