Ordinary Differential Equations (ODE)

Prepared as part of MOD510 Computational Engineering and Modeling

Sep 16, 2021

1 ODE Notebook

Learning objectives:

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

Exercise 1: Runge-Kutta Methods

The Euler method only have an accuracy of order h, and a global error that do not go to zero as the step size decrease. The 2. order Runge-Kutta method is accurate to h^2.

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 4. order Runge-Kutta: 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.

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

In the following we are going to solve the CSTR equation:

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

Seawater contains about 35 gram salt/liter fluid, if we assume that the fresh water contains no salt, we have the boundary conditions $C_{\text{in}}(t) = 0$, $C(0)$ =35gram/l. Complete the code below, and test it both with the Runge-Kutta 2. and 4. order method:

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

def fm(c_old,c_in):
    return ...

def rk4_step(c_old, c_in, h):
    ...
    return

def rk2_step(c_old, c_in, h):
    ...
    return
# The following code is a suggestion, most likely it can be done more efficiently
def ode_solv(c_into,c_init,t_final,h):
    f=[];t=[]
    c_in  = c_into #freshwater into tank
    c_old = c_init #seawater present
    ti=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        c_new = rk4_step(c_old,c_in,h) # or rk2_step
        c_old = c_new
        ti   += h
    return t,f
# rest of code is to make a figure
```

```
# choose some step sizes to investigate
h = [0.4,0.8,1.6,2.8]
# initial values
vol=1000;q=1;c_into = 0; c_init = 35
tau=vol/q;t_final=10 # end of simulation in min
t=[];f=[]
for dti in h:
    ti,fi = ode_solv(c_into,c_init,t_final,dti)
    t.append(np.array(ti));f.append(fi)
f_an = c_init*np.exp(-np.array(t[0]))
symb = ['-p','-v','-*','-s']
fig = plt.figure(dpi=150)
for i in range(0,len(h)):
    plt.plot(tau*t[i], f[i], symb[i], label='$\Delta t =$'
             +str(h[i]*tau)+" min")
plt.plot(tau*t[0], f_an, '-', color='k',label='analytical')
plt.legend(loc='upper right', ncol=1)
plt.ylim([0,50])
plt.grid()
plt.xlabel('Time [min]')
plt.ylabel('Concentration')
plt.show()
```

Exercise 2: 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 *greatly* enhance the efficiency of the code. We are going to use the following result from the compendium (to get a good understanding it is advised to derive them, see the compendium [1])

$$|\epsilon| = \frac{|\Delta|}{2^p - 1} = \frac{|y_1^* - y_1|}{2^p - 1}, \tag{4}$$

$$h' = \beta h \left| \frac{\epsilon}{\epsilon_0} \right|^{\frac{1}{p+1}}, \tag{5}$$

$$\hat{y_1} = y_1 - \epsilon = \frac{2^p y_1 - y_1^*}{2^p - 1}, \tag{6}$$

where 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 $y_1^* \simeq y_1 \simeq 0$.

Use the equations above, and implement an adaptive step size algorithm for the 2. and 4. order Runge-Kutta methods. Use the CSTR reactor, and the seawater case as above to test your implementation. It might be a good idea to use a safety limit on the step size 'min(hi*(tol/toli)**(0.2),1)'.

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

```
def adaptive_ode_solv(c_into,c_init,t_final,tol=1e-4):
    f=[];t=[]
    tau_inv = q/vol
    c_in    = c_into #freshwater into tank
    c_old   = c_init #seawater present
    ti=0.; h_new=1;
    toli=1.; # a high init tolerance to enter while loop
    no_steps=0
    global_err=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        tol = tol + tol*np.fabs(c_old)
        while(toli>tol):
            hi=h_new
        # first two small steps:
            k1 = rk4_step(c_old,c_in, ...)
            k2 = rk4_step(k1,c_in, ...)
            # ... and one large step
            k3 = rk4_step(c_old,c_in,...)
        # estimate local error
            toli = ... (equation above)
        # new step size
            h_new=min(... (equation above) ... ,1)
            no_steps+=3
        toli=1.
    # estimate new solution
        c_old= ... (equation above)...
        ti   += hi
    print("No steps=", no_steps)
    return t,f
# rest of code is to make a figure
vol=1;q=1;c_into = 0; c_init =1
t_final=10 # end of simulation
t=[];f=[]
tol=[1e-8,1e-7,1e-6,1e-5]
for toli in tol:
    ti,fi = adaptive_ode_solv(c_into,c_init,t_final,toli)
    t.append(ti);f.append(fi)

f_an = np.exp(-np.array(t[0]))
symb = ['-p','-v','-*','-s']
fig = plt.figure(dpi=150)
for i in range(0,len(tol)):
    plt.plot(t[i], f[i], symb[i], label='tol='+str(tol[i]))
plt.plot(t[0], f_an, '-', color='k',label='analytical')
plt.legend(loc='lower left', ncol=1)
plt.grid()
plt.yscale('log')
plt.xlabel(r'Time [min/$\tau$]')
plt.ylabel('Concentration')
```

```
plt.show()
```

- How many steps do you need for the Runge-Kutta of 2. order compared to 4. order
- The tolerance might 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. $= 10^{-4}$).

Exercise 3: 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 a salty lake by injecting fresh water into it, but at the same time this lake was connected to two nearby fresh water lakes, as illustrated in figure 1. 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 flow 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.

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

$$\begin{aligned} C_0(t+\Delta t)\cdot V - C_0(t)\cdot V &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t, \\ C_1(t+\Delta t)\cdot V - C_1(t)\cdot V &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t, \\ C_2(t+\Delta t)\cdot V - C_2(t)\cdot V &= q(t)\cdot C_1(t)\cdot \Delta t - q(t)\cdot C_2(t)\cdot \Delta t. \end{aligned} \tag{7}$$

Taking the limit $\Delta t \to 0$, we can write equation (7) as:

$$V\frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right], \tag{8}$$

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right], \tag{9}$$

$$V\frac{dC_2(t)}{dt} = q(t)\left[C_1(t) - C_2(t)\right]. \tag{10}$$

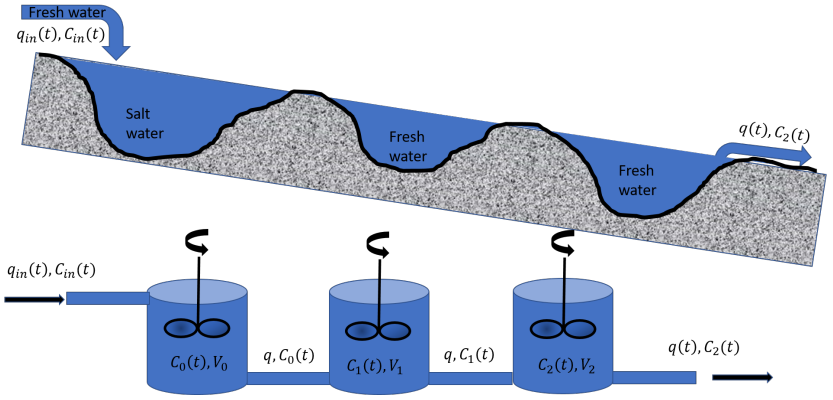

Figure 1: A simple model for cleaning a salty lake that is connected to two lakes down stream.

Show that the analytical solution is:

$$V\frac{dC_2(t)}{dt} = q(t)\left[\frac{C_{0,0}t}{\tau}e^{-t/\tau} - C_2(t)\right], \tag{11}$$

$$\frac{d}{dt}\left[e^{t/\tau}C_2\right] = \frac{C_{0,0}t}{\tau}, \tag{12}$$

$$C_2(t) = \frac{C_{0,0}t^2}{2\tau^2}e^{-t/\tau}. \tag{13}$$

The numerical solution follows the exact same pattern as before if we introduce a vector notation. Before doing that, we rescale the time $t \to t/\tau$ and the concentrations, $\hat{C}_i = C_i/C_{0,0}$ for $i = 0, 1, 2$, hence:

$$\frac{d}{dt}\begin{pmatrix}\hat{C}_0(t)\\ \hat{C}_1(t)\\ \hat{C}_2(t)\end{pmatrix} = \begin{pmatrix}\hat{C}_{\rm in}(t) - \hat{C}_0(t)\\ \hat{C}_0(t) - \hat{C}_1(t)\\ \hat{C}_1(t) - \hat{C}_2(t)\end{pmatrix},$$

$$\frac{d\hat{\mathbf{C}}(t)}{dt} = \mathbf{f}(\hat{\mathbf{C}}, t). \tag{14}$$

Finnish the implementation of the code:

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

def fm(c_old,c_in,tau):
    return (c_in-c_old)/tau

def rk4_step(c_old, c_in, tau, h):
    #vectorize the code
    return c_next
```

```
def ode_solv(c_into,c_init,t_final,tau,h):
    f=[];t=[]
    c_in  = c_into #freshwater into first tank
    c_old = c_init #seawater present
    ti=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        c_new = rk4_step(c_old,c_in,tau,h)
        c_old = c_new
        # add boundary condition - put concentration of tank 0 into tank 1 etc.

        ti   += h
    return np.array(t),np.array(f)
h = 1e-2
# initial values
vol=1;q=1;c_into = [0,0,0]; c_init = [1,0,0]
tau=[1,1,1];t_final=10 # end of simulation
t,f = ode_solv(c_into,c_init,t_final,tau,h)
# rest of code is to make a figure

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

symb = ['-p','-v','-*','-s']
fig = plt.figure(dpi=150)
for i in range(0,len(c_init)):
    legi = '$\hat{C}_'+str(i)+'(\\tau)$'
    plt.plot(t, f[:,i], '-', label=legi,lw=4)
    plt.plot(t, 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')
plt.show()
```

Exercise 4: Stiff sets of ODE and implicit methods

As already mentioned a couple of times, our system could be part of a much larger system. To illustrate this, let us now assume that we have two tanks in series. The first tank is similar to our original tank, but the second tank is a sampling tank, 1000 times smaller.

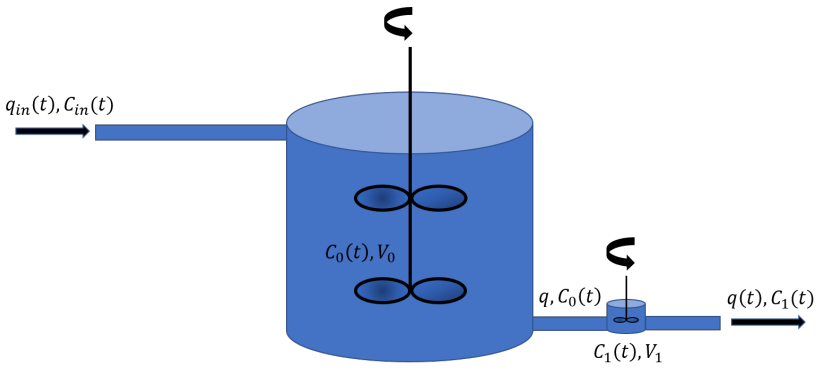

Figure 2: A continuous stirred tank model with a sampling vessel.

The governing equations can be found by requiring mass balance for each of the tanks:

$$\begin{aligned} C_0(t+\Delta t)\cdot V_0 - C_0(t)\cdot V_0 &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t. \\ C_1(t+\Delta t)\cdot V_1 - C_1(t)\cdot V_1 &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t. \end{aligned} \tag{15}$$

Taking the limit $\Delta t \to 0$, we can write equation (15) as:

$$V_0 \frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right]. \tag{16}$$

$$V_1 \frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right]. \tag{17}$$

Assume that the first tank is filled with seawater, $C_0(0) = C_{0,0}$, and fresh water is flooded into the tank, i.e. $C_{\text{in}} = 0$. Before we start to consider a numerical solution, let us first find the analytical solution: As before the solution for the first tank (equation (16)) is:

$$C_0(t) = C_{0,0}e^{-t/\tau_0}, \tag{18}$$

where $\tau_0 \equiv V_0/q$. Inserting this equation into equation (17), we get:

$$\begin{aligned} \frac{dC_1(t)}{dt} &= \frac{1}{\tau_1}\left[C_{0,0}e^{-t/\tau_0} - C_1(t)\right], \\ \frac{d}{dt}\left[e^{t/\tau_2}C_1\right] &= \frac{C_{0,0}}{\tau_1}e^{-t(1/\tau_0 - 1/\tau_1)}, \qquad (19) \\ C_1(t) &= \frac{C_{0,0}}{1-\frac{\tau_1}{\tau_0}}\left[e^{-t/\tau_0} - e^{-t/\tau_1}\right], \qquad (20) \end{aligned}$$

where $\tau_1 \equiv V_1/q$.

The methods we have considered so far are known as *explicit*, whenever we replace the solution in the right hand side of our algorithm with $y(t+\Delta t)$ or

(y_{n+1}), the method is known as *implicit*. Implicit methods are always stable, meaning that we can take as large a time step that we would like, without getting oscillating solution.

- Show that:

$$C_{0n+1} = \frac{C_{0n} + \frac{\Delta t}{\tau_0} C_{\mathrm{in}n+1}}{1 + \frac{\Delta t}{\tau_0}},$$
$$C_{2n+1} = \frac{C_{1n} + \frac{\Delta t}{\tau_1} C_{0n+1}}{1 + \frac{\Delta t}{\tau_1}}. \tag{21}$$

- Finnish the implementation below

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

def fm(c_old,c_in,tau,h):
    return ....

def euler_step(c_old, c_in, tau, h):
    c_next=[]
    for i in range(len(c_old)):
        if(i>0): c_in[i]=c_next[i-1] # c_new in next tank
        c_next.append(fm(c_old[i],c_in[i],tau[i],h))
    return c_next

def ode_solv(c_into,c_init,t_final,tau,h):
    f=[];t=[]
    c_in  = c_into #freshwater into first tank
    c_old = c_init #seawater present
    ti=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        c_new = euler_step(c_old,c_in,tau,h)
        c_old = c_new
        # put concentration of tank 0 into tank 1 etc.
        # ...
        ti   += h
    return np.array(t),np.array(f)
h = 0.01
# initial values
vol=1;q=1;c_into = [0,0]; c_init = [1,0]
tau=[1,1e-3];t_final=10 # end of simulation
t,f = ode_solv(c_into,c_init,t_final,tau,h)
# rest of code is to make a figure

f_an = [];t_an=np.arange(0,t_final,0.01)
```

```
f_an.append(c_init[0]*np.exp(-t_an))
f_an.append(c_init[0]*(1-tau[1]/tau[0])*(np.exp(-t_an/tau[0])-np.exp(-t_an/tau[1])))
symb = ['-p','-v','-*','-s']
fig = plt.figure(dpi=150)
for i in range(0,len(c_init)):
    legi = '$\hat{C}_'+str(i)+'(\\tau)$'
    plt.plot(t, f[:,i], '-', label=legi,lw=4)
    plt.plot(t_an, 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')
plt.show()
```

- Do you need more or less step to reach the same accuracy with the implicit method, compared to Eulers (explicit) method?

Exercise 5: Truncation Error in Eulers Method

We know that Eulers algorithm is accurate to second order. Our estimate of the new value, y_1^* (where we have used a * to indicate that we have used a step size of size h), should then be related to the true solution $y(t_1)$ in the following way:

$$y_1^* = y(t_1) + ch^2. \tag{22}$$

The constant c is unknown, but it can be found by taking two smaller steps of size $h/2$. If the steps are not too large, our new estimate of the value y_1 will be related to the true solution as:

$$y_1 = y(t_1) + 2c\left(\frac{h}{2}\right)^2. \tag{23}$$

In the following we will take a closer look at the adaptive Eulers algorithm and show that the constant c is indeed the same in equation (22) and (23). The true solution $y(t)$, obeys the following equation:

$$\frac{dy}{dt} = f(y,t), \tag{24}$$

and Eulers method to get from y_0 to y_1 by taking one (large) step, h is:

$$y_1^* = y_0 + hf(y_0,t_0), \tag{25}$$

We will also assume (for simplicity) that in our starting point $t = t_0$, the numerical solution, y_0, is equal to the true solution, $y(t_0)$, hence $y(t_0) = y_0$.

a) Show that when we take one step of size h from t_0 to $t_1 = t_0 + h$, $c = y''(t_0)/2$ in equation (22).

Answer. The local error, is the difference between the numerical solution and the true solution:

$$\epsilon^* = y(t_0 + h) - y_1^* = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + hf(y_0, t_0 + h)\right], \tag{26}$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (25). Using equation (24) to replace $y'(t_0)$ with $f(y_0, t_0)$, we find:

$$\epsilon^* = y(t_0 + h) - y_1^* = \frac{1}{2}y''(t_0)h^2 \equiv ch^2, \tag{27}$$

hence $c = y''(t_0)/2$.

b) Show that when we take two steps of size $h/2$ from t_0 to $t_1 = t_0 + h$, Eulers algorithm is:

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{28}$$

Answer.

$$y_{1/2} = y_0 + \frac{h}{2}f(y_0, t_0), \tag{29}$$

$$y_1 = y_{1/2} + \frac{h}{2}f(y_{1/2}, t_0 + h/2), \tag{30}$$

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{31}$$

Note that we have inserted equation (29) into equation (30) to arrive at equation (31).

c) Find an expression for the local error when using two steps of size $h/2$, and show that the local error is: $\frac{1}{2}ch^2$

Answer.

$$\epsilon = y(t_0 + h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2)\right]. \tag{32}$$

This equation is slightly more complicated, due to the term involving f inside the last parenthesis, we can use Taylor expansion to expand it about (y_0, t_0):

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) \\ + \frac{h}{2}\left[f(y_0, t_0)\left.\frac{\partial f}{\partial y}\right|_{y=y_0, t=t_0} + \frac{h}{2}\left.\frac{\partial f}{\partial t}\right|_{y=y_0, t=t_0}\right] + \mathcal{O}(h^2). \tag{33}$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (24):

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y}f(y,t) + \frac{\partial f(y,t)}{\partial t}. \quad (34)$$

Hence, equation (33) can be written:

$$f(y_0 + \frac{h}{2}f(y_0,t_0), t_0 + h/2) = f(y_0,t_0) + \frac{h}{2}y''(t_0,y_0), \quad (35)$$

hence the truncation error in equation (32) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4}y''(y_0,t_0) = \frac{1}{2}ch^2, \quad (36)$$

Solution. The local error, is the difference between the numerical solution and the true solution:

$$\epsilon^* = y(t_0+h) - y_1^* = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) - \left[y_0 + hf(y_0, t_0+h)\right], \quad (37)$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (25). Using equation (24) to replace $y'(t_0)$ with $f(y_0,t_0)$, we find:

$$\epsilon^* = y(t_0+h) - y_1^* = \frac{1}{2}y''(t_0)h^2 \equiv ch^2, \quad (38)$$

where we have ignored terms of higher order than h^2, and defined c as $c = y''(t_0)/2$. Next we take two steps of size $h/2$ to reach y_1:

$$y_{1/2} = y_0 + \frac{h}{2}f(y_0,t_0), \quad (39)$$

$$y_1 = y_{1/2} + \frac{h}{2}f(y_{1/2}, t_0 + h/2), \quad (40)$$

$$y_1 = y_0 + \frac{h}{2}f(y_0,t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0,t_0), t_0 + h/2). \quad (41)$$

Note that we have inserted equation (39) into equation (40) to arrive at equation (41). The truncation error in this case is, as before:

$$\epsilon = y(t_0+h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) - \left[y_0 + \frac{h}{2}f(y_0,t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0,t_0), t_0 + h/2)\right]. \quad (42)$$

This equation is slightly more complicated, due to the term involving f inside the last parenthesis, we can use Taylor expansion to expand it about (y_0, t_0):

$$f(y_0 + \frac{h}{2}f(y_0,t_0), t_0 + h/2) = f(y_0,t_0) + \frac{h}{2}\left[f(y_0,t_0)\left.\frac{\partial f}{\partial y}\right|_{y=y_0,t=t_0} + \left.\frac{\partial f}{\partial t}\right|_{y=y_0,t=t_0}\right] + \mathcal{O}(h^2). \quad (43)$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (24):

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y}f(y,t) + \frac{\partial f(y,t)}{\partial t}. \quad (44)$$

Hence, equation (43) can be written:

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2}y''(t_0, y_0), \quad (45)$$

hence the truncation error in equation (42) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4}y''(y_0, t_0) = \frac{1}{2}ch^2, \quad (46)$$

References

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