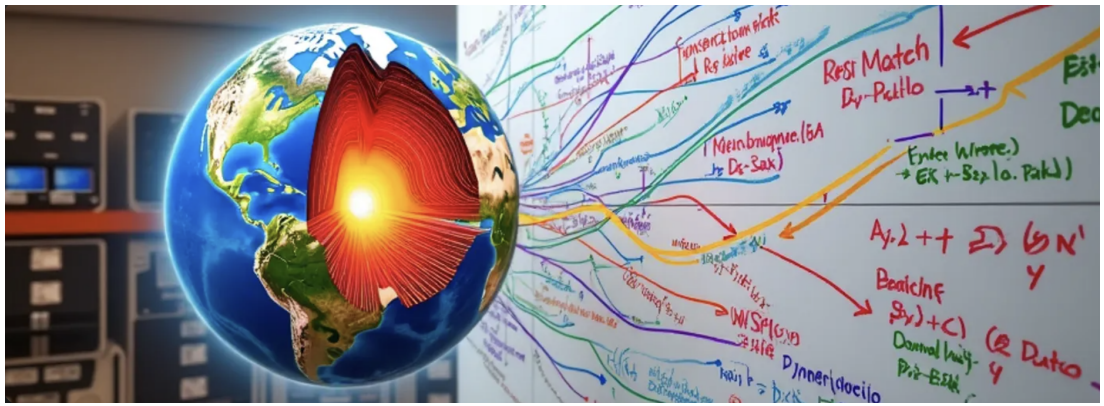
Principles of Numerical Modelling in Geosciences

Lecture 2 – Unfinished L1 Business, Implicit/Explicit for Nonlinear ODEs, and Finite Differences

Mattia de' Michieli Vitturi

PhD Course in Earth Sciences



Overview of Today's Lecture

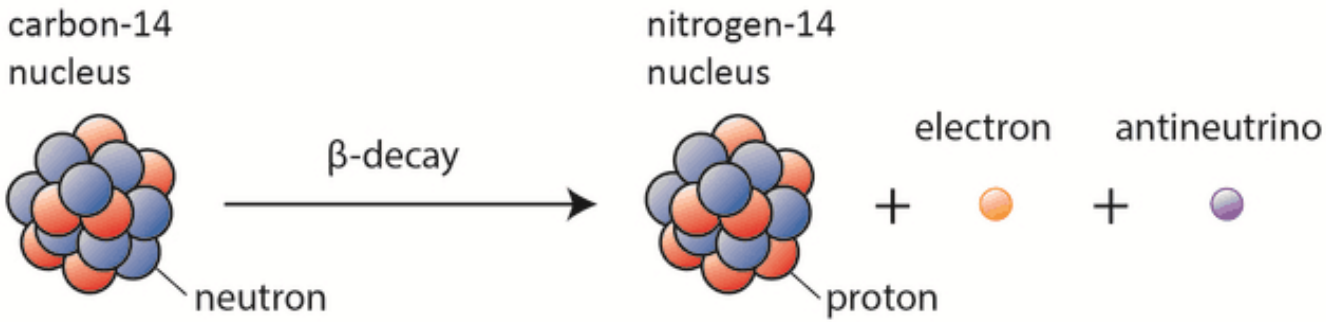
1. Recap and Continuation of Lecture 1
   - Numerical accuracy and stability (from Lecture 1)
2. Numerical accuracy and stability
3. Implicit vs Explicit for Nonlinear ODEs
4. Finite Difference Approximations
   - The Finite Difference Method (FDM)
   - Accuracy of Finite Difference Schemes
   - Finite Differences: Practical Examples and Order Verification
5. Summary

Outline

Continuing Lecture 1: Numerical Solutions

We will now pick up where we left off in Lecture 1, discussing numerical solutions for ODEs, their implementation, and aspects of accuracy and stability.

- Numerical solution of Radioactive Decay (Euler's Method)
- Explicit vs. Implicit Euler
- Numerical Accuracy and Stability
- Summary of Lecture 1



Numerical Solution of Radioactive Decay (2/2)

Rearranging the equation:

$$N^{n+1} = N^n - \lambda \Delta t N^n = N^n(1 - \lambda \Delta t)$$

This is the **explicit Euler method** for our problem.

Summary:

- Initial value: $N^0 = N_0$
- Update formula: $N^{n+1} = N^n(1 - \lambda \Delta t)$
- Iterate for $n = 0, 1, \ldots$

We will now implement this method in Python and compare the result with the analytical solution.

Python Implementation: Euler's Method

Numerical approximation of radioactive decay

Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
lambda_ = 0.1
N0 = 100
T = 50
dt = 1.0
time = np.arange(0, T + dt, dt)

# Numerical solution (Euler method)
N_euler = np.zeros_like(time)
N_euler[0] = N0
for n in range(len(time)-1):
    N_euler[n+1] = N_euler[n] * (1 - lambda_ * dt)
```

Plotting the Numerical Solution Only

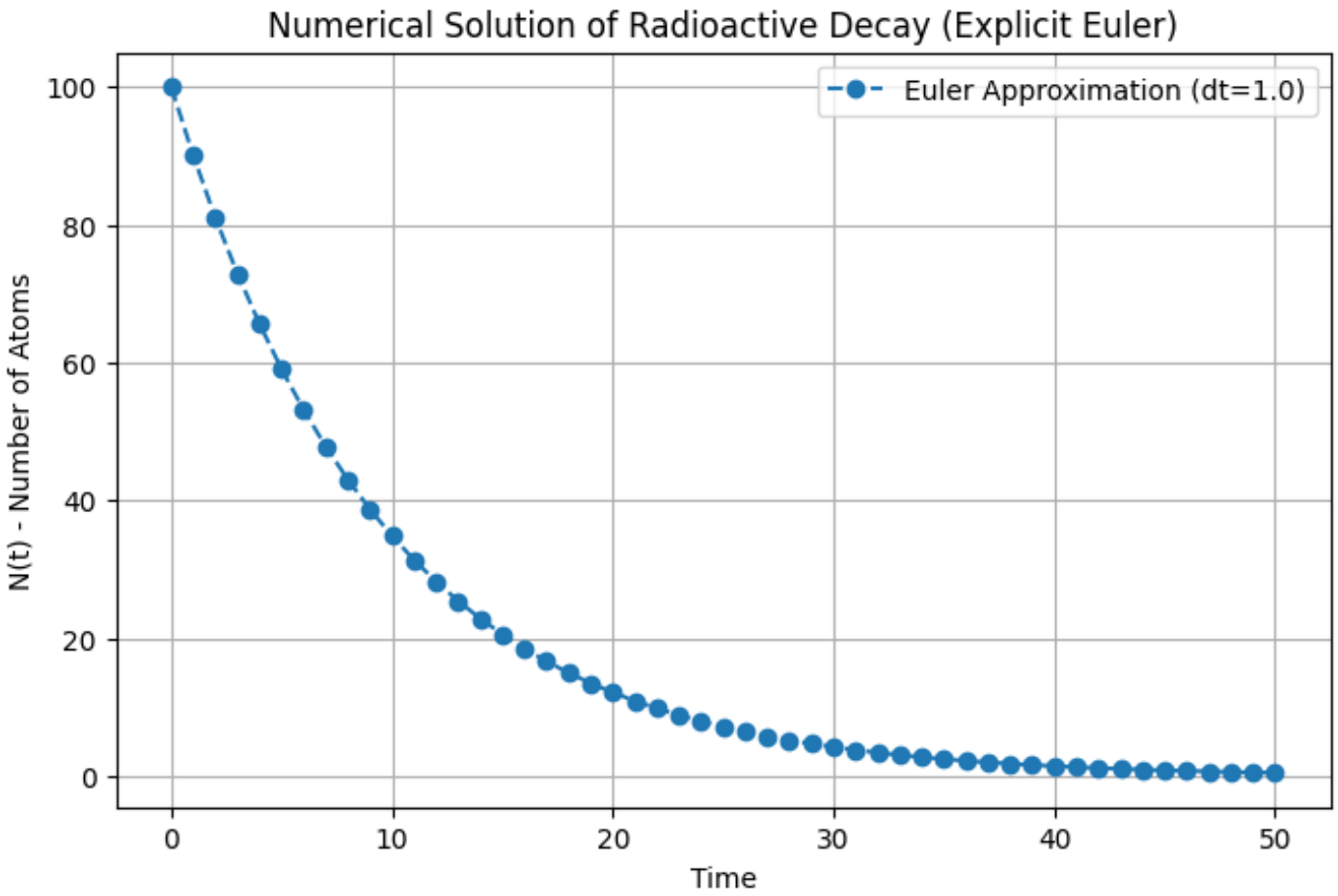
Euler's Method applied to radioactive decay

Python code

```
...
plt.figure(figsize=(8,4))
plt.plot(time, N_euler, 'o--', label='Euler Approximation')
plt.xlabel('Time')
plt.ylabel('N(t)')
plt.title('Numerical Solution of Radioactive Decay')
plt.grid(True)
plt.legend()
plt.show()
```

Plotting the Numerical Solution Only

Euler's Method applied to radioactive decay



Exercise: Try a Larger Time Step

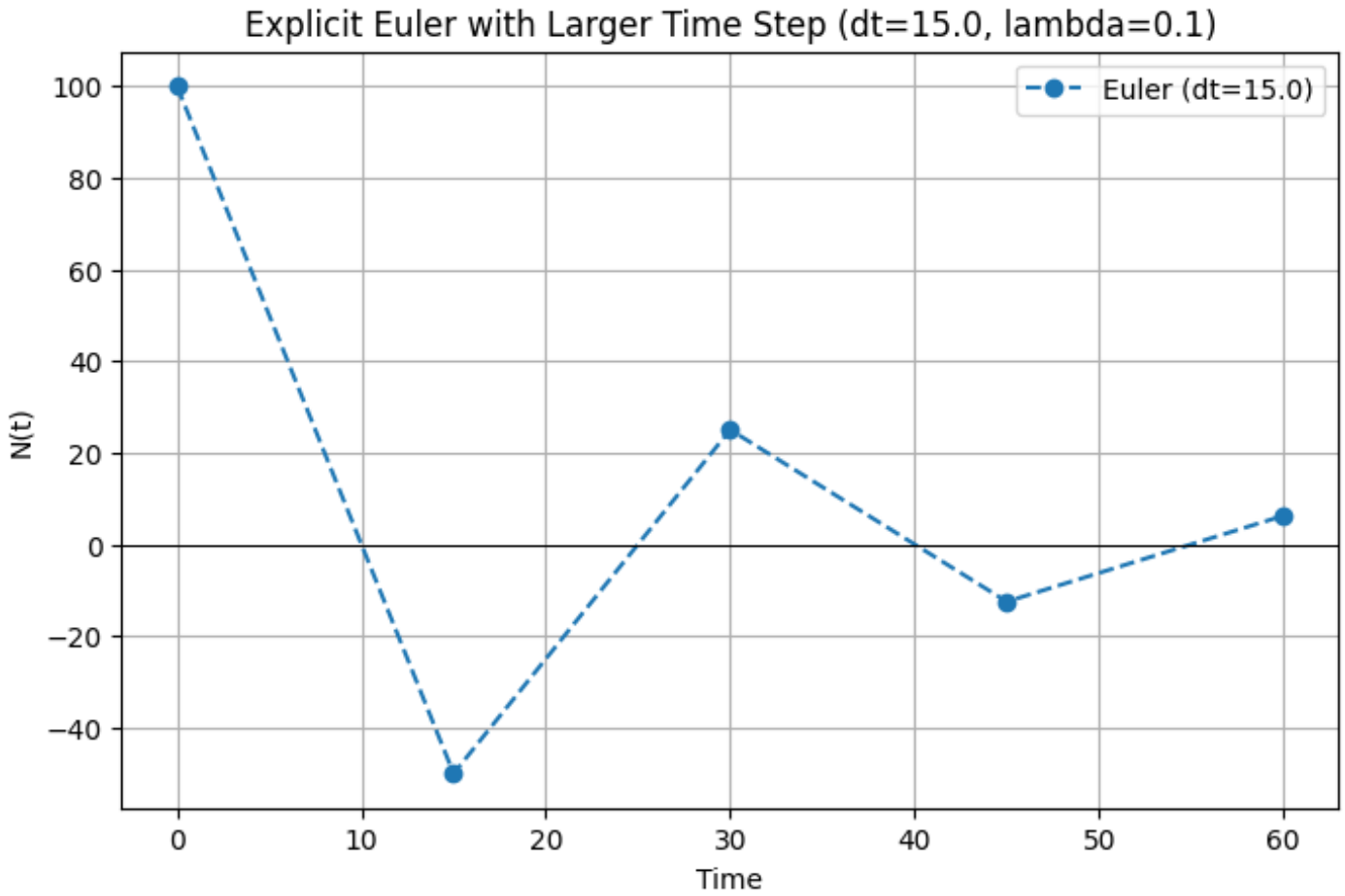
Change the time step and observe the result

Python code

```
dt = 15.0
time_large_dt = np.arange(0, T + dt, dt)
N_euler_large_dt = np.zeros_like(time_large_dt)
N_euler_large_dt[0] = N0
for n in range(len(time_large_dt)-1):
    N_euler_large_dt[n+1] = N_euler_large_dt[n]
                            * (1 - lambda_ * dt)

plt.plot(time_large_dt, N_euler_large_dt, 'o--')
plt.xlabel('Time'); plt.ylabel('N(t)')
plt.title('Euler Method with Larger Time Step (dt=5.0)')
plt.grid(True)
plt.show()
```

Exercise: Try a Larger Time Step

What happens to the solution? Does it remain physically meaningful?



Numerical Instability for Large Δt

Observations:

- With $\Delta t = 1.0$ the solution was, at least qualitatively, correct.
- With $\Delta t = 5.0$ we observe a strange behavior in the solution.

A large time step can lead to unphysical results

- When Δt is too large, $N(t)$ can become negative.
- This is clearly nonphysical: the amount of radioactive material cannot be less than zero.
- The update rule becomes unstable if:

$$1 - \lambda \Delta t < 0 \quad \Rightarrow \quad \Delta t > \frac{1}{\lambda}$$

(For $\lambda = 0.1$, this is $\Delta t > 10$).

This leads us to the concept of stability of a numerical scheme.

Interpreting the Explicit Euler Scheme

What does the discretization mean?

- Recall the ODE:

$$\frac{dN}{dt} = -\lambda N(t)$$

- The decay rate $-\lambda N(t)$ changes continuously with time.
- In the explicit Euler method:

$$N^{n+1} = N^n - (\lambda N^n)\Delta t$$

- This means we **assume the decay rate is constant** over the time step Δt, and equal to its value at the **beginning** of the interval $[t^n, t^{n+1}]$.

But is this the only possible choice?

An Alternative Assumption

What if we assume the rate is constant over $[t^n, t^{n+1}]$, but equal to its value at the end of the interval?

- That would lead to:

$$N^{n+1} = N^n - (\lambda N^{n+1})\Delta t$$

- This is known as the **Backward (Implicit) Euler Method**.
- Now the decay rate used in the update depends on the unknown future value N^{n+1}.
- Rearranging:

$$N^{n+1} = \frac{N^n}{1 + \lambda \Delta t}$$

- This scheme is *unconditionally stable*.

Choosing where to evaluate the rate leads to different numerical methods.

Deriving the Implicit Euler Method

Assumption: the decay rate is constant and equal to its value at t^{n+1}:

$$\frac{dN}{dt} \approx \frac{N^{n+1} - N^n}{\Delta t} = -\lambda N^{n+1}$$

Rearranging:

$$N^{n+1} - N^n = -\lambda \Delta t N^{n+1} \quad \Rightarrow \quad N^{n+1}(1 + \lambda \Delta t) = N^n$$

$$\Rightarrow \quad N^{n+1} = \frac{N^n}{1 + \lambda \Delta t}$$

Key features:

- Future value N^{n+1} appears on both sides (implicit).
- Stable for any Δt, even large ones (for this problem).

Backward Euler Method (Summary)

Alternative discretizations: explicit and implicit Euler method

- Recall the ODE:

$$\frac{dN}{dt} = -\lambda N$$

- Forward Euler (explicit):

$$N^{n+1} = N^n - \lambda \Delta t N^n = N^n(1 - \lambda \Delta t)$$

- Backward Euler (implicit):

$$N^{n+1} = N^n - \lambda \Delta t N^{n+1}$$

- Solve for N^{n+1}:

$$N^{n+1}(1 + \lambda \Delta t) = N^n \quad \Rightarrow \quad N^{n+1} = \frac{N^n}{1 + \lambda \Delta t}$$

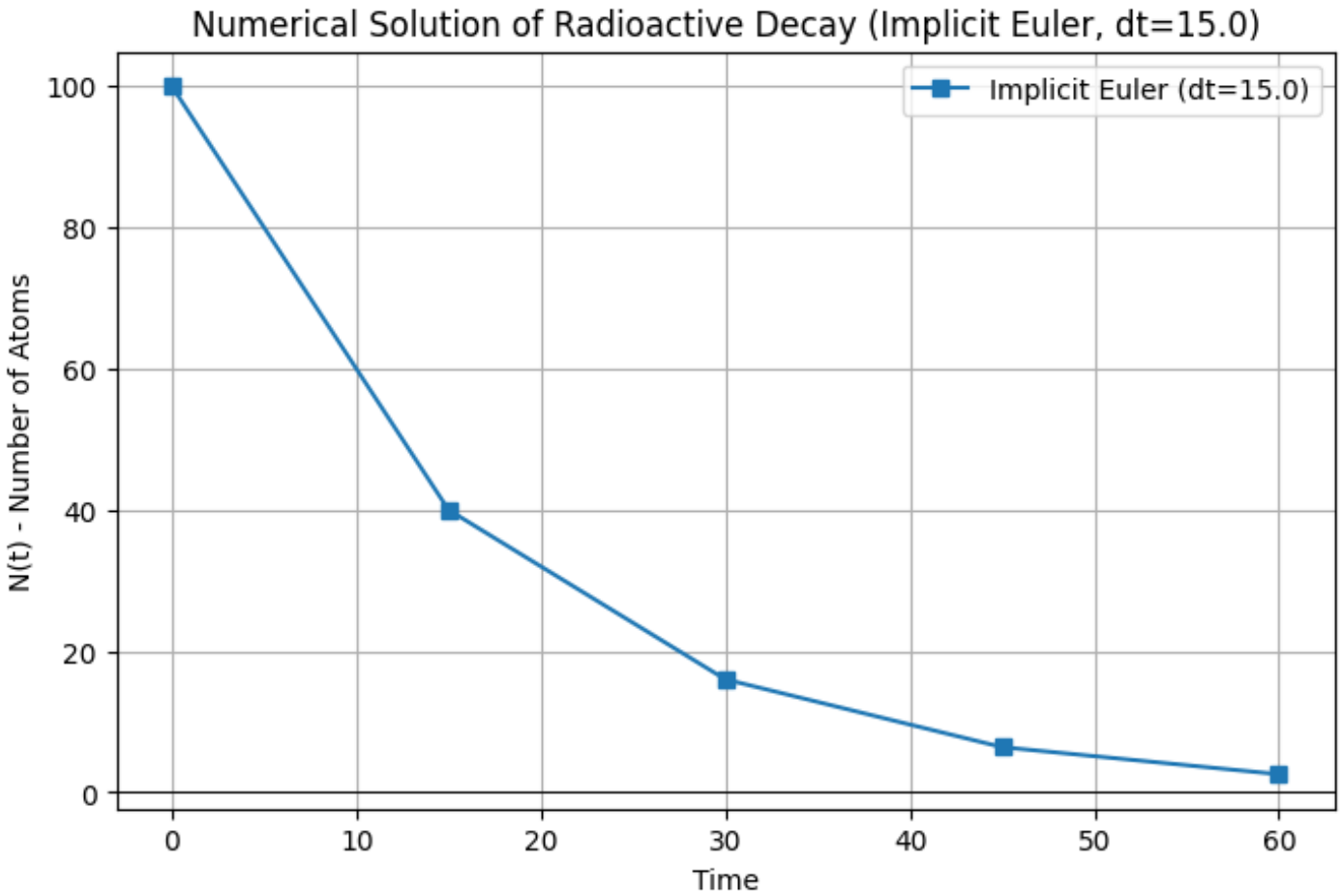
This is always stable (for this ODE), even for large time steps.

Implicit Euler in Python

Python code for Implicit Euler

```
import numpy as np
import matplotlib.pyplot as plt

lambda_val = 0.1 # Decay constant
dt = 15.0          # Time step
T = 60             # Total time
N0 = 100            # Initial quantity
times = np.arange(0, T + dt, dt)

N_implicit = np.zeros_like(times)
N_implicit[0] = N0
for n in range(len(times) - 1):
    N_implicit[n+1] = N_implicit[n] / (1 + lambda_val * dt)

plt.plot(times, N_implicit, 'o-', label='Implicit Euler')
plt.xlabel('Time'); plt.ylabel('N(t)')
plt.title('Implicit Euler for Radioactive Decay')
plt.legend(); plt.grid(True); plt.show()
```

Implicit Euler in Python



Exercises – Numerical Schemes for ODEs (from Lecture 1)

Exercise 1: Time Step Sensitivity (Implicit Euler)

- Use the implicit Euler method to solve the decay equation with:
  - $\Delta t = 5.0$, and $\Delta t = 20.0$
- Plot the solutions and comment on their behavior and stability.

Exercise 2: Explicit vs Implicit Euler

- Solve the same decay equation using both explicit and implicit Euler methods.
- Use different Δt values and compare:
  - Accuracy of the solution vs. the analytical one
  - Stability and qualitative behavior

Exercise 3: Apply to Newton's Law of Cooling

- Derive the discrete form of Newton's cooling law:

$$\frac{dT}{dt} = -k(T - T_{\text{env}})$$

- Implement both explicit and implicit Euler schemes.
- Choose parameters and compare the numerical results.

Outline

Analytical vs Numerical (Explicit Euler)

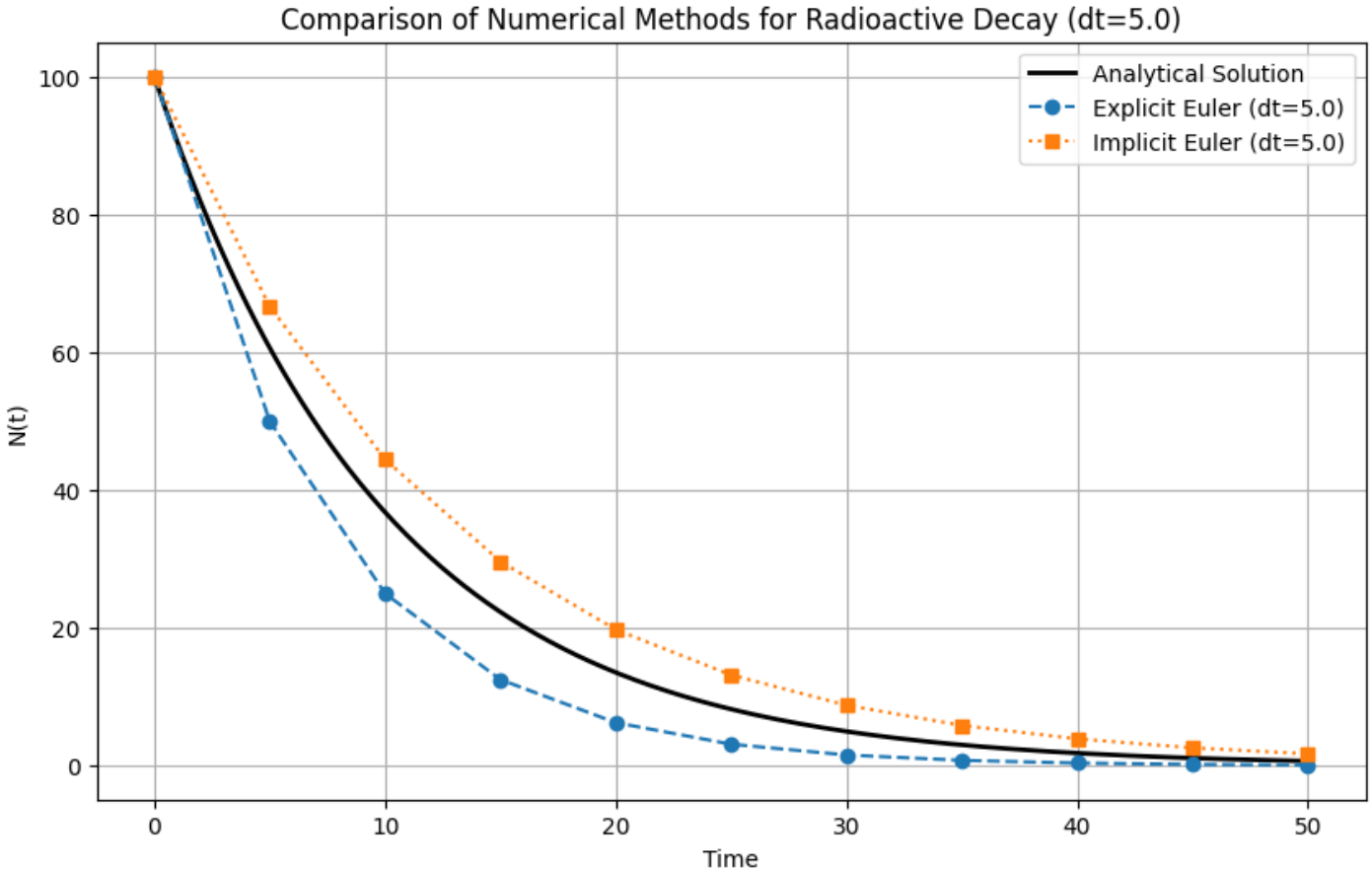
Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
lambda_ = 0.5
N0 = 100
t_max = 10
dt = 0.5
t_values = np.arange(0, t_max + dt, dt)

# Analytical solution
N_analytical = N0 * np.exp(-lambda_ * t_values)
```

Analytical vs Numerical (Explicit Euler)

```
# Explicit Euler method
N_explicit = [N0]
for n in range(1, len(t_values)):
    N_new = N_explicit[-1] - lambda_ * N_explicit[-1] * dt
    N_explicit.append(N_new)

# Plotting
plt.plot(t_values, N_analytical, label='Analytical', lw=2)
plt.plot(t_values, N_explicit, 'o--', label='Explicit Euler')
plt.xlabel('Time')
plt.ylabel('N(t)')
plt.legend()
plt.grid()
plt.title('Radioactive decay: Analytical vs Explicit Euler')
plt.show()
```

Analytical vs Numerical



Definition of Error

How can we assess the quality of a numerical solution?

We compare the numerical solution $N_n^{(num)}$ with the exact (analytical) solution $N(t_n)$.

Pointwise (absolute) error at time t_n:

$$\varepsilon_n = N_n^{(num)} - N(t_n)$$

- If the error is small, the numerical solution is a good approximation.
- If it grows over time or becomes large, the time step or numerical method may need adjustment.

Note: The error generally depends on:

- the time step size Δt
- the numerical method used
- the regularity (smoothness) of the true solution

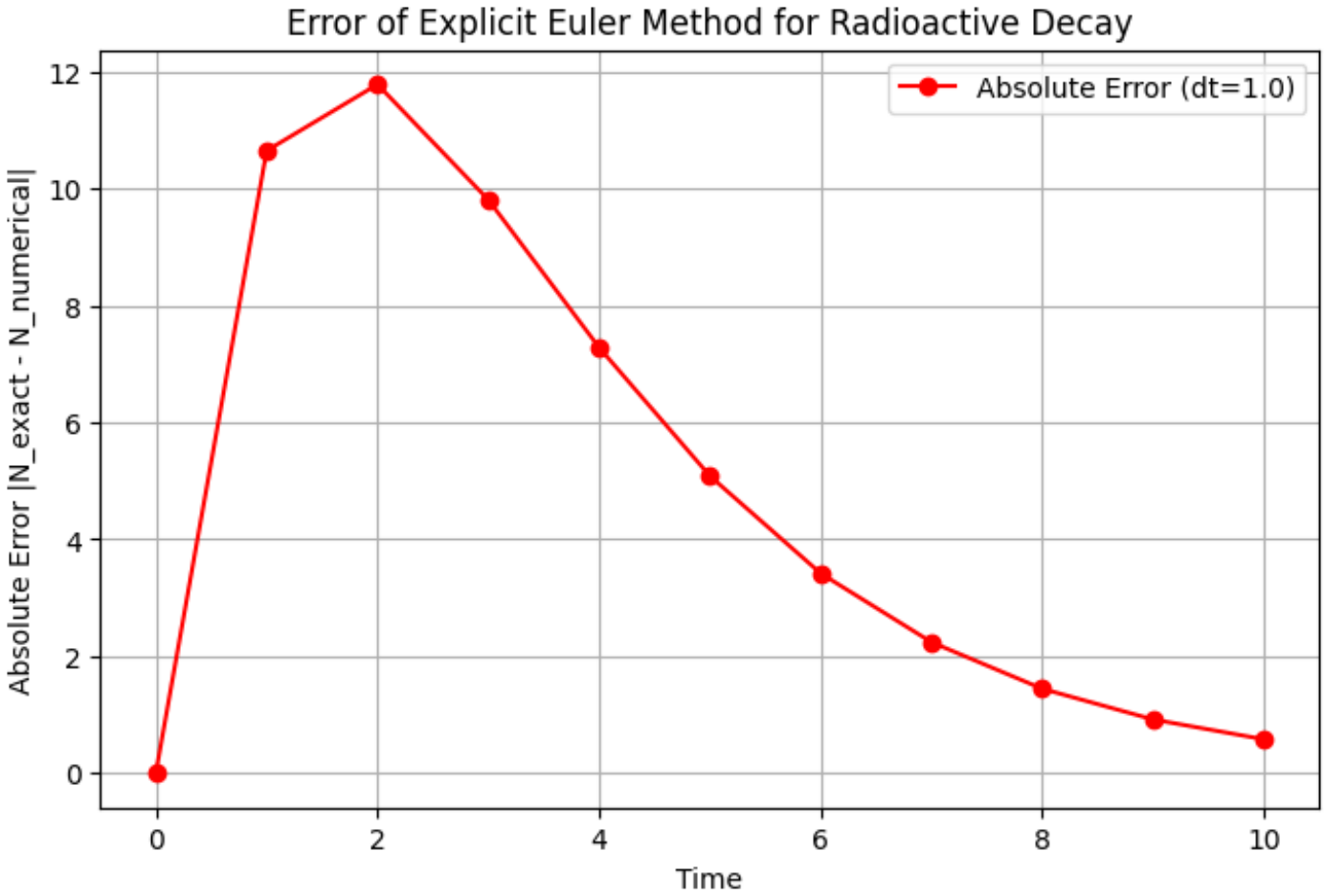
Plotting the Error in Python

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
lambda_ = 0.5
N0 = 100
T = 10
dt = 1.0
t = np.arange(0, T + dt, dt)

# Analytical solution
N_exact = N0 * np.exp(-lambda_ * t)
```

Plotting the Error in Python

```
# Explicit Euler method
N_num = np.zeros_like(t)
N_num[0] = N0
for n in range(1, len(t)):
    N_num[n] = N_num[n-1] - lambda_ * N_num[n-1] * dt

# Compute error
error = np.abs(N_exact - N_num)

# Plot
plt.plot(t, error, 'r-o', label='Absolute error')
plt.xlabel('Time')
plt.ylabel('Error')
plt.title('Error between analytical and numerical solution')
plt.legend()
plt.grid(True)
plt.show()
```

Plotting the Error in Python



Relative Error

Why consider relative error?

- The **absolute error** is defined as:

$$\epsilon_{\text{abs}}(t) = |N_{\text{exact}}(t) - N_{\text{num}}(t)|$$

- The **relative error** is:

$$\epsilon_{\text{rel}}(t) = \frac{|N_{\text{exact}}(t) - N_{\text{num}}(t)|}{|N_{\text{exact}}(t)|}$$

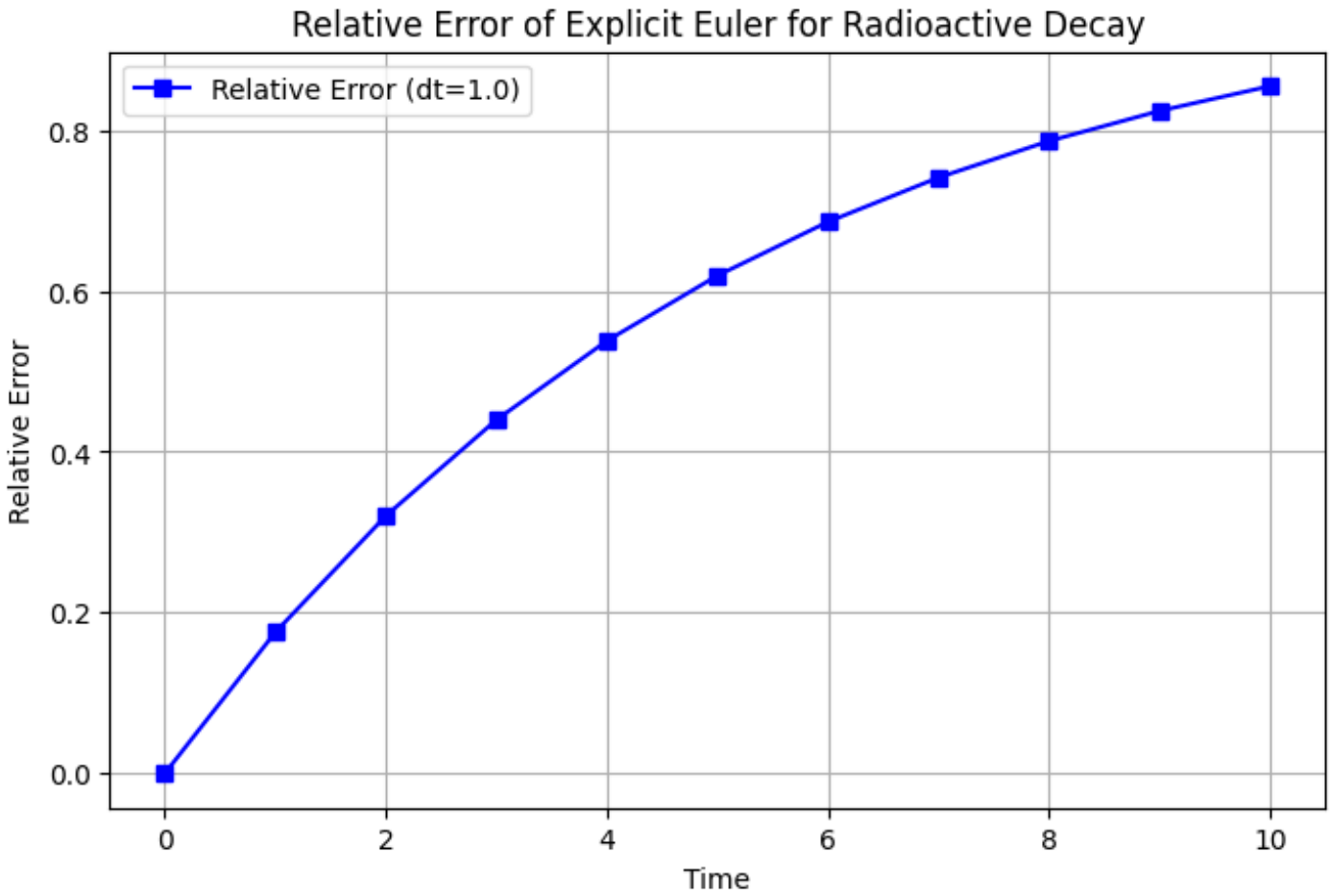
- Relative error is more informative when:
  - The quantity of interest becomes very small.
  - We want to assess the error proportionally to the expected value.
- Example: in radioactive decay, $N(t)$ tends to 0 $\rightarrow$ relative error becomes more relevant.

Plotting the Relative Error

Python Code

```
# Compute relative error, avoiding division by zero
rel_error = np.abs(N_exact - N_num) / np.abs(N_exact)
rel_error[N_exact == 0] = 0  # Handle division by zero safely

# Plot
plt.plot(t, rel_error, 'b-s', label='Relative error')
plt.xlabel('Time')
plt.ylabel('Relative Error')
plt.title('Relative Error between Analytical and Numerical Solution
plt.legend()
plt.grid(True)
plt.show()
```

Plotting the Relative Error



Numerical Accuracy

What is numerical accuracy?

Accuracy refers to how close the numerical solution is to the exact (analytical) solution.

- With the explicit Euler method, reducing Δt improves the accuracy.
- We saw that for smaller time steps, the numerical curve approached the exact solution more closely.

Important:

- Accuracy improves with smaller Δt, but at the cost of more computations.
- Accuracy is influenced by the method's **order**: Euler's method is *first-order accurate*.

Numerical Stability

What is numerical stability?

A numerical method is **stable** if errors (from rounding, approximation, etc.) do not grow uncontrollably as the computation proceeds.

- In the radioactive decay example, using a large time step with the explicit Euler method caused the solution to become negative or oscillate.
- This indicates **instability** of the explicit method for large Δt.
- Stability often depends on the time step size Δt and the method used.

Key idea: Even with a consistent and accurate method, instability can make the numerical solution completely unreliable.

Outline

Explicit vs Implicit: Recap

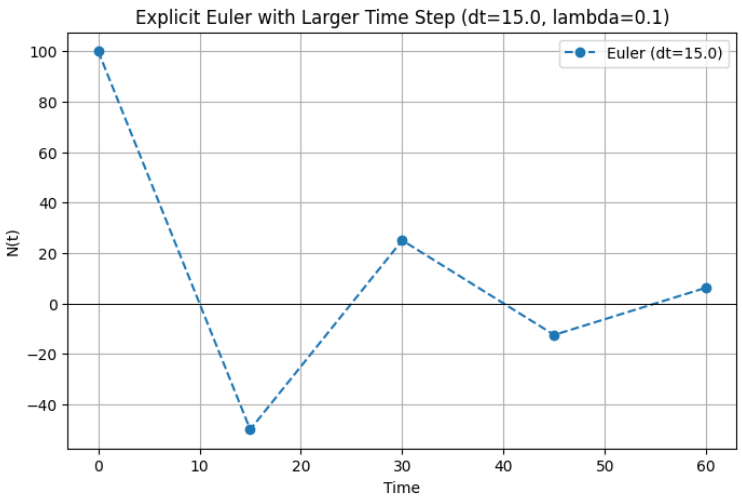
In the previous slides, we implemented both explicit and implicit time integration schemes:

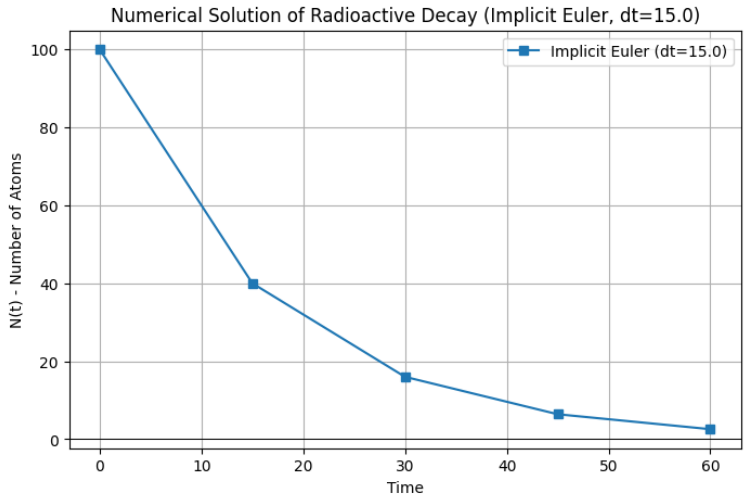
- **Explicit (Euler)**: the future value depends only on known quantities.
- **Implicit (Backward Euler)**: the future value appears on both sides of the equation.

We saw that:

- Explicit schemes are simple but conditionally stable.
- Implicit schemes are more robust and can be unconditionally stable.
- For simple ODEs (e.g., exponential decay), implementing both was straightforward.

A Natural Question





Should we always use implicit schemes?

They are more stable... so why not always use them?

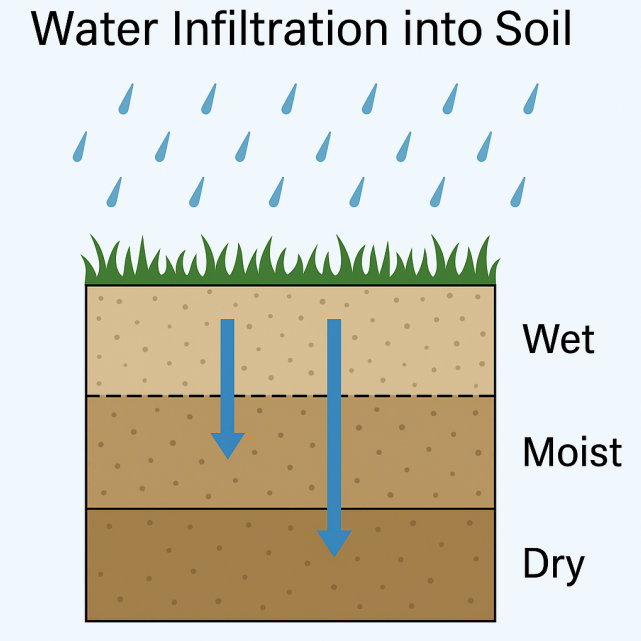
In this part of the lecture, we will look at an example where:

- the governing ODE is **nonlinear**,
- and implementing an implicit scheme requires more effort.

Water Infiltration into Soil

One common process in Earth sciences is the infiltration of rainwater into unsaturated soil.

- This process is described by **Richards' equation**, a nonlinear PDE.
- Here we consider a simplified, lumped version: spatially homogeneous, only time-dependent.
- We model the volumetric water content $\theta(t)$ as it decreases due to percolation.



Reference: Hillel, D. (1998). *Environmental Soil Physics*. Academic Press.

Water content is expressed as a ratio, which can range from 0 (completely dry) to the value of the materials' porosity at saturation.

A Simplified Model for Water Infiltration

We assume that water is lost from the soil due to gravity-driven flow, with a conductivity depending on the current water content:

$$\frac{d\theta}{dt} = -K(\theta)$$

A commonly used empirical expression for the hydraulic conductivity is:

$$K(\theta) = K_s \cdot \theta^n$$

- K_s: saturated conductivity (e.g., 10^{-5} to 10^{-6} m/s)
- $n > 1$: nonlinearity parameter (e.g., $n = 3$ or 4)

Explicit Euler Scheme for Infiltration

We discretize using the forward Euler method:

$$\theta^{k+1} = \theta^k - \Delta t \cdot K_s \cdot (\theta^k)^n$$

Python Code

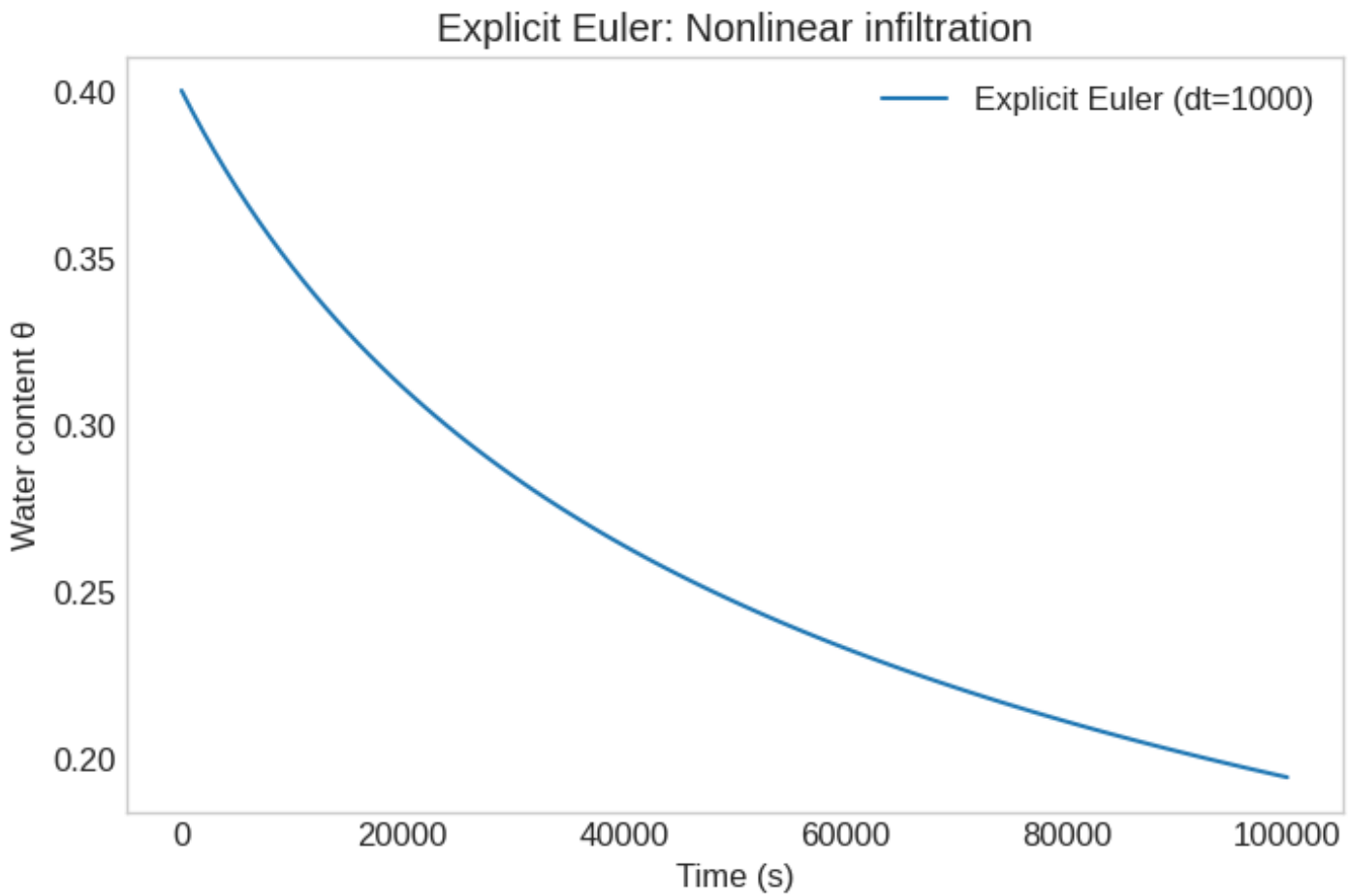
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
Ks = 1e-4          # Saturated hydraulic conductivity
n_exp = 3          # Nonlinearity exponent
dt = 1000           # Time step (s)
Tmax = 100000       # Total time
theta0 = 0.4       # Initial water content
```

Explicit Euler Scheme for Infiltration

Python Code

```
t = np.arange(0., Tmax + dt, dt)
theta = np.zeros_like(t)
theta[0] = theta0

# Time integration
for k in range(1, len(t)):
    theta[k] = theta[k-1] - dt * Ks * theta[k-1]**n_exp

# Plot
plt.plot(t, theta)
plt.xlabel("Time (s)")
plt.ylabel("Water content ")
plt.title("Explicit Euler: Nonlinear infiltration")
plt.grid(); plt.show()
```

Explicit Euler Scheme for Infiltration



Implicit Scheme (Backward Euler) for Infiltration

The backward Euler scheme for $\frac{d\theta}{dt} = -K_s\theta^n$ reads:

$$\theta^{k+1} = \theta^k - \Delta t \cdot K_s \cdot (\theta^{k+1})^n$$

Here, θ^{k+1} is the unknown water content at the next time step. Unlike the linear case, we cannot simply rearrange to solve for θ^{k+1} directly because it appears on both sides, and one term is $(\theta^{k+1})^n$. To solve for θ^{k+1}, let $X = \theta^{k+1}$. We need to find X such that:

$$X = \theta^k - \Delta t \cdot K_s \cdot X^n$$

Rearranging all terms to one side, we get a function $G(X)$ which we want to be zero:

$$G(X) = X + \Delta t \cdot K_s \cdot X^n - \theta^k = 0$$

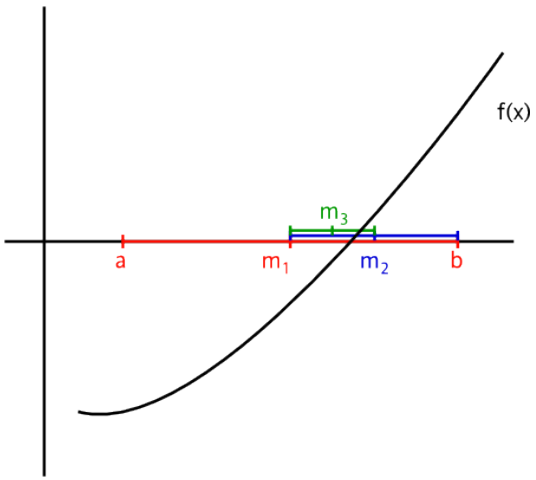
This is a **nonlinear algebraic equation** for $X = \theta^{k+1}$. We need a numerical method to find the root of $G(X)$.

Root-Finding: The Bisection Method - Idea

The **bisection method** is a simple and robust numerical technique for finding a root of a continuous function $G(X)$.

Core Idea:

- Start with an interval $[a, b]$ such that $G(a)$ and $G(b)$ have *opposite signs* (i.e., $G(a) \cdot G(b) < 0$).
- By the *Intermediate Value Theorem*, if $G(X)$ is continuous, there must be at least one root X^* in (a, b).
- The method repeatedly halves the interval while keeping the subinterval where the sign change occurs.



Steps visually:

1. Find a, b so $G(a)G(b) < 0$.
2. Calculate midpoint $c = (a + b)/2$.
3. If $G(c) \approx 0$, c is the root.
4. If $G(a)G(c) < 0$, root is in $[a, c]$.
5. Else, root is in $[c, b]$.
6. Repeat with new interval.

The Bisection Method - Algorithm

Given a function $G(X)$ and an initial interval $[a_0, b_0]$ such that $G(a_0)G(b_0) < 0$, and a desired tolerance ϵ:

1. **Initialize:** Set $a = a_0$, $b = b_0$.
2. **Iterate** (e.g., for a max number of iterations or until convergence):
   1. Calculate the midpoint: $c = (a + b)/2$.
   2. Evaluate $G(c)$.
   3. **Check for convergence:**
      - If $|G(c)| < \epsilon_F$ (function value tolerance), OR
      - If $(b - a)/2 < \epsilon_X$ (interval width tolerance),
      - then c is our approximate root. Stop.
   4. **Update interval:**
      - If $G(a) \cdot G(c) < 0$, then the root is in $[a, c]$. Set $b = c$.
      - Else (if $G(c) \cdot G(b) < 0$), the root is in $[c, b]$. Set $a = c$.
      - (If $G(c) = 0$ exactly, c is the root, stop. This is rare with floating point numbers).
3. If max iterations reached without convergence, report failure.

The bisection method is guaranteed to converge, but can be slower than other methods like Newton-Raphson.

Why Python Functions for Root-Finding?

The bisection algorithm requires evaluating the function $G(X)$ (and potentially its derivative for other methods) many times within a loop.

- Defining $G(X)$ as a Python **function** makes the code cleaner, reusable, and easier to understand.
- It separates the logic of "what function to solve" from "how to solve it".

In Python a function is defined using the *def* keyword.

Defining a function in Python.

```
def function_name(parameter1, parameter2, ...):
    """Optional: Docstring explaining what the function
    does."""
    # Body of the function: calculations
    result = parameter1 + parameter2 # Example
    return result # Value returned by the function
```

Why Python Functions for Root-Finding?

Let see, as example, how we can create a function for the sum of two numbers.

- Input parameters: x, y;
- Output: $x + y$.

Example.

```
def add_numbers(x, y):
    """This function returns the sum of x and y."""
    sum_val = x + y
    return sum_val

# How to use (call) the function:
z = add_numbers(5, 3)  # z will be 8
print(z)
```

Implicit Infiltration: Python Functions for Bisection

For our problem, $G(X) = X + \Delta t \cdot K_s \cdot X^n - \theta^k = 0$. Let $X = \theta^{k+1}$ (current guess for next theta) and $\theta_{prev} = \theta^k$.

1. Function to Solve: $G(X)$

```
def G_infiltration(X_current, theta_prev, dt, Ks, n_exponent
    ):
    """
    Calculates G(X) = X + dt*Ks*X^n - theta_prev for root
        finding.
    X_current:  The current guess for theta^{k+1}.
    theta_prev: Water content from previous step, theta^k.
    dt: Time step.
    Ks: Saturated hydraulic conductivity.
    n_exponent: Nonlinearity exponent.
    """
    term_nonlinear = dt * Ks * (X_current ** n_exponent)
    return X_current + term_nonlinear - theta_prev
```

Implicit Infiltration: Python Functions for Bisection

2. Bisection Solver Function - Part 1

```
def bisection_solver(func_G, a, b, tol=1e-7, max_iter=100,
                     args_for_G=()):
    """
    Finds a root of func_G(x, *args_for_G) = 0 in [a,b] by
        bisection.
    args_for_G: A tuple of additional fixed arguments for
        func_G.
    """
    Ga = func_G(a, *args_for_G)
    Gb = func_G(b, *args_for_G)

    if Ga * Gb >= 0:
        raise ValueError("G(a) and G(b) must have opposite
            signs.")

    for _ in range(max_iter):
        c = (a + b) / 2.0
        Gc = func_G(c, *args_for_G)
```

Implicit Infiltration: Python Functions for Bisection

2. Bisection Solver Function - Part 2

```
        if abs(Gc) < tol or (b - a) / 2.0 < tol:
            return c # Converged

        if Ga * Gc < 0:
            b = c
            # Gb = Gc # Not strictly needed but good
                practice if Gc is stored
        else:
            a = c
            Ga = Gc # Update Ga to the G(c) of the new 'a'

    raise RuntimeError("Bisection did not converge within
        max_iter.")
```

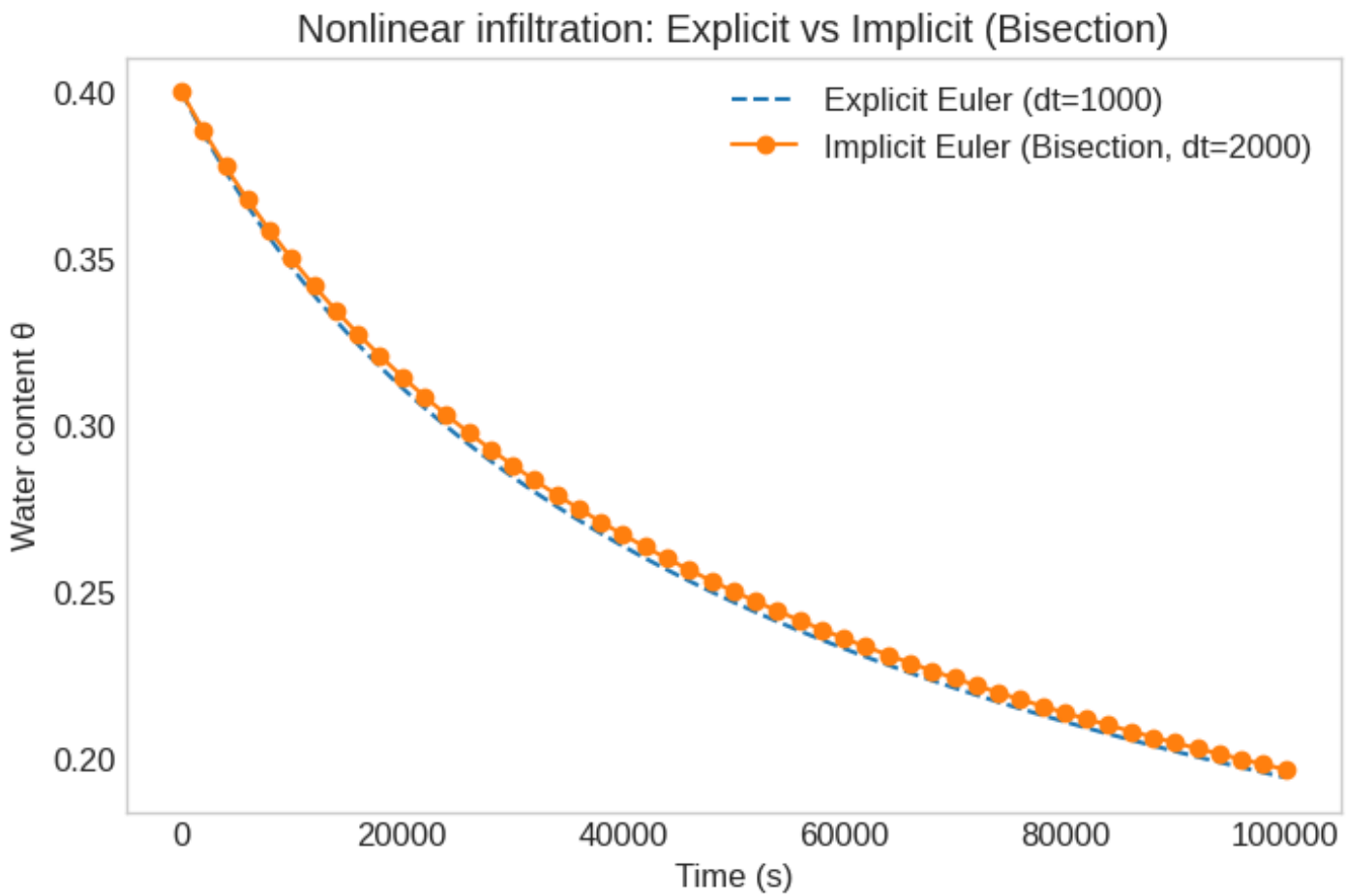
Implicit Euler with Bisection: Python Implementation

Algorithm Steps for each time step k:

1. We know $\theta[k-1]$ (i.e., θ^k).
2. We want to find $\theta[k]$ (i.e., θ^{k+1}).
3. The function to solve is $G(X) = X + \Delta t K_s X^n - \theta[k-1] = 0$.
4. Choose an interval $[a, b]$ for X. For physical reasons (water content decreases but ≥ 0), $a = 0$ and $b = \theta[k-1]$ is a good choice.
   - $G(0) = -\theta[k-1] \leq 0$.
   - $G(\theta[k-1]) = \Delta t K_s (\theta[k-1])^n \geq 0$.
   - So if $\theta[k-1] > 0$, signs are opposite.
5. Use 'bisection_solver' to find $X = \theta[k]$ that makes $G(X) \approx 0$.

Implicit Euler with Bisection: Python Implementation

See the code in the Jupyter Notebook.

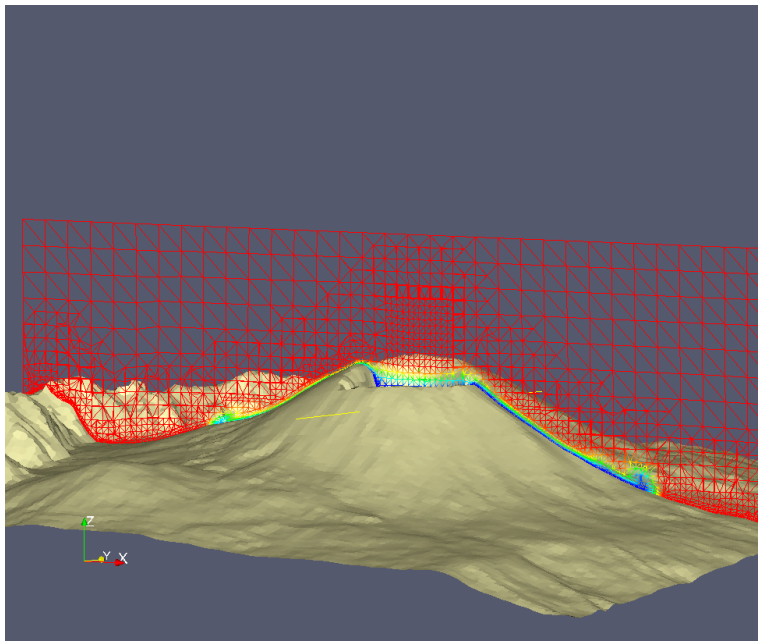


Outline

Numerical methods: General Idea

When we want to solve a mathematical model with a computer (we search for a numerical solution) we have to consider that computers have a finite amount of memory. We have then to select a set of discrete locations in space and/or time where the numerical solution is computed.

The discrete locations at which the variables are to be calculated are defined by a numerical grid, which is essentially a discrete representation of the geometric domain where the problem has to be solved.



Thus, our numerical solution will always be an approximation of the real solution, defined everywhere in the original domain.

From Differential Equations to Approximating Derivatives

So far we have already used numerical grids when we have solved numerically some ODEs.

In Lecture 1, and earlier in this lecture, we've seen that many natural processes in Earth Sciences are described by **differential equations**.

- Radioactive Decay: $\frac{dN}{dt} = -\lambda N$
- Newton's Law of Cooling: $\frac{dT}{dt} = -k(T - T_a)$
- Water Infiltration (simplified): $\frac{d\theta}{dt} = -K_s\theta^n$

These equations involve derivatives, which represent rates of change. To solve these equations numerically, when analytical solutions are not available or practical, we need ways to **approximate these derivatives** using discrete values of our variables (e.g., N, T, θ) at specific points in time or space.

Recalling Time Derivatives: Our First Finite Differences

In the next slides, we will formalize some key elements of the **Finite Differences schemes**, used to approximate the derivatives.
We have already encountered the core idea of finite differences when discretizing the **time derivative** $\frac{dy}{dt}$ in our ODE solvers:
Explicit (Forward) Euler for $\frac{dy}{dt} = f(y,t)$**:**

$$\frac{y^{k+1} - y^k}{\Delta t} \approx f(y^k, t^k)$$

Here, $\frac{y^{k+1}-y^k}{\Delta t}$ is a *forward difference approximation* of $\frac{dy}{dt}$ at time t^k. It uses values at t^k and t^{k+1}.
Implicit (Backward) Euler for $\frac{dy}{dt} = f(y,t)$**:**

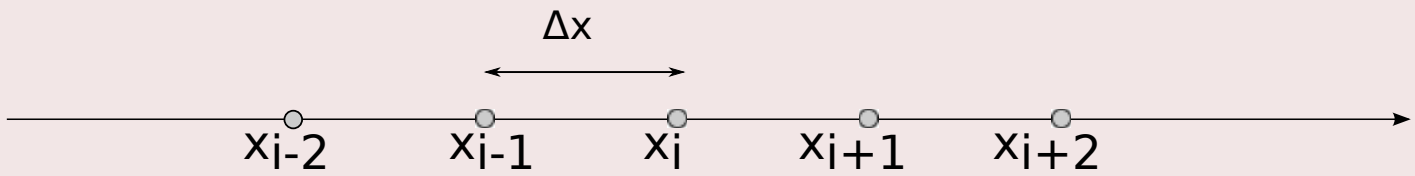
$$\frac{y^{k+1} - y^k}{\Delta t} \approx f(y^{k+1}, t^{k+1})$$

Here, $\frac{y^{k+1}-y^k}{\Delta t}$ is still an approximation of $\frac{dy}{dt}$ (evaluated conceptually around t^{k+1} or over the interval).

Finite Difference Method: Introduction

- Introduced by Euler in the 18th century.
- Governing equations are in differential form; replacing the partial derivatives with approximations in terms of node values of the functions leads to an algebraic equation for each grid node, and then to a system of algebraic equations.
- Generally applied to structured grids.
- The most attractive feature of finite differences is that they can be very easy to implement.

For convenience, in the description of the method, we will use a uniform 1D grid with spacing Δx. We will refer to the points of the grid as x_i.

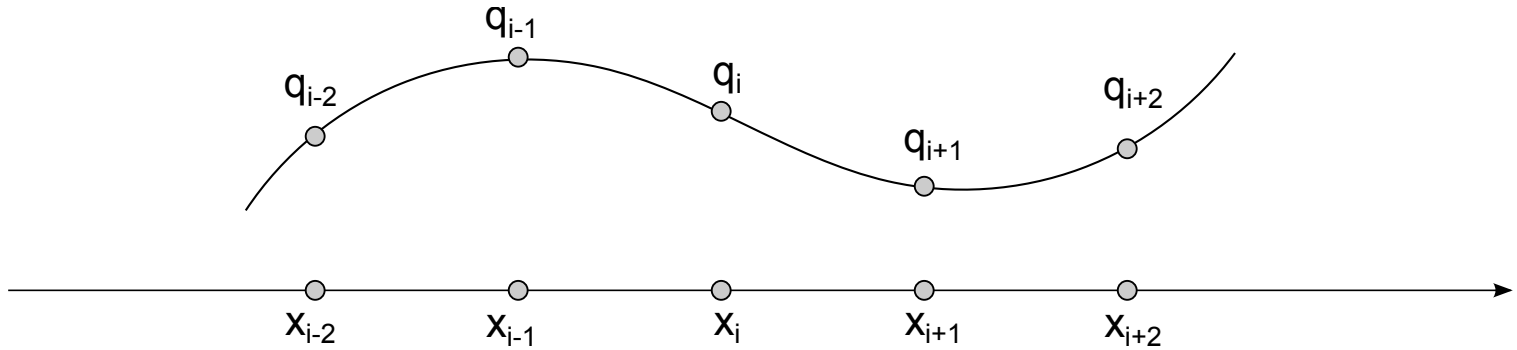


Finite Difference Method: Approximating Derivatives

In the finite difference method, at each grid point x_i, the differential equation is solved numerically by replacing derivatives with approximations in terms of the nodal values of the function $q(x)$:

$$\left(\frac{\partial q}{\partial x}\right)_{x_i} \approx F(\ldots, q_{i-1}, q_i, q_{i+1}, \ldots)$$

where $q_j = q(x_j)$ are the (possibly unknown) values of the function at the points x_j of the numerical grid.



Finite Difference Method: Foundation

The idea behind finite difference approximation is borrowed directly from the definition of a derivative:

$$\left(\frac{dq}{dx}\right)_{x_i} = \lim_{\delta x \to 0} \frac{q(x_i + \delta x) - q(x_i)}{\delta x}$$

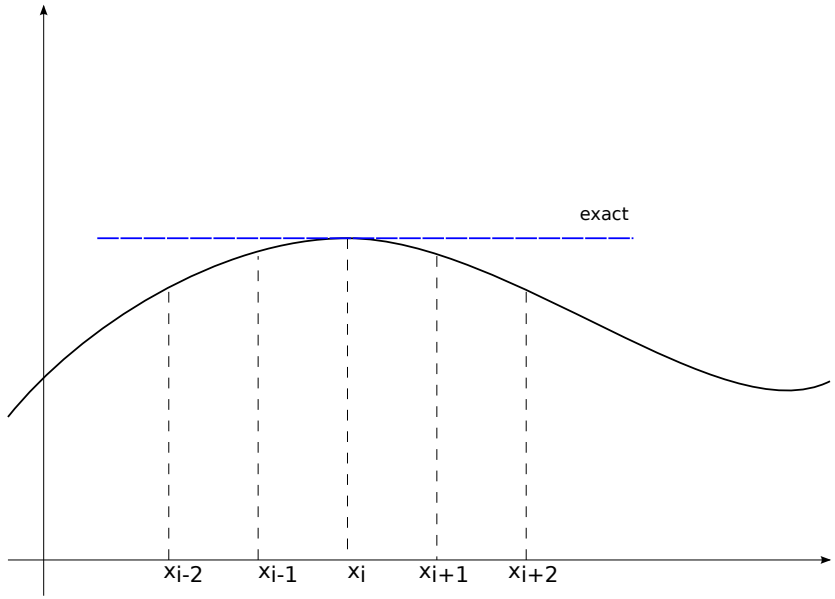
We replace the infinitesimal δx with a finite difference Δx related to our grid spacing.

Time Discretization Reminder

Likewise, for time-dependent problems, we discretize the time interval into discrete times $t_k = k\Delta t$, separated by a time step Δt. We compute the solution only at these times t_k.

Approximating the First Derivative

We want to approximate the exact first derivative (the slope) of a function $q(x)$ at a grid point x_i, knowing the values of the function at various grid points.

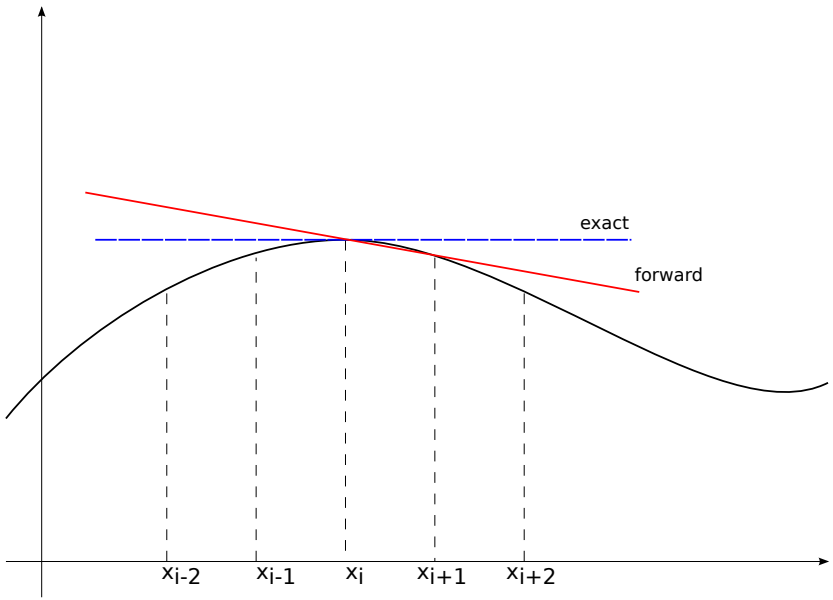


There are several common ways to do this.

Forward Difference Scheme

The value of the solution at the current node x_i and at the next node x_{i+1} (downwind if x increases) are used.

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_i + \Delta x) - q(x_i)}{\Delta x} = \frac{q(x_{i+1}) - q(x_i)}{\Delta x} = \frac{q_{i+1} - q_i}{\Delta x}$$

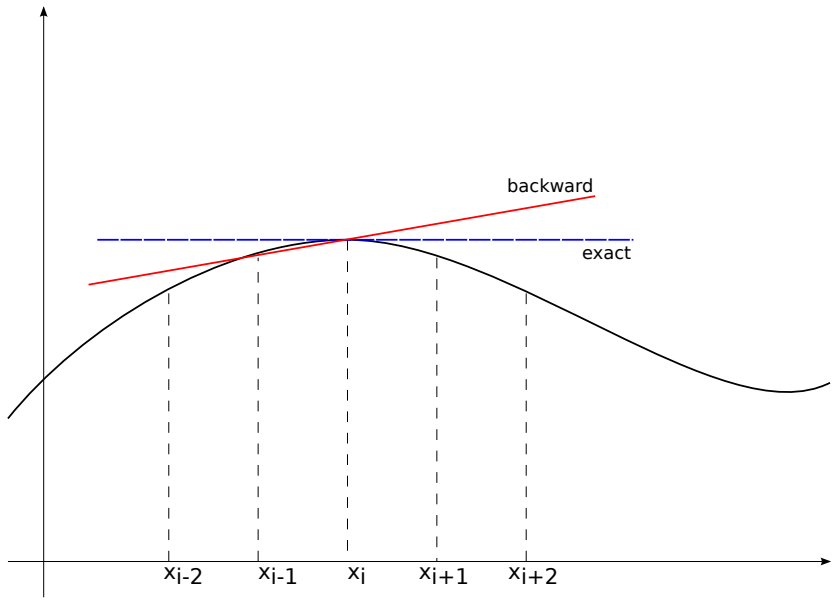


This is called a first-order forward difference.

Backward Difference Scheme

The value of the solution at the current node x_i and at the previous node x_{i-1} (upwind if x increases) are used.

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_i) - q(x_i - \Delta x)}{\Delta x} = \frac{q(x_i) - q(x_{i-1})}{\Delta x} = \frac{q_i - q_{i-1}}{\Delta x}$$

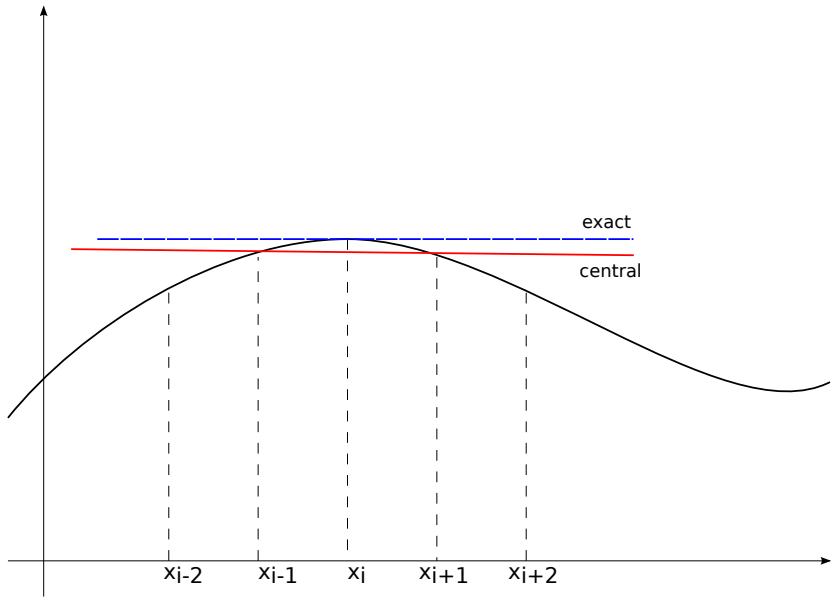


This is called a first-order backward difference.

Central Difference Scheme

The values of the solution at the nodes before (x_{i-1}) and after (x_{i+1}) the current node x_i are used.

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_{i+1}) - q(x_{i-1})}{2\Delta x} = \frac{q_{i+1} - q_{i-1}}{2\Delta x}$$



This is called a second-order central difference.

Accuracy and Truncation Error

It is obvious that some approximations are better than others. The accuracy of the approximation depends on the truncation error, i.e., the difference between the exact derivative and the finite difference approximation. For example, for the forward difference:

$$\frac{q(x_{i+1}) - q(x_i)}{\Delta x} = \left(\frac{dq}{dx}\right)_{x_i} + \text{Truncation Error}$$

The truncation error arises because we have truncated a Taylor series expansion.

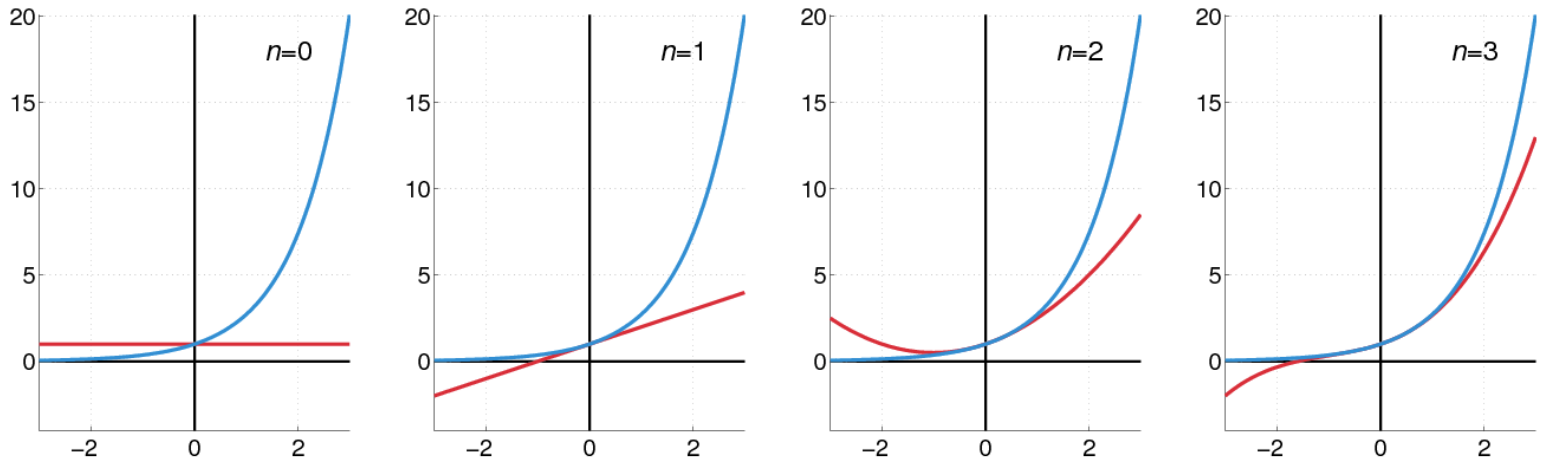
Exercise 1 (from old lecture)

Evaluate the truncation error of the forward, backward, and central finite difference approximations of the derivative of the function $q(x) = x^2$ at the point $x = 1$, using a numerical grid with $\Delta x = 1$ (i.e., points $x_0 = 0, x_1 = 1, x_2 = 2, \dots$).

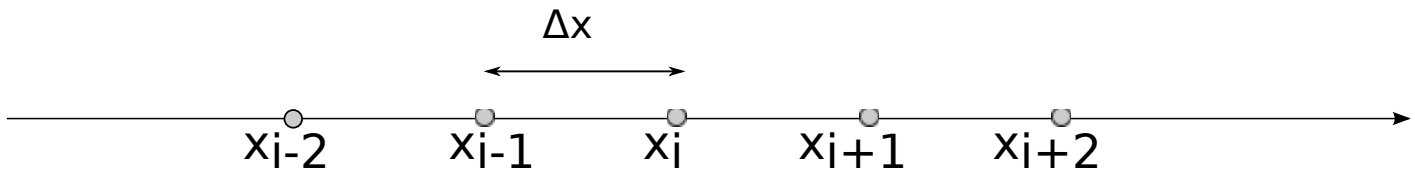
Taylor Series Expansion

To evaluate these errors systematically, we use the Taylor series expansion of $q(x)$ in a point x around x_i:

$$q(x) = q(x_i) + (x - x_i)\left(\frac{dq}{dx}\right)_{x_i} + \frac{(x - x_i)^2}{2!}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \frac{(x - x_i)^3}{3!}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \cdots + \frac{(x - x_i)^k}{k!}\left(\frac{d^kq}{dx^k}\right)_{x_i} + \mathcal{O}((x - x_i)^$$



Taylor Series Expansion



Specifically, for $x = x_{i+1} = x_i + \Delta x$:

$$q(x_{i+1}) = q(x_i) + \Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)$$

And for $x = x_{i-1} = x_i - \Delta x$:

$$q(x_{i-1}) = q(x_i) - \Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} - \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)$$

The notation $\mathcal{O}(\Delta x^k)$ means "terms of order Δx^k and higher".

Truncation Error: Forward Difference

From the Taylor expansion for $q(x_{i+1})$:

$$q(x_{i+1}) = q(x_i) + \Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)$$

Rearranging for the derivative term:

$$\left(\frac{dq}{dx}\right)_{x_i} = \frac{q(x_{i+1}) - q(x_i)}{\Delta x} - \underbrace{\left[\frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \frac{\Delta x^2}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^3)\right]}_{\text{Truncation Error}}$$

So, the forward difference approximation is:

$$\frac{q_{i+1} - q_i}{\Delta x} = \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \mathcal{O}(\Delta x^2)$$

The leading term of the truncation error is proportional to Δx. Thus, the forward difference approximation is **first-order accurate**, denoted $\mathcal{O}(\Delta x)$.

Truncation Error: Backward Difference

From the Taylor expansion for $q(x_{i-1})$:

$$q(x_{i-1}) = q(x_i) - \Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} - \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)$$

Rearranging for the derivative term:

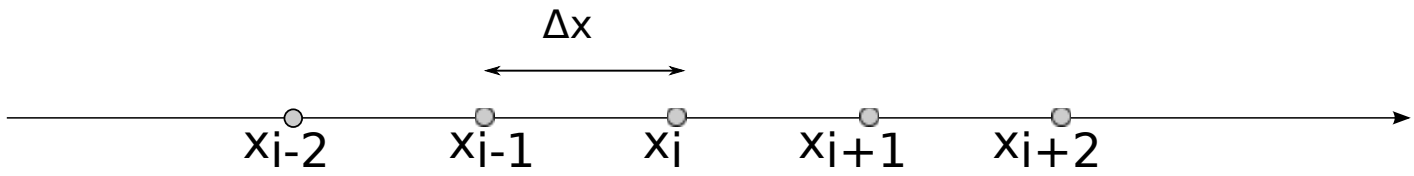
$$\left(\frac{dq}{dx}\right)_{x_i} = \frac{q(x_i) - q(x_{i-1})}{\Delta x} + \underbrace{\left[\frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} - \frac{\Delta x^2}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^3)\right]}_{\text{Truncation Error (note sign)}}$$

So, the backward difference approximation is:

$$\frac{q_i - q_{i-1}}{\Delta x} = \left(\frac{dq}{dx}\right)_{x_i} - \frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \mathcal{O}(\Delta x^2)$$

The leading term of the truncation error is also proportional to Δx. Thus, the backward difference approximation is also **first-order accurate**, $\mathcal{O}(\Delta x)$.

Truncation Error: Central Difference



Subtract the Taylor expansion for $q(x_{i-1})$ from that for $q(x_{i+1})$:

$$q(x_{i+1}) = q_i + \Delta x \left(\frac{dq}{dx}\right)_i + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_i + \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_i + \mathcal{O}(\Delta x^4)$$

$$q(x_{i-1}) = q_i - \Delta x \left(\frac{dq}{dx}\right)_i + \frac{\Delta x^2}{2}\left(\frac{d^2q}{dx^2}\right)_i - \frac{\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_i + \mathcal{O}(\Delta x^4)$$

Subtracting the second from the first:

$$q(x_{i+1}) - q(x_{i-1}) = 2\Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{2\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^5)$$

Truncation Error: Central Difference

$$q(x_{i+1}) - q(x_{i-1}) = 2\Delta x \left(\frac{dq}{dx}\right)_{x_i} + \frac{2\Delta x^3}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^5)$$

(Note: $\mathcal{O}(\Delta x^4)$ terms might not fully cancel if they have different coefficients, but $\mathcal{O}(\Delta x^5)$ is the next non-zero term after Δx^3 for symmetric differences) Rearranging for the derivative:

$$\left(\frac{dq}{dx}\right)_{x_i} = \frac{q(x_{i+1}) - q(x_{i-1})}{2\Delta x} - \underbrace{\left[\frac{\Delta x^2}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)\right]}_{\text{Truncation Error}}$$

So, the central difference approximation is:

$$\frac{q_{i+1} - q_{i-1}}{2\Delta x} = \left(\frac{dq}{dx}\right)_{x_i} + \frac{\Delta x^2}{6}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \mathcal{O}(\Delta x^4)$$

The leading term of the truncation error is proportional to Δx^2. Thus, the central difference approximation is **second-order accurate**, $\mathcal{O}(\Delta x^2)$.

Finite Differences: Summary and Remarks

Things to remember:

- The terms neglected in the Taylor series constitute the **truncation error**.
- Truncation error measures the *local* accuracy of the approximation and determines the rate at which the error decreases as the grid spacing Δx is reduced.
- The first non-zero term in the truncation error expansion is usually the principal source of error for small Δx. Its power in Δx defines the **order of accuracy**.
- Higher order approximations can be obtained by using more points in the stencil to cancel out more terms in the Taylor expansion.

Finite Differences: Practical Examples

Let's apply the Forward, Backward, and Central difference schemes to approximate derivatives of known functions. This will allow us to:

- See how they perform in practice.
- Calculate the actual error by comparing with the exact analytical derivative.
- Numerically verify their order of accuracy.

We will use two example functions:

1. $f(x) = x^2$ (a simple polynomial)
2. $f(x) = \sin(x)$ (a transcendental function)

For both, we'll choose a point x_0 to evaluate the derivative and vary the step size Δx (denoted 'dx' in code).

Example 1: $f(x) = x^2$ - Setup

Consider the function $f(x) = x^2$. The analytical (exact) derivative is $f'(x) = 2x$.
Let's choose to approximate the derivative at $x_0 = 1.0$. The exact derivative at this point is $f'(1.0) = 2 \cdot 1.0 = 2.0$.
We will use a step size Δx, e.g., $\Delta x = 0.1$.

- For Forward Difference: we need $f(x_0)$ and $f(x_0 + \Delta x)$.
- For Backward Difference: we need $f(x_0)$ and $f(x_0 - \Delta x)$.
- For Central Difference: we need $f(x_0 - \Delta x)$ and $f(x_0 + \Delta x)$.

Before presenting the Python scripts to compute the errors for the forward, backward and central difference schemes, let's see something more about the Python 'print()' function, which will be useful to format the results when printed on screen.

Printing Formatted Output: F-Strings in Python

In Lecture 1, we introduced the basic ‘print()‘ function.
Python offers a powerful way to create formatted strings called **f-strings** (formatted string literals), available from Python 3.6+.
Basic Idea: F-strings allow you to embed expressions (like variable names) directly inside string literals by prefixing the string with an ‘f‘ or ‘F‘ and writing expressions inside curly braces ‘‘.

Syntax

```
variable_name = "World"
number = 42
print(f"Hello, {variable_name}! The number is {number}.")
# Output: Hello, World! The number is 42.
```

Printing Formatted Output: F-Strings in Python

Formatting Numbers within F-Strings: You can control how numbers are displayed by adding a colon ':' followed by a format specifier within the curly braces.

- 'variable:.Nf': Displays a floating-point number with 'N' digits after the decimal point.
- 'variable:.Ne': Displays a floating-point number in scientific notation with 'N' digits after the decimal point.
- 'variable:M.Nf': Displays a float, attempting to use 'M' total characters (including decimal point and sign), with 'N' digits after the decimal. (Less commonly needed for simple prints).

Syntax

```
print(f"Forward Diff (dx={dx:.2f}): {f_prime_forward:.4f},
      Error: {error_forward:.4e}")

# Output: Forward Diff (dx=0.10): 1.9000, Error: 1.0000e-01
```

Example 1: $f(x) = x^2$ - Python Code (Part 1)

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function and its exact derivative
def func_x_squared(x):
    return x**2

def deriv_x_squared(x):
    return 2*x

# Point at which to evaluate the derivative
x0 = 1.0
# Step size
dx = 0.1
```

Example 1: $f(x) = x^2$ - Python Code (Part 2)

Python Code

```
# (Continuing from previous slide)

# Exact derivative value
f_prime_exact = deriv_x_squared(x0)
print(f"Exact derivative of x^2 at x={x0}: {f_prime_exact:.4f}")

# Calculate function values needed for differences
f_x0 = func_x_squared(x0)
f_x0_plus_dx = func_x_squared(x0 + dx)
f_x0_minus_dx = func_x_squared(x0 - dx)

# Forward Difference
f_prime_forward = (f_x0_plus_dx - f_x0) / dx
error_forward = abs(f_prime_forward - f_prime_exact)
print(f"Forward Diff (dx={dx:.2f}): {f_prime_forward:.4f},
      Error: {error_forward:.4e}")
```

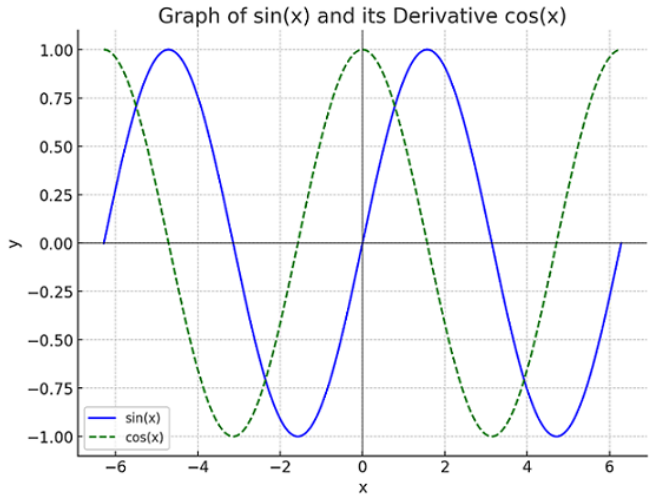
Example 1: $f(x) = x^2$ - Python Code (Part 2)

Python Code

```
# (Continuing from previous slide)

# Backward Difference
f_prime_backward = (f_x0 - f_x0_minus_dx) / dx
error_backward = abs(f_prime_backward - f_prime_exact)
print(f"Backward Diff (dx={dx:.2f}): {f_prime_backward:.4f},
      Error: {error_backward:.4e}")

# Central Difference
f_prime_central = (f_x0_plus_dx - f_x0_minus_dx) / (2 * dx)
error_central = abs(f_prime_central - f_prime_exact)
print(f"Central Diff (dx={dx:.2f}): {f_prime_central:.4f},
      Error: {error_central:.4e}")

# For f(x)=x^2, the central difference is exact!
# The error might be a very small float number due to precision.
```

Example 2: $f(x) = \sin(x)$ - Setup

Consider the function $f(x) = \sin(x)$. The analytical (exact) derivative is $f'(x) = \cos(x)$.



Let's choose to approximate the derivative at $x_0 = \pi/4$ (approx 0.7854). The exact derivative at this point is $f'(\pi/4) = \cos(\pi/4) = \sqrt{2}/2 \approx 0.7071$.
We will use a step size Δx, e.g., $\Delta x = 0.1$. The principles for calculating needed function values are the same as for x^2.

Example 2: $f(x) = \sin(x)$ - Python Code (Part 1)

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function and its exact derivative
def func_sin(x):
    return np.sin(x)

def deriv_sin(x):
    return np.cos(x)

# Point at which to evaluate the derivative
x0_sin = np.pi / 4.0
# Step size
dx_sin = 0.1
```

Example 2: $f(x) = \sin(x)$ - Python Code (Part 2)

Python Code Continuing from previous slide for sin(x))

```
# Exact derivative value
f_prime_exact_sin = deriv_sin(x0_sin)
print(f"Exact derivative of sin(x) at x={x0_sin:.4f}:
      {f_prime_exact_sin:.4f}")

# Calculate function values
f_x0_sin = func_sin(x0_sin)
f_x0_plus_dx_sin = func_sin(x0_sin + dx_sin)
f_x0_minus_dx_sin = func_sin(x0_sin - dx_sin)

# Forward Difference
f_prime_forward_sin = (f_x0_plus_dx_sin - f_x0_sin) / dx_sin
error_forward_sin = abs(f_prime_forward_sin - f_prime_exact_sin)
print(f"Fwd Sin (dx={dx_sin:.2f}): {f_prime_forward_sin:.4f},
      Error: {error_forward_sin:.4e}")
```

Example 2: $f(x) = \sin(x)$ - Python Code (Part 3)

Python Code Continuing from previous slide for sin(x))

```
# Backward Difference
f_prime_backward_sin = (f_x0_sin - f_x0_minus_dx_sin) / dx_sin
error_backward_sin = abs(f_prime_backward_sin - f_prime_exact_sin)
print(f"Bwd Sin (dx={dx_sin:.2f}): {f_prime_backward_sin:.4f},
      Error: {error_backward_sin:.4e}")

# Central Difference
f_prime_central_sin = (f_x0_plus_dx_sin - f_x0_minus_dx_sin) \
                      / (2 * dx_sin)
error_central_sin = abs(f_prime_central_sin - f_prime_exact_sin)
print(f"Ctrl Sin (dx={dx_sin:.2f}): {f_prime_central_sin:.4f},
      Error: {error_central_sin:.4e}")
```

Observe that for $\sin(x)$, the central difference is not exact but is significantly more accurate than forward/backward for the same Δx.

Numerically Verifying the Order of Accuracy

We've seen that the truncation error for a scheme of order p behaves like:

$$\text{Error} \approx C \cdot (\Delta x)^p$$

where C is a constant (related to higher derivatives of the function) and p is the order of accuracy.
If we take the logarithm of both sides:

$$\log(\text{Error}) \approx \log(C) + p \cdot \log(\Delta x)$$

This equation has the form $Y = A + pX$, where:

- $Y = \log(\text{Error})$
- $X = \log(\Delta x)$
- $A = \log(C)$ (the intercept)
- p is the **slope** of the line.

So, if we plot $\log(\text{Error})$ versus $\log(\Delta x)$, we should get a straight line whose slope is the order of accuracy p. This is called a **log-log plot**.

Understanding Log-Log Plots

Log-log plots are very common in Earth Sciences and engineering to visualize relationships that span several orders of magnitude or follow power-law behavior.

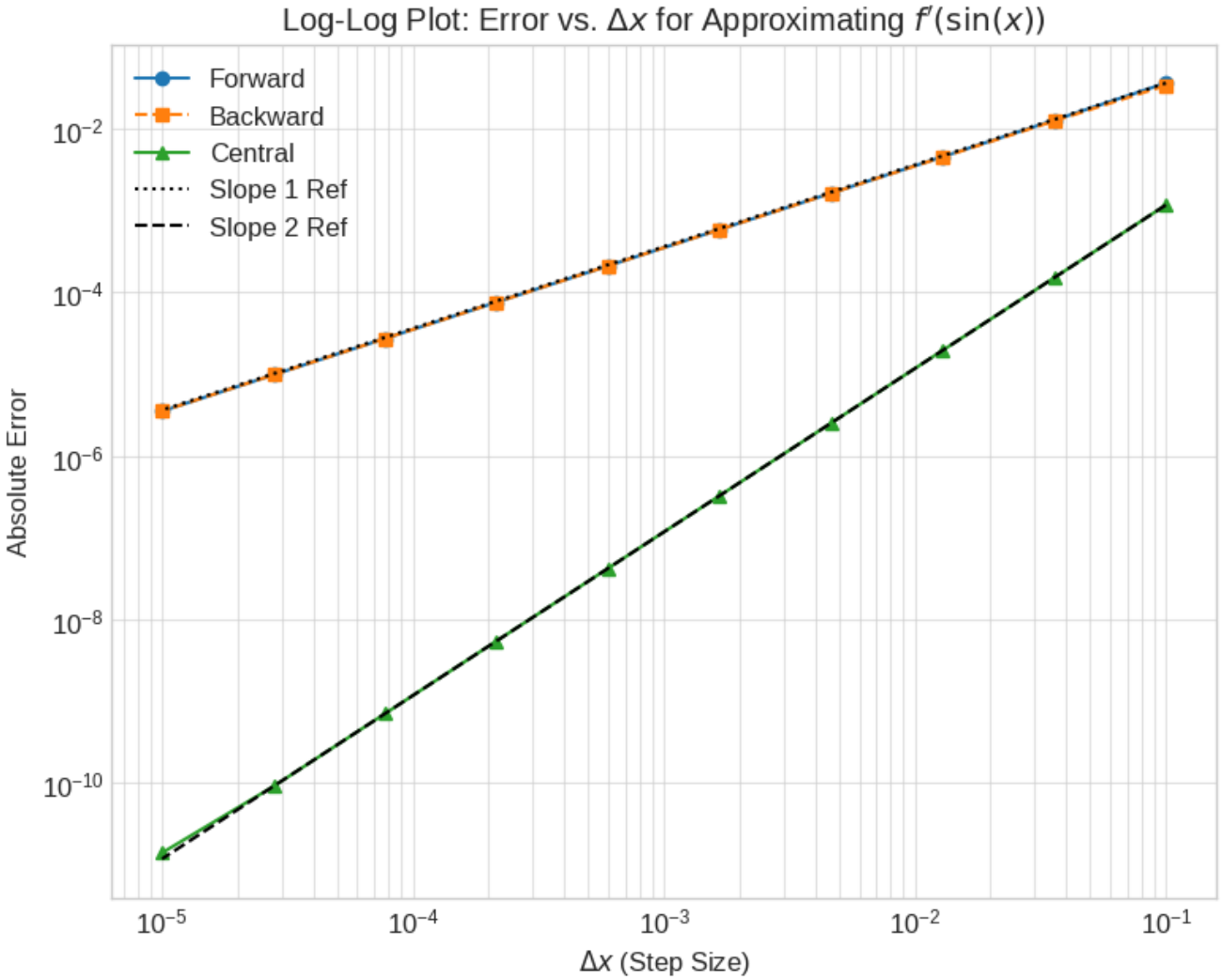
Why are they useful for error analysis?

- **Visualizing Order:** As Δx decreases (moves left on the x-axis), the error also decreases. The *rate* of this decrease is key.
- A scheme with order $p = 1$ (e.g., Forward Euler) means if you halve Δx, the error roughly halves. On a log-log plot, this gives a line with slope 1.
- A scheme with order $p = 2$ (e.g., Central Difference) means if you halve Δx, the error roughly quarters $(1/2^2)$. On a log-log plot, this gives a steeper line with slope 2.
- Higher order means faster convergence (error decreases more rapidly as Δx gets smaller).

The slope directly tells us how much more "bang for our buck" we get by decreasing Δx.

Order Verification for $\sin(x)$ - Python (Part 1)

Please see the Python code in the Jupyter Notebook.



Interpreting the Log-Log Plot

When you look at the generated log-log plot:

- **Linear Segments:** For larger Δx values, the data points for each method should form approximately straight lines. The slope of these lines indicates the observed order of accuracy.
- **Forward and Backward Differences:** Should have a slope close to 1.
- **Central Differences:** Should have a slope close to 2, indicating it's more accurate and converges faster.
- **Round-off Error Domination:** If Δx becomes *extremely* small (e.g., 10^{-8} or smaller, depending on function complexity and machine precision), you might see the error *increase* or behave erratically. This is where **round-off errors** (due to finite precision of computer arithmetic) start to dominate the truncation error.
  - For example, in $(f(x_0 + \Delta x) - f(x_0))/\Delta x$, if Δx is tiny, $f(x_0 + \Delta x)$ is very close to $f(x_0)$. The subtraction $f(x_0 + \Delta x) - f(x_0)$ might lose significant precision (this is called "catastrophic cancellation").
- For the Δx range typically used in these examples (10^{-1} to 10^{-4}), truncation error usually dominates.

Outline

Summary of New Topics in Lecture 2

Today, after concluding Lecture 1 topics, we covered:

- **Nonlinear ODEs:**
  - Showed an example (soil water infiltration) where implicit methods (like Backward Euler) lead to nonlinear algebraic equations.
  - This requires iterative solvers (e.g., Newton-Raphson) at each time step, increasing computational effort compared to linear ODEs or explicit methods.
- **Finite Difference Approximations for Derivatives:**
  - Defined Forward, Backward, and Central difference schemes for dq/dx.
  - Used Taylor series expansions to analyze their **truncation error**.
  - Determined the **order of accuracy**:
    - Forward/Backward: $\mathcal{O}(\Delta x)$ (first-order)
    - Central: $\mathcal{O}(\Delta x^2)$ (second-order)

Understanding these concepts is crucial for choosing and implementing numerical methods for solving differential equations.