L4: Ordinary differential equations

1 Brief Introduction

Ordinary differential equations (ODEs) can be used to describe a large number of physical phenomena. Being able to handle ODEs is also very relevant when solving partial differential equations. Like integrals and non-linear equations, ODEs are often difficult or impossible to solve analytically, so numerical methods are used instead.

- ODEs can be divided into two main types: *initial value problems* and *boundary value problems*. In this lab we will only look at initial value problems. Such a problem can be written as

$$\begin{aligned} y'(t) &= f(t, y), \quad t > a \\ y(a) &= y_0, \end{aligned} \tag{1}$$

  where a and y_0 are known parameters and $f(t, y)$ is a known function.

- The basic principle when ODEs are solved numerically is that the interval on which you want to solve the problem is divided into points (discretized) and then the solution is approximated according to some principle at these points. The distance from one point to the next is called *step length*. This is similar to the discretization we employed when numerically integrating a function on an interval.

- A so-called *order* of accuracy is linked to the numerical methods used to solve ODEs (similar to the methods for calculating integrals). Whether a certain method is effective and "good" is determined among other things (but not exclusively) by this concept. It is therefore important to understand what the term stands for. In the lab, we will empirically investigate and illustrate this concept for three different numerical methods: Euler's method, Heun's method and classical Runge-Kutta (sometimes called Runge-Kutta 4 or RK4).

- Another important concept is *stability*. In the lab we will investigate the concept of numerical stability and how it is connected to so-called stiff differential equations. You will also be able to see the difference in how explicit and implicit methods work for that type of problem.

Start by downloading the files for this lab: `Eulerdemo.py`, `accuracydemo.py` and `stabdemo.py`.

Note! If you are using an IDE, you may need to set it up so that interactive figures can be created. For example in Spyder: Preferences → IPython Console → Graphics → Backend → Automatic.

2 Euler's method

You will now use a demo program to look at how Euler's method works for the non-linear ODE

$$\begin{aligned} y'(t) &= \cos(yt), \quad 0 \leq t \leq 2\pi, \\ y(0) &= 0. \end{aligned} \tag{2}$$

Other names for the same method are explicit Euler and forward Euler. The method starts with the initial value at $t = 0$ and then steps forward in time one step length at a time.

1. Run the program `Eulerdemo` with step lengths 0.4, 0.2 and 0.1. Pay attention to the effect of the choice of the step length on the result and accuracy.

3 Accuracy

There are many other and better methods for solving ODEs than Euler's method, but all are based on similar basic principles. You will now study the order of accuracy for Euler's method, Heun's method and classical Runge-Kutta.

2. Run the program `accuracydemo` and try to understand the figure drawn. The program solves a linear ODE and plots the error as a function of step length for Euler's method, Heun's method and classical Runge-Kutta. The slope of each curve corresponds to the order of accuracy of the methods, read the printed text. Note that the axes have a logarithmic scaling. There seems to be a limit to how accurate a solution can be calculated; think about why. At approximately what error is this limit?

3. When solving ODEs, you want the method to solve the problem with a certain accuracy, a certain error. Here you will look at how this is connected to the order of accuracy. In the figure from the previous task (i.e. the program `accuracydemo`), select an error tolerance and press `Get statistics`. Start with error tolerance 10^{-4} (can be written `0.0001` or `1e-4`) and study the information that is printed. The program estimates the step length required to achieve the given tolerance and calculates the corresponding number of steps and runtime for each method. Pay attention to the differences in step length and number of calculation steps for the different methods. Also be aware that the methods take different amounts of time per calculation step.

4 Stability

To investigate the stability of a method, the differential equation

$$\begin{aligned} y'(t) &= \lambda y, \quad t \geq 0, \\ y(0) &= y_0, \end{aligned} \tag{3}$$

is often used, where λ is a negative constant; this equation is sometimes called *the test equation*. The exact solution to (3) is $y(t) = y_0 e^{\lambda t}$. We will now investigate the stability of Euler's method (explicit Euler) and Euler's implicit method, which is usually called implicit Euler or backward Euler. Both methods have the same order of accuracy, but they differ when it comes to stability. For explicit Euler, one can theoretically show that there is an upper limit for how large a step length h can be chosen, a so called stability condition. For the ODE (3), this condition is $h < 2/|\lambda|$. For implicit Euler, there is no restriction on the step length to achieve a stable solution, i.e. it is unconditionally stable.

4. Run the program `stabdemo`. Set the step length $h = 0.05$ and $\lambda = -10$ and compare the methods. Change λ to, for example, -100. Study the effect it has on the solutions.

5. Examine what the solution looks like with explicit Euler if the step length is chosen to be just below, just above and exactly on the stability limit. For example, use

$\lambda = -20$ and the step lengths 0.09, 0.1 and 0.11. With this λ, the stability condition is $h < 0.1$. Try to understand from the plots what is meant by the concept of stability? What happens to the solution for the different choices of step length?

5 SciPy functions for ordinary differential equations

For solving ODEs in Python, the SciPy function `solve_ivp` can be used. In the function, a number of methods for initial value problems (IVPs) are implemented: which method is used is determined by the arguments to the function. We will use the RK45 method here. This method is an *adaptive* Runge-Kutta method, which means that the step length in each step is chosen such that a certain accuracy is obtained. In this way, the step length is adapted to what the function looks like, similar to adaptive methods for calculating integrals. The function is called as follows:

```
1  import scipy.integrate as ode
2  SOL = ode.solve_ivp(fun,tspan,y0,method='RK45',args=args)
```

Here `fun` is a Python function `fun(t,y,...)` that defines the right-hand side of the ODE, `tspan` a tuple with starting and stop time and `y0` the initial data. The argument `args` is used to pass additional parameters to the right method. The solution and other useful information are stored in the variable `SOL`. For further information and examples of how `solve_ivp` is used, see the SciPy documentation at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`.

As an example problem, we will study a mathematical model for epidemics. Denote the number of individuals susceptible to a disease at time t by $S(t)$ ("susceptible"), the number of infected ("infectious") that can spread the infection by $I(t)$ and the number of immune ("recovered") for $R(t)$. The model is given by the ODE system

$$\begin{aligned} S'(t) &= bN - dS - \beta\frac{I}{N}S - vS, \\ I'(t) &= \beta\frac{I}{N}S - uI - dI, \\ R'(t) &= uI - dR + vS, \end{aligned} \tag{4}$$

where N is the size of the population, b corresponds to the proportion of immigrants and newborns, d is the proportion of dead or emigrated, β is the number of contacts an individual has per unit of time that cause infection, u is the proportion of sick that become immune per time unit and v is the proportion of susceptibles that become vaccinated per time unit.

The task is to simulate an epidemic from the time $t = 0$ to 60 days using the SciPy function `solve_ivp`. Let $\beta = 0.3$, $u = 1/7$, which corresponds to an infection time of 7 days, and $v = 0$, i.e. no vaccination. With $b = 0.002/365$ and $d = 0.0016/365$, approximately the same proportion of births and deaths per day is obtained as in Sweden. Suppose that at time $t = 0$, 5 people are infected and none are immune. Note that $S(t) + I(t) + R(t) = N$, i.e. everyone in the population always belongs to one of the groups. Let $N = 1000$. Write the program so that you can easily change all parameters in one place.

6. Start by defining the right-hand side function. The function header should be:

```
1   def f(t,y,N,b,d,beta,u,v):
```

In this case `y` is a Numpy array with three elements corresponding to S, I and R and the function should return a Numpy array with three elements corresponding to S', I' and R'. Note that the independent variable, in this case t, must be included as an input parameter even if it is not used in the right-hand side function.

7. Solve the ODE using `solve_ivp` and the RK45 method. Plot the solution, i.e. plot $S(t)$, $I(t)$, and $R(t)$ as a function of t. Does the result look reasonable?

8. When you have got your program working, you can try adding a vaccination program, e.g. by setting $v = 0.01$. Also vary other parameters and investigate what happens. For example, you can reduce the duration of illness to 3 days, i.e. set $u = 1/3$, and increase or decrease β.

Hint: With the argument `max_step` you can set an upper limit on how big steps the adaptive method can take. Add `max_step = 1` in the function call of `solve_ivp` to ensure that the state is calculated at least once a day. This way, a smoother plot is obtained.