Numerical integration of Ordinary Differential Equations

Kristina Kislyakova

University of Vienna - 19.11.2025, 26.11.2025

Outline

MAIN CONCEPTS

- Introduction
- Discretization of ODEs
- Single and multi-step methods
- Implicit vs explicit methods
- Sources of errors
- Summary

Outline

MAIN CONCEPTS

- Introduction
- Discretization of ODEs
- Single and multi-step methods
- Implicit vs explicit methods
- Sources of errors
- Summary

PRACTICAL EXAMPLES

- Python built-in functions
- Discussion about choosing the best method

Introduction

Definitions

"An ordinary differential equation (ODE) is a differential equation containing one or more functions of one independent variable and its derivatives. The term ordinary is used in contrast with the term partial differential equation which may be with respect to more than one independent variable." (Wikipedia)

EXAMPLES:

$$\boxed{m\frac{dy}{dx} + g - x^2 + 0.5x = 0}$$

$$\boxed{a\frac{d^2y}{dx^2} + b\frac{dy}{dx} = x^{-1} + 1}$$

These equations are linear ODEs, because they do not contain higher powers of the derivatives e.g. $(dy/dt)^2$, $(dy/dx)^{0.5}$, etc.

Introduction

Definitions

“An ordinary differential equation (ODE) is a differential equation containing one or more functions of one independent variable and its derivatives. The term ordinary is used in contrast with the term partial differential equation which may be with respect to more than one independent variable.” (Wikipedia)

A **system of ODEs** is a set of coupled ODEs for several variables:

$$\boxed{\begin{aligned}\frac{dy_1}{dx} &= f_1(x, y_1, y_2, y_3)\\ \frac{dy_2}{dx} &= f_2(x, y_1, y_2, y_3)\\ \frac{dy_3}{dx} &= f_3(x, y_1, y_2, y_3)\end{aligned}}$$

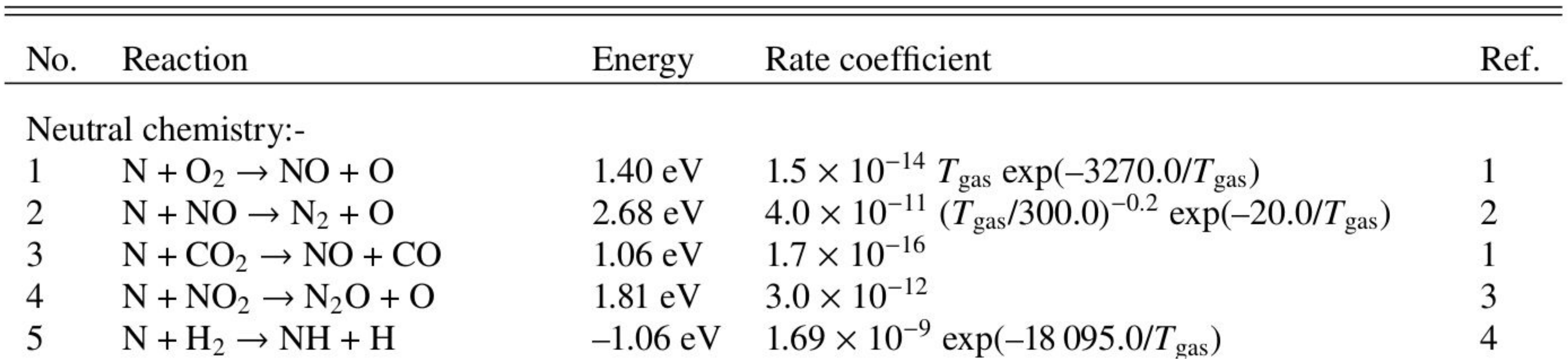
Introduction

For simplicity for the rest of the lecture, we assume that the independent variable is time, ***t***, and the ODEs being solved express time derivatives (i.e. evolution).

Initial value problems: problems in which we know a value of the dependent variable and we can calculate the time-derivative of the variable, and we want to calculate the dependent variable at all times

$$\boxed{\begin{aligned} &\left.\begin{aligned} y(t_0) &= y_0 \\ \frac{dy}{dt} &= f(t, y) \end{aligned}\right\} \textbf{Known} \\ &y(t) \longleftarrow \textbf{Desired} \end{aligned}}$$

Example: chemical kinetics



No.	Reaction	Energy	Rate coefficient	Ref.
Neutral chemistry:-				
1	$N + O_2 \rightarrow NO + O$	1.40 eV	$1.5 \times 10^{-14}\ T_{gas} \exp(-3270.0/T_{gas})$	1
2	$N + NO \rightarrow N_2 + O$	2.68 eV	$4.0 \times 10^{-11}\ (T_{gas}/300.0)^{-0.2} \exp(-20.0/T_{gas})$	2
3	$N + CO_2 \rightarrow NO + CO$	1.06 eV	1.7×10^{-16}	1
4	$N + NO_2 \rightarrow N_2O + O$	1.81 eV	3.0×10^{-12}	3
5	$N + H_2 \rightarrow NH + H$	–1.06 eV	$1.69 \times 10^{-9} \exp(-18\,095.0/T_{gas})$	4

Johnstone+, 2018

Rate of change of jth species density

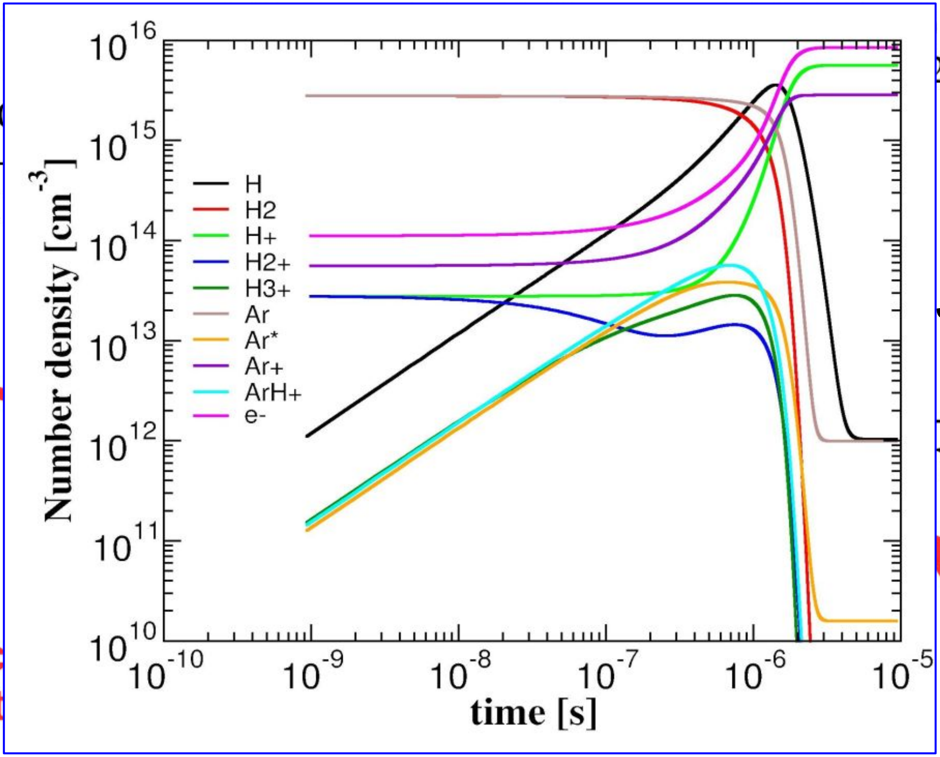
$$\frac{dn_j}{dt} = \sum_k R_k - \sum_i R_i$$

Reactions that create jth species

Reactions that destroy jth species

Example: chemical kinetics

No.	Reaction	Energy	Rate coefficient	Ref.
Neutral chemistry:-				
1	$N + O_2 \rightarrow NO + O$			1
2	$N + NO \rightarrow N_2 + O$		$20.0/T_{gas})$	2
3	$N + CO_2 \rightarrow NO +$			1
4	$N + NO_2 \rightarrow N_2O +$			3
5	$N + H_2 \rightarrow NH + H$			4



Johnstone+, 2018

Rate of change of jth species density

Reac
creat

R_i

Reactions that destroy jth species

Discretization of ODEs

Almost every problem that has to be solved numerically involves discretization.
Example:

$$\boxed{\frac{dy(x)}{dx} \approx \frac{y(x+\delta x) - y(x-\delta x)}{2\delta x}}$$

Other options:

$$\boxed{\frac{dy(x)}{dx} \approx \frac{y(x) - y(x-\delta x)}{\delta x}}$$

$$\boxed{\frac{dy(x)}{dx} \approx \frac{y(x+\delta x) - y(x)}{\delta x}}$$

Numerical integration of ODEs

A discretized ODE can be **numerically integrated:**

$$\boxed{\frac{dy}{dt} \approx \frac{\overset{\text{DESIRED}}{y(t_{n+1})} - \overset{\text{KNOWN}}{y(t_n)}}{t_{n+1} - t_n}}$$

For simplicity, let's write some of these terms **as follows**:

$$\boxed{\begin{aligned} y_n &= y(t_n) \\ y_{n+1} &= y(t_{n+1}) \end{aligned}} \qquad \boxed{\Delta t = t_{n+1} - t_n}$$

Numerical integration of ODEs

Rearranging for the desired quantity gives:

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt} \right)}$$

Question 1: how to calculate the *dy/dt* term? There are many schemes, and they differ mostly in how they deal with this term.
Question 2: how to calculate the timestep length?

Numerical integration of ODEs

Rearranging for the desired quantity gives:

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt} \right)}$$

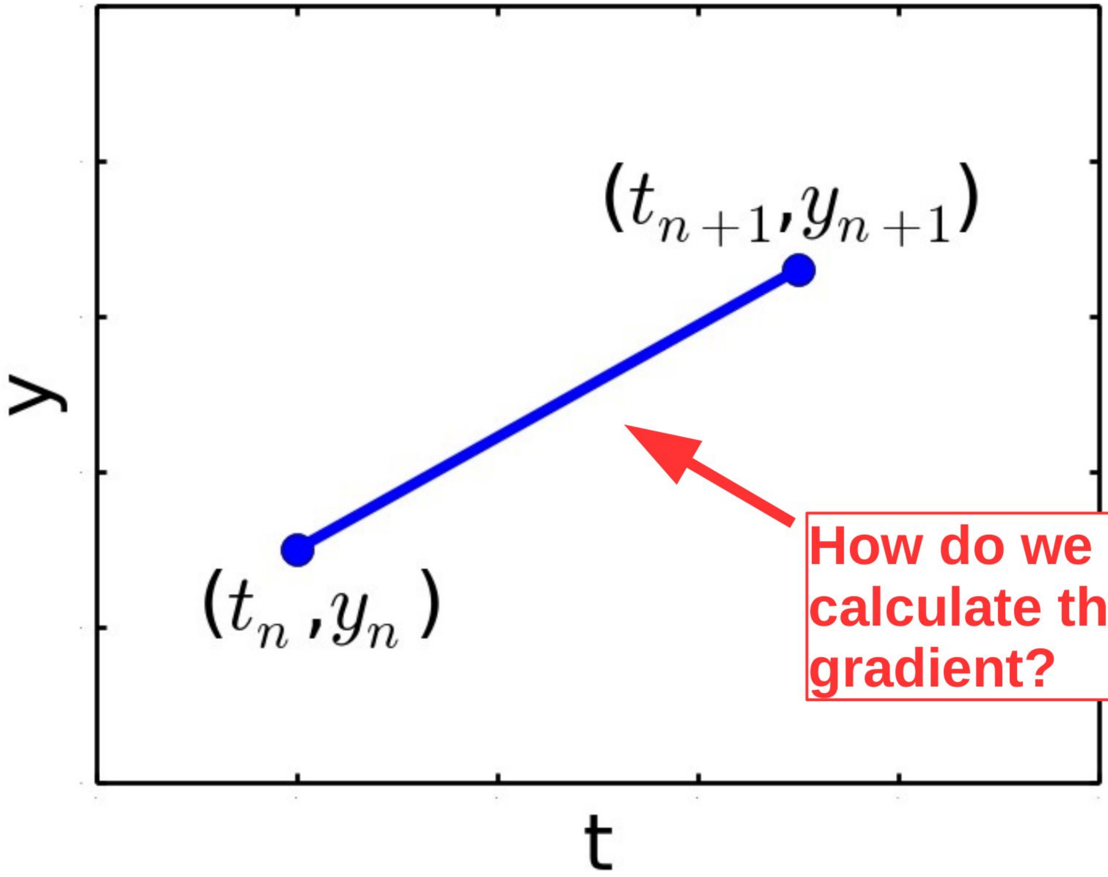
Question 1: how to calculate the *dy/dt* term? There are many schemes, and they differ mostly in how they deal with this term.
Question 2: how to calculate the timestep length?

Assumption: we know that

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

Numerical integration of ODEs

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt} \right)}$$



How do we calculate this gradient?

The simplest scheme: forward Euler

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

The simplest and easiest assumption that we can make is that the *dy/dt* term is equal to the value at the current time:

FUNDAMENTAL ASSUMPTION

$$\boxed{\frac{dy}{dt} = f(t_n, y_n)}$$

The simplest scheme: forward Euler

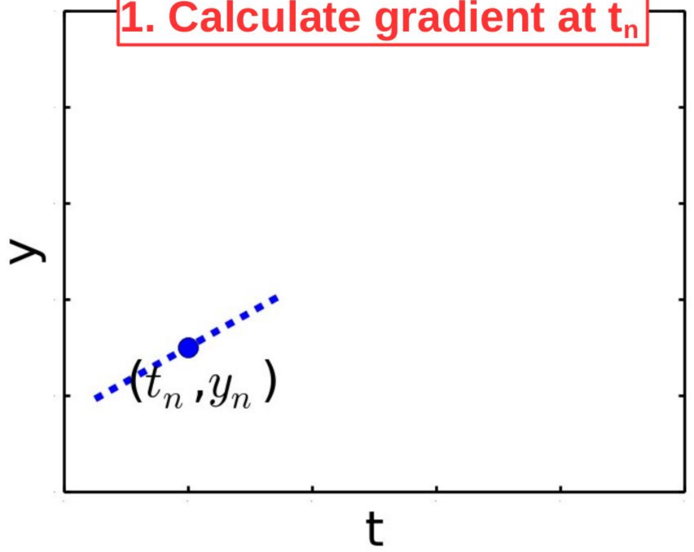
$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

The simplest and easiest assumption that we can make is that the *dy/dt* term is equal to the value at the current time:

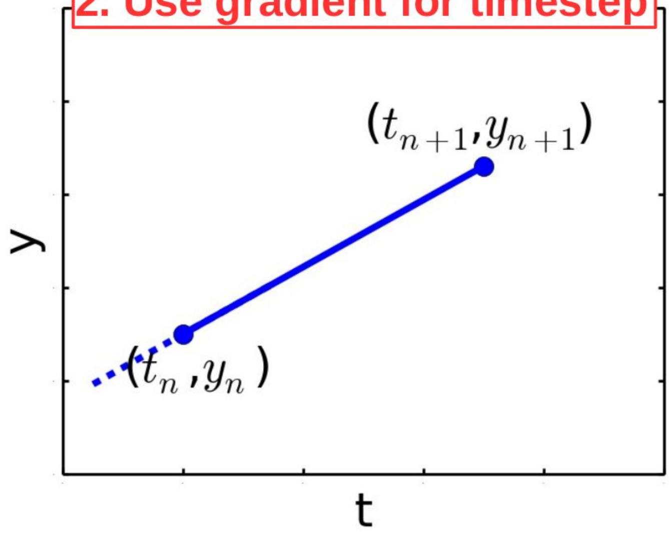
FUNDAMENTAL ASSUMPTION

$$\boxed{\frac{dy}{dt} = f(t_n, y_n)}$$

1. Calculate gradient at t_n



2. Use gradient for timestep



Derivation of forward Euler

Express function *y(t)* as Taylor expansion of y at t_n:

$$\boxed{y(t) = y_n + [t - t_n]f_n + \frac{1}{2}[t - t_n]^2 f_n' + \frac{1}{6}[t - t_n]^3 f_n'' + \ldots}$$

Derivation of forward Euler

Express function *y(t)* as Taylor expansion of y at t_n:

$$y(t) = y_n + [t - t_n] f_n + \frac{1}{2}[t - t_n]^2 f'_n + \frac{1}{6}[t - t_n]^3 f''_n + \ldots$$

Use this to get $y(t_{n+1})$:

$$y_{n+1} = y_n + \Delta t f_n + \frac{1}{2}\Delta t^2 f'_n + \frac{1}{6}\Delta t^3 f''_n + \ldots$$

Derivation of forward Euler

Express function *y(t)* as Taylor expansion of y at t_n:

$$\boxed{y(t) = y_n + [t - t_n] f_n + \frac{1}{2}[t - t_n]^2 f'_n + \frac{1}{6}[t - t_n]^3 f''_n + \ldots}$$

Use this to get $y(t_{n+1})$:

$$\boxed{y_{n+1} = y_n + \Delta t f_n + \frac{1}{2}\Delta t^2 f'_n + \frac{1}{6}\Delta t^3 f''_n + \ldots}$$

Ignore all but the first two terms on the right hand side:

Forward Euler scheme

$$\boxed{y_{n+1} = y_n + \Delta t f_n}$$

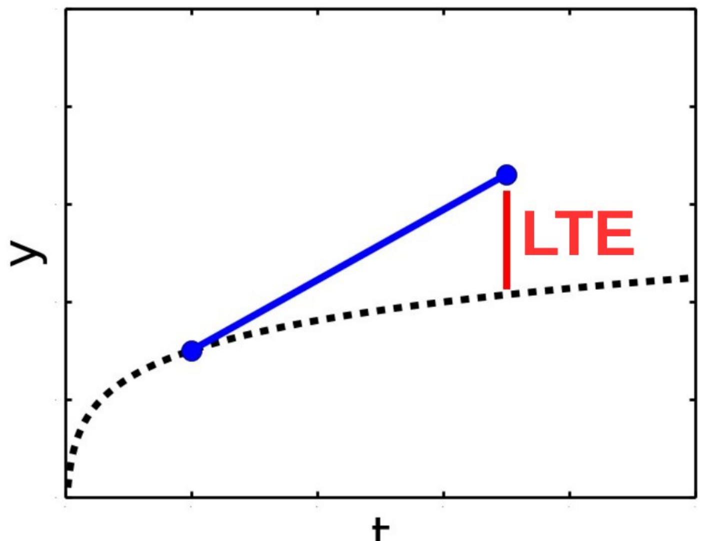
Local truncation error

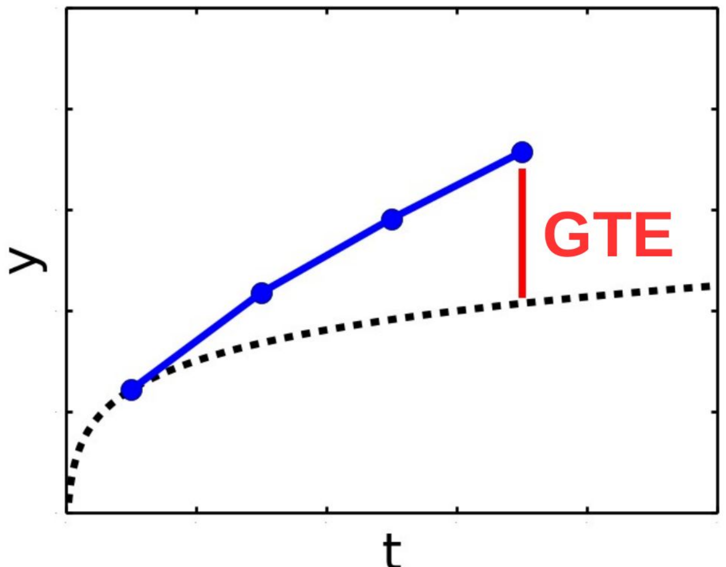
$$\boxed{\text{LTE} = \frac{1}{2}\Delta t^2 f'_n + \frac{1}{6}\Delta t^3 f''_n + \ldots}$$

Truncation errors

Local truncation error: difference between true solution and approximate solution for an individual iteration

Global truncation error: difference between true solution and approximate solution after all iterations (accumulation of LTEs aka error propagation)



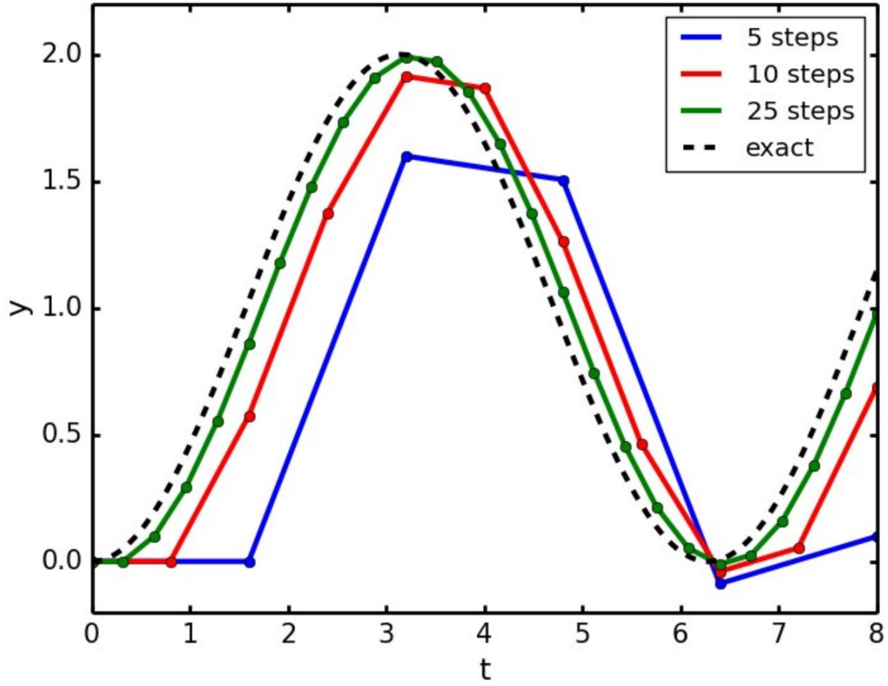


Initial value problem:

$$t_0 = 0 \qquad y_0 = 0$$

$$\frac{dy}{dt} = \sin(t)$$

The quality of the solution depends on the timestep length that is used. Smaller timesteps give better results always, but take longer to calculate.



Analytical solution: $y = -\cos(t) + 1$

Expansion of forward Euler (and other methods) to higher order

Many ODEs we encounter are of higher order. Example: Newton's second law $F = ma.$
Here, $a = d^2y/dt^2$, so if we write it as an ODE, it becomes

$$\boxed{\frac{d^2y}{dt^2} = F(t, y', y)} \qquad (1)$$

Expansion of forward Euler (and other methods) to higher order

Many ODEs we encounter are of higher order. Example: Newton's second law $F = ma.$
Here, $a = d^2y/dt^2$, so if we write it as an ODE, it becomes

$$\boxed{\frac{d^2y}{dt^2} = F(t, y', y)} \qquad (1)$$

To solve this equation numerically, we can define two variables, $y_1 = dy/dt$, and $y_2 = y$.

Expansion of forward Euler (and other methods) to higher order

Many ODEs we encounter are of higher order. Example: Newton's second law $F = ma.$
Here, $a = d^2y/dt^2$, so if we write it as an ODE, it becomes

$$\boxed{\frac{d^2y}{dt^2} = F(t, y', y)} \qquad (1)$$

To solve this equation numerically, we can define two variables, $y_1 = dy/dt$, and $y_2 = y$.
Now our equation (1) can be written as

$$\boxed{\frac{dy_1}{dt} = \frac{d^2y}{dt^2} = F(t, y', y)}$$

By definition, we have $\boxed{\frac{dy_2}{dt} = \frac{dy}{dt} = y_1}$

Expansion of forward Euler (and other methods) to higher order

Many ODEs we encounter are of higher order. Example: Newton's second law $F = ma.$
Here, $a = d^2y/dt^2$, so if we write it as an ODE, it becomes

$$\boxed{\frac{d^2y}{dt^2} = F(t, y', y)} \qquad (1)$$

To solve this equation numerically, we can define two variables, $y_1 = dy/dt$, and $y_2 = y$.
Now our equation (1) can be written as

$$\boxed{\frac{dy_1}{dt} = \frac{d^2y}{dt^2} = F(t, y', y)}$$

By definition, we have $\boxed{\frac{dy_2}{dt} = \frac{dy}{dt} = y_1}$

So, equation (1) becomes a system of ODEs
that can be solved using e.g. forward Euler:

$$\boxed{\begin{aligned} \frac{dy_1}{dt} &= F(t, y_1, y_2) \\ \frac{dy_2}{dt} &= y_1 \end{aligned}} \qquad (2)$$

Disadvantages of Forward Euler

Forward Euler is very simple, which leads to a variety of problems.

- **Stability issues:** for "stiff" ODEs, where solutions can change rapidly, it requires a very small step size to stay stable. If the step size is too large, numerical errors cause the solution to diverge

- **Low accuracy:** it is a first-order method, which means that the error per step is proportional to the square of the step size

- **Inefficiency for small step sizes:** since a very small step size is required for stability and/or accuracy, the forward Euler method quickly becomes computationally inefficient.

- **Error accumulation:** errors accumulate over each step due to the simple approximation, leading to significant deviation from the true solution over time → bad for long-term integration

In summary, forward Euler is not suitable for high precision calculations.

- **Stability issues:** for "stiff" ODEs, where solutions can change rapidly, it requires a very small step size to stay stable. If the step size is too large, numerical errors cause the solution to diverge

- **Low accuracy:** it is a first-order method, which means that the error per step is proportional to the square of the step size

- **Inefficiency for small step sizes:** since a very small step size is required for stability and/or accuracy, the forward Euler method quickly becomes computationally inefficient.

- **Error accumulation:** errors accumulate over each step due to the simple approximation, leading to significant deviation from the true solution over time → bad for long-term integration

How to make it better? A possible solution: the Euler-Cromer method

The **semi-implicit Euler method** is a modification of the Euler method for solving Hamilton's equations, a system of ODEs that arises in classical mechanics. The method can be applied to a pair of ODEs:

$$\boxed{\begin{aligned}\frac{dx}{dt} &= f(t,v)\\ \frac{dv}{dt} &= g(t,x),\end{aligned}}$$

How to make it better? A possible solution: the Euler-Cromer method

The **semi-implicit Euler method** is a modification of the Euler method for solving Hamilton's equations, a system of ODEs that arises in classical mechanics. The method can be applied to a pair of ODEs:

t and *v* may be scalars or vectors

$$\boxed{\begin{aligned} \frac{dx}{dt} &= f(t, v) \\ \frac{dv}{dt} &= g(t, x), \end{aligned}}$$

f and *g* are given functions

How to make it better? A possible solution: the Euler-Cromer method

The **semi-implicit Euler method** is a modification of the Euler method for solving Hamilton's equations, a system of ODEs that arises in classical mechanics. The method can be applied to a pair of ODEs:

t and *v* may be scalars or vectors

$$\boxed{\begin{aligned}\frac{dx}{dt} &= f(t,v)\\ \frac{dv}{dt} &= g(t,x),\end{aligned}}$$

f and *g* are given functions

We also need some initial conditions:

$$x(t_0) = x_0, \qquad v(t_0) = v_0.$$

The Hamiltonian describes the connection between the potential and kinetic energies:

$$H = T(t,v) + V(t,x).$$

The Euler-Cromer method

Solution:

$$\boxed{\begin{aligned} v_{n+1} &= v_n + g(t_n, x_n)\,\Delta t \\ x_{n+1} &= x_n + f(t_n, v_{n+1})\,\Delta t \end{aligned}}$$

The difference with the standard Euler method is that the semi-implicit Euler method uses $\boldsymbol{v_{n+1}}$ in the equation for x_{n+1}, while the Euler method uses $\boldsymbol{v_n}$.

We can also rearrange the solution, if we want to swap *x* and *v* in it:

$$\boxed{\begin{aligned} x_{n+1} &= x_n + f(t_n, v_n)\,\Delta t \\ v_{n+1} &= v_n + g(t_n, x_{n+1})\,\Delta t \end{aligned}}$$

The Euler-Cromer method

Solution:

$$\boxed{\begin{aligned} v_{n+1} &= v_n + g(t_n, x_n)\,\Delta t \\ x_{n+1} &= x_n + f(t_n, v_{n+1})\,\Delta t \end{aligned}}$$

The difference with the standard Euler method is that the semi-implicit Euler method uses $\boldsymbol{v_{n+1}}$ in the equation for x_{n+1}, while the Euler method uses $\boldsymbol{v_n}$.

We can also rearrange the solution, if we want to swap *x* and *v* in it:

$$\boxed{\begin{aligned} x_{n+1} &= x_n + f(t_n, v_n)\,\Delta t \\ v_{n+1} &= v_n + g(t_n, x_{n+1})\,\Delta t \end{aligned}}$$

Just like the standard Forward Euler method, the Euler-Cromer method is the 1st order method (which means that the error is of the order of Δt). However, it is a **symplectic integrator** (an integrator for Hamiltonian systems), which means that it *almost conserves the energy*

The Euler-Cromer method: an example

The motion of a spring (can also describe other oscillatory behavior):

$$\boxed{\begin{aligned}\frac{dx}{dt} &= v(t)\\ \frac{dv}{dt} &= -\frac{k}{m}\,x = -\omega^2\,x.\end{aligned}}$$

The semi-implicit Euler method for these equations gives:

$$\boxed{\begin{aligned}v_{n+1} &= v_n - \omega^2\,x_n\,\Delta t\\ x_{n+1} &= x_n + v_{n+1}\,\Delta t.\end{aligned}}$$

The iteration preserves the energy functional $E_h(x,v) = \frac{1}{2}\left(v^2 + \omega^2\,x^2 - \omega^2\,\Delta t\,vx\right)$

If the step size is sufficiently small, the difference between the analytical solution and the calculated solution at any given time is of the order of $O(\Delta t)$

Another similar solution: Leapfrog integration

This is a method specifically developed for integration of the equations of the form

$$\boxed{\ddot{x} = \frac{d^2 x}{dt^2} = A(x),}$$

Another similar solution: Leapfrog integration

This is a method specifically developed for integration of the equations of the form

$$\boxed{\ddot{x} = \frac{d^2x}{dt^2} = A(x),}$$

which can again be presented as two equations (a typical problem from classical mechanics).

$$\boxed{\dot{v} = \frac{dv}{dt} = A(x), \qquad \dot{x} = \frac{dx}{dt} = v}$$

Another similar solution: Leapfrog integration

This is a method specifically developed for integration of the equations of the form

$$\ddot{x} = \frac{d^2 x}{dt^2} = A(x),$$

which can again be presented as two equations (a typical problem from classical mechanics).

$$\dot{v} = \frac{dv}{dt} = A(x), \qquad \dot{x} = \frac{dx}{dt} = v$$

Leapfrog integration is equivalent to updating positions and velocities at different interleaved time points, staggered in such a way that they **"leapfrog"** over each other. The solutions for position and velocity are (below a_i is acceleration):

Another similar solution: Leapfrog integration

This is a method specifically developed for integration of the equations of the form

$$\ddot{x} = \frac{d^2 x}{dt^2} = A(x),$$

which can again be presented as two equations (a typical problem from classical mechanics).

$$\dot{v} = \frac{dv}{dt} = A(x), \qquad \dot{x} = \frac{dx}{dt} = v$$

Leapfrog integration is equivalent to updating positions and velocities at different interleaved time points, staggered in such a way that they **"leapfrog"** over each other. The solutions for position and velocity are (below a_i is acceleration):

$$\begin{aligned} a_i &= A(x_i), \\ v_{i+1/2} &= v_{i-1/2} + a_i\,\Delta t, \\ x_{i+1} &= x_i + v_{i+1/2}\,\Delta t, \end{aligned}$$

Or, if you want the velocity also at integer steps:

$$\begin{aligned} x_{i+1} &= x_i + v_i\,\Delta t + \tfrac{1}{2}\,a_i\,\Delta t^{\,2}, \\ v_{i+1} &= v_i + \tfrac{1}{2}(a_i + a_{i+1})\,\Delta t. \end{aligned}$$

Leapfrog integration: why is it sometimes so useful

- It solves a second order ODE **directly**, and a very important type of ODE

- It is **reversible**: if you advance the solution of an ODE from its initial condition to a future point, make that point your new initial condition, and reverse time, you can step back to the point where you started (there will be only the loss of accuracy due to floating point)

- Similar to the Euler-Cromer method, the leapfrog method (approximately) **conserves energy**

Classical Runge-Kutta

The classical Runge-Kutta method is an explicit 4th-order single step method very useful in many cases. The method is given by

$$y_{n+1} = y_n + \frac{1}{6}\Delta t\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$

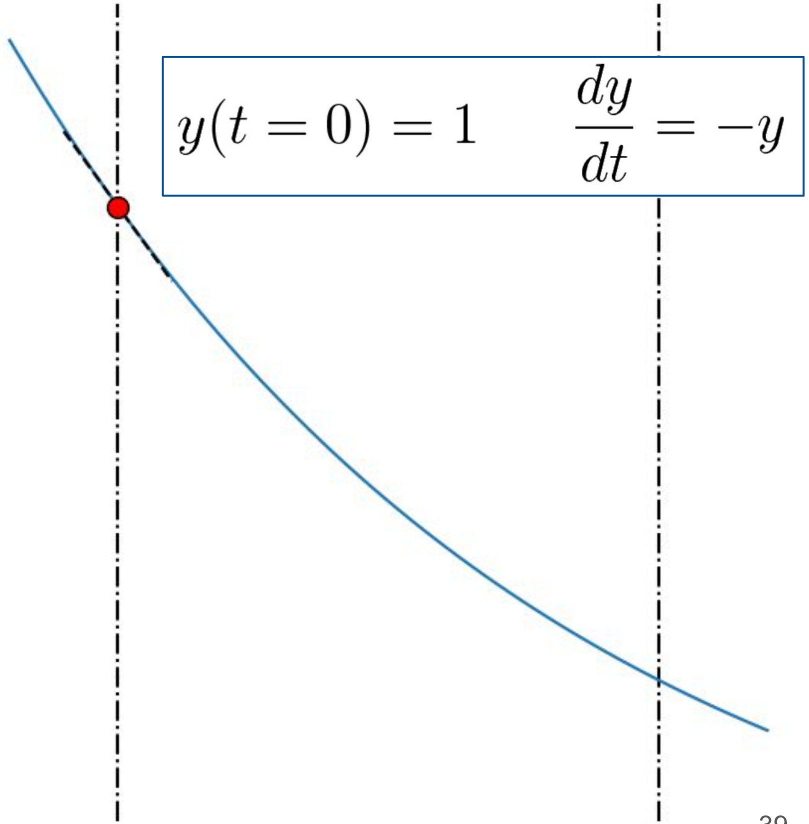
The four *k* coefficients are all gradients given at different points in the timestep. The first, k_1, is calculated at t_n. The second and third, k_2 and k_3, are calculated at the half-way point, and the forth, k_4, is calculated at the end of the timestep.

Classical Runge-Kutta

Fourth-order Runge-Kutta

One of the most often used single step methods

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



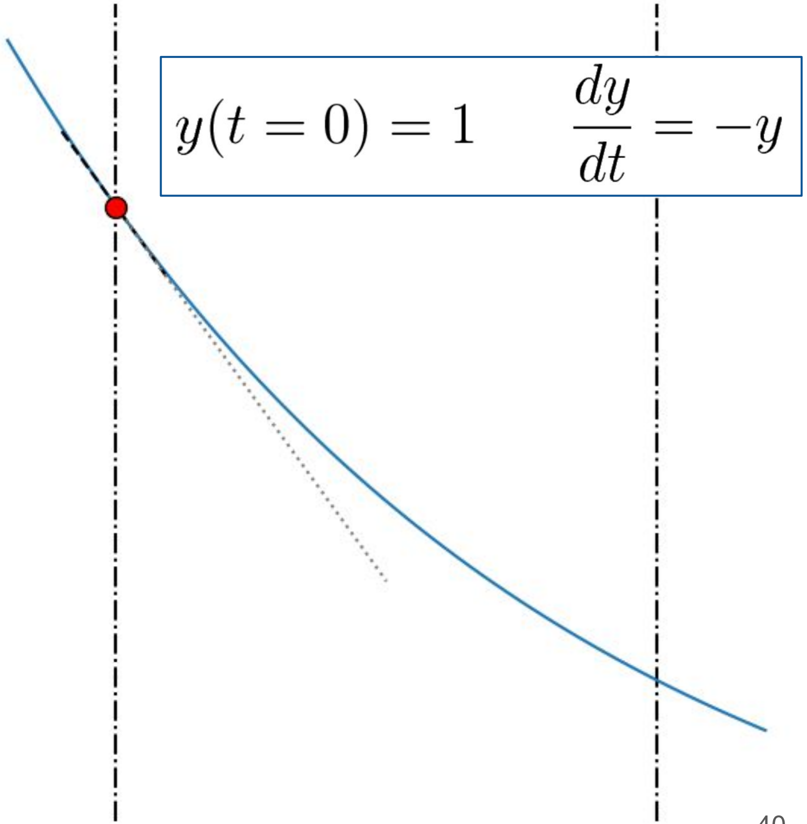
$$\boxed{y(t=0) = 1 \qquad \frac{dy}{dt} = -y}$$

Classical Runge-Kutta

Step 1: Get k_1 from gradient at t_n

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

$$k_1 = f(t_n, y_n)$$



$$y(t=0) = 1 \qquad \frac{dy}{dt} = -y$$

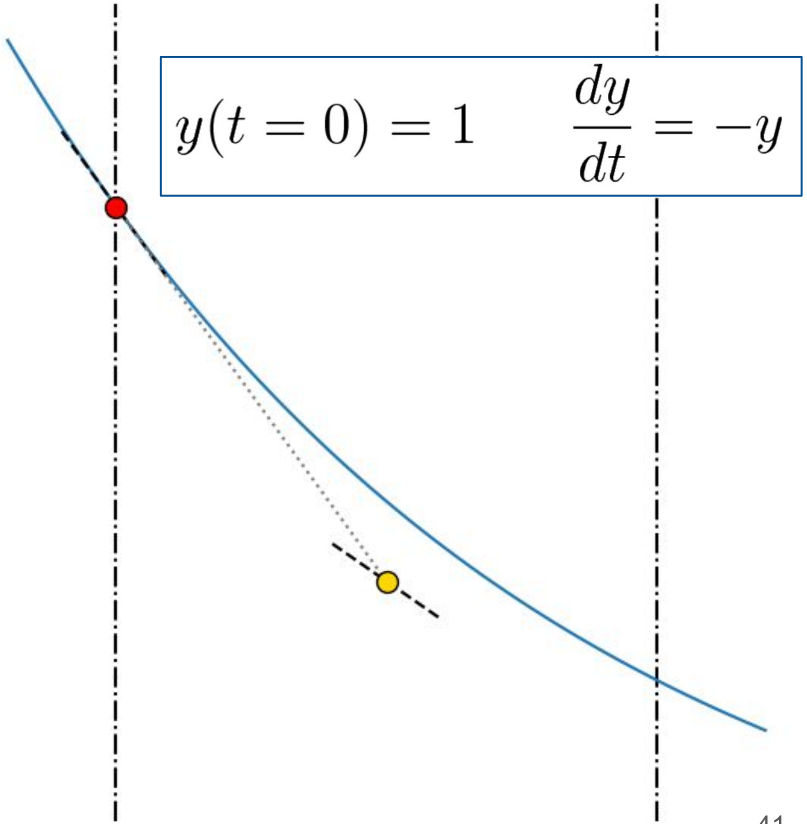
Classical Runge-Kutta

Step 2: Make half step with k_1 and get k_2 from gradient at new point

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

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$



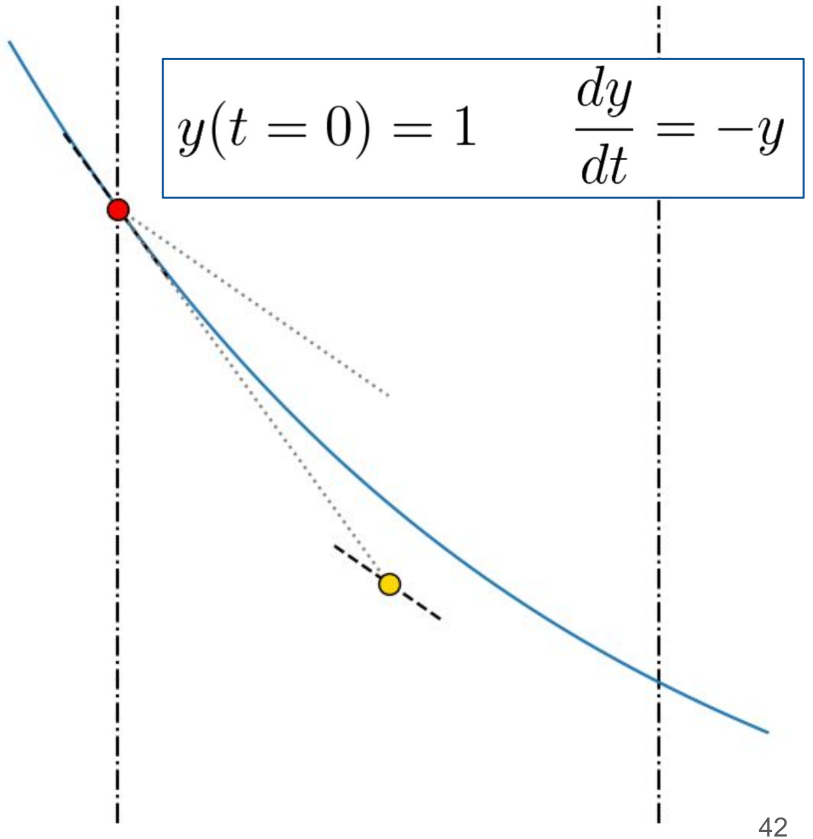
Classical Runge-Kutta

Step 2: Make half step with k_1 and get k_2 from gradient at new point

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

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$



$$y(t=0) = 1 \qquad \frac{dy}{dt} = -y$$

Classical Runge-Kutta

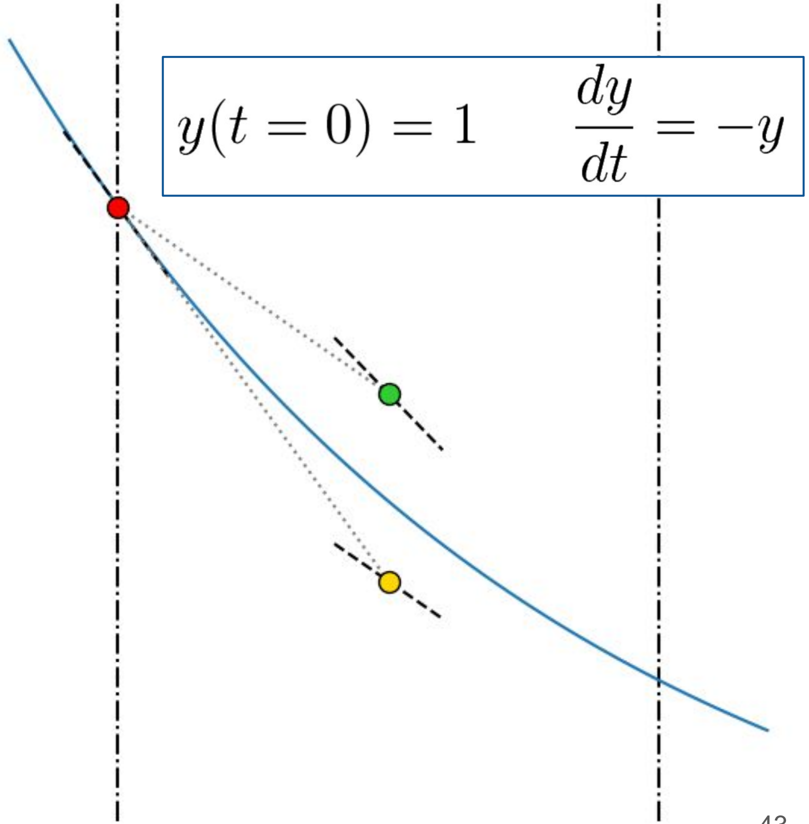
Step 3: Make half step with k_2 and get k_3 from gradient at new point

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

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$



Classical Runge-Kutta

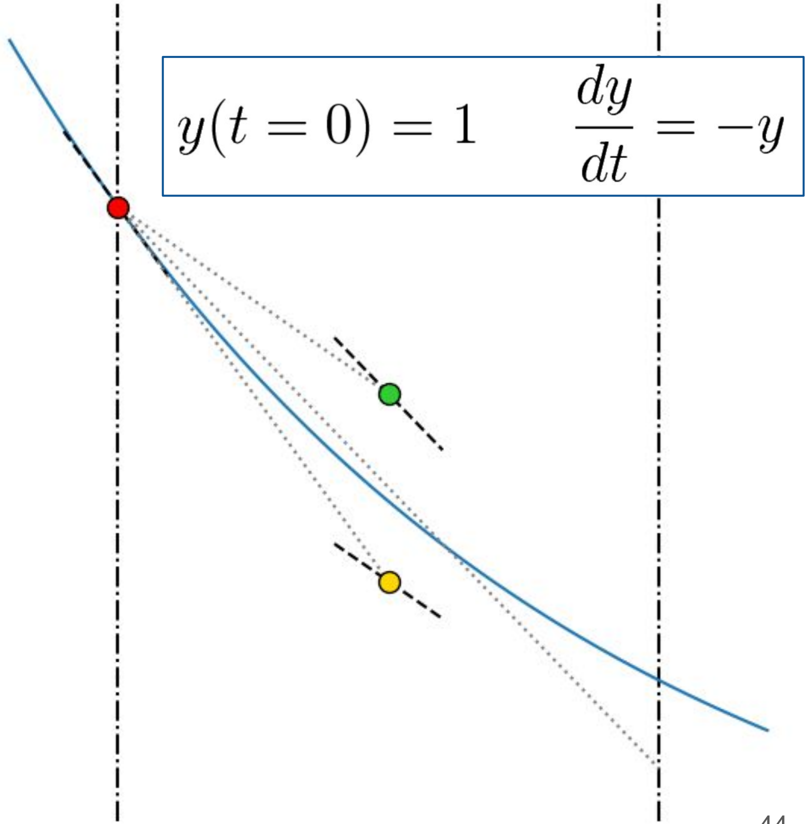
Step 3: Make half step with k_2 and get k_3 from gradient at new point

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

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$



Classical Runge-Kutta

Step 4: Make full step with k_3 and get k_4 from gradient at new point

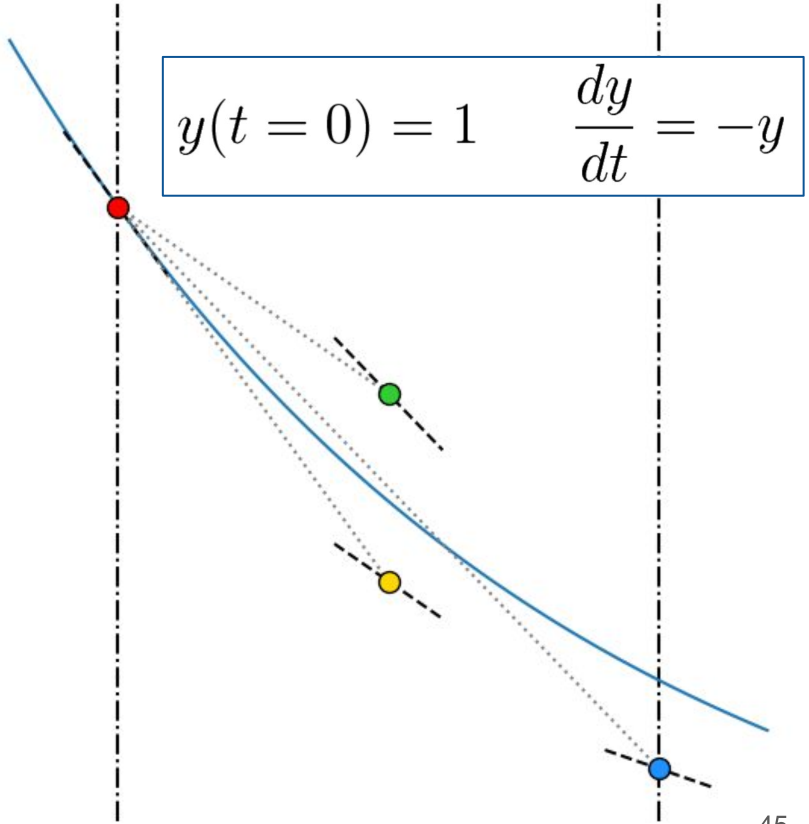
$$\frac{dy}{dt} = f(t, y)$$

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$



Classical Runge-Kutta

Final step: average all four gradients and use it for full step

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

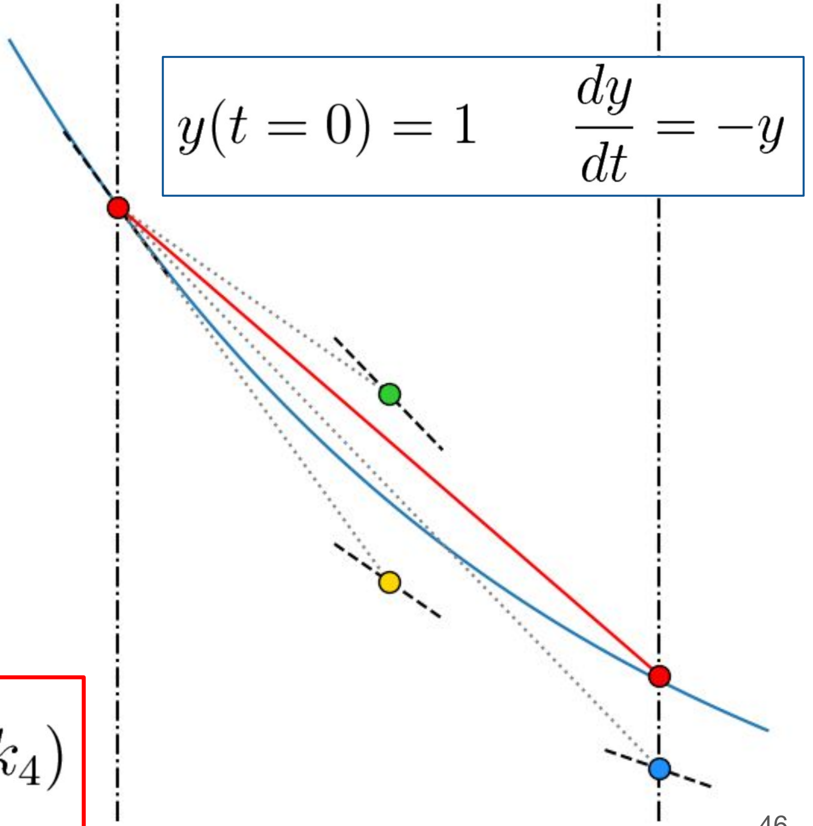
$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$

$$\boxed{y_{n+1} = y_n + \frac{1}{6}\Delta t\left(k_1 + 2k_2 + 2k_3 + k_4\right)}$$



$$\boxed{y(t=0) = 1 \qquad \frac{dy}{dt} = -y}$$

Classical Runge-Kutta

Final step: average all four gradients and use it for full step

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

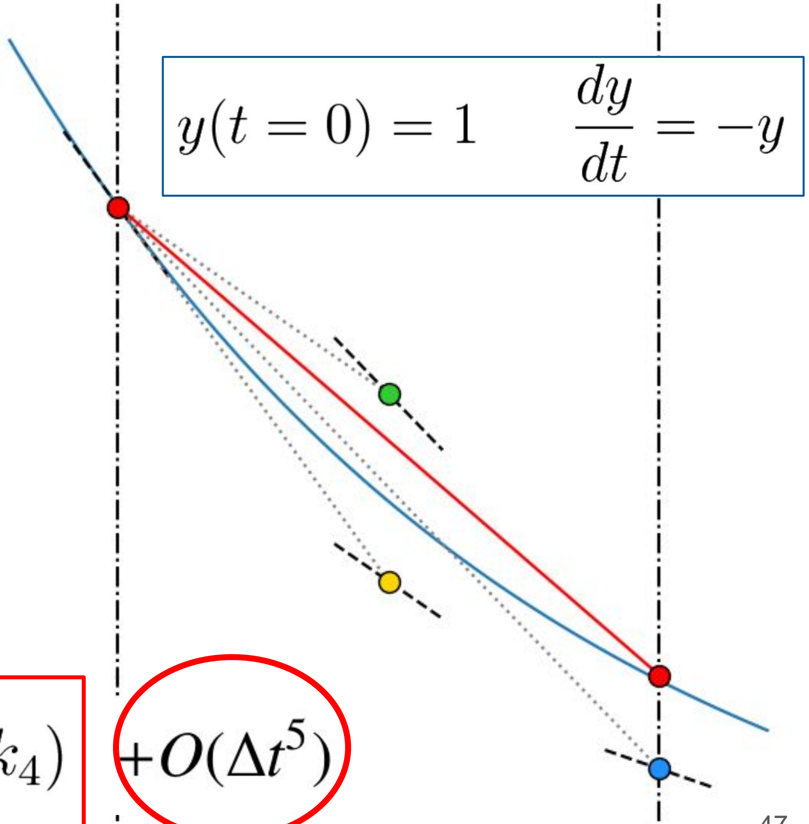
$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1)$$

$$k_3 = f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_2)$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$

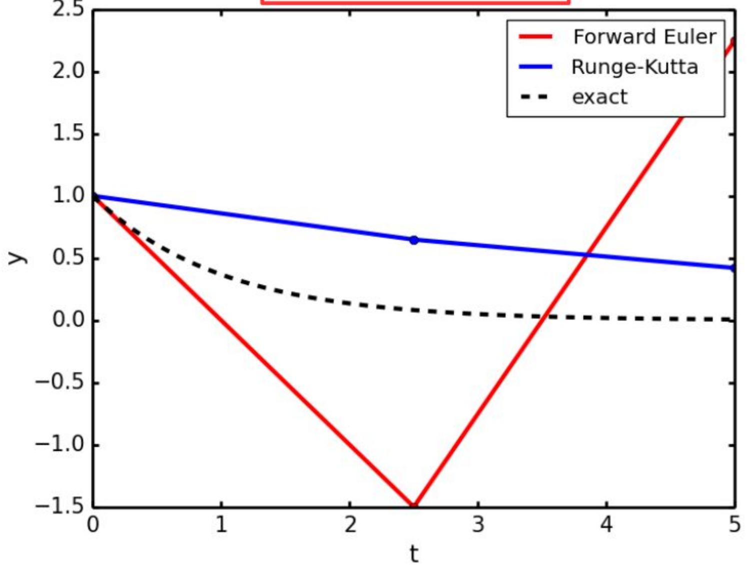
$$\boxed{y_{n+1} = y_n + \frac{1}{6}\Delta t \left(k_1 + 2k_2 + 2k_3 + k_4\right)} + O(\Delta t^5)$$



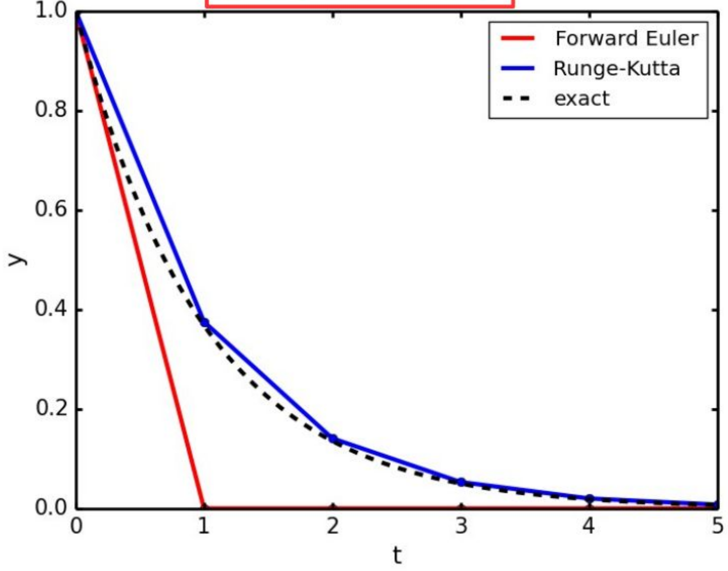
Example

$$y(t=0)=1 \qquad \frac{dy}{dt}=-y$$

2 STEPS



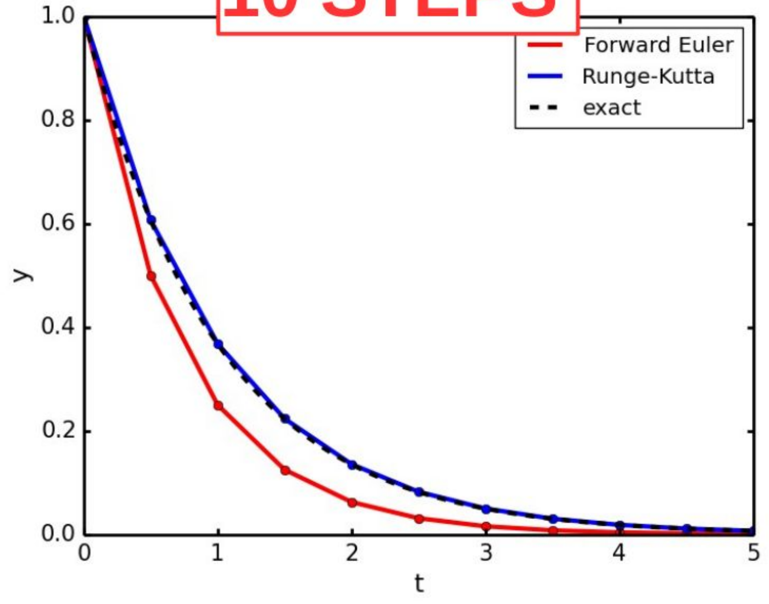
5 STEPS



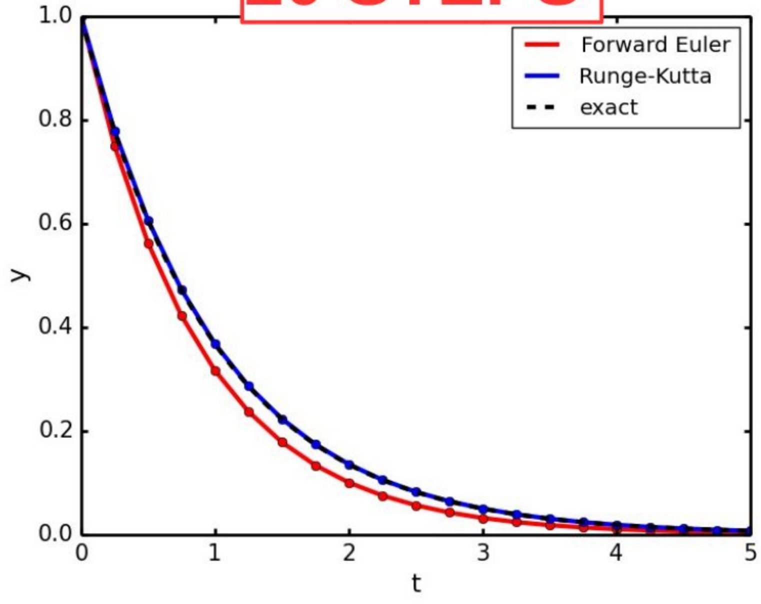
Example

$$y(t=0)=1 \qquad \frac{dy}{dt}=-y$$

10 STEPS



20 STEPS



Generalised explicit Runge-Kutta and ‘Butcher tableaus’

$$\boxed{k_i = f\left(t_n + c_i\Delta t, y_n + \Delta t\sum_{j=1}^{i-1} a_{ij}k_j\right)}$$

In general, $c_1 = 0$ and $a_{1j} = 0$ and the gradient terms can be written

$$\begin{aligned}
k_1 &= f(t_n, y_n),\\
k_2 &= f(t_n + c_2 h, y_n + (a_{21}k_1)h),\\
k_3 &= f(t_n + c_3 h, y_n + (a_{31}k_1 + a_{32}k_2)h),\\
&\ \vdots\\
k_s &= f(t_n + c_s h, y_n + (a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1})h).
\end{aligned}$$

Generalised explicit Runge-Kutta and ‘Butcher tableaus’

$$\boxed{k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right)}$$

In general, c_1 = *0* and a_{1j} = *0* and the gradient terms can be written

$$\begin{aligned}
k_1 &= f(t_n, y_n),\\
k_2 &= f(t_n + c_2 h, y_n + (a_{21} k_1) h),\\
k_3 &= f(t_n + c_3 h, y_n + (a_{31} k_1 + a_{32} k_2) h),\\
&\ \ \vdots\\
k_s &= f(t_n + c_s h, y_n + (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}) h).
\end{aligned}$$

To specify a particular method, one needs to provide the integer *s* (the number of stages), and the coefficients a_i (for $1 \le j < i \le s$), b_i (for $i = 1, 2, ..., s$) and c_i (for $i = 2, 3, ..., s$). The matrix $[a_{ij}]$ is called the *Runge–Kutta matrix*, while the b_i and c_i are known as the *weights* and the *nodes*.

Generalised explicit Runge-Kutta and ‘Butcher tableaus’

In general, $c_1 = 0$ and $a_{1j} = 0$ and the gradient terms can be written as

$$\begin{array}{c|c} c & A \\ \hline & b^\top \end{array}$$

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

Generalised explicit Runge-Kutta and 'Butcher tableaus'

In general, $c_1 = 0$ and $a_{1j} = 0$ and the gradient terms can be written as

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

A Taylor series expansion shows that the Runge–Kutta method is consistent only if $\sum_{i=1}^{s} b_i = 1.$

Generalised explicit Runge-Kutta and 'Butcher tableaus'

In general, $c_1 = 0$ and $a_{1j} = 0$ and the gradient terms can be written as

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

A Taylor series expansion shows that the Runge–Kutta method is consistent only if $\sum_{i=1}^{s} b_i = 1.$

One can specify that the method has a certain order *p*, meaning that the local truncation error is $O(h^{p+1})$

Generalised explicit Runge-Kutta and 'Butcher tableaus'

In general, $c_1 = 0$ and $a_{1j} = 0$ and the gradient terms can be written as

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

A Taylor series expansion shows that the Runge–Kutta method is consistent only if $$\sum_{i=1}^{s} b_i = 1.$$

One can specify that the method has a certain order p, meaning that the local truncation error is $O(h^{p+1})$

RK methods can be explicit and implicit. They can also be one step and multi-step.
It is a big family of methods

Multi-step methods

Basic idea

Predictor-corrector methods store the solution along the way, and use those results to extrapolate the solution for the following step. Then, these methods introduce a correction for the extrapolation by using the derivative information at the new point. These are best for very smooth functions.

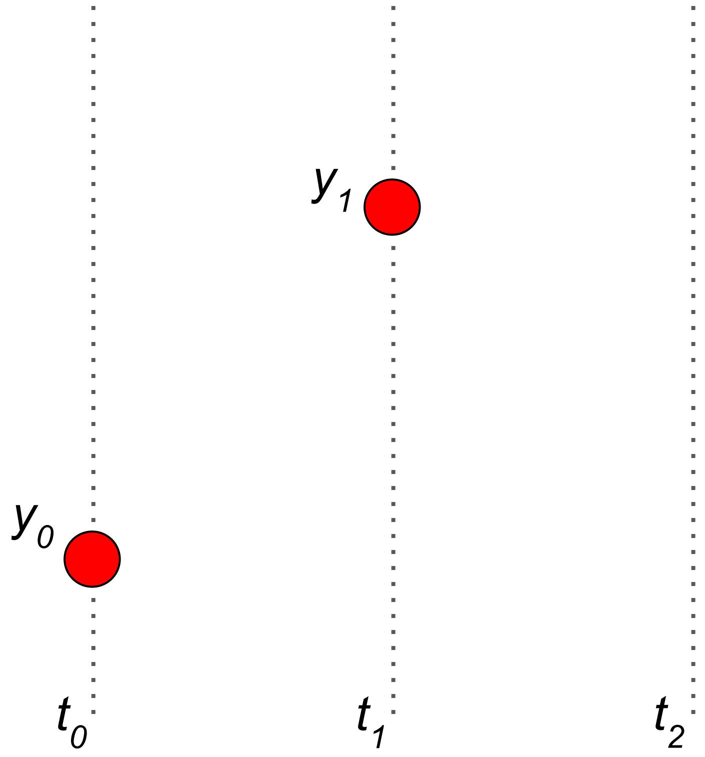
$$\boxed{\frac{dy}{dt} = f(t, y)}$$

Multi-step methods

Basic idea

Predictor-corrector methods store the solution along the way, and use those results to extrapolate the solution for the following step. Then, these methods introduce a correction for the extrapolation by using the derivative information at the new point. These are best for very smooth functions.

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

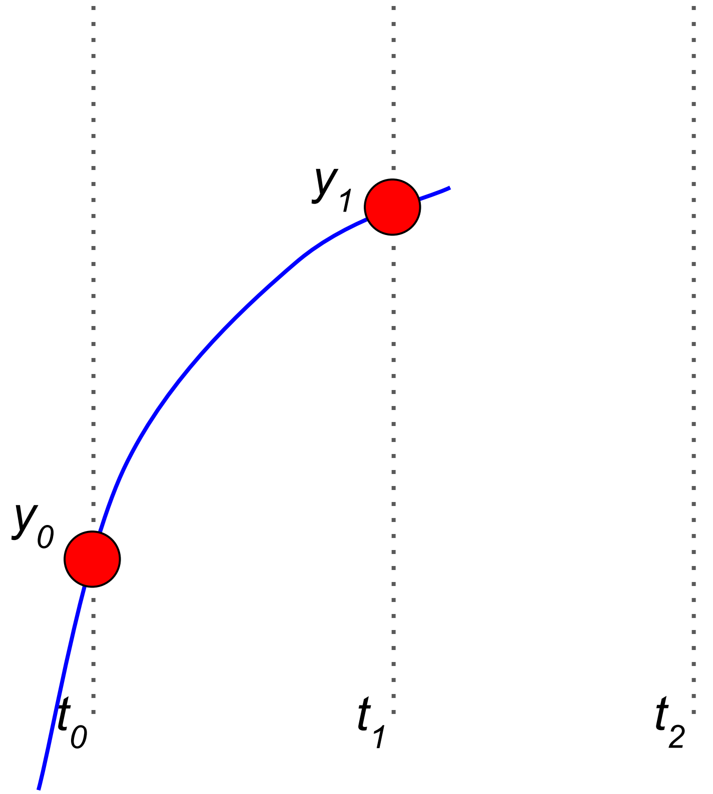


Multi-step methods

Basic idea

Predictor-corrector methods store the solution along the way, and use those results to extrapolate the solution for the following step. Then, these methods introduce a correction for the extrapolation by using the derivative information at the new point. These are best for very smooth functions.

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

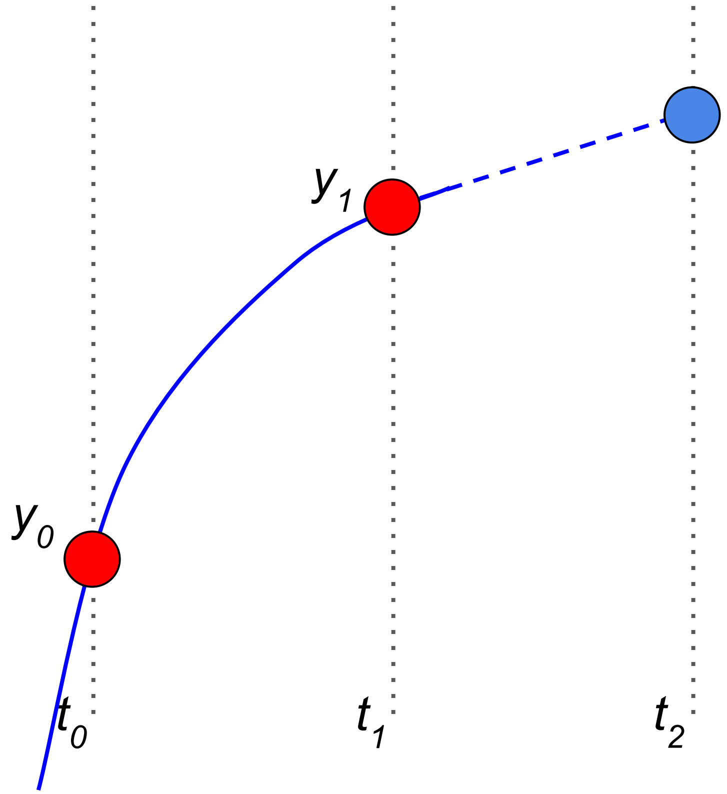


Multi-step methods

Basic idea

Predictor-corrector methods store the solution along the way, and use those results to extrapolate the solution for the following step. Then, these methods introduce a correction for the extrapolation by using the derivative information at the new point. These are best for very smooth functions.

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

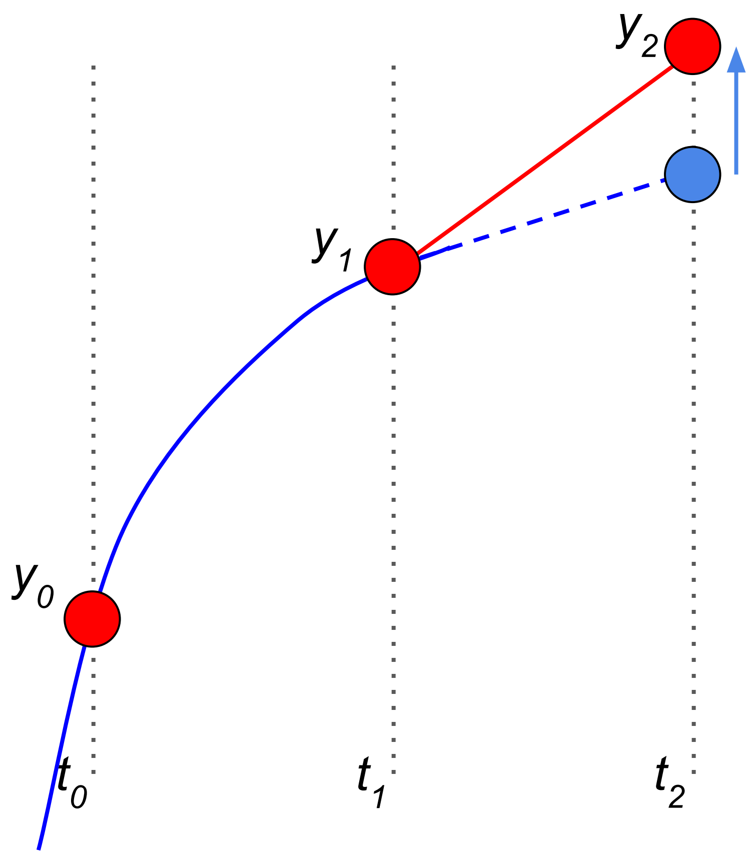


Multi-step methods

Basic idea

Predictor-corrector methods store the solution along the way, and use those results to extrapolate the solution for the following step. Then, these methods introduce a correction for the extrapolation by using the derivative information at the new point. These are best for very smooth functions.

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



Multi-step methods

Predictor-corrector method

Multi-step methods

Predictor-corrector method

To advance the solution of *y' = f(t, y)* from t_n to t_{n+1}, we have

$$dy = f(t, y)\, dt \longrightarrow$$

Multi-step methods

Predictor-corrector method

To advance the solution of *y’* = *f(t, y)* from t_n to t_{n+1}, we have

$$dy = f(t, y)\,dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t', y)dt' \qquad (1)$$

Multi-step methods

Predictor-corrector method

To advance the solution of $y' = f(t, y)$ from t_n to t_{n+1}, we have

$$dy = f(t,y)\,dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t',y)dt' \qquad (1)$$

In a multi-step method, we approximate $f(t, y)$ by a polynomial passing through several previous points t_n, t_{n-1}, ... and possibly also t_{n+1}. The result of evaluating the integral above at $t = t_{n+1}$ is then of the form

Multi-step methods

Predictor-corrector method

To advance the solution of $y' = f(t, y)$ from t_n to t_{n+1}, we have

$$dy = f(t,y)\,dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t',y)dt' \qquad (1)$$

In a multi-step method, we approximate $f(t, y)$ by a polynomial passing through several previous points t_n, t_{n-1}, ... and possibly also t_{n+1}. The result of evaluating the integral above at $t = t_{n+1}$ is then of the form

$$y'_{n+1} = y_n + h\,(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \ldots) \qquad (2)$$

Multi-step methods

Predictor-corrector method

To advance the solution of *y' = f(t, y)* from t_n to t_{n+1}, we have

$$dy = f(t, y)\, dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t', y) dt' \qquad (1)$$

In a multi-step method, we approximate *f(t, y)* by a polynomial passing through several previous points t_n, t_{n-1}, ... and possibly also t_{n+1}. The result of evaluating the integral above at $t = t_{n+1}$ is then of the form

$$y'_{n+1} = y_n + h\,(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \ldots) \qquad (2)$$

$$y'_n = f(t_n, y_n)$$

Multi-step methods

Predictor-corrector method

To advance the solution of *y' = f(t, y)* from t_n to t_{n+1}, we have

$$dy = f(t, y)\, dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t', y) dt' \qquad (1)$$

In a multi-step method, we approximate *f(t, y)* by a polynomial passing through several previous points t_n, t_{n-1}, ... and possibly also t_{n+1}. The result of evaluating the integral above at $t = t_{n+1}$ is then of the form

$$y'_{n+1} = y_n + h\,(\beta_0 y'_{n+1} + \beta_1 \underbrace{y'_n}_{y'_n = f(t_n, y_n)} + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \ldots) \qquad (2)$$

The order of this method is determined by the number of previous steps that are used to get each new value of *y*.

Multi-step methods

Predictor-corrector method

To advance the solution of *y' = f(t, y)* from t_n to t_{n+1}, we have

$$dy = f(t,y)\,dt \longrightarrow y(t) = y_n + \int_{t_n}^{t_{n+1}} f(t',y)dt' \qquad (1)$$

In a multi-step method, we approximate *f(t, y)* by a polynomial passing through several previous points t_n, t_{n-1}, ... and possibly also t_{n+1}. The result of evaluating the integral above at $t = t_{n+1}$ is then of the form

$$y'_{n+1} = y_n + h\,(\beta_0 \boxed{y'_{n+1}} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \ldots) \qquad (2)$$

$$y'_n = f(t_n, y_n)$$

The order of this method is determined by the number of previous steps that are used to get each new value of *y*.

Multi-step methods

Predictor-corrector method

In practice, how do we do this? We need to make some initial guess for y_{n+1}, insert it into the right-hand side of (2) to get an updated value of y_{n+1}, then insert this value back into the right-hand side, and continue iterating.

Multi-step methods

Predictor-corrector method
In practice, how do we do this? We need to make some initial guess for y_{n+1}, insert it into the right-hand side of (2) to get an updated value of y_{n+1}, then insert this value back into the right-hand side, and continue iterating.

- To get an initial guess for y_{n+1} we use an explicit formula of the same form as (2). This is called the **predictor step**. Here we essentially extrapolate the polynomial fit to the derivative from the previous points to the new point t_{n+1} and then compute the integral (1) in a Simpson-like manner from t_n to t_{n+1}.

Multi-step methods

Predictor-corrector method

In practice, how do we do this? We need to make some initial guess for y_{n+1}, insert it into the right-hand side of (2) to get an updated value of y_{n+1}, then insert this value back into the right-hand side, and continue iterating.

- To get an initial guess for y_{n+1} we use an explicit formula of the same form as (2). This is called the **predictor step**. Here we essentially extrapolate the polynomial fit to the derivative from the previous points to the new point t_{n+1} and then compute the integral (1) in a Simpson-like manner from t_n to t_{n+1}.
- The subsequent Simpson-like integration, using the prediction step's value of y_{n+1} to interpolate the derivative, is called the **corrector step**.

Multi-step methods

Predictor-corrector method
In practice, how do we do this? We need to make some initial guess for y_{n+1}, insert it into the right-hand side of (2) to get an updated value of y_{n+1}, then insert this value back into the right-hand side, and continue iterating.

- To get an initial guess for y_{n+1} we use an explicit formula of the same form as (2). This is called the **predictor step**. Here we essentially extrapolate the polynomial fit to the derivative from the previous points to the new point t_{n+1} and then compute the integral (1) in a Simpson-like manner from t_n to t_{n+1}.
- The subsequent Simpson-like integration, using the prediction step's value of y_{n+1} to interpolate the derivative, is called the **corrector step**.
- The difference between the predicted and corrected function values supplies information on the **local truncation error** that can be used to control accuracy and to adjust stepsize.

Multi-step methods

Adams-Bashforth-Moulton schemes
The most popular predictor-corrector methods with good stability properties.

Multi-step methods

Adams-Bashforth-Moulton schemes
The most popular predictor-corrector methods with good stability properties.

- The **Adams-Bashforth** part is the **predictor** - here is the third order case:

$$y_{n+1} = y_n + \frac{\Delta t}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(\Delta t^4)$$

Multi-step methods

Adams-Bashforth-Moulton schemes

The most popular predictor-corrector methods with good stability properties.

- The **Adams-Bashforth** part is the **predictor** - here is the third order case:

$$y_{n+1} = y_n + \frac{\Delta t}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(\Delta t^4)$$

  Here the information at the current point t_n, together with the two previous points t_{n-1} and t_{n-2} (assumed equally spaced), is used to predict the value y_{n+1} at the next point, t_{n+1}.

Multi-step methods

Adams-Bashforth-Moulton schemes

The most popular predictor-corrector methods with good stability properties.

- The **Adams-Bashforth** part is the **predictor** - here is the third order case:

  $$y_{n+1} = y_n + \frac{\Delta t}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(\Delta t^4)$$

  Here the information at the current point t_n, together with the two previous points t_{n-1} and t_{n-2} (assumed equally spaced), is used to predict the value y_{n+1} at the next point, t_{n+1}.
- **Adams-Moulton** part is the **corrector**, which in the third order case is:

  $$y_{n+1} = y_n + \frac{\Delta t}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) + O(\Delta t^4)$$

  Without the trial value of y_{n+1} from the predictor step to insert on the right-hand side, the corrector would be a nasty implicit equation for y_{n+1}.

Multi-step methods

Predictor-corrector method

There are actually three separate processes in a predictor-corrector method:

Multi-step methods

Predictor-corrector method

There are actually three separate processes in a predictor-corrector method:

- *P*, the predictor step
- *E*, the evaluation of the derivative y_{n+1} from the latest value of y
- *C*, the corrector step

Multi-step methods

Predictor-corrector method

There are actually three separate processes in a predictor-corrector method:

- *P*, the predictor step
- *E*, the evaluation of the derivative y_{n+1} from the latest value of y
- *C*, the corrector step

So, iterating m times with the corrector (a practice which is NOT recommended) would be $P(EC)^m$. We can also choose if we want to finish with a *C* or *E* step. A final *E* is superior: the strategy usually recommended is therefore *PECE*.

Multi-step methods

Predictor-corrector method

There are actually three separate processes in a predictor-corrector method:

- *P*, the predictor step
- *E*, the evaluation of the derivative y_{n+1} from the latest value of *y*
- *C*, the corrector step

So, iterating *m* times with the corrector (a practice which is NOT recommended) would be $P(EC)^m$. We can also choose if we want to finish with a *C* or *E* step. A final *E* is superior: the strategy usually recommended is therefore *PECE*.

Multistep methods suffer from two serious difficulties:

Multi-step methods

Predictor-corrector method

There are actually three separate processes in a predictor-corrector method:

- *P*, the predictor step
- *E*, the evaluation of the derivative y_{n+1} from the latest value of y
- *C*, the corrector step

So, iterating *m* times with the corrector (a practice which is NOT recommended) would be $P(EC)^m$. We can also choose if we want to finish with a *C* or *E* step. A final *E* is superior: the strategy usually recommended is therefore *PECE*.

Multistep methods suffer from two serious difficulties:

- the formulas require equally spaced steps, so adjusting stepsize is difficult
- starting and stopping also present problems (for starting, we need initial values and several previous steps; stopping is a problem because equal steps are unlikely to land directly on the desired termination point)

Sources of errors

Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**.

Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**. The accumulation of these errors (error propagation) is known as **global truncation error**, and is an important quantity to keep under control in the calculation.

Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**. The accumulation of these errors (error propagation) is known as **global truncation error**, and is an important quantity to keep under control in the calculation.

There are other sources of error in addition to the truncation error.

Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**. The accumulation of these errors (error propagation) is known as **global truncation error**, and is an important quantity to keep under control in the calculation.

There are other sources of error in addition to the truncation error.

- **Original data errors**: if the initial conditions are not known exactly or are expressed inexactly, the solution will be affected to a greater/lesser degree depending on the sensitivity of the equation.

Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**. The accumulation of these errors (error propagation) is known as **global truncation error**, and is an important quantity to keep under control in the calculation.

There are other sources of error in addition to the truncation error.

- **Original data errors**: if the initial conditions are not known exactly or are expressed inexactly, the solution will be affected to a greater/lesser degree depending on the sensitivity of the equation.
- **Round-off errors**: arising from floating-point and fixed-point calculations.

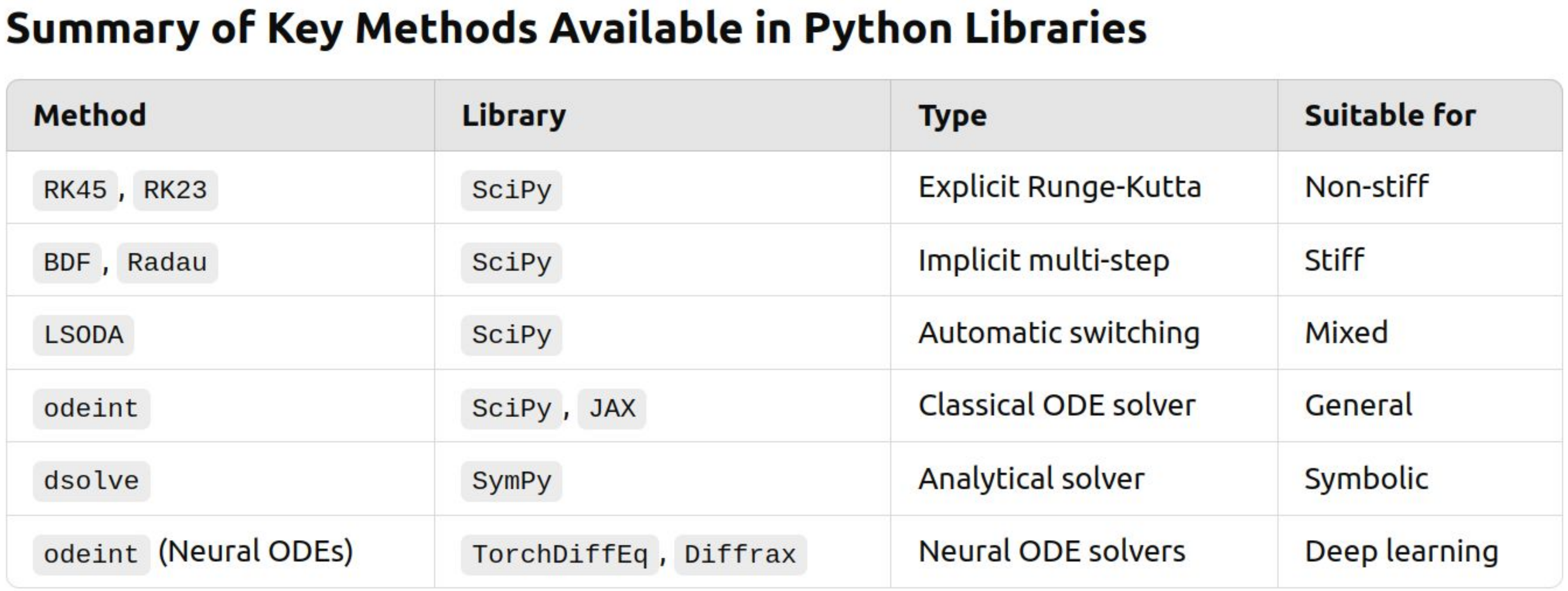
Sources of errors

The errors that we mentioned so far are the ones relative to a single step in the solution, and are called **local truncation errors**. The accumulation of these errors (error propagation) is known as **global truncation error**, and is an important quantity to keep under control in the calculation.

There are other sources of error in addition to the truncation error.

- **Original data errors**: if the initial conditions are not known exactly or are expressed inexactly, the solution will be affected to a greater/lesser degree depending on the sensitivity of the equation.
- **Round-off errors**: arising from floating-point and fixed-point calculations.
- **Truncation errors of the method**: due to the use of truncated series for approximation, when infinite series is needed for exactness.

Implementation in Python



Summary of Key Methods Available in Python Libraries

Method	Library	Type	Suitable for
`RK45`, `RK23`	`SciPy`	Explicit Runge-Kutta	Non-stiff
`BDF`, `Radau`	`SciPy`	Implicit multi-step	Stiff
`LSODA`	`SciPy`	Automatic switching	Mixed
`odeint`	`SciPy`, `JAX`	Classical ODE solver	General
`dsolve`	`SymPy`	Analytical solver	Symbolic
`odeint` (Neural ODEs)	`TorchDiffEq`, `Diffrax`	Neural ODE solvers	Deep learning

END OF LECTURE 1

Previously, in ODEs part I...

We want to get the next step of the function y_{n+1} based on y_n:

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt} \right)}$$

Question 1: how to calculate the *dy/dt* term? There are many schemes, and they differ mostly in how they deal with this term.
Question 2: how to calculate the timestep length?

Assumption: we know that

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

Systems of ODEs. Stiffness

Stiffness has no exact mathematical definition.

A stiff differential equation is one that requires very small timesteps when integrating numerically for the solution to be accurate and stable.

A stiff system of ODEs is one where the timescales of change of the various quantities are very different, making it very difficult to intregrate numerically with simple techniques.

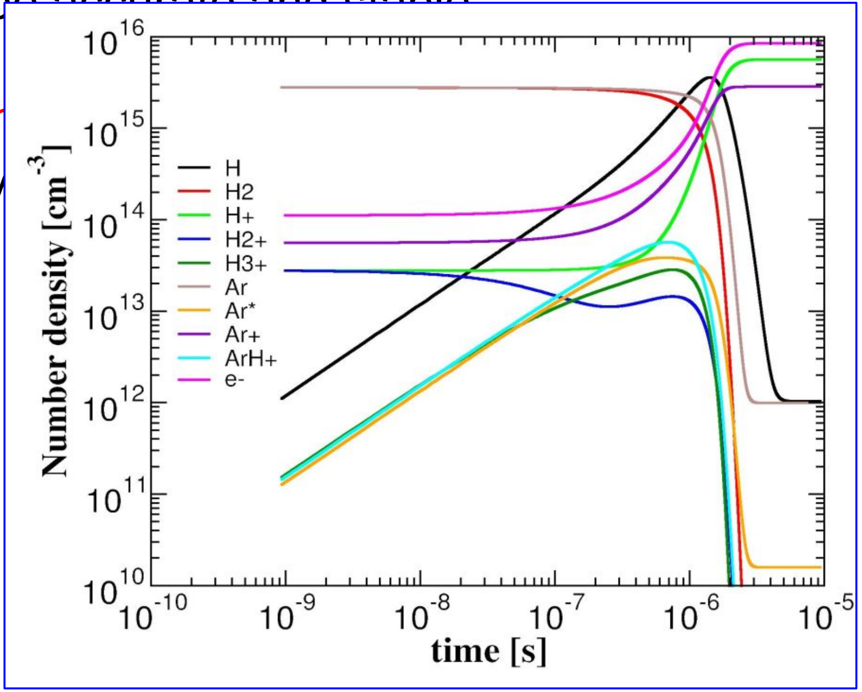
Systems of ODEs. Stiffness

Stiffness has no exact mathematical definition.

A stiff differential equation is one that requires very small timesteps when integrating numerically for the solution to b

A stiff system of ODEs is one where the ti
quantities are very different, making it very
simple techniques.

e.g. chemical kinetics
Large networks of chemical reactions often make stiff systems of ODEs because of very different reaction rates requiring different timesteps



Explicit VS implicit schemes

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

Schemes for the numerical solutions of differential equations can be usefully broken down into ***explicit*** and ***implicit*** schemes.

Explicit schemes calculate gradient term, *dy/dt*, based entirely on information at t_n or earlier timesteps (one step vs multi-steps methods).

Implicit schemes use information about the state of the dependent variable at the end of the timestep, i.e. y_{n+1}, to calculate the gradient.

Explicit VS implicit schemes

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

Schemes for the numerical solutions of differential equations can be usefully broken down into ***explicit*** and ***implicit*** schemes.

Explicit schemes calculate gradient term, *dy/dt*, based entirely on information at t_n or earlier timesteps (one step vs multi-steps methods).

Implicit schemes use information about the state of the dependent variable at the end of the timestep, i.e. y_{n+1}, to calculate the gradient.

Example:

Forward Euler scheme: $y_{n+1} = y_n + \Delta t f(t_n, y_n)$

Backward Euler scheme: $y_{n+1} = y_n + \Delta t f(t_{n+1}, y_{n+1})$

Explicit VS implicit schemes

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

It might seem strange to use the value of y_{n+1} to to calculate the gradient since we need to know the gradient in order to get y_{n+1}. Here is a simple example for how it can work:

Assumption:

$$y_{n+1} = y_n + \Delta t f(t_{n+1}, y_{n+1})$$

Explicit VS implicit schemes

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

It might seem strange to use the value of y_{n+1} to to calculate the gradient since we need to know the gradient in order to get y_{n+1}. Here is a simple example for how it can work:

Assumption:

$$y_{n+1} = y_n + \Delta t f(t_{n+1}, y_{n+1})$$

Gradient given by:

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

Explicit VS implicit schemes

$$\boxed{y_{n+1} = y_n + \Delta t \left(\frac{dy}{dt}\right)}$$

It might seem strange to use the value of y_{n+1} to to calculate the gradient since we need to know the gradient in order to get y_{n+1}. Here is a simple example for how it can work:

Assumption: $$y_{n+1} = y_n + \Delta t f(t_{n+1}, y_{n+1})$$

Gradient given by: $$\frac{dy}{dt} = y$$

The update can then be calculated as: $$y_{n+1} = y_n + \Delta t y_{n+1}$$

$$\boxed{y_{n+1} = \frac{y_n}{1 - \Delta t}}$$

Backward Euler

The previous example is very specific, and in most cases, it is not possible to simply rearrange the equation and get the equation for y_{n+1}. We instead need a general method that can be applied to arbitrary ODEs and systems of ODEs.

Most common method: **Newton iteration**

Backward Euler

The previous example is very specific, and in most cases, it is not possible to simply rearrange the equation and get the equation for y_{n+1}. We instead need a general method that can be applied to arbitrary ODEs and systems of ODEs.

Most common method: **Newton iteration**

The method is used to solve the equations of type $\boxed{G(x) = 0}$

Step 1: make first estimate of x
Step 2: iteratively improve estimate of x from

$$\boxed{x_{(m+1)} = x_{(m)} - \frac{G(x_{(m)})}{G'(x_{(m)})}}$$

Step 3: stop iterating when reached convergence

Backward Euler

How can Newton iteration be useful for Backward Euler method?

We need equation in the form: $\boxed{G(x) = 0}$

The Backward Euler method is given by

$$\boxed{y_{n+1} = y_n + \Delta t\, f_{n+1}}$$

Backward Euler

How can Newton iteration be useful for Backward Euler method?

We need equation in the form: $\boxed{G(x) = 0}$

The Backward Euler method is given by

$$\boxed{y_{n+1} = y_n + \Delta t\, f_{n+1}}$$

This can be rearranged to give the equation to solve

$$\boxed{G(y_{n+1}) = y_{n+1} - y_n - \Delta t\, f_{n+1} = 0}$$

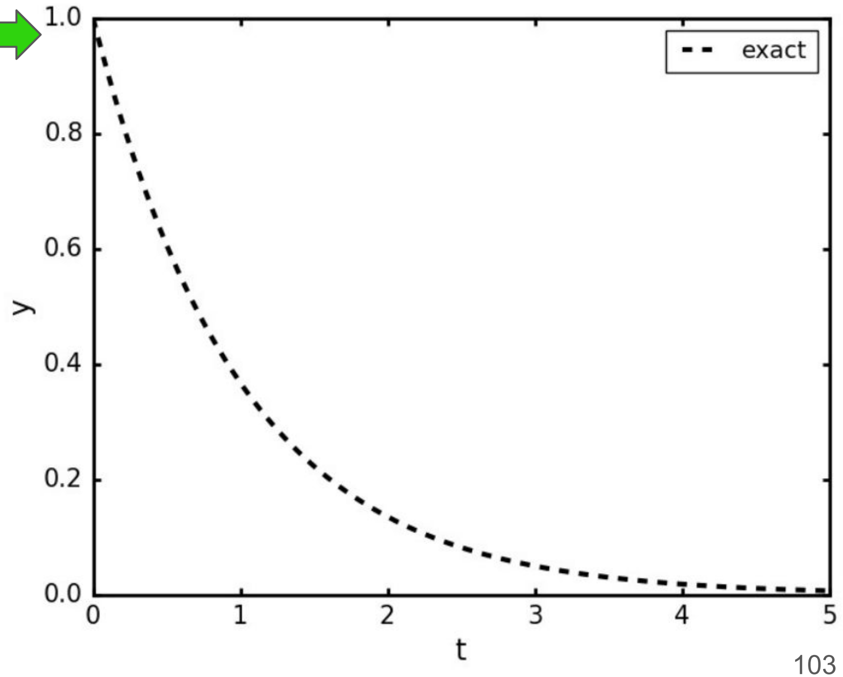
For Newton Iteration, we also need the *G'* term

$$\boxed{G'(y_{n+1}) = \frac{dG(y_{n+1})}{dy_{n+1}} = 1 - \Delta t\, f'_{n+1}}$$

Backward Euler: example

Initial value problem: $y(t=0)=1 \qquad \frac{dy}{dt}=-y$

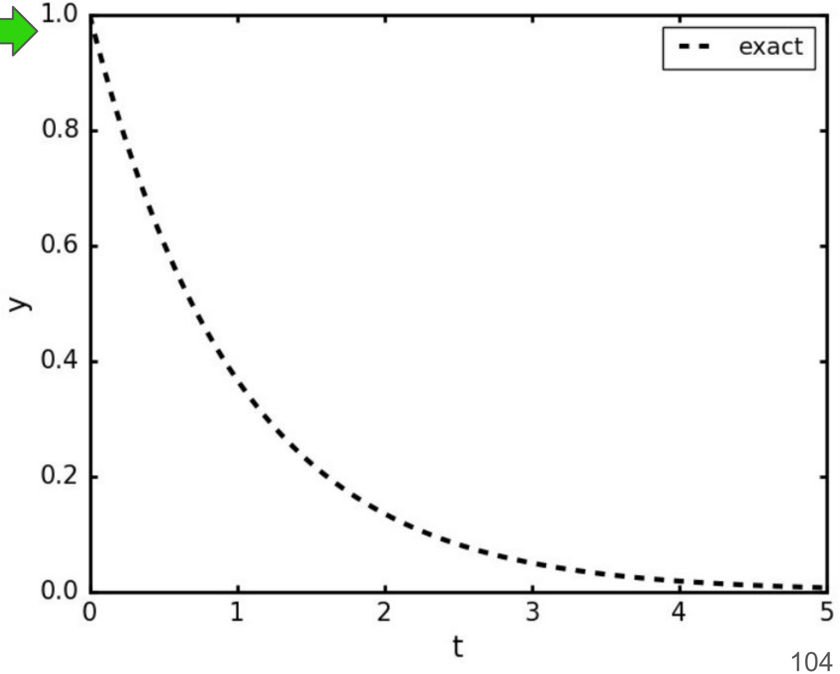
The analytic solution is $y(t)=\mathrm{e}^{-t}$



Backward Euler: example

Initial value problem: $$y(t=0)=1 \qquad \frac{dy}{dt}=-y$$

The analytic solution is $y(t)=\mathrm{e}^{-t}$



Several definitions:
(here, *m* is the index of the iteration)

$$G_{n+1}^{(m)}=G(y_{n+1}^{(m)})$$
$$f_{n+1}^{(m)}=f(y_{n+1}^{(m)})$$

$$f'_{n+1}{}^{(m)}=\frac{df_{n+1}^{(m)}}{dy_{n+1}^{(m)}} \qquad G'_{n+1}{}^{(m)}=\frac{dG_{n+1}^{(m)}}{dy_{n+1}^{(m)}}$$

Backward Euler: example

The basic equation for the improvement in the update of *y* is

$$\boxed{y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \frac{G_{n+1}^{(m)}}{{G'}_{n+1}^{(m)}}}$$

Backward Euler: example

The basic equation for the improvement in the update of *y* is

$$\boxed{y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \frac{G_{n+1}^{(m)}}{{G'}_{n+1}^{(m)}}}$$

Where the *G* function are given by

$$\boxed{\begin{aligned} G_{n+1}^{(m)} &= y_{n+1}^{(m)} - y_n - \Delta t f_{n+1}^{(m)} \\ {G'}_{n+1}^{(m)} &= 1 - \Delta t {f'}_{n+1}^{(m)} \end{aligned}}$$

Backward Euler: example

The basic equation for the improvement in the update of *y* is

$$\boxed{y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \frac{G_{n+1}^{(m)}}{{G'_{n+1}}^{(m)}}}$$

Where the *G* function are given by

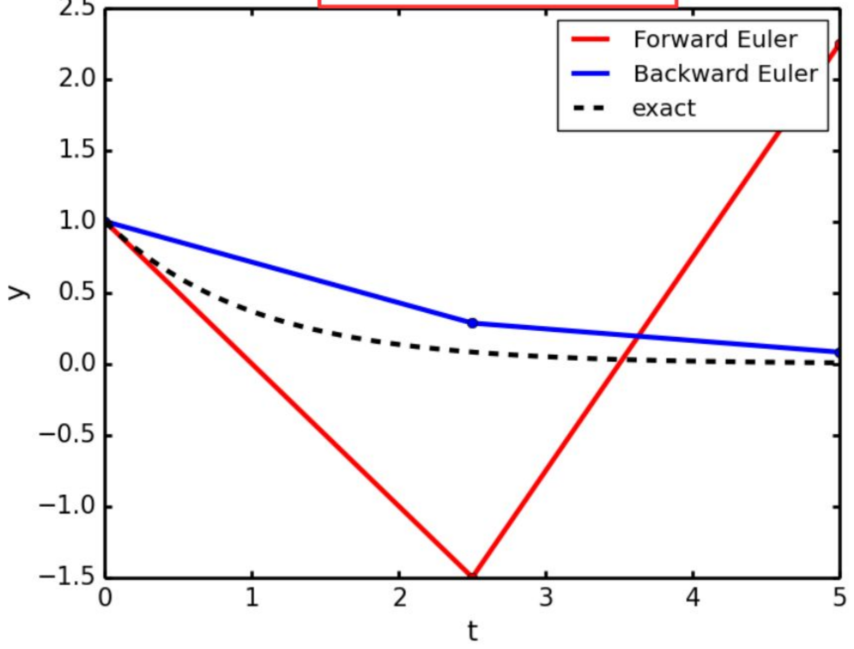
$$\boxed{\begin{aligned} G_{n+1}^{(m)} &= y_{n+1}^{(m)} - y_n - \Delta t f_{n+1}^{(m)} \\ {G'_{n+1}}^{(m)} &= 1 - \Delta t {f'_{n+1}}^{(m)} \end{aligned}}$$

For this problem the *f* functions can be easily derived:

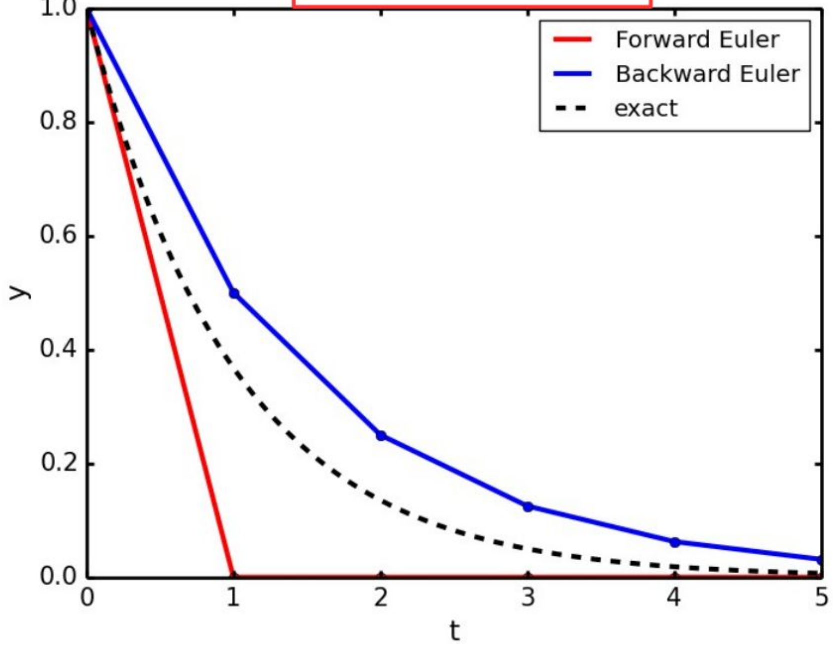
$$\boxed{\begin{aligned} f_{n+1}^{(m)} &= -y_{n+1}^{(m)} \\ {f'_{n+1}}^{(m)} &= -1 \end{aligned}}$$

Backward Euler: example

2 STEPS

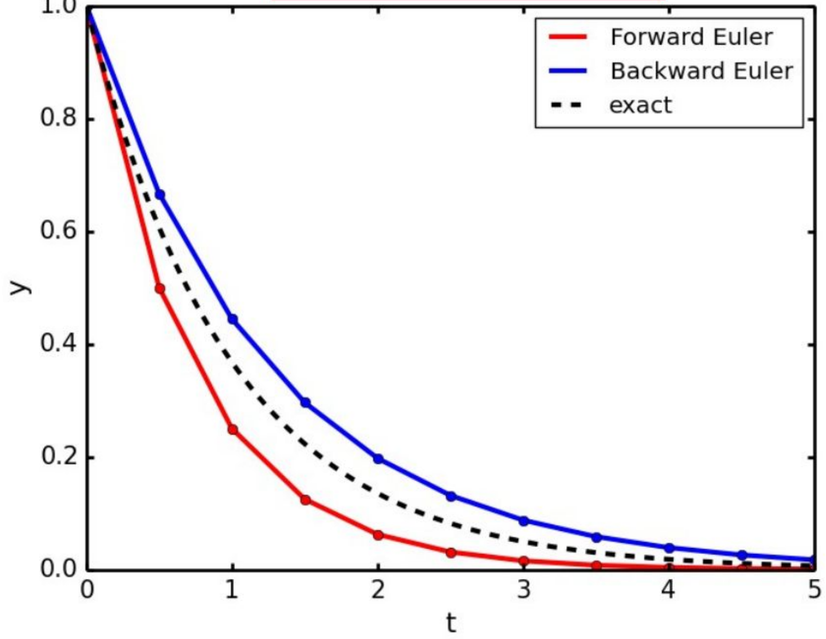


5 STEPS

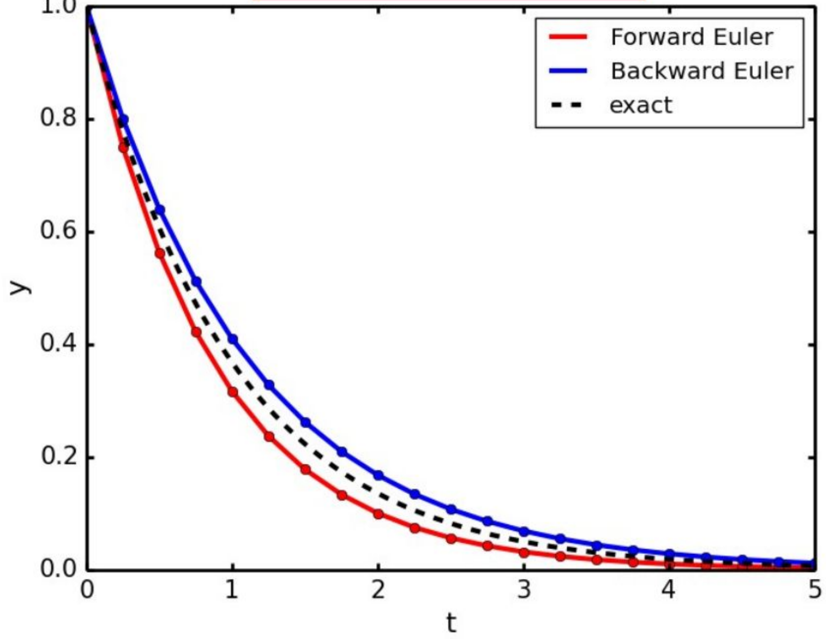


Backward Euler: example

10 STEPS



20 STEPS



Implicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

Implicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

Here, k_i are intermediate stage values that satisfy the equations:

$$k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, \ldots, s.$$

The values of a_{ij}, b_i, and c_i define the specific implicit Runge-Kutta method being used.

Implicit Runge-Kutta methods

$$\boxed{\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0}$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$\boxed{y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i}$$

Here, k_i are intermediate stage values that satisfy the equations:

the number of stages in the method

$$\boxed{k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, \ldots, s.}$$

The values of a_{ij}, b_i, and c_i define the specific implicit Runge-Kutta method being used.

Implicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

Here, k_i are intermediate stage v uations:

A = $\{a_{ij}\}$ is the Butcher tableau matrix

the number of stages in the method

$$k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1} a_{ij} k_j\right), \quad i = 1, \ldots, s.$$

The values of a_{ij}, b_i, and c_i define the specific implicit Runge-Kutta method being used.

Implicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

Weights

Here, k_i are intermediate stage values obtained by solving the equations:

A = {a_{ij}} is the Butcher tableau matrix

the number of stages in the method

$$k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1} a_{ij} k_j\right), \quad i = 1, \ldots, s.$$

The values of a_{ij}, b_i, and c_i define the specific implicit Runge-Kutta method being used.

Implicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

Implicit Runge-Kutta methods compute the solution by solving a set of coupled nonlinear equations at each time step. The solution at the next step y_{n+1}:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

Weights

nodes (collocation points) within the time step $c_i \in [0,1]$

tage v

A = {a_{ij}} is the Butcher tableau matrix

uations:

the number of stages in the method

$$k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1} a_{ij} k_j\right), \quad i = 1, \ldots, s.$$

The values of a_{ij}, b_i, and c_i define the specific implicit Runge-Kutta method being used.

Radau IIa method

Uses collocation at Radau points within each time step:

1. The endpoint $c_s=1$
2. $s-1$ other collocation points in (0,1) are chosen as roots of a shifted Legendre polynomial.

For $s = 3$:

$$c_1 = \frac{1}{3}, \quad c_2 = \frac{1}{5}(3+\sqrt{6}), \quad c_3 = 1.$$

Radau IIa method

Uses collocation at Radau points within each time step:

1. The endpoint c_s=*1*
2. *s−1* other collocation points in (0,1) are chosen as roots of a shifted Legendre polynomial.

For *s* = *3*:

$$\boxed{c_1 = \frac{1}{3}, \quad c_2 = \frac{1}{5}(3+\sqrt{6}), \quad c_3 = 1.}$$

Algorithm:

1. Compute the stage values k_i:

$$k_i = f\left(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, 2, \ldots, s.$$

2. Update the solution:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i.$$

Radau IIa method - Butcher tableau

$$\begin{array}{c|ccc}
c_1 & a_{11} & a_{12} & a_{13} \\
c_2 & a_{21} & a_{22} & a_{23} \\
c_3 & a_{31} & a_{32} & a_{33} \\
\hline
 & b_1 & b_2 & b_3
\end{array}$$

Radau IIa method - Butcher tableau

$$
\begin{array}{c|ccc}
\frac{1}{3} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\
\frac{1}{5}(3+\sqrt{6}) & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\
1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\
\hline
 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9}
\end{array}
$$

What makes it implicit

An explicit method computes the values k_i directly, without requiring them as inputs:

$$\boxed{k_i = f\left(t_n + c_i\Delta t, y_n + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right)}$$

EXPLICIT: only $j < i$

In the Radau method, the equations for k_i involve all values, including k_i itself

$$k_1, k_2, \ldots, k_s \text{ are solutions to: } \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_s \end{bmatrix} = \begin{bmatrix} f(t_n + c_1\Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{1j} k_j) \\ f(t_n + c_2\Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{2j} k_j) \\ \vdots \\ f(t_n + c_s\Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{sj} k_j) \end{bmatrix}$$

When to use Backward Euler VS Radau

Backward Euler:

- When simplicity and stability are prioritized over accuracy.
- For rough approximations or when computational resources are very limited.

Radau:

- When solving highly stiff problems where accuracy and efficiency are essential.
- For higher-order accuracy in simulations, especially over long integration times.

Implicit methods are terrible to implement! Why use them?

The Backward Euler method has two major disadvantages to the Forward Euler method:

1. Much more difficult to implement.
2. It takes much longer to perform a timestep.

Implicit methods are terrible to implement! Why use them?

The Backward Euler method has two major disadvantages to the Forward Euler method:

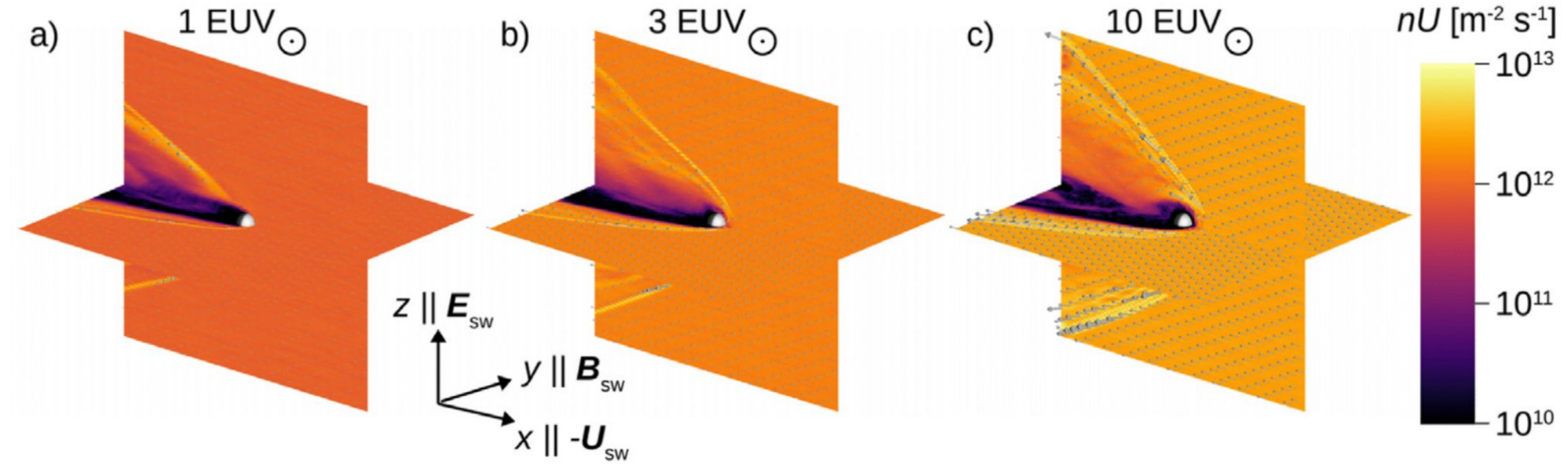
1. Much more difficult to implement.
2. It takes much longer to perform a timestep.

However, it also has a big advantage that can be seen in the previous slides. Notice that even when only two steps were taken, the Backward Euler method game a reasonable result. Implicit methods have two major advantages:

1. They are very stable, allowing large timesteps
2. They can be very accurate when using large timesteps.

In many situations, the ODEs you are solving require timesteps that are too small to be realistically solved. Therefore, implicit methods must be used

Implicit methods are terrible to implement! Why use them?



Lichtenegger et al. 2022. Ion fluxes around early Mars.
Beware of magnetic fields

Which timestep should I use?

In the schemes described so far, the length of the timestep was specified:

$$\boxed{\Delta t = t_{n+1} - t_n}$$

Which timestep should I use?

In the schemes described so far, the length of the timestep was specified:

$$\boxed{\Delta t = t_{n+1} - t_n}$$

How large this timestep should be?
This is an important question because making it too large can lead to inaccurate results, but making it too small can lead to very long calculation times. The situation is made more complicated by the fact that the ideal timestep length usually changes during an integration. Often short steps are needed at the start of an integration and longer steps can be taken as the integration progresses.

Which timestep should I use?

In the schemes described so far, the length of the timestep was specified:

$$\boxed{\Delta t = t_{n+1} - t_n}$$

How large this timestep should be?

One (not necessarily good) option is to define a ‘change timescale’ for the quantity being evolved and then make the timestep length a fixed fraction of this timescale. For example

$$\Delta t = a \left| \frac{y_n}{(dy/dt)_n} \right|$$

where *a<1* (typically *a~0.1*)

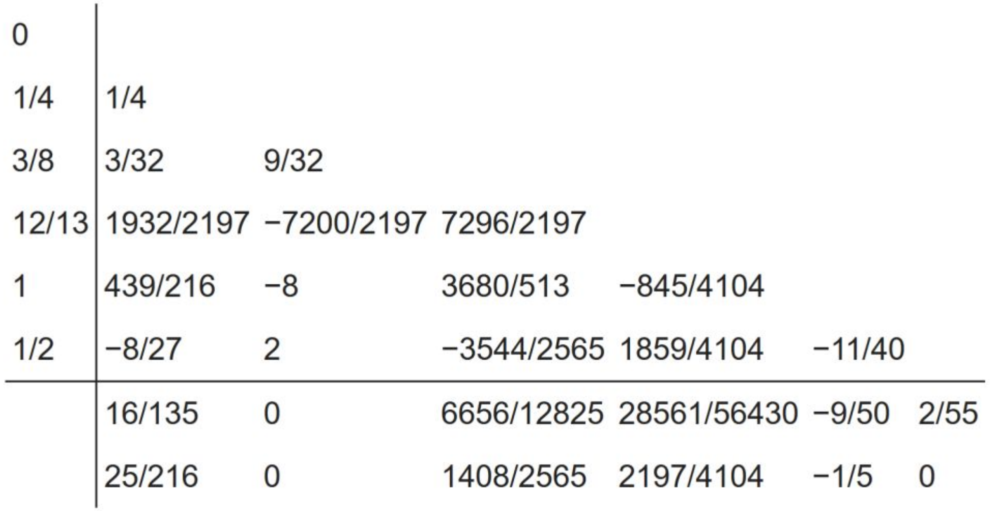
Runge-Kutta-Fehlberg (RKF45)

This is a 5th order scheme that has the very useful property of being able to estimate the timestep length itself. Instead of specifying a timestep length, you specify a tolerance parameter that determines how accurate the scheme should be and it determined the timestep length from that.

Runge-Kutta-Fehlberg (RKF45)

This is a 5th order scheme that has the very useful property of being able to estimate the timestep length itself. Instead of specifying a timestep length, you specify a tolerance parameter that determines how accurate the scheme should be and it determined the timestep length from that.

RK45 is a six step method (s=6) and is given by the following Butcher tableau

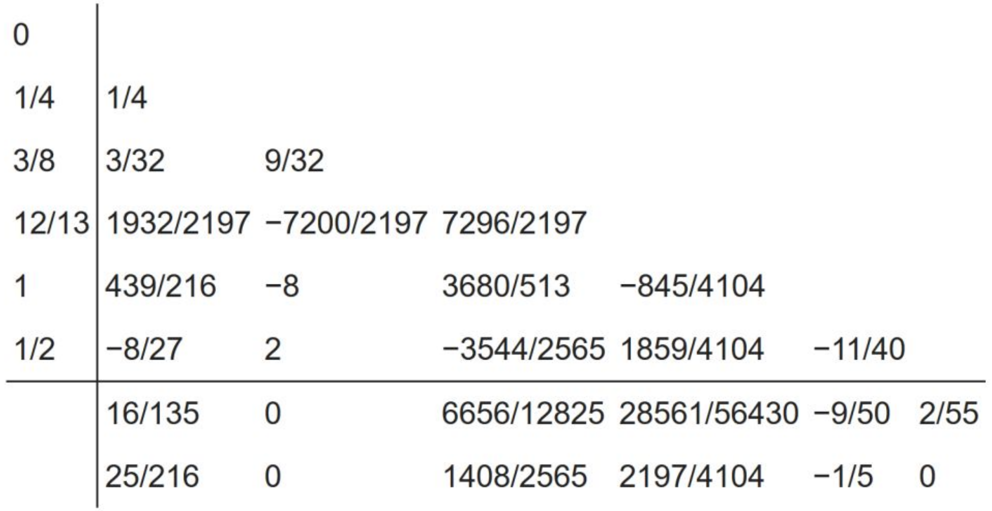


0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
	16/135	0	6656/12825	28561/56430	−9/50	2/55
	25/216	0	1408/2565	2197/4104	−1/5	0

Runge-Kutta-Fehlberg (RKF45)

This is a 5th order scheme that has the very useful property of being able to estimate the timestep length itself. Instead of specifying a timestep length, you specify a tolerance parameter that determines how accurate the scheme should be and it determined the timestep length from that.

RK45 is a six step method (s=6) and is given by the following Butcher tableau



0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
	16/135	0	6656/12825	28561/56430	−9/50	2/55
	25/216	0	1408/2565	2197/4104	−1/5	0

Note. You can tailor these coefficients to your own problem.

Runge-Kutta-Fehlberg (RKF45)

Typically, RKF combines two (the 4th and 5th order) Runge-Kutta schemes.

1. Compute $k_1, \ldots k_6$

Runge-Kutta-Fehlberg (RKF45)

Typically, RKF combines two (the 4th and 5th order) Runge-Kutta schemes.

1. Compute $k_1, \ldots k_6$
2. Calculate the 4th-order estimate $y_{n+1}^{(4)}$ using a weighted sum of k_1 through k_6 with one set of coefficients $b_i^{(4)}$. Calculate the 5th-order estimate $y_{n+1}^{(5)}$ similarly, with a different set of coefficients $b_i^{(5)}$.

Runge-Kutta-Fehlberg (RKF45)

Typically, RKF combines two (the 4th and 5th order) Runge-Kutta schemes.

1. Compute $k_1, \dots k_6$
2. Calculate the 4th-order estimate $y_{n+1}^{(4)}$ using a weighted sum of k_1 through k_6 with one set of coefficients $b_i^{(4)}$. Calculate the 5th-order estimate $y_{n+1}^{(5)}$ similarly, with a different set of coefficients $b_i^{(5)}$.
3. Estimate the error as the difference between the two: $E_n = y_{n+1}^{(5)} - y_{n+1}^{(4)}$

Runge-Kutta-Fehlberg (RKF45)

Typically, RKF combines two (the 4th and 5th order) Runge-Kutta schemes.

1. Compute $k_1, \ldots k_6$
2. Calculate the 4th-order estimate $y_{n+1}^{(4)}$ using a weighted sum of k_1 through k_6 with one set of coefficients $b_i^{(4)}$. Calculate the 5th-order estimate $y_{n+1}^{(5)}$ similarly, with a different set of coefficients $b_i^{(5)}$.
3. Estimate the error as the difference between the two: $E_n = y_{n+1}^{(5)} - y_{n+1}^{(4)}$
4. If E_n is within a specified tolerance ϵ, the step is accepted. Otherwise, the step size is adjusted.

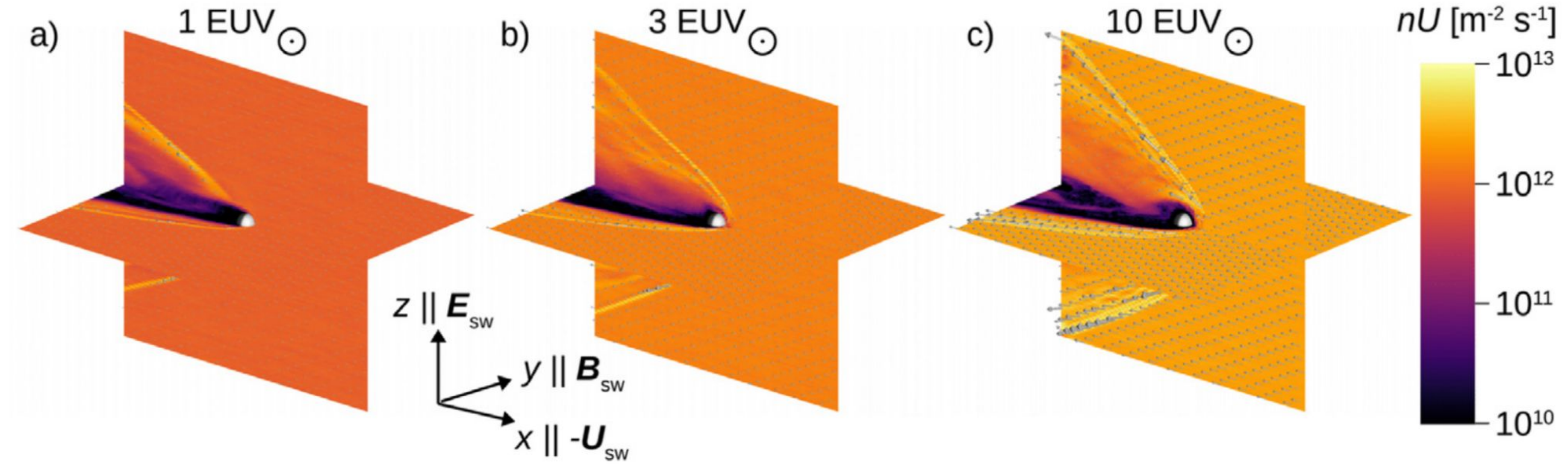
Runge-Kutta-Fehlberg (RKF45)

Typically, RKF combines two (the 4th and 5th order) Runge-Kutta schemes.

1. Compute k_1, … k_6
2. Calculate the 4th-order estimate $y_{n+1}^{(4)}$ using a weighted sum of k_1 through k_6 with one set of coefficients $b_i^{(4)}$. Calculate the 5th-order estimate $y_{n+1}^{(5)}$ similarly, with a different set of coefficients $b_i^{(5)}$.
3. Estimate the error as the difference between the two: $E_n = y_{n+1}^{(5)} - y_{n+1}^{(4)}$
4. If E_n is within a specified tolerance ϵ, the step is accepted. Otherwise, the step size is adjusted.
5. The next step size chosen based on the error to try and keep the error within the tolerance:

$$\boxed{\Delta t_{new} = \Delta t \left(\frac{\epsilon}{|E_n|} \right)^{1/4}}$$

Adaptive step methods are terrible to implement! Why use them?



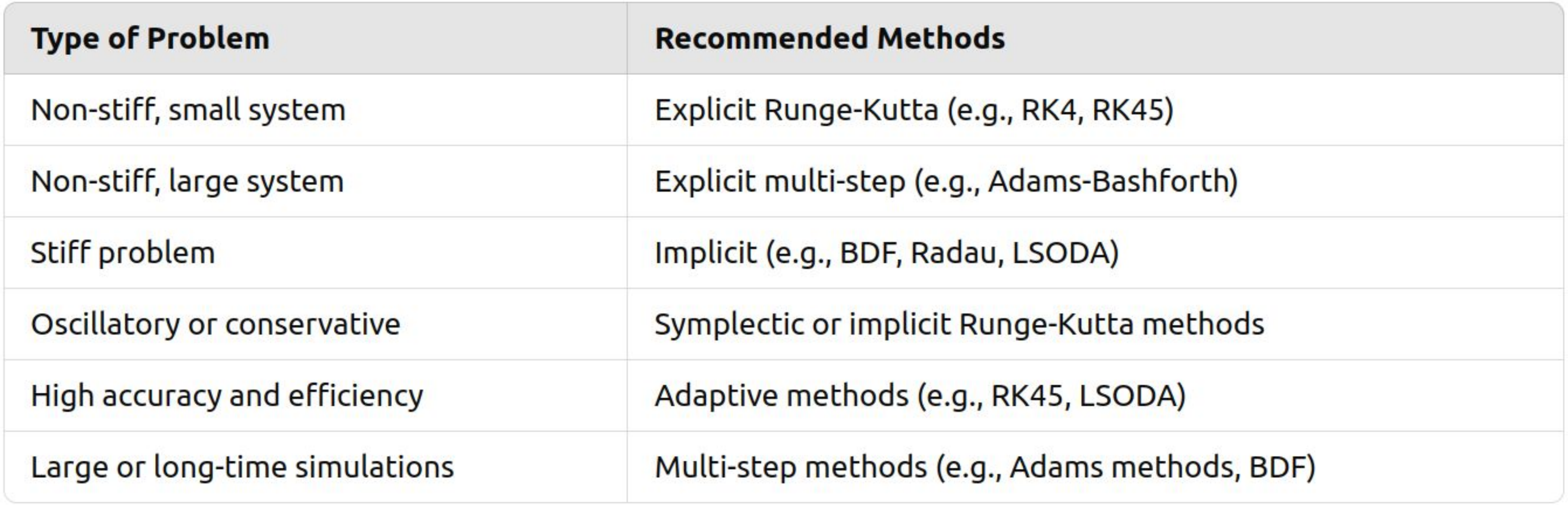
Lichtenegger et al. 2022. Ion fluxes around early Mars.
Beware of magnetic fields!

Summary

There are too many schemes available! How does one choose the best one?

Summary

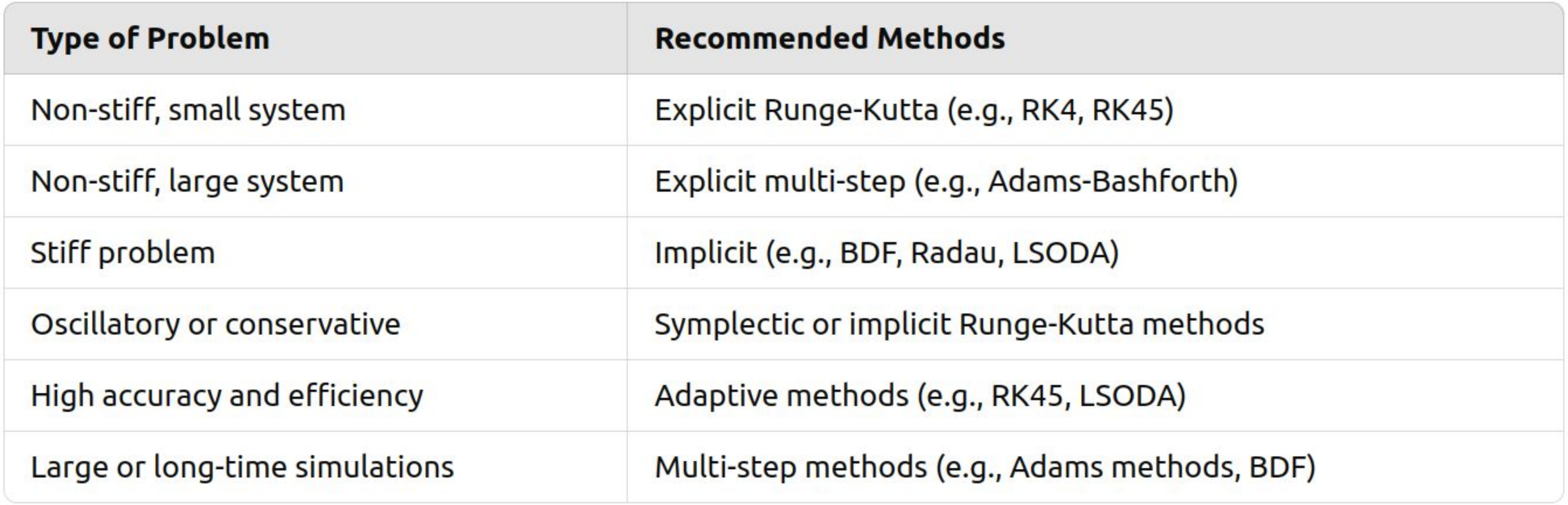
There are too many schemes available! How does one choose the best one?



Type of Problem	Recommended Methods
Non-stiff, small system	Explicit Runge-Kutta (e.g., RK4, RK45)
Non-stiff, large system	Explicit multi-step (e.g., Adams-Bashforth)
Stiff problem	Implicit (e.g., BDF, Radau, LSODA)
Oscillatory or conservative	Symplectic or implicit Runge-Kutta methods
High accuracy and efficiency	Adaptive methods (e.g., RK45, LSODA)
Large or long-time simulations	Multi-step methods (e.g., Adams methods, BDF)

Summary

There are too many schemes available! How does one choose the best one?



Type of Problem	Recommended Methods
Non-stiff, small system	Explicit Runge-Kutta (e.g., RK4, RK45)
Non-stiff, large system	Explicit multi-step (e.g., Adams-Bashforth)
Stiff problem	Implicit (e.g., BDF, Radau, LSODA)
Oscillatory or conservative	Symplectic or implicit Runge-Kutta methods
High accuracy and efficiency	Adaptive methods (e.g., RK45, LSODA)
Large or long-time simulations	Multi-step methods (e.g., Adams methods, BDF)

Unfortunately, often the best method reveals itself only after multiple rounds of trial and error

Implementation in Python

1. SciPy (`scipy.integrate.solve_ivp`)

`SciPy` is one of the most popular libraries for scientific computing in Python and includes several ODE solvers, primarily in the `solve_ivp` function.

- **Runge-Kutta Methods**:
  - `"RK45"`: A popular 5th-order Runge-Kutta method, also known as the Dormand-Prince method.
  - `"RK23"`: A lower-order, 3rd-order Runge-Kutta method, useful when higher precision isn’t needed.
- **Implicit Multistep Methods**:
  - `"Radau"`: An implicit Runge-Kutta method that is highly stable and suited for stiff problems.
  - `"BDF"`: Backward Differentiation Formula, which is a multi-step method effective for stiff problems.
  - `"LSODA"`: An automatic method that switches between non-stiff (Adams) and stiff (BDF) methods as needed.
- **Adams Multistep Method**:
  - `solve_ivp` uses the Adams-Bashforth-Moulton method for non-stiff problems as a part of the `"LSODA"` option.

Implementation in Python

2. SymPy

`SymPy` is a symbolic mathematics library in Python that also provides support for solving ODEs, often analytically if possible. For numerical solutions, `SymPy` relies on `SciPy` under the hood.

- **Analytical Solvers**:
  - `dsolve()`: Attempts to solve ODEs symbolically, returning a closed-form solution when possible.

Implementation in Python

3. JAX

`JAX` is a numerical computing library optimized for high-performance, automatic differentiation, and it provides solvers for ODEs.

- **ODE Solver**:
  - `jax.experimental.ode.odeint`: A flexible, differentiable ODE solver simila to SciPy's `odeint`, typically used for neural differential equations and physics simulations.

Implementation in Python

3. JAX

`JAX` is a numerical computing library optimized for high-performance, automatic differentiation, and it provides solvers for ODEs.

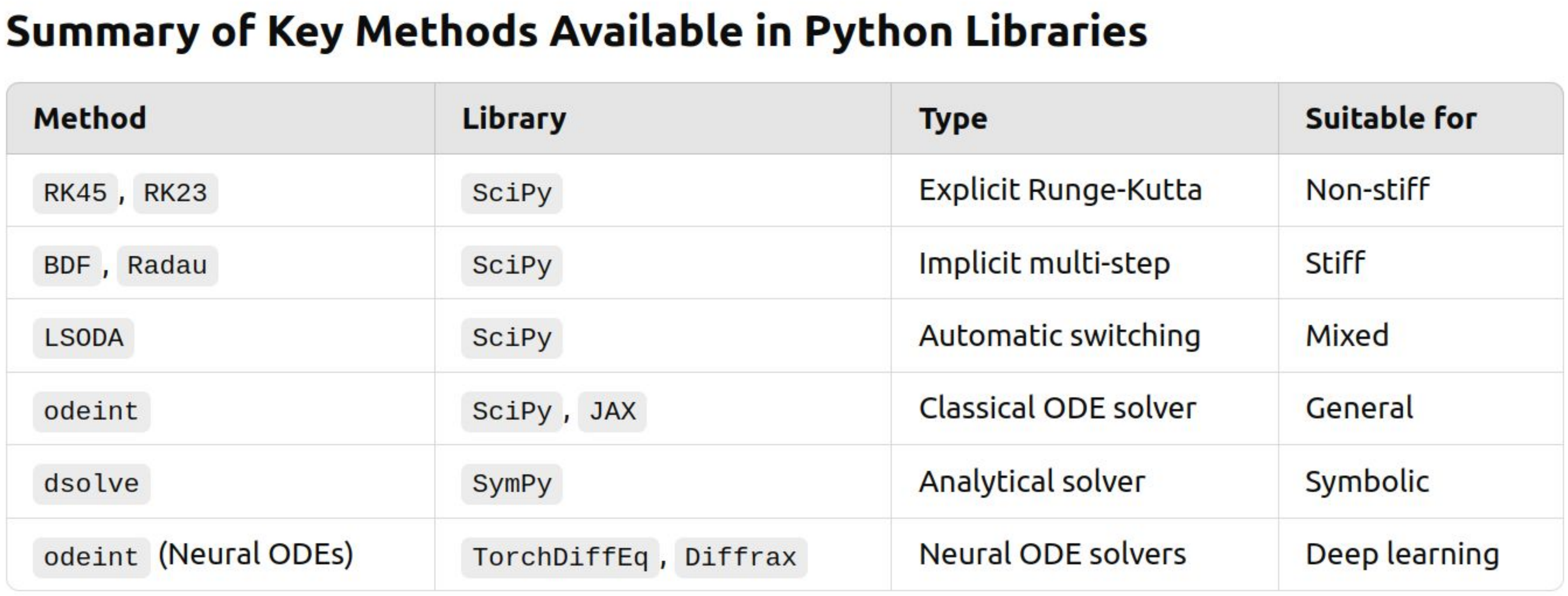
- **ODE Solver**:
  - `jax.experimental.ode.odeint`: A flexible, differentiable ODE solver simila to SciPy's `odeint`, typically used for neural differential equations and physics simulations.

4. TorchDiffEq and Diffrax (for Neural ODEs)

Libraries like `TorchDiffEq` and `Diffrax` are used for neural differential equations, where ODEs are incorporated into neural networks and optimized with automatic differentiation.

- **Neural ODE Solvers**:
  - `TorchDiffEq` and `Diffrax` both implement adaptive solvers similar to `scipy.integrate.solve_ivp`, including Runge-Kutta methods and BDF methods for stiff ODEs.

Implementation in Python



Summary of Key Methods Available in Python Libraries

Method	Library	Type	Suitable for
`RK45`, `RK23`	`SciPy`	Explicit Runge-Kutta	Non-stiff
`BDF`, `Radau`	`SciPy`	Implicit multi-step	Stiff
`LSODA`	`SciPy`	Automatic switching	Mixed
`odeint`	`SciPy`, `JAX`	Classical ODE solver	General
`dsolve`	`SymPy`	Analytical solver	Symbolic
`odeint` (Neural ODEs)	`TorchDiffEq`, `Diffrax`	Neural ODE solvers	Deep learning