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 (6BER03), 2024-2025

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

Today's outline

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:

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

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:

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

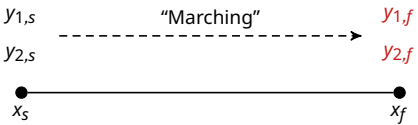
Now define and solve using z as a new variable:

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

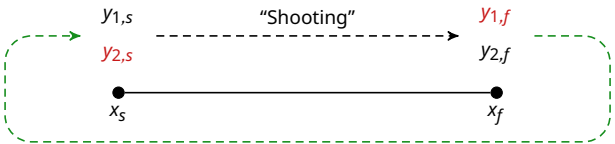
Importance of boundary conditions

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

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

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:

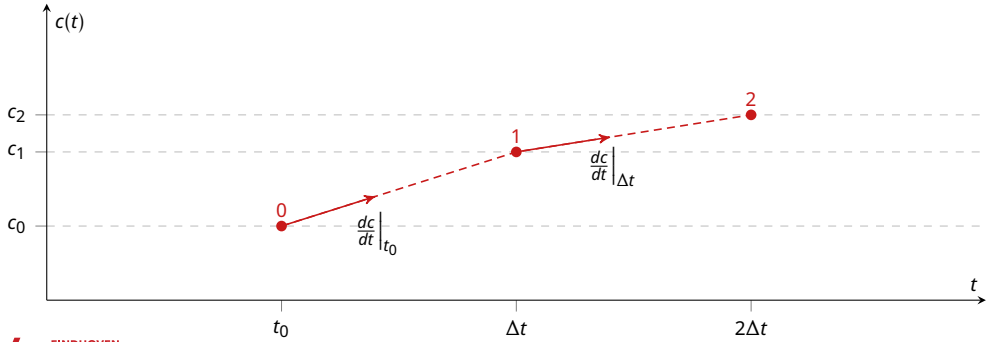
$$c(t_0 + \Delta t) \approx c(t_0) + \left.\frac{dc}{dt}\right|_{t_0} \Delta t + \frac{1}{2} \left.\frac{d^2c}{dt^2}\right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

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:

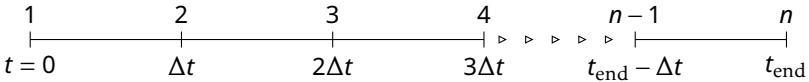
$$\frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t} = f(c_0, t_0) \Rightarrow c(t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$$

Euler's method: graphical example

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



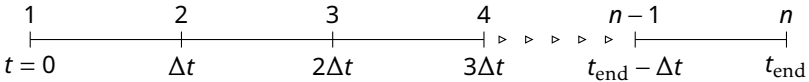
Euler's method - solution method



c_0

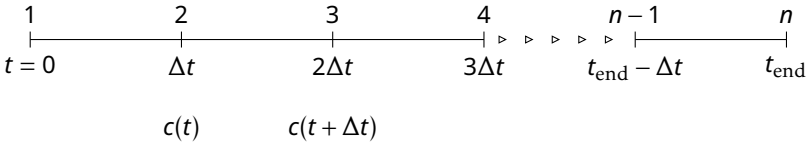
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



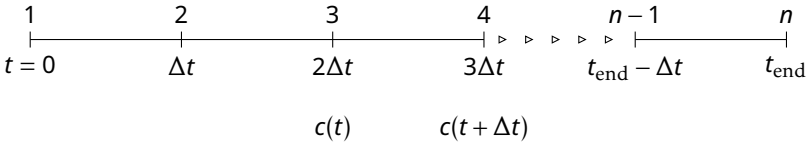
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method



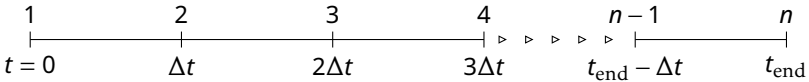
Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

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
   $x_{i+1} = x_i + h$; $y_{i+1} = y_i + h f(x_i, y_i)$

Euler's method - example

First order reaction in a batch reactor:

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

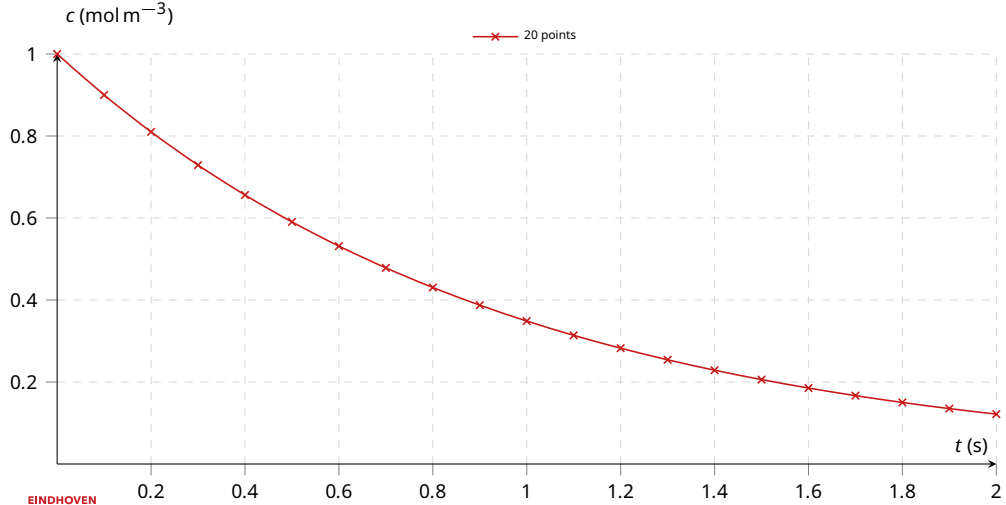
Euler's method - example

First order reaction in a batch reactor:

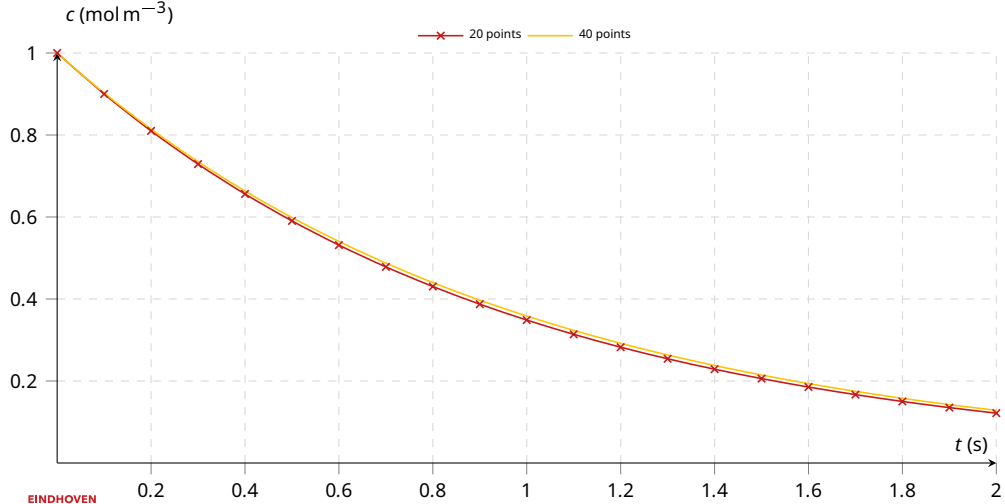
$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1\,\text{mol}\,\text{m}^{-3}, \quad k = 1\,\text{s}^{-1}, \quad t_{\text{end}} = 2\,\text{s}$$

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

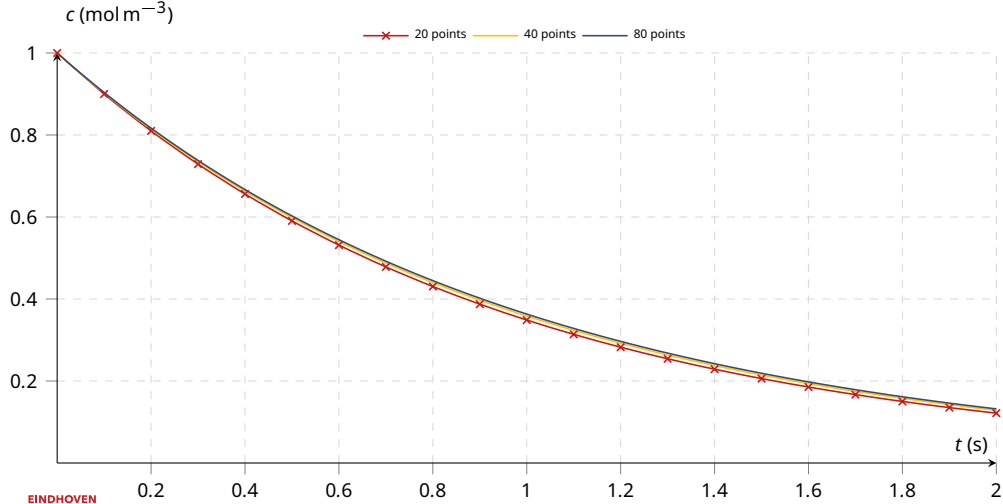
Euler's method - example



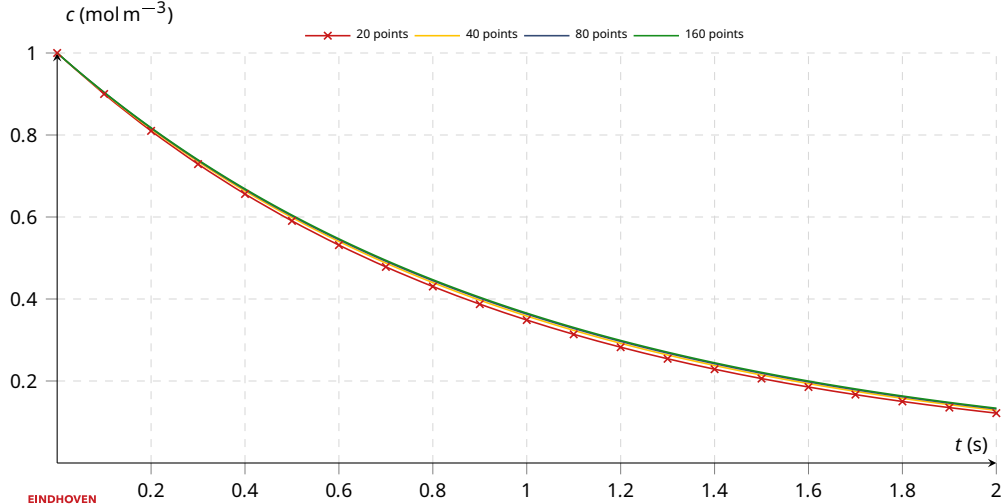
Euler's method - example



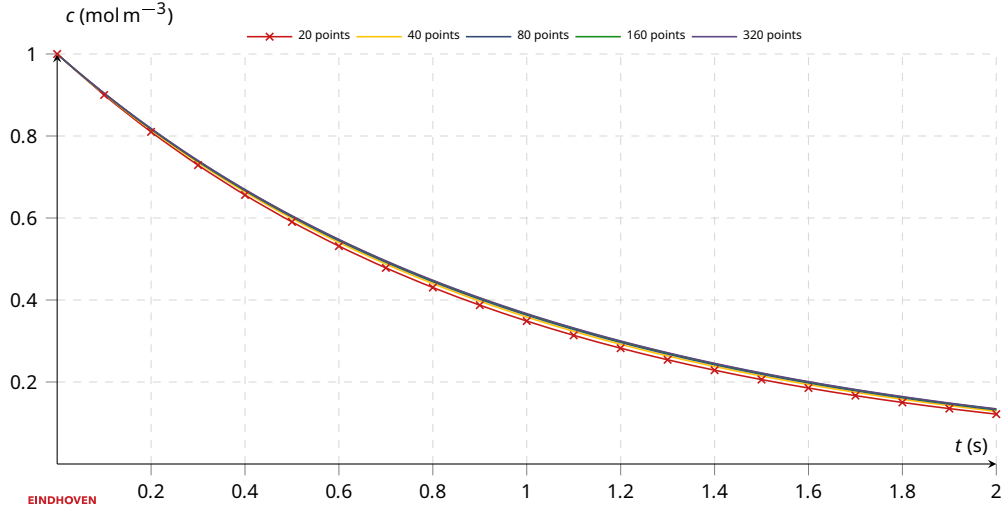
Euler's method - example



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`:

```
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`:

```
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`:

```
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:

```
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:

```
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`:

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

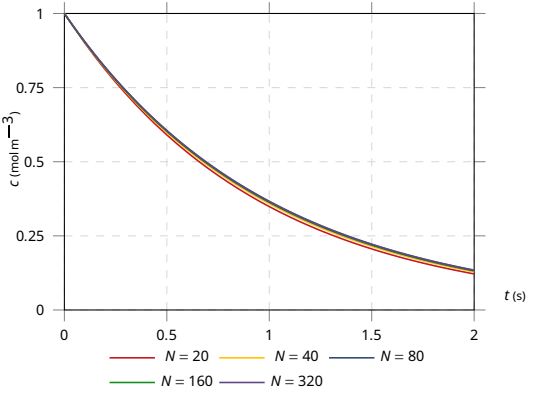
Alternatively, we can pass a *lambda function* in-place:

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

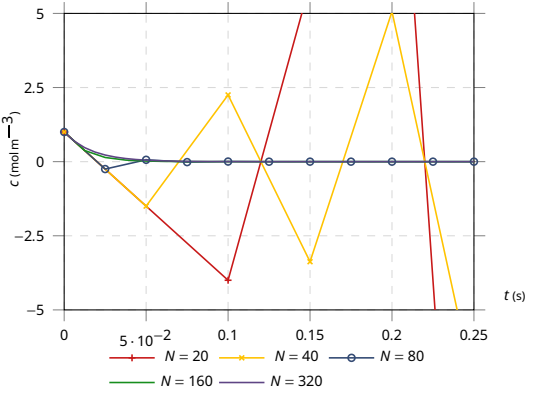
Problems with Euler's method

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



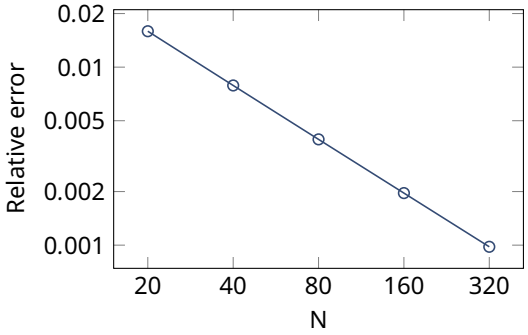
Reaction rate: $k = 50\ \mathrm{s}^{-1}$

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:

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

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

Today's outline

Errors and convergence rate

L_2 norm (Euclidean norm)

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

L_∞ norm (maximum norm)

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

Absolute difference

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

Relative difference

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

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: $\epsilon = \frac{||\bar{\varepsilon}||_1}{N}$
   - Based on L_2-norm: $\epsilon = \frac{||\bar{\varepsilon}||_2}{\sqrt{N}}$
   - 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 $\bar{\epsilon}$. Take the norm to compute a single error value ϵ following:
   - Based on L_1-norm: $\epsilon = \frac{\|\bar{\epsilon}\|_1}{N}$
   - Based on L_2-norm: $\epsilon = \frac{\|\bar{\epsilon}\|_2}{\sqrt{N}}$
   - Based on L_∞-norm: $\epsilon = \|\bar{\epsilon}\|_\infty$
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:

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

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

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:

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

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

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

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

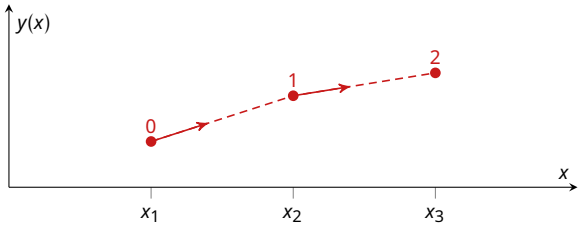
- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Python

Runge-Kutta methods

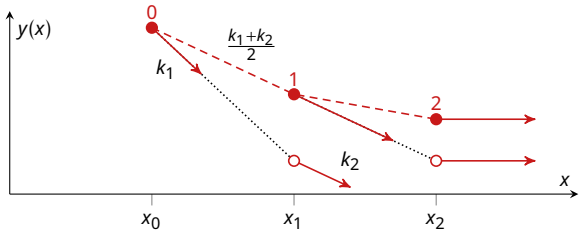
Propagate a solution by combining the information of several Euler-style steps (each involving one function evaluation) to match a Taylor series expansion up to some higher order.

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:

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

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

Concluding:

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

Rewriting:

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

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

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

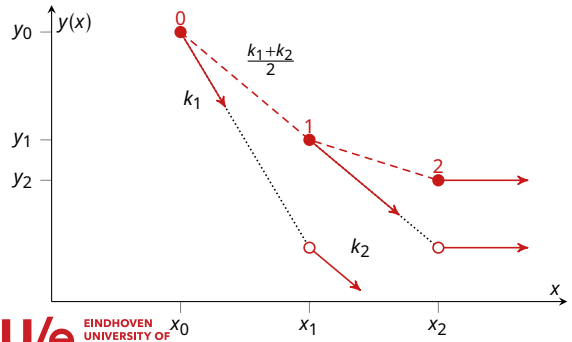
$$y_{i+1} = y_i + hf_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

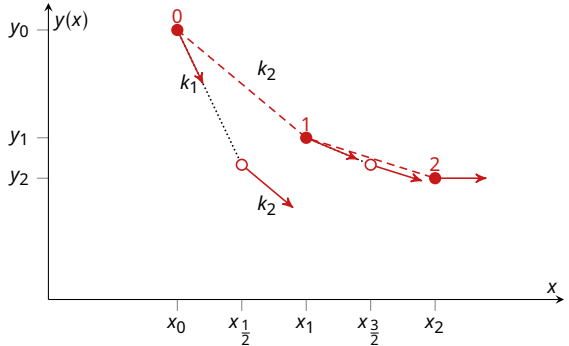
$\Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

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)
$k_1 = f_i$	$k_1 = f_i$
$k_2 = f(x_i + h, y_i + hk_1)$	$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$
$y_{i+1} = y_i + \frac{1}{2}h(k_1 + k_2)$	$y_{i+1} = y_i + hk_2$

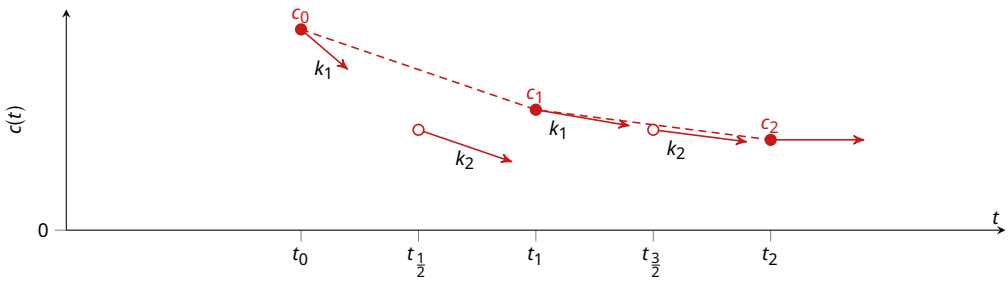




Second order Runge-Kutta method — Example

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

Time [s]	C [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

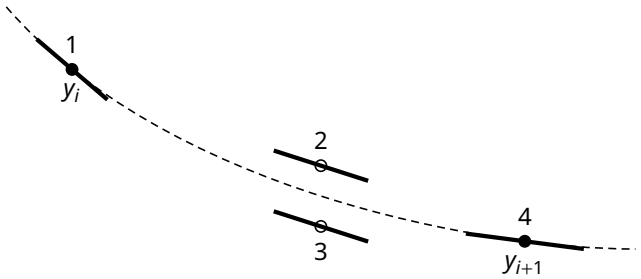
RK2 method — order of convergence

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)



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

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

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

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

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

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

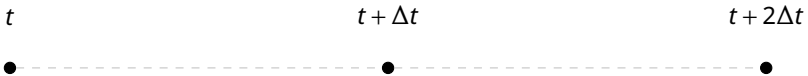
Adaptive step size control

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.

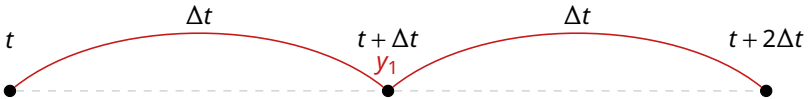
Globally two different approaches can be used:

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

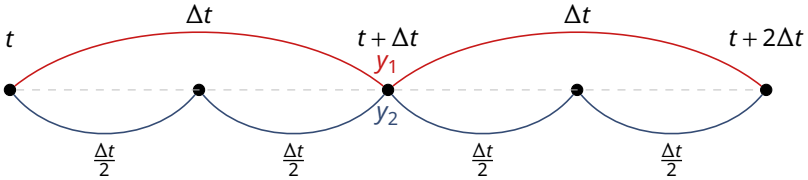


Adaptive step size control: step doubling



- 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:

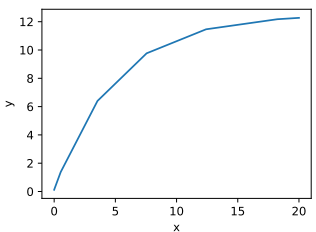
```
1 dydx = lambda x,y: (-0.2*y + 2.5)
```

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

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

- Draw the results by calling the relevant Matplotlib commands:

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



Solving ODEs in Python: example 2

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

Create an ODE function

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

Create a solution script

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

Solving ODEs in Python: example 2

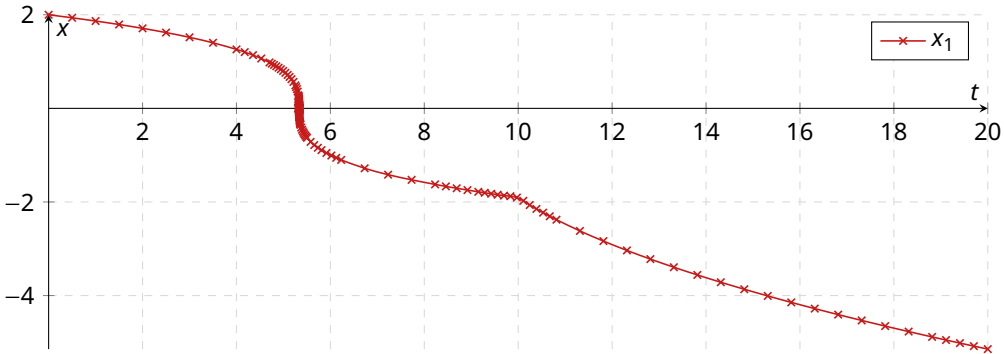
Plot the solution:

```
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:

```
1 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!