Ordinary Differential Equations (ODE)

Prepared as part of MOD510 Computational Engineering and Modeling

Sep 5, 2022

1 ODE Notebook

Learning objectives:

- being able to implement an ODE solver in python
- quantify numerical uncertainty
- test different methods and have basic understanding of the strength and weaknesses of each method

1.1 Runge-Kutta Methods

The 2. order Runge-Kutta method is accurate to h^2, with an error term of order h^3

The 2. order Runge-Kutta:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\ y_{n+1} &= y_n + k_2 \end{aligned} \tag{1}$$

The Runge-Kutta fourth order method is one of he most used methods, it is accurate to order h^4, and has an error of order h^5.

The 4. order Runge-Kutta:

$$
\begin{aligned}
k_1 &= hf(y_n, t_n) \\
k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\
k_3 &= hf(y_n + \frac{1}{2}k_2, t_n + h/2) \\
k_4 &= hf(y_n + k_3, t_n + h) \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad (2)
\end{aligned}
$$

2 Exercise: Implement the Runge-Kutta method

In the following we are going to model a contaminated lake using a mixing tank. As an example we are going to use Norways largest lake, Mjosa, see figure 1.

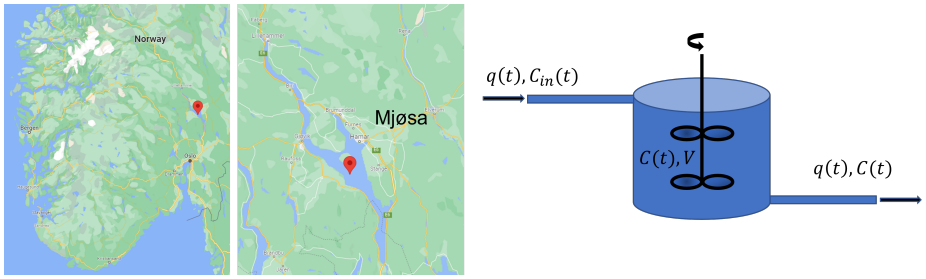

Figure 1: The location of Mjosa and a mixing tank. The mixing tank assumes that at all times the concentrations inside the tank is uniform.

The volume of Mjosa is, V =56 km^3, and the discharge is q = 321 m^3/s=27.734·10^{6}m^3/day. We will assume that some contaminant are present *uniformly* in the lake, and fresh water is flowing into Mjosa. Applying mass balance to the system, and assuming that the flow pattern in Mjosa is such that the contaminant is distributed uniformly at all times, the following equation should hold

$$V\frac{dC(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C(t)\right]. \qquad (3)$$

Assume that the initial concentration of the contaminant is $C_0 = 1$, and assume that water flowing into contains no contaminant, we have the boundary conditions $C_{\text{in}}(t) = 0$, $C(0) = 1$. In this case equation (3) is

$$\frac{dC(t)}{dt} = -\frac{1}{\tau}C(t), \qquad (4)$$

where $\tau \equiv V/q$. The analytical solution is simply

$$C(t) = e^{-t/\tau}. \tag{5}$$

Part 1. In the following you are going to first implement a *general* solver, then you are to test it on equation (5). The solution of ODE equations are based on solving a generic equation of the form

$$\frac{dy(t)}{dt} = f(y,t). \tag{6}$$

Thus the solver should take in *as argument*, the right hand side, $f(y,t)$, starting values y_0, the start time t_0 and end time t_f.

Complete the code below

```
import matplotlib.pyplot as plt
import numpy as np

#global parameters
c_in=0
c0=1
q=321*24*60*60*365 #m^3/year
V=56*1000**3    #m^3
tau=V/q        #day

def func(y,t):
    """
    the right hand side of ode
    """
    return ...

def rk4_step(func,y,t,dt):
    """
    t : time
    dt : step size (dt=h)
    func : the right hand side of the ode
    """
    ...
    return

def rk2_step(func, y, t, dt):
     """
    t : time
    dt : step size (dt=h)
    func : the right hand side of the ode
    """
    return

def ode_solv(func,y0,dt,t0,t_final):
```

```
    y=[];t=[]
    ti=t0
    y_old=y0
    while(ti <= t_final):
        t.append(ti); y.append(y_old)
        y_new = y_old+rk4_step(func,y_old,ti,dt) # or rk2_step
        y_old = y_new
        ti   += dt
    return np.array(t),np.array(y)
```

2.1 Solution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
c_in=0
c0=1
q=321*24*60*60*365 #m^3/year
V=56*1000**3    #m^3
tau=V/q        #day
def rk2_step(func,y,t,dt):
    """
    Integrates from time t to t+h using RK2
    func = right hand side of ODE
    y = solution vector
    t = current time
    h = step size
    """
    k1=dt*func(y,t)
    k2=dt*func(y+0.5*k1,t+dt*0.5)
    return k2

def rk4_step(func,y,t,dt):
    """
    Integrates from time t to t+h using RK4
    func = right hand side of ODE
    y = solution vector
    t = current time
    dt = step size
    """
    k1=dt*func(y,t)
    k2=dt*func(y+0.5*k1,t+0.5*dt)
    k3=dt*func(y+0.5*k2,t+0.5*dt)
    k4=dt*func(y+k3,t+dt)
    return (k1+2*k2+2*k3+k4)/6

def ode_solv(func,y0,dt,t0,tf):
    """
```

```
    solves a set of ODE with known initial conditions
    from time ti to t_final
    ti = start time
    y0 = initial conditions at time t0
    tf = end time
    dt = step size
    """
    y=[];t=[]
    ti=t0
    y_old=y0
    while(ti <= tf):
        t.append(ti); y.append(y_old)
        y_new = y_old+rk4_step(func,y_old,ti,dt) # or rk2_step
        y_old = y_new
        ti   += dt
    return np.array(t),np.array(y)

def func(y,t):
    return 1/tau*(c_in-y)
dt=.1
t,y=ode_solv(func,c0,dt,0,20)
print('Numerical Error ', np.abs(y[-1]-np.exp(-t[-1]/tau)))

plt.plot(t,y,'*',label='numerical')
plt.plot(t,np.exp(-t/tau),label='analytical')
plt.xlabel('Time [years]')
plt.legend()
```

Part 2.

1. How much time does it take for the contaminant to drop to 10% of its original value?
2. Is this model a good model for the cleaning of Mjosa?
3. Does the numerical error scales as expected

2.2 Solution

Part 2.

1. From the figure we see that it takes about 12.7 years (can also be estimated analytically $t = -\tau \ln 0.1 \simeq 12.73$ years)
2. Most likely the model is to simplistic, but there is a good chance that to always assume that the mixing is uniformly will give an upper bound of the cleaning (assuming that the contaminant does not adsorb or is taken up by the living organism)

3. Yes, changing dt from 1 to 0.1, reduces the error by a factor of 1000 in the case of `rk2_step` and 100000 in the case of `rk4_step`.

Exercise 1: Adaptive step size - Runge-Kutta Method

In this exercise you are going to improve the algorithms above by choosing a step size that is not too large or too small. This will serve two purposes i) *greatly* enhance the efficiency of the code, and ii) ensure that we find the correct numerical solution that is *close enough* to the true solution. We are going to use the following result from the compendium [1] (to get a good understanding it is advised to derive them)

$$|\epsilon| = \frac{|\Delta|}{2^p - 1} = \frac{|y_1^* - y_1|}{2^p - 1}, \tag{7}$$

$$dt' = \beta dt \left| \frac{\epsilon'}{\epsilon} \right|^{\frac{1}{p+1}}, \tag{8}$$

$$\hat{y_1} = y_1 - \epsilon = \frac{2^p y_1 - y_1^*}{2^p - 1}, \tag{9}$$

where ϵ' is the desired accuracy. β is a safety factor $\beta \simeq 0.8, 0.9$, and you should always be careful that the step size do not become too large so that the method breaks down. This can happens when ϵ is very low, which may happen if $y_1^* \simeq y_1$ and/or if $y_1^* \simeq y_1 \simeq 0$.

Part 1. Use the equations above, and implement an adaptive step size algorithm for the 4. order Runge-Kutta methods. Use the Mjosa example above to test your code. It might be a good idea to use a safety limit on the step size `min(dt*(tol/toli)**(0.2),dt_max)`, where `dt_max` is the maximum step size you allow. The tolerance should be calculated as $\epsilon' = atol + |y|rtol$, where 'atol' is the absolute tolerance and 'rtol' is the relative tolerance. A sensible choice would be to set 'atol=rtol' (e.g. $= 10^{-5}$).

Below is some code to help you get started

```
def rk_adpative(func,y0,t0,tf,rel_tol=1e-5,abs_tol=1e-5,p=4):
    y=[]
    t=[]
    ti=t0
    y.append(y0)
    t.append(ti)
    dt=1e-2 # start with a small step
    while(ti<=tf):
        y_old=y[-1]
        EPS=np.abs(y_old)*rel_tol+abs_tol
        eps=10*EPS
        while(eps>EPS): # continue while loop until correct dt
```

```
            DT=dt
            y_new =  ....  # one large step from t to t+dt
            y1    =  ....  # and two small steps - from t -> t+dt/2
            y2    =  ....  # and from t+dt/2 to t + dt
            eps   =  ....  # estimate numerical error
            dt    =  ....  # calculate new time step

        y.append( ... )
        ti=ti+DT # important to add DT not dt
        t.append(ti)
    return np.array(t),np.array(y) # cast to numpy arrays
```

Part 2.

1. How many steps do you need to get a reasonable solution?
2. Is the numerical error what you expect?

2.3 Solution

```
def rk_adpative(func,y0,t0,tf,rel_tol=1e-5,abs_tol=1e-5):
    """
    solves an ODE with known initial conditions
    from time ti to t_final
    ti = start time
    y0 = initial conditions at time t0
    tf = end time
    rel_tol relative toleranse
    abs_tol absolute toleranse
    p order of numerical method, could set p=2 and
    change rk4_step to rk2_step
    """
    p=4
    y=[]
    t=[]
    ti=t0
    y.append(y0)
    t.append(ti)
    dt=1e-2 # start with a small step
    while(ti<=tf):
        y_old=y[-1]
        EPS=np.abs(y_old)*rel_tol+abs_tol
        eps=10*EPS
        while(eps>EPS): # find correct dt
            DT=dt
            # one large step from t to t+dt
            y_new = y_old + rk4_step(func,y_old,ti,DT)
```

```
            # and two small steps - from t -> t+dt/2
            y1    = y_old + rk4_step(func,y_old,ti,DT*0.5)
            # and from t+dt/2 to t + dt
            y2    = y1 + rk4_step(func,y1,ti+0.5*DT,DT*0.5)
            eps   = np.abs(y2-y_new)/(2**p-1) # estimate numerical error
            dt = 0.9*DT*(EPS/eps)**(1/(p+1))  # calculate new time step
        y.append((2**p*y2-y_new)/(2**p-1))
        ti=ti+DT # important to add DT not dt
        t.append(ti)
    return np.array(t),np.array(y) # cast to numpy arrays

xa,ya=rk_adpative(func,c0,0,20)
print('Numerical Error adaptive ', np.abs(ya[-1]-np.exp(-xa[-1]/tau)))
```

Part 2.

1. Only 7-8 steps are needed.
2. Changing setting `rel_tol=0`, and `abs_tol=1e-5` (or any other value) the numerical solution is closer to the analytical solution than the `abs_tol` value. This is partly due to the improved estimate of the solution where we use $\hat{y_1} = y_1 - \epsilon = (2^p y_1 - y_1^*)/(2^p - 1)$, and not y_1 as our final estimate.

Exercise 2: Solving a set of ODE equations

What happens if we have more than one equation that needs to be solved? If we continue with our current example, we might be interested in what would happen if we had multiple tanks in series. This could be a very simple model to describe the cleaning of drinking water infiltrated by salt water (a typical challenge in many countries) by injecting fresh water into it. Assume that the lake was connected to two nearby fresh water lakes, as illustrated in figure 2. The weakest part of the model is the assumption about complete mixing, in a practical situation we could enforce complete mixing with the salty water in the first tank by injecting fresh water at multiple point in the lake. For the two next lakes, the degree of mixing is not obvious, but salt water is heavier than fresh water and therefore it would sink and mix with the fresh water. Thus if the discharge rate was slow, one might imaging that a more or less complete mixing could occur. Our model then could answer questions like, how long time would it take before most of the salt water is removed from the first lake, and how much time would it take before most of the salt water was cleared from the whole system? The answer to these questions would give practical input on how much and how fast one should inject the fresh water to clean up the system. If we had data from an actual system, we could compare our model predictions with data from the physical system, and investigate if our model description was correct.

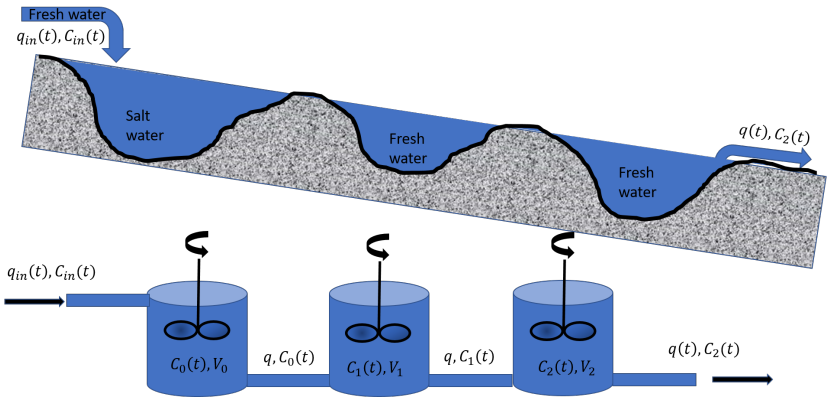

Figure 2: A simple model for cleaning a salty lake that is connected to two lakes down stream.

For simplicity we will assume that all the lakes have the same volume, V. The governing equations follows as before, by assuming mass balance:

$$\begin{aligned} C_0(t+\Delta t)\cdot V - C_0(t)\cdot V &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t, \\ C_1(t+\Delta t)\cdot V - C_1(t)\cdot V &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t, \\ C_2(t+\Delta t)\cdot V - C_2(t)\cdot V &= q(t)\cdot C_1(t)\cdot \Delta t - q(t)\cdot C_2(t)\cdot \Delta t. \end{aligned} \tag{10}$$

Taking the limit $\Delta t \to 0$, we can write equation (10) as:

$$V\frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right], \tag{11}$$

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right], \tag{12}$$

$$V\frac{dC_2(t)}{dt} = q(t)\left[C_1(t) - C_2(t)\right]. \tag{13}$$

Show that the analytical solution is:

$$\begin{aligned} C_0(t) &= C_{0,0}e^{-t/\tau} \\ C_1(t) &= C_{0,0}\frac{t}{\tau}e^{-t/\tau} \\ C_2(t) &= \frac{C_{0,0}t^2}{2\tau^2}e^{-t/\tau}. \end{aligned} \tag{14}$$

The numerical solution follows the exact same pattern as before if we introduce a vector notation.

$$\begin{aligned} \frac{d}{dt}\begin{pmatrix} C_0(t) \\ C_1(t) \\ C_2(t) \end{pmatrix} &= \frac{1}{\tau}\begin{pmatrix} C_{\text{in}}(t) - C_0(t) \\ C_0(t) - C_1(t) \\ C_1(t) - C_2(t) \end{pmatrix}, \\ \frac{d\mathbf{C}(t)}{dt} &= \mathbf{f}(\mathbf{C}, t). \end{aligned} \tag{15}$$

Part 1.

1. Extend the code in the previous exercises to be able to handle vector equations - note that if you have consistently used Numpy arrays you should actually be able to run your code without any modifications! (not the Richardson extrapolation algorithm)

2. Solve the set of equations in equation (15), and compare with the analytical solution

2.4 Solution

```
def fm(c,t): # rhs of tanks in series
    c_in=0
    rhs=[]
    rhs.append(c_in-c[0])
    rhs.append(c[0]-c[1])
    rhs.append(c[1]-c[2])
    return np.array(rhs)

# initial values
vol=1;q=1;c_into = 0; c_init = [1,0,0]
t_final=10 # end of simulation
dt=1e-2
#note same solver as before
t_vec,f_vec=ode_solv(fm,c_init,dt,0,t_final)

f_an = []
f_an.append(c_init[0]*np.exp(-t_vec))
f_an.append(c_init[0]*t_vec*np.exp(-t_vec))
f_an.append(c_init[0]*0.5*t_vec*t_vec*np.exp(-t_vec))

symb = ['-p','-v','-*','-s']
for i in range(len(c_init)):
    legi = '$\hat{C}_'+str(i)+'(\\tau)$'
    plt.plot(t_vec, f_vec[:,i], '-', label=legi,lw=4)
    plt.plot(t_vec, f_an[i], '--', color='k')
plt.plot(0,0 , '--', color='k',label='analytical')
plt.legend(loc='upper right', ncol=1)
#plt.ylim([0,50])
plt.grid()
plt.xlabel('Time')
plt.ylabel('Concentration')
```

3 Exercise: Adaptive method for a general ODE

Part 1. In this exercise we ask you to extend your implementation of the Richardson extrapolation to also be valid if the right hand side is a vector. We will not give the solution to this exercise, but rather tell you exactly how to do it and then you can try for yourself. The following recipe applies to the suggested solution (see separate pdf document)

1. There are only minor changes to the code, first we need to consider `EPS=np.abs(y_old)*rel_tol+abs_tol`. This expression is ambiguous, because `y_old` is a vector and `EPS` should be a single number. We suggest to simply replace the absolute value with the norm $\sqrt{y_0^2 + y_1^2 + \cdots}$, which can be achieved by `EPS=np.linalg.norm(y_old)*rel_tol+abs_tol`
2. The same also applies to the line `eps = np.abs(y2-y_new)/(2**p-1)`, and this should be changed to `eps = np.linalg.norm(y2-y_new)/(2**p-1)`
3. Make the suggested changes and test your code on equation (15)

4 Exercise: Second order equations

Test your solver on the following equation

$$xy''(x) + 2'(x) + x = 1, \tag{16}$$

where the initial conditions are $y(1) = 2$, and $y'(1) = 1$. The analytical solution is

$$y(x) = \frac{5}{2} - \frac{5}{6x} + \frac{x}{2} - \frac{x^2}{6}. \tag{17}$$

4.1 Solution

First we need to rewrite the equations to the general form

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \mathbf{f}(\mathbf{y}, \mathbf{x}). \tag{18}$$

We do this by defining $Z_0 = y$, and $Z_1 = y'(x) = dZ_0/dx$, hence $dZ_1/dx = y''$. Then it follow from equation (16)

$$\frac{dZ_1}{dx} = \frac{1 - 2y' - x}{x} = \frac{1 - 2Z_1}{x} - 1. \tag{19}$$

We can then write the following equation on vector form

$$\frac{d}{dx}\begin{pmatrix} Z_0(x) \\ Z_1(x) \end{pmatrix} = \begin{pmatrix} Z_1 \\ \frac{1-2Z_1}{x} - 1. \end{pmatrix}. \tag{20}$$

The initial condition is $\mathbf{Z}(1) = [Z_0(1), Z_1(1)] = [y(1), y'(1)] = [2, 1]$. Below is an implementation

```
def f2(y,t):
    return np.array([y[1],1/t-2*y[1]/t-1])

def analytical(x):
    return 5/2-5/(6*x)+x/2-x*x/6

y0=np.array([2,1])
x,y=ode_solv(f2,y0,1e-1,1,10)

plt.plot(x,y[:,0],'*',label='numerical')
plt.plot(x,analytical(x),'-',label='analytical')
```

References

[1] Aksel Hiorth. *Computational Engineering and Modeling.* https://github.com/ahiorth/CompEngineering, 2019.