1 Ordinary Differential Equations

Physical systems evolves in space and time, and very often they are described by a ordinary differential equations (ODE) and/or partial differential equations (PDE). The difference between an ODE and a PDE is that an ODE only describes the changes in one spatial dimension *or* time, whereas a PDE describes a system that evolves in the $x-$, $y-$, $z-$ dimension and/or in time. In the following we will spend a significant amount of time to explore one of the simplest algorithm, Eulers method. Sometimes this is exactly the algorithm you would like to use, but with very little extra effort much more sophisticated algorithms can easily be implemented, such as the Runge-Kutta fourth order method. However, all these algorithms, will at some point run into the same kind of troubles if used reckless. Thus we will use the Eulers method as a playground, investigate when the algorithm run into trouble and suggests ways to fix it, these approaches can easily be extended to the higher order methods. Most of the other algorithms boils down to the same idea of extrapolating a function using derivatives multiplied with a small step size.

2 A Simple Model for Fluid Flow

Let us consider a simple example from chemical engineering, a continuous stirred tank reactor (CSTR), see figure 1. The flow is incompressible ($q_{\text{out}} = q_{\text{in}}$), a fluid is entering on the top and exiting at the bottom, the tank has a fixed volume V. Assume that the tank is filled with saltwater, and that freshwater is pumped into it, how much time does it take before 90% of the saltwater is replaced with freshwater? The tank is *well mixed*, illustrated with the propeller, this means that at every time the concentration is uniform in the tank, i.e. that $C(t) = C_{\text{out}}(t)$.

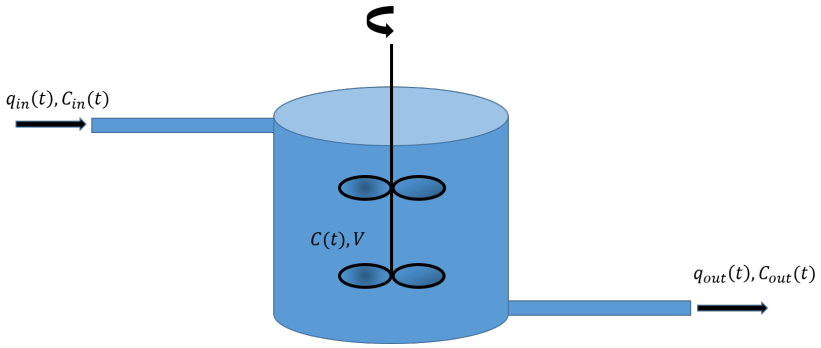

Figure 1: A continuous stirred tank model, $C(t) = C_{\text{out}}(t)$, and $q_{\text{out}} = q_{\text{in}}$.

The concentration C is measured in gram of salt per liter water, and the flow rate q is liter of water per day. The model for the salt balance in this system

can be described in words by:

$$\begin{aligned}[\text{accumulation of salt}] = [\text{salt into the system}] - [\text{salt out of the system}] \\ + [\text{generation of salt}]. \end{aligned} \tag{1}$$

In our case there are no generation of salt within the system so this term is zero. The flow of salt into the system during a time Δt is: $q_{\text{in}}(t) \cdot C_{\text{in}}(t) \cdot \Delta t = q(t) \cdot C_{\text{in}}(t) \cdot \Delta t$, the flow of salt out of the system is: $q_{\text{out}}(t) \cdot C_{\text{out}}(t) \cdot \Delta t = q(t) \cdot C(t) \cdot \Delta t$, and the accumulation during a time step is: $C(t + \Delta t) \cdot V - C(t) \cdot V$, hence:

$$C(t + \Delta t) \cdot V - C(t) \cdot V = q(t) \cdot C_{\text{in}}(t) \cdot \Delta t - q(t) \cdot C(t) \cdot \Delta t. \tag{2}$$

Note that it is not a priori apparent, which time the concentrations and flow rates on the right hand side should be evaluated at, we could have chosen to evaluate them at $t + \Delta t$, or at any time $t \in [t, t + \Delta t]$. We will return to this point later in this chapter. Dividing by Δt, and taking the limit $\Delta t \to 0$, we can write equation (2) as:

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

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. The equation (3) the reduces to:

$$V \frac{dC(t)}{dt} = -qC(t), \tag{4}$$

this equation can easily be solved, by dividing by C, multiplying by dt and integrating:

$$\begin{aligned} V \int_{C_0}^{C} \frac{dC}{C} &= -q \int_0^t dt, \\ C(t) &= C_0 e^{-t/\tau}, \text{ where } \tau \equiv \frac{V}{q}. \end{aligned} \tag{5}$$

This equation can be inverted to give $t = -\tau \ln[C(t)/C]$. If we assume that the volume of the tank is 1m^3=1000liters, and that the flow rate is 1 liter/min, we find that τ=1000min=0.69days and that it takes about $-0.69 \ln 0.9 \simeq 1.6$days to reduce the concentration by 90% to 3.5 gram/liter.

The CSTR.

You might think that the CSTR is a very simple model, and it is, but this type of model is the basic building blocks in chemical engineering. By putting CSTR tanks in series and/or connecting them with pipes, the efficiency of manufacturing various type of chemicals can be investigated. Altough the CSTR is an idealized model for the part of a chemical factory, it is actually a *very good* model for fluid flow in a porous media. By connecting

CSTR tanks in series, one can model how chemical tracers propagate in the subsurface. The physical reason for this is that dispersion in porous media will play the role of the propellers and mix the concentration uniformly.

3 Eulers Method

If the system gets slightly more complicated, e.g several tanks in series with a varying flow rate or if salt was generated in the tank, there is a good chance that we have to solve the equations numerically to obtain a solution. Actually, we have already developed a numerical algorithm to solve equation (3), before we arrived at equation (3) in equation (2). This is a special case of Eulers method, which is basically to replace the derivative in equation (3), with $(C(t+\Delta t) - C(t))/\Delta t$. By rewriting equation (2), so that we keep everything related to the new time step, $t + \Delta t$, on one side, we get:

$$VC(t+\Delta t) = VC(t) + qC_{\text{in}}(t) - qC(t), \tag{6}$$

$$C(t+\Delta t) = C(t) + \frac{\Delta t}{\tau}\left[C_{\text{in}}(t) - C(t)\right], \tag{7}$$

we introduce the short hand notation: $C(t) = C_n$, and $C(t+\Delta t) = C_{n+1}$, hence the algorithm can be written more compact as:

$$C_{n+1} = \left(1 - \frac{\Delta t}{\tau}\right) C_n + \frac{\Delta t}{\tau} C_{\text{in},n}, \tag{8}$$

In the script below, we have implemented equation (8).

```
def analytical(x):
    return np.exp(-x)

def euler_step(c_old, c_in, tau_inv,dt):
    fact=dt*tau_inv
    return (1-fact)*c_old+fact*c_in

def ode_solv(c_into,c_init,t_final,vol,q,dt):
    f=[];t=[]
    tau_inv = q/vol
    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 = euler_step(c_old,c_in,tau_inv,dt)
        c_old = c_new
        ti   += dt
    return t,f
```

In figure 2 the result of the implementation is shown for different values of Δt Clearly we see that the results are dependent on the step size, as the step increases the numerical solution deviates from the analytical solution. At some point the numerical algorithm fails completely, and produces results that have no meaning.

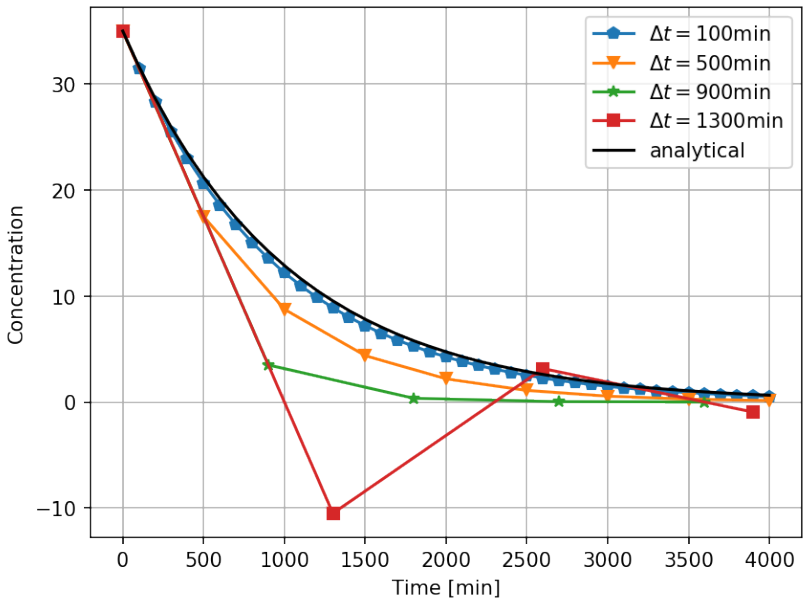

Figure 2: The concentration in the tank for different step size Δt.

3.1 Error Analysis - Eulers Method

There are two obvious questions:

1. When does the algorithm produce unphysical results?
2. What is an appropriate step size?

Let us consider the first question, clearly when the concentrations gets negative the solution is unphysical. From equation (8), we see that when $\Delta t/\tau > 1$, the concentration become negative. For this specific case (the CSTR), there is a clear physical interpretation of this condition. Inserting $\tau = V/q$, we can rewrite the condition $\Delta t/\tau < 1$ as $q\Delta t < V$. The volume into the tank during one time step is: $q\Delta t$, which means that whenever we *flush more than one tank volume through the tank during one time step, the algorithm fails*. When this happens the new concentration in the tank cannot be predicted from the old one. This makes sense, because we could have switched to a new solution (e.g. seawater) during that time step, then the new solution does not have any relation to the old solution.

The second question, "what is an appropriate step size?", is a bit more difficult to answer. One strategy could be to simply use the results from chapter [Taylor], where we showed that the truncation error had a minimum value with a step size of 10^{-8} (when using a first order Taylor approximation). How does the value 10^{-8} relate to the step sizes in minutes used in our Euler implementation? In

order to see the connection, we need to rewrite equation (3) in a dimensionless form, by making the following substitution: $t \to t/\tau$:

$$\frac{dC(\tau)}{d\tau} = \left[C_{\text{in}}(\tau) - C(\tau)\right]. \tag{9}$$

As we found earlier $\tau = 1000$min, thus a step size of e.g. 1 min would correspond to a dimensionless time step of $\Delta t \to$1min/1000min$= 10^{-3}$. This number can be directly compared to the value 10^{-8}, which is the lowest value we can choose without getting into trouble with round off errors on the machine.

Dimensionless variables.

It is a good idea (necessary) to formulate our equations in terms of dimensionless variables. The algorithms we develop can then be used in the same form regradless of changes in the system size and flow rates. Thus we do not need to rewrite the algorithm each time the physical system changes. This also means that if you use an algorithm developed by someone else (e.g. in Matlab or Python), you should always formulate the ODE system in dimensionless form before using the algorithm.

A second reason is that from a pure modeling point of view, dimensionless variables is a way of getting some understanding of what kind of combination of the physical parameters that describes the behavior of the system. For the case of the CSTR, there is a time scale $\tau = V/q$, which is an intrinsic measure of time in the system. No matter what the flow rate through the tank or the volume of the tank is, it will always take 0.1τ before the concentration in the tank is reduced by 90%.

As already mentioned a step size of 10^{-8}, is probably the smallest we can choose with respect to round off errors, but it is smaller than necessary and would lead to large simulation times. If it takes 1 second to run the simulation with a step size of 10^{-3}, it would take 10^5 seconds or 1 day with a step size of 10^{-8}. To continue the error analyses, we write our ODE for a general system as:

$$\frac{dy}{dt} = f(y,t), \tag{10}$$

or in discrete form:

$$\frac{y_{n+1} - y_n}{h} - \frac{h}{2}y''(\eta_n) = f(y,t).$$

$$y_{n+1} = y_n + hf(y,t) + \frac{h^2}{2}y''(\eta_n). \tag{11}$$

h is now the (dimmensionless) step size, equal to Δt if the derivative is with respect to t or Δx if the derivative is respect to x etc. Note that we have also included the error term related to the numerical derivative, $\eta_n \in [t_n, t_n + h]$. At

each step we get an error term, and the distance between the true solution and our estimate, the *local error*, after N steps is:

$$\epsilon = \sum_{n=0}^{N-1} \frac{h^2}{2} y''(\eta_n) = \frac{h^2}{2} \sum_{n=0}^{N-1} f'(y_n, \eta_n) \simeq \frac{h}{2} \int_{t_0}^{t_f} f'(y, \eta) d\eta$$
$$= \frac{h}{2} \left[f(y(t_f), t_f) - f(y(t_0), t_0) \right]. \tag{12}$$

Note that when we replace the sum with an integral in the equation above, this is only correct if the step size is not too large. From equation (12) we see that even if the error term on the numerical derivative is h^2, the local error is proportional to h (one order lower). This is because we accumulate errors for each step.

In the following we specialize to the CSTR, to see if we can gain some additional insight. First we change variables in equation (4): $y = C(t)/C_0$, and $x = t/\tau$, hence:

$$\frac{dy}{dx} = -y. \tag{13}$$

The solution to this equation is $y(x) = e^{-x}$, substituting back for the new variables y and x, we reproduce the result in equation (5). The local error, equation (12), reduces to:

$$\epsilon = \frac{h}{2} \left[-y(x_f) + y(x_0) \right] = \frac{h}{2} \left[1 - e^{-x_f} \right], \tag{14}$$

we have assumed that $x_0 = t_0/\tau = 0$. This gives the estimated local error at time x_f. For $x_f = 0$, the numerical error is zero, this makes sense because at $x = 0$ we know the exact solution because of the initial conditions. When we move further away from the initial conditions, the numerical error increases, but equation (14) ensures us that as long as the step size is low enough we can get as close as possible to the true solution, since the error scales as h (at some point we might run into trouble with round off error in the computer).

Can we prove directly that we get the analytical result? In this case it is fairly simple, if we use Eulers method on equation (13), we get:

$$\frac{y_{n+1} - y_n}{h} = -y_n f.$$
$$y_{n+1} = (1 - h) y_n, \tag{15}$$

or alternatively:

$$y_1 = (1 - h) y_0,$$
$$y_2 = (1 - h) y_1 = (1 - h)^2 y_0,$$
$$\vdots$$
$$y_{N+1} = (1 - h)^N y_0 = (1 - h)^{x_f/h} y_0. \tag{16}$$

In the last equation, we have used the the fact the number of steps, N, is equal to the simulation time divided by the step size, hence: $N = x_f/h$. From calculus, the equation above is one of the well known limits for the exponential function: $\lim_{x\to\infty}(1 + k/x)^{mx} = e^{mk}$, hence:

$$y_n = (1 - h)^{x_f/h} y_0 \to e^{-x_f}, \tag{17}$$

when $h \to 0$. Below is an implementation of the Euler algorithm in this simple case, we also estimate the local error, and global error after N steps.

```
import matplotlib.pyplot as plt
import numpy as np
def euler(tf,h):
    t=[];f=[]
    ti=0.;fi=1.
    t.append(ti);f.append(fi)
    global_err=0.
    while(ti<= tf):
        ti+=h
        fi=fi*(1-h)
        global_err += abs(np.exp(-ti)-fi)
        t.append(ti);f.append(fi)
    print("error= ", np.exp(-ti)-fi," est.err=", .5*h*(1-np.exp(-ti)))
    print("global error=",global_err)
    return t,f

t,f=euler(1.,1e-5)
```

By changing the step size h, you can easily verify that the local error systematically increases or decreases proportional to h. Something curious happens with the global error when the step size is changed, it does not change very much. The global error involves a second sum over the local error for each step, which can be approximated as a second integration in equation (14):

$$\epsilon_{\text{global}} = \frac{1}{2}\int_0^{x_f} \left[-y(x) + y(0)\right] dx = \frac{1}{2}\left[x_f + e^{-x_f} - 1\right]. \tag{18}$$

Note that the global error does not go to zero when the step size decreases, which can easily be verified by changing the step size. This is strange, but can be understood by the following argument: when the step size decreases the local error scales as $\sim h$, but the number of steps scales as $1/h$, so the global error must scale as $h \times 1/h$ or some constant value. Usually it is much easier to control the local error than the global error, this should be kept in mind if you ever encounter a problem where it is important control the global error. For the higher order methods that we will discuss later in this chapter, the global error will go to zero when h decreases.

The answer to our original question, "What is an appropriate step size?", will depend on what you want to achieve in terms of local or global error. In most practical situations you would specify a local error that is acceptable for the problem under investigation and then choose a step size where the local error always is lower than this value. In the next subsection we will investigate how to achieve this in practice.

3.2 Adaptive step size - Eulers Method

We want to be sure that we use a step size that achieves a certain accuracy in our numerical solution, but at the same time that we do not waste simulation time using a too low step size. The following approach is similar to the one we derived for the Romberg integration, and a special case of what is known as Richardson Extrapolation. The method is easily extended to higher order methods.

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:

$$y_1^* = y(t_1) + ch^2. \tag{19}$$

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:

$$y_1 = y(t_1) + 2c\left(\frac{h}{2}\right)^2. \tag{20}$$

The factor 2 in front of c is because we now need to take two steps, and we accumulate a total error of $2c(h/2)^2 = ch^2/2$. It might not be completely obvious that the constant c should be the same in equation (19) and (20). If you are not convinced, there is an exercise at the end of the chapter. We define:

$$\Delta \equiv y_1^* - y_1 = c\frac{h^2}{2}. \tag{21}$$

The truncation error in equation (20) is:

$$\epsilon = y(t_1) - y_1 = 2c\left(\frac{h}{2}\right)^2 = \Delta. \tag{22}$$

Now we have everything we need: We want the local error to be smaller than some prespecified tolerance, ϵ', or equivalently that $\epsilon \leq \epsilon'$. To achieve this we need to use an optimal step size, h', that gives us exactly the desired error:

$$\epsilon' = c\frac{h'^2}{2}. \tag{23}$$

Dividing equation (23) by equation (22), we can estimate the optimal step size:

$$h' = h\sqrt{\left|\frac{\epsilon'}{\epsilon}\right|}, \tag{24}$$

where the estimated error, ϵ, is calculated from equation (22). Equation (24) serves two purposes, if the estimated error ϵ is higher than the tolerance, ϵ', we

have specified it will give us an estimate for the step size we should choose in order to achieve a higher accuracy, if on the other hand $\epsilon' > \epsilon$, then we get an estimate for the next, larger step. Before the implementation we note, as we did for the Romberg integration, that equation (22) also gives us an estimate for the error term in equation (20) as an improved estimate of y_1. This we get for free and will make our Euler algorithm accurate to h^3, hence the improved Euler step, $\hat{y_1}$, is to *subtract* the error term from our previous estimate:

$$\hat{y_1} = y_1 - \epsilon = 2y_1 - y_1^*. \tag{25}$$

Below is an implementation of the adaptive Euler algorithm:

```
def one_step(c_old, c_in,h):
    return (1-h)*c_old+h*c_in

def adaptive_euler(c_into,c_init,t_final,tol=1e-4):
    f=[];t=[]
    c_in    = c_into #freshwater into tank
    c_old   = c_init #seawater present
    ti=0.; h_new=1e-3;
    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)
        while(toli>tol):# first two small steps
            hi=h_new
            k1 = one_step(c_old,c_in,hi*.5)
            k2 = one_step(k1,c_in,hi*.5)
            # ... and one large step
            k3 = one_step(c_old,c_in,hi)
            toli = abs(k3-k2)
            h_new=hi*np.sqrt(tol/toli)
            no_steps+=3
        toli=1.
        c_old=2*k2-k3 # higher order correction
 # normal Euler, uncomment and inspect the global error
 #       c_old = k2
        ti   += hi
        global_err += abs(np.exp(-ti)-c_old)
    print("No steps=", no_steps, "Global Error=", global_err)
    return t,f
```

In figure 3 the result of the implementation is shown. Note that the number of steps for an accuracy of 10^{-6} is only about 3000. Without knowing anything about the accuracy, we would have to assume that we needed a step size of the order of h in order to reach a local accuracy of h because of equation (12). In the current case, we would have needed 10^7 steps, which would lead to unnecessary long simulation times.

Local error and bounds.

In the previous example we set an absolute tolerance, and required that our estimate y_n always is within a certain bound of the true solution $y(t_n)$,

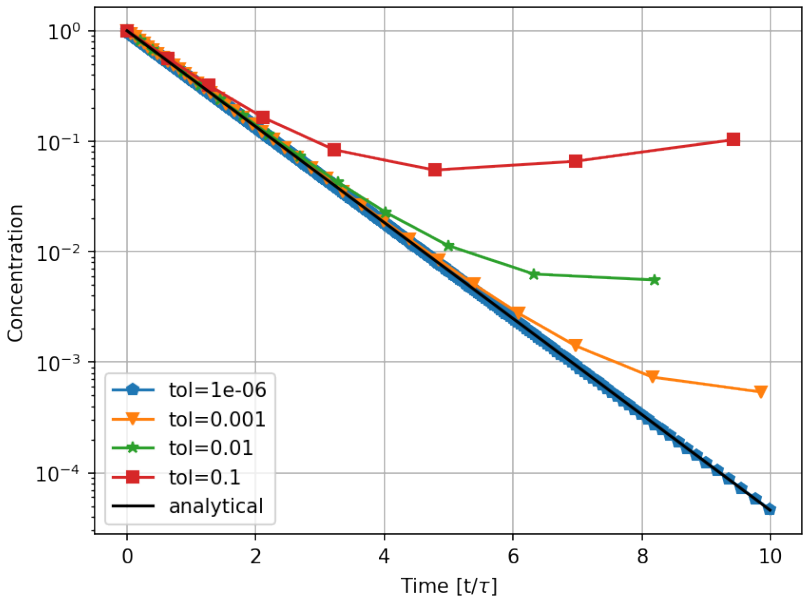

Figure 3: The concentration in the tank using adaptive Euler. Number of Euler steps are: 3006, 117, 48 and 36 for the different step sizes.

i.e. $|y(t_n) - y_n| \leq \epsilon'$. This is a very strong demand, and sometimes it makes more sense to require that we also accept a relative tolerance proportional to function value. In some areas the solution might have a very large value, and then another possibility would be to have an ϵ' that varied with the function value:

$$\epsilon' = atol + |y|rtol, \tag{26}$$

where 'atol' is the absolute tolerance and 'rtol' is the relative tolerance. A sensible choice would be to set 'atol=rtol' (e.g. $= 10^{-4}$).

4 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 Runge-Kutta methods may be motivated by inspecting the Euler method in figure 4. The Euler method uses information from the previous time step to estimate the value at the new time step. The Runge Kutta methods uses the information about the slope between the points t_n and $t_n + h$. By inspecting figure 4, we clearly see that by using the slope at $t_n + h/2$ would give us a significant improvement. The 2. order Runge-Kutta method can be derived by Taylor expanding the solution around

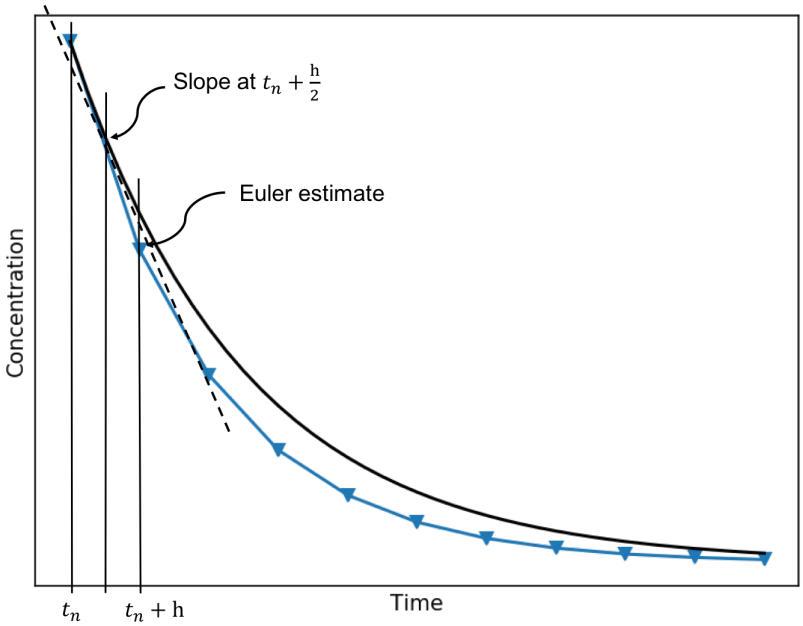

Figure 4: Illustration of the Euler algorithm, and a motivation for using the slope a distance from the t_n.

$t_n + h/2$, we do this by setting $t_n + h = t_n + h/2 + h/2$:

$$y(t_n + h) = y(t_n + \frac{h}{2}) + \frac{h}{2}\left.\frac{dy}{dt}\right|_{t=t_n+h/2} + \frac{h^2}{4}\left.\frac{d^2y}{dt^2}\right|_{t=t_n+h/2} + \mathcal{O}(h^3). \quad (27)$$

Similarly we can expand the solution in $y(t_n)$ about $t_n + h/2$, by setting $t_n = t_n + h/2 - h/2$:

$$y(t_n) = y(t_n + \frac{h}{2}) - \frac{h}{2}\left.\frac{dy}{dt}\right|_{t=t_n+h/2} + \frac{h^2}{4}\left.\frac{d^2y}{dt^2}\right|_{t=t_n+h/2} - \mathcal{O}(h^3). \quad (28)$$

Substracting these two equations the term $y(t_n + \frac{h}{2})$, and all even powers in the derivative cancels out:

$$y(t_n + h) = y(t_n) + h\left.\frac{dy}{dt}\right|_{t=t_n+h/2} + \mathcal{O}(h^3),$$
$$y(t_n + h) = y(t_n) + hf(y_{n+h/2}, t_n + h/2) + \mathcal{O}(h^3). \quad (29)$$

In the last equation, we have used equation (10). Note that we now have an expression that is very similar to Eulers algorithm, but it is accurate to order h^3. There is one problem, and that is that the function f is to be evaluated at the point $y_{n+1/2} = y(t_n + h/2)$ which we do not know. This can be fixed

by using Eulers algorithm: $y_{n+1/2} = y_n + h/2 f(y_n, t_n)$. We can do this even if Eulers algorithm is only accurate to order h^2, because the f in equation (29) is multiplied by h, and thus our algorithm is still accurate up to order h^3.

The 2. order Runge-Kutta:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\ y_{n+1} &= y_n + k_2 \end{aligned} \tag{30}$$

Below is a Python implementation of equation (30):

```
def fm(c_old,c_in):
    return c_in-c_old

def rk2_step(c_old, c_in, h):
    k1=h*fm(c_old,c_in)
    k2=h*fm(c_old+0.5*k1,c_in)
    return c_old+k2

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 = rk2_step(c_old,c_in,h)
        c_old = c_new
        ti   += h
    return t,f
```

In figure 5 the result of the implementation is shown. Note that when comparing Runge-Kutta 2. order with Eulers method, see figure 5 and 2, we of course have the obvious result that a larger step size can be taken, without loosing numerical accuracy. It is also worth noting that we can take steps that is larger than the tank volume. Eulers method failed whenever the time step was larger than one tank volume ($h = t/\tau > 1$), whereas the Runge-Kutta method finds a physical solution for step sizes lower than twice the tank volume. If the step size is larger, we see that the concentration in the tank increases, which is clearly unphysical.

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 development of the algorithm itself is similar to the 2. order method, but of course more involved. We just quote the result:

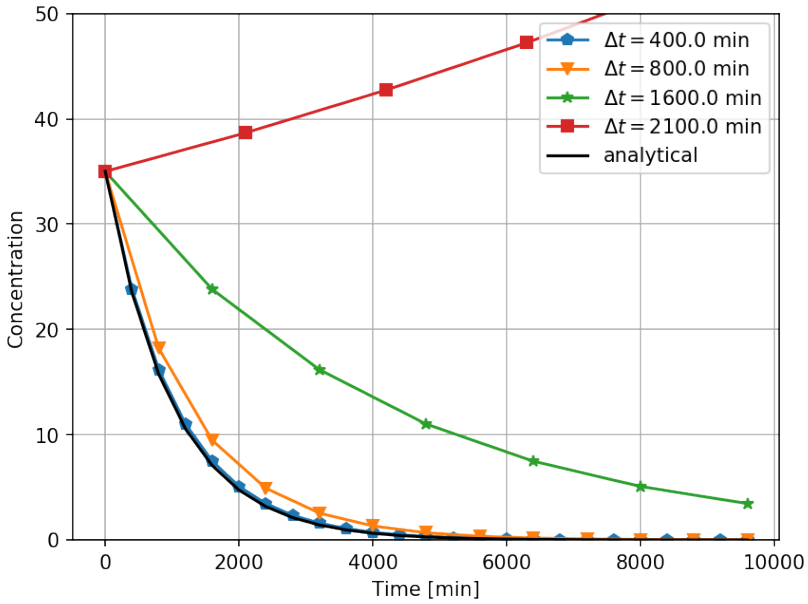

Figure 5: The concentration in the tank for different step size Δt.

The 4. order Runge-Kutta:

$$
\begin{aligned}
k_1 &= hf(y_n, t_n) \\
k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\
k_3 &= hf(y_n + \frac{1}{2}k_2, t_n + h/2) \\
k_4 &= hf(y_n + k_3, t_n + h) \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned} \tag{31}
$$

Below is a Python implementation of equation (31):

```
def fm(c_old,c_in):
    return c_in-c_old

def rk4_step(c_old, c_in, h):
    k1=h*fm(c_old,c_in)
    k2=h*fm(c_old+0.5*k1,c_in)
    k3=h*fm(c_old+0.5*k2,c_in)
    k4=h*fm(c_old+    k3,c_in)
    return c_old+(k1+2*k2+2*k3+k4)/6
```

```
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)
        c_old = c_new
        ti   += h
    return t,f
```

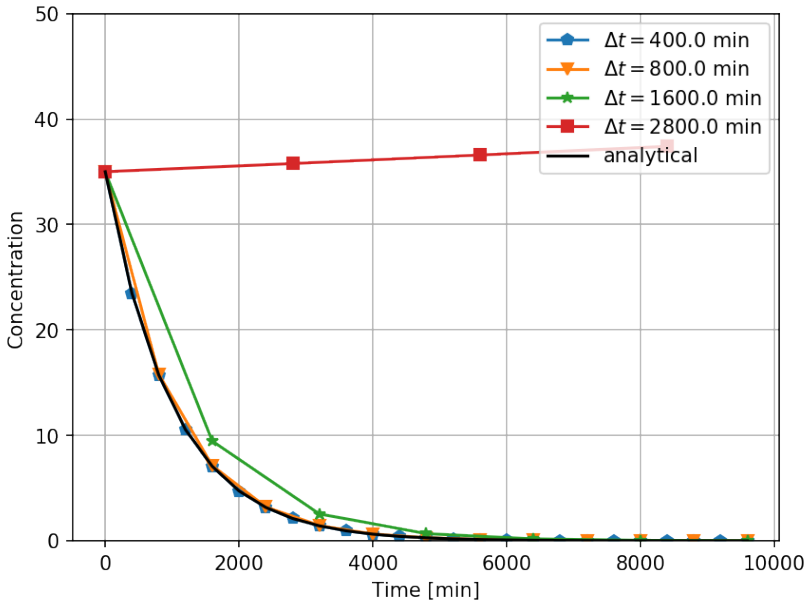

Figure 6: The concentration in the tank for different step size Δt.

In figure 6 the result of the implementation is shown.

4.1 Adaptive step size - Runge-Kutta Method

Just as we did with Eulers method, we can implement an adaptive method. The derivation is exactly the same, but this time our method is accurate to fourth order, hence the error term is of order h^5. We start by taking one large step of size h, our estimate, y_1^* is related to the true solution, $y(t_1)$, in the following way:

$$y_1^* = y(t_1) + ch^5, \tag{32}$$

Next, we take two steps of half the size, $h/2$, hence:

$$y_1 = y(t) + 2c\left(\frac{h}{2}\right)^5. \tag{33}$$

Subtracting equation (32) and (33), we find an expression similar to equation (21):

$$\Delta \equiv y_1^* - y_1 = c\frac{15}{16}h^5, \tag{34}$$

or $c = 16\Delta/(15h^5)$. For the Euler scheme, Δ also happened to be equal to the truncation error, but in this case it is:

$$\epsilon = 2c\left(\frac{h}{2}\right)^5 = \frac{\Delta}{15} \tag{35}$$

we want the local error, ϵ, to be smaller than some tolerance, ϵ'. The optimal step size, h', that gives us exactly the desired error is then:

$$\epsilon' = 2c\left(\frac{h'}{2}\right)^5. \tag{36}$$

Dividing equation (36) by equation (35), we can estimate the optimal step size:

$$h' = h\left|\frac{\epsilon}{\epsilon}\right|^{1/5}, \tag{37}$$

ϵ can be calculated from equation (35). Below is an implementation

```
def fm(c_old,c_in):
    return c_in-c_old

def rk4_step(c_old, c_in, h):
    k1=h*fm(c_old,c_in)
    k2=h*fm(c_old+0.5*k1,c_in)
    k3=h*fm(c_old+0.5*k2,c_in)
    k4=h*fm(c_old+    k3,c_in)
    return c_old+(k1+2*k2+2*k3+k4)/6

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*c_old
        while(toli>tol):# first two small steps
            hi=h_new
            k1 = rk4_step(c_old,c_in,hi*.5)
```

```
            k2 = rk4_step(k1,c_in,hi*.5)
            # ... and one large step
            k3 = rk4_step(c_old,c_in,hi)
            toli = abs(k3-k2)/15
            h_new=min(hi*(tol/toli)**(0.2),1)
            no_steps+=3
        toli=1.
        c_old=k2-(k3-k2)/15
        ti   += hi
        global_err += abs(np.exp(-ti)-c_old)
    print("No steps=", no_steps, "Global Error=", global_err)
    return t,f
```

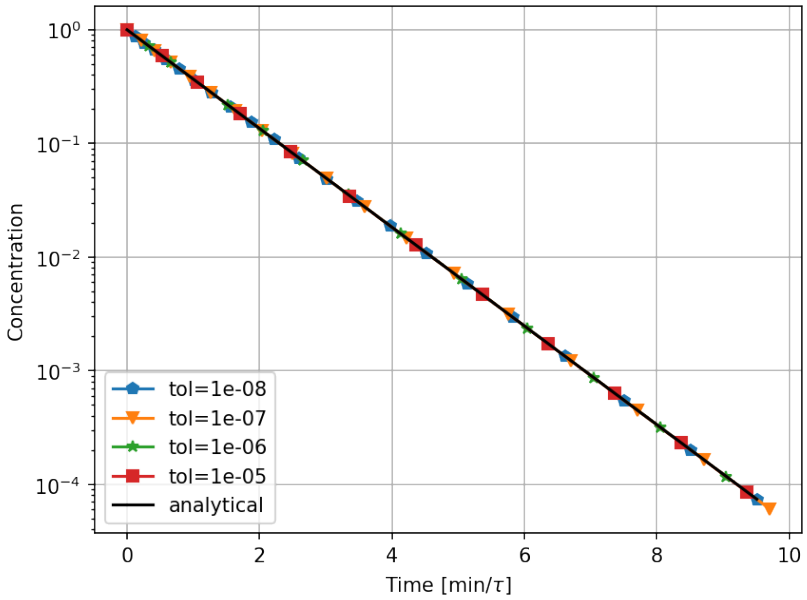

Figure 7: The concentration in the tank for different step size Δt. Number of rk4 steps are: 138, 99, 72 and 66 for the different step sizes and 'rtol=0', for 'rtol=tol' the number of rk4 steps are 81, 72, 63, 63.

In figure 7 the result of the implementation is shown. Note that we put a safety limit on the step size 'min(hi*(tol/toli)**(0.2),1)'.

In general we can use the same procedure any method accurate to order h^p, and you can easily verify that:

Error term and step size for a h^p method:

$$\epsilon = \frac{|\Delta|}{2^p - 1} = \frac{|y_1^* - y_1|}{2^p - 1}, \tag{38}$$

$$h' = \beta h \left| \frac{\epsilon}{\epsilon_0} \right|^{\frac{1}{p+1}}, \tag{39}$$

$$\hat{y_1} = y_1 - \epsilon = \frac{2^{p-1} y_1 - y_1^*}{2^{p-1} - 1}, \tag{40}$$

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

4.2 Conservation of Mass

A mathematical model of a physical system should always be formulated in such a way that it is consistent with the laws of nature. In practical situations this statement is usually equivalent to state that the mathematical model should respect conservation laws. The conservation laws can be conservation of mass, energy, momentum, electrical charge, etc. In our example with the mixing tank, we were able to derive an expression for the concentration of salt out of the tank, equation (5), by *demanding* conservation of mass (see equation (2)).

A natural question to ask is then: If our mathematical model respect conservation of mass, are we sure that our solution method respect conservation of mass? We of course expect that when the grid spacing approaches zero our numerical solution will get closer and closer to the analytical solution. Clearly when $\Delta x \to 0$, the mass is conserved. So what is the problem? The problem is that in many practical problems we cannot always have a step size that is small enough to ensure that our solution always is close enough to the analytical solution. The physical system we consider might be very complicated (e.g. a model for the earth climate), and our ODE system could be a very small part of a very big system. A very good test of any code is to investigate if the code respect the conservation laws. If we know that our implementation respect e.g. mass conservation at the discrete level, we can easily test mass conservation by summing up all the mass entering, and subtracting the mass out of and present in our system. If the mass is not conserved exactly, there is a good chance that there is a bug in our implementation.

If we now turn to our system, we know that the total amount of salt in the system when we start is $C(0)V$. The amount entering is zero, and the amount leaving each time step is $q(t)C(t)\Delta t$. Thus we should expect that if we add the amount of salt in the tank to the amount that has left the system we should always get an amount that is equal to the original amount. Alternatively, we expect $\int_{t_0}^{t} qC(t)dt + C(t)V - C(0)V = 0$. Adding the following code in the `while(ti <= t_final):` loop:

```
mout += 0.5*(c_old+c_new)*q*dt
mbal  = (c_new*vol+mout-vol*c_init)/(vol*c_init)
```

it is possible to calculate the amount of mass lost (note that we have used the trapezoidal formula to calculate the integral). In the table below the fraction of mass lost relative to the original amount is shown for the various numerical methods.

Δt	h	Euler	RK 2. order	RK 4. order
900	0.9	-0.4500	0.3682	0.0776
500	0.5	-0.2500	0.0833	0.0215
100	0.1	-0.0500	0.0026	0.0008
10	0.01	-0.0050	2.5E-05	8.3E-06

We clearly see from the table that the Runge-Kutta methods performs better than Eulers method, but *all of the methods violates mass balance.*

This might not be a surprise as we know that our numerical solution is always an approximation to the analytical solution. How can we then formulate an algorithm that will respect conservation laws at the discrete level? It turns out that for Eulers method it is not so difficult. Eulers algorithm at the discrete level (see equation (6)) is actually a two-step process: first we inject the fresh water while we remove the "old" fluid *and then we mix*. By thinking about the problem this way, it makes more sense to calculate the mass out of the tank as $\sum_k q_k C_k \Delta t_k$. If we in our implementation calculates the mass out of the tank as:

```
mout += c_old*q*dt
mbal  = (c_new*vol+mout-vol*c_init)/(vol*c_init)
```

We easily find that the mass is exactly conserved at every time for Eulers method. The concentration in the tank will of course not be any closer to the analytical solution, but if our mixing tank was part of a much bigger system we could make sure that the mass would always be conserved if we make sure that the mass out of the tank and into the next part of the system was equal to $qC(t)\Delta t$.

5 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 8. 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.

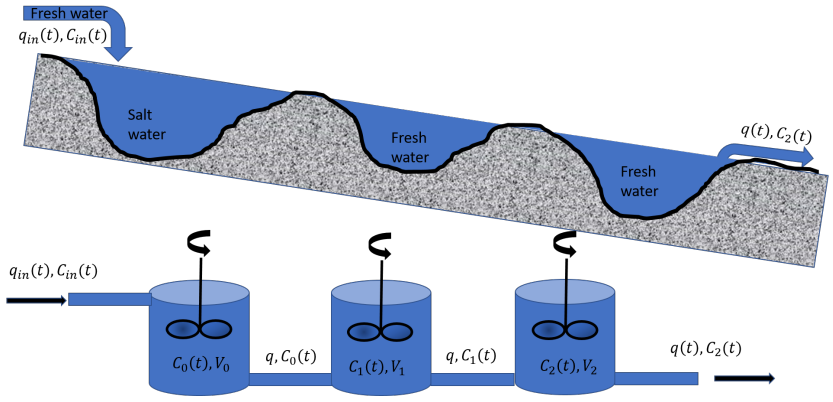

Figure 8: A simple model for cleaning a salty lake that is connected to two lakes down stream.

For simplicity we will assume that all the lakes have the same volume, V. The governing equations follows as before, by assuming mass balance (equation (1)):

$$\begin{aligned} C_0(t+\Delta t)\cdot V - C_0(t)\cdot V &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t, \\ C_1(t+\Delta t)\cdot V - C_1(t)\cdot V &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t, \\ C_2(t+\Delta t)\cdot V - C_2(t)\cdot V &= q(t)\cdot C_1(t)\cdot \Delta t - q(t)\cdot C_2(t)\cdot \Delta t. \end{aligned} \tag{41}$$

Taking the limit $\Delta t \to 0$, we can write equation (41) as:

$$V\frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right], \tag{42}$$

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right], \tag{43}$$

$$V\frac{dC_2(t)}{dt} = q(t)\left[C_1(t) - C_2(t)\right]. \tag{44}$$

Let us first derive the analytical solution: Only the first tank is filled with salt water $C_0(0) = C_{0,0}$, $C_1(0) = C_2(0) = 0$, and $C_{\text{in}} = 0$. The solution to equation (42) is, as before $C_0(t) = C_{0,0}e^{-t/\tau}$, inserting this equation into equation (43)

we find:

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_{0,0}e^{-t/\tau} - C_1(t)\right], \tag{45}$$

$$\frac{d}{dt}\left[e^{t/\tau}C_1\right] = \frac{C_{0,0}}{\tau}, \tag{46}$$

$$C_1(t) = \frac{C_{0,0}t}{\tau}e^{-t/\tau}. \tag{47}$$

where we have use the technique of integrating factors when going from equation (45) to (46). Inserting equation (47) into equation (44), solving the equation in a similar way as for C_1 we find:

$$V\frac{dC_2(t)}{dt} = q(t)\left[\frac{C_{0,0}t}{\tau}e^{-t/\tau} - C_2(t)\right], \tag{48}$$

$$\frac{d}{dt}\left[e^{t/\tau}C_2\right] = \frac{C_{0,0}t}{\tau}, \tag{49}$$

$$C_2(t) = \frac{C_{0,0}t^2}{2\tau^2}e^{-t/\tau}. \tag{50}$$

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:

$$\frac{d}{dt}\begin{bmatrix}\hat{C}_0(t)\\ \hat{C}_1(t)\\ \hat{C}_2(t)\end{bmatrix} = \begin{bmatrix}\hat{C_{\text{in}}}(t) - \hat{C}_0(t)\\ \hat{C}_0(t) - \hat{C}_1(t)\\ \hat{C}_1(t) - \hat{C}_2(t)\end{bmatrix},$$

$$\frac{d\hat{\mathbf{C}}(t)}{dt} = \mathbf{f}(\hat{\mathbf{C}}, t). \tag{51}$$

Below is an implementation using the Runge Kutta 4. order method:

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

def rk4_step(c_old, c_in, tau, h):
    c_next=[]
    for i in range(len(c_old)):
        k1=h*fm(c_old[i],c_in[i],tau[i])
        k2=h*fm(c_old[i]+0.5*k1,c_in[i],tau[i])
        k3=h*fm(c_old[i]+0.5*k2,c_in[i],tau[i])
        k4=h*fm(c_old[i]+    k3,c_in[i],tau[i])
        c_next.append(c_old[i]+(k1+2*k2+2*k3+k4)/6)
    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
            # put concentration of tank 0 into tank 1 etc.
            for i,ci in enumerate(c_old[:len(c_old)-1]):
                c_in[i+1]=ci
            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)
```

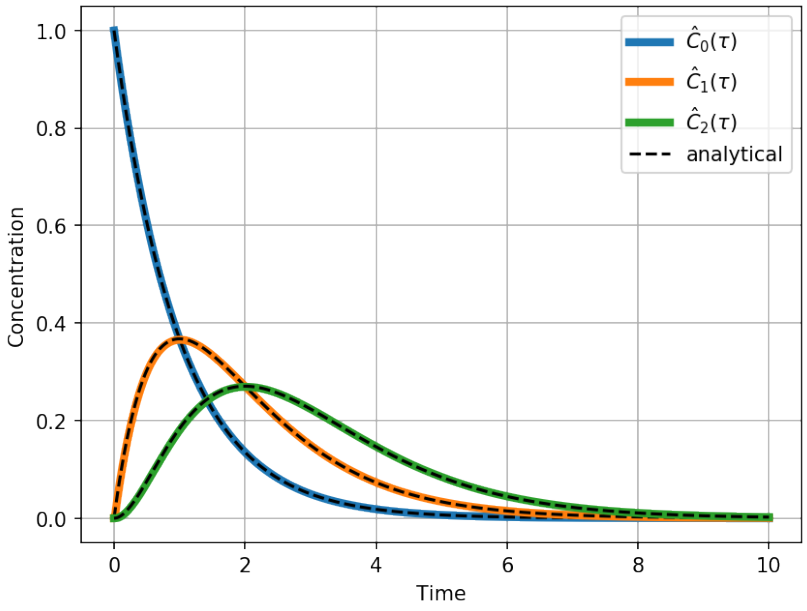

Figure 9: The concentration in the tanks.

In figure 9 the result of the implementation is shown.

6 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.

The governing equations can be found by requiring mass balance for each of the tanks (see equation (1)):

$$\begin{aligned} C_0(t+\Delta t)\cdot V_0 - C_0(t)\cdot V_0 &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t. \\ C_1(t+\Delta t)\cdot V_1 - C_1(t)\cdot V_1 &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t. \end{aligned} \tag{52}$$

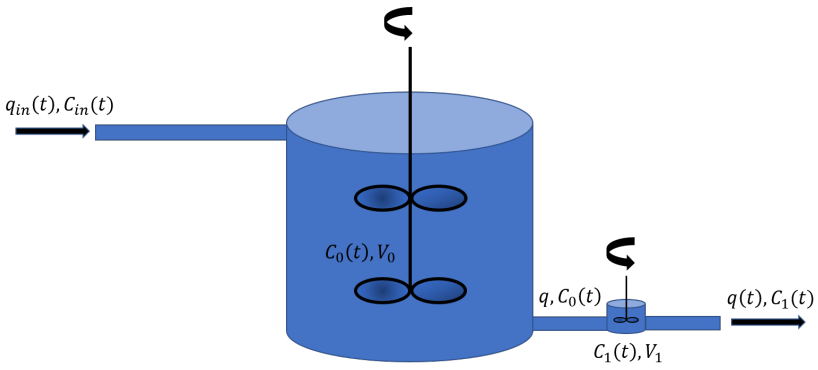

Figure 10: A continuous stirred tank model with a sampling vessel.

Taking the limit $\Delta t \to 0$, we can write equation (52) as:

$$V_0 \frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right]. \tag{53}$$

$$V_1 \frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right]. \tag{54}$$

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 (53)) is:

$$C_0(t) = C_{0,0}e^{-t/\tau_0}, \tag{55}$$

where $\tau_0 \equiv V_0/q$. Inserting this equation into equation (54), we get:

$$\frac{dC_1(t)}{dt} = \frac{1}{\tau_1}\left[C_{0,0}e^{-t/\tau_0} - C_1(t)\right],$$

$$\frac{d}{dt}\left[e^{t/\tau_2}C_1\right] = \frac{C_{0,0}}{\tau_1}e^{-t(1/\tau_0 - 1/\tau_1)}, \tag{56}$$

$$C_1(t) = \frac{C_{0,0}}{1-\frac{\tau_1}{\tau_0}}\left[e^{-t/\tau_0} - e^{-t/\tau_1}\right], \tag{57}$$

where $\tau_1 \equiv V_1/q$.

Next, we will consider the numerical solution. You might think that these equations are more simple to solve numerically than the equations with three tanks in series discussed in the previous section. Actually, this system is much harder to solve with the methods we have discussed so far. The reason is that there are now *two time scales* in the system, τ_1 and τ_2. The smaller tank sets a strong limitation on the step size we can use, because we should never use step sizes larger than a tank volume. Thus if you use the code in the previous section to solve equation (53) and (54), it will not find the correct solution, unless the step size is lower than 10^{-3}. Equations of this type are known as *stiff*.

Stiff equations.

There is no precise definition of "stiff", but it is used to describe a system of differential equations, where the numerical solution becomes unstable unless a very small step size is chosen. Such systems occurs because there are several (length, time) scales in the system, and the numerical solution is constrained by the shortest length scale. You should always be careful on how you scale your variables in order to make the system dimensionless, which is of particular importance when you use adaptive methods.

These types of equations are often encountered in practical applications. If our sampling tank was extremely small, maybe 10^6 smaller than the chemical reactor, then we would need a step size of the order of 10^{-8} or lower to solve the system. This step size is so low that we easily run into trouble with round off errors in the computer. In addition the simulation time is extremely long. How do we deal with this problem? The solution is actually quite simple. The reason we run into trouble is that we require that the concentration leaving the tank must be a small perturbation of the old one. This is not necessary, and it is best illustrated with Eulers method. As explained earlier Eulers method can be viewed as a two step process: first we inject a volume (and remove an equal amount: $qC(t)\Delta t$), and then we mix. Clearly when we try to remove more than what is left, we run into trouble. What we want to do is to remove or flood much more than one tank volume through the tank during one time step, this can be achieved by $q(t)C(t)\Delta t \rightarrow q(t+\Delta t)C(t+\Delta t)\Delta t$. The term $q(t+\Delta t)C(t+\Delta t)\Delta t$ now represents *the mass out of the system during the time step* Δt.

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. It does not mean that we will get a more accurate solution, actually explicit methods are usually more accurate.

Explicit and Implicit methods.

Explicit methods are often called *forward* methods, as they use only information from the previous step to estimate the next value. The explicit methods are easy to implement, but get into trouble if the step size is too large. Implicit methods are often called *backward* methods as the next step cannot be calculated directly from the previous solution, usually a non-linear equation has to be solved. Implicit methods are generally much more stable, but the price is often lower accuracy. Many commercial simulators uses implicit methods extensively because they are stable, and

stability is often viewed as a much more important criterion than numerical accuracy.

Let us consider our example further, and for simplicity use the implicit Eulers method:

$$\begin{aligned} C_{0n+1}V_0 - C_{0n}V_0 &= q(t+\Delta t)C_{\mathrm{in}n+1}\Delta t - q(t+\Delta t)C_{0n+1}\Delta t. \\ C_{1n+1}V_1 - C_{1n}V_1 &= q(t+\Delta t)C_{0n+1}\Delta t - q(t+\Delta t)C_{1n+1}\Delta t. \end{aligned} \tag{58}$$

This equation is equal to equation (52), but the concentrations on the right hand side are now evaluated at the next time step. The immediate problem is now that we have to find an expression for C_{n+1} that is given in terms of known variables. In most cases one needs to use a root finding method, like Newtons method, in order to solve equation (58). In this case it is straight forward to show:

$$\begin{aligned} C_{0n+1} &= \frac{C_{0n} + \frac{\Delta t}{\tau_0} C_{\mathrm{in}n+1}}{1 + \frac{\Delta t}{\tau_0}}, \\ C_{2n+1} &= \frac{C_{1n} + \frac{\Delta t}{\tau_1} C_{0n+1}}{1 + \frac{\Delta t}{\tau_1}}. \end{aligned} \tag{59}$$

Below is an implementation

```
def fm(c_old,c_in,tau,h):
    return (c_old+c_in*h/tau)/(1+h/tau)

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.
        for i,ci in enumerate(c_old[:len(c_old)-1]):
            c_in[i+1]=ci
        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)
```

In figure 11 the result of the implementation is shown.

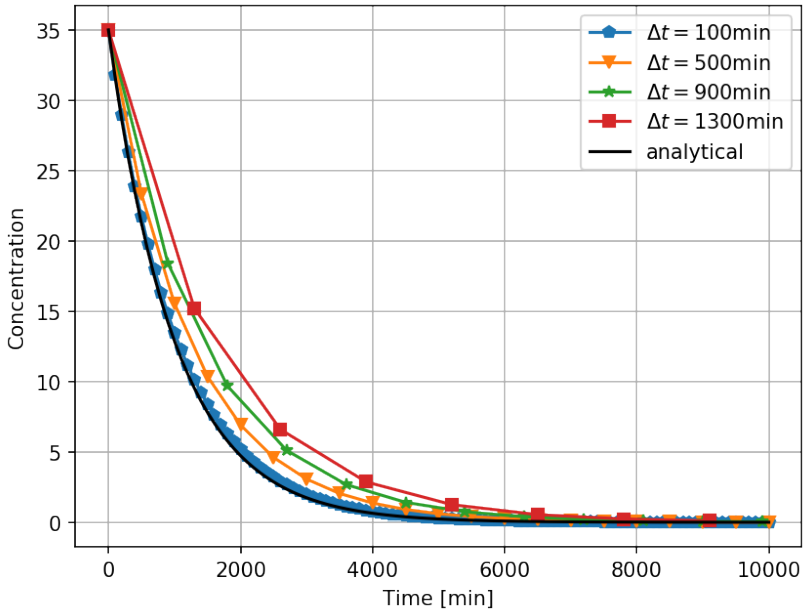

Figure 11: The concentration in the tanks for $h = 0.01$.

Exercise 1: Truncation Error in Eulers Method

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 (19) and (20). The true solution $y(t)$, obeys the following equation:

$$\frac{dy}{dt} = f(y, t), \tag{60}$$

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

$$y_1^* = y_0 + hf(y_0, t_0), \tag{61}$$

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 (19).

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

$$\begin{aligned}\epsilon^* = y(t_0 + h) - y_1^* &= y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ &- [y_0 + hf(y_0, t_0 + h)]\,,\end{aligned} \tag{62}$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (61). Using equation (60) to replace $y'(t_0)$ with $f(y_0, t_0)$, we find:

$$\epsilon^* = y(t_0 + h) - y_1^* = \frac{1}{2}y''(t_0)h^2 \equiv ch^2, \tag{63}$$

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:

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{64}$$

Answer.

$$y_{1/2} = y_0 + \frac{h}{2}f(y_0, t_0), \tag{65}$$

$$y_1 = y_{1/2} + \frac{h}{2}f(y_{1/2}, t_0 + h/2), \tag{66}$$

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{67}$$

Note that we have inserted equation (65) into equation (66) to arrive at equation (67).

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.

$$\epsilon = y(t_0 + h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3)$$
$$- \left[y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2)\right]. \tag{68}$$

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):

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0)$$
$$+ \frac{h}{2}\left[f(y_0, t_0)\left.\frac{\partial f}{\partial y}\right|_{y=y_0, t=t_0} + \frac{h}{2}\left.\frac{\partial f}{\partial t}\right|_{y=y_0, t=t_0}\right] + \mathcal{O}(h^2). \tag{69}$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (60):

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y}f(y,t) + \frac{\partial f(y,t)}{\partial t}. \tag{70}$$

Hence, equation (69) can be written:

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2}y''(t_0, y_0), \tag{71}$$

hence the truncation error in equation (68) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4}y''(y_0, t_0) = \frac{1}{2}ch^2, \tag{72}$$

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

$$\begin{aligned}\epsilon^* = y(t_0 + h) - y_1^* = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + hf(y_0, t_0 + h)\right],\end{aligned} \tag{73}$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (61). Using equation (60) to replace $y'(t_0)$ with $f(y_0, t_0)$, we find:

$$\epsilon^* = y(t_0 + h) - y_1^* = \frac{1}{2}y''(t_0)h^2 \equiv ch^2, \tag{74}$$

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:

$$y_{1/2} = y_0 + \frac{h}{2}f(y_0, t_0), \tag{75}$$

$$y_1 = y_{1/2} + \frac{h}{2}f(y_{1/2}, t_0 + h/2), \tag{76}$$

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{77}$$

Note that we have inserted equation (75) into equation (76) to arrive at equation (77). The truncation error in this case is, as before:

$$\begin{aligned}\epsilon = y(t_0 + h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2)\right].\end{aligned} \tag{78}$$

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):

$$\begin{aligned}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) \\ + \frac{h}{2}\left[f(y_0, t_0)\left.\frac{\partial f}{\partial y}\right|_{y=y_0, t=t_0} + \left.\frac{\partial f}{\partial t}\right|_{y=y_0, t=t_0}\right] + \mathcal{O}(h^2).\end{aligned} \tag{79}$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (60):

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y}f(y,t) + \frac{\partial f(y,t)}{\partial t}. \tag{80}$$

Hence, equation (79) can be written:

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2}y''(t_0, y_0), \tag{81}$$

hence the truncation error in equation (78) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4}y''(y_0, t_0) = \frac{1}{2}ch^2, \tag{82}$$