Computational practicum: Lecture 5 Ordinary Differential Equations (ODEs) and Initial Value Problems (IVPs)

Zoë J.G. Gromotka, Artur M. Schweidtmann, Ferdinand Grozema, Tanuj Karia

With support from Lukas S. Balhorn and Monica I. Lacatus

Computational Practicum
Dept. Chemical Engineering
Delft University of Technology

TU Delft

Process Intelligence RESEARCH

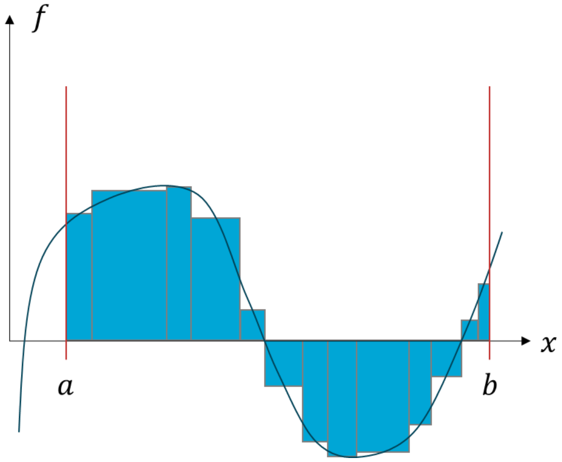
Delft Institute of Applied Mathematics

Wednesday, 1st October 2025

Recap last lecture

Numerical integration

- Quadrature rules
  - Rectangle method
  - Newton-Cotes rules



Numerical differentiation

- Finite difference method
  - First order difference schemes

$$f'(x) = \frac{df}{dx} \to \left.\frac{df}{dx}\right|_{x_i} \quad i = 0, \dots, N-1$$

  - Second order difference schemes

$$f''(x) = \frac{d^2 f}{dx^2} \to \left.\frac{d^2 f}{dx^2}\right|_{x_i} \quad i = 0, \dots, N-1$$

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- categorize ordinary differential equations (ODEs).
- derive the linear, 1st order, autonomous form of an ODE.
- implement different numerical solution approaches to ODEs from scratch, namely,
  - Backward Euler.
  - Forward Euler.
- use Python libraries' built-in functions for numerical solution approaches to ODEs.
- discuss numerical errors and stability of numerical solution approaches to ODEs.

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Ordinary differential equations in reaction engineering

- Take the reaction in a batch reactor with const. density:

First order elementary reaction

$$A \xrightarrow{k} B$$

- Component mass balances gives the corresponding **ordinary differential equations** (ODE), which describe the evolution of the concentrations of A.

concentration

$$\frac{dC_A}{dt} = -kC_A$$

time

rate constant

- The concentration of B follows from the following algebraic relationship assuming constant total concentration equal to 1:

Total concentration, const.

$$C_B = 1 - C_A$$

- Initial condition for concentration C_A is needed, otherwise there exist infinitely many solutions.

Ordinary differential equations in reaction engineering

- Take the reaction in a batch reactor with const. density:

$$A \xrightarrow{k} B$$

First order elementary reaction

- Component mass balances gives the corresponding ODE for C_A and the algebraic equation for C_B, which describe the evolution of the concentrations of A and B.

$$\frac{dC_A}{dt} = -kC_A, \qquad C_B = 1 - C_A$$

- Known initial condition

$$C_A(t=0) = 1$$

→ Called **initial value problem** (short: IVP)

- Analytical solutions

$$C_A(t) = e^{-kt}, \qquad C_B(t) = 1 - e^{-kt}$$

Try at home to verify that this solves the ODEs by inserting the solution in the original ODE system

See PDC course module 2, lecture 3

Ordinary differential equations in reaction engineering

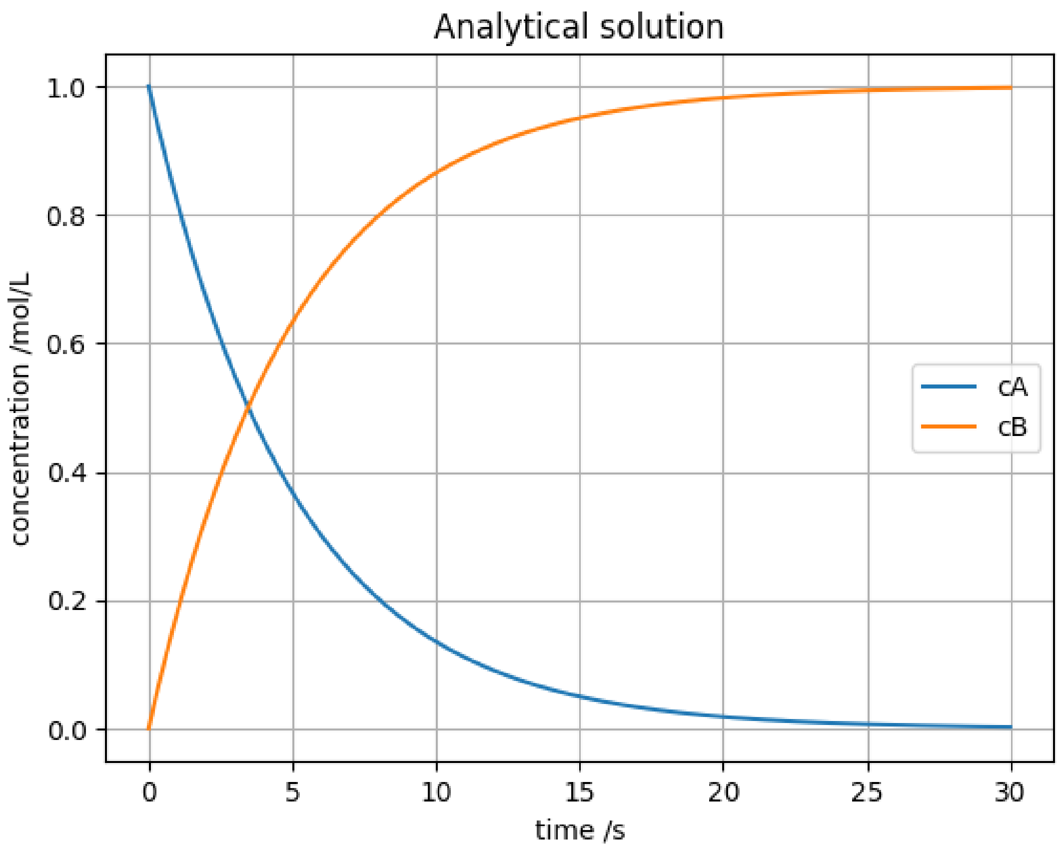
```python
import numpy as np
import matplotlib. pyplot as plt

# Define time vector t
t = np.linspace(0, 30, 100)

# Define kinetic constant k
k = 0.2

# Define concentration function
def cA(t:float, k:float) -> float:
    return np.exp(-k*t)
def cB(t:float, k:float) -> float:
    return 1 - np.exp(-k*t)

# Create the plot
fig, ax = plt.subplots()
ax.plot(t, cA(t,k), label='cA')
ax.plot(x, cB(t,k), label='cB')
ax.set_xlabel('time /s')
ax.set_ylabel('concentration /mol/L')
ax.set_title('Analytical solution')
ax.legend()
ax.grid()
```



Classification of ODEs – Definitions

Definition

An ODE of order n is an equation of the form

$$g\left(t, y, \frac{dy}{dt}, \dots, \frac{d^n y}{dt^n}\right) = 0, \qquad \text{Implicit form}$$

or

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \dots, \frac{d^{n-1} y}{dt^{n-1}}\right). \qquad \text{Explicit form}$$

where

- t : independent variable
- $y(t)$: dependent variable

Classification of ODEs – Definitions

Definition

An ODE is linear, if it is linear in the dependent variable $y(t)$ and its derivatives (i.e. if $y(t)$ and its derivatives appear only to the first power and are never multiplied together). Else it is non-linear.

An ODE is autonomous if it does not include the independent variable t, i.e.

$$g\left(t, y, \frac{dy}{dt}, \dots, \frac{d^n y}{dt^n}\right) = g\left(y, \frac{dy}{dt}, \dots, \frac{d^n y}{dt^n}\right) = 0,$$

Or

$$f\left(t, y, \frac{dy}{dt}, \dots, \frac{d^n y}{dt^n}, \frac{d^{n-1} y}{dt^{n-1}}\right) = f\left(y, \frac{dy}{dt}, \dots, \frac{d^{n-1} y}{dt^{n-1}}\right).$$

Else it is non-autonomous.

Classification of ODEs – Examples

- Examples of different forms of ODE equations:
  - 1st order $\frac{dy}{dt} = f(y)$ Dependent variable Independent variable
  - 2nd order $\frac{d^2y}{dt^2} + y\frac{dy}{dt} = f(y)$
  - 3rd order $\frac{d^3y}{dt} + a\frac{d^2y}{dt^2} + b\frac{dy}{dt} = f(y)$
  - Non-autonomous $\frac{dy}{dt} = f(y, t)$
  - Linear $\frac{dy}{dt} = t$
  - Non-linear $\mathrm{y} \cdot \frac{dy}{dt} = t$

ODE form

See PDC course week 2

- ODEs are easiest to solve if they are **linear**, **1st order**, **autonomous** ODEs!

Original ODE:
(3rd order, linear, autonomous
→ not easy to solve)

$$\frac{d^3y}{dt^3} + a\underbrace{\frac{d^2y}{dt^2}}_{y_2} + b\underbrace{\frac{dy}{dt}}_{y_1} + y = 0$$

New variables:

$$y_1 \equiv \frac{dy}{dt}$$

$$y_2 \equiv \frac{dy_1}{dt} = \frac{d^2y}{dt^2}$$

differentiate

Hint: Insert in original ODE with $\frac{dy_2}{dt} = \frac{d^3y}{dt^3}$ and re-arrange

New system of ODEs:

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = -ay_2 - by_1 - y$$

→ All are linear, 1st order and autonomous.

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Linear ordinary differential equations

- Take the reaction

Series of reversible reactions

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B \underset{k_4}{\overset{k_3}{\rightleftharpoons}} C$$

- Corresponding ODEs giving the evolution of the concentrations of A, B and C:

$$\frac{dC_A(t)}{dt} = -k_1 C_A + k_2 C_B$$
$$\frac{dC_B(t)}{dt} = k_1 C_A - (k_2 + k_3) C_B + k_4 C_C$$
$$\frac{dC_C(t)}{dt} = -k_4 C_C + k_3 C_B$$

Writing linear ordinary differential equations

$$\frac{dC_A(t)}{dt} = -k_1 C_A + k_2 C_B$$
$$\frac{dC_B(t)}{dt} = k_1 C_A + k_4 C_C - (k_2 + k_3) C_B$$
$$\frac{dC_C(t)}{dt} = -k_4 C_C + k_3 C_B$$

- Collect into matrix form:

$$\longrightarrow \begin{pmatrix} dC_A/dt \\ dC_B/dt \\ dC_C/dt \end{pmatrix} = \begin{pmatrix} -k_1 & k_2 & 0 \\ k_1 & -(k_2 + k_3) & k_4 \\ 0 & k_3 & -k_4 \end{pmatrix} \cdot \begin{pmatrix} C_A \\ C_B \\ C_C \end{pmatrix}$$

$$\longrightarrow \frac{d}{dt}\mathbf{c} = \mathbf{K} \cdot \mathbf{c}$$

HINT

We commonly denote vectors and matrices in bold. Vectors are lower case and matrices are upper case.

Analytic solution of linear ODEs

See PDC course module 3, lecture 2

- From process dynamics and control (PDC) course, you know that we can solve linear ODE systems analytically.

$$\frac{d}{dt}\mathbf{c} = \mathbf{K}\cdot\mathbf{c}$$
$$\Rightarrow \mathbf{c} = e^{\mathbf{K}t}\mathbf{c}_0$$

Exponential of a matrix!

As taught in CP lecture 3

- In general, the exponential of a matrix is defined by the power series: $e^{\mathbf{K}t} = \sum_{k=0}^{\infty}\frac{(\mathbf{K}t)^k}{k!}$.
- However, if $\mathbf{K}$ is **diagonalizable**:
  - It can be expressed as $\mathbf{K} = \mathbf{U}\boldsymbol{\Lambda}\,\mathbf{U}^{-1}$, where $\boldsymbol{\Lambda} = \mathbf{diag}(\boldsymbol{\lambda}_1, \dots, \boldsymbol{\lambda}_n)$ is the diagonal matrix of eigenvalues and $\mathbf{U}$ is the matrix whose columns are eigenvectors of $\mathbf{K}$.
  - Then the exponential of a matrix becomes $e^{\mathbf{K}t} = \mathbf{U}e^{\boldsymbol{\Lambda}t}\mathbf{U}^{-1}$.

As taught in CP lecture 3

Solving a system of linear ODEs

$$\begin{array}{c} k_1 \quad k_3 \\ A \rightleftharpoons B \rightleftharpoons C, \\ k_2 \quad k_4 \end{array} \qquad \mathbf{c} = e^{\mathbf{K}t}\mathbf{c}_0$$

Series of reactions

System of ODEs

- Initial condition

$$C_A(0) = 1, \qquad C_B(0) = 0, \qquad C_C(0) = 0$$

- Rates

$$k_1 = 1\ \mathrm{min}^{-1}, \qquad k_2 = 0\ \mathrm{min}^{-1}, \qquad k_3 = 2\ \mathrm{min}^{-1}, \qquad k_4 = 3\ \mathrm{min}^{-1}$$

$$\begin{pmatrix} C_A(t_p) \\ C_B(t_p) \\ C_C(t_p) \end{pmatrix} = exp\left[\begin{pmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{pmatrix} t_p\right] \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Solving a system of linear ODEs

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B \underset{k_4}{\overset{k_3}{\rightleftharpoons}} C, \qquad \mathbf{c} = e^{\mathbf{K}t}\mathbf{c}_0$$

- Time discretization

$$T = [0, \Delta t, 2\Delta t, 3\Delta t, \dots, (N-1)\Delta t]$$

equidistant grid

- Solution

$$\begin{aligned}\mathbf{c}(t_{N-1} = (N-1)\Delta t) &= e^{\mathbf{K}(\mathrm{N}-1)\Delta t}\mathbf{c}_0 = \left(e^{\mathbf{K}\Delta t}\right)^{(N-1)}\mathbf{c}_0 \\ &= \left(e^{\mathbf{K}\Delta t}\right)\left(e^{\mathbf{K}\Delta t}\right)\dots\underbrace{\left(e^{\mathbf{K}\Delta t}\right)\mathbf{c}_0}_{\mathbf{c}(t_1 = \Delta t)}\end{aligned}$$

HINT

This solution is only possible if we choose equidistant grid points.

Solving a system of linear ODEs

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B \underset{k_4}{\overset{k_3}{\rightleftharpoons}} C$$

- Initial condition

$$C_A(0) = 1, \qquad C_B(0) = 0, \qquad C_C(0) = 0$$

- Rates

$$k_1 = 1 \text{ min}^{-1}, \qquad k_2 = 0 \text{ min}^{-1}, \qquad k_3 = 2 \text{ min}^{-1}, \qquad k_4 = 3 \text{ min}^{-1}$$

Time

$$\begin{aligned} &\mathbf{c}_0 \\ &\mathbf{c}_1 = e^{\mathbf{K}\Delta t}\mathbf{c}_0 \\ &\mathbf{c}_2 = e^{\mathbf{K}\Delta t}\mathbf{c}_1 \\ &\quad \dots \\ &\quad \dots \\ &\mathbf{c}_{N-1} = e^{\mathbf{K}\Delta t}\mathbf{c}_{N-2} \end{aligned}$$

We only need to compute one exponential of the matrix $e^{\mathbf{K}\Delta t}$.

Live coding: Analytical solution

- Open Colab: Analytical solution

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

What if we do not know the analytical solution?

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Non-Linear ODEs: Initial value problem

- General form

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

- Initial condition

$$y(t_0) = y_0$$

HINT

For every n-th order ODE, we need n initial conditions.

- Example

Second order elementary reaction

$$2A \xrightarrow{k} B$$

Non-linear term

$$\frac{dC_A}{dt} = -kC_A^2$$
$$\frac{dC_B}{dt} = kC_A^2$$

- Initial condition

$$C_A(0) = 1$$
$$C_B(0) = 0$$

Numerical solution through discretization

- Discretization

$$t = [t_0, t_1, \dots, t_i, \dots, t_{N-1}] \Rightarrow y = [y_0, y_1, \dots, y_i, \dots, y_{N-1}]$$

- How can we get from an initial value at $y_i = y(t_i)$ to the next value at $y_{i+1} = y(t_{i+1})$?

Non-Linear ODEs: Initial value problem

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

- Find the solution by integrating from an initial value y_i to the subsequent value y_{i+1}.

$$\int_{y_i}^{y_{i+1}} dy = \int_{t_i}^{t_{i+1}} f(t, y) dt$$

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f\big(t, y(t)\big) dt$$

$$y(t_{i+1}) = y(t_i) + \underbrace{\int_{t_i}^{t_{i+1}} f\big(t, y(t)\big) dt}$$

Numerical approximation
for definite integrals!

As taught
in lecture 4

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Non-Linear ODEs: Initial value problem
The forward Euler method

- Rewriting the ODE

$$\frac{dy}{dt} = f(t,y) \qquad \Big|\ \text{Integrate over } \int_{t_i}^{t_{i+1}} dt$$

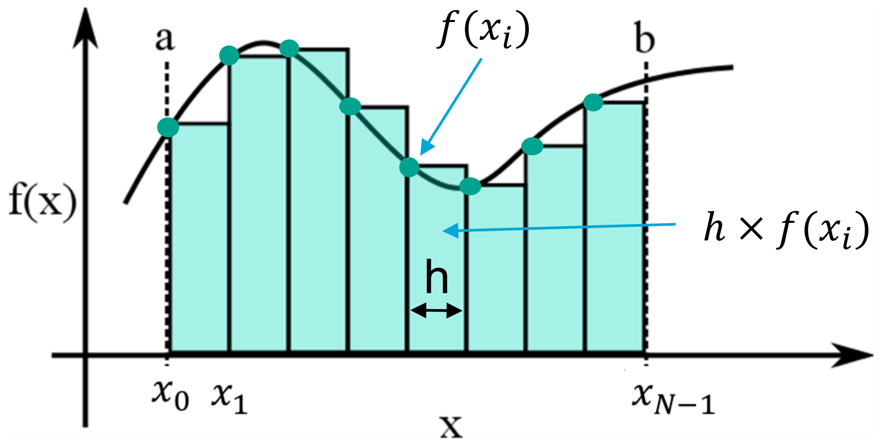
$$\int_{t_i}^{t_{i+1}} \frac{dy}{dt} dt = \int_{t_i}^{t_{i+1}} f(t,y)dt$$

$$y_{i+1} - y_i = \int_{t_i}^{t_{i+1}} f(t,y)dt \qquad \Big|\ +y_i$$

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t,y)dt \qquad \Big|\ \text{Substitute}$$

$$y_{i+1} \approx y_i + hf(t_i, y_i)$$

Rectangle approach for integration
x_i^* in left corner (c.f. lecture 4)



$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_i, y_i)$$

Non-Linear ODEs: Initial value problem
The Forward Euler method

- Using Euler for ODEs

Definition

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

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt$$

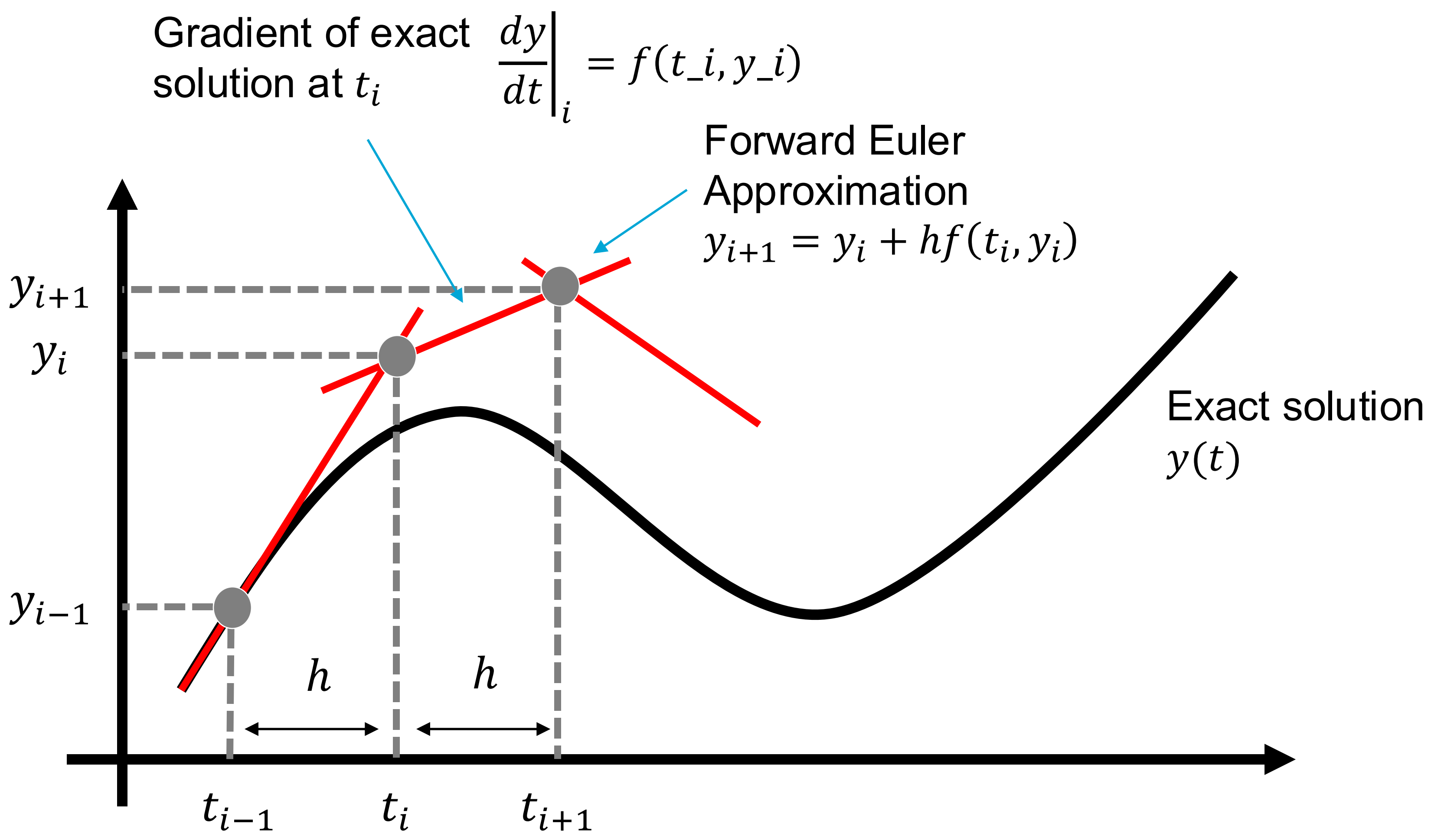
$$y_{i+1} = y_i + hf(t_i, y_i)$$

HINT

Note that step size h can be taken to be negative.

- Forward Euler is also called the **explicit** Euler method, because it gives an explicit expression for y_{i+1} ($y_{i+1} = g(t_i, y_i)$).

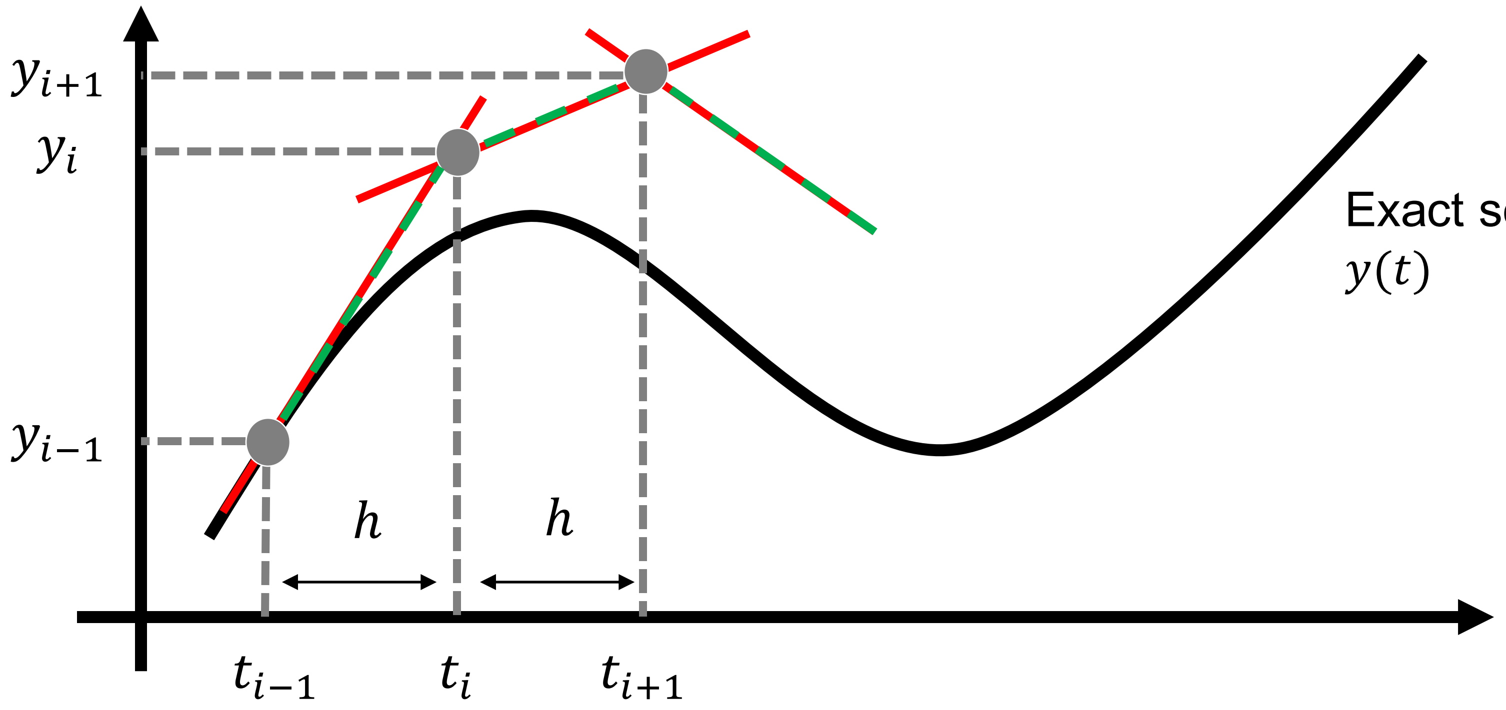
Non-Linear ODEs: Initial value problem
The forward Euler method



Non-Linear ODEs: Initial value problem
Grid points vs. evaluation points

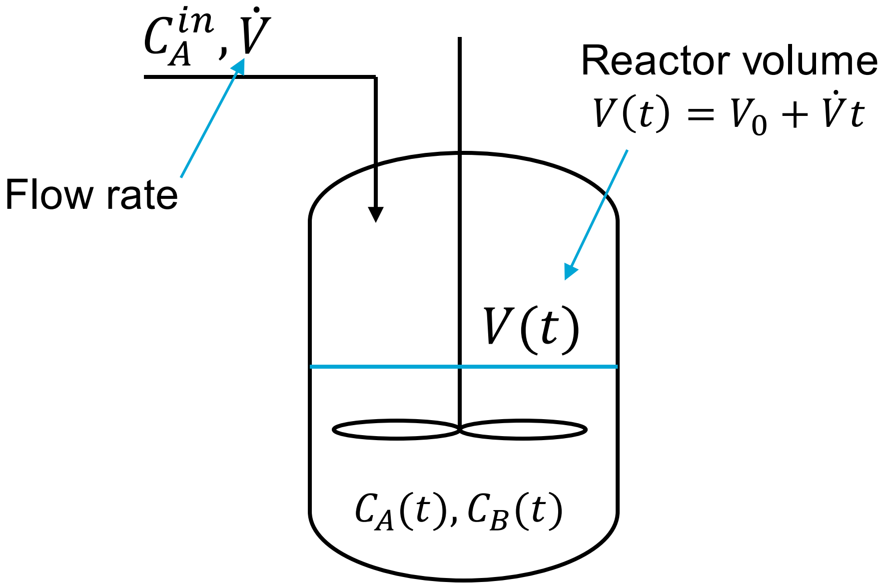
Grid points:
- Used to obtain numerical solution
- Chosen a priori

Evaluation points:
- Evaluate numerical solution
- Chosen a posteriori



Case study for lecture coding: Semi batch example

- 1st order reaction scheme $A \xrightarrow{k} B$.
- Semi-batch reactor model, assuming constant density.



$$\begin{cases} \dfrac{dC_A}{dt} = \dfrac{\dot{V}}{V_0 + \dot{V}t}\left(C_A^{in} - C_A\right) - kC_A \\ \dfrac{dC_B}{dt} = kC_A - \dfrac{\dot{V}}{V_0 + \dot{V}t} C_B \end{cases}$$

$$with:\ C_A(0) = 1, C_B(0) = 0$$

Non-autonomous term which contains the independent variable t explicitly.

Try at home:
1. Derive the ODE system equations by performing component balances.
2. Transform the semi batch ODE system to the 1st order, linear, autonomous form.

Live coding: Forward Euler

- Open Colab: Forward Euler

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Non-Linear ODEs: Initial value problem
The backward Euler method

- Using Euler for ODEs

Definition

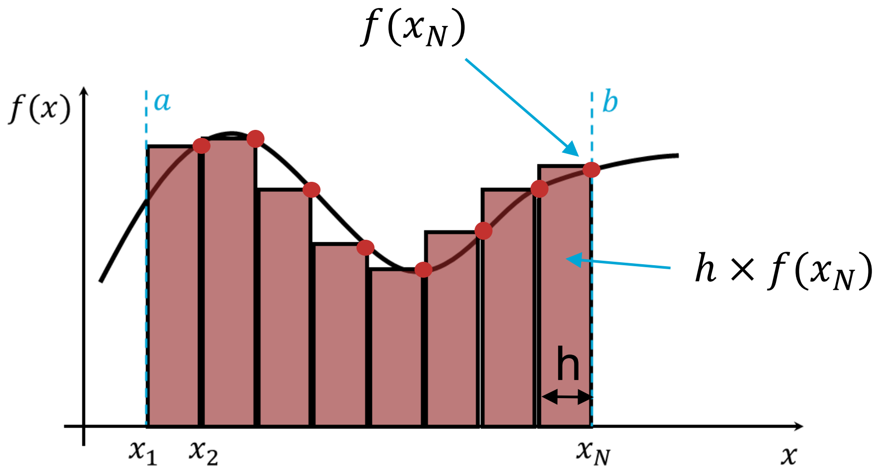
$$\frac{dy}{dt} = f(t,y)$$

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t,y)dt$$

$$y_{i+1} \approx y_i + hf(t_{i+1}, y_{i+1})$$

Substitute

Rectangle approach for integration
x_i^* in right corner (c.f. lecture 4)



$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_{i+1}, y_{i+1})$$

- Backward Euler is also called the **implicit** Euler method, because it gives an implicit expression for y_{i+1} ($y_{i+1} = g(t_i, t_{i+1}, y_i, y_{i+1})$).

Non-Linear ODEs: Initial value problem
The backward Euler method

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1})$$

- The term y_{i+1} appears on both sides of the equation (*implicit*).
  - We can formulate this as a root finding problem.

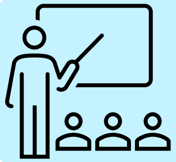
$$y_{i+1} = g(y_{i+1})$$

$$\text{e.g.}\ \frac{dy}{dt} = y^2 \rightarrow y_{i+1} = y_i + hy_{i+1}^2$$

  - We can exactly reformulate the problem to be explicit (not possible for all equations).

$$y_{i+1} = f^*(t_i, y_i, h)$$

$$\text{e.g.}\ \frac{dy}{dt} = y \rightarrow y_{i+1} = y_i + hy_{i+1} \rightarrow y_{i+1} = \frac{y_i}{1-h}$$



As taught in lecture 2

Root-finding problem for the backward Euler method

$$y_{i+1} = g(y_{i+1})$$

- Root finding methods: Fixed-point iteration, Newton-Raphson method.
- At every integration step, we need to solve the root finding problem.

Root finding step

$$y_{i+1}^{[0]} = y_i, \qquad y_{i+1}^{[k+1]} = y_i + hf\left(t_{i+1}, y_{i+1}^{[k]}\right)$$

Integration step

- → expansive numerical method for ODEs
- … but there are advantages, stay tuned!

As taught in lecture 2

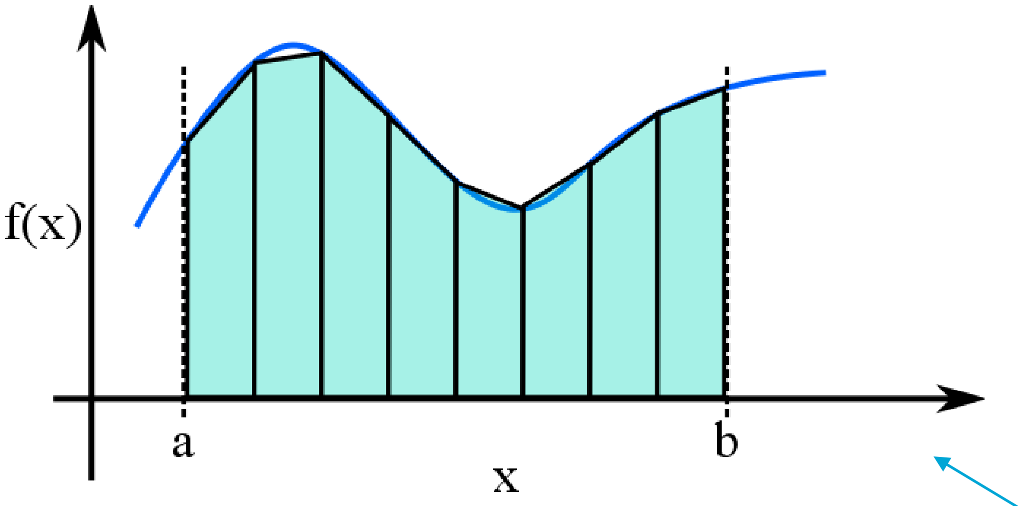
Live coding: Backward Euler

- Open Colab: Backward Euler

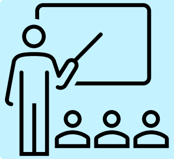
- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Non-Linear ODEs: Initial value problem
The Heun's method

- In addition, other numerical integration methods, such as the trapezoid method, can be used to solve the IVP.
  → Heun's method (or modified Euler method, improved Euler method).



Trapezoid method



Heun's method in Assignment 5

Non-Linear ODEs: Initial value problem Runge-Kutta method

See PDC course lecture

- There exist more advanced methods like Runge-Kutta (RK4) which will be covered in the Process Dynamics and Control (PDC) lecture.
- Runge-Kutta is the most widely used method in practice.
- Explicit method that uses multiple intermediate points to estimate the slope, e.g. RK4:

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

- Greater accuracy per step than forward or backward Euler.

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

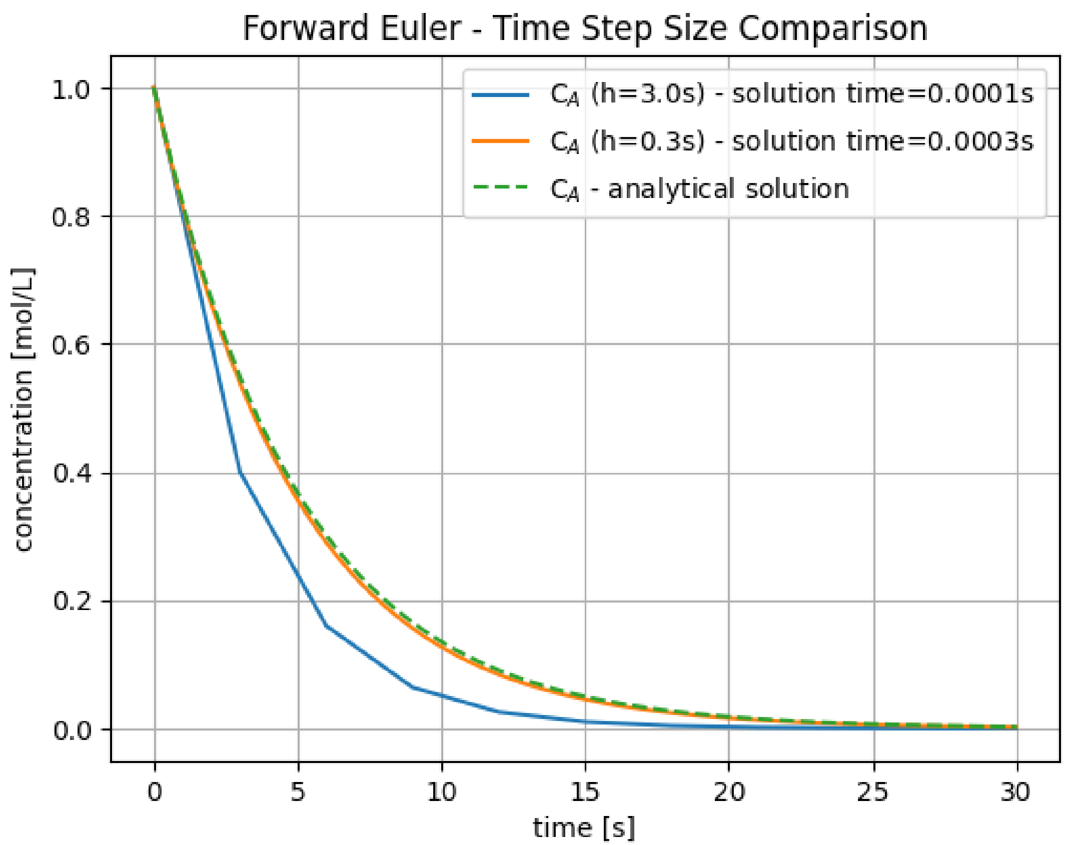
Influence of the time step size

$$A \xrightarrow{k} B$$
$$C_A(0) = 1$$

- Solution

$$C_A(t) = e^{-kt}$$
$$C_B(t) = 1 - e^{-kt}$$

There is a trade-off between the solution time (size of the problem, N) and the accuracy of the solution (error).



Live coding: Numerical error

- Open Colab: Numerical error

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Errors in numerical solution of ODEs

- Rounding error
  - Due to finite precision of floating-point arithmetic (cf. lecture 1)
- Truncation error
  - Due to the approximate nature of the method
  - Global truncation error $e_i = y_i - y(t_i)$
  - Local truncation error $\ell_i = y_i - u_{i-1}(t_i)$

Numerical solution y at point i

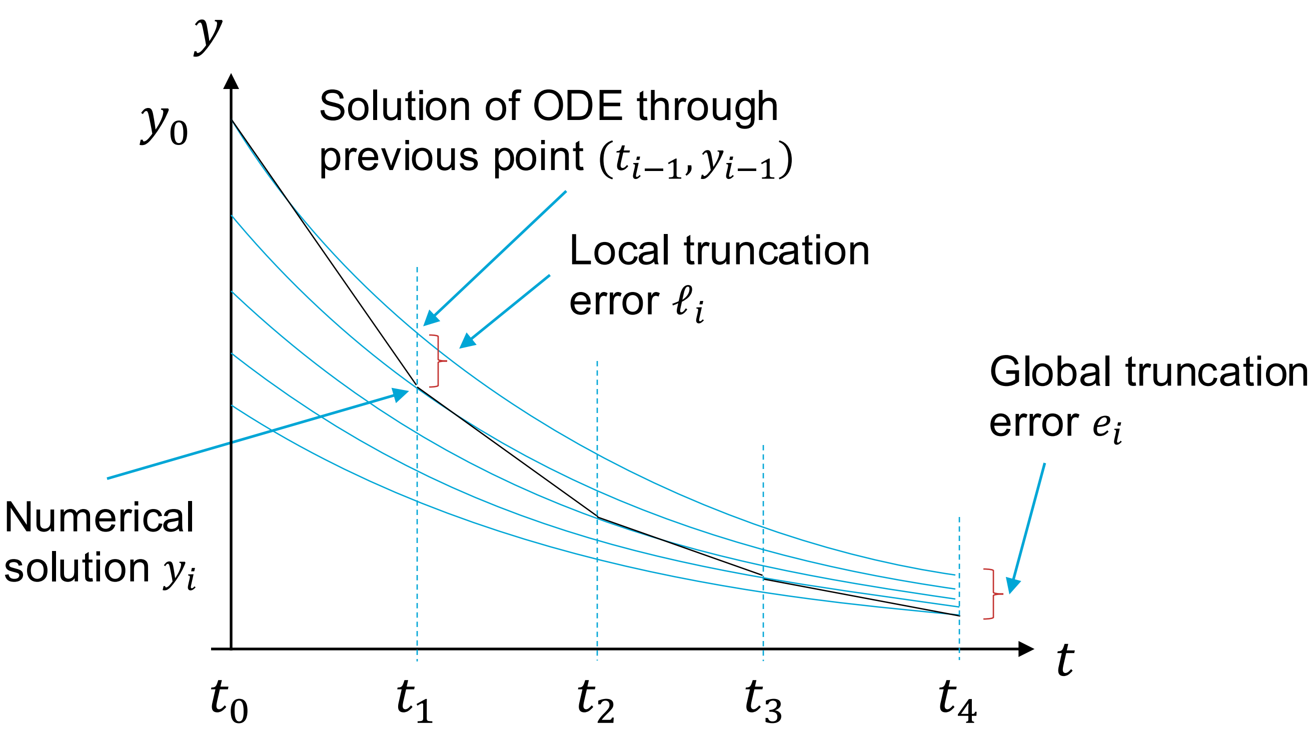
True solution y at point i

True solution of ODE through previous point (t_{i-1}, y_{i-1})

Heath M.T - Scientific computing: an introductory survey, SIAM (2018)

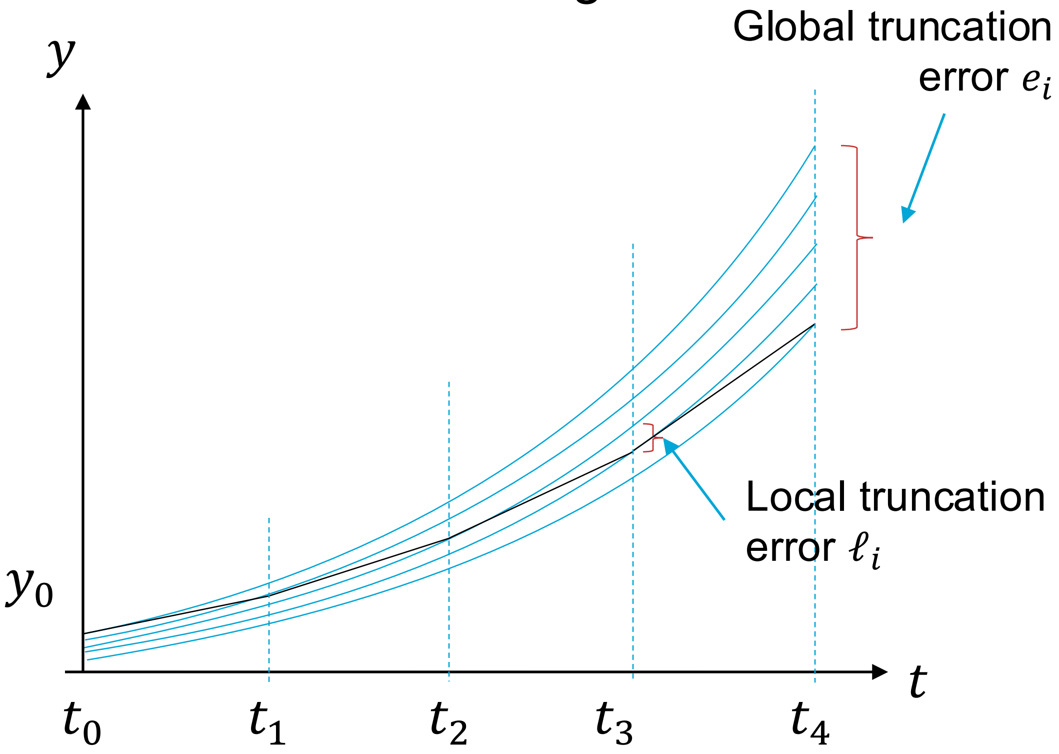
Errors in numerical solution of ODEs

- Stable solution,
  errors in numerical solution may diminish



$$\frac{dy}{dt} = -y$$

- Unstable solution,
  errors in numerical solution grow



$$\frac{dy}{dt} = y$$

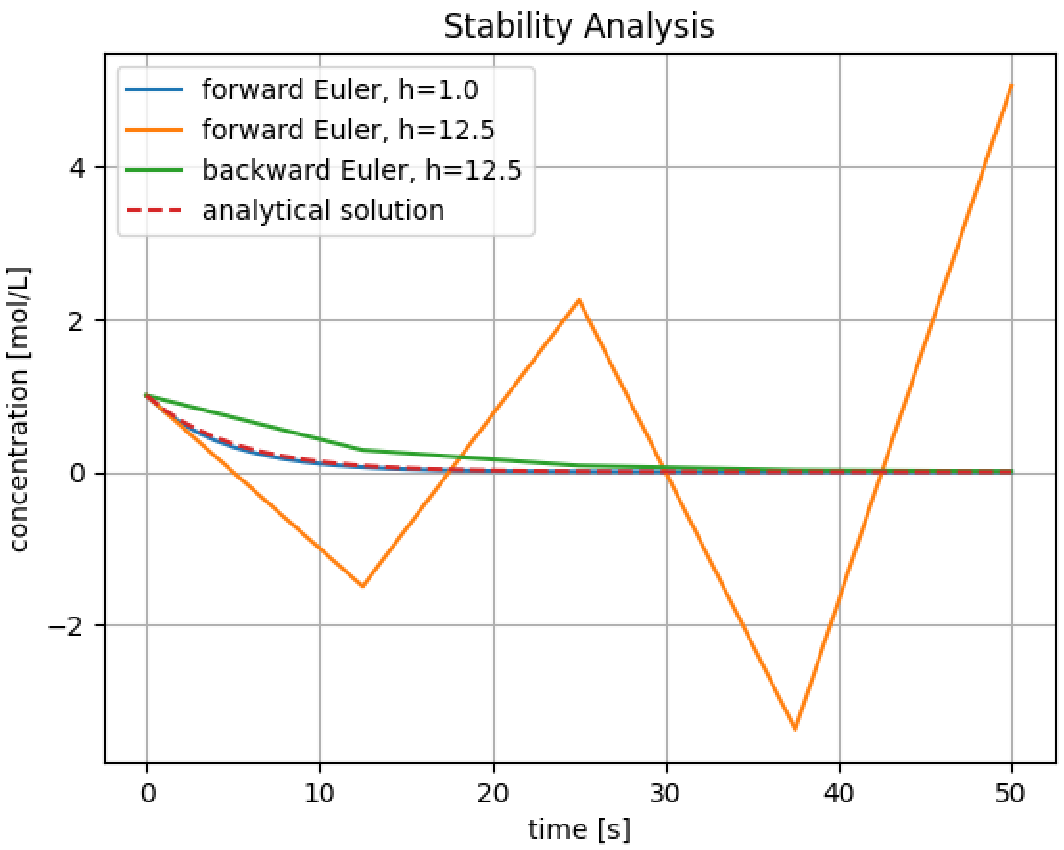
Heath M.T - Scientific computing: an introductory survey, SIAM (2018)

Stability analysis

- Backward (implicit) Euler *unconditionally* stable
- Forward (explicit) Euler and Runge-Kutta *conditionally* stable
  - Depending on step size h and stiffness of the problem

Definition

Stiff ODEs often involve competing physical phenomena with **widely varying time and/or spatial scales**. There is no precise definition in literature for stiffness. In general, an ODE is stiff if the eigenvalues of the Jacobian differ greatly in magnitude.



Amos Gilat, Vish Subramaniam - Numerical Methods for Engineers and Scientists: An Introduction with Applications using MATLAB, Wiley (2013)

Live coding: Stability

- Open Colab: Stability

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Agenda

- **Ordinary differential equations (ODEs)**
  - Classification of ODEs
  - System of linear ODEs
- **Numerical solution methods for Initial value problems (IVPs)**
  - Forward Euler
  - Backward Euler
  - Errors in numerical solution of ODEs and stability
  - ODE solver in scipy

Solving IVPs using scipy's *solve_ivp*

- scipy provides a built-in IVP solver called *solve_ivp*
- Function arguments:
  - *fun*: Function containing the ODE system.
  - *t_span*: Border of integration interval.
  - *y0*: Initial values.
  - *method*: Integration method. There are explicit (e.g., "RK45") and implicit methods (e.g., "Radau") available. Choose wisely!
  - *t_eval:* Grid points for which solution is returned. Solver uses dynamic grid points to calculate solution.

Read the complete description of *solve_ivp* (link) at home.

scipy.integrate.solve_ivp

`scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options)` [source]

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [11]). To solve a problem in the complex domain, pass y0 with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

Parameters:

fun : ***callable***

Right-hand side of the system: the time derivative of the state `y` at time `t`. The calling signature is `fun(t, y)`, where `t` is a scalar and `y` is an ndarray with `len(y) = len(y0)`. Additional arguments need to be passed if `args` is used (see documentation of `args` argument). `fun` must return an array of the same shape as `y`. See *vectorized* for more information.

t_span : ***2-member sequence***

Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf. Both t0 and tf must be floats or values interpretable by the float conversion function.

y0 : ***array_like, shape (n,)***

Initial state. For problems in the complex domain, pass *y0* with a complex data type (even if the initial value is purely real).

method : ***string or*** `OdeSolver`***, optional***

Live coding: *solve_ivp*

- Open Colab: [solve_ivp]

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- categorize ordinary differential equations (ODEs).
- derive the derive the linear, 1^{st} order, autonomous form of an ODE.
- implement different numerical solution approaches to ODEs from scratch, namely,
  - Backward Euler.
  - Forward Euler.
- use Python libraries' built-in functions for numerical solution approaches to ODEs.
- discuss numerical errors and stability of numerical solution approaches to ODEs.

Thank you very much for your attention!