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
- 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



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

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

accurately and efficiently?



What is an ODE?

Algebraic equation:

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

• First order ODF:

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

Second order ODE:

$$f\left(\frac{d^2y}{dx^2}(x), \frac{dy}{dx}(x), y(x), x\right) = 0$$
 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!)

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:

$$\frac{dy}{dx} = z(x)$$

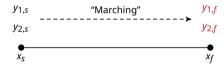
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$



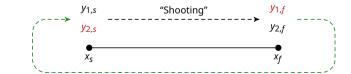
Importance of boundary conditions

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

Boundary value problems

Shooting method



Introduction Euler's method Rates of convergence Runge-Kutta methods. Step size control. Solving ODEs in Python. Introduction Systems of ODEs Boundary value problems

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
- 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



Euler's method

Consider the following single initial value problem:

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

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

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

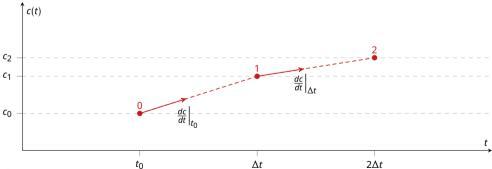
Neglect terms with higher order than two: $\frac{dc}{dt}\Big|_{t_0} = \frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t}$ Substitution:

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



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

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 $x < x_{end}$ do $x_{i+1} = x_i + h$; $y_{i+1} = y_i + hf(x_i, y_i)$



Euler's method - example

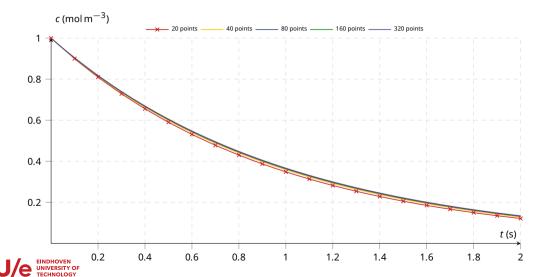
First order reaction in a batch reactor:

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

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	$= 0.81 + 0.1 \cdot (-1 \cdot 0.81) = 0.729$
•••	•••
$t_{i+1} = t_i + \Delta t$	$c_{i+1} = c_i + \Delta t \cdot (-kc_i)$
•••	•••
$t_{20} = 2.0$	$c_{20} = c_{19} + \Delta t \cdot (-kc_{19}) = 0.121577$



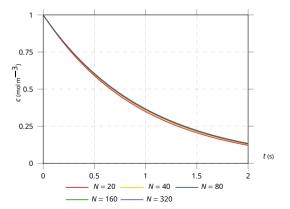
Euler's method - example



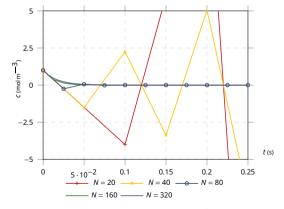
Problems with Euler's method

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

- **1** Accuracy ⇒ need information on numerical error!
- Stability ⇒ need information on stability limits!



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



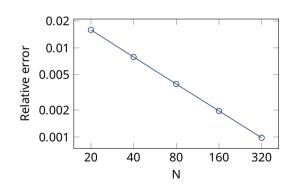
Reaction rate: $k = 50 \text{ s}^{-1}$

Accuracy

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

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

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:

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

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



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

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

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



hod Rates of convergence Runge-Kutta methods Step size control Solving ODEs in Python Introduction Systems of ODEs Boundary value problems

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
- 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



Errors and convergence rate

L₂ norm (Euclidean norm)

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

L_{∞} norm (maximum norm)

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

Absolute difference

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

Relative difference

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



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 $\mathbf{0}$ or $\mathbf{2}$ for a particular number of grid points N:

- **1** Compute the relative or absolute error vector $\overline{\varepsilon}$. Take the norm to compute a single error value ε following:
 - Based on L_1 -norm: $\epsilon = \frac{||\overline{\epsilon}||_1}{N}$
 - Based on L_2 -norm: $\epsilon = \frac{\|\overline{\varepsilon}\|_2}{\sqrt{N}}$
 - Based on L_{∞} -norm: $\epsilon = ||\overline{\epsilon}||_{\infty}$
- 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:

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

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

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	$\frac{\zeta_{ m numerical} - \zeta_{ m analytical}}{\zeta_{ m analytical}}$	$r = \frac{\log\left(\frac{\epsilon_{i}}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_{i}}\right)}$
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

⇒ 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

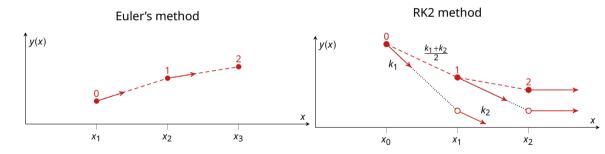
- Introduction
- Euler's method
 - Forward Euler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python
- Introduction
 - Backward Euler
 - Implicit midpoint method
- Systems of ODEs
 - Solution methods for systems of ODEs
 - Solving systems of ODEs in Python
 - Stiff systems of ODEs



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 $h = \Delta x_i$ i.e. slope $= k_1 = f(x_i, y_i)$.





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_{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
```



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

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

$$\frac{dy}{dx}\bigg|_{i} = f(x_{i}, y_{i}) \equiv f_{i}$$

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

Substitution gives:

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

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



Note multivariate Taylor expansion:

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

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

Concluding:

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

Rewriting:

$$k_{1} = f(x_{i}, y_{i})$$

$$k_{2} = f(x_{i} + h, y_{i} + hk_{1})$$

$$\Rightarrow y_{i+1} = y_{i} + \frac{h}{2}(k_{1} + k_{2})$$

Generalization:
$$y_{i+1} = y_i + h(b_1k_1 + b_2k_2) + \mathcal{O}(h^3)$$
 with $k_1 = f_i$, $k_2 = f(x_i + c_2h, y_1 + a_{2,1}hk_1)$ (Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_{i} + c_{2}h, y_{i} + a_{2,1}hk_{1}) = f_{i} + c_{2}h \frac{\partial f}{\partial x}\Big|_{i} + a_{2,1}hk_{1} \frac{\partial f}{\partial y}\Big|_{i} + \mathcal{O}(h^{2})$$

$$y_{i+1} = y_{i} + h(b_{1}k_{1} + b_{2}k_{2}) + \mathcal{O}(h^{3})$$

$$= y_{i} + h\Big[b_{1}f_{i} + b_{2}f(x_{i} + c_{2}h, y_{1} + a_{2,1}hk_{1})\Big] + \mathcal{O}(h^{3})$$

$$= y_{i} + h\Big[b_{1}f_{i} + b_{2}\Big\{f_{i} + c_{2}h \frac{\partial f}{\partial x}\Big|_{i} + a_{2,1}hk_{1} \frac{\partial f}{\partial y}\Big|_{i} + \mathcal{O}(h^{2})\Big\}\Big] + \mathcal{O}(h^{3})$$

$$= y_{i} + h(b_{1} + b_{2})f_{i} + h^{2}b_{2}\Big[c_{2}\frac{\partial f}{\partial x}\Big|_{i} + a_{2,1}f_{i}\frac{\partial f}{\partial y}\Big|_{i}\Big] + \mathcal{O}(h^{3})$$

Comparison with Taylor:

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

Using $b_1 + b_2 = 1$, $c_2b_2 = \frac{1}{2}$, $a_{2,1}b_2 = \frac{1}{2} \Rightarrow 3$ eqns and 4 unknowns \Rightarrow multiple possibilities!

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

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

 \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

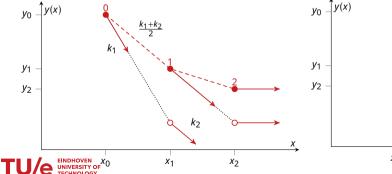
Classical RK2 method (= Heun's method, improved Euler method) $k_1 = f_i$ $k_2 = f(x_i + h, y_i + hk_1)$ $y_{i+1} = y_i + \frac{1}{2}h(k_1 + k_2)$

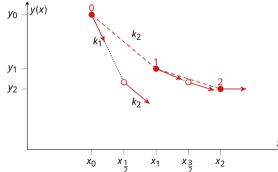
Explicit midpoint rule (modified Euler method)

$$k_1 = f_i$$

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

$$y_{i+1} = y_i + hk_2$$

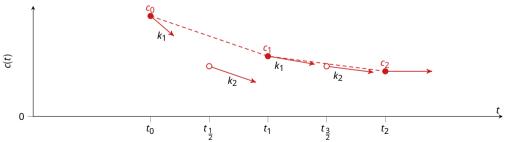




First order reaction in a batch reactor: $\frac{dc}{dt} = -kc$ with c(t = 0) = 1 mol m⁻³, k = 1 s⁻¹, $t_{\text{end}} = 2$ s.

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

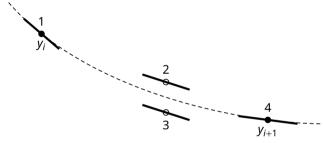
N	ζ	$rac{\zeta_{ m numerical} - \zeta_{ m analytical}}{\zeta_{ m analytical}}$	$r = \frac{\log\left(\frac{\varepsilon_j}{\varepsilon_{j-1}}\right)}{\log\left(\frac{N_{j-1}}{N_i}\right)}$
20	0.864178	5.634 × 10 ⁻⁴	_
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048 × 10 ⁻⁶	2.007

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

Can we do even better?



RK4 method (classical fourth order Runge-Kutta method)



$$k_{1} = f(x_{i}, y_{i})$$

$$k_{2} = f(x_{i} + \frac{1}{2}h, y_{i} + \frac{1}{2}hk_{1})$$

$$k_{3} = f(x_{i} + \frac{1}{2}h, y_{i} + \frac{1}{2}hk_{2})$$

$$k_{4} = f(x_{i} + h, y_{i} + hk_{3})$$

$$y_{i+1} = y_{i} + h\left(\frac{1}{6}k_{1} + \frac{1}{3}(k_{2} + k_{3}) + \frac{1}{6}k_{4}\right)$$

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{ m numerical} - \zeta_{ m analytical}}{\zeta_{ m analytical}}$	$r = \frac{\log\left(\frac{e_i}{e_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
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?



n Euler's method Rates of convergence Runge-Kusta methods Step size control Solving ODEs in Python Introduction Systems of ODEs Boundary value problems

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
- 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



Adaptive step size control

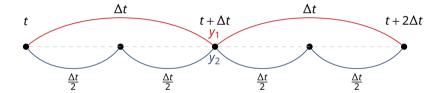
The step size (be it either position, time or both (PDEs)) cannot be decreased indefinitely to favour a higher accuracy, since each additional grid point causes additional computation time. It may be wise to adapt the step size according to the computation requirements.

Globally two different approaches can be used:

- 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



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



- 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: $h_{\text{new}} = \alpha h_{\text{old}} \left| \frac{\text{tol}}{\Lambda} \right|^{\frac{1}{5}}$ 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 Fuler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python
- 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



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 solt as the independent variable vector, and a soly the solution vector (or matrix for systems of ODEs).



We solve the system: $\frac{dx}{dt} = -k_1x + k_2, k_1 = 0.2, k_2 = 2.5$

• Create a lambda function:

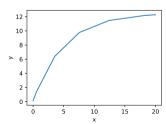
```
dvdx = lambda x, y: (-0.2*y + 2.5)
```

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

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

Draw the results by calling the relevant Matplotlib commands:

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



Solving ODEs in Python: example 2

We solve the system:
$$\frac{dx}{dt} = \begin{cases} \frac{k_1}{x^2} & t \le 10 \\ \frac{k_2}{x} - \frac{k_1}{v^2} & t > 10 \end{cases}$$
 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
```

Create a solution script

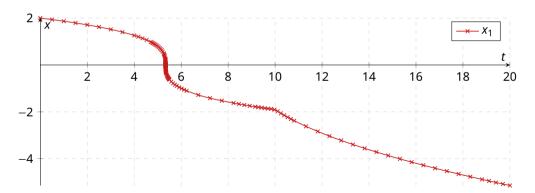
```
tspan = [0, 20]
x_init = [2]
sol = solve_ivp(myEqnFunction, tspan, x_init, rtol=1e-8, atol=1e-6)
```



Solving ODEs in Python: example 2

Plot the solution:

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



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))
```

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

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

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



Solving systems of ODEs in Python: example

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

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

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!

