Principles of numerical modelling in geosciences

Lecture 1 – From Physical Processes to Numerical Solutions

Mattia de' Michieli Vitturi

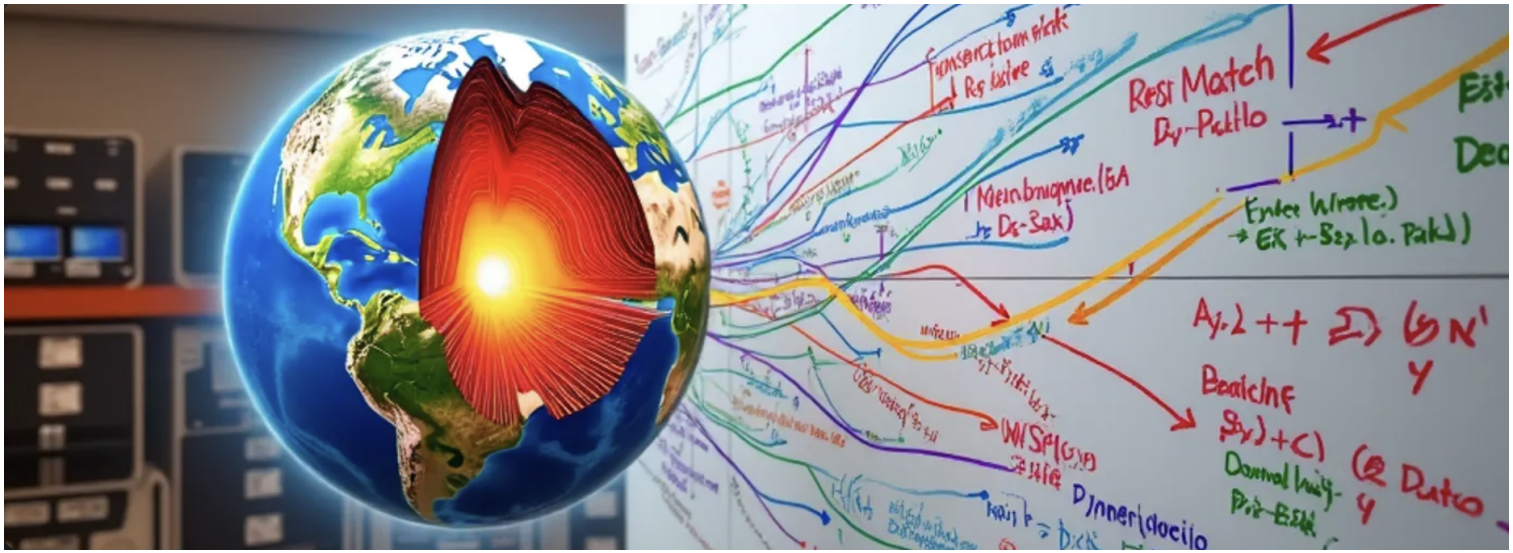
PhD Course in Earth Sciences

Overview

Table of Contents

Why Study Differential Equations? (1/2)

- Many Earth Science phenomena involve changes of physical quantities over space and time: heat transfer, fluid flow, seismic waves.
- These are governed by fundamental conservation laws expressed as differential equations.
- Understanding them allows us to predict and simulate natural processes.

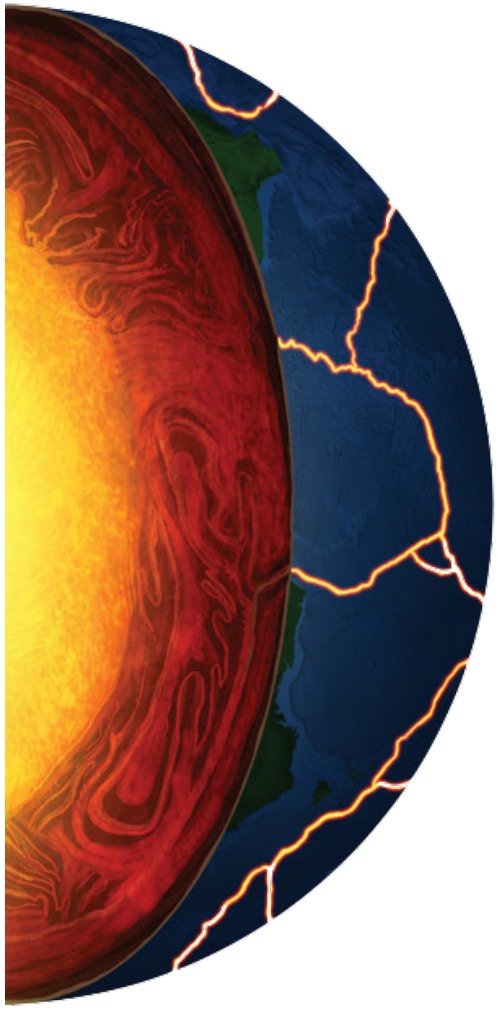


The Role of Differential Equations in Earth Sciences

- Differential equations describe how quantities evolve in time and/or space ($\frac{d\rho}{dt}$, ∇P).
- They arise naturally when expressing fundamental physical laws (e.g., conservation of mass, momentum, and energy).
- Most of the geophysical process can be described using *ordinary* or *partial* differential equations.
- Examples include:
  - Cooling of lava flows
  - Radioactive decay
  - Groundwater flow
  - Atmospheric circulation
- Unfortunately, analytical solutions exist only for simple cases.

Cooling in Earth Sciences

- **Examples of cooling processes:**
  - Lava flow cooling after emplacement.
  - Cooling of the Earth's lithosphere over geological timescales.
  - Thermal relaxation of volcanic deposits.
- **What do they have in common?**
  - Heat is transferred from a hot material to a cooler environment.
  - The temperature of the material decreases over time.
- To describe this behavior, we can try to write a mathematical model.



Newton's Cooling Law – Discrete Formulation

- We start from the physical observation that heat loss is proportional to temperature difference.
- Then we introduce some notation:
  - $T(t)$ be the temperature of the body at time t.
  - T_a be the constant ambient temperature.
  - $k > 0$ the cooling coefficient.
- Over a small time interval Δt, the change in temperature is:

$$\Delta T = T(t + \Delta t) - T(t) = -k\,(T(t) - T_a)\,\Delta t$$

- Rearranged:

$$\frac{T(t + \Delta t) - T(t)}{\Delta t} = -k\,(T(t) - T_a)$$

- This is a *finite difference* expression for the rate of temperature change.

Newton's Cooling Law – Differential Formulation

- When $\Delta t \to 0$, the finite difference becomes a derivative:

$$\frac{dT}{dt} = \lim_{\Delta t \to 0} \frac{T(t + \Delta t) - T(t)}{\Delta t}$$

- Taking the limit of the previous equation:

$$\frac{dT}{dt} = -k\,(T(t) - T_a)$$

- This is a first-order linear ordinary differential equation (ODE).
- Widely used in Earth sciences to model cooling:
  - Lava temperature evolution.
  - Permafrost thawing.
  - Post-eruption thermal relaxation.

Historical note: Newton introduced this law in his 1701 paper "Scala graduum caloris". It was the first heat transfer formulation and serves as the formal basis of convective heat transfer.

From Physical Law to Differential Equation – Newton's Cooling Law

- **Empirical observation (Newton, 1701):**

  "The rate at which an object cools is proportional to the difference in temperature with its surroundings."

- **Discrete formulation:**

$$\Delta T = -k(T - T_a)\,\Delta t$$

- **Differential formulation:**

$$\frac{dT}{dt} = -k(T - T_a)$$

- This first-order ODE is widely used to describe thermal processes in Earth sciences.

Ordinary Differential Equations (ODEs)

- An **Ordinary Differential Equation** (ODE) involves one or more derivatives with respect to a **single independent variable**, usually time (t).

$$\frac{dT}{dt} = -k(T - T_a)$$

- The unknown is a function of t only (e.g., temperature, concentration, velocity...).
- ODEs are widely used in Earth Sciences to model:
  - Radioactive decay.
  - Cooling/heating of materials.
  - Chemical reaction kinetics.
  - Mass transfer in simple systems.
- We will focus first on ODEs and how to solve them numerically.

ODEs vs PDEs

- **Partial Differential Equations (PDEs)** involve derivatives with respect to **multiple independent variables**, e.g. time and space.

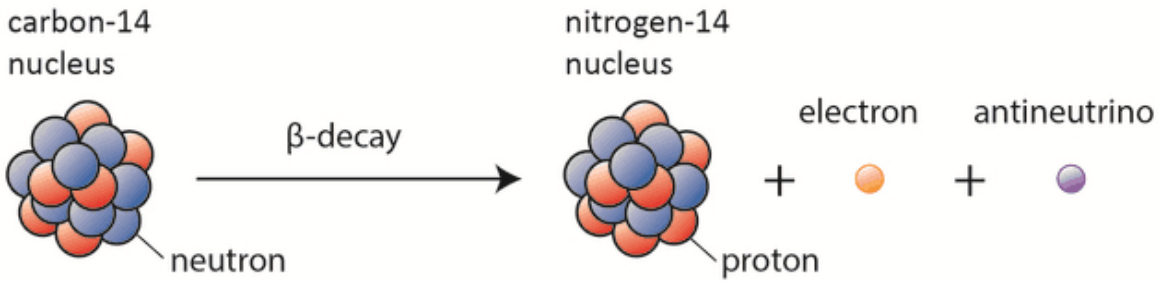
$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$$

- Example: the heat equation in a 1D rock layer ($T = T(x, t)$).
- PDEs are essential to describe:
  - Heat diffusion in the Earth's crust.
  - Groundwater flow.
  - Deformation of rocks under stress.
  - Atmospheric or oceanic dynamics.
  - Conservation laws (mass, momentum, energy).
- In this course, we start with ODEs and later introduce some examples of PDEs, discussing how and why their numerical treatment differs.

Radioactive Decay in Earth Sciences

Let consider now another example of physical process described by an ODE.

- Radioactive isotopes are used for **radiometric dating** of rocks and minerals.
- Examples:
  - $^{238}\text{U} \rightarrow {}^{206}\text{Pb}$ (half-life $\approx$ 4.5 billion years)
  - $^{14}\text{C} \rightarrow {}^{14}\text{N}$ (half-life $\approx$ 5730 years)



- The amount of the radioactive parent isotope decreases over time.
- We want to model the evolution of the quantity $N(t)$ = number of atoms at time t.

From Discrete Steps to Continuous Model

- Let N_k be the number of atoms at time $t_k = k\Delta t$.
- Assume a constant fraction α of atoms decays in each time step:

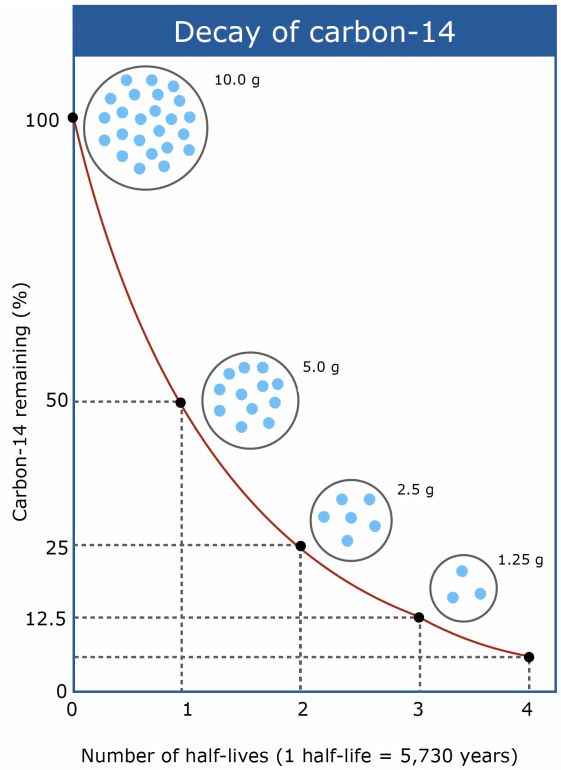
$$N_{k+1} = N_k - \alpha N_k = (1 - \alpha) N_k$$

- Recursive expression:

$$N_k = (1 - \alpha)^k N_0$$

- Taking the limit as $\Delta t \to 0$ and defining $\lambda = \lim_{\Delta t \to 0} \frac{\alpha}{\Delta t}$:

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



The Differential Equation of Radioactive Decay

- We have derived the ODE:

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

- Notation:
  - $N(t)$: number of atoms (or concentration) at time t.
  - t: time (independent variable).
  - λ: decay constant, specific to the isotope.
- This is a first-order linear ODE with known analytical solution:

$$N(t) = N_0 e^{-\lambda t}$$

- This model is the foundation of many dating techniques in geochronology.

Table of Contents

Why Python?

- **Python** is a modern, open-source programming language.
- Widely used in science and engineering:
  - Simple and readable syntax.
  - Powerful libraries for scientific computing.
  - Strong community and many online resources.
- In Earth Sciences, Python is used for:
  - Data analysis (e.g., satellite or geochemical data)
  - Numerical modeling (e.g., climate, volcanology, geodynamics)
  - Visualization (e.g., topography, maps, time series)

Jupyter Notebooks

- We will use **Jupyter Notebooks** to run Python code interactively.
- A notebook is a web-based interface combining:
  - Code execution
  - Documentation (markdown)
  - Visualizations (plots, maps)
- Ideal for scientific workflows:
  - Step-by-step exploration
  - Documentation of the analysis
  - Reproducibility and sharing
- You can run notebooks locally or online (e.g., via Google Colab or VS Code).

Python Syntax Basics

- Variables and assignment:

Example

```
x = 3.0         # float
name = "rock" # string
is_hot = True # boolean
```

- Indentation defines code blocks (e.g., loops, functions):

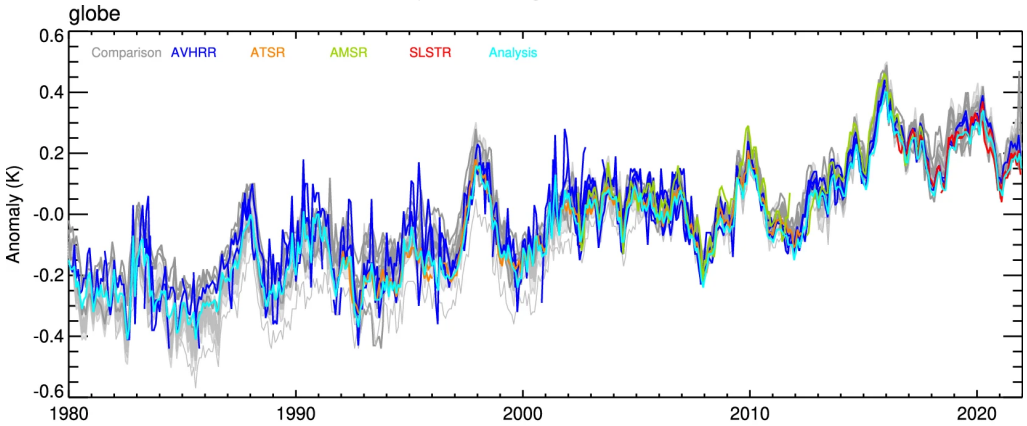
Example

```
for i in range(5):
    print(i)
```

- Comments start with #, and help explain your code.

Why We Need More Than Single Variables

- In many scientific tasks, we deal with collections of data, not just single values.
- For example:
  - A series of temperature measurements over time.
  - A list of rock sample names.
  - Coordinates (x, y, z) of multiple seismic events.
  - Porosity values from different core samples.
- Storing each value in a separate variable (e.g., ‘temp1‘, ‘temp2‘, ‘temp3‘, ...) would be impractical and inefficient.
- We need a way to group related data items together.



From: Satellite-based time-series of sea-surface temperature since 1980 for climate applications

Introducing Lists in Python

- Python's **list** is a versatile and commonly used data structure to store an ordered sequence of items.
- **Syntax:** Lists are created using square brackets '[]', with items separated by commas.

Creating a list

```
temperatures = [15.5, 16.1, 15.8, 17.0]
rock_types = ["basalt", "granite", "shale"]
mixed_data = [10, "andesite", 25.3, True]
empty_list = []
```

- **Characteristics of Python Lists:**
  - **Ordered:** Items are stored in a specific sequence.
  - **Mutable:** You can change, add, or remove items after the list is created.
  - **Can contain mixed data types:** A single list can hold numbers, strings, booleans, and even other lists.

Displaying Output: The ‘print()‘ Function

- The ‘print()‘ function is essential for displaying information, variable values, or results to the console.
- **Basic Syntax:** ‘print(object1, object2, ..., sep=’ ’, end=’\n’)‘
  - ‘object1, object2, ...‘: The items to be printed.
  - ‘sep=’ ’‘: (Optional) The separator between items (default is a space).
  - ‘end=’\n’‘: (Optional) What to print at the end (default is a newline character ‘\n‘).

Displaying Output: The 'print()' Function

Examples of 'print()'

```
print("Hello, Earth Scientists!")

year = 2024
print("Current year:", year)

temperatures = [15.5, 16.1, 17.0]
print("Temperature readings:", temperatures)

# Printing multiple items with a custom separator
name = "Vesuvio"
elevation = 1281
print(name, elevation, "meters", sep=" - ")
```

Generating Sequences: The ‘range()‘ Function (Part 1)

- The ‘range()‘ function generates a sequence of numbers. It’s particularly useful in loops.
- It does *not* create a list directly, but an ”iterable” object.
- **Common Syntaxes:**
  1. ‘range(stop)‘:
     - Generates numbers from ‘0‘ up to (but not including) ‘stop‘.
     - Example: ‘range(5)‘ produces ‘0, 1, 2, 3, 4‘.
  2. ‘range(start, stop)‘:
     - Generates numbers from ‘start‘ up to (but not including) ‘stop‘.
     - Example: ‘range(2, 6)‘ produces ‘2, 3, 4, 5‘.
  3. ‘range(start, stop, step)‘:
     - Generates numbers from ‘start‘ up to (but not including) ‘stop‘, with an increment of ‘step‘.
     - Example: ‘range(1, 10, 2)‘ produces ‘1, 3, 5, 7, 9‘.
     - ‘step‘ can also be negative for counting down.

Generating Sequences: The 'range()' Function (Part 2 - Examples)

Using 'range()'

```
# To see the numbers, we can convert range to a list
seq1 = list(range(5))
print("range(5):", seq1) # Output: [0, 1, 2, 3, 4]

seq2 = list(range(3, 8))
print("range(3, 8):", seq2) # Output: [3, 4, 5, 6, 7]

seq3 = list(range(10, 0, -2))
print("range(10, 0, -2):", seq3) # Output: [10, 8, 6, 4, 2]

# range() is often used in for loops (more on loops later)
print("Looping with range:")
for i in range(3):
    print("Iteration number:", i)
```

Accessing List Elements: Indexing (Part 1 - Basics)

- Each item in a list has a position, called its **index**.
- **Crucial: Python uses 0-based indexing!**
  - The first item is at index ‘0‘.
  - The second item is at index ‘1‘, and so on.
- **Syntax:** ‘list_name[index]‘

Basic Indexing

```
rock_samples = ["granite", "basalt", "shale", "sandstone"]
print("First sample:", rock_samples[0])  # Output: granite
print("Third sample:", rock_samples[2])  # Output: shale

# Store an element in a variable
first_rock = rock_samples[0]
print("Stored first rock:", first_rock)
```

Accessing List Elements: Indexing (Part 2 - Negative Indexing)

Common Error

Accessing an index that is out of bounds will result in an ‘IndexError‘. For ‘rock_samples‘ above, ‘rock_samples[4]‘ would cause an error.

- Python also supports **negative indexing**, which is very handy.
- Negative indices count from the end of the list:
  - ‘-1‘ refers to the **last** item.
  - ‘-2‘ refers to the **second-to-last** item, and so on.

Negative Indexing

```
elements = ["Oxygen", "Silicon", "Aluminum", "Iron", "Calcium"]

print("Last element:", elements[-1])     # Output: Calcium
print("Second to last:", elements[-2]) # Output: Iron

# Useful for getting the last few items without knowing the list's length
```

Accessing List Elements: Slicing (Part 1 - Subsets)

- **Slicing** allows you to get a sub-list (a "slice") from a list.
- **Basic Syntax:** ‘list_name[start:stop]‘
  - ‘start‘: The index of the first item to include (inclusive). If omitted, defaults to ‘0‘.
  - ‘stop‘: The index of the first item **not** to include (exclusive). If omitted, defaults to the end of the list.
- The result of a slice is a new list.

Basic Slicing

```
measurements = [10.1, 12.5, 11.3, 13.0, 12.8, 10.9]

# Get elements from index 1 up to (but not including) index 4
sub_list1 = measurements[1:4]
print("measurements[1:4]:", sub_list1)

# Output: [12.5, 11.3, 13.0]
```

Accessing List Elements: Slicing (Part 1 - Subsets)

Basic Slicing

```
measurements = [10.1, 12.5, 11.3, 13.0, 12.8, 10.9]

# Get elements from the beginning up to index 3 (exclusive)
sub_list2 = measurements[:3]
print("measurements[:3]:", sub_list2)
# Output: [10.1, 12.5, 11.3]

# Get elements from index 2 to the end
sub_list3 = measurements[2:]
print("measurements[2:]:", sub_list3)
# Output: [11.3, 13.0, 12.8, 10.9]

# Create a copy of the entire list
full_copy = measurements[:]
print("measurements[:]:", full_copy)
```

Accessing List Elements: Slicing (Part 2 - Step)

- Slicing can also include a **step** value.
- **Syntax:** 'list_name[start:stop:step]'
  - 'step': The increment between indices. Defaults to '1'.
  - Can be used to select every Nth item, or to reverse a list.

Slicing with a Step

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get every second element from the beginning
every_other = numbers[::2]

print("numbers[::2]:", every_other)
# Output: [0, 2, 4, 6, 8]
```

Accessing List Elements: Slicing (Part 2 - Step)

Slicing with a Step

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get elements from index 1 to 7, with a step of 3
stepped_slice = numbers[1:8:3]
print("numbers[1:8:3]:", stepped_slice)
# Output: [1, 4, 7]

# A common trick to reverse a list
reversed_list = numbers[::-1]
print("numbers[::-1]:", reversed_list)
# Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Variables: Names Pointing to Data (References)

- In Python, when you assign a variable to an existing **mutable object** (like a list), you are not creating a new copy of the object.
- Instead, the new variable becomes another **name (or reference)** pointing to the *same object* in memory.
- Consider this example:

Assigning a list to another variable

```
a = [1, 2, 3]  # 'a' points to a list object
b = a          # 'b' now points to the *same* list object as 'a'
```

- Visualize ‘a‘ and ‘b‘ as two labels attached to the same box containing ‘[1, 2, 3]‘.
- If you modify the list through one variable, the change will be visible through the other variable as well, because they refer to the exact same data.

Modifying Through a Reference: Shared Changes

- Let's continue with 'a' and 'b' from the previous slide:

Modifying the list via 'b'

```
a = [1, 2, 3]
b = a

# Modify the list using the variable 'b'
b[0] = 5  # Change the first element of the list b points to

# Now, let's check both 'a' and 'b'
print("List a:", a)
print("List b:", b)

# Output:
# List a: [5, 2, 3]
# List b: [5, 2, 3]
```

Modifying Through a Reference: Shared Changes

```
# Output:
# List a: [5, 2, 3]
# List b: [5, 2, 3]
```

- As you can see, changing ‘b[0]‘ also changed ‘a[0]‘. This is because ‘a‘ and ‘b‘ reference the **same list object**.
- This behavior is important to understand to avoid unintended side effects when working with mutable data types (lists, dictionaries, sets, custom objects).
- For immutable types (numbers, strings, tuples), this direct assignment behaves more like a copy of the value itself because immutable objects cannot be changed in place.

How to Create Actual Copies (Shallow and Deep)

- If you need a true, independent copy of a list (or other mutable object), you must explicitly create one.
- **Shallow Copy:** Creates a new object, but if the original object contains other mutable objects (e.g., a list of lists), the inner objects are still references.
  - Using slicing: ‘new_list = old_list[:]‘
  - Using the ‘list()‘ constructor: ‘new_list = list(old_list)‘
  - Using the ‘copy()‘ method: ‘new_list = old_list.copy()‘ (from the ‘copy‘ module or built-in for lists)

Creating a shallow copy

```
original = [10, 20, 30]
# Create a shallow copy
copied_list = original[:]
# Alternatively: copied_list = original.copy()
```

How to Create Actual Copies (Shallow and Deep)

Creating a shallow copy

```
...
copied_list[0] = 99

print("Original:", original)
# Output: Original: [10, 20, 30]

print("Copied List:", copied_list)
# Output: Copied List: [99, 20, 30]

# Modifying copied_list does NOT affect original
```

- **Deep Copy:** Creates a new object and recursively copies all objects found in the original. This is useful for complex nested structures.
  - Requires the ‘copy‘ module: ‘import copy; new_list = copy.deepcopy(old_list)‘

What is NumPy?

- **NumPy** (Numerical Python) is the fundamental package for scientific computing in Python.
- It provides:
  - A powerful N-dimensional **array object** ('ndarray').
  - Useful linear algebra, Fourier transform, and random number capabilities.
  - Tools for integrating C/C++ and Fortran code.
- NumPy arrays are the core data structure for many other scientific Python libraries (e.g., SciPy, Pandas, Matplotlib, Scikit-learn).
- **Convention:** NumPy is typically imported as 'np'.

Importing NumPy

```
import numpy as np
```

Python Lists vs. NumPy Arrays (1/2)

- Python lists are general-purpose, flexible containers.
- NumPy arrays are designed for efficient numerical operations.
- **Key Differences:**
  - **Type Homogeneity:**
    - *Lists:* Can store elements of different data types (e.g., '[1, "rock", 3.14]').
    - *NumPy Arrays:* Typically store elements of the **same data type** (e.g., all integers or all floats). This allows for optimized storage and operations.
  - **Performance for Numerical Operations:**
    - *Lists:* Operations on numerical data often require explicit Python loops, which can be slow.
    - *NumPy Arrays:* Support **vectorized operations** (element-wise operations) that are implemented in C and are significantly faster.

Python Lists vs. NumPy Arrays (2/2)

- **Key Differences (continued):**
  - **Functionality:**
    - *Lists:* Basic sequence operations (append, insert, etc.).
    - *NumPy Arrays:* Offer a vast array of mathematical functions, linear algebra routines, random number generation, etc., that operate directly on arrays.

When to use which?

- Use Python lists for general-purpose collections, especially if you need to store mixed data types or require dynamic resizing with ‘append‘/‘insert‘.
- Use NumPy arrays when working with numerical data, especially for mathematical operations, large datasets, and when performance is critical.

Creating NumPy Arrays

- From Python lists or tuples:

From a list

```
import numpy as np

py_list = [1, 2, 3, 4, 5]
np_array_from_list = np.array(py_list)
print(np_array_from_list)
print(type(np_array_from_list)) # <class 'numpy.ndarray'>
```

- Using built-in NumPy functions:
  - 'np.zeros(shape)': Array of zeros.
  - 'np.ones(shape)': Array of ones.
  - 'np.arange(start, stop, step)': Similar to Python's 'range', but returns an array.
  - 'np.linspace(start, stop, num)': Array of evenly spaced values over an interval.
  - 'np.random.rand(shape)': Array of random values.

NumPy Array Attributes and Basic Operations

- NumPy arrays have useful attributes:
  - ‘ndarray.ndim‘: Number of dimensions (axes).
  - ‘ndarray.shape‘: Tuple of array dimensions.
  - ‘ndarray.size‘: Total number of elements.
  - ‘ndarray.dtype‘: Data type of the elements.

Array Attributes

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array

print("Array:\n", arr)
print("ndim:", arr.ndim)      # Output: 2
print("shape:", arr.shape)    # Output: (2, 3)
print("size:", arr.size)      # Output: 6
print("dtype:", arr.dtype)    # Output: int64
```

NumPy Array Attributes and Basic Operations

- **Vectorized Operations:** Mathematical operations apply element-wise.

Element-wise operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print("a + b =", a + b)
# Output: [5 7 9]

print("a * 2 =", a * 2)
# Output: [2 4 6]

print("a ** 2 =", a ** 2)
# Output: [1 4 9]
```

Quick Comparison Summary

Feature	Python List	NumPy Array
Data Types	Heterogeneous	Homogeneous (typically)
Memory	Higher overhead	More compact
Numerical Ops	Slow (Python loops)	Fast (Vectorized, C-based)
Functionality	Basic sequence ops	Rich mathematical functions
Flexibility	High (dynamic add/remove)	Less flexible (fixed size)*
Primary Use	General purpose	Numerical computation

*NumPy arrays are generally fixed-size once created, though functions exist to resize/concatenate, often creating new arrays.

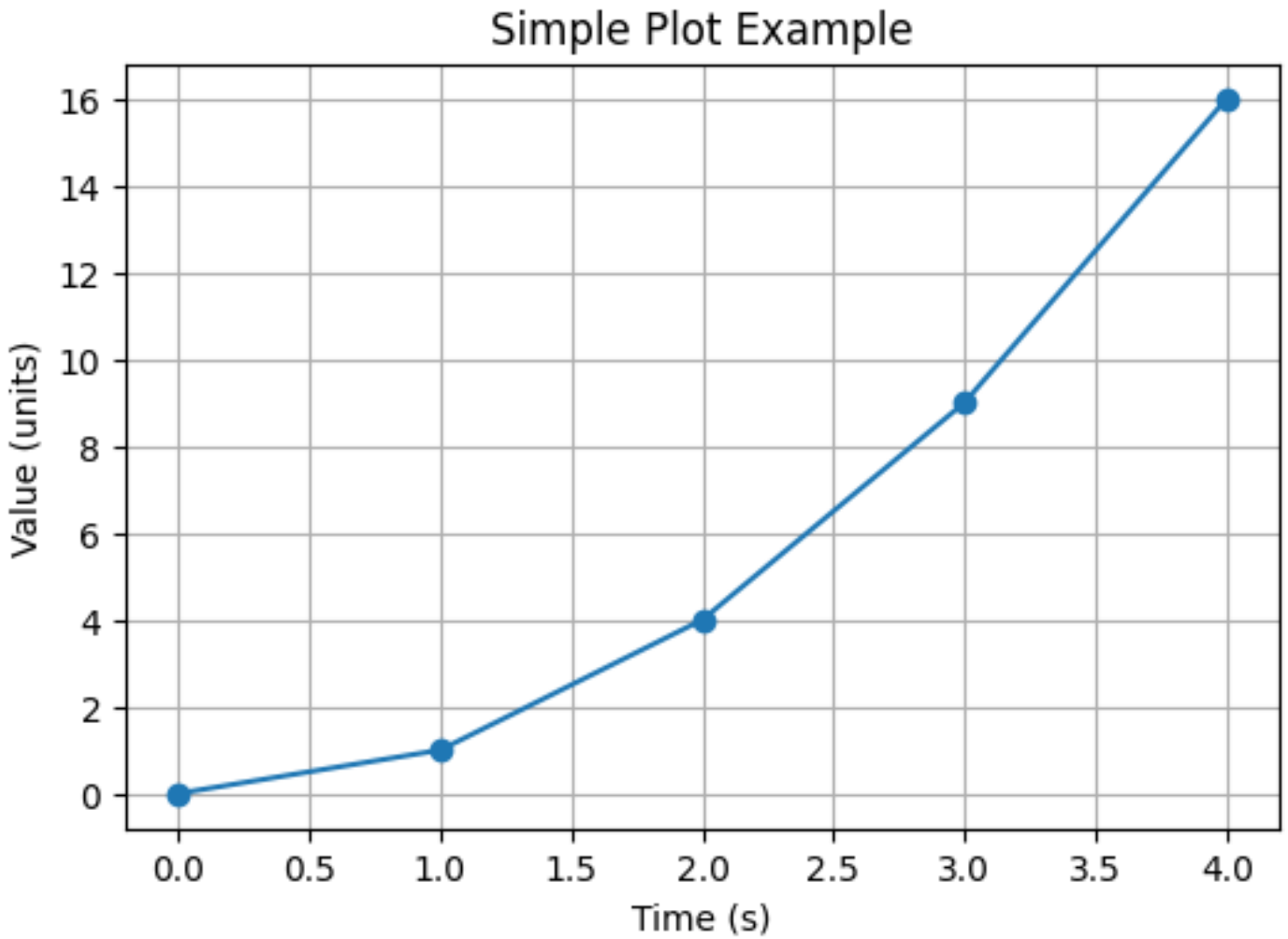
Visualizing Data with Matplotlib

- **Matplotlib** is the main Python library for plotting scientific data.
- We use the `pyplot` interface for quick and intuitive plots.

Basic usage

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3]
y = [0, 1, 4, 9]
plt.plot(x, y)
plt.xlabel("Time")
plt.ylabel("Value")
plt.title("Simple plot")
plt.show()
```

Visualizing Data with Matplotlib



Plotting ODE Solutions with NumPy and Matplotlib

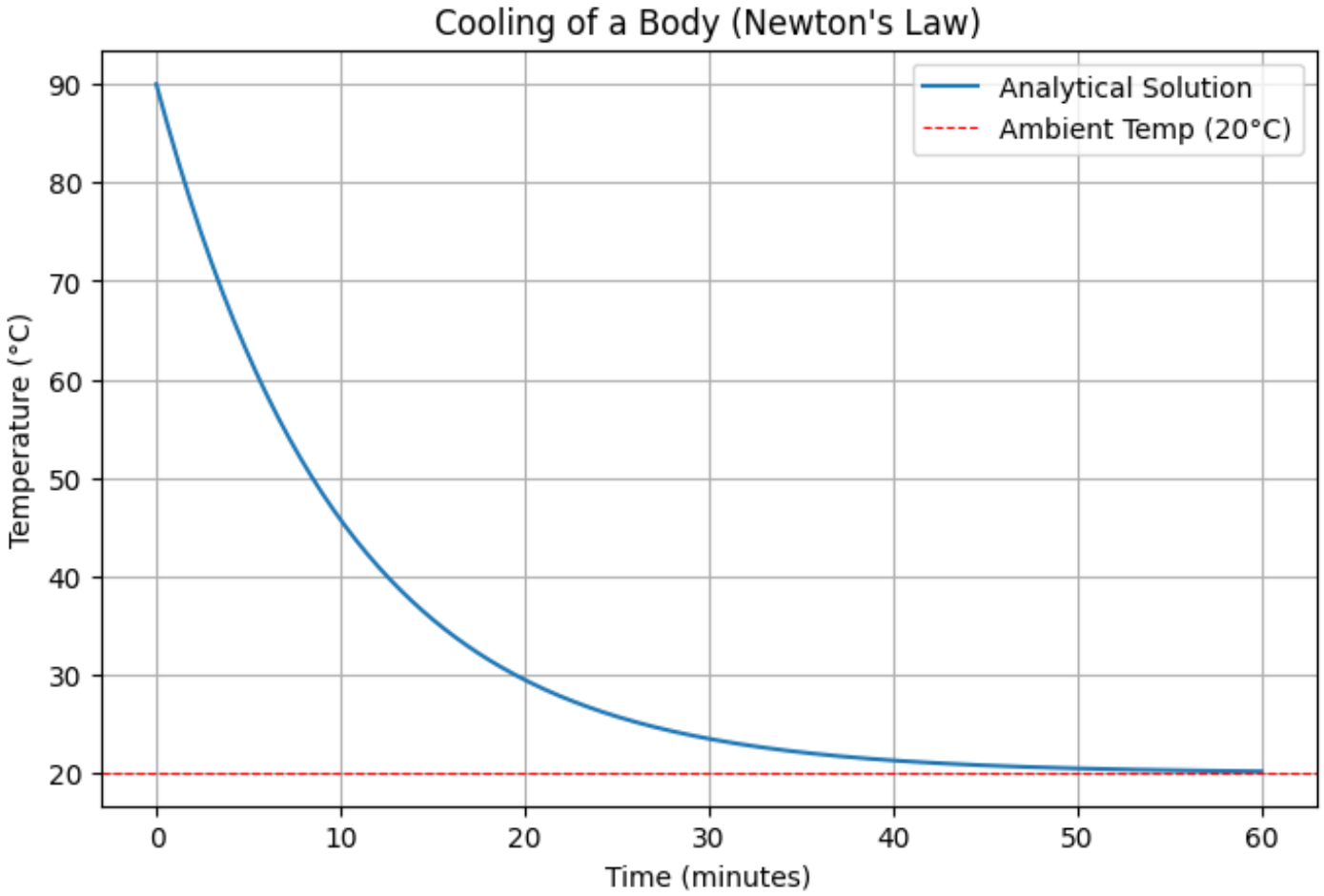
- Let's visualize the analytical solution of the cooling equation:

$$T(t) = T_a + (T_0 - T_a)e^{-kt}$$

Python code

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

T_a = 20
T_0 = 90
k = 0.1
t = np.linspace(0, 60, 300)
T = T_a + (T_0 - T_a) * np.exp(-k * t)

plt.plot(t, T)
plt.xlabel("Time (min)")
plt.ylabel("Temperature (°C)")
plt.title("Cooling of a body")
plt.grid()
plt.show()
```

Plotting ODE Solutions with NumPy and Matplotlib



Elements of a Matplotlib Plot

A basic plot can contain several elements:

- `plt.plot()` — plots a line or data series.
- `plt.xlabel()` and `plt.ylabel()` — axis labels.
- `plt.title()` — adds a title to the plot.
- `plt.grid()` — adds a background grid.
- `plt.legend()` — shows labels for multiple data series.
- `plt.show()` — displays the figure (necessary in scripts).

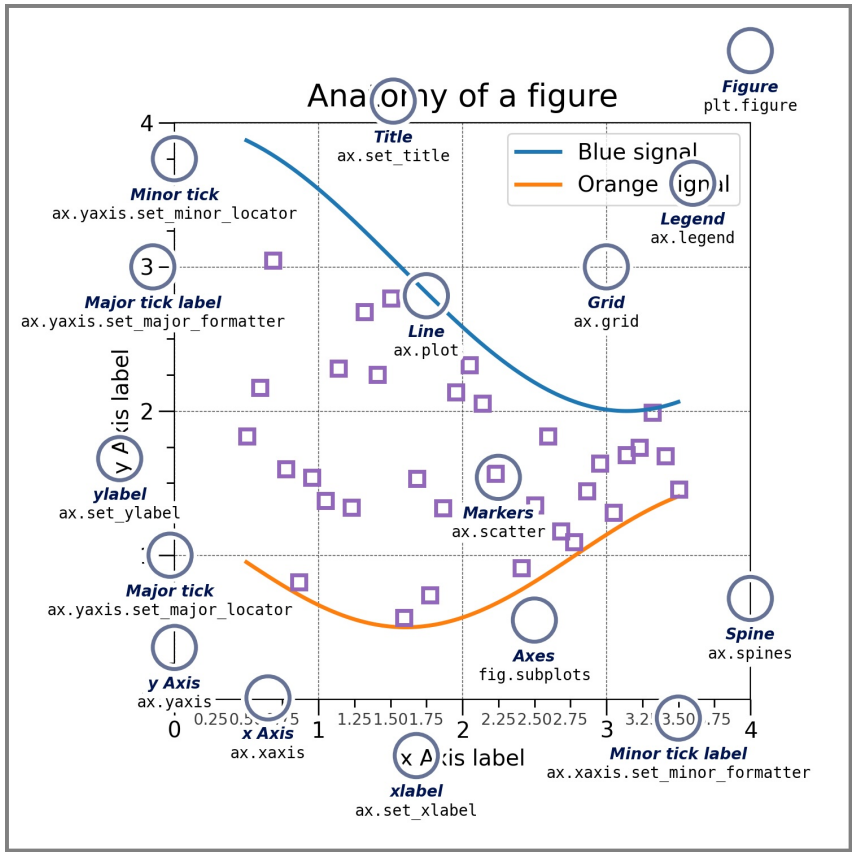


Table of Contents

Why Numerical Solutions?

- In some cases, we can find analytical solutions to differential equations.
- However, most real-world problems do not admit an analytical solution.
- Even when analytical solutions exist, they might be too complex to handle or not useful for computations.
- **Numerical methods** allow us to *approximate* solutions at discrete points.
- Our goal: find a practical method to approximate the solution of the radioactive decay ODE.

Numerical Solution of Radioactive Decay (1/2)

Continuous equation:

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

We want to approximate the solution at discrete times:

$$t_0, t_1 = t_0 + \Delta t, \ldots, t_n = t_0 + n\Delta t$$

Idea: Replace the derivative with a finite difference:

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

Substitute into the equation:

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

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

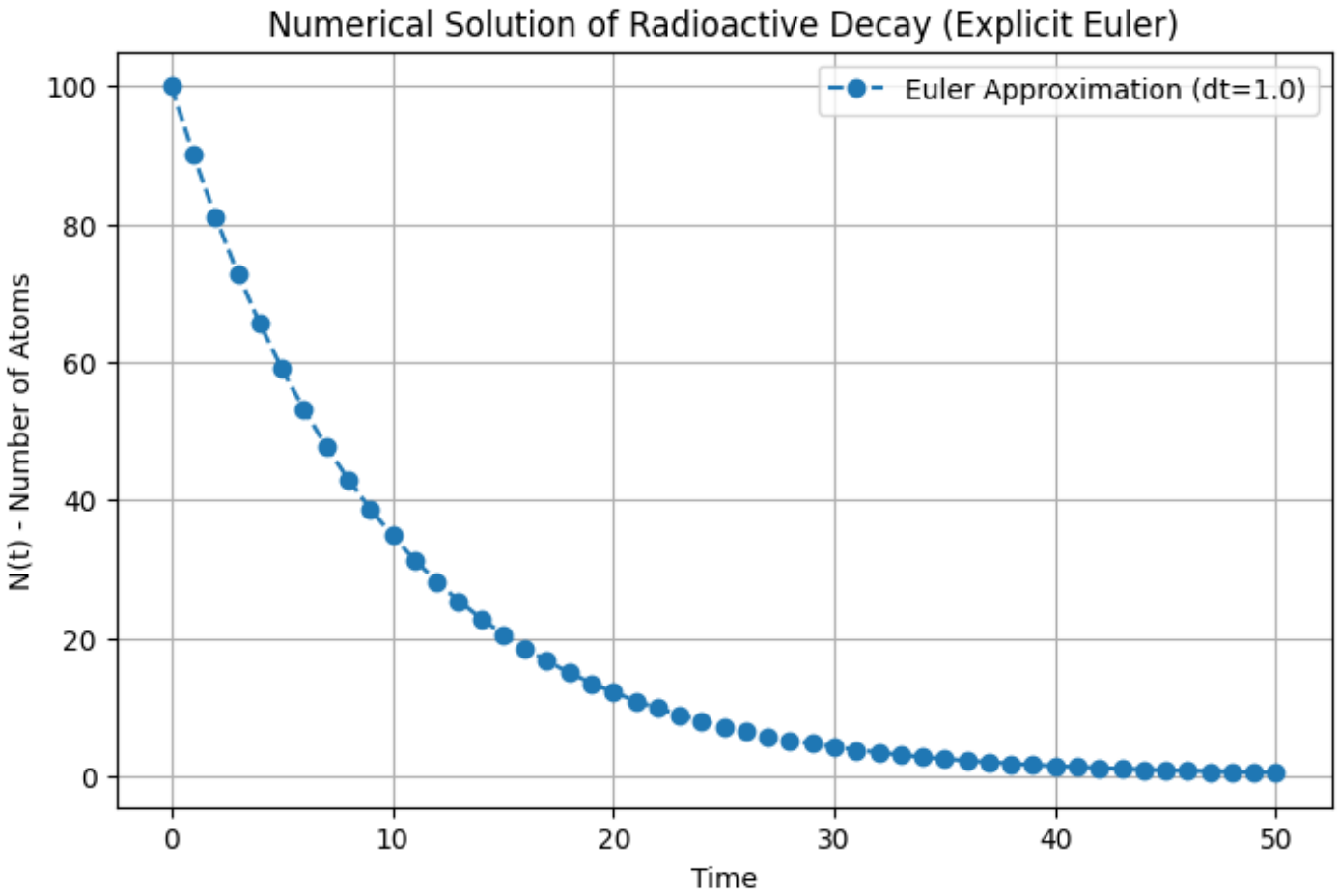
Euler's Method applied to radioactive decay

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()
```

Note: This is purely a numerical result, not yet compared to the exact solution.

Plotting the Numerical Solution Only



Exercise: Try a Larger Time Step

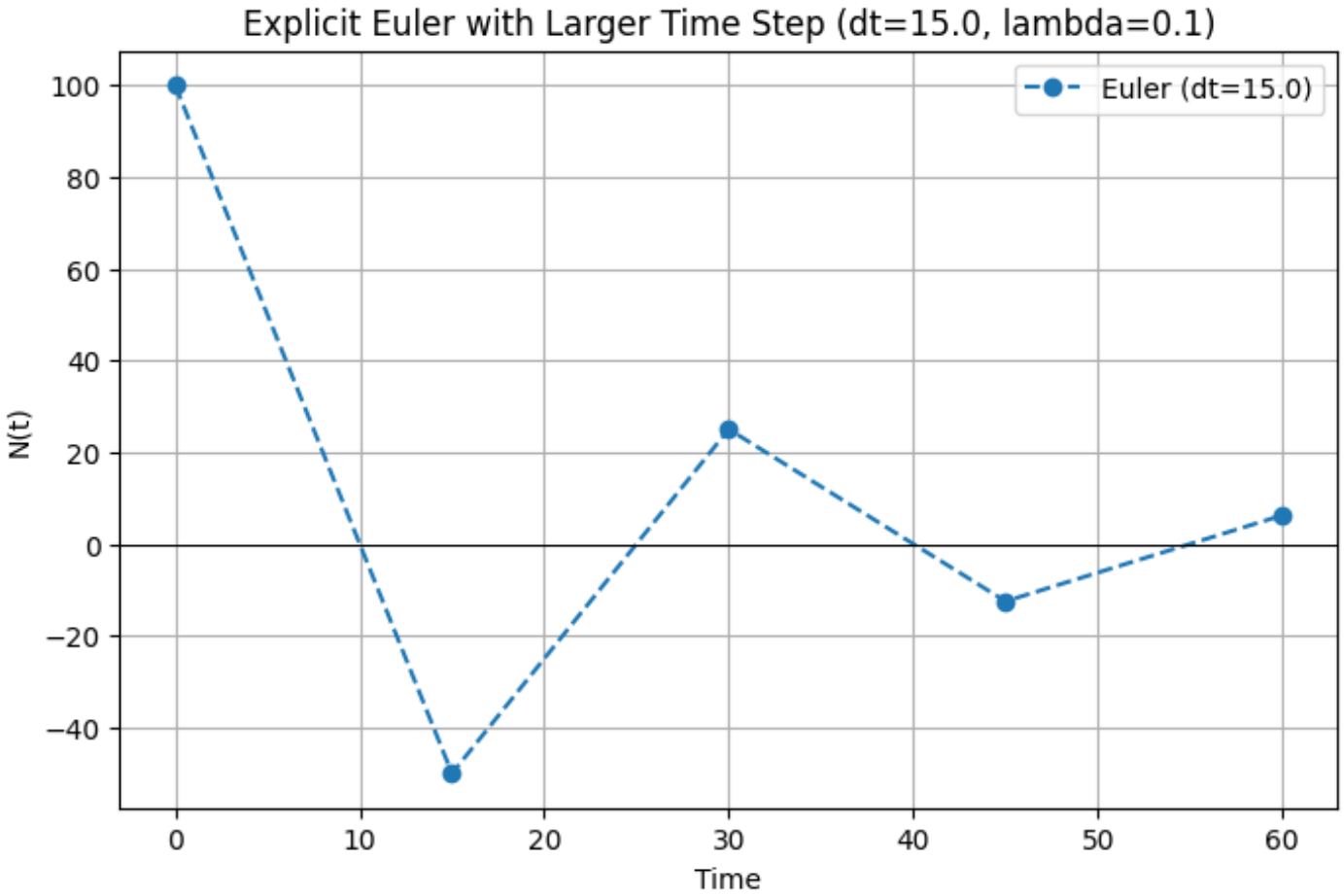
Change the time step and observe the result

Python code

```
# Try dt = 5.0 instead of 1.0
dt = 5.0
time = np.arange(0, T + dt, dt)
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)

plt.plot(time, N_euler, 'o--')
plt.title('Euler Method with Larger Time Step')
plt.grid(True)
plt.show()
```

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

Exercise: Try a Larger Time Step



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:

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

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.

Backward Euler Method

Alternative discretizations: implicit 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, even for large time steps.

Implicit Euler in Python

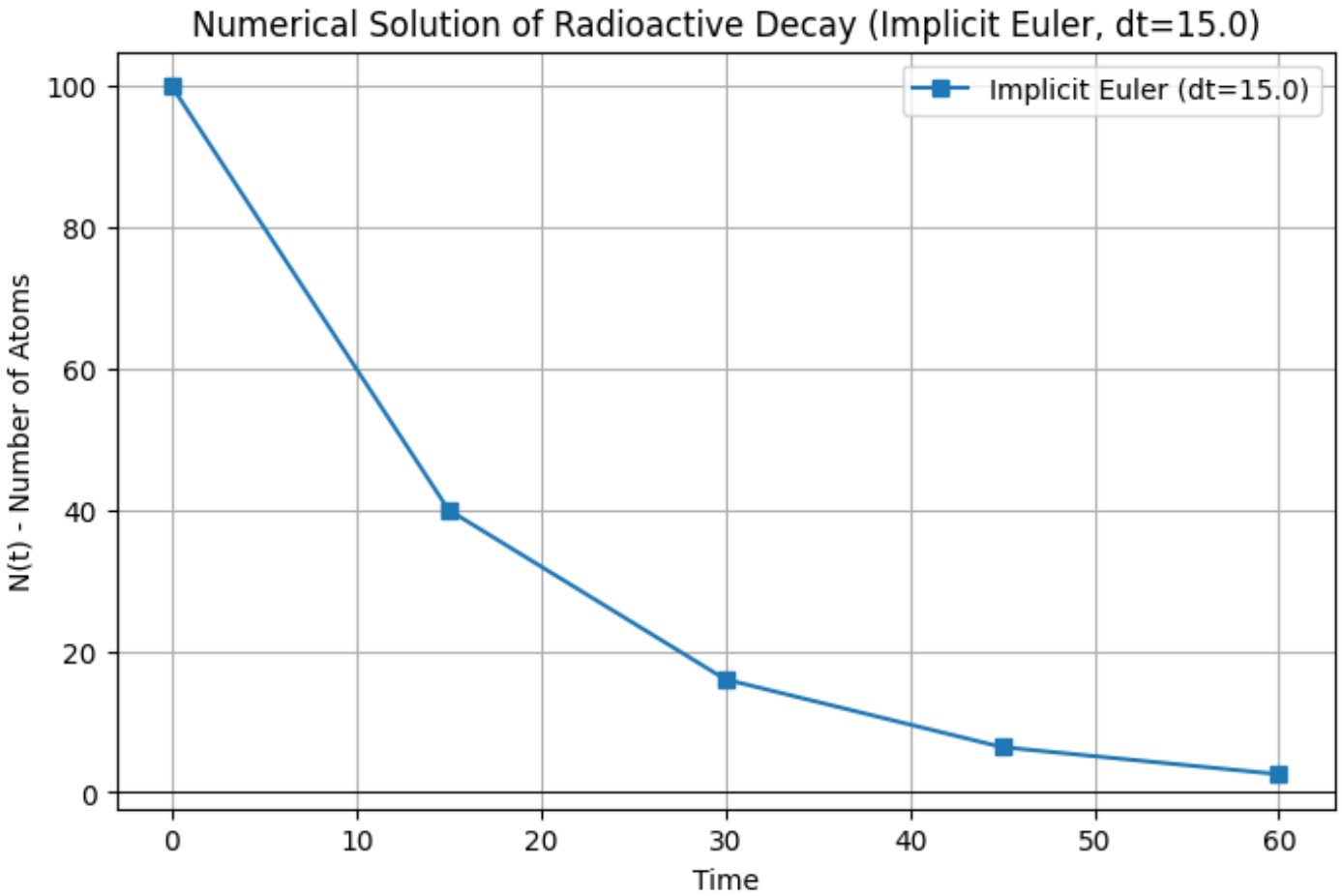
Python code

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

lambda_val = 0.5
dt = 0.2
T = 5
N0 = 1
times = np.arange(0, T + dt, dt)

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

Implicit Euler in Python

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

Implicit Euler in Python



Exercises – Numerical Schemes for ODEs

Exercise 1: Time Step Sensitivity (Implicit Euler)

- Use the implicit Euler method to solve the decay equation with:
  - $\Delta t = 0.5$, $\Delta t = 1.0$, and $\Delta t = 2.0$
- 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.

Table of Contents

Analytical vs Numerical (Explicit Euler)

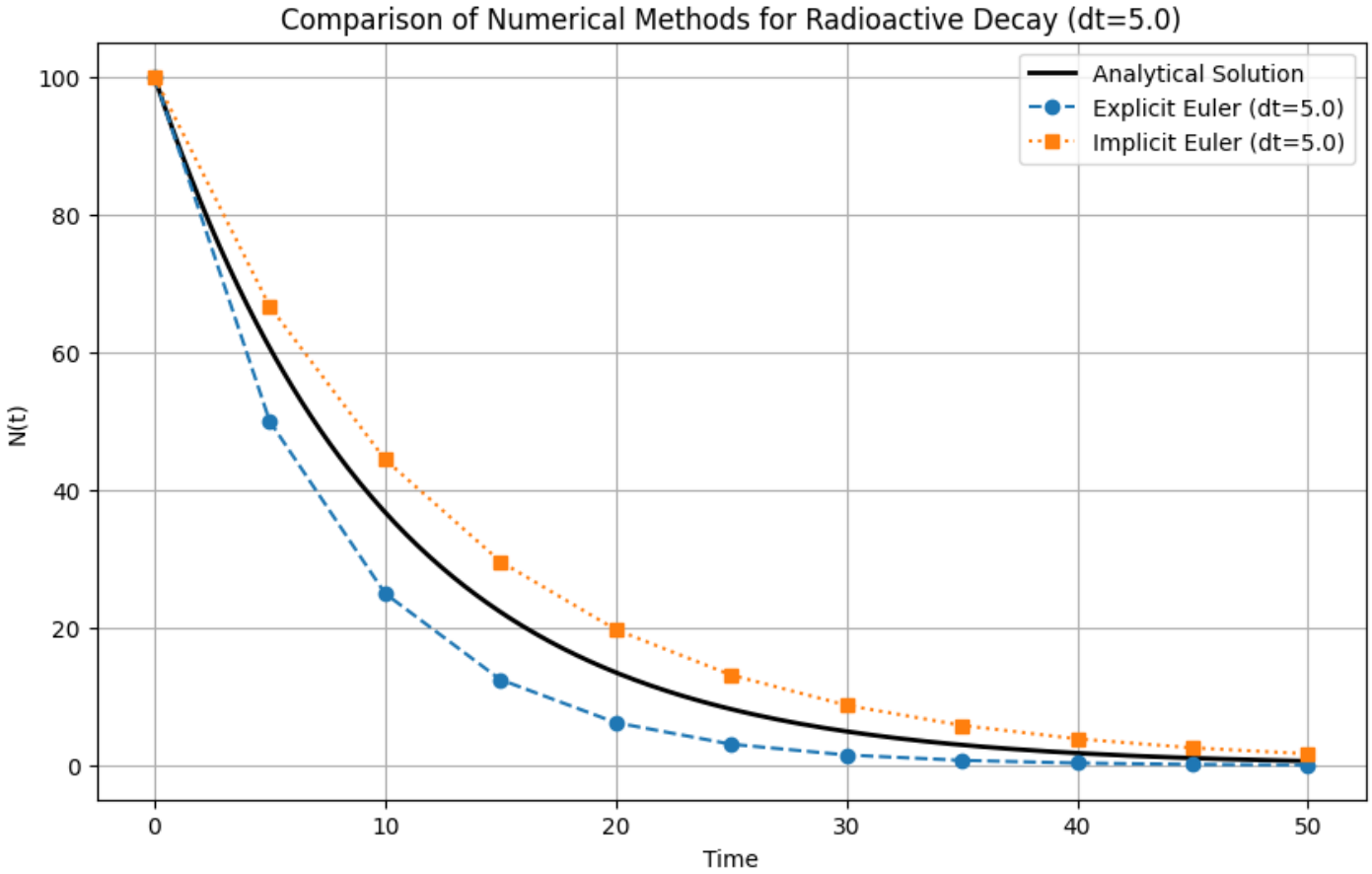
Python code

```
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

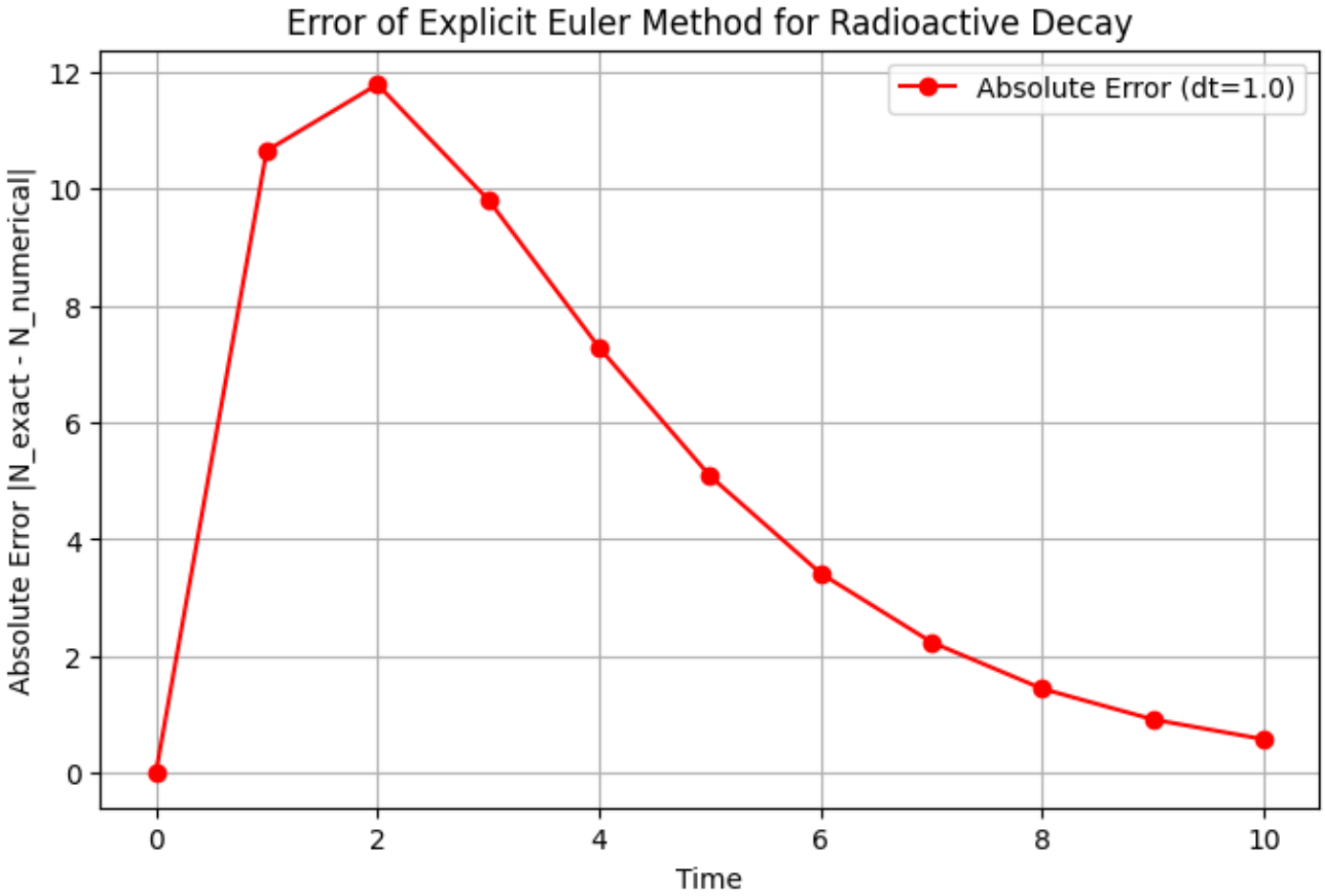
Plotting the Error in Python

```
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:

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

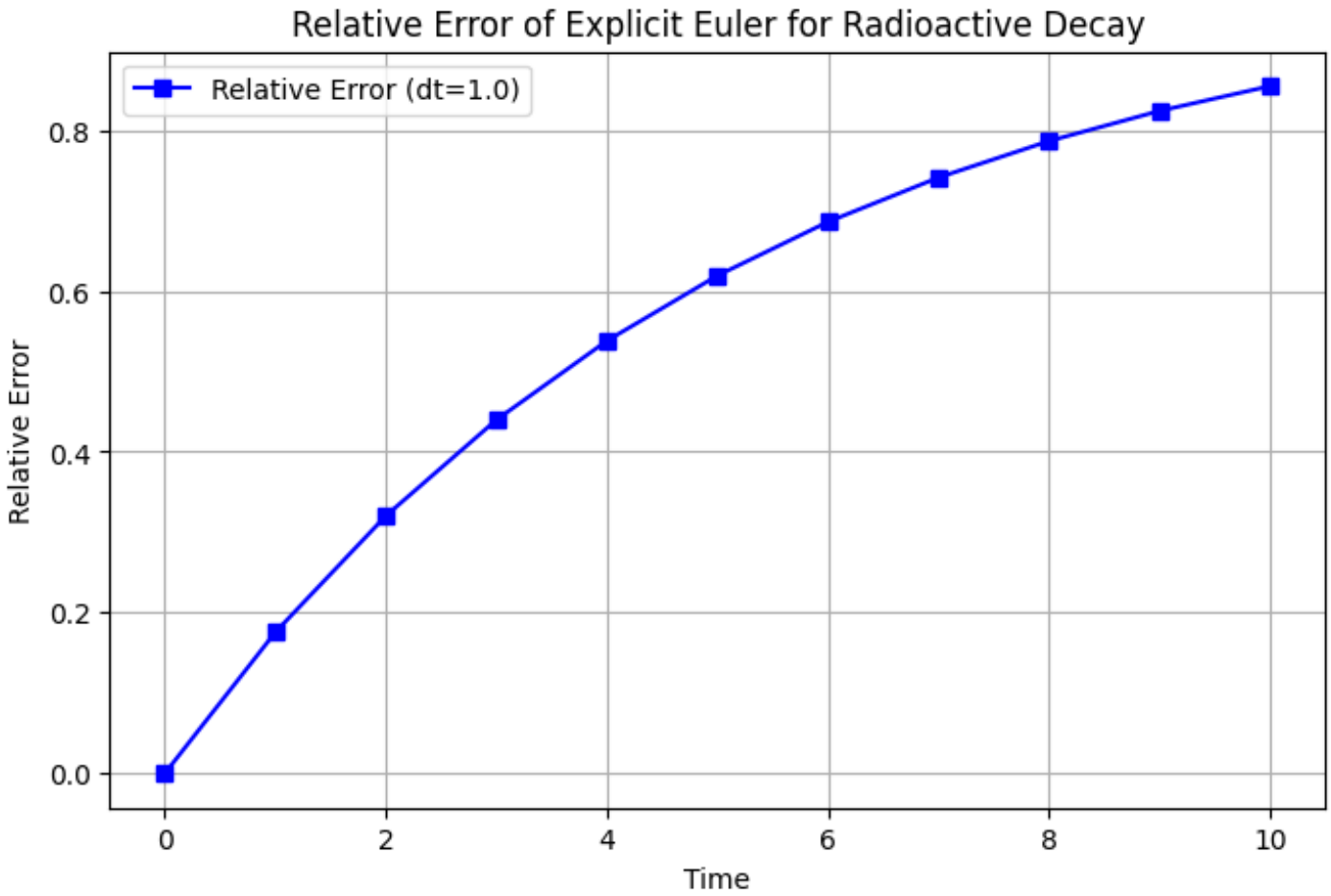
- Relative error is more informative when:
  - The quantity of interest becomes very small.
  - We want to assess the error proportionally to the expected value.
- Example: in radioactive decay, $N(t)$ tends to $0 \rightarrow$ relative error becomes more relevant.

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.

Table of Contents

Summary (Part 1): ODEs and Python Foundations

Understanding Differential Equations (ODEs):

- Ordinary Differential Equations (ODEs) describe the time evolution of systems and are fundamental in Earth Sciences.
- Examples:
  - Newton's Law of Cooling: $\frac{dT}{dt} = -k(T - T_a)$
  - Radioactive Decay: $\frac{dN}{dt} = -\lambda N$
- These equations model physical processes like heat transfer, population dynamics, and decay.
- Analytical (exact) solutions provide a formula for the solution but are often unavailable for complex, real-world problems, necessitating numerical approaches.

Summary (Part 2): Numerical Methods and Tools

Python Fundamentals

Python Fundamentals for Scientific Computing:

- **Variables** store data (numbers, text, etc.).
- **Lists** are ordered, mutable collections (e.g., '[1, 'rock', 3.0]').
  - Accessed via indexing (0-based, negative indexing) and slicing.
- **'for' loops** allow iteration over sequences (lists, 'range()').
- Understanding that variable assignment for mutable types (like lists) creates **references (not copies)** is crucial to avoid unintended side effects. Explicit copying (e.g., 'new_list = old_list[:]') is needed for independent objects.

Summary (Part 3): Numerical Methods and Tools

Numerical Methods for ODEs

Numerical Methods for ODEs:

- Goal: Approximate the solution of an ODE at discrete time steps Δt.
- **Explicit (Forward) Euler Method:**
  - Update rule: $y^{n+1} = y^n + \Delta t \cdot f(t^n, y^n)$
  - Simple to implement.
  - Conditionally stable: Δt must be small enough relative to the problem's characteristics (e.g., λ).
- **Implicit (Backward) Euler Method:**
  - Update rule: $y^{n+1} = y^n + \Delta t \cdot f(t^{n+1}, y^{n+1})$
  - Often requires solving an algebraic equation for y^{n+1} at each step.
  - Generally more stable than explicit methods, can handle larger Δt.
- Key Concepts:
  - **Accuracy:** How close the numerical solution is to the true solution.
  - **Stability:** Whether errors grow or remain bounded during computation.
  - **Error:** The difference between numerical and true solutions; depends on Δt and method order.

Summary (Part 4): Numerical Methods Tools

Essential Python Libraries for This Course:

- **NumPy:** For efficient N-dimensional arrays (‘ndarray‘) and fast, vectorized mathematical operations on numerical data.
- **Matplotlib (‘pyplot‘):** For creating a wide range of static, animated, and interactive visualizations (e.g., ‘plt.plot()‘, ‘plt.xlabel()‘, ‘plt.show()‘).

Next Steps: Building on these foundations to tackle more complex ODEs, explore higher-order numerical methods, and analyze their performance.