Ordinary differential equations 1

Explicit techniques for ODEs

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2023-2024

Today's outline

Today's outline

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Overview

Ordinary differential equations

An equation containing a function of one independent variable and its derivatives, in contrast to a *partial differential equation*, which contains derivatives with respect to more independent variables.

Overview

Ordinary differential equations

An equation containing a function of one independent variable and its derivatives, in contrast to a *partial differential equation*, which contains derivatives with respect to more independent variables.

Main question

How to solve

$$\frac{d\boldsymbol{y}}{dx} = f(\boldsymbol{y}(x), x) \quad \text{with} \quad \boldsymbol{y}(x=0) = \boldsymbol{y}_0$$

accurately and efficiently?

What is an ODE?

- Algebraic equation:

$$f(y(x), x) = 0 \qquad \text{e.g.} -\ln(K_{eq}) = (1 - \zeta)$$

- First order ODE:

$$f\left(\frac{dy}{dx}(x), y(x), x\right) = 0 \quad \text{e.g.} \frac{dc}{dt} = -kc^n$$

- Second order ODE:

$$f\left(\frac{d^2y}{dx^2}(x), \frac{dy}{dx}(x), y(x), x\right) = 0 \quad \text{e.g.} \quad \mathcal{D}\frac{d^2c}{dx^2} = -\frac{kc}{1 + Kc}$$

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

Example

Recall:

$$\mathcal{D}\frac{d^2c}{dx^2} = -\frac{kc}{1+Kc}$$

Define $y = -\mathcal{D}\frac{dc}{dx}$, then $\frac{dy}{dx} = \frac{kc}{1+Kc}$, thus solve system:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}y$$
$$\frac{dy}{dx} = \frac{kc}{1+Kc}$$

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

More general

Consider the second order ODE:

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

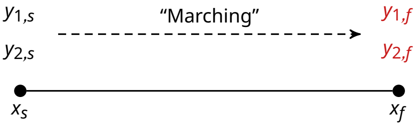
Now define and solve using z as a new variable:

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

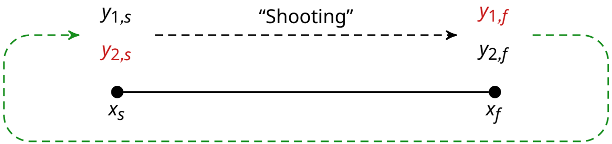
Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

- *Initial value problems (IVP)*
  We know the values of all y_i at some starting position x_s, and it is desired to find the values of y_i at some final point x_f.



- *Boundary value problems (BVP)*
  Boundary conditions are specified at more than one x. Typically, some of the BC are specified at x_s and the remainder at x_f.



Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Boundary value problems

- Shooting method

Today's outline

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Easiest solution algorithm: Euler's method, derived here via Taylor series expansion:

$$c(t_0 + \Delta t) \approx c(t_0) + \left.\frac{dc}{dt}\right|_{t_0} \Delta t + \frac{1}{2} \left.\frac{d^2c}{dt^2}\right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Easiest solution algorithm: Euler's method, derived here via Taylor series expansion:

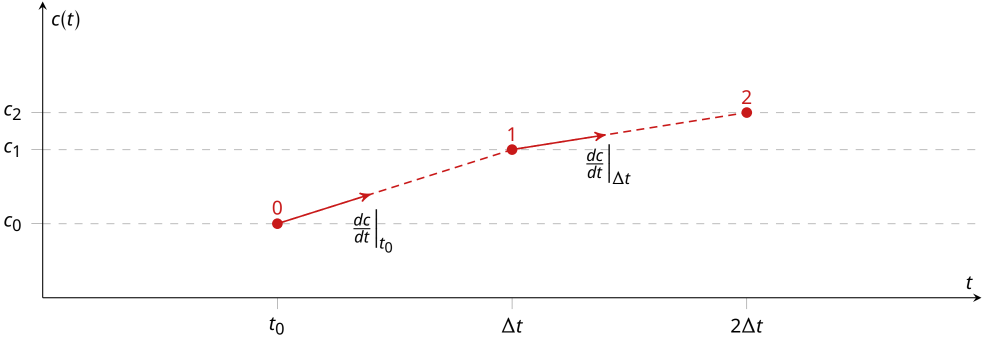
$$c(t_0 + \Delta t) \approx c(t_0) + \left.\frac{dc}{dt}\right|_{t_0} \Delta t + \frac{1}{2} \left.\frac{d^2c}{dt^2}\right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

Neglect terms with higher order than two: $\left.\frac{dc}{dt}\right|_{t_0} = \frac{c(t_0+\Delta t)-c(t_0)}{\Delta t}$ Substitution:

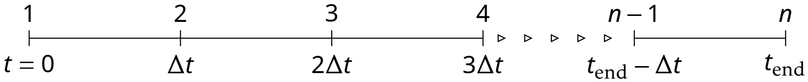
$$\frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t} = f(c_0, t_0) \Rightarrow c(t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$$

Euler's method: graphical example

$$\frac{c(t_0+\Delta t)-c(t_0)}{\Delta t}=f(c_0,t_0)\Rightarrow c(t_0+\Delta t)=c(t_0)+\Delta t f(c_0,t_0)$$



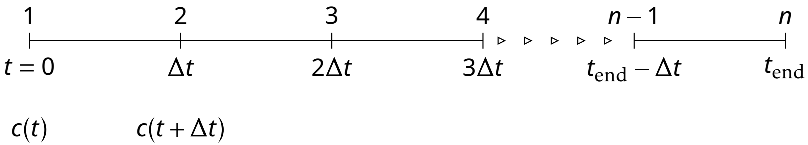
Euler's method - solution method



c_0

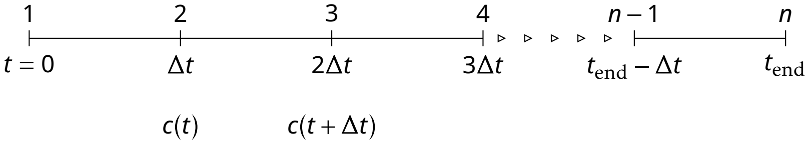
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



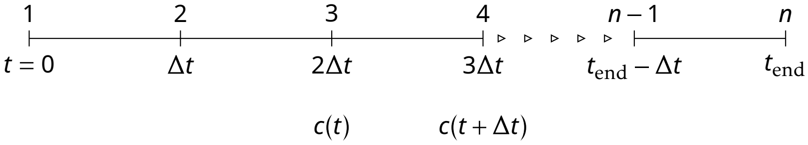
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



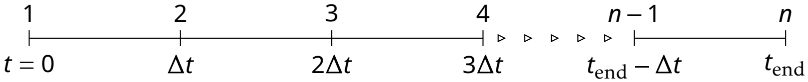
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Pseudo-code Euler's method: $\frac{dy}{dx} = f(x,y)$ and $y(x_0) = y_0$.

1. Initialize variables, functions; set $h = \frac{x_1 - x_0}{N}$
2. Set $x = x_0$, $y = y_0$
3. While $x < x_{\text{end}}$ do
   $x_{i+1} = x_i + h; \quad y_{i+1} = y_i + hf(x_i, y_i)$

Euler's method - example

First order reaction in a batch reactor:

$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1\,\mathrm{mol\,m^{-3}}, \quad k = 1\,\mathrm{s^{-1}}, \quad t_{\mathrm{end}} = 2\,\mathrm{s}$$

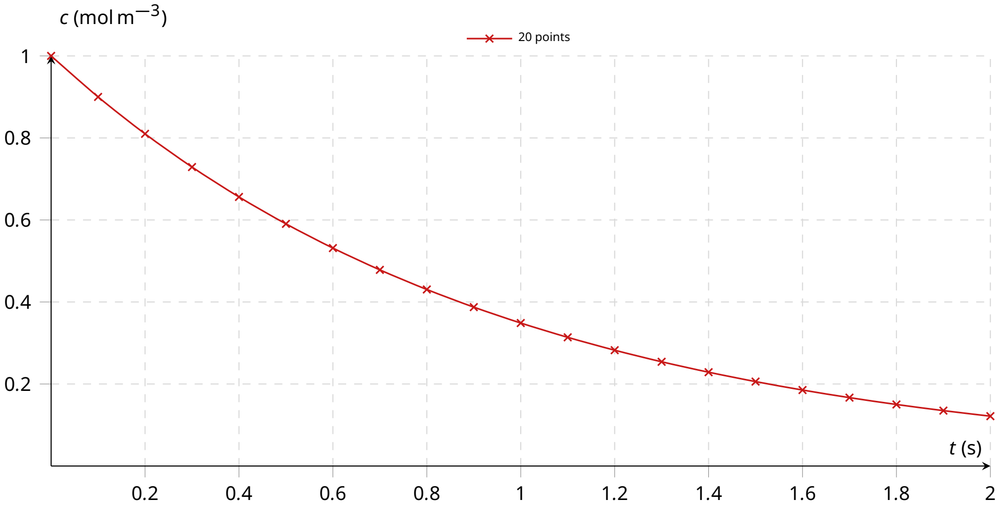
Euler's method - example

First order reaction in a batch reactor:

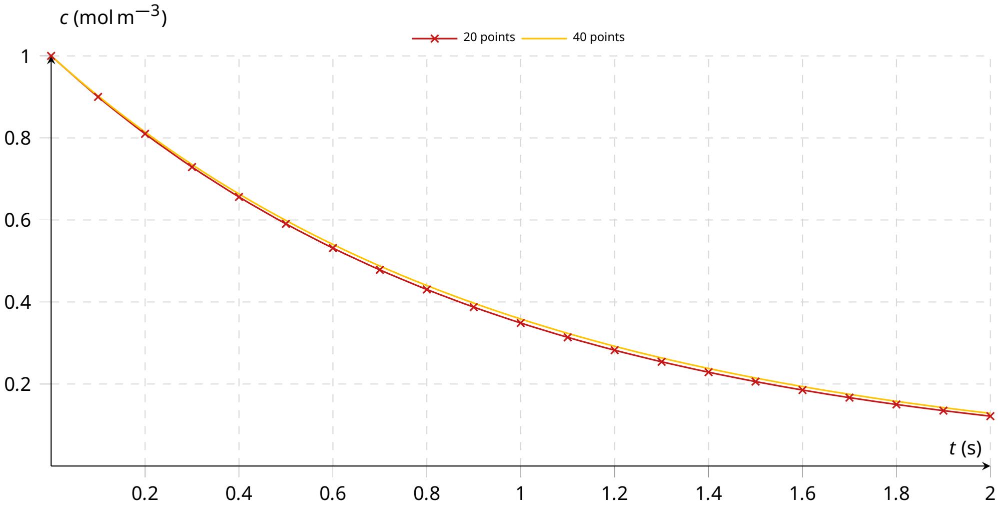
$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1\,\mathrm{mol\,m^{-3}}, \quad k = 1\,\mathrm{s^{-1}}, \quad t_{\mathrm{end}} = 2\,\mathrm{s}$$

Time [s]	Concentration [$\mathrm{mol\,m^{-3}}$]
$t_0 = 0$	$c_0 = 1.00$
$t_1 = t_0 + \Delta t$	$c_1 = c_0 + \Delta t \cdot (-kc_0)$
$= 0 + 0.1 = 0.1$	$= 1 + 0.1 \cdot (-1 \cdot 1) = 0.9$
$t_2 = t_1 + \Delta t$	$c_2 = c_1 + \Delta t \cdot (-kc_1)$
$= 0.1 + 0.1 = 0.2$	$= 0.9 + 0.1 \cdot (-1 \cdot 0.9) = 0.81$
$t_3 = t_2 + \Delta t$	$c_3 = c_2 + \Delta t \cdot (-kc_2)$
$= 0.2 + 0.1 = 0.3$	$= 0.81 + 0.1 \cdot (-1 \cdot 0.81) = 0.729$
$\dots$	$\dots$
$t_{i+1} = t_i + \Delta t$	$c_{i+1} = c_i + \Delta t \cdot (-kc_i)$
$\dots$	$\dots$
$t_{20} = 2.0$	$c_{20} = c_{19} + \Delta t \cdot (-kc_{19}) = 0.121577$

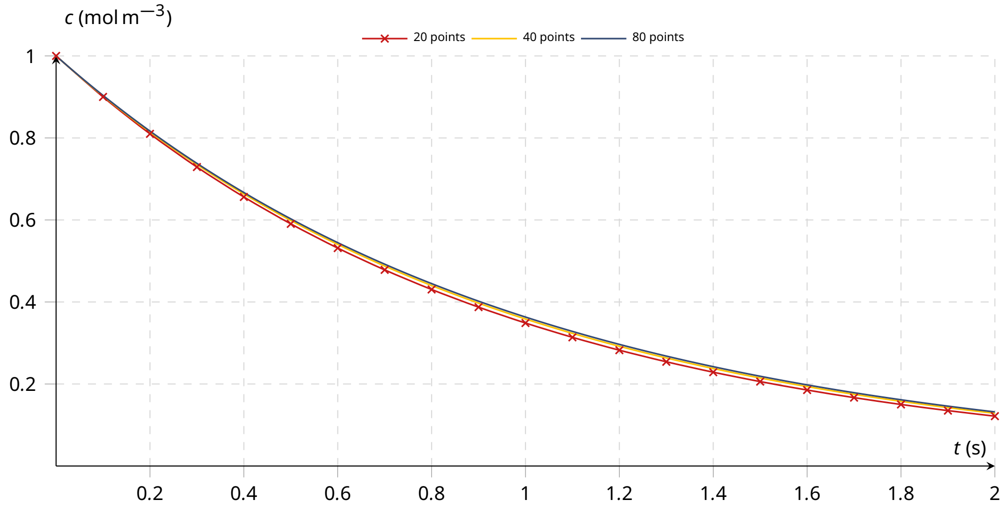
Euler's method - example



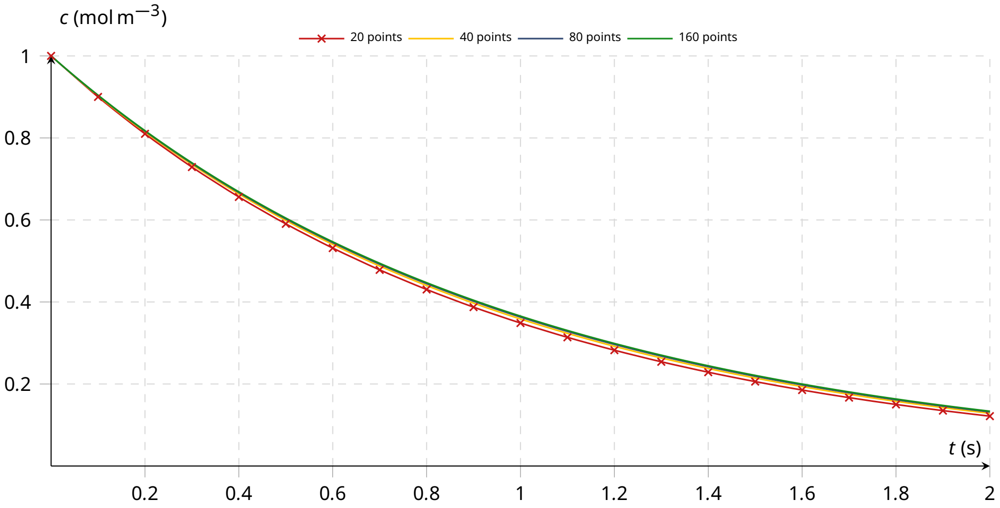
Euler's method - example



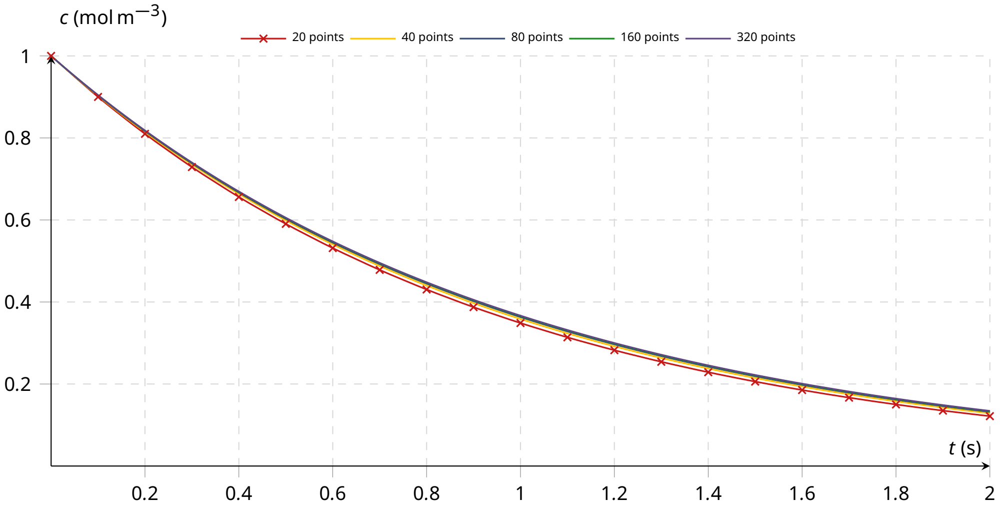
Euler's method - example



Euler's method - example



Euler's method - example



Euler's method - implementation

A basic function of Euler's method is given in `ode_scalar_explicit.py`:

```python
def euler_basic(func, c0, t0, tend, n=100):
   dt = (tend - t0)/n
   t,c = t0, c0
   print(f"t: {t:1.2f}, c: {c:1.6f}")
   for i in range(n):
      k1 = func(c, t)
      t += dt
      c += dt*k1
      print(f"t: {t:1.2f}, c: {c:.6f}")
```

Euler's method - implementation

A basic function of Euler's method is given in `ode_scalar_explicit.py`:

```python
def euler_basic(func, c0, t0, tend, n=100):
    dt = (tend - t0)/n
    t,c = t0, c0
    print(f"t: {t:1.2f}, c: {c:1.6f}")
    for i in range(n):
        k1 = func(c, t)
        t += dt
        c += dt*k1
        print(f"t: {t:1.2f}, c: {c:.6f}")
```

We define the ODE function to be solved, e.g. $\frac{dc}{dt} = -kc$ with $k = 1$, and pass it as an argument to `euler_basic`:

```python
def first_order_react(c,t):
    dcdt = -c
    return dcdt

euler_basic(first_order_react, 1, 0, 2, 100)
```

```
t: 0.00, c: 1.000000
t: 0.20, c: 0.800000
t: 0.40, c: 0.640000
t: 0.60, c: 0.512000
t: 0.80, c: 0.409600
t: 1.00, c: 0.327680
t: 1.20, c: 0.262144
t: 1.40, c: 0.209715
t: 1.60, c: 0.167772
t: 1.80, c: 0.134218
t: 2.00, c: 0.107374
```

Euler's method - implementation

By storing the intermediate results we can return the results for post processing:

```python
import numpy as np
def euler(func, c0, t0, tend, n=100):
   dt = (tend - t0)/n
   t = np.linspace(t0,tend,n+1)
   c = np.zeros(n+1)
   c[0] = c0
   for i in range(n):
      k1 = func(c[i], t[i])
      c[i+1] = c[i] + dt*k1
   return c,t
```

Euler's method - implementation

By storing the intermediate results we can return the results for post processing:

```python
import numpy as np
def euler(func, c0, t0, tend, n=100):
    dt = (tend - t0)/n
    t = np.linspace(t0,tend,n+1)
    c = np.zeros(n+1)
    c[0] = c0
    for i in range(n):
        k1 = func(c[i], t[i])
        c[i+1] = c[i] + dt*k1
    return c,t
```

The function `euler` can be imported from `ode_scalar_explicit.py`:

```python
from ode_scalar_explicit import euler
c, t = euler(first_order_react, 1, 0, 2, 100)
print(np.vstack([t,c]).T)
```

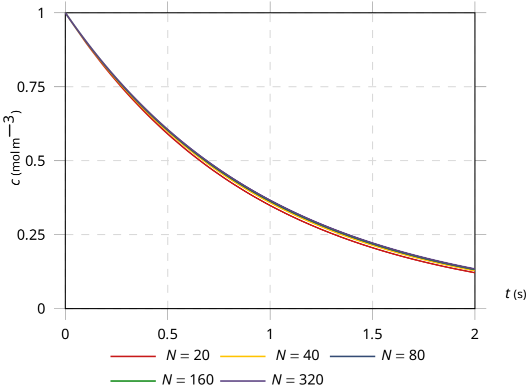
Alternatively, we can pass a *lambda function* in-place:

```python
c, t = euler(lambda c, t: -1.0*c, 1, 0, 2, 100)
```

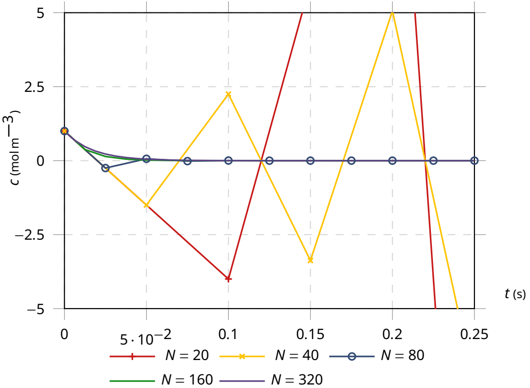
Problems with Euler's method

The question is: What step size, or how many steps to use?

1. *Accuracy* $\Rightarrow$ need information on numerical error!
2. *Stability* $\Rightarrow$ need information on stability limits!



Reaction rate: $k = 1\ \mathrm{s}^{-1}$



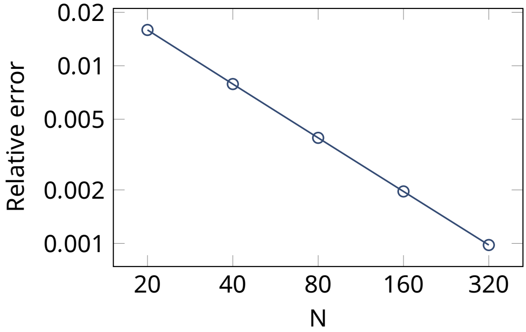
Reaction rate: $k = 50\ \mathrm{s}^{-1}$

Accuracy

Comparison with analytical solution for $k = 1\ \mathrm{s}^{-1}$:

$$c(t) = c_0 \exp(-kt) \Rightarrow \zeta = 1 - \exp(-kt) \Rightarrow \zeta_{\text{analytical}} = 0.864665$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$
20	0.878423	0.015912
40	0.871488	0.007891
80	0.868062	0.003929
160	0.866360	0.001961
320	0.865511	0.000979



Accuracy

For Euler's method: Error halves when the number of grid points is doubled, i.e. error is proportional to Δt: first order method.

Error estimate:

$$\left.\frac{dx}{dt}\right|_{t_0} = \frac{x(t_0+\Delta t)-x(t_0)}{\Delta t} + \frac{1}{2}\left.\frac{d^2x}{dt^2}\right|_{t_0}(\Delta t) + \mathcal{O}(\Delta t)^2$$

$$\frac{x(t_0+\Delta t)-x(t_0)}{\Delta t} = f(x_0,t_0) - \frac{1}{2}\left.\frac{d^2x}{dt^2}\right|_{t_0}(\Delta t) + \mathcal{O}(\Delta t)^2$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

What to do when there is no analytical solution available?

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

What to do when there is no analytical solution available? Compare to calculations with different number of steps: $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r:

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1}\right)^r \Rightarrow \log\left(\frac{\epsilon_2}{\epsilon_1}\right) = \log\left(\frac{\Delta x_2}{\Delta x_1}\right)^r$$

$$\Rightarrow \quad r \quad = \quad \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{\Delta x_2}{\Delta x_1}\right)} = \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{N_1}{N_2}\right)} \quad \text{in the limit of} \quad \Delta x \to 0 \quad \text{or} \quad N \to \infty$$

Today's outline

Errors and convergence rate

L_2 norm (Euclidean norm)

$$||\mathbf{v}||_2 = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

L_∞ norm (maximum norm)

$$||\mathbf{v}||_\infty = \max\left(|v_1|, \ldots, |v_n|\right)$$

Absolute difference

$$\epsilon_{\text{abs}} = \left|\left|\mathbf{y}_{\text{numerical}} - \mathbf{y}_{\text{analytical}}\right|\right|_{2,\infty}$$

Relative difference

$$\epsilon_{\text{rel}} = \left|\left|\frac{\mathbf{y}_{\text{numerical}} - \mathbf{y}_{\text{analytical}}}{\mathbf{y}_{\text{analytical}}}\right|\right|_{2,\infty}$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

Computing the rate of convergence

When the analytical solution is available, choose 1 or 2 for a particular number of grid points N:

1. Compute the relative or absolute error vector $\bar{\varepsilon}$. Take the norm to compute a single error value ϵ following:
   - Based on L_1-norm: $\epsilon = \dfrac{||\bar{\varepsilon}||_1}{N}$
   - Based on L_2-norm: $\epsilon = \dfrac{||\bar{\varepsilon}||_2}{\sqrt{N}}$
   - Based on L_∞-norm: $\epsilon = ||\bar{\varepsilon}||_\infty$
2. Compute the relative or absolute error at a single indicative points (e.g. middle of domain, outlet).

Computing the rate of convergence

When the analytical solution is available, choose ① or ② for a particular number of grid points N:

① Compute the relative or absolute error vector $\overline{\varepsilon}$. Take the norm to compute a single error value ϵ following:

- Based on L_1-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_1}{N}$
- Based on L_2-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_2}{\sqrt{N}}$
- Based on L_∞-norm: $\epsilon = \|\overline{\varepsilon}\|_\infty$

② Compute the relative or absolute error at a single indicative points (e.g. middle of domain, outlet).

Compare to calculations with different number of steps: $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r:

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1}\right)^r \Rightarrow \log\left(\frac{\epsilon_2}{\epsilon_1}\right) = \log\left(\frac{\Delta x_2}{\Delta x_1}\right)^r$$

$$\Rightarrow r = \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{\Delta x_2}{\Delta x_1}\right)} = \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{N_1}{N_2}\right)} \quad \text{in the limit of} \quad \Delta x \to 0 \quad \text{or} \quad N \to \infty$$

Computing the rate of convergence

When the analytical solution is **not** available:

1. Compute the solution with $N+1$, N, $N-1$ and $N-2$ grid points
2. Select a single indicative grid point (e.g. middle of domain, outlet) that lies at exactly the same position in each computation
3. Use the solution c at this grid point for various grid sizes to compute:

$$r = \frac{\log \dfrac{c_{N+1} - c_N}{c_N - c_{N-1}}}{\log \dfrac{c_N - c_{N-1}}{c_{N-1} - c_{N-2}}}$$

4. Alternative for simulations with $2N$, N and $\frac{N}{2}$ grid points:

$$r = \frac{\log \left| \dfrac{c_{2N} - c_N}{c_N - c_{\frac{N}{2}}} \right|}{\log \left| \dfrac{N}{2N} \right|}$$

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Wouldn't it be great to have a method that can give the answer using much less steps?

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Wouldn't it be great to have a method that can give the answer using much less steps? $\Rightarrow$ Higher order methods

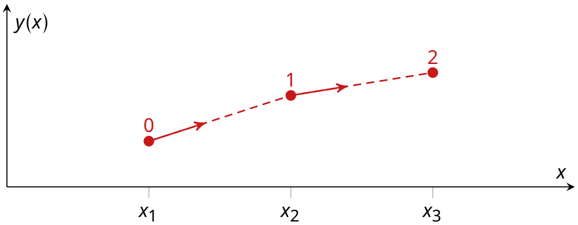
Today's outline

Runge-Kutta methods

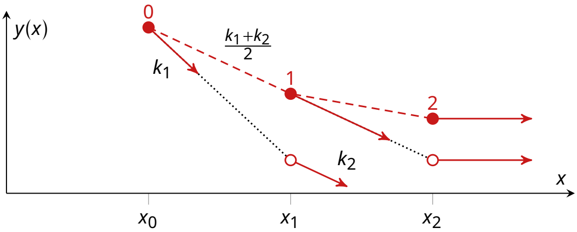
Propagate a solution by combining the information of several Euler-style steps (each involving one function evaluation) to match a Taylor series expansion up to some higher order.

Euler: $y_{i+1} = y_i + hf(x_i, y_i)$ with $h = \Delta x$, i.e. slope $= k_1 = f(x_i, y_i)$.

Euler's method



RK2 method



Classical second order Runge-Kutta (RK2) method

This method is also called Heun's method, or improved Euler method:

1. Approximate the slope at x_i: $k_1 = f(x_i, y_i)$
2. Approximate the slope at x_{i+1}: $k_2 = f(x_{i+1}, y_{i+1})$ where we use Euler's method to approximate $y_{i+1} = y_i + hf(x_i, y_i) = y_i + hk_1$
3. Perform an Euler step with the average of the slopes: $y_{i+1} = y_i + h\frac{1}{2}(k_1 + k_2)$

Classical second order Runge-Kutta (RK2) method

This method is also called Heun's method, or improved Euler method:

1. Approximate the slope at x_i: $k_1 = f(x_i, y_i)$
2. Approximate the slope at x_{i+1}: $k_2 = f(x_{i+1}, y_{i+1})$ where we use Euler's method to approximate $y_{i+1} = y_i + hf(x_i, y_i) = y_i + hk_1$
3. Perform an Euler step with the average of the slopes: $y_{i+1} = y_i + h\frac{1}{2}(k_1 + k_2)$

In pseudocode:

$x = x_0$, $y = y_0$
while $x < x_{\text{end}}$ **do**
  $x_{i+1} = x_i + h$
  $k_1 = f(x_i, y_i)$
  $k_2 = f(x_i + h, y_i + hk_1)$
  $y_{i+1} = y_i + h\frac{1}{2}(k_1 + k_2)$
end while

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Using Taylor series expansion: $y_{i+1} = y_i + h \left.\frac{dy}{dx}\right|_i + \frac{h^2}{2} \left.\frac{d^2y}{dx^2}\right|_i + \mathcal{O}(h^3)$

$$\left.\frac{dy}{dx}\right|_i = f(x_i, y_i) \equiv f_i$$

$$\left.\frac{d^2y}{dx^2}\right|_i = \left.\frac{d}{dx} f(x, y(x))\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i \left.\frac{\partial y}{\partial x}\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \quad \text{(chain rule)}$$

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Using Taylor series expansion: $y_{i+1} = y_i + h \left.\frac{dy}{dx}\right|_i + \frac{h^2}{2} \left.\frac{d^2y}{dx^2}\right|_i + \mathcal{O}(h^3)$

$$\left.\frac{dy}{dx}\right|_i = f(x_i, y_i) \equiv f_i$$

$$\left.\frac{d^2y}{dx^2}\right|_i = \left.\frac{d}{dx} f(x, y(x))\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i \left.\frac{\partial y}{\partial x}\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \quad \text{(chain rule)}$$

Substitution gives:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2}\left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

$$y_{i+1} = y_i + \frac{h}{2} f_i + \frac{h}{2}\left(f_i + h \left.\frac{\partial f}{\partial x}\right|_i + h f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)$$

Runge-Kutta methods — derivation

Note multivariate Taylor expansion:

$$f(x_i+h, y_i+k) = f_i + h\left.\frac{\partial f}{\partial x}\right|_i + k\left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\Rightarrow \frac{h}{2}\left(f_i + h\left.\frac{\partial f}{\partial x}\right|_i + hf_i\left.\frac{\partial f}{\partial y}\right|_i\right) = \frac{h}{2}f(x_i+h, y_i+hf_i) + \mathcal{O}(h^3)$$

Concluding:

$$y_{i+1} = y_i + \frac{h}{2}f_i + \frac{h}{2}f(x_i+h, y_i+hf_i) + \mathcal{O}(h^3)$$

Rewriting:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i+h, y_i+hk_1) \\ \Rightarrow y_{i+1} &= y_i + \frac{h}{2}(k_1+k_2) \end{aligned}$$

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned} y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\ &= y_i + h\left[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\right] + \mathcal{O}(h^3) \\ &= y_i + h\left[b_1 f_i + b_2 \left\{f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)\right\}\right] + \mathcal{O}(h^3) \\ &= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i\right) + \mathcal{O}(h^3) \end{aligned}$$

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned}
y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\right] + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 \left\{ f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2) \right\}\right] + \mathcal{O}(h^3) \\
&= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)
\end{aligned}$$

Comparison with Taylor:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

Using $b_1 + b_2 = 1$, $c_2 b_2 = \frac{1}{2}$, $a_{2,1} b_2 = \frac{1}{2}$ $\Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned}
y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\right] + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 \left\{f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)\right\}\right] + \mathcal{O}(h^3) \\
&= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i\right) + \mathcal{O}(h^3)
\end{aligned}$$

Comparison with Taylor:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2}\left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i\right) + \mathcal{O}(h^3)$$

Using $b_1 + b_2 = 1$, $c_2 b_2 = \frac{1}{2}$, $a_{2,1} b_2 = \frac{1}{2}$ $\Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

Runge-Kutta methods — derivation

$$y_{i+1} = y_i + h(b_1 + b_2)f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)$$

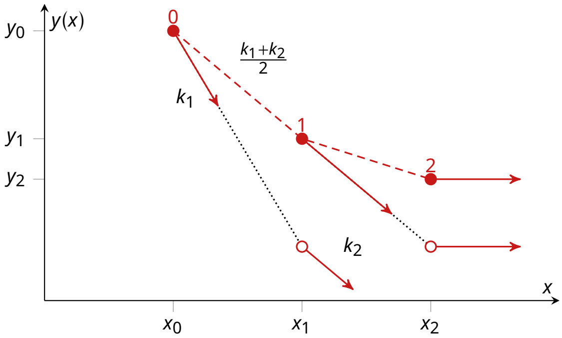
$$y_{i+1} = y_i + hf_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

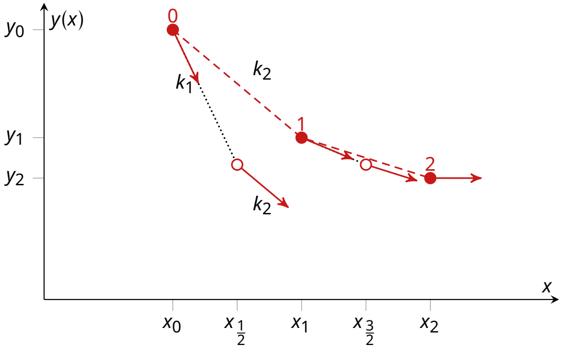
$\Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

1. Classical RK2:
   $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$
2. Midpoint rule (modified Euler):
   $b_1 = 0,\ b_2 = 1,\ c_2 = a_{2,1} = \frac{1}{2}$

Second order Runge-Kutta methods

Classical RK2 method (= Heun's method, improved Euler method)	Explicit midpoint rule (modified Euler method)
$k_1 = f_i$	$k_1 = f_i$
$k_2 = f(x_i + h, y_i + hk_1)$	$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$
$y_{i+1} = y_i + \frac{1}{2}h(k_1 + k_2)$	$y_{i+1} = y_i + hk_2$

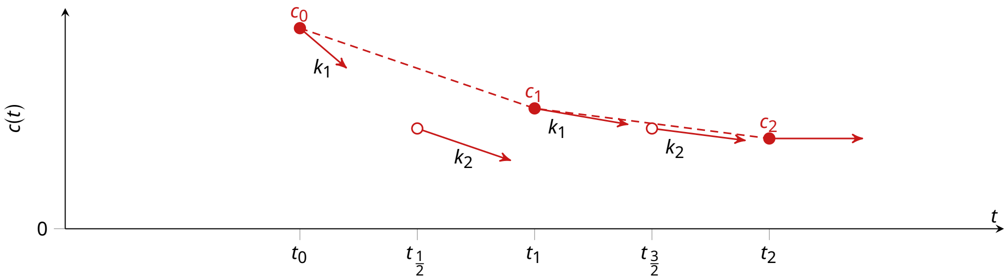




Second order Runge-Kutta method — Example

First order reaction in a batch reactor: $\frac{dc}{dt} = -kc$ with $c(t=0) = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{s^{-1}}$, $t_{\mathrm{end}} = 2$ s.

Time [s]	C [$\mathrm{mol\,m^{-3}}$]	$k_1 = hf(x_i, y_i)$	$k_2 = hf(x_i + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$
0	1.00	$0.1 \cdot (-1 \cdot 1) = -0.1$	$0.1 \cdot (-1 \cdot (1 - 0.5 \cdot 0.1)) = -0.095$
0.1	$1 - 0.095 = 0.905$	$0.1 \cdot (-1 \cdot 0.0905) = -0.0905$	$0.1 \cdot (-1 \cdot (0.905 - 0.5 \cdot 0.0905)) = -0.085975$
...	...	...	...
2	0.1358225	-0.0135822	-0.0129031



RK2 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864178	5.634×10^{-4}	—
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048×10^{-6}	2.007

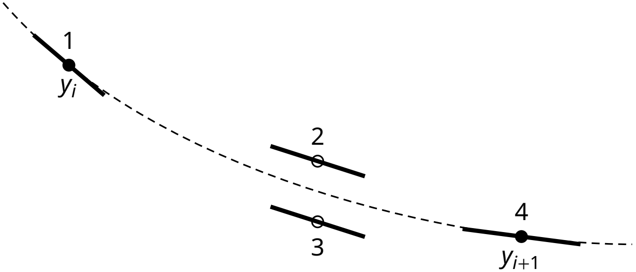
RK2 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864178	5.634×10^{-4}	—
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048×10^{-6}	2.007

$\Rightarrow$ RK2 is a second order method. Doubling the number of cells reduces the error by a factor 4!

Can we do even better?

RK4 method (classical fourth order Runge-Kutta method)



$$k_1 = f(x_i, y_i)$$

$$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$$

$$k_3 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2)$$

$$k_4 = f(x_i + h, y_i + hk_3)$$

$$y_{i+1} = y_i + h\left(\frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4\right)$$

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864664472	2.836×10^{-7}	—
40	0.864664702	1.700×10^{-8}	4.060
80	0.864664716	1.040×10^{-9}	4.030
160	0.864664717	6.435×10^{-11}	4.015
320	0.864664717	4.001×10^{-12}	4.007

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864664472	2.836×10^{-7}	—
40	0.864664702	1.700×10^{-8}	4.060
80	0.864664716	1.040×10^{-9}	4.030
160	0.864664717	6.435×10^{-11}	4.015
320	0.864664717	4.001×10^{-12}	4.007

$\Rightarrow$ RK4 is a fourth order method: Doubling the number of cells reduces the error by a factor 16!

Can we do even better?

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Python

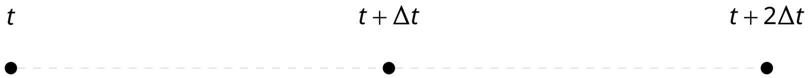
Adaptive step size control

The step size (be it either position, time or both (PDEs)) cannot be decreased indefinitely to favour a higher accuracy, since each additional grid point causes additional computation time. It may be wise to adapt the step size according to the computation requirements.

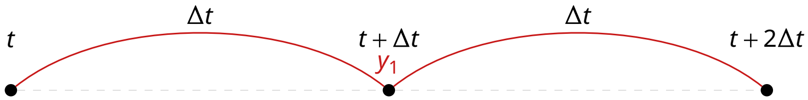
Globally two different approaches can be used:

1. Step doubling: compare solutions when taking one full step or two consecutive halve steps
2. Embedded methods: Compare solutions when using two approximations of different order

Adaptive step size control: step doubling

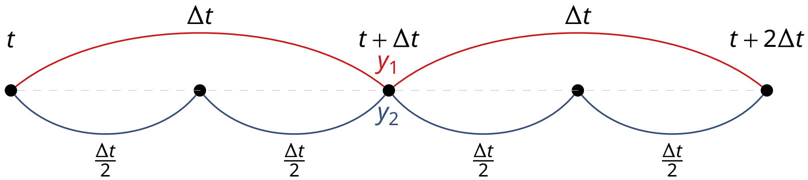


Adaptive step size control: step doubling



- RK4 with one large step of h: $y_{i+1} = y_1 + ch^5 + \mathcal{O}(h^6)$

Adaptive step size control: step doubling



- RK4 with one large step of h: $y_{i+1} = y_1 + ch^5 + \mathcal{O}(h^6)$
- RK4 with two steps of $\frac{1}{2}h$: $y_{i+1} = y_2 + 2c(\frac{1}{2}h)^5 + \mathcal{O}(h^6)$

Adaptive step size control: step doubling

- Estimation of truncation error by comparing y_1 and y_2:
  $\Delta = y_2 - y_1$
- If Δ too large, reduce step size for accuracy
- If Δ too small, increase step size for efficiency.
- Ignoring higher order terms and solving for c:
  $\Delta = \frac{15}{16} ch^5 \Rightarrow ch^5 = \frac{16}{15}\Delta \Rightarrow y_{i+1} = y_2 + \frac{\Delta}{15} + \mathcal{O}(h^6)$
  (local Richardson extrapolation)

Note that when we specify a tolerance *tol*, we can estimate the maximum allowable step size as: $h_{\text{new}} = \alpha h_{\text{old}} \left| \frac{\text{tol}}{\Delta} \right|^{\frac{1}{5}}$ with α a safety factor (typically $\alpha = 0.9$).

Adaptive step size control: embedded methods

Use a special fourth and a fifth order Runge Kutta method to approximate y_{i+1}

- The fourth order method is special because we want to use the same positions for the evaluation for computational efficiency.
- RK45 is the preferred method (minimum number of function evaluations) (this is the default method in `scipy.integrate.solve_ivp`).

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Python

Solving ODEs in Python

SciPy provides convenient procedures to solve (systems of) ODEs automatically.

The procedure is as follows:

1. Create a function that specifies the ODE(s). Specifically, this function returns the $\frac{dy}{dx}$ value (vector).
2. Initialise solver variables and settings (e.g. step size, initial conditions, tolerance)
3. Call the ODE solver function, passing the ODE function as argument
    - The ODE solver will return a solution oject (e.g. `sol`), with attribute `sol.t` as the independent variable vector, and a `sol.y` the solution vector (or matrix for systems of ODEs).

Solving ODEs in Python: example 1

We solve the system: $\frac{dx}{dt} = -k_1 x + k_2, k_1 = 0.2, k_2 = 2.5$

- Create a lambda function:

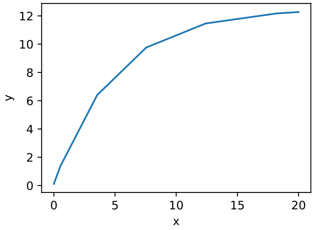
```
dydx = lambda x,y: (-0.2*y + 2.5)
```

- Solve with a call to `solve_ivp(function, timespan, initial_condition)`:

```
from scipy.integrate import solve_ivp
sol = solve_ivp(dydx, tspan, y0)
```

- Draw the results by calling the relevant Matplotlib commands:

```
import matplotlib.pyplot as plt
plt.plot(sol.t, sol.y[0,:])
plt.show()
```



Solving ODEs in Python: example 2

We solve the system: $\dfrac{dx}{dt} = \begin{cases} -\dfrac{k_1}{x^2} & t \leq 10 \\ \dfrac{k_2}{x} - \dfrac{k_1}{x^2} & t > 10 \end{cases}$ with $k_1 = 0.5, k_2 = 1, x(0) = 2$

Create an ODE function

```
def myEqnFunction(t,x):
    k1 = 0.5;
    k2 = 1;
    dxdt = int(t>10)*k2/x - k1/x**2;
    return dxdt
```

Solving ODEs in Python: example 2

We solve the system: $$\frac{dx}{dt} = \begin{cases} -\frac{k_1}{x^2} & t \leq 10 \\ \frac{k_2}{x} - \frac{k_1}{x^2} & t > 10 \end{cases} \quad \text{with } k_1 = 0.5, k_2 = 1, x(0) = 2$$

Create an ODE function

```python
def myEqnFunction(t,x):
   k1 = 0.5;
   k2 = 1;
   dxdt = int(t>10)*k2/x - k1/x**2;
   return dxdt
```

Create a solution script

```python
tspan = [0, 20]
x_init = [2]
sol = solve_ivp(myEqnFunction, tspan, x_init, rtol=1e-8, atol=1e-6)
```

Solving ODEs in Python: example 2

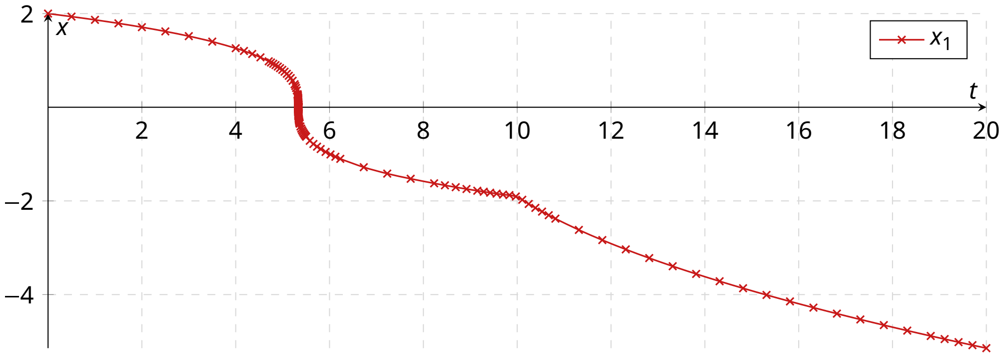
Plot the solution:

```
plt.plot(sol.t, sol.y[0,:],'r-x')
plt.grid()
plt.show()
```

Solving ODEs in Python: example 2

Plot the solution:

```
plt.plot(sol.t, sol.y[0,:],'r-x')
plt.grid()
plt.show()
```



Note the refinement in regions where large changes occur.

Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
func = lambda t,x,k1,k2: k1*x+k2
# or
def func(t,x,k1,k2):
   return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as* `args` *list*:

```
sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
func = lambda t,x,k1,k2: k1*x+k2
# or
def func(t,x,k1,k2):
    return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as* `args` *list*:

```
sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Of course, in the solver script, the variables do not have to be called `k1` and `k2`:

```
sol = solve_ivp(func,[0,5],[1],args=(q, u))
```

These variables may be of any type (scalar, vector, dictionary, list). For carrying over many variables, a dictionary is useful and descriptive.

Solving systems of ODEs in Python: example

You have noticed that the step size in t varies. This is because we have given just the begin and end times of our time span:

```
tspan = [0, 5];
```

Solving systems of ODEs in Python: example

You have noticed that the step size in t varies. This is because we have given just the begin and end times of our time span:

```
tspan = [0, 5];
```

You can also obtain the solution at specific points, by supplying a list `t_eval`:

```
sol = solve_ivp(func,tspan,[1],args=(some_k1, some_k2),t_eval=np.linspace(tspan[0],tspan[1],31))
```

This example provides 31 explicit time steps between 0 and 5 seconds. Note that the results are interpolated to these data points afterwards; you do not influence the efficiency and accuracy of the solver algorithm this way!

Ordinary differential equations 2

Implicit methods, systems of ODEs and boundary value problems

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2023-2024

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Python
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Today's outline

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

First order approximation of derivative: $\frac{dy}{dx} = \frac{y_{i+1} - y_i}{\Delta x}$.

Where to evaluate the function f?

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

First order approximation of derivative: $\frac{dy}{dx} = \frac{y_{i+1} - y_i}{\Delta x}$.

Where to evaluate the function f?

1. Evaluation at x_i: Explicit Euler method (forward Euler)
2. Evaluation at x_{i+1}: Implicit Euler method (backward Euler)

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - kc_i\Delta t \Rightarrow \frac{c_{i+1}}{c_i} = 1 - k\Delta t$$

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - kc_i\Delta t \Rightarrow \frac{c_{i+1}}{c_i} = 1 - k\Delta t$$

It follows that unphysical results are obtained for $k\Delta t \geq 1$!!

Stability requirement

$$k\Delta t < 1$$

(but probably accuracy requirements are more stringent here!)

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k c_{i+1} \Delta t \Rightarrow \frac{c_{i+1}}{c_i} = \frac{1}{1+k\Delta t}$$

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - kc_{i+1}\Delta t \Rightarrow \frac{c_{i+1}}{c_i} = \frac{1}{1+k\Delta t}$$

This equation does never give unphysical results!
The implicit Euler method is *unconditionally stable*
(but maybe not very accurate or efficient).

Semi-implicit Euler method

Usually f is a non-linear function of y, so that linearization is required (recall Newton's method).

$$\frac{dy}{dx} = f(y) \Rightarrow y_{i+1} = y_i + hf(y_{i+1}) \quad \text{using} \quad f(y_{i+1}) = f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i) + \ldots$$

$$\Rightarrow y_{i+1} = y_i + h\left[f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)\right]$$

$$\Rightarrow \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_i + hf(y_i)$$

$$\boxed{\Rightarrow \quad y_{i+1} \quad = \quad y_i \quad + \quad h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(y_i)}$$

Semi-implicit Euler method

Usually f is a non-linear function of y, so that linearization is required (recall Newton's method).

$$\frac{dy}{dx} = f(y) \Rightarrow y_{i+1} = y_i + hf(y_{i+1}) \quad \text{using} \quad f(y_{i+1}) = f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i) + \dots$$

$$\Rightarrow y_{i+1} = y_i + h\left[f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)\right]$$

$$\Rightarrow \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_i + hf(y_i)$$

$$\boxed{\Rightarrow \quad y_{i+1} \quad = \quad y_i \quad + \quad h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(y_i)}$$

For the case that $f(x, y(x))$ we could add the variable x as an additional variable $y_{n+1} = x$. Or add one fully implicit Euler step (which avoids the computation of $\frac{\partial f}{\partial x}$):

$$y_{i+1} = y_i + hf(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(x_{i+1}, y_i)$$

Implicit Euler's method - implementation

A basic function of the implicit Euler method is given in `ode_scalar_implicit.py`:

```
def implicit_euler(func, c0, t0, tend, n):
    h = 1e-8
    dt = (tend - t0)/n
    times = np.linspace(t0,tend,n+1)
    c = np.zeros(n+1)
    c[0] = c0
    for i,t in enumerate(times[:-1]):
        f = func(c[i],t)
        fh = func(c[i]+h,t)
        dfdc = (fh - f)/h
        c[i+1] = c[i] + dt*f/(1 - dt*dfdc)
        print(f"{t=:0.4f}, c: {c[i+1]:.8f}")
    print(f"t={times[-1]:0.4f}, c: {c[-1]:.8f}")
    return times, c
```

Implicit Euler's method - implementation

A basic function of the implicit Euler method is given in `ode_scalar_implicit.py`:

```
def implicit_euler(func, c0, t0, tend, n):
    h = 1e-8
    dt = (tend - t0)/n
    times = np.linspace(t0,tend,n+1)
    c = np.zeros(n+1)
    c[0] = c0
    for i,t in enumerate(times[:-1]):
        f = func(c[i],t)
        fh = func(c[i]+h,t)
        dfdc = (fh - f)/h
        c[i+1] = c[i] + dt*f/(1 - dt*dfdc)
        print(f"{t=:0.4f}, c: {c[i+1]:.8f}")
    print(f"t={times[-1]:0.4f}, c: {c[-1]:.8f}")
    return times, c
```

```
from ode_scalar_implcit import implicit_euler
t,c = implicit_euler(lambda c,t: -1.0*c**2, 1, 0, 2, 10)
plt.plot(t,c,'-o',label='Implicit Euler')
print(f"Conversion = {conv_e}")
```

Implicit Euler's method - implementation

A basic function of the implicit Euler method is given in `ode_scalar_implicit.py`:

```python
def implicit_euler(func, c0, t0, tend, n):
    h = 1e-8
    dt = (tend - t0)/n
    times = np.linspace(t0,tend,n+1)
    c = np.zeros(n+1)
    c[0] = c0
    for i,t in enumerate(times[:-1]):
        f = func(c[i],t)
        fh = func(c[i]+h,t)
        dfdc = (fh - f)/h
        c[i+1] = c[i] + dt*f/(1 - dt*dfdc)
        print(f"{t=:0.4f}, c: {c[i+1]:.8f}")
    print(f"t={times[-1]:0.4f}, c: {c[-1]:.8f}")
    return times, c
```

```python
from ode_scalar_implcit import implicit_euler
t,c = implicit_euler(lambda c,t: -1.0*c**2, 1, 0, 2,
     10)
plt.plot(t,c,'-o',label='Implicit Euler')
print(f"Conversion = {conv_e}")
```

```
t=0.0000, c: 0.85714286
t=0.2000, c: 0.74772036
t=0.4000, c: 0.66164680
t=0.6000, c: 0.59241445
t=0.8000, c: 0.53566997
t=1.0000, c: 0.48840819
t=1.2000, c: 0.44849689
t=1.4000, c: 0.41438638
t=1.6000, c: 0.38492630
t=1.8000, c: 0.35924657
t=2.0000, c: 0.35924657
Conversion = 0.64075343
```

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc \Rightarrow c_{i+1} = c_i - \frac{hkc_i^2}{1+2hkc_i}$.

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc \Rightarrow c_{i+1} = c_i - \frac{hkc_i^2}{1+2hkc_i}$.

N	ζ	$\frac{\zeta_{\mathrm{numerical}}-\zeta_{\mathrm{analytical}}}{\zeta_{\mathrm{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.654066262	1.89×10^{-2}	—
40	0.660462687	9.31×10^{-3}	1.02220
80	0.663589561	4.62×10^{-3}	1.01162
160	0.665134433	2.30×10^{-3}	1.00594
320	0.665902142	1.15×10^{-3}	1.00300

Second order implicit method: Implicit midpoint method

Implicit midpoint rule (second order)	Explicit midpoint rule (modified Euler method)
$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, \frac{1}{2}(y_i + y_{i+1})\right)$	$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$

in case $f(y)$ then:

$$f\left(\frac{1}{2}(y_i + y_{i+1})\right) = f_i + \left.\frac{df}{dy}\right|_i \left(\frac{1}{2}(y_i + y_{i+1}) - y_i\right) = f_i + \frac{1}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

Second order implicit method: Implicit midpoint method

Implicit midpoint rule (second order)	Explicit midpoint rule (modified Euler method)
$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, \frac{1}{2}(y_i + y_{i+1})\right)$	$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$

in case $f(y)$ then:

$$f\left(\frac{1}{2}(y_i + y_{i+1})\right) = f_i + \left.\frac{df}{dy}\right|_i \left(\frac{1}{2}(y_i + y_{i+1}) - y_i\right) = f_i + \frac{1}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

Implicit midpoint rule reduces to:

$$y_{i+1} = y_i + hf_i + \frac{h}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

$$\Rightarrow \left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right) y_i + hf_i$$

$$\boxed{\Rightarrow \quad y_{i+1} \quad = \quad y_i \; + \; h\left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right)^{-1} f_i}$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\,\mathrm{mol\,m^{-3}}$, $k = 1\,\mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\,\mathrm{s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Substitution:

$$
\begin{aligned}
c_{i+1} &= c_i + h\left(1 - \frac{h}{2}\cdot(-2kc_i)\right)^{-1}\cdot(-kc_i^2) \\
&= c_i - \frac{hkc_i^2}{1+hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1+hkc_i} \Rightarrow c_{i+1} = \frac{c_i}{1+hkc_i}
\end{aligned}
$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Substitution:

$$
\begin{aligned}
c_{i+1} &= c_i + h\left(1 - \frac{h}{2}\cdot(-2kc_i)\right)^{-1}\cdot(-kc_i^2) \\
&= c_i - \frac{hkc_i^2}{1+hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1+hkc_i} \Rightarrow c_{i+1} = \frac{c_i}{1+hkc_i}
\end{aligned}
$$

You will find that this method is exact for all step sizes h because of the quadratic source term!

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

$$c_{i+1} = \frac{c_i}{1 + hkc_i}$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$

$$c_{i+1} = \frac{c_i}{1+hkc_i}$$

N	ζ	$\frac{\zeta_{\mathrm{numerical}}-\zeta_{\mathrm{analytical}}}{\zeta_{\mathrm{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.6666666667	1.665×10^{-16}	—
40	0.6666666667	0	—
80	0.6666666667	0	—
160	0.6666666667	0	—
320	0.6666666667	0	—

Implicit midpoint method — example

Third order reaction in a batch reactor: $\frac{dc}{dt} = -kc^3$

Analytical solution: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

$$c_{i+1} = c_i - \frac{hkc_i^3}{1 + \frac{3}{2}hkc_i^2}$$

Implicit midpoint method — example

Third order reaction in a batch reactor: $\frac{dc}{dt} = -kc^3$
Analytical solution: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

$$c_{i+1} = c_i - \frac{hkc_i^3}{1 + \frac{3}{2}hkc_i^2}$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.5526916174	1.71×10^{-4}	—
40	0.5527633731	4.17×10^{-5}	2.041
80	0.5527807304	1.03×10^{-5}	2.021
160	0.5527849965	2.55×10^{-6}	2.011
320	0.5527860538	6.34×10^{-7}	2.005

Today's outline

Systems of ODEs

A system of ODEs is specified using vector notation:

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x, \boldsymbol{y}(x))$$

for

$$\frac{dy_1}{dx} = f_1(x, y_1(x), y_2(x)) \quad \text{or} \quad f_1(x, y_1, y_2)$$
$$\frac{dy_2}{dx} = f_2(x, y_1(x), y_2(x)) \quad \text{or} \quad f_2(x, y_1, y_2)$$

Systems of ODEs

A system of ODEs is specified using vector notation:

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x, \boldsymbol{y}(x))$$

for

$$\frac{dy_1}{dx} = f_1(x, y_1(x), y_2(x)) \quad \text{or} \quad f_1(x, y_1, y_2)$$
$$\frac{dy_2}{dx} = f_2(x, y_1(x), y_2(x)) \quad \text{or} \quad f_2(x, y_1, y_2)$$

The solution techniques discussed before can also be used to solve systems of equations.

Systems of ODEs: Explicit methods

Forward Euler method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{f}(x_i, \boldsymbol{y}_i)$$

Improved Euler method (classical RK2)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \frac{h}{2}(\boldsymbol{k}_1 + \boldsymbol{k}_2) \quad \text{using} \quad \begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(x_i + h, \boldsymbol{y}_i + h\boldsymbol{k}_1) \end{aligned}$$

Modified Euler method (midpoint rule)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{k}_2 \quad \text{using} \quad \begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_1) \end{aligned}$$

Systems of ODEs: Explicit methods

Classical fourth order Runge-Kutta method (RK4)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\frac{\boldsymbol{k}_1}{6} + \frac{1}{3}(\boldsymbol{k}_2 + \boldsymbol{k}_3) + \frac{\boldsymbol{k}_4}{6}\right)$$

using

$$\boldsymbol{k}_1 = \boldsymbol{f}(x_i, \boldsymbol{y}_i)$$

$$\boldsymbol{k}_2 = \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_1)$$

$$\boldsymbol{k}_3 = \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_2)$$

$$\boldsymbol{k}_4 = \boldsymbol{f}(x_i + h, \boldsymbol{y}_i + h\boldsymbol{k}_3)$$

Solving systems of ODEs in Python

Solving systems of ODEs in Python is completely analogous to solving a single ODE:

1. Create a function that specifies the ODEs. This function returns the $\frac{d\mathbf{y}}{dx}$ vector.
2. Initialise solver variables and settings (e.g. step size, initial conditions, tolerance), in a separate script. Initial conditions and tolerances should be given per-equation, i.e. as a vector.
3. Call the ODE solver function, using a function argument to the ODE function described in point 1.
   - The ODE solver will return the vector for the independent variable (e.g. time), and a solution matrix, with a column as the solution for each equation in the system.

Solving systems of ODEs in Python: example

We solve the system $\frac{dx_0}{dt} = ax_0 - x_1, \quad \frac{dx_1}{dt} = bx_1 + x_0$, with $a = -1$ and $b = -2$:

- Create an ODE function:

```python
# Example scipy solve_ivp/Example scipy solve_ivp vector.py
def func(t, x, a, b):
    #output can be of list or np.array type:
    dxdt = np.zeros(2)

    dxdt[0] = a*x[0] - x[1]
    dxdt[1] = b*x[1] + x[0]
    return dxdt
```

- Solve by calling `solve_ivp`

```python
from scipy.integrate import solve_ivp
x_init = [0,1]; % Initial conditions
tspan = [0,10]; % Time span
sol = solve_ivp(func, tspan, x_init, args=(-1,-2), rtol=1e-12)
```

Solving systems of ODEs in Python: example

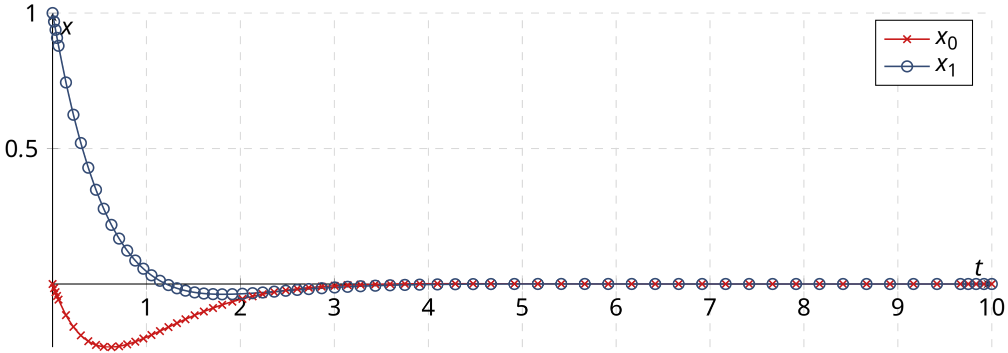
Plot the solution (note: the solution is attribute `sol.y`):

```
import matplotlib.pyplot as plt
plt.plot(sol.t, sol.y[0], 'r-x', linewidth=2)
plt.plot(sol.t, sol.y[1], 'b-o', linewidth=2)
```

Solving systems of ODEs in Python: example

Plot the solution (note: the solution is attribute `sol.y`):

```
import matplotlib.pyplot as plt
plt.plot(sol.t, sol.y[0], 'r-x', linewidth=2)
plt.plot(sol.t, sol.y[1], 'b-o', linewidth=2)
```



Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
func = lambda t,x,k1,k2: k1*x+k2
# or
def func(t,x,k1,k2):
    return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as* `args` *list*:

```
sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
func = lambda t,x,k1,k2: k1*x+k2
# or
def func(t,x,k1,k2):
    return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as* `args` *list*:

```
sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Of course, in the solver script, the variables do not have to be called `k1` and `k2`:

```
sol = solve_ivp(func,[0,5],[1],args=(q, u))
```

These variables may be of any type (scalar, vector, dictionary, list). For carrying over many variables, a dictionary is useful and descriptive.

Solving systems of ODEs in Python: example

You may have noticed that the step size in t varies. This happens when only the begin and end times of the time span are defined, and `scipy.integrate.solve_ivp` uses adaptive step size for efficiency:

```
tspan = [0, 10]
```

Solving systems of ODEs in Python: example

You may have noticed that the step size in t varies. This happens when only the begin and end times of the time span are defined, and `scipy.integrate.solve_ivp` uses adaptive step size for efficiency:

```
tspan = [0, 10]
```

You can also retrieve the solution at specific steps, by supplying all steps explicitly as an additional argument to `solve_ivp`, e.g.:

```
sol = solve_ivp(func, tspan, x_init, args=(-1,-2), t_eval=np.linspace(0, 10, 101), rtol=1e-12)
```

This example provides 101 explicit time steps between 0 and 10 seconds. It can be useful if you need a direct comparison with e.g. measurements at specific times.

Note that this is an interpolated result. The solver uses, in the background, still the adaptive step size functionality!

Systems of ODEs: Implicit methods

Backward Euler method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\boldsymbol{I} - h\left.\frac{d\boldsymbol{f}}{d\boldsymbol{y}}\right|_i\right)^{-1}\boldsymbol{f}(\boldsymbol{y}_i)$$

Implicit midpoint method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\boldsymbol{I} - \frac{h}{2}\left.\frac{d\boldsymbol{f}}{d\boldsymbol{y}}\right|_i\right)^{-1}\boldsymbol{f}(\boldsymbol{y}_i)$$

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method.

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method. For example:

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \qquad \frac{dc_2}{dt} = -999c_1 - 1999c_2$$

with boundary conditions $c_1(t=0) = 1$ and $c_2(t=0) = 0$.
The analytical solution is:

$$c_1 = 2e^{-t} - e^{-1000t} \qquad c_2 = -e^{-t} + e^{-1000t}$$

For the explicit method we require $\Delta t < 10^{-3}$ despite the fact that the term is completely negligible, but essential to keep stability.

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method. For example:

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \qquad \frac{dc_2}{dt} = -999c_1 - 1999c_2$$

with boundary conditions $c_1(t = 0) = 1$ and $c_2(t = 0) = 0$.
The analytical solution is:

$$c_1 = 2e^{-t} - e^{-1000t} \qquad c_2 = -e^{-t} + e^{-1000t}$$

For the explicit method we require $\Delta t < 10^{-3}$ despite the fact that the term is completely negligible, but essential to keep stability.

> The "disease" of stiff equations: we need to follow the solution on the shortest length scale to maintain stability of the integration, although accuracy requirements would allow a much larger time step.

Demonstration with example

Forward Euler (explicit)

$$\frac{c_{1,i+1} - c_{1,i}}{dt} = 998c_{1,i} + 1998c_{2,i}$$
$$\frac{c_{2,i+1} - c_{2,i}}{dt} = -999c_{1,i} - 1999c_{2,i}$$

$$\Rightarrow \begin{array}{l} c_{1,i+1} = (1 + 998\Delta t)\, c_{1,i} + 1998\Delta t c_{2,i} \\ c_{2,i+1} = -999\Delta t c_{1,i} + (1 - 1999\Delta t)\, c_{2,i} \end{array}$$

Demonstration with example

Backward Euler (implicit)

$$\frac{c_{1,i+1} - c_{1,i}}{\Delta t} = 998c_{1,i+1} + 1998c_{2,i+1}$$
$$\frac{c_{2,i+1} - c_{2,i}}{\Delta t} = -999c_{1,i+1} - 1999c_{2,i+1}$$

$$\Rightarrow \begin{array}{l} (1 - 998\Delta t)\, c_{1,i+1} - 1998\Delta t c_{2,i} = c_{1,i} \\ 999\Delta t c_{1,i+1} + (1 + 999\Delta t)\, c_{2,i+1} = c_{2,i} \end{array}$$

Demonstration with example

Backward Euler (implicit)

$$\frac{c_{1,i+1} - c_{1,i}}{\Delta t} = 998c_{1,i+1} + 1998c_{2,i+1}$$
$$\frac{c_{2,i+1} - c_{2,i}}{\Delta t} = -999c_{1,i+1} - 1999c_{2,i+1}$$

$$\Rightarrow \begin{array}{l} (1 - 998\Delta t)\, c_{1,i+1} - 1998\Delta t c_{2,i} = c_{1,i} \\ 999\Delta t c_{1,i+1} + (1 + 999\Delta t)\, c_{2,i+1} = c_{2,i} \end{array}$$

$$A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i \text{ with } A = \begin{pmatrix} 1 - 998\Delta t & -1998\Delta t \\ 999\Delta t & 1 + 1999\Delta t \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$$

Demonstration with example

$$\text{Backward Euler (implicit) } A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i \text{ with } A = \begin{pmatrix} 1 - 998\Delta t & -1998\Delta t \\ 999\Delta t & 1 + 1999\Delta t \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$$

Demonstration with example

Backward Euler (implicit) $A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i$ with $A = \begin{pmatrix} 1-998\Delta t & -1998\Delta t \\ 999\Delta t & 1+1999\Delta t \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$

Cramers rule:

$$c_{1,i+1} = \frac{\begin{vmatrix} c_{1,i} & -1998\Delta t \\ c_{2,i} & 1+1999\Delta t \end{vmatrix}}{\det |A|} = \frac{(1+1999\Delta t)c_{1,i}+1998\Delta t c_{2,i}}{(1-998\Delta t)(1+1999\Delta t)+1998 \cdot 999\Delta t^2}$$

$$c_{2,i+1} = \frac{\begin{vmatrix} 1-998\Delta t & c_{1,i} \\ 999\Delta t & c_{2,i} \end{vmatrix}}{\det |A|} = \frac{-999\Delta t c_{1,i}+(1-998\Delta t)c_{2,i}}{(1-998\Delta t)(1+1999\Delta t)+1998 \cdot 999\Delta t^2}$$

Forward Euler: $\Delta t \leq 0.001$ for stability

Backward Euler: always stable, even for $\Delta t > 100$ (but then not very accurate!)

Demonstration with example

Cure for stiff problems: use implicit methods! To find out whether your system is stiff: check whether one of the eigenvalues have an imaginary part

Implicit methods in Python

SciPy offers a solver that detects stiff systems, using `method='LSODA'`.

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \quad \frac{dc_2}{dt} = -999c_1 - 1999c_2, c_1(0) = 1, c_2(0) = 0$$

- Create the ode function (see `slide_example_solve_ivp_implicit.py`)

```
function [dcdt] = stiff_ode(t,c)
dcdt = zeros(2,1); % Pre-allocation
dcdt(1) = 998 * c(1) + 1998*c(2);
dcdt(2) = -999 * c(1) - 1999*c(2);
return
```

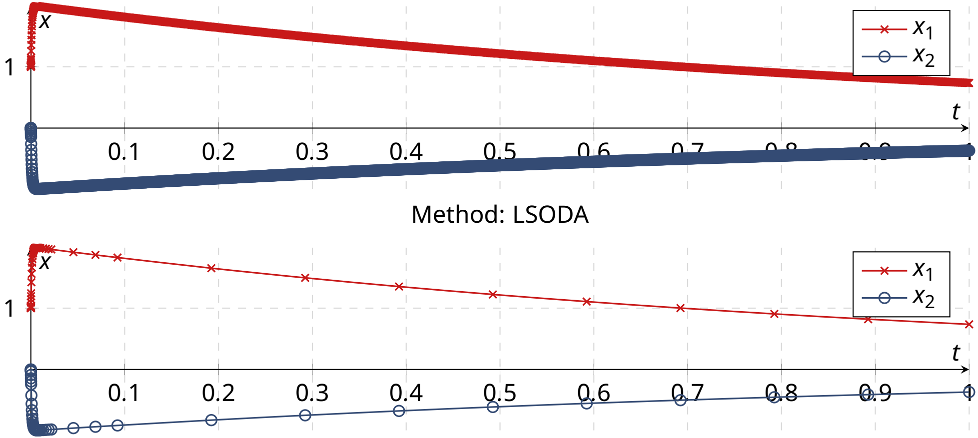
- Compare the resolution of the solutions (see next slide)

```
sol1 = solve_ivp(stiff_ode, [0, 1], [1, 0])
# plot sol1
sol2 = solve_ivp(stiff_ode, [0, 1], [1, 0],method = 'LSODA')
# plot sol2
```

Implicit methods in Python

Default settings

Method: LSODA



The explicit solver requires 1245 data points (default settings), the implicit solver just 48!

Implicit methods in Matlab: Generic backward Euler

```python
def euler_implicit(func, c0, t0, tend, n):
    dt = (tend - t0)/n
    t = np.linspace(t0, tend, num=n+1, endpoint=True)
    c0 = np.asarray(c0, dtype=float)
    c = np.zeros((n+1, c0.size))
    c[0] = c0
    print(f"t: {t[0]:f}, c: {np.array2string(c[0])}")
    for i in range(n):
        f = func(c[i])
        dfdc = jac(func, c[i])
        dc = np.linalg.solve(np.eye(c0.size) - dt*dfdc, dt*f)
        c[i+1] = c[i] + dc
        print(f"t: {t[i+1]:f}, c: {np.array2string(c[i+1])}")
    return c, t
```

```python
def jac(func, c):
    n = c.size
    jac = np.eye(n)
    h = 1e-8
    f = func(c)
    for i in range(n):
        cs = c[i]
        c[i] = c[i] + h
        fh = func(c)
        jac[:,i] = (fh - f)/h
        c[i] = cs
    return jac
```

Vector output needs a bit of processing:

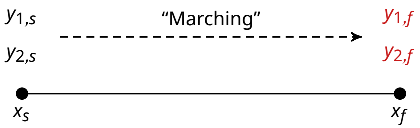
```python
c, t = euler_implicit(func, [1, 0, 0], 0, 2, 100)
c = c.T
fig = plt.figure()
plt.plot(t, c[0], 'ro-', label='A')
plt.plot(t, c[1], 'bs-', label='B')
plt.plot(t, c[2], 'g^-', label='C')
plt.show()
```

Today's outline

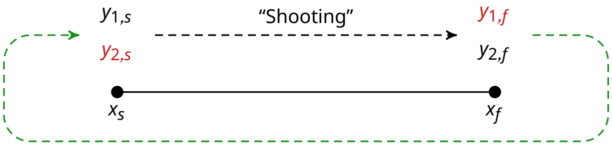
Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

- *Initial value problems (IVP)*
  We know the values of all y_i at some starting position x_s, and it is desired to find the values of y_i at some final point x_f.

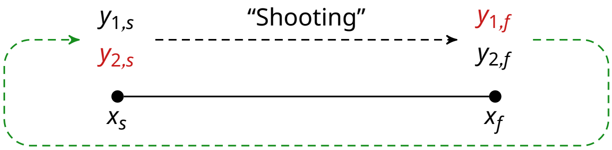


- *Boundary value problems (BVP)*
  Boundary conditions are specified at more than one x. Typically, some of the BC are specified at x_s and the remainder at x_f.



Shooting method

How to solve a BVP using the shooting method:



- Define the system of ODEs
- Provide an initial guess for the unknown boundary condition
- Solve the system and compare the resulting boundary condition to the expected value
- Adjust the guessed boundary value, and solve again. Repeat until convergence.
    - Of course, you can subtract the expected value from the computed value at the boundary, and use a non-linear root finding method

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with } \begin{array}{ll} c(x=0) = C_{A,i,L} = 1 & \text{(interface concentration)} \\ c(x=\delta) = 0 & \text{(bulk concentration)} \end{array}$$

Question: compute the concentration profile in the film layer.

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with} \qquad \begin{aligned} c(x=0) &= C_{A,i,L} = 1 && \text{(interface concentration)} \\ c(x=\delta) &= 0 && \text{(bulk concentration)} \end{aligned}$$

Question: compute the concentration profile in the film layer.

Step 1: Define the system of ODEs

This second-order ODE can be rewritten as a system of first-order ODEs, if we define the flux q as:

$$q = -\mathcal{D}\frac{dc}{dx}$$

Now, we find:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$
$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with} \qquad \begin{aligned} c(x=0) &= C_{A,i,L} = 1 && \text{(interface concentration)} \\ c(x=\delta) &= 0 && \text{(bulk concentration)} \end{aligned}$$

Question: compute the concentration profile in the film layer.

Step 2: Set the boundary conditions

The boundary conditions for the concentrations at $x = 0$ and $x = \delta$ are known.

The flux at the interface, however, is not known, and should be solved for.

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Solving the two first-order ODEs in Excel. First, the cells with constants:

	A	B	C
1	CAiL	1	mol/m3
2	D	1e–8	m2/s
3	kR	10	1/s
4	delta	1e–4	m
5	N	100	
6	dx	=B4/B5	

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Solving the two first-order ODEs in Excel. First, the cells with constants:

	A	B	C
1	CAiL	1	mol/m3
2	D	1e–8	m2/s
3	kR	10	1/s
4	delta	1e–4	m
5	N	100	
6	dx	=B4/B5	

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}} q$$

$$\frac{dq}{dx} = -k_R c$$

Now, we program the forward Euler (explicit) schemes for c and q below:

	A	B	C
10	x	c	q
11	0	=B1	10
12	=A11+$B$6	=B11+$B$6*(–1/$B$2*C11)	=C11+$B$6*(-$B$3*B11)
13	=A12+$B$6	=B12+$B$6*(–1/$B$2*C12)	=C12+$B$6*(-$B$3*B12)
...	...	...	...
111	=A110+$B$6	=B110+$B$6*(–1/$B$2*C110)	=C110+$B$6*(-$B$3*B110)

BVP: example in Excel

- We now have profiles for c and q as a function of position x.
- The concentration $c(x = \delta)$ depends (eventually) on the boundary condition at the interface $q(x = 0)$
- We can use the solver to change $q(x = 0)$ such that the concentration at the bulk meets our requirement: $c(x = \delta) = 0$

BVP: example in Python

We first program the system of ODEs in a separate function:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$
$$\frac{dq}{dx} = -k_R c$$

```python
# slides_example_bvp_1.py
def diffReactSystem(x, y, param):
    c, q = y
    f = np.zeros_like(y)
    f[0] = -q/param['Diff']
    f[1] = -param['kR']*c
    return f
```

BVP: example in Python

We first program the system of ODEs in a separate function:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$
$$\frac{dq}{dx} = -k_R c$$

```python
# slides_example_bvp_1.py
def diffReactSystem(x, y, param):
    c, q = y
    f = np.zeros_like(y)
    f[0] = -q/param['Diff']
    f[1] = -param['kR']*c
    return f
```

Note that we pass a variable (type: dictionary) that contains required parameters: `param`.

BVP: example in Python

Let's first try to solve the ODE system using `scipy.integrate.solve_ivp`:

```
# slides_example_bvp_1.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

### Definition of diffReactSystem here (see slide 151 )

# Set up parameters
q0 = 1e-3 # Initial guess flux@t=0
param = {'cAiL': 1.0,'Diff':1e-8,'kR': 10,'delta': 1e-4,'N': 100}

# Solve ODE system
sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param), # ODE func with params
                             [0, param['delta']], # Time span
                             [param['cAiL'], q0]) # Initial conditions

fig,ax1 = plt.subplots()
ax1.plot(sol.t,sol.y[0,:],'-b',label='Concentration $mol/m^3$')
ax2 = ax1.twinx() # Create y-y axis
ax2.plot(sol.t,sol.y[1,:],'-r',label='Flux $mol/m^2/s$')
fig.legend(bbox_to_anchor=(0.5, 0.5))
plt.show()
```

BVP: example in Python

That seems to work! Now we want to fit the value for q at $x = 0$ (defined below as `bcq`), such that the concentration at $x = \delta$ equals zero. We create a function with the output defined as the deviation from the target value:

```python
# slides_example_bvp_2.py
def diffReactFitCriterium(bcq, param):
   # Solve the ODE system using changeable parameter bcq
   # (boundary condition for q), other parameters are defined in param
   sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param), [0, param['delta']], [param['cAiL'
       ], bcq])
   # Return the last value of the concentration (column 0 in y) at x=delta (hence [-1])
   return sol.y[0,-1] - 0
```

BVP: example in Python

That seems to work! Now we want to fit the value for q at $x = 0$ (defined below as `bcq`), such that the concentration at $x = \delta$ equals zero. We create a function with the output defined as the deviation from the target value:

```
# slides_example_bvp_2.py
def diffReactFitCriterium(bcq, param):
    # Solve the ODE system using changeable parameter bcq
    # (boundary condition for q), other parameters are defined in param
    sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param), [0, param['delta']], [param['cAiL'
        ], bcq])
    # Return the last value of the concentration (column 0 in y) at x=delta (hence [-1])
    return sol.y[0,-1] - 0
```

Note the following:

- We use the interval $0 \leq x \leq \delta$
- Boundary conditions are given as: $c(x = 0) = 1$ and $q(x = 0) =$ `bcq`, which is given as a separate argument to the function (i.e. changable from 'outside'!)
- The function returns the concentration at $x = \delta$

BVP: example in Python

Finally, we should solve the system so that we obtain the right boundary condition $q =$ `bcq` such that $c(x = \delta) = 0$. We can use the `scipy.optimize.root_scalar` function to do this. Extend the script from slide 152 by:

```python
# slides_example_bvp_2.py
from scipy.optimize import root_scalar

### Define diffReactSystem, diffReactFitCriterium, parameters

# Solve such that c(delta)=0:
sol = root_scalar(lambda x: diffReactFitCriterium(x, param),
              method='brentq',bracket=[0,1], xtol=1e-15, rtol=1e-15)
q0 = sol.root
print(f"{q0 = }")

# Solve ODE once more such that we can plot the final data
sol = solve_ivp(lambda x, y: diffReactSystem(x, y,param),
            [0, param['delta']], [param['cAiL'], q0],
            t_eval = np.linspace(0, param['delta'], 101))
```

Postprocessing of the data can be done similar to the example in slide 152.

BVP example: analytical solution

Compare with the analytical solution:

$$q = k_L E_A C_{A,i,L} \quad \text{with}$$

$$E_A = \frac{\mathrm{Ha}}{\tanh \mathrm{Ha}} \qquad \text{(Enhancement factor)}$$

$$\mathrm{Ha} = \frac{\sqrt{k_R \mathcal{D}}}{k_L} \qquad \text{(Hatta number)}$$

$$k_L = \frac{\mathcal{D}}{\delta} \qquad \text{(mass transfer coefficient)}$$

Today's outline

Other methods

Other explicit methods:

- Bulirsch-Stoer method (Richardson extrapolation + modified midpoint method)

Other implicit methods:

- Rosenbrock methods (higher order implicit Runge-Kutta methods)
- Predictor-corrector methods

Summary

- Several solution methods and their derivation were discussed:
  - Explicit solution methods: Euler, Improved Euler, Midpoint method, RK45
  - Implicit methods: Implicit Euler and Implicit midpoint method
  - A few examples of their spreadsheet implementation were shown
- We have paid attention to accuracy and instability, rate of convergence and step size
- Systems of ODEs can be solved by the same algorithms. Stiff problems should be treated with care.
- An example of solving ODEs with Python was demonstrated.