# TDT4127 Programming and Numerics
# Week 43

Solving ordinary differential equations

# Important note

- Next week: **auditorium exercise 2**
  - **Bring your own computer**, or borrow one from NTNU!
  - **No** computer lab hours or exercise lectures next week
  - Instead, do some exercises live in an **auditorium** (2 hours)
  - Day: Friday, November 2
    - Due to space constraints, you are given a **fixed time/place**
    - Check this link to see where/when you should show up:
    https://www.ntnu.no/wiki/pages/viewpage.action?pageId=83234235
  - Install **Safe Exam Browser** (link above)
- One of the two auditorium exercises **must be approved** in order to take the exam
  - Also, we use **Inspera** to get acquainted before the exam
  - **New** kinds of **exercises** being tested this time!

# Learning goals

- Goals
  - Solving **ordinary differential equations**
  - Algorithm:
    - ***Explicit Euler method***
    - ***Implicit Euler method***
- Curriculum
  - Exercise set 9
  - Programming for Computations - Python
    - Ch. 4.1, 4.2

# Ordinary differential equations

- With Ordinary Differential Equations (**ODEs**) we know the **time derivative** of a function, $\dot{x}$, but not $x$ itself:
$$\dot{x}(t) = f(x, t)$$
- Ex: The *speed* of an object is known but not the *position*
- To solve an ODE we need the information of an *initial value* $x(0) = x_0$
  - *It could be that we have knowledge of* $x(t_0), t_0 \neq 0$ *instead*
    - *This does not change anything*
- The solution to an ODE is a time-dependent function $x(t)$ valid for $t > 0$

# How to solve an ODE numerically

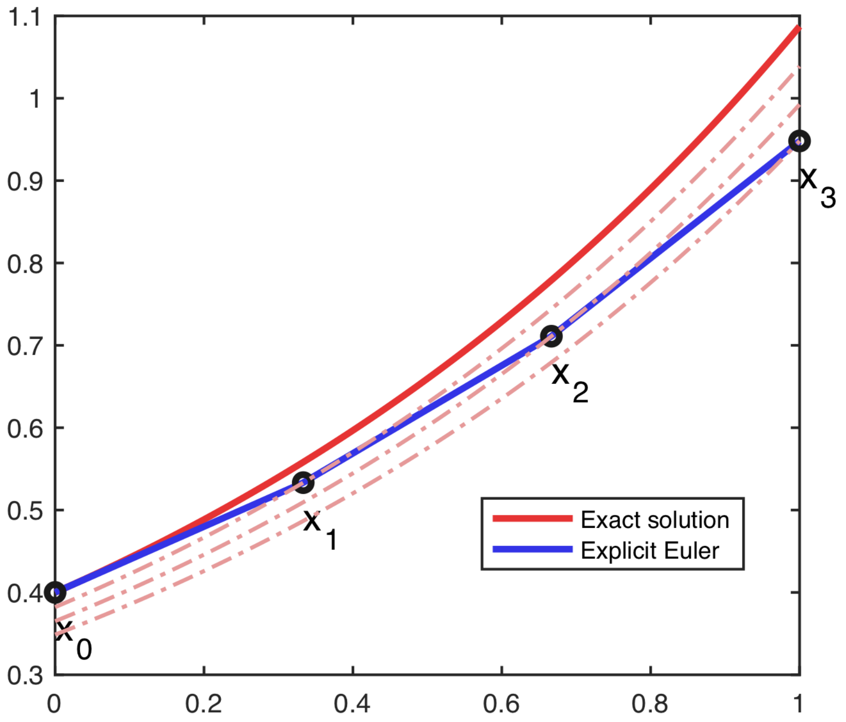
- The solution to an ODE is a time-dependent function $x(t)$ valid for **all** $t > 0$
- Numerical tradeoff no. 1: How about we settle for getting solutions only at snapshots in time?
- Introduce *discrete* times $0 = t_0 < t_1 < t_2 < \dots$
- For simplicity, space them with equal time step sizes $h$:

$$t_j = jh$$

- Numerical tradeoff no. 2: How about we solve the ODE for a **limited time** only?
- Instead of $t \in (0, \infty)$, let $t_k \in (0, T)$
  - Gives us a finite amount of time steps
  - Specify *stopping time* $T$ and no. of time steps $N$, then take $h = T/N$

# Geometric description of explicit Euler

- Starting at $x_0$, follow the tangent line of $x(t)$ until $t_1$
- At the next point, $x_1$, calculate the tangent of the solution with initial condition $x(t_1) = x_1$
- Follow this tangent line until $t_2$, call this point $x_2$, etc.



# Formulaic description of explicit Euler

- Starting with an **ODE**

$$\dot{x}(t) = f(x(t), t)$$

we approximate, using *Taylor's theorem*:

$$x(t+h) \approx x(t) + h\,\dot{x}(t) = x(t) + hf(x(t), t)$$

- Starting at $t_0 = 0$ this means

$$x(h) \approx x(0) + hf(x(0), 0)$$
$$x(2h) \approx x(h) + hf(x(h), h)$$
$$\vdots$$
$$x(t_{j+1}) \approx x(t_j) + hf(x(t_j), t_j)$$

- This leads to the **explicit Euler scheme**

$$x^{j+1} = x^j + hf(x^j, t_j)$$

# The explicit Euler algorithm

1. Set the initial condition $x_0$, choose the number of time steps $N$ and the stopping time $T$. Compute $h = T/N$
2. for j in range(0,N):

$$x^{j+1} = x^j + hf(x^j, t_j)$$

- This scheme is *explicit*: You can calculate the right-hand side directly.
- The step size $h$ cannot be too large (examples next week). The smaller $h$ is (the larger $N$ is), the better the approximations.
  - Intuition: We are taking smaller steps, thus making smaller errors

# The implicit Euler method

- How about another derivative: $f(x^{j+1})$ instead of $f(x^j)$?
- This amounts to using Taylor's theorem differently:

$$x(t-h) \approx x(t) - hf(x(t), t)$$
$$\Rightarrow x(t) \approx x(t-h) + hf(x(t), t).$$

- Starting at $\mathrm{t} = h$ this means we can take

$$x(h) \approx x(0) + hf(x(h), h)$$
$$x(2h) \approx x(h) + hf(x(2h), 2h)$$
$$\vdots$$
$$x(t_{j+1}) \approx x(t_j) + hf(x(t_{j+1}), t_{j+1})$$

- This leads to the implicit Euler scheme

$$x^{j+1} = x^j + hf(x^{j+1}, t_{j+1})$$

# The implicit Euler algorithm

1. Set the initial condition $x_0$, choose the number of time steps $N$ and the stopping time $T$. Compute $h = T/N$
2. for j in range(0,N):

   solve $x^{j+1} = x^j + hf(x^{j+1}, t_{j+1})$

- This scheme is *implicit*: both sides of the expression depend on $x^{j+1}$ and so we must solve an equation for $x^{j+1}$ each step.
- The step size $h$ can be larger here than in the explicit Euler method (more next week). Still: the smaller $h$ is (the larger $N$ is), the better the approximations.

# Heun's method

- **Explicit Euler** uses information from the «old» point only
- **Implicit Euler** uses information from the «new» point but requires solution of an equation each step
- We can make a compromise with **Heun's method**
  - Take a step with explicit Euler to find an inexact solution $(t_{j+1}, s^{j+1})$
  - Use the mean of the derivatives at $(t_j, x^j)$ and $(t_{j+1}, s^{j+1})$
  - Follow this to get a better estimate:

$$s^{j+1} = x^j + hf(x^j, t_j)$$

$$x^{j+1} = x^j + h\frac{f(x^j, t_j) + f(s^{j+1}, t_{j+1})}{2}$$

- This is a *two-stage, explicit* method
  - Requires two step calculations
- An example of a Runge-Kutta method

# The form of a general ODE solver

- **Explicit Euler**, **implicit Euler** and **Heun's method** all belong to the class of **Runge–Kutta methods**
- Runge–Kutta methods can be labelled as
  - Explicit/implicit
    - Explicit methods are fast and compute straightforward, but have step size restrictions
    - Implicit methods are slower due to requiring equation solving, but are generally more stable. More suited for tough problems
  - $K$-stage
    - Need to calculate $K$ steps to get $x^{j+1}$
    - Like Heun's method, a 2-stage method
    - Explicit/implicit Euler are 1-stage methods

# Implementation of ODE solvers

1. Initialize variables $(N, T, h, x_0)$
2. A `for` loop going through $j$ from 1 to $N$
3. A function for taking time steps, depending on the method
   - Number of stages, implicit/explicit etc.

- Stopping condition?
  - Not necessary, we are taking N steps and specifying how far we want to go by that
  - …But there are methods that could require stopping conditions.
    - Adaptive methods (step sizes can vary from step to step)
    - Not curriculum, but you may encounter them e.g. in MATLAB
- Skeleton code: **ODE_solver_skeleton.py**

# Summary

- ODEs can be solved by numerical methods. We have seen three:
  - Explicit Euler is straightforward and simple, but not stable
    - Requires small time steps to work at all
  - Implicit Euler requires the solution of an equation every step, but is far more stable
    - No restrictions on time steps
  - Heun's method tries to improve upon these methods
    - It's an explicit two-stage method, has better stability properties than explicit Euler but not as good as the implicit Euler method
  - Lots of other ODE solver algorithms exist, e.g. **Runge-Kutta methods**
- Next time:
  - Generalizing to several dimensions (really easy, actually!)
  - Closer analysis of today's methods
    - Stability, accuracy etc.

# Questions?