# 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 (6BER03), 2024-2025

# 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

- 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

# 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 $f$?

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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:

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

# 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:

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

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).

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

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

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

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

# 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 $\frac{\partial f}{\partial x}$):

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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`:

```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}")
```

# 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: $c(t) = \frac{c_0}{1+kc_0 t}$

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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

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

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

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

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

TECHNOLOGY

# 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:

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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:

$$c_{i+1} = c_i + h\left(1 - \frac{h}{2}\cdot(-2kc_i)\right)^{-1}\cdot(-kc_i^2)$$
$$= c_i - \frac{hkc_i^2}{1+hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1+hkc_i} \Longrightarrow c_{i+1} = \frac{c_i}{1+hkc_i}$$

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: $c(t) = \frac{c_0}{1+kc_0t}$

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

# 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: $c(t) = \frac{c_0}{1+kc_0t}$

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

| $N$ | $\zeta$ | $\frac{\zeta_{\mathrm{numerical}}-\zeta_{\mathrm{analytical}}}{\zeta_{\mathrm{analytical}}}$ | $r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$ |
|---|---|---|---|
| 20 | 0.6666666667 | $1.665 \times 10^{-16}$ | — |
| 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: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Implicit midpoint method — example

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

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

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

# 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

# Systems of ODEs

A system of ODEs is specified using vector notation:

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

for

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

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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)

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

# 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:

```
# 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`

```
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

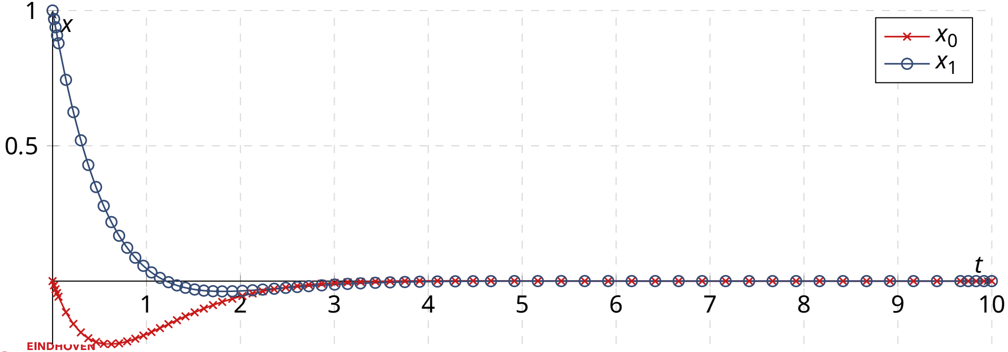
Plot the solution (note: the solution is attribute `sol.y`):

```python
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 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)

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

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

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

# Demonstration with example

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 $\boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$

# 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'`.

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

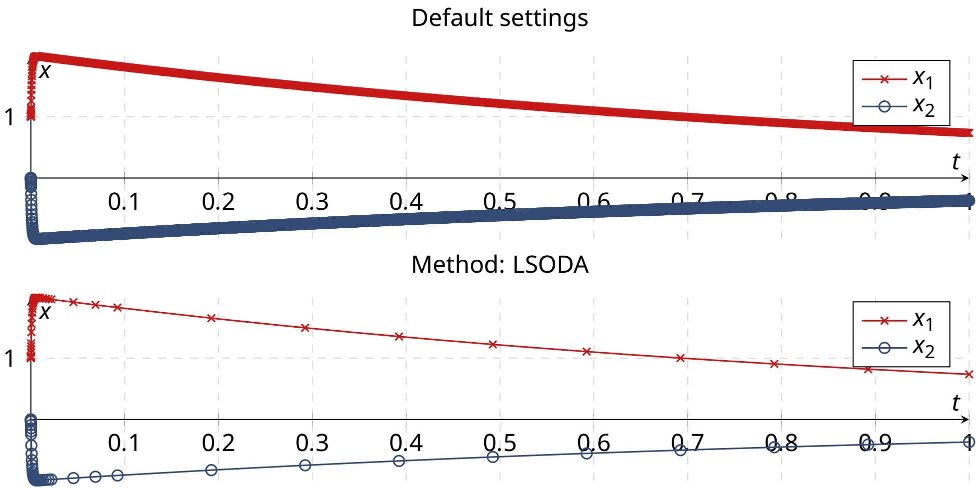
- Create the ode function (see `slide_example_solve_ivp_implicit.py`)

```
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




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

# Implicit methods in Matlab: Generic backward Euler

```
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
```

```
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:

```
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()
```

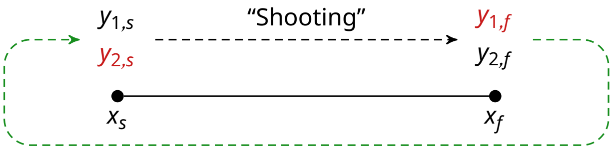
# Today's outline



EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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 $\delta$:

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

Question: compute the concentration profile in the film layer.

# BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness $\delta$:

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

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 $\delta$:

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

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
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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`.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# BVP: example in Python

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

```python
# 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 449 )

# 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()
```

TU/e UNIVERSITY OF TECHNOLOGY

# 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:

```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
```

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 $q = \texttt{bcq}$ such that $c(x = \delta) = 0$. We can use the `scipy.optimize.root_scalar` function to do this. Extend the script from slide 450 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 450.

# 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

- 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

# 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.