Click here to view the PDF version.

```julia
begin
    using Plots
    using PlutoUI
    using PlutoTeachingTools
    using LaTeXStrings
    using DifferentialEquations
    using ForwardDiff
    using HypertextLiteral: @htl, @htl_str
    using LinearAlgebra
end
```

Table of Contents

Initial value problems

In computational science we are often faced with quantitites that change continuously in space or time. For example the temperature profile of a hot body or the population of an animal species over time. Such problems are often modelled by differential equations. If there is only a single indepentent variable t we call the model an **ordinary differential equation**. The usual setup is that for initial value $t = 0$ we have some knowledge about our problem, e.g. by performing a measurement, and the key question is thus how the situation evolves for $t > 0$.

Example: Ducks on the Leman

Suppose we want to model the population of ducks on the Leman, i.e. let $u(t)$ denote the number of ducks on the lake at time t. To simplify the mdoelling we allow for "fractional ducks", i.e. we allow u to be any real number.

First we assume a constant growth rate C for the number of ducks at any time t, i.e. that the number of ducks born minus the number of ducks deceased in one unit of time is proportional to $u(t)$ — the current population of ducks. This leads to the ordinary differential equation (ODE)

$$\begin{cases} \dfrac{du(t)}{dt} = C\,u(t) & t > 0 \\ u(0) = u_0 \end{cases}$$

where u_0 is the initial number of ducks we observed at $t = 0$. In comparison to the general definition shown above we thus have $f(t, u) = C\,u$.

The solution of this linear equation is

$$u(t) = u_0\,e^{Ct},$$

i.e. exponential growth.

Clearly as the size of the Leman is finite, this is not a reasonable model as both food and space on the lake is limited, which should cap the growth. To improve the model we assume that the death rate itself is proportional to the size of the population, i.e. $d\,u$ for a constant $d > 0$. Keeping the idea of a constant birth rate $b > 0$ one thus obtains an improved model

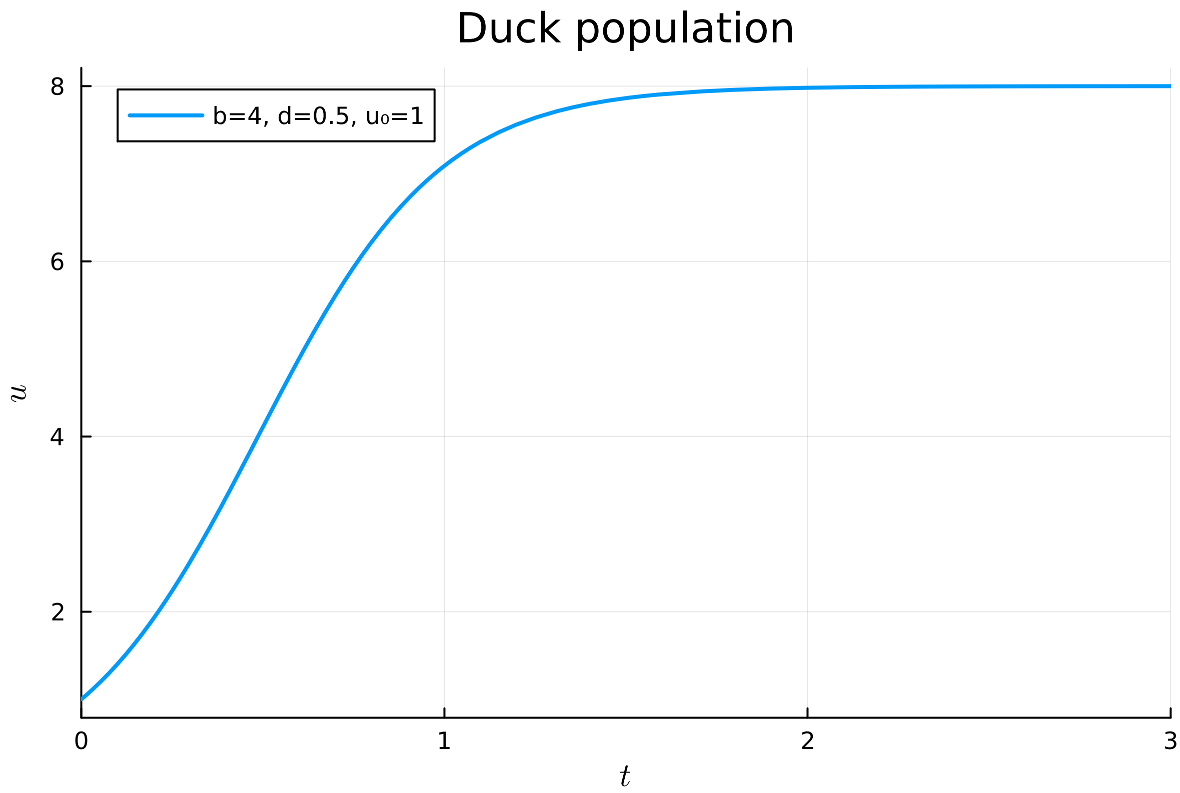
$$\begin{cases} \dfrac{du(t)}{dt} = b\,u(t) - d\,(u(t))^2 & t > 0 \\ u(0) = u_0 \end{cases}$$

i.e. we have the case of $f(u,t) = b\,u - d\,u^2$ when considering the above definition.

This is the **logistic equation**, which in fact has multiple solutions. The solution relevant for population models has the form

$$u(t) = \frac{b/d}{1 + \left(\frac{b}{d\,u_0} - 1\right)e^{-bt}}$$

As a plot shows, this solution varies smoothly from some initial population u_0 to a final population b/d:



```
let
    u₀ = 1
    b = 4
    d = 0.5
    u(t) = b/d / (1 + (b/(d*u₀) - 1) * exp(-b*t))

    plot(u; xlims=(0, 3), label="b=4, d=0.5, u₀=1", xlabel=L"t", ylabel=L"u",
    title="Duck population", lw=2)
end
```

This problem is an example for the a class of problems one calls **initial value problem**, because based on some initial knowledge at $t = 0$ one wants to know how a quantity (e.g. here the population) evolves.

Definition: Initial-value problem

A scalar, first-order initial value problem (IVP) can be formulated as

$$\begin{cases} \dfrac{du(t)}{dt} = f(t, u(t)) \qquad a \leq t \leq b \\ u(a) = u_0, \end{cases} \tag{1}$$

We call t the **indepentent variable**, u the **dependent variable** and u_0 the initial conditions.

A **solution** of an initial-value problem is a differentiable, continuous function $u : \mathbb{R} \to \mathbb{R}$ which makes both $u'(t) = f(t, u(t))$ (for all $a \leq t \leq b$) and $u(a) = u_0$ true equations.

For the specific case where $f(t, u) = g(t) + uh(t)$ for two functions g and h the differential equation (1) is called **linear**.

Often (but not always) t plays the role of time and (1) thus models the time-dependence of a quantity u.

Solving initial value problems numerically

For simple examples like the Duck problem above analytic solutions can still be found with a little practice. However, initial value problems quickly become more involved. For example consider the problem

$$\begin{cases} \dfrac{du(t)}{dt} = \sin\left[(u+t)^2\right] \qquad 0 \leq t \leq 4 \\ u(0) = -1, \end{cases} \tag{2}$$

for which an analytical solution is not so easy to obtain.

Before we introduce a few key ideas how to solve such problems, we first need a reference technique to compare against. Here the DifferentialEquations.jl Julia package provides a range of production-grade numerical solvers for ODE problems.

We first translate our problem (2) into Julia code.

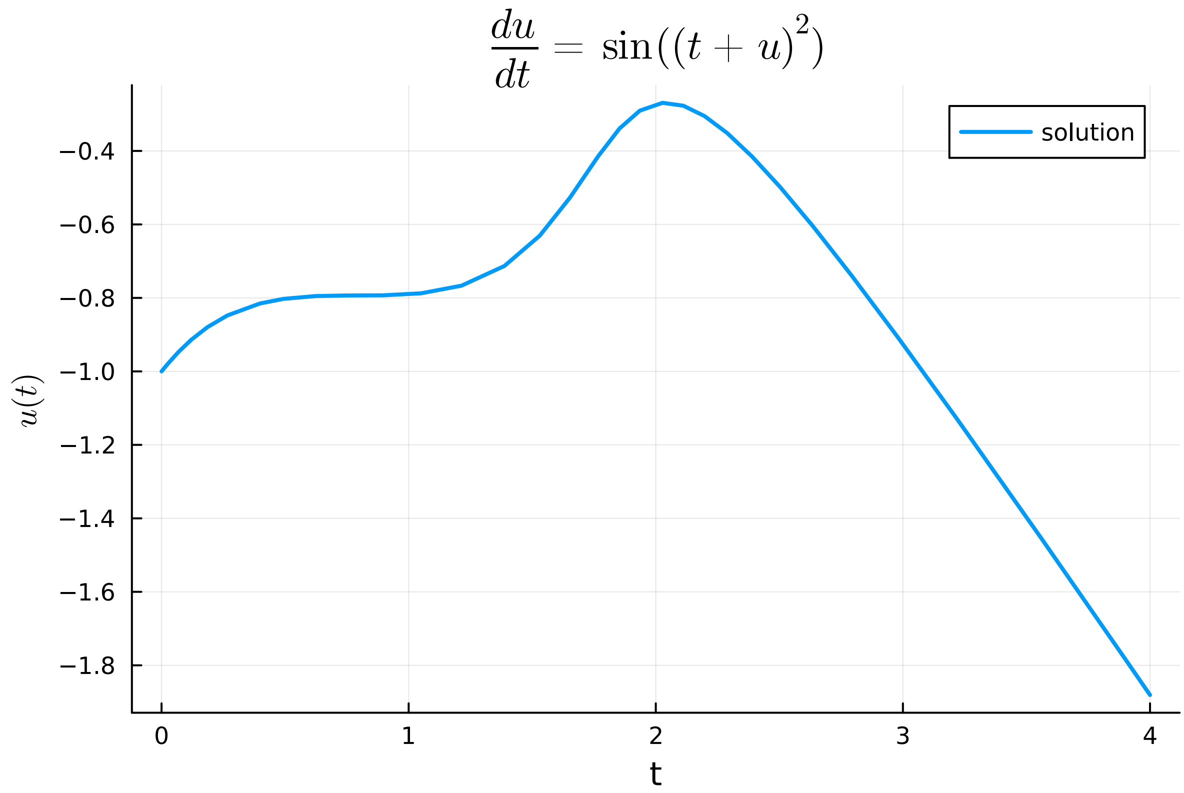
```julia
begin
    f(u, t) = sin((t + u)^2)  # defines du/dt
    u₀ = -1.0                 # initial value
    tstart = 0.0              # Start time
    tend   = 4.0              # Final time
end;
```

This we can now solve using `DifferentialEquations`, the details of which is beyond the scope of this course and hidden in the function `solve_reference`.

```
solve_reference (generic function with 1 method)
```

```julia
sol = solve_reference(f, u₀, tstart, tend);
```

The obtained solution can easily be visualised:



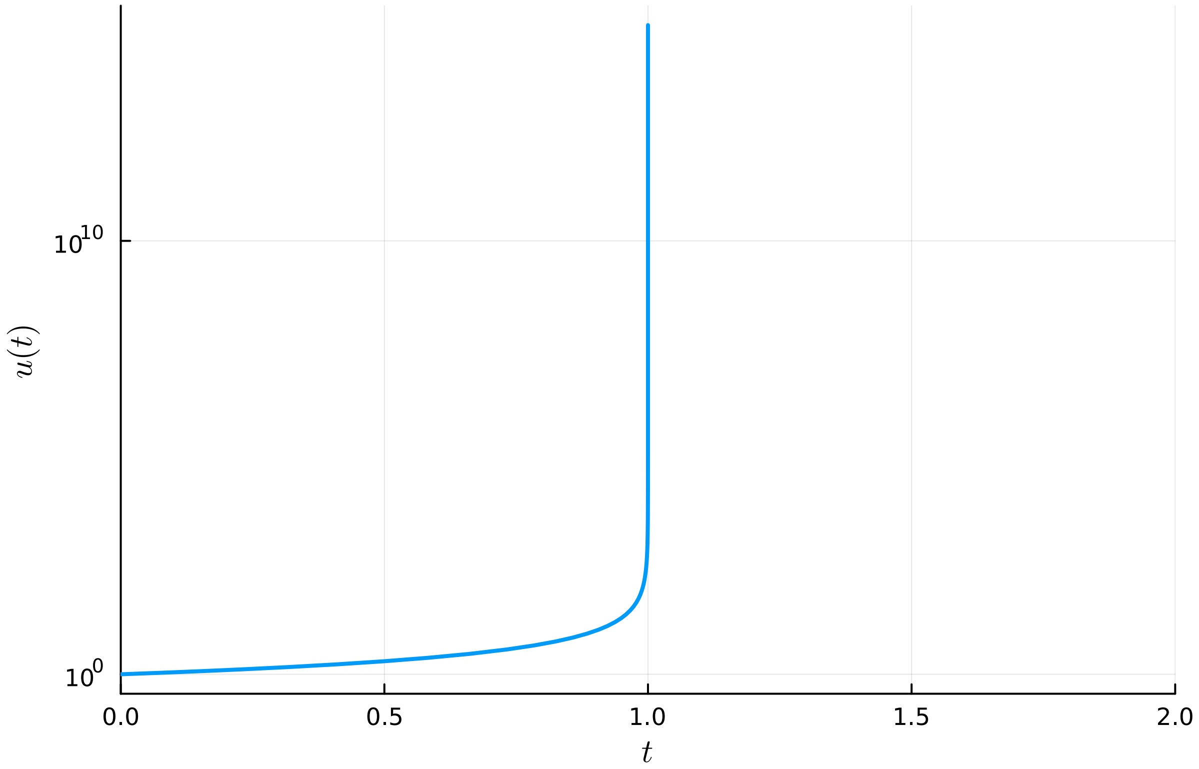
```julia
plot(sol.t, sol.u; label="solution", lw=2, xlabel="t", ylabel=L"u(t)",
      title=L"\frac{du}{dt} = \sin((t+u)^2)")
```

Existence and uniqueness of solutions

Let us remark that the **existence** of a solution to problem (1) is not guaranteed for all t with $a \leq t \leq b$. For example consider the problem

$$\frac{du(t)}{dt} = (u(t))^2, \, 0 \leq t \leq 2, \qquad u(0) = 1,$$

which has solution $u(t) = \frac{1}{1-t}$. This solution only exists for $t < 1$. When attempting a numerical solution beyond $t = 1$ of such a problem we are faced with problems:



```julia
let
    f(u, t) = u^2
    u₀      = 1.0
    tstart  = 0
    tend    = 10

    sol = solve_reference(f, u₀, tstart, tend)
    plot(sol.t, sol.u; lw=2, label="", xlabel=L"t", yaxis=:log, ylabel=L"u(t)", xlims=(0, 2))
end
```

```
⚠ At t=1.0000007880693895, dt was forced below floating point epsilon 2.220446049
  250313e-16, and step error estimate = 1.244050736875679. Aborting. There is eit
  her an error in your model specification or the true solution is unstable (or t
  he true solution can not be represented in the precision of Float64).
```

A second issue is that the solution may not necessarily be **unique**. Consider for example the equation

$$\frac{du(t)}{dt} = \sqrt{u(t)},\, t > 0, \qquad u(0) = 0,$$

which has the two solutions $u_1(t) = 0$ and $u_2(t) = \frac{1}{4}t^2$.

Both cases cause additional difficulties to numerical techniques, which we do not want to discuss here. For the scope of the course **we will assume that all IVP problems we consider, have a unique solution.**

If you are curious, the precise conditions for existence and uniqueness are given below:

▶ **Optional: Theorem governing Existence and uniquens of first-order ODEs**

Forward Euler

Let us return to the question how to solve the IVP (1)

$$\begin{cases} \dfrac{du(t)}{dt} = f(t, u(t)) \qquad a \leq t \leq b \\ u(a) = u_0, \end{cases} \tag{3}$$

where the time derivative f is a known function, e.g. $f(t, u) = C\,u$ in the case of the duck population. Our goal is thus to trace how how the intial condition u_0 (at $t = a$) evolves in time until the final time $t = b$ is reached.

Since the entire time interval $[a, b]$ could be large, we will not attempt to solve this problem in one step. Instead we will perform a **time discretisation**. That is we split this time interval even further into smaller subintervals $[t_n, t_{n+1}]$ with

$$a = t_0 < t_1 < \cdots < t_n < t_{n+1} < \cdots t_N = b$$

and consider algorithms, which **propagate the solution** across one such interval $[t_n, t_{n+1}]$, i.e. which take the solution **at time t_n to the solution at t_{n+1}**.

Here, for simplicity we only consider a discretisation using **intervals of equal length h**, i.e.

$$t_n = a + n\,h \quad n = 0, \ldots, N \qquad h = \frac{b-a}{N}, \tag{4}$$

such that $t_0 = a$ and $t_N = b$. The parameter h is often called the **stepsize**. At time t_n we need to satisfy

$$\frac{d\,u(t_n)}{dt} = f(t_n, u(t_n)),$$

where we assume that we know $u(t_n)$, the solution at t_n, but the derivative $\frac{d\,u(t_n)}{dt}$ is unknown. Our task is to find $u(t_{n+1})$.

We make progress by approximating the dervative of u using one of the finite differences formulas discussed in the chapter on Numerical differentiation.

The simplest approach is to employ forward finite differences, i.e.

$$\frac{d\,u(t_n)}{dt} \approx D_h^+ u(t_n) = \frac{1}{h}\Big(u(t_n+h) - u(t_n)\Big) = \frac{1}{h}\Big(u(t_{n+1}) - u(t_n)\Big) \tag{5}$$

Introducing a short-hand notation $u^{(n)}$ for $u(t_n)$ and $u^{(n+1)}$ for $u(t_{n+1})$ we obtain from (3):

$$\begin{cases} \dfrac{u^{(n+1)} - u^{(n)}}{h} = f(t_n, u^{(n)}), \qquad \forall n = 0, \ldots, N-1 \\ u^{(0)} = u(a) = u_0 \end{cases} \tag{6}$$

Notably the first line can be rearranged to $u^{(n+1)} = u^{(n)} + h\, f(t_n, u^{(n)})$, thus providing us with a numerical scheme how to obtain $u^{(n+1)}$ — an approximation to the solution at t_{n+1} — from $u^{(n)}$ — an approximation to the solution at t_n.

This scheme is known as the **forward Euler method**:

Algorithm 1: Forward Euler method

Given an initial value problem $\frac{du}{dt} = f(u,t)$, $u(a) = u_0$ with $t \in [a, b]$ and nodes $t_n = a + n\,h$, $h = \frac{b-a}{N}$ iteratively compute the sequence

$$u^{(n+1)} = u^{(n)} + h\, f(t_n, u^{(n)}) \qquad \forall n = 0, \ldots, N-1. \tag{7}$$

Then $u^{(n)}$ is *approximately* the solution u at $t = t_n$.

$u^{(n)} \neq u(t_n)$

Notice, that the replacement of the derivative $\frac{d\,u(t_n)}{dt}$ in (6) by the forward finite difference approximation implies that the computed points $u^{(0)}, u^{(1)}, \ldots, u^{(N)}$ are **not equal to** the function values of the exact solution $u(t_0), u(t_1), \ldots, u(t_N)$.

A basic implementation of Euler's method is shown below. It expects the definition of the derivative (`f`), the initial value (`u₀`), the time interval $[a, b]$ and the number of time intervals N. The output is the vector of nodes t_n and the vector of approximate solution values $u^{(n)} \approx u(t_n)$.

```
forward_euler (generic function with 1 method)
```

```julia
function forward_euler(f, u₀, a, b, N)
    # f:  Derivative function
    # u₀: Initial value
    # a:  Start of the time interval
    # b:  End of the time interval

    # Discrete time nodes to consider:
    h = (b - a) / N
    t = [a + i * h for i in 0:N]

    # Setup output vector with all elements initialised to u₀
    u = [float(copy(u₀)) for i in 0:N]

    # Time integration
    u[1] = u₀
    for i in 1:N
        u[i+1] = u[i] + h * f(u[i], t[i])
    end

    (; t, u)
end
```

Let us apply this approach to the test problem (2)

$$\left\{\begin{aligned} \frac{du(t)}{dt} &= \sin\left[(u+t)^2\right] \qquad 0 \leq t \leq 4 \\ u(0) &= -1, \end{aligned}\right.$$

which we solved previously using `DifferentialEquations`. The variables `f`, `u₀`, `tstart` and `tend` already contains a setup of this initial value problem

```
f (generic function with 1 method)
```

```julia
f
```

```
-1.0
```

```julia
u₀
```

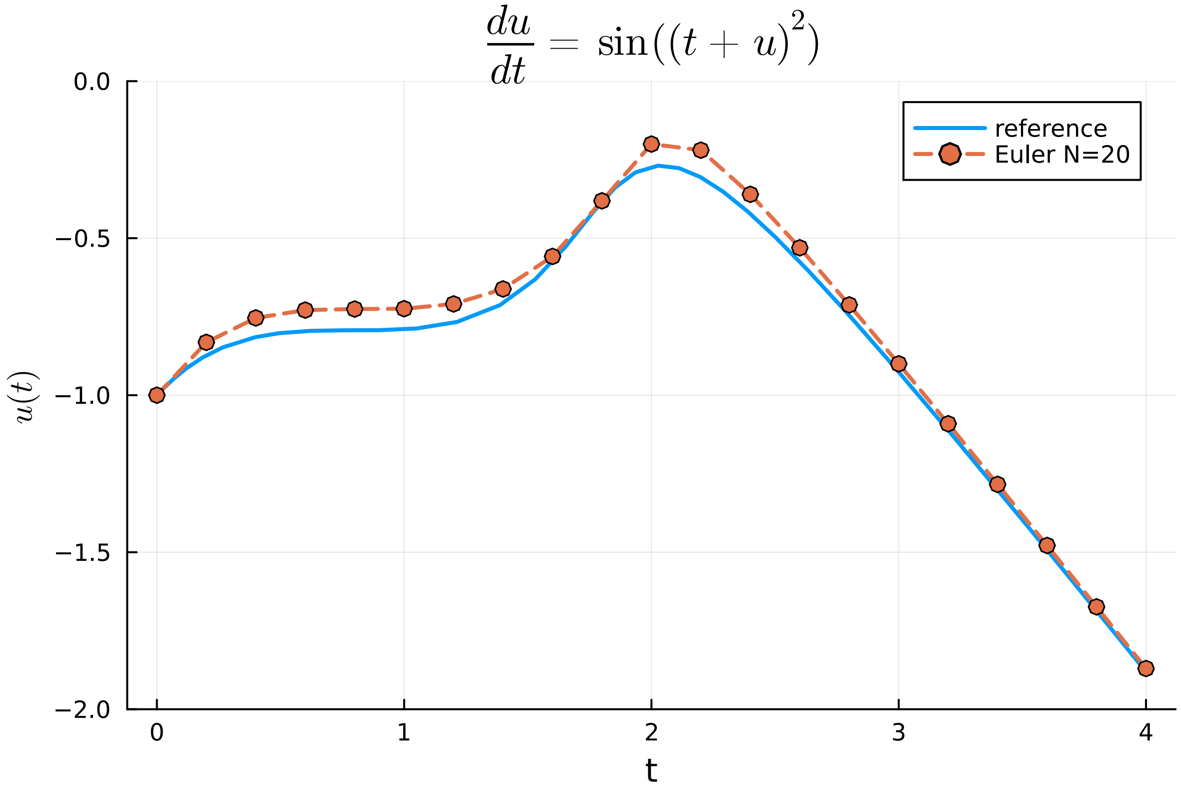
```
▸ (0.0, 4.0)
```

```julia
(tstart, tend)
```

and we can directly proceed to solving it:

```
res_euler = forward_euler(f, u₀, tstart, tend, Neuler);
```

and plot the result:

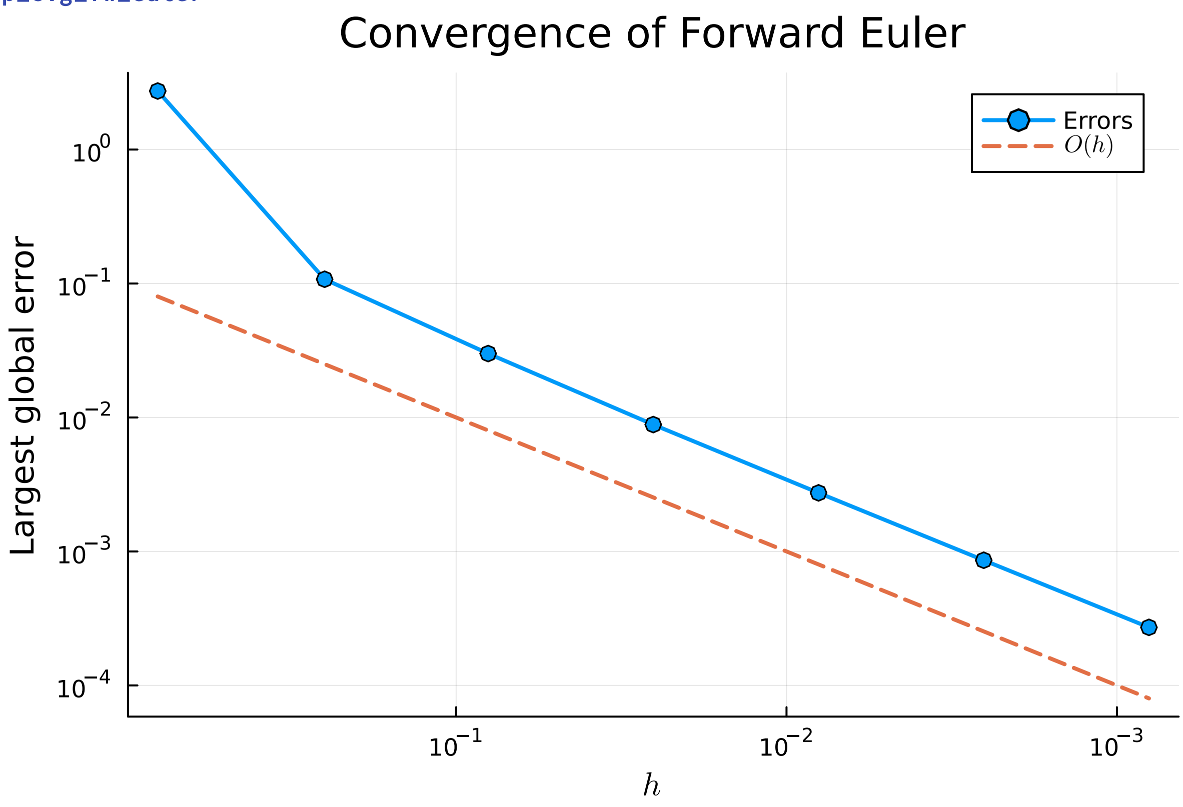


- Number of subintervals: `Neuler =` 20

We observe that the approximated solution becomes **more and more accurate** as we **increase the number of sub-intervals** N and as the step size h decreases. We obtain **linear convergence** (see plot below).

But we also observe that for **too small values** of N (e.g. $N = 9$) that the Euler **solution notably deviates** from the reference.

```
p_cvg_fw_euler =
```



Forward Euler is just one example of a broader class of so-called **explicit numerical methods** for initial value problems, which broadly speaking differ by the operations performed to obtain $u^{(n+1)}$ from $u^{(n)}$. We denote

Explicit methods for solving initial value problems

An **explicit method** to solve (1) is a method of the form

$$u^{(n+1)} = P_f(u^{(n)}, t_n, h) \qquad \text{for } n = 0, 1, \ldots, N-1, \tag{8}$$

where $P_f(u, t, h)$ is a real-valued function and $u^{(0)} = u_0$.

For forward Euler we simply have $P_f(u^{(n)}, t_n, h) = u^{(n)} + hf(t_n, u^{(n)})$.

Error analysis

As discussed above applying Forward Euler to an initial value problem (1) with N subintervals leads to a sequence of computed points $u^{(0)}, u^{(1)}, \ldots, u^{(N)}$, which approximate $u(t_0), u(t_1), \ldots, u(t_N)$ — the solution u evaluated at the nodes $\{t_i\}_{i=0}^N$. Our goal is to **quantify the error**

$$|e^{(n)}| = |u(t_n) - u^{(n)}| \qquad \text{for } n = 0, 1, \ldots, N \tag{9}$$

between the computed points and the exact values of the solution.

In employing the Forward Euler scheme (Algorithm 1) we actually make **two approximations**, namely:

1. Instead of evaluating the exact derivative $\frac{d\,u(t_n)}{dt}$ we employ a **finite differences formula** in (5). This error contribution is usually called the **local truncation error**.
2. When computing $u^{(n+1)}$ in (6) we cannot employ $u(t_n)$ — the exact value of the solution at time t_n — since this value is unknown to us. Instead we **employ $u^{(n)}$ as an approximation to** $u(t_n)$.

Both approximations contribute to the total error (9).

Local truncation error

We want to isolate the error in the Forward Euler scheme introduced by employing the finite difference formula from the other error contribution. For this let us assume for a second that we actually had access to the exact solution values $u(t_n)$, which we could use as part of a Forward Euler step (7).

The error of the Euler step is then simply $u(t_{n+1}) - P_f(u(t_n), t_n, h)$. To make the scaling of errors comparable as $h \to 0$ it is usually more convenient ot investigate the size of this value relative to the chosen stepsize h, leading to the following definition:

Definition: Local truncation error

The **local truncation error** of an explicit method (8) is the quantity

$$\tau_h^{(n)} = \frac{u(t_{n+1}) - P_f(u(t_n), t_n, h)}{h} \tag{10}$$

Making reference to (1) we first note for Forward Euler:

$$P_f(u(t_n), t_n, h) = u(t_n) + h\, f(t_n, u(t_n)) = u(t_n) + h\, u'(t_n)$$

while a Taylor expansion of $u(t_{n+1}) = u(t_n + h)$ around t_n yields

$$u(t_{n+1}) = u(t_n + h) = u(t_n) + hu'(t_n) + \frac{1}{2}h^2 u''(t_n) + O(h^3)$$

such that the **local truncation error of Forward Euler** can be obtained as

$$\tau_h^{(n)} = \frac{1}{2} h\, u''(t_n) + O(h^2) \tag{11}$$

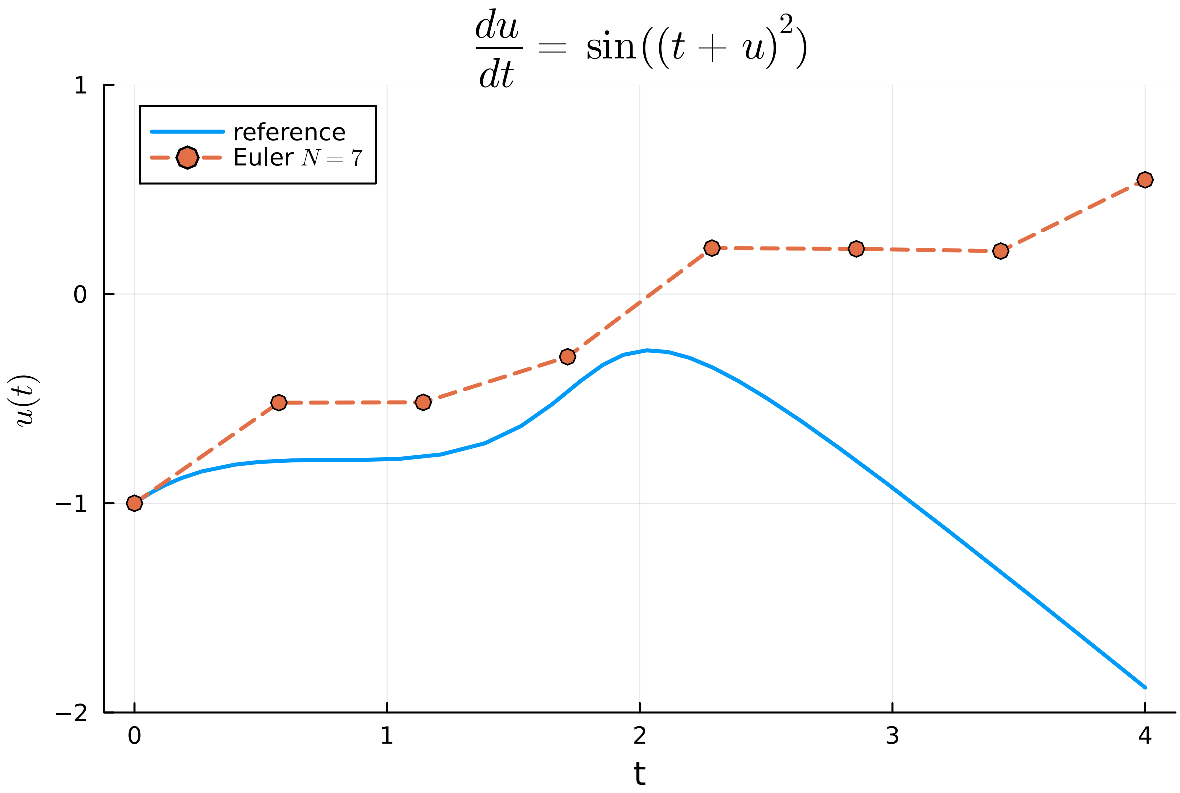
Global error

We return to the question of estimating the error

$$|e^{(n)}| = |u(t_n) - u^{(n)}| \qquad \text{for } n = 0, 1, \ldots, N.$$

In contrast to the *local* truncation error $\tau_h^{(n)}$ — which really only estimates the error committed in a single time step — the error $e^{(n)}$ is a **global error**, since it *accumulates* all error contributions up to the n-th timestep.

Naively one might think that simply adding all local error contributions $\tau_h^{(k)}$ with $k \leq n$ provides an estimate for this global error $e^{(n)}$. However, this neglects something important. Namely, that in our explicit algorithms (8), the *already erroneous estimate* $u^{(n)}$ is used to estimate $u^{(n+1)}$. **The error of step n *propagates* to step $n+1$**. For small values of N this can cause the error $|u(t_n) - u^{(n)}|$ in later time steps n to grow *exponentially*. For example consider:



```julia
let
    res_euler = forward_euler(f, u₀, tstart, tend, 7)
    plot(sol.t, sol.u, label="reference", lw=2, xlabel="t", ylabel=L"u(t)",
     title=L"\frac{du}{dt} = \sin((t+u)^2)", ylims=(-2, 1))
    plot!(res_euler.t, res_euler.u; label=L"Euler $N=7$", mark=:o, lw=2, ls=:dash)
end
```

More precisely one can formulate:

Theorem 2 (Convergence of explicit methods, simple version)

If one has an explicit method with local truncation error satisfying

$$|\tau_h^{(n)}| \leq C\,h^p,$$

as $h \to 0$ then as $h \to 0$ the global error satisfies

$$|e^{(n)}| = |u(t_n) - u^{(n)}| \leq \frac{Ch^p}{L}\left(e^{L(t_n-a)} - 1\right)$$

where $C > 0$ and $L > 0$ are constants.

We note:

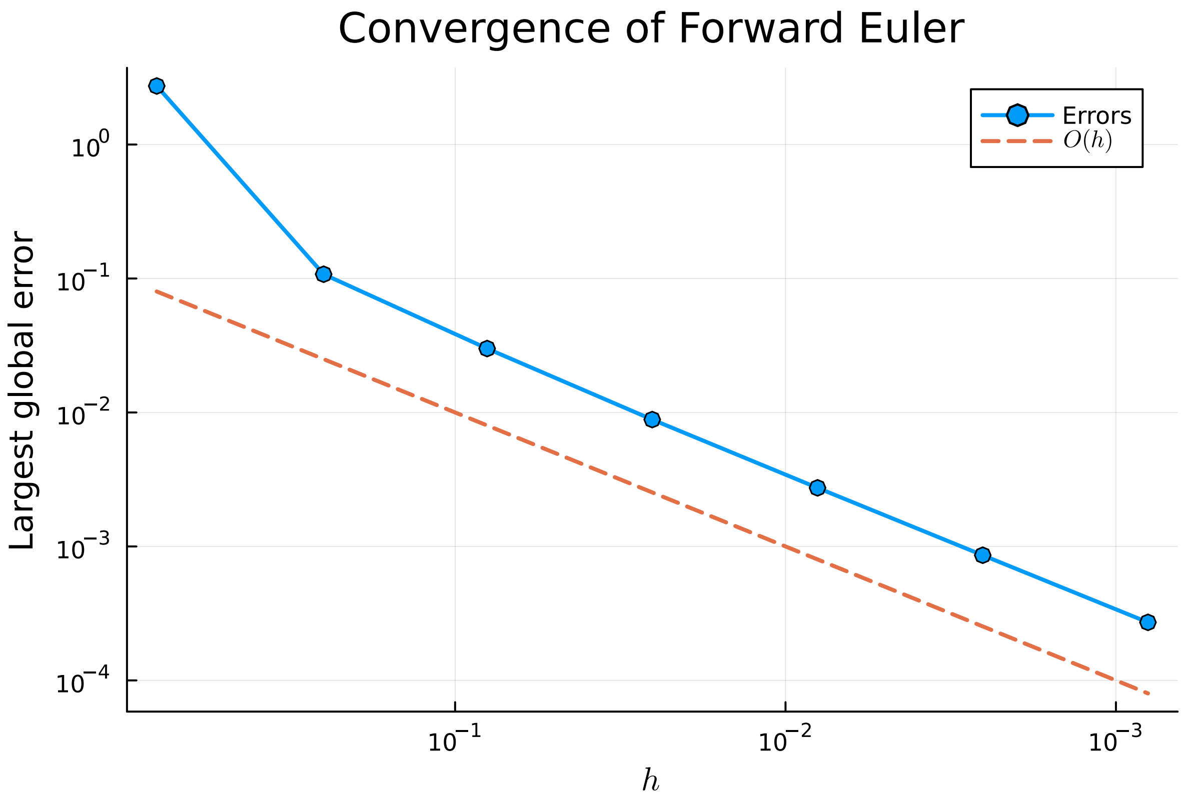
- If the local truncation error $\tau_h^{(n)}$ converges with order p, then the explicit methods also converges globally with order p.
- However, the global error has an **additional prefactor** $(e^{L(t_n-a)} - 1)$, which **grows exponentially in time**. This is an effect of the accumulation of error from one time step to the next. In particular if $b \gg a$ or results can get rather inaccurate **even for higher-order methods** beyond Forward Euler where $p > 1$. This point we will pick up in the section on *Stability and implicit methods* below.
- For Theorem 2 to hold there are a few more details to consider (e.g. problem (1) should have a unique solution). More information can be unfolded below.

▶ **Optional: More details on Theorem 2**

Observation: Convergence of Forward Euler

As discussed in (11) Forward Euler has local truncation error $\tau_h^{(n)} = \frac{1}{2} h\, u''(t_n) + O(h^2)$. Therefore $|\tau_h^{(n)}| \leq \frac{1}{2} \max_{t\in[t_n, t_{n+1}]} |u''(t)|\, h$ and by using Theorem 2 we conclude that **Forward Euler converges linearly**.

This is demonstrated graphically below:



Runge-Kutta methods

Similar to the previous chapters we now ask the question: How can we improve the convergence of a numerical method for an IVP ? The answer to this question leads to the major and most widely employed types of methods for solving initial-value problems, the **Runge-Kutta** (RK) **methods**.

We start with a second-order RK method, namely the midpoint method.

Optional: Midpoint method

The goal of the midpoint method is to construct a second-order method for solving the IVP (1)

$$\begin{cases} \dfrac{du(t)}{dt} = f\Big(t, u(t)\Big) & \qquad a \le t \le b \\ u(a) = u_0. \end{cases}$$

We stay within our general framework of (8), i.e. we want to iterate for $n = 0, 1, \dots, N-1$

$$u^{(n+1)} = P_f(u^{(n)}, t_n, h).$$

The ideal method would obtain the exact $u(t_{n+1})$ from the exact $u(t_n)$, implying a local truncation error $\tau_h^{(n)} = 0$. To build such a method we expand

$$\begin{aligned} u(t_{n+1}) = u(t_n + h) = u(t_n) + h\, u'(t_n) \qquad\qquad &+ \frac{1}{2} h^2 u''(t_n) + O(h^3) \\ = u(t_n) + h\, f\Big(t_n, u(t_n)\Big) &+ \frac{1}{2} h^2 u''(t_n) + O(h^3). \end{aligned} \tag{15}$$

where we used the definition of the IVP to replace u'. If in this equation we keep only the first two terms we recover the Forward Euler method (7). To obtain more accuracy we therefore need to compute or approximate $u''(t_n)$ as well. Using the chain rule we note

$$\frac{d^2u}{dt^2} = \frac{df}{dt} \overset{(*)}{=} \frac{\partial f}{\partial t} + \frac{\partial f}{\partial u} \frac{\partial u}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial u} f,$$

where notably step $(*)$ is neccessary because both f and its argument u depend on t. Inserting this expression into (15) we obtain

$$\begin{aligned} u(t_{n+1}) = u(t_n + h) = u(t_n) + h \bigg[f\Big(t_n, u(t_n)\Big) + \frac{h}{2} \frac{\partial f}{\partial t} \Big(t_n, u(t_n)\Big) \\ + \frac{h}{2} \frac{\partial f}{\partial u} \Big(t_n, u(t_n)\Big) f\Big(t_n, u(t_n)\Big) \bigg] + O(h^3). \end{aligned} \qquad (1$$

Since computing and implementing the partial derivatives $\frac{\partial f}{\partial t}$ and $\frac{\partial f}{\partial u}$ can in general become ticky, we will also approximate these further. Expanding f in a multi-dimensional Taylor series we observe

$$\begin{aligned} f\Big(t_n + \zeta, u(t_n) + \xi\Big) = f\Big(t_n, u(t_n)\Big) + \zeta \frac{\partial f}{\partial t} \Big(t_n, u(t_n)\Big) + \xi \frac{\partial f}{\partial u} \Big(t_n, u(t_n)\Big) \\ + O(\zeta^2 + |\zeta\xi| + \xi^2). \end{aligned}$$

With the choice $\zeta = \frac{h}{2}$ and $\xi = \frac{h}{2} f\big(t_n, u(t_n)\big)$ this expressions becomes equal to lowest order with the term in the square brackets of (16). This leads to

$$\begin{aligned} u(t_{n+1}) = u(t_n + h) = u(t_n) + h\, f\Big(t_n + \frac{h}{2}, u(t_n) + \frac{h}{2} f\big(t_n, u(t_n)\big)\Big) \\ + O(h^3 + h\zeta^2 + h|\zeta\xi| + h\xi^2). \end{aligned}$$

This is the **midpoint method** we mentioned earlier:

Algorithm 2: Midpoint method

Given an initial value problem $\frac{du}{dt} = f(u,t)$, $u(0) = u_0$ with $t \in [a,b]$ and nodes $t_n = a + n\,h$, $h = \frac{b-a}{N}$ iteratively compute the sequence

$$u^{(n+1)} = P_f(u^{(n)}, t_n, h) \qquad \forall n = 0, \ldots, N-1.$$

with

$$P_f(u^{(n)}, t_n, h) = u^{(n)} + h\left[f\left(t_n + \frac{h}{2}, u^{(n)} + \frac{h}{2} f(t_n, u^{(n)})\right)\right]$$

From our derivation the local truncation error for the midpoint method can be identified as

$$\tau_h^{(n)} = \frac{u(t_{n+1}) - P_f(u(t_n), t_n, h)}{h} = \frac{1}{h} O(h^3 + h\zeta^2 + h|\zeta\xi| + h\xi^2) = O(h^2).$$

From Theorem 2 we observe that the **midpoint method is** indeed **a second-order method**.

Runge-Kutta methods, such as the midpoint methods, are what one calls **multi-stage methods**. To make this term clear, let us rewrite equation (7) in two stages by isolating the green part:

$$\begin{aligned} u^{(n+\frac{1}{2})} &:= u^{(n)} + \frac{h}{2} f(t_n, u^{(n)}) \\ u^{(n+1)} &= u^{(n)} + h\,f\big(t_n + \frac{h}{2}, u^{(n+\frac{1}{2})}\big) \end{aligned}$$

Written like this we notice that the **first stage** (in green) performs a Forward Euler step with **half the stepsize**, namely from time t_n to $t_n + \frac{h}{2}$. The **second stage** then performs an Euler-style update over the whole timestep, but employing the slope $u^{(n+\frac{1}{2})}$ from the first stage in the computation of f.

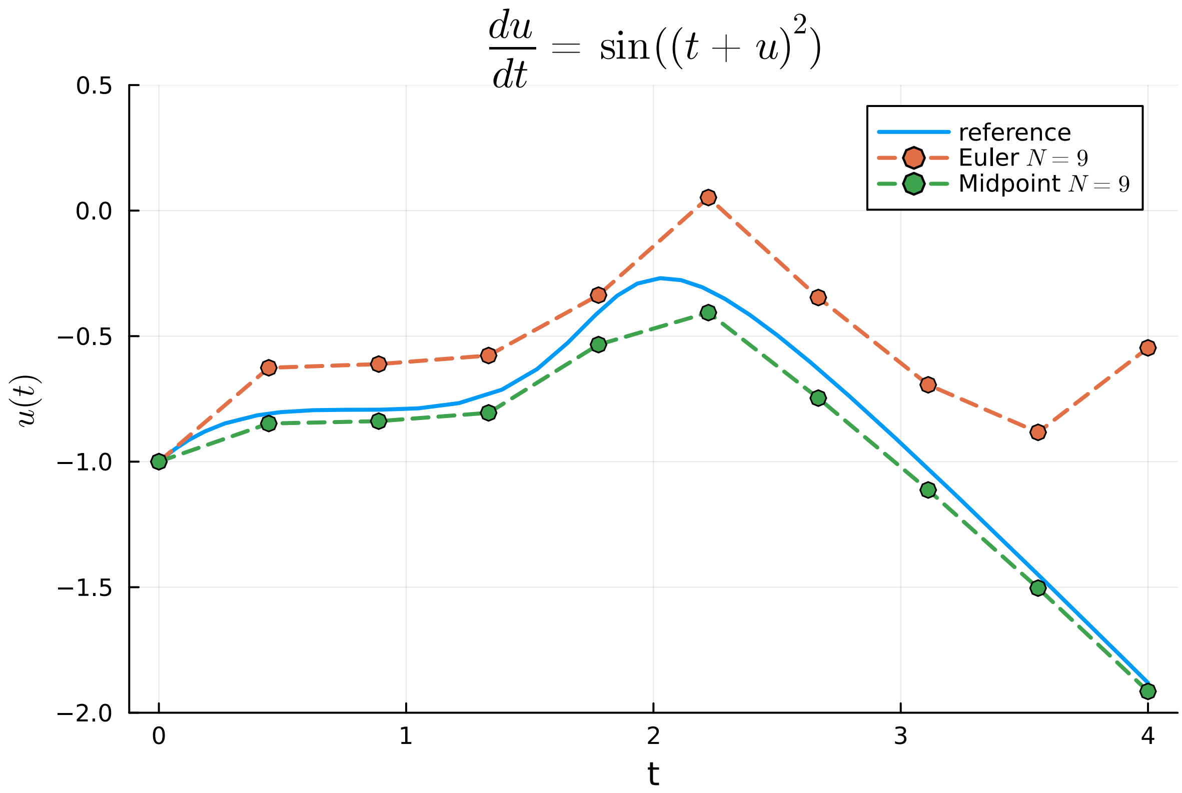
This interpretation also explains why the method is called **midpoint method**.

We obtain an implementation of the midpoint method as:

```
midpoint (generic function with 1 method)
```

```
function midpoint(f, u₀, a, b, N)
    # f:  Derivative function
    # u₀: Initial value
    # a:  Start of the time interval
    # b:  End of the time interval

    # Discrete time nodes to consider:
    h = (b - a) / N
    t = [a + i * h for i in 0:N]

    # Setup output vector with all elements initialised to u₀
    u = [float(copy(u₀)) for i in 0:N]

    # Time integration ... this is what changes over forward_euler
    u[1] = u₀
    for i in 1:N
        uhalf  = u[i] + h/2 * f(u[i],  t[i]      )
        u[i+1] = u[i] + h   * f(uhalf, t[i] + h/2)
    end

    (; t, u)
end
```

In comparison with Forward Euler we notice this method to be clearly more accurate for the case of rather small number of timesteps:



Optional: Higher-order Runge-Kutta methods

The ideas we used for constructing the midpoint method point to a clear path how one could construct even higher-order methods: All we need to do is to introduce further intermediate stages, i.e. half, third or other intermediate timesteps. In this way we can generate additional equations and effectively match the higher-order derivatives in the Taylor expansion (15), which will in turn reduce the local truncation error $\tau_h^{(n)}$. The methods generated in this form are called **Runge-Kutta methods**.

The algebra required to work out the details grows in complexity, so we will not attempt to do this here and only present a general overview. Constructing an s-stage Runge-Kutta methods leads to the set of equations

$$
\begin{aligned}
v_1 &= h\, f(t_n, \qquad\quad u_n) \\
v_2 &= h\, f(t_n + c_1\, h,\ u_n + a_{11}\, v_1) \\
v_3 &= h\, f(t_n + c_2\, h,\ u_n + a_{21}\, v_1 + a_{22}\, v_2) \\
&\ \vdots \\
v_s &= h\, f\left(t_n + c_{s-1}\, h,\ u_n + \sum_{i=1}^{s-1} a_{s-1,i}\, v_i\right) \\
u^{(n+1)} &= u^{(n)} + \sum_{i=1}^{s} b_i\, v_i.
\end{aligned}
$$

where specifying both the number of stages s as well as the constants a_{ij}, b_i and c_i uniquely determines an RK method. Both one-step methods we discussed so far actually match this framework:

- **Forward Euler:** $s = 1$, $b_1 = 1$ and no a_{ij} and no c_i.
- **Midpoint method:** $s = 2$, $b_1 = 0$, $b_2 = 1$, $c_1 = \frac{1}{2}$ and $a_{11} = \frac{1}{2}$.

Additionally we want to specify one additionall **fourth-order RK** method, often called **RK4**, which is the most commonly used IVP approach:

Algorithm 3: Fourth-order Runge-Kutta method (RK4)

Given an initial value problem $\frac{du}{dt} = f(u,t)$, $u(0) = u_0$ with $t \in [a,b]$ and nodes $t_n = a + n\,h$, $h = \frac{b-a}{N}$ iteratively compute the sequence

$$
\begin{aligned}
v_1 &= h\, f(t_n \qquad , u^{(n)}) \\
v_2 &= h\, f(t_n + \frac{h}{2}, u^{(n)} + \frac{v_1}{2}) \\
v_3 &= h\, f(t_n + \frac{h}{2}, u^{(n)} + \frac{v_2}{2}) \\
v_4 &= h\, f(t_n + h,\ \ u^{(n)} + v_3) \\
u^{(n+1)} &= u^{(n)} + \frac{v_1}{6} + \frac{v_2}{3} + \frac{v_3}{3} + \frac{v_4}{6}
\end{aligned}
$$

Let us mention in passing, that our reference solution routine `solve_reference` is using a method called `Tsit5()`, which is also based on a Runge-Kutta scheme.

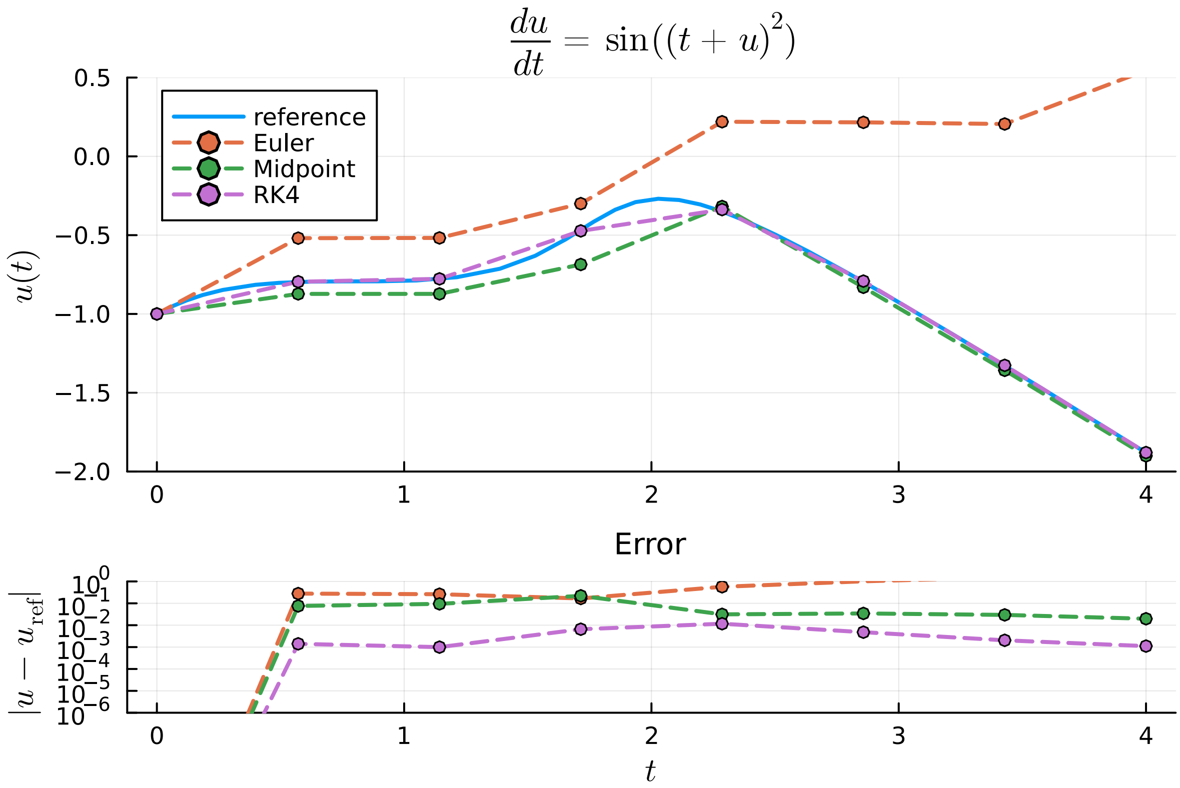
An implementation of RK4 is given by:

```
rk4 (generic function with 1 method)
```

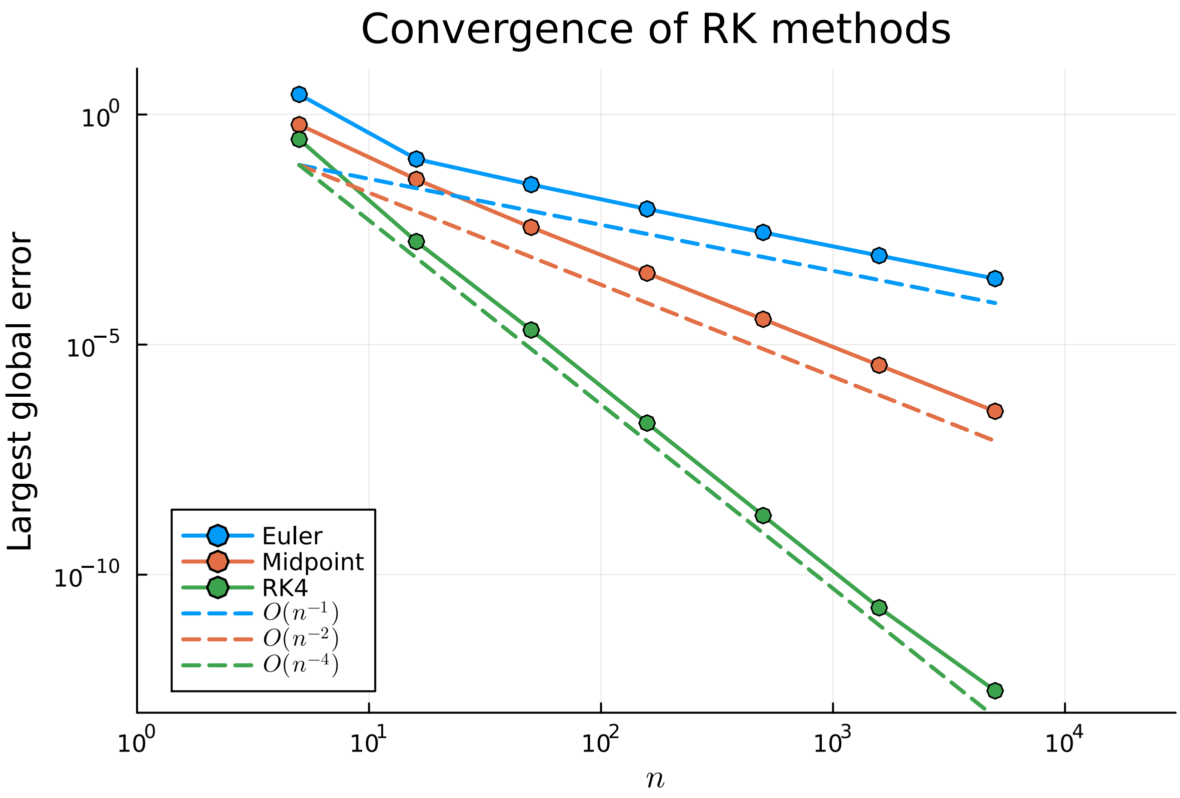
```julia
function rk4(f, u₀, a, b, N)
    # f:  Derivative function
    # u0: Initial value
    # a:  Start of the time interval
    # b:  End of the time interval

    # Discrete time nodes to consider:
    h = (b - a) / N
    t = [a + i * h for i in 0:N]

    # Setup output vector with all elements initialised to u₀
    u = [float(copy(u₀)) for i in 0:N]

    # Time integration ... this is what changes over forward_euler
    u[1] = u₀
    for i in 1:N
        v₁ = h * f(u[i],        t[i]      )
        v₂ = h * f(u[i] + v₁/2, t[i] + h/2)
        v₃ = h * f(u[i] + v₂/2, t[i] + h/2)
        v₄ = h * f(u[i] + v₃,   t[i] + h  )

        u[i+1] = u[i] + (v₁/6 + v₂/3 + v₃/3 + v₄/6)
    end

    (; t, u)
end
```

Let us compare all methods we saw in this chapter:

- N = 7

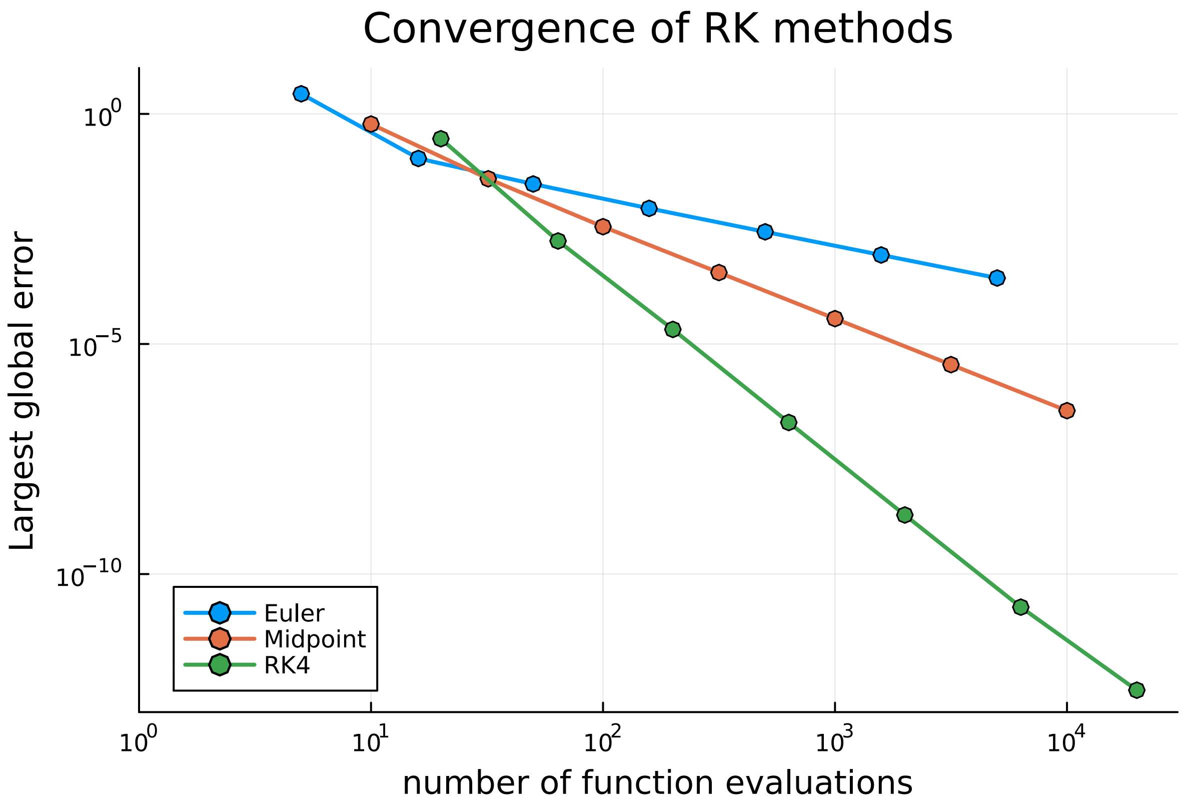


RK4 converges faster than the other two methods. As the name suggests it is indeed a fourth-order method:



On the other hand the four stages of RK4 also have a **disadvantage**: For each timestep four times as many evaluations of the function f are required than Forward Euler. Since this is typically the cost-dominating step we might ask if RK4 should always be employed or if other schemes with less stages (like Midpoint or Euler) are sometimes more appropriate.

To answer this question we take our previous plot and multiply the x-values by the number of stages in order to track the error versus the number of evaluations of the function f. With this we obtain:



We observe that even though RK4 needs four f evaluations per time step it still provides the best accuracy versus f-evaluations ratio for almost the entire range of evaluations we considered.

Stability and implicit methods

Let us consider the innocent looking initial value problem

$$\left\{\begin{aligned} \frac{du(t)}{dt} &= -Cu(t) \qquad t > 0\\ u(0) &= 10. \end{aligned}\right.$$

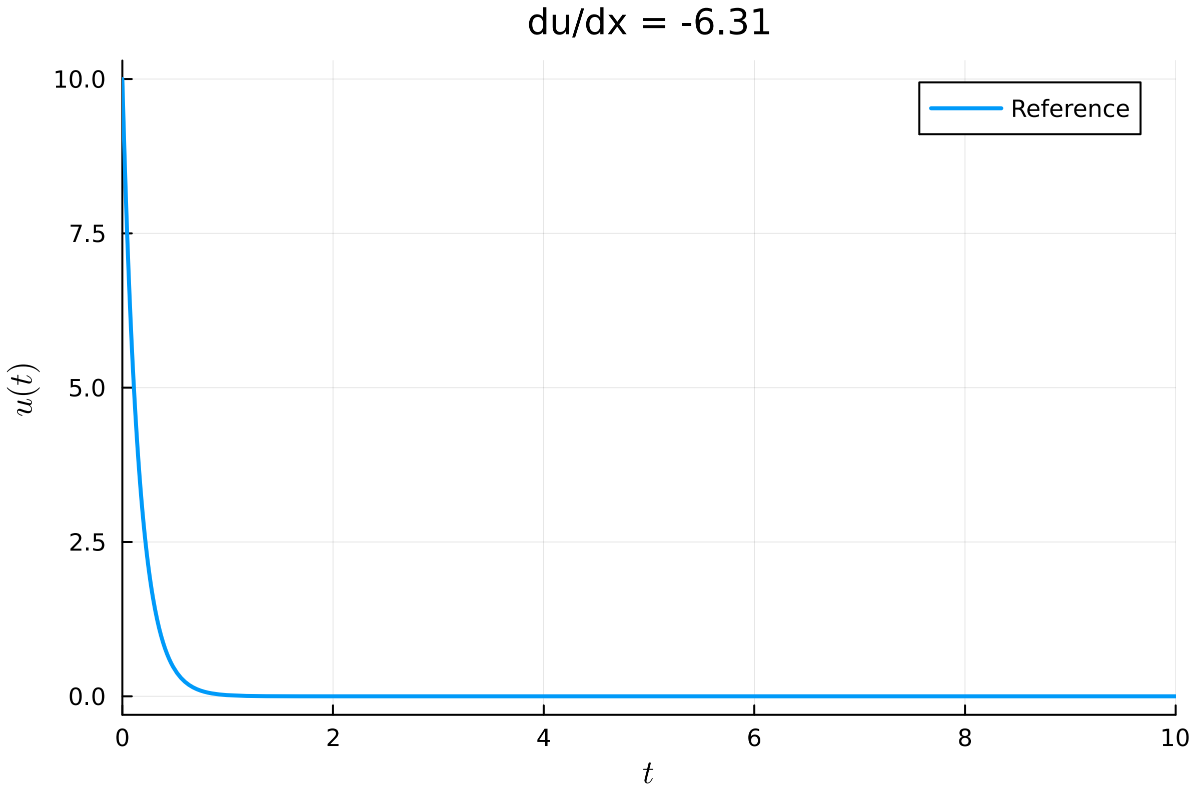
This model governs for example the decay of a radioactive species with initial concentration 10 over time. The rate of decay is $C \approx 6.31$.

By simple integration we find the exact solution of this problem as:

```
u (generic function with 1 method)
```

```julia
u(t) = 10 * exp(-C * t)
```

This function rapidly decays to zero as $t \to \infty$:



```julia
plot(u; xlims=(0, 10), lw=2, title="du/dx = $(round(-C; sigdigits=3))",
titlefontsize=12, label="Reference", xlabel=L"t", ylabel=L"u(t)")
```

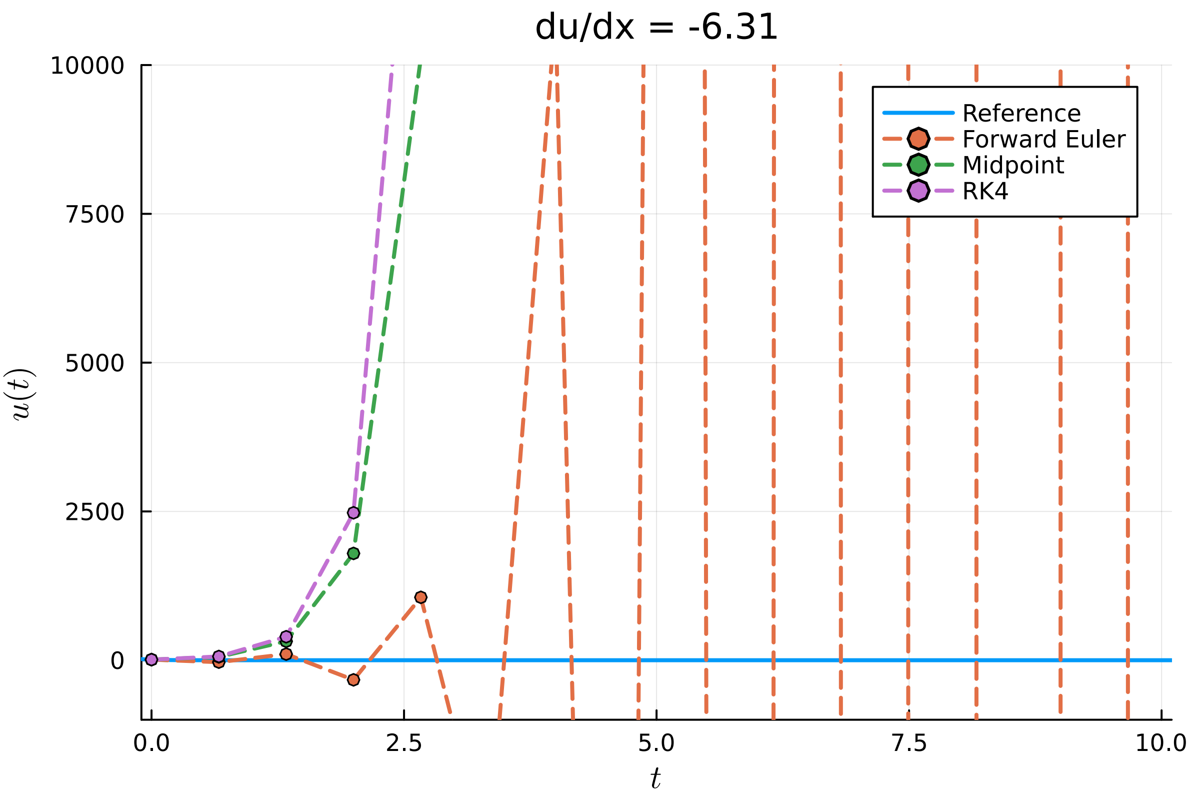
Applying our numerical methods from the previous sections, we would expect to recover this behaviour. So let's check this.

We first set up the problem in a bunch of variables prefixed by `dcy_`:

```julia
begin
    dcy_f(u, t) = -C * u
    dcy_u₀       = 10.0
    dcy_tstart  =  0.0
    dcy_tend    = 10.0
end;
```

… and try it on our previously implemented methods:

- `C =:` 6.309573444801933
- `Ndecay =:` 15



We notice for too few nodal points or too large decay constants C our numerical methods **all fail** to recover the decay behaviour. In other words while the exact solution satisfies $\lim_{t\to\infty} u(t) = 0$, the numerical methods are **all qualitatively wrong**: instead of the reproducing the correct long-time limit numerically, i.e. $\lim_{n\to\infty} u^{(n)} = 0$, the numerical solution actually grows over time.

Recall that in Theorem 2 we found that the global error satisfies

$$|e^{(n)}| \leq \frac{Ch^p}{L}\left(e^{L(t_n-a)} - 1\right) = \frac{Ch^p}{L}\left(e^{L\,hn} - 1\right).$$

In the limit of infinite time, i.e. $n \to \infty$, the upper bound on the RHS thus grows exponentially. This bound is in general pessimistic, i.e. not all numerical methods may actually reproduce the exponentially increasing error behaviour, but it clearly shows that unless one demands additional properties from a method for solving IVPs, one may fail to reproduce a limiting behaviour like $\lim_{t\to\infty} u(t) = 0$.

Without going into further details the property we ask for is called **stability**:

Definition: Absolute stability

Given an initial value problem (1) with exact solution $u(t)$ satisfying $\lim_{t\to\infty} u(t) = u_*$ for any initial condition u_0. In other words we have a problem where the solution converges to u_* independent of the initial value u_0.

Then a numerical method is called **absolutely stable** for a fixed stepsize h if

$$\lim_{n\to\infty} u^{(n)} = u_* \qquad \text{for any } u_0.$$

A method is further called **unconditionally absolutely stable** if it is stable for all $h > 0$.

In the case of our decay problem, one can show for example that the forward Euler method is absolutely stable if

$$h < \frac{2}{C},$$

which explains the condition we chose to switch the axis limits in the plot above.

We also observe that **none of our explicit methods is unconditionally absolutely stable**.

Backward Euler

The construction and implementation of unconditionally absolutely stable methods is in general more involved than the methods we considered so far. We will therefore only construct a single approach here.

For this we return to the derivation of the Forward Euler method. Recall that in order to solve the IVP (1)

$$\begin{cases} \dfrac{du(t)}{dt} = f(t, u(t)) \qquad a \le t \le b \\ u(a) = u_0, \end{cases}$$

we introduced a time discretisation into N subintervals $[t_n, t_{n+1}]$, such that in each interval we needed to solve the problem

$$\frac{d\,u(t_n)}{dt} = f(t_n, u(t_n)). \tag{20}$$

Instead of employing the forward finite differences formula, leading to the Forward Euler method (7), we now instead employ the **backward finite differences**, leading to

$$\frac{d\,u(t_{n+1})}{dt} \approx D_h^- u(t_{n+1}) = \frac{1}{h}\Big(u(t_{n+1}) - u(t_{n+1} - h)\Big) = \frac{1}{h}\Big(u(t_{n+1}) - u(t_n)\Big). \quad (2$$

Following the same idea as before, i.e. to employ for $u(t_{n+1})$ the approximation $u^{(n+1)}$ of the $(n+1)$-st time step and instead of $u(t_n)$ employ $u^{(n)}$ we obtain from (20) and (21):

$$\frac{u^{(n+1)} - u^{(n)}}{h} = f(t_{n+1}, u^{(n+1)}), \qquad \forall n = 0, \ldots, N-1, \tag{22}$$

which is the **Backwards Euler method**.

Notice, that in contrast to the case of Forward Euler we cannot immediately deduce an explicit update formula like (7) from this expression, since the **dependency on $u^{(n+1)}$ is both on the LHS as well as in f**. This is why such methods are called **implicit methods** as $u^{(n+1)}$ is only implicitly defined.

To obtain $u^{(n+1)}$ from $u^{(n)}$ one **needs to solve (22) iteratively**: For this we introduce the map

$$x = g^{(n)}(x) = u^{(n)} + h\, f(t_{n+1}, x) \tag{23}$$

and notice that its fixed-point $x_* = g^{(n)}(x_*)$ is exactly $u^{(n+1)}$. For each time step n we thus need to solve a fixed-point problem using one of the methods we described in Root finding and fixed-point problems.

In summary our algorithm becomes:

Algorithm 4: Backward Euler method

Given an initial value problem $\frac{du}{dt} = f(u,t)$, $u(0) = u_0$ with $t \in [a,b]$ and nodes $t_n = a + n\,h$, $h = \frac{b-a}{N}$ we iterate for $n = 1, \ldots N$:

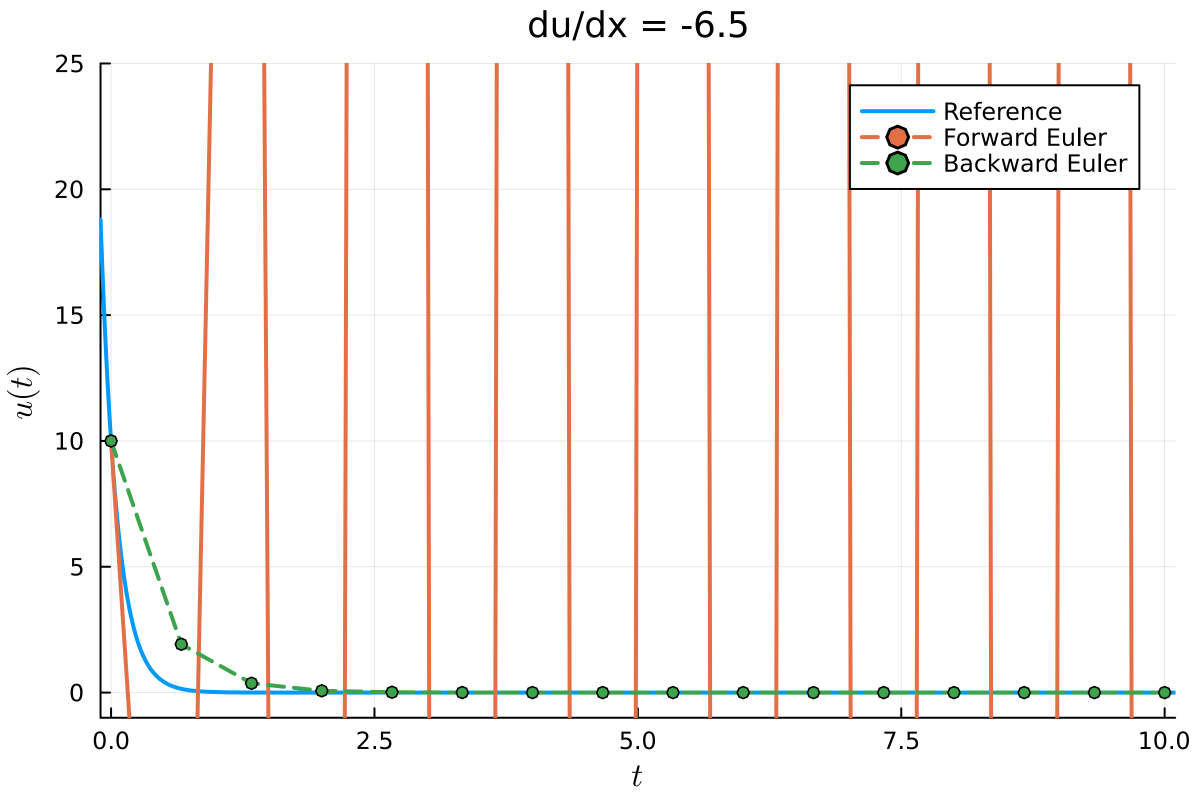
- Find a fixed-point x_* of the map $g^{(n)}(x) = u^{(n)} + h\, f(t_{n+1}, x)$, e.g. using the Newton method.
- Set $u^{(n+1)} = x_*$

An implementation of Backward Euler employing Newton's method to solve the fixed-point problem is given below. The implementation of Newton's method is repeated in the appendix.

backward_euler (generic function with 1 method)

```julia
function backward_euler(f, u₀, a, b, N; tol=1e-10)
    # f:  Derivative function
    # u₀: Initial value
    # a:  Start of the time interval
    # b:  End of the time interval


    # Discrete time nodes to consider:
    h = (b - a) / N
    t = [a + i * h for i in 0:N]

    # Setup output vector with all elements initialised to u₀
    u = [float(copy(u₀)) for i in 0:N]

    # Time integration
    u[1] = u₀
    for i in 1:N
        g(x)    = u[i] + h * f(x, t[i+1])
        dg(x)   = ForwardDiff.derivative(g, x)
        u[i+1] = fixed_point_newton(g, dg, u[i]; tol).fixed_point
    end

    (; t, u)
end
```

- Nbw = 15



```
let
    C = 6.5

    soln_euler    = forward_euler(dcy_f, dcy_u₀, dcy_tstart, dcy_tend, Nbw)
    soln_bw_euler = backward_euler(dcy_f, dcy_u₀, dcy_tstart, dcy_tend, Nbw)

    p = plot(u; lw=2, title="du/dx = $(round(-C; sigdigits=3))", titlefontsize=12, label="Reference", xlabel=L"t", ylabel=L"u(t)", xlims=(-0.1, 10.1), ylims=(-1, 25))

    plot!(p, soln_euler.t, soln_euler.u; mark=:o, c=2, lw=2, markersize=3, ls=:dash, label="Forward Euler")
    plot!(p, soln_bw_euler.t, soln_bw_euler.u; mark=:o, c=3, lw=2, markersize=3, ls=:dash, label="Backward Euler")
end
```

While Forward Euler stays absolutely stable only for large values of `Nbw` (respectively small values of h), Backward Euler stays stable no matter what value of `Nbw` is chosen.

Let us conclude by mentioning that similar to Forward Euler, **Backward Euler** is also only a **first order method**. Higher-order implicit methods can also be constructed. E.g. the Runge-Kutta family of methods can be extended to the implicit setting as well. An example is the Crank–Nicolson method, a second-order implicit Runge-Kutta method.

In standard libraries, such as DifferentialEquatios.jl a zoo of ODE methods is typically available.

Optional: Higher-order differential equations

Consider the setting of a mass m hanging on a spring from the ceiling:

```
# TODO("Image")
```

In its rest position show above, gravity is exactly cancelled by the upward force excerted by the spring, such that we can ignore gravity for our discussion. Now displacing the mass by a length x leads to a restoring force $-kx$ where k is the spring constant of the spring. By Newton's law this accelerates the mass by $-\frac{k}{m}x$. Employing $m = 1$ for simplicity this leads to the **second-order ODE**

$$\begin{cases} \dfrac{d^2x(t)}{dt^2} = -k\,x(t) & t > 0 \\ x(0) = 0, \end{cases}$$

which is usually referred to as the **simple harmonic oscillator model**, abbreviated HO. To solve this problem numerically we first introduce the velocity function $v(t)$ and rewrite it as a system of first-order ODEs:

$$\begin{cases} \dfrac{d\,x(t)}{dt} = v(t) & t > 0 \\ \dfrac{d\,v(t)}{dt} = -k\,x(t) & t > 0 \\ x(0) = 0 \\ v(0) = 0.5, \end{cases} \tag{17}$$

where we choose to start at the rest position, but with an initial velocity of 0.5. Collecting the position and velocity into a single vector-valued variable, i.e.

$$\mathbf{u}(t) = \begin{pmatrix} u_1(t) \\ u_2(t) \end{pmatrix} = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}, \qquad \frac{d\mathbf{u}(t)}{dt} = \begin{pmatrix} \frac{d\,u_1(t)}{dt} \\ \frac{d\,u_2(t)}{dt} \end{pmatrix} = \begin{pmatrix} \frac{d\,x(t)}{dt} \\ \frac{d\,v(t)}{dt} \end{pmatrix}, \qquad \mathbf{u}_0 = \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}$$

and introducing the function $\mathbf{f} : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ with

$$\mathbf{f}(\mathbf{u}, t) = \begin{pmatrix} u_2(t) \\ -k\,u_1(t) \end{pmatrix}$$

we can rewrite this as

$$\begin{cases} \dfrac{d\,\mathbf{u}(t)}{dt} = \mathbf{f}(\mathbf{u}, t) \qquad t > 0 \\ \mathbf{u}(0) = \mathbf{u}_0. \end{cases} \tag{18}$$

Notice, that this has exactly the same structure as (1), just all quantities are vector valued.

All algorithms we discussed in this chapter so far can be extended to this setting. For example Algorithm 1 (Forward Euler) simply becomes:

Algorithm 1a: Forward Euler for vector-valued problems

Given an initial value problem $\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u})$, $\mathbf{u}(0) = \mathbf{u}_0$ with $t \in [a, b]$ and nodes $t_n = a + n\,h$, $h = \frac{b-a}{N}$ iteratively compute the sequence

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + h\,\mathbf{f}(t_n, \mathbf{u}^{(n)}) \qquad \forall n = 0, \ldots, N-1. \tag{19}$$

which is identical to Algorithm 1, just with all quantities replaced by their vector-valued analogues. In fact even the implementations of `forward_euler` can just be used without changing a single line of code as we will demonstrate now.

First we setup the HO model function and parameters:

```
k = 2
```

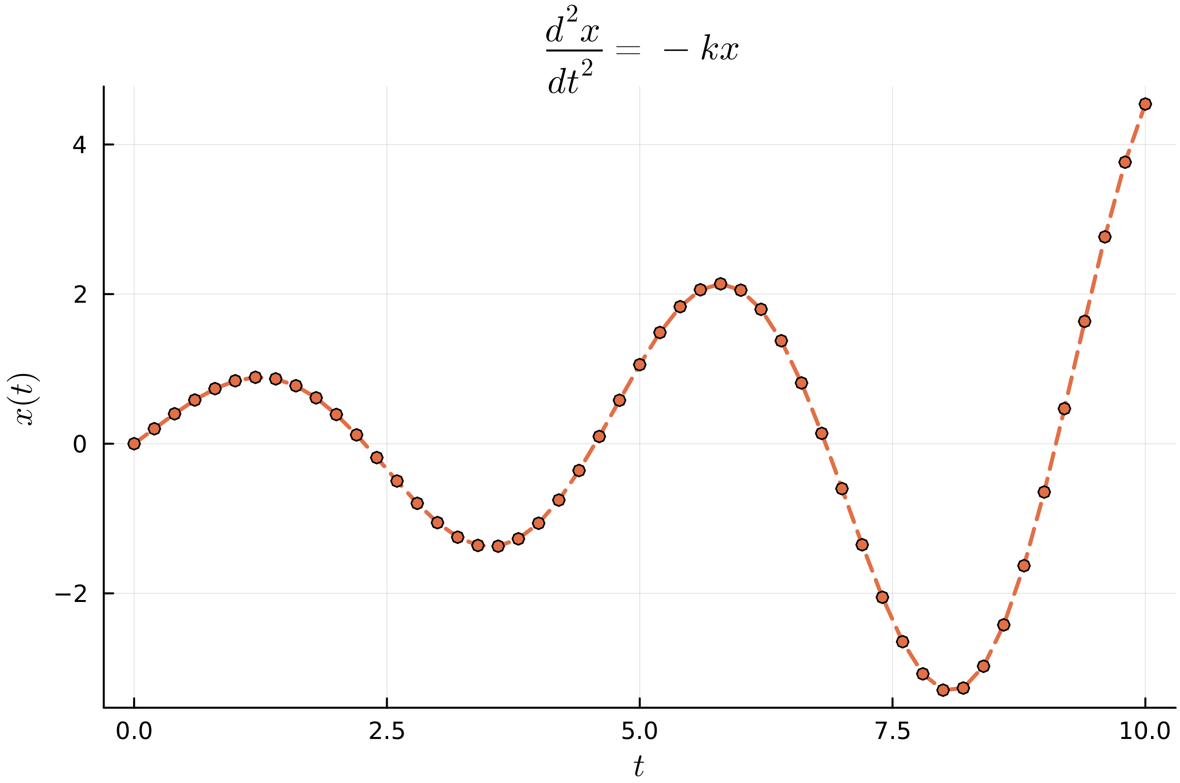
```
k = 2  # Force constant
```

```
10.0
```

```
begin
    # To setup f we assume u is a vector of size 2 and return a vector of size 2
    ho_f(u, t) = [     u[2];
                  -k * u[1]]

    # As the initial value again we supply a vector of size 2
    ho_u₀ = [0.0;
             1.0]

    # and we are interested in the behaviour from 0 to tend (defined below)
    ho_tstart =  0.0
    ho_tend   = 10.0
end
```

Then running the dynamics just takes a call to `forward_euler` as before:

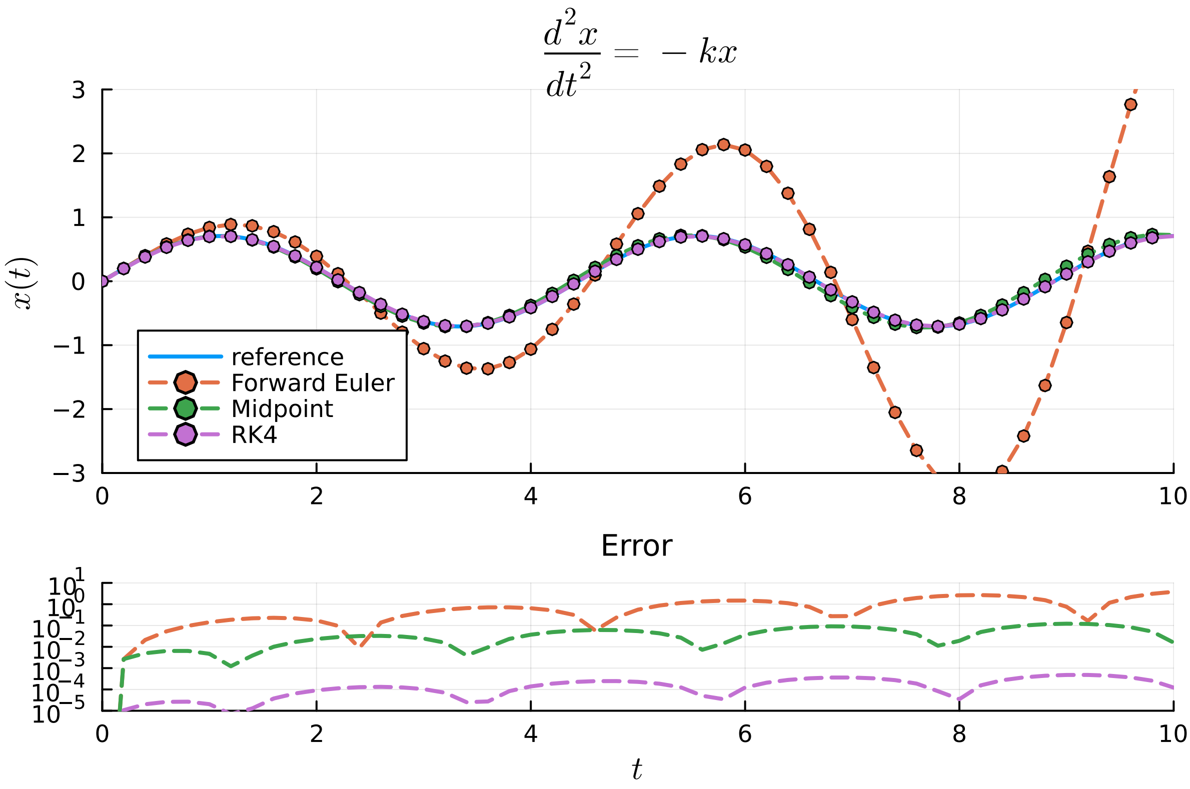
```
ho_euler = forward_euler(ho_f, ho_u₀, ho_tstart, ho_tend, 50);
```



```julia
let
    x = [u[1] for u in ho_euler.u]  # Extract the particle position
    plot(ho_euler.t, x, mark=:o, c=2, lw=2, xlabel=L"t", ylabel=L"x(t)", label="", title=L"\frac{d^2 x}{d t^2} = -k x", titlefontsize=12, markersize=3, ls=:dash)
end
```

In fact the `midpoint` and `rk4` implementations we provide are similarly generic with respect to being used with scalar-valued or vector-valued u. Applying them all we obtain:

- `Nho =` 50



Appendix

```
fixed_point_newton (generic function with 1 method)
```

```julia
function fixed_point_newton(g, dg, xstart; maxiter=40, tol=1e-6)
    f(x)  = g(x)  - x
    df(x) = dg(x) - 1
    (; root, n_iter, history_x, history_r) = newton(f, df, xstart; maxiter, tol)
    (; fixed_point=root, n_iter, history_x, history_r)
end
```

```
newton (generic function with 1 method)
```

```julia
function newton(f, df, xstart; maxiter=40, tol=1e-6)
    # f:  Function of which we seek the roots
    # df: Function, which evaluates its derivatives
    # xstart: Start of the iterations
    # maxiter: Maximal number of iterations
    # tol: Convergence tolerance

    history_x = [float(xstart)]
    history_r = empty(history_x)

    r = Inf  # Dummy to enter the while loop
    k = 0

    # Keep running the loop when the residual norm is beyond the tolerance
    # and we have not yet reached maxiter
    while norm(r) ≥ tol && k < maxiter
        k = k + 1

        # Pick most recent entry from history_x (i.e. current iterate)
        x = last(history_x)

        # Evaluate function, gradient and residual
        r = - f(x) / df(x)

        push!(history_r, r)      # Push residual and
        push!(history_x, x + r)  # next iterate to history
    end

    (; root=last(history_x), n_iter=k, history_x, history_r)
end
```

Numerical analysis

12. Boundary value problems