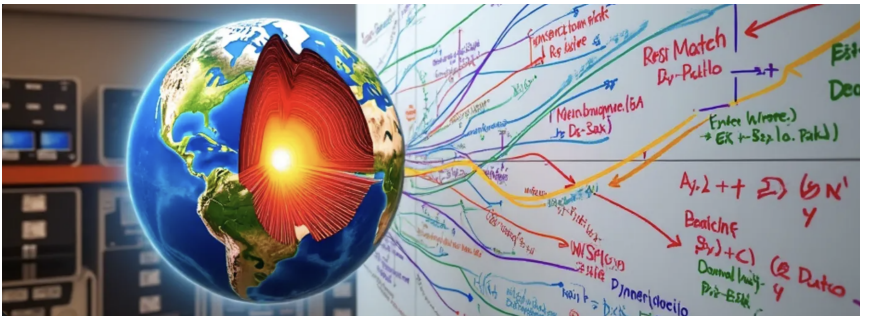
Principles of Numerical Modelling in Geosciences



Lecture Notes from a short course for the PhD in Geosciences and Environment

First Edition

Principles of Numerical Modelling in Geosciences

Lecture Notes from a short course for the PhD in Geosciences and Environment

First Edition

Mattia de' Michieli Vitturi
INGV Pisa, Italy

Wirehaired Editions

This volume serves as a companion to the course "**Principles of Numerical Modelling in Geosciences**" for the PhD in Geosciences and Environment, at the University of Pisa, Italy. The material presented herein has been developed by the author.

This book was typeset using LaTeX software.

Portions of the text and some figures were drafted and refined with the assistance of AI language models, including OpenAI's ChatGPT and Google's AI Studio, under the author's direct guidance and revision.

Preface

This volume serves as a companion to the PhD course, "**Principles of Numerical Modelling in Geosciences**", offered within the PhD Program in Geosciences and Environment at the University of Pisa during the academic year 2024/2025. The material presented herein has been developed and refined based on the lectures, discussions, and practical Python-based exercises of that course.

The primary motivation for compiling these notes into a more structured format is to provide students with a lasting resource that consolidates the foundational concepts and practical implementations covered. While a lecture setting allows for dynamic interaction and immediate clarification, a written volume offers the opportunity for deeper reflection, self-paced study, and a ready reference for future research endeavors.

The field of numerical modelling is vast and ever-evolving. This text aims to lay a basic groundwork, focusing on the fundamental principles that underpin more advanced techniques and specialized applications in the diverse disciplines of Earth Sciences. **Our approach prioritizes conceptual understanding and practical application over exhaustive mathematical formalism.** While ensuring correctness, the emphasis is on conveying the core ideas behind various *mathematical models (differential equations) derived from physical laws*, and then on understanding the *numerical methods used to approximate their solutions*, rather than on rigorous mathematical proofs for every numerical scheme. The intention is to equip students with the intuition to understand how physical laws governing Earth science processes translate into differential equations, and how these equations can then be tackled numerically, paying close attention to implementing schemes that are both accurate and stable. I hope that this volume will not only serve the students of the aforementioned course but also be of value to other graduate students and researchers beginning their journey into the computational aspects of geoscience.

I extend my gratitude to the students of the 2024/2025 cohort for their engagement and questions, which have invariably helped in shaping this material.

Finally, I would like to express my sincere gratitude to Prof. Carlo Doglioni, who served as President of the Istituto Nazionale di Geofisica e Vulcanologia (INGV) from April 2016 to February 2025. His unwavering commitment to fostering collaboration between INGV researchers and the academic world has greatly contributed to the scientific and educational mission of our institution. His support has been instrumental in enabling initiatives like this course and in strengthening the link

between research and higher education.

Mattia de' Michieli Vitturi
Pisa, July 2025

Table of Contents

1 Introduction **1**
1.1 The Dynamic Earth and the Language of Mathematics 1
1.2 Why Numerical Modelling? The Limits of Analytical Solutions 2
1.3 Philosophy and Objectives of this Volume 3
1.4 Structure of this Volume 4

2 Differential Equations **7**
2.1 Differential Equations in Earth Sciences 7
2.1.1 Example: Cooling Processes and Newton's Law 8
2.1.2 Example: Radioactive Decay 10
2.1.3 Ordinary vs. Partial Differential Equations: A First Look 12
2.1.4 Characterizing Differential Equations: Order and Linearity . . 13
Order of a Differential Equation 14
Linearity and Nonlinearity in Differential Equations 15
2.2 Introduction to Python for Numerical Modelling 16
2.2.1 Why Python for Scientific Computing? 16
2.2.2 Jupyter Notebooks: An Interactive Coding Environment 18
2.2.3 Python Syntax Basics 18
Variables and Assignment 19
Indentation 20
Comments 21
Line Length and Line Continuation 21
Defining Reusable Blocks of Code: Functions 23
2.2.4 Fundamental Python Data Structures: Lists 25
Introducing Python Lists 26
Accessing List Elements: Indexing 26
Extracting Sub-Lists: Slicing 27
Mutability: Lists Can Be Changed 29
Variables, References, and Copying Lists 30
2.2.5 Repeating Actions: Loops and the range() Function 31
The for Loop 32
The range() Function 32

2.3 Essential Libraries for Scientific Computing in Python . . . 33
2.3.1 NumPy: Numerical Computing with Arrays . . . 33
Python Lists vs. NumPy Arrays: Key Differences . . . 34
Creating NumPy Arrays . . . 35
NumPy Array Attributes . . . 37
Basic Operations: Vectorization . . . 37
2.3.2 Matplotlib: Visualizing Data . . . 38
Basic Plotting with Pyplot . . . 40
Key Elements of a Matplotlib Plot . . . 40
Plotting an Analytical ODE Solution . . . 41
2.4 Chapter Summary: Foundations and Tools . . . 43

3 Numerical Solution of ODEs 47
3.1 Numerical Solutions for ODEs . . . 48
3.1.1 Numerical Solution of Radioactive Decay: The Explicit Euler Method . . . 48
Python Implementation of the Explicit Euler Method . . . 50
Interpreting the Explicit Euler Discretization . . . 50
3.1.2 Numerical Stability of the Explicit Euler Method . . . 52
Experimenting with a Larger Time Step . . . 52
Understanding Numerical Instability . . . 53
3.1.3 The Implicit Euler (Backward Euler) Method . . . 54
Python Implementation of the Implicit Euler Method . . . 55
3.1.4 Assessing Numerical Solutions: Accuracy and Error . . . 56
Defining Numerical Error . . . 56
Visualizing Numerical vs. Analytical Solutions . . . 57
Plotting the Absolute Error . . . 58
Relative Error . . . 60
Numerical Accuracy and Order of a Method . . . 61
3.2 Challenges with Nonlinear ODEs . . . 62
3.2.1 Example: Water Infiltration into Soil – A Nonlinear ODE . . . 62
Explicit Euler Scheme for the Infiltration ODE . . . 63
Implicit Euler Scheme for the Infiltration ODE: The Challenge 64
3.2.2 Root-Finding: The Bisection Method . . . 65
The Core Idea of Bisection . . . 65
Algorithm for the Bisection Method . . . 65
A Simple Bisection Example in Python . . . 66
Implementing the Implicit Infiltration Solver with Bisection . 69

4 Spatial Derivatives and Finite Differences 79
4.1 From Time Derivatives to Spatial Derivatives: A Unified Concept . . . 79
4.2 The Finite Difference Method (FDM) for First Derivatives . . . 80
Forward Difference Scheme . . . 80

Backward Difference Scheme 81
Central Difference Scheme 81
4.3 Accuracy of Finite Difference Schemes 84
Taylor Series Expansion: The Foundation for Error Analysis . 84
Truncation Error of the Forward Difference Scheme 86
Truncation Error of the Backward Difference Scheme 86
Truncation Error of the Central Difference Scheme 87
4.4 Practical Examples 87
Printing Formatted Output in Python: f-strings 88
Example 1: Approximating the Derivative of $f(x) = x^2$ 88
Example 2: Approximating the Derivative of $f(x) = \sin(x)$ . . 90
Numerically Verifying the Order of Accuracy 92

5 Intro to PDEs and Conservation Laws 101
5.1 From ODEs to PDEs 101
5.1.1 Geoscience Examples of PDEs and Their Significance 102
The Heat (or Diffusion) Equation 102
Richards' Equation for Unsaturated Flow 104
The Advection-Diffusion Equation 105
5.2 Conservation Laws and Frames of Reference 105
5.2.1 The Continuum Hypothesis in Geosciences 106
5.2.2 Describing Motion: Eulerian and Lagrangian Viewpoints . . . 106
The Lagrangian Description 106
The Eulerian Description 107
The Material (or Substantial) Derivative: Connecting Lagrangian and Eulerian Views 108
5.2.3 Conservation Laws as the Foundation of Transport PDEs . . . 110
Accumulation Term 111
Generation Term 112
Net Influx Term 112
Advective and Diffusive Flux Components 113
5.2.4 The Roles of Advection and Diffusion in Mixing: Stirring vs. Homogenization 114
5.3 Deriving the General Scalar Transport Equation 116
5.4 Special Cases of the Transport Equation 119
5.4.1 The Mass Conservation Equation (Continuity Equation) 119
Lagrangian Form of the Continuity Equation and the Role of Divergence 120
5.4.2 The Heat (Diffusion) Equation 123

6 Numerical Solution of the Heat Equation 131
6.1 The Crucial Role of Boundary Conditions in PDE Problems 131
Common Types of Boundary Conditions 132
6.2 Numerical Discretization of the 1D Heat Equation 133
Spatial and Temporal Grids 133

Finite Difference Approximations for the Derivatives 134
The Forward-Time Central-Space (FTCS) Scheme: Formulation 135
6.3 Python Implementation for the 1D Heat Equation 136
Scenario 1: Dirichlet Boundary Conditions 136
6.3.1 Stability Analysis of the FTCS Scheme for the Heat Equation . 138
Visualizing Stable and Unstable FTCS Solutions 139
6.3.2 Practical Guidance: Selecting Step Sizes for FTCS 140
Considerations for Choosing Spatial Step Size (Δx) 140
Considerations for Choosing Time Step Size (Δt) 141
Scenario 2: Implementing a Neumann Boundary Condition . 142
6.3.3 Beyond Explicit Schemes: An Introduction to Implicit Methods for Diffusion 145

7 Numerical Solution of the Advection Equation 151
7.1 The Linear Advection Equation 152
7.1.1 What is Advection? The "Carrying Along" Process in Earth Systems 152
7.1.2 The 1D Linear Advection Equation: Formulation and Interpretation 154
7.1.3 Analytical Solution: Pure Translation and Characteristic Curves 155
Visualizing the Analytical Solution with Python 158
7.2 Boundary Conditions for Advection Problems 159
7.2.1 Recap of Common Boundary Condition Types 161
7.2.2 Inflow and Outflow Boundaries for Hyperbolic Problems . . . 161
7.2.3 Periodic Boundary Conditions: Closing the Loop 163
Relevance in Geosciences and Numerical Testing 163
Numerical Implementation of Periodic Boundary Conditions . 165
7.3 Finite Difference Schemes for Linear Advection 167
7.3.1 Recap: Discretizing Derivatives 167
7.3.2 Scheme 1: Forward Time, Centered Space (FTCS) 168
Python Implementation and Initial Observation of FTCS . . . 169
7.3.3 Scheme 2: Upwind Schemes - Aligning with the Flow Direction 172
Forward Time, Backward Space (FTBS) Scheme (for $u > 0$) . 173
Python Implementation and Initial Observation of FTBS (Upwind $u > 0$) 174
Upwind Scheme for $u < 0$: Forward Time, Forward Space (FTFS) 177
7.4 Stability and CFL Condition 177
7.4.1 What is Numerical Stability? A Critical Requirement 177
7.4.2 The Courant Number: Linking Space, Time, and Velocity . . . 178
7.4.3 The Courant-Friedrichs-Lewy (CFL) Condition 179
7.4.4 Stability of Specific Advection Schemes 180
FTCS (Forward Time, Centered Space) Scheme 180
Upwind Schemes (e.g., FTBS for $u > 0$) 181
7.4.5 Practical Implications of the CFL Condition and Choosing Δt 182

7.5 Numerical Artifacts 184
7.5.1 Numerical Diffusion (Artificial Dissipation) 184
7.5.2 Numerical Dispersion 185
7.6 Reflections on Advection Schemes and Model Selection 186
7.6.1 Performance Recap: FTCS vs. First-Order Upwind 186
7.6.2 The Inescapable Trade-off: Stability, Accuracy, and Numerical Artifacts 188
7.6.3 Choosing an Advection Scheme: Problem-Dependent Considerations 188
7.6.4 A Brief Outlook: Pathways to Improving Advection Schemes 189

8 Concluding Remarks **195**
8.1 Revisiting the Numerical Modeller's Workflow 195
8.2 Key Lessons and Broader Implications from Our Journey 197

Chapter 1

Introduction: Bridging Geoscience and Computation

1.1 The Dynamic Earth and the Language of Mathematics

The Earth is a profoundly dynamic system, characterized by a multitude of interacting processes occurring across vast scales of space and time. From the slow creep of tectonic plates and the convective overturn of the mantle to the rapid cooling of a lava flow, the infiltration of rainwater into soil, the dispersion of volcanic ash in the atmosphere, or the decay of radioactive isotopes within rocks, Earth scientists constantly seek to understand, quantify, and predict these diverse phenomena.

At their core, many of these processes can be described by fundamental physical laws – notably the conservation of mass, momentum, and energy. When these governing laws are expressed in the precise language of mathematics, they often take the form of **differential equations**. Simply put, a differential equation is a statement that links the rate at which a quantity changes to the value of the quantity itself, or to other related quantities. For instance, consider the population of a particular species of plankton in the ocean: the rate at which this population grows or shrinks might depend directly on how many plankton individuals are currently present (more individuals can reproduce more, but also consume resources faster or attract more predators). Similarly, the rate at which a hot rock body cools depends on how much hotter it currently is compared to its surroundings. These differential equations, therefore, capture the rates of change and interdependencies that define how Earth systems evolve.

We primarily encounter two main classes of differential equations in this context:

- **Ordinary Differential Equations (ODEs)** describe systems where the quantities of interest change with respect to a *single* independent variable. Most

commonly in our studies, this variable is time, t, leading to solutions of the form $u(t)$. An example is the cooling of a small, well-mixed magma body, where its temperature is assumed to be uniform throughout and changes only with time. In some simplified steady-state scenarios, an ODE might also describe variation along a single spatial dimension.

- **Partial Differential Equations (PDEs)** become essential when the *quantities of interest* vary with respect to *multiple* independent variables simultaneously. Common examples include variations over time t *and* one or more spatial dimensions (e.g., x; or x and y; or x, y, and z). Thus, a quantity like temperature might be described as $T(x, t)$, or concentration as $c(x, y, t)$. Such quantities, which have a value at every point in a region of space and at every moment in time, can be thought of as *distributions* or *fields* that evolves in time and space. The temperature distribution within a cooling lava dike as it solidifies, or the changing pattern of a pollutant concentration as it spreads through groundwater, are classic examples of phenomena whose description typically requires PDEs.

Concept Check

Try to give two examples from Earth systems where rates of change in space and time are related.

1.2 Why Numerical Modelling? The Limits of Analytical Solutions

While some idealized differential equations can be solved analytically, yielding exact, closed-form mathematical expressions for the solution (as will be seen for simple radioactive decay or Newton's cooling in the early parts of this volume), most real-world geoscience problems present complexities that render analytical approaches intractable. Factors such as:

- Complex or irregular geometries of the domain (e.g., a fractured aquifer system, a tortuous volcanic conduit).
- Heterogeneous material properties that vary in space (e.g., the permeability of different rock layers, the thermal conductivity of compositionally diverse magma).
- Nonlinear relationships between variables and their derivatives (e.g., hydraulic conductivity in unsaturated soils strongly depending on water content, or viscosity of magma depending on temperature and crystal content).
- The coupling of multiple physical processes (e.g., fluid flow, chemical reactions, and heat transfer occurring simultaneously).

make it impossible to find an exact formula for the solution.

This is where **numerical modelling** emerges as an indispensable tool for the modern Earth scientist. Numerical methods provide a pathway to obtain *approximate* solutions to these complex differential equations. By discretizing the continuous problem into a finite set of algebraic equations that can be solved on a computer, we can:

- Simulate processes for which no analytical solution is known.
- Explore the behavior of complex Earth systems under a wide range of conditions, allowing for "what if" scenario analysis.
- Test hypotheses about underlying physical mechanisms by comparing model outputs with observations.
- Gain deeper, quantitative insights into the dynamics of geological and environmental processes.
- Make quantitative predictions, which can be crucial for applications such as natural hazard assessment (e.g., forecasting volcanic eruption plumes, landslide runout) or subsurface resource management (e.g., optimizing oil and gas recovery from reservoirs, predicting reservoir behavior under production).

1.3 Philosophy and Objectives of this Volume

This volume aims to provide graduate students and researchers in Earth Sciences with a foundational understanding of the principles and practice of numerical modelling of systems described by differential equations. The focus is not merely on presenting a collection of numerical recipes, but rather on building an intuition for *why* certain methods work, their inherent assumptions and theoretical underpinnings, their practical limitations, and how to critically approach the task of simulation and interpret its results. We believe that a solid grasp of these fundamentals is essential for conducting robust and meaningful numerical research in the geosciences.

Our journey throughout this volume will cover:

1. A review of fundamental concepts of **Ordinary and Partial Differential Equations** most relevant to geoscience applications.
2. An introduction to basic **Python programming for scientific computation**, using powerful libraries such as NumPy for numerical operations and Matplotlib for visualization.
3. The core concepts of **numerical discretization**, with a primary focus on the versatile **finite difference method** to discretize first and second order derivatives.
4. The development, implementation, and analysis of numerical schemes for:

- Simple Ordinary Differential Equations (e.g., Explicit and Implicit Euler methods).
- Challenges posed by Nonlinear ODEs, introducing root-finding techniques like the bisection method.
- The 1D Diffusion (or Heat) Equation, a fundamental parabolic PDE.
- The 1D Linear Advection Equation, a key hyperbolic PDE.

5. An exploration of crucial properties of numerical methods, including **accuracy (truncation error, order of accuracy), stability (including the Courant-Friedrichs-Lewy – CFL – condition), convergence, and common numerical artifacts like numerical diffusion and dispersion.**

6. The critical role of **initial and boundary conditions** in defining well-posed problems and obtaining physically relevant solutions.

Concept Check

What makes a numerical model different from an analytical solution? When might an analytical solution become practically impossible?

The emphasis throughout is on building both a strong conceptual understanding and practical computational skills, enabling readers to confidently approach more complex modelling tasks in their own research fields.

1.4 Structure of this Volume

This volume is structured to build knowledge incrementally, guiding the reader from fundamental concepts to practical numerical implementation. The chapters are organized as follows:

Chapter 1: Introduction – Bridging Geoscience and Computation.
This current chapter discusses the pivotal role of differential equations in quantifying dynamic Earth systems and outlines why numerical modelling is an indispensable tool for modern geoscientists. It also sets out the philosophy and objectives of this volume.

Chapter 2: Differential Equations in Geosciences and Python Essentials.
The chapter delves into the nature of Ordinary and Partial Differential Equations (ODEs and PDEs) with geoscience examples like Newton's cooling and radioactive decay. It then provides an introduction to Python programming for scientific computation, covering syntax basics, fundamental data structures (notably lists), control flow, and a look at the essential libraries NumPy (for numerical arrays and vectorized operations) and Matplotlib (for data visualization).

Chapter 3: Numerical Solution of Ordinary Differential Equations.
This chapter focuses on developing and implementing our first numerical methods for ODEs. The Explicit Euler and Implicit Euler schemes are derived and applied to the radioactive decay problem. Crucial concepts of numerical stability (conditional vs. unconditional) and accuracy (absolute and relative error) are explored. The chapter also addresses the challenges posed by nonlinear ODEs, introducing the bisection method as a root-finding technique for solving implicit formulations, illustrated with a soil water drainage model.

Chapter 4: Approximating Spatial Derivatives – The Finite Difference Method.
This chapter transitions to the spatial domain, which is essential for PDEs. It introduces the Finite Difference Method (FDM) for approximating first-order spatial derivatives. Forward, backward, and central difference schemes are derived using Taylor series expansions, and their truncation error and order of accuracy are analyzed. Practical Python examples demonstrate these approximations and guide the reader through numerically verifying the order of accuracy using log-log plots.

Chapter 5: Partial Differential Equations and the Physics of Conservation.
This chapter lays the physical and mathematical groundwork for understanding many common PDEs. It revisits the distinction between ODEs and PDEs with more geoscience examples. Key concepts like the continuum hypothesis and Eulerian/Lagrangian frames of reference (including the material derivative) are discussed. The core of the chapter is the derivation of the general scalar transport equation from fundamental conservation laws applied to a control volume, detailing advective and diffusive flux components. This general equation is then specialized to derive the Continuity Equation (mass conservation) and the Heat (Diffusion) Equation (energy conservation).

Chapter 6: Numerical Solution of the 1D Heat (Diffusion) Equation.
The chapter applies the principles from previous chapters to numerically solve a fundamental parabolic PDE. It details the role and implementation of boundary conditions (Dirichlet and Neumann). The Forward-Time Central-Space (FTCS) explicit scheme is derived by approximating the second spatial derivative with central differences. A significant focus is placed on the conditional stability of the FTCS scheme (the $\alpha \leq 1/2$ criterion) and its practical implications. Python implementations are provided to simulate heat diffusion under different boundary conditions and to visualize the effects of stability. The chapter concludes with practical guidance on choosing step sizes and a brief look at implicit methods as an alternative for overcoming stability limitations.

Chapter 7: Numerical Solution of the Linear Advection Equation.
This chapter shifts focus to a fundamental first-order hyperbolic PDE: the 1D Linear Advection Equation. We will explore its physical basis as a model for pure transport phenomena. The chapter will cover the derivation of the linear advection equation and its exact analytical solution, which reveals the

concept of shape-preserving translation along characteristic curves. We will then develop and analyze explicit finite difference schemes for its numerical approximation, including the Forward-Time Centered-Space (FTCS) scheme and first-order Upwind methods. A significant emphasis will be placed on understanding the critical concept of numerical stability, particularly through the Courant-Friedrichs-Lewy (CFL) condition, which governs the choice of time and space steps for these explicit schemes. Common numerical artifacts associated with advection schemes, such as numerical diffusion and dispersion, will be examined. Periodic boundary conditions, which are invaluable for testing the intrinsic properties of advection algorithms, will be introduced and their numerical implementation discussed. Python examples and exercises will facilitate a practical exploration of these schemes, their stability, and their performance in simulating advective transport.

Chapter 8: Concluding Remarks and the Path Forward in Numerical Modelling. This final chapter provides a reflective overview of the core principles and practices of numerical modelling discussed throughout the volume. It revisits the comprehensive "Numerical Modeller's Workflow," from understanding the underlying physics of a geoscience problem to formulating mathematical models, selecting and implementing appropriate numerical methods, and critically verifying, validating, and interpreting simulation results. Key lessons regarding the nature of numerical approximations, the paramount importance of stability, the multifaceted aspects of accuracy (including numerical artifacts), and the problem-dependent nature of choosing the "best" scheme are synthesized. The chapter concludes by offering a perspective on further studies and more advanced topics within the broad and evolving field of computational geosciences, aiming to equip the reader for continued learning and application of numerical modelling in their research.

Each chapter involving numerical methods will be accompanied by Python code examples and suggestions for exercises to reinforce understanding and build practical computational skills. We hope this structure provides a clear and progressive path through the principles of numerical modelling.

Online Resources: Jupyter Notebooks

To further enhance this hands-on learning experience, this volume is complemented by a complete set of Jupyter Notebooks containing all the code examples.
The notebooks are available at: `github.com/demichie/Principles-of-Numerical-Modelling-in-Geosciences`

Chapter 2

Differential Equations in Geosciences and Python Essentials

This chapter serves as an introduction to the fundamental role of differential equations in describing Earth science phenomena and lays the groundwork for using Python as a computational tool for their numerical solution. We begin by exploring why differential equations are so prevalent in geosciences, illustrating with examples such as cooling processes and radioactive decay. Subsequently, we introduce the Python programming language, covering basic syntax, essential data structures like lists, and the powerful libraries NumPy and Matplotlib, which are indispensable for scientific computing and visualization.

2.1 The Language of Change: Differential Equations in Earth Sciences

Many of the processes that shape our planet and its environment involve quantities that change over time and/or space. Consider the transfer of heat within the Earth's interior, the flow of water in rivers and aquifers, the propagation of seismic waves during an earthquake, or the evolving concentration of a chemical species in a magmatic system. These dynamic phenomena are governed by fundamental physical laws, often expressed as laws of conservation (e.g., conservation of mass, momentum, and energy). When these physical laws are translated into precise mathematical statements, they frequently take the form of differential equations.

A differential equation is a mathematical equation that relates some quantity described by a function (e.g. the temperature $T(x,t)$ or the density $\rho(x,t)$) with its derivatives. In essence, derivatives represent rates of change. Therefore, differential equations describe how quantities, and the rates at which they change, are

interrelated. Understanding these equations allows Earth scientists to:

- Quantify the relationships between different physical variables.
- Simulate the behavior of complex natural systems under various conditions.
- Predict future states of these systems.
- Gain deeper insights into the underlying mechanisms driving geological and environmental processes.

Most geophysical processes can be described using either *ordinary differential equations* (ODEs), where the unknown function depends on a single independent variable (typically time), or *partial differential equations* (PDEs), where the unknown function depends on multiple independent variables (such as time and one or more spatial coordinates). Examples abound in Earth sciences, including:

- The cooling of lava flows after emplacement.
- The radioactive decay of isotopes used in geochronology.
- The flow of groundwater through porous rock.
- The large-scale circulation patterns in the atmosphere and oceans.

While the underlying physics may be well understood, and thus a governing equation may be written, obtaining an *analytical solution* (an exact, closed-form mathematical expression) to the resulting differential equations is often only possible for highly simplified scenarios. For the majority of real-world problems, which involve complex geometries, heterogeneous material properties, and nonlinear interactions, we must turn to **numerical methods** to find approximate solutions. This volume will guide you through the foundational principles of these numerical techniques.

2.1.1 Example: Cooling Processes and Newton's Law

Cooling is a ubiquitous process in Earth sciences. Examples range from the cooling of newly emplaced lava flows and volcanic deposits, to the cooling of magma and formation of plutons in the crust, to the secular cooling of the Earth's lithosphere over geological timescales. What these processes have in common is the transfer of heat from a hotter body or region to a cooler environment, resulting in a decrease in the temperature of the hotter material over time.

To describe such behavior mathematically, we can start with a simple model. Sir Isaac Newton, in the early 18th century, proposed an empirical law for cooling. He observed that the rate at which an object cools is proportional to the temperature difference between the object and its surroundings. Let's formalize this: let $T(t)$ be the temperature of an object at time t, and let T_{env} be the constant temperature of the surrounding environment. Let k be a positive cooling coefficient, which

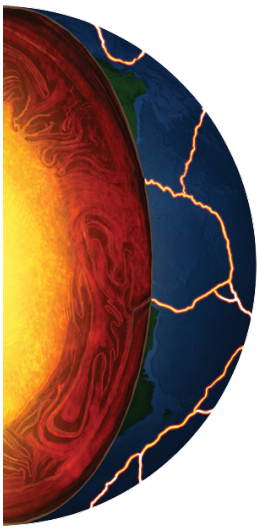


Figure 2.1. Conceptual illustration of the Earth's internal structure and ongoing cooling processes. Heat is transported from the hot interior (core and mantle) towards the cooler surface, driving geological phenomena like mantle convection and plate tectonics. Understanding these large-scale thermal processes often involves modelling heat transfer over geological timescales.

encapsulates how efficiently the object loses heat (depending on factors like its surface area, material properties, and the nature of the heat transfer mechanism).

Over a small time interval Δt, the change in the object's temperature, $\Delta T = T(t+\Delta t) - T(t)$, can be expressed according to Newton's observation as:

$$\Delta T = -k(T(t) - T_{\text{env}})\Delta t \tag{2.1}$$

This equation highlights a key aspect of Newton's model in its discrete form: for a given object (fixed k), the change in temperature ΔT over a small time interval Δt is **linearly proportional** to two factors:

- The **temperature difference** between the object and its environment at the start of the interval, $(T(t) - T_{\text{env}})$. A larger initial temperature difference leads to a proportionally larger temperature change (i.e., faster cooling if the object is hotter, or faster warming if it is cooler).
- The **duration of the time interval** itself, Δt. For a given temperature difference, a longer interval will naturally result in a proportionally larger total change in temperature.

The constant of proportionality that links the product of these two factors to ΔT is $-k$, where k is the positive cooling coefficient characteristic of the object and its interaction with the environment.

The negative sign indicates that if $T(t) > T_{\text{env}}$ (the object is hotter than its surroundings), then ΔT is negative, meaning the temperature decreases. Rearranging this equation, we get an expression for the average rate of temperature change over the interval Δt:

$$\frac{T(t+\Delta t)-T(t)}{\Delta t} = -k(T(t)-T_{\text{env}}) \tag{2.2}$$

This expression is a **finite difference** approximation of the rate of temperature change. It uses temperature values at two distinct time points.

To move from this discrete formulation to a continuous one, we consider what happens as the time interval Δt becomes infinitesimally small. In the limit as $\Delta t \to 0$, the left-hand side of Equation 2.2 becomes the definition of the derivative of T with respect to t:

$$\frac{dT}{dt} = \lim_{\Delta t \to 0} \frac{T(t+\Delta t)-T(t)}{\Delta t} \tag{2.3}$$

Thus, taking the limit of Equation 2.2, we obtain Newton's Law of Cooling in its differential form:

$$\frac{dT}{dt} = -k(T(t)-T_{\text{env}}) \tag{2.4}$$

This is a **first-order linear ordinary differential equation (ODE)**. Do not worry if terms like "first-order," "linear", or "ordinary differential equation" are new or seem daunting at this stage; we will clarify what these classifications mean and why they are important in the upcoming sections (specifically in Section 2.1.3 and Section 2.1.4). For now, the key takeaway is that we have derived a mathematical equation involving a derivative that describes the cooling process. This equation is widely used in Earth sciences to model various cooling (or heating) phenomena, such as lava temperature evolution, permafrost thawing, and the post-eruption thermal relaxation of volcanic systems. The crucial assumption here is that the temperature T is uniform throughout the object at any given time t; it is a function of time only. Such models are known as *lumped-parameter models*.

2.1.2 Example: Radioactive Decay

Another fundamental process in geosciences that is described by a simple ODE is radioactive decay. Radioactive isotopes are instrumental in **radiometric dating**, allowing us to determine the ages of rocks, minerals, and organic materials. Well-known examples include the decay of ^{238}U to ^{206}Pb (with a half-life of approximately 4.5 billion years, used for dating ancient rocks) and the decay of ^{14}C to ^{14}N (with a half-life of about 5730 years, used for dating younger, carbon-bearing materials).

In radioactive decay, the amount of the parent (radioactive) isotope decreases over time as it transforms into a daughter (stable or also radioactive) isotope. We want to model the evolution of $N(t)$, the number of atoms (or concentration) of the parent isotope at time t.

The physical principle governing radioactive decay is that the rate of decay (the number of atoms decaying per unit time) is proportional to the number of

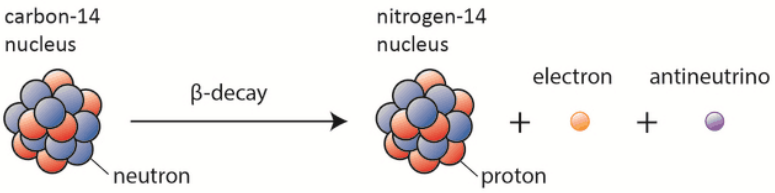


Figure 2.2. Schematic representation of the beta (β^-) decay of a Carbon-14 nucleus. A neutron within the Carbon-14 nucleus transforms into a proton, an electron (beta particle), and an antineutrino. This process changes the Carbon-14 atom (6 protons, 8 neutrons) into a Nitrogen-14 atom (7 protons, 7 neutrons). This specific decay is fundamental to radiocarbon dating.

radioactive atoms currently present. Let N_k be the number of parent atoms at a discrete time $t_k = k\Delta t$. If we assume that a constant fraction α of the existing atoms decays in each small time step Δt, then the number of atoms at the next time step, N_{k+1}, is:

$$N_{k+1} = N_k - \alpha N_k = (1-\alpha)N_k \tag{2.5}$$

This recursive relationship implies that if we start with N_0 atoms at $t = 0$, after k time steps, $N_k = (1-\alpha)^k N_0$.

To transition to a continuous model, we consider the rate of change. From Equation 2.5, the change in atoms over Δt is $\Delta N = N_{k+1} - N_k = -\alpha N_k$. The rate of change is thus

$$\frac{\Delta N}{\Delta t} = -\frac{\alpha}{\Delta t} N_k$$

Here, α represents the *fraction* of atoms present at time t_k that decay specifically *within the time interval* Δt. It is crucial to understand that α itself is not a fixed constant independent of Δt; rather, α is proportional to Δt for small Δt. If we halve the time interval Δt, we would expect the fraction of atoms decaying within that smaller interval also to roughly halve. More formally, we can write $\alpha \approx \lambda_{\text{intrinsic}} \Delta t$, where $\lambda_{\text{intrinsic}}$ is an intrinsic property of the isotope representing its decay probability per unit time.

We then define the **decay constant**, denoted by λ, as the limit of the ratio $\frac{\alpha}{\Delta t}$ as Δt approaches zero:

$$\lambda = \lim_{\Delta t \to 0} \frac{\alpha}{\Delta t} \tag{2.6}$$

It might seem concerning that the denominator Δt goes to zero. However, as explained above, the numerator α (the fraction decaying *in* that specific Δt) also goes to zero proportionally with Δt. Therefore, their ratio, λ, converges to a well-defined, finite, positive constant. This constant λ represents the intrinsic probability per unit time that any single unstable atom will undergo decay. For a given radioactive isotope, λ is a fundamental physical constant and is assumed not to

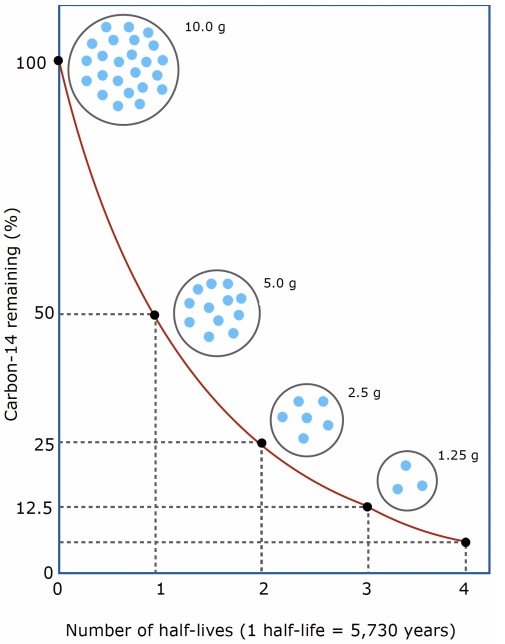


Figure 2.3. The characteristic exponential decay curve for Carbon-14. The graph shows the percentage of ^{14}C remaining as a function of the number of half-lives passed. Each point on the curve corresponds to a halving of the initial amount of the radioactive isotope, a fundamental principle in radiometric dating.

change over time or be affected by external conditions like temperature or pressure (within typical terrestrial environments).

Taking the limit as $\Delta t \to 0$ for the rate of change, $\frac{N(t+\Delta t)-N(t)}{\Delta t}$ becomes the derivative $\frac{dN(t)}{dt}$, and we obtain the differential equation for radioactive decay:

$$\frac{dN(t)}{dt} = -\lambda N(t) \tag{2.7}$$

This is also a first-order linear ODE. Given an initial number of atoms $N(0) = N_0$, its analytical solution is:

$$N(t) = N_0 e^{-\lambda t} \tag{2.8}$$

This exponential decay model is the cornerstone of many geochronological techniques.

2.1.3 Ordinary vs. Partial Differential Equations: A First Look

The two examples we have discussed, Newton's Law of Cooling (Equation 2.4) and Radioactive Decay (Equation 2.7), are both **Ordinary Differential Equations (ODEs)**. An ODE involves one or more derivatives of a function with respect to a *single independent variable*. In our examples, this variable was time t, and the unknown functions ($T(t)$, $N(t)$) depended only on time. ODEs are widely used in Earth Sciences to model phenomena where spatial variations are considered negligible or are averaged out in time, such as:

- The evolution of the bulk concentration of a chemical species in a well-mixed reactor.
- Simple models of population dynamics.
- The kinetic rates of chemical reactions.

We will focus first on ODEs and methods for their numerical solution.

However, many, if not most, Earth science phenomena involve changes that depend on *multiple independent variables*, typically time t *and* one or more spatial coordinates (e.g., x, y, z). Equations involving partial derivatives with respect to multiple independent variables are called **Partial Differential Equations (PDEs)**. For example, the cooling of a large lava flow is not uniform; the temperature T will vary with depth x and time t, $T(x,t)$. A simplified 1D model for this (the heat equation) might look like:

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \tag{2.9}$$

Here, $\frac{\partial T}{\partial t}$ is a partial derivative with respect to time (how T changes at a fixed point x), and $\frac{\partial^2 T}{\partial x^2}$ is a partial second derivative with respect to space (related to how T varies spatially at a fixed time). PDEs are essential to describe a vast range of complex processes:

- Heat diffusion and advection in the Earth's crust and mantle.
- Groundwater flow in aquifers.
- The deformation of rocks under stress (elasticity, viscoelasticity).
- Atmospheric and oceanic circulation dynamics.
- The propagation of seismic waves.

In this course, we will start by building our numerical toolkit with ODEs and then extend these concepts to tackle PDEs.

Concept Check

Can you think of a geoscience process where something changes over time or space? What changes, and what causes the change?

2.1.4 Characterizing Differential Equations: Order and Linearity

Before we delve into the computational tools for solving differential equations, it is helpful to introduce a few fundamental concepts used to classify and describe them. This is not intended to be a formal or exhaustive mathematical treatment, but rather an introduction to some useful definitions and distinctions that will arise as

we encounter various equations relevant to Earth science problems. We will focus these definitions in the context of Partial Differential Equations (PDEs), though similar concepts apply to ODEs.

Order of a Differential Equation

The **order** of a differential equation (either ODE or PDE) is determined by the **highest-order derivative** present in the equation.

- **First-Order Equations:** The highest derivative appearing is a first derivative.
  - *Example (ODE):* Radioactive decay, $\frac{dN}{dt} = -\lambda N$. (Highest derivative is dN/dt).
  - *Example (PDE):* The 1D linear advection equation (which we will study in detail later), $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$. This equation describes the transport of a quantity ϕ (like the concentration of a pollutant or a tracer) by a constant velocity u in one spatial dimension x, without any change in the shape of the ϕ profile. (The highest derivatives are first-order: $\partial \phi/\partial t$ and $\partial \phi/\partial x$).
- **Second-Order Equations:** The highest derivative appearing is a second derivative.
  - *Example (PDE):* The 1D heat (diffusion) equation, $\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$. (Highest derivative is $\partial^2 T/\partial x^2$).
  - *Example (PDE):* The 1D wave equation, $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$. This equation governs the propagation of various types of waves. In Earth sciences, it can describe, for example, the displacement u of particles in a medium as a seismic wave (like a P-wave or S-wave simplified to 1D) travels through it with velocity c. It also models phenomena like waves on a string or acoustic waves. Unlike diffusion which smooths things out, wave propagation typically preserves or transports disturbances. (The highest derivatives, $\partial^2 u/\partial t^2$ and $\partial^2 u/\partial x^2$, are second order in both time and space).
- **Higher-Order Equations:** Equations involving third, fourth, or even higher derivatives exist in specialized applications (e.g., equations describing the bending of elastic plates in geodynamics can be fourth-order PDEs).

The order of an equation often influences the number of boundary and/or initial conditions required to specify a unique solution, and it can also affect the complexity of numerical methods needed to solve it. For instance, approximating a second derivative numerically typically requires information from more grid points than approximating a first derivative.

Linearity and Nonlinearity in Differential Equations

Another crucial distinction is whether a differential equation is **linear** or **nonlinear**. This property significantly impacts both the possibility of finding analytical solutions and the behavior and complexity of numerical solutions.

A differential equation is said to be **linear** if the unknown function (e.g., $T(x,t)$, $\phi(x,t)$) and all of its derivatives appear in the equation in a linear fashion. This means they are:

- Raised to the power of one (or zero, i.e., not present).
- Not multiplied by each other (e.g., no terms like $T\frac{\partial T}{\partial x}$ or $(\frac{\partial \phi}{\partial x})^2$).
- Not arguments of nonlinear functions (e.g., no terms like $\sin(T)$ or e^{ϕ}).

The coefficients multiplying the unknown function or its derivatives can be constants or functions of the independent variables (e.g., x,t) only.

Linear equations possess an important property called the *principle of superposition*: if u_1 and u_2 are two solutions to a linear homogeneous equation, then any linear combination $c_1u_1 + c_2u_2$ (where c_1, c_2 are constants) is also a solution. This property is fundamental to many analytical solution techniques (like Fourier series).

Examples of Linear PDEs:

- The 1D Heat Equation with constant κ: $\frac{\partial T}{\partial t} = \kappa\frac{\partial^2 T}{\partial x^2}$. (Here, T, $\frac{\partial T}{\partial t}$, and $\frac{\partial^2 T}{\partial x^2}$ all appear linearly.)
- The 1D Linear Advection Equation with constant u: $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$.

A common point of confusion can arise from terms like the second derivative, $\frac{\partial^2 T}{\partial x^2}$. It is important to remember that **differentiation itself is a linear operation**. This means that the derivative of a sum of functions is the sum of their derivatives. This property holds for derivatives of any order. For instance, for two functions $f(x)$ and $g(x)$ and constants c_1, c_2:

$$\frac{d}{dx}(c_1 f + c_2 g) = c_1\frac{df}{dx} + c_2\frac{dg}{dx} \quad \text{(Linearity of the first derivative)}$$
$$\frac{d^2}{dx^2}(c_1 f + c_2 g) = c_1\frac{d^2 f}{dx^2} + c_2\frac{d^2 g}{dx^2} \quad \text{(Linearity of the second derivative)}$$

Therefore, a term like $\frac{\partial^2 T}{\partial x^2}$ in an equation is a linear term with respect to the unknown function T.

An equation that is not linear is called **nonlinear**. Nonlinearities can arise in many ways:

- The unknown function or its derivatives are raised to a power other than one (e.g., $u^2 + \frac{\partial u}{\partial x}$ which contains u^2, making it nonlinear, or a term like $(\frac{\partial u}{\partial x})^2$).

- Products of the unknown function and its derivatives (e.g., $\phi \frac{\partial \phi}{\partial x}$).
- The unknown function appears as the argument of a nonlinear function (e.g., $\sin(\phi)$).
- Coefficients that depend on the unknown function itself (e.g., in Richards' equation for unsaturated flow, the hydraulic conductivity $K(\theta)$ depends on the water content θ, making the term $\frac{\partial}{\partial z}(K(\theta)\frac{\partial \psi}{\partial z})$ nonlinear).

Examples of Nonlinear PDEs in Geosciences:

- **Richards' Equation for unsaturated flow (simplified):** $\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[D(\theta)\frac{\partial \theta}{\partial z}\right]$. The diffusivity $D(\theta)$ depends on θ, making the equation nonlinear.
- **Navier-Stokes Equations for fluid flow (general form):** These equations contain advective terms like $u\frac{\partial u}{\partial x}$ (where u is a velocity component), which are nonlinear. These govern atmospheric circulation, ocean currents, and magma flow.
- **Nonlinear heat conduction:** If thermal conductivity $\kappa(T)$ depends on temperature, the heat equation $\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}(\kappa(T)\frac{\partial T}{\partial x})$ becomes nonlinear. This is common for rocks over large temperature ranges.

Nonlinear equations are generally much harder to solve analytically than linear ones. Their solutions can exhibit complex behaviors such as shock waves, solitons, or chaotic dynamics, which are not typically found in linear systems. Numerically, nonlinearities often require iterative solution techniques at each time step for implicit methods (as we we will see in the next chapters with the bisection method) or can lead to more restrictive stability conditions for explicit methods.

Understanding the order and linearity of a differential equation is a first step in choosing appropriate analytical or numerical solution strategies and anticipating the nature of its solutions.

2.2 Introduction to Python for Numerical Modelling

To implement and explore numerical methods, a suitable programming environment is essential. In this course, and throughout this volume, we will use **Python**. Python is a versatile, high-level, open-source programming language that has gained immense popularity in scientific computing and data science, including many applications within the Earth sciences.

2.2.1 Why Python for Scientific Computing?

There are several compelling reasons for choosing Python:

Figure 2.4. The Python language logo. Python's combination of readability and powerful scientific libraries makes it a popular choice for numerical modelling.

- **Readability and Simplicity:** Python's syntax is designed to be clear and intuitive, resembling plain English in many aspects. This makes it relatively easy to learn, write, and maintain code, even for those new to programming.
- **Extensive Scientific Libraries:** Python boasts a rich ecosystem of powerful libraries specifically designed for scientific and numerical tasks. Key among these are:
  - **NumPy (Numerical Python):** Provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays efficiently. We will use NumPy extensively.
  - **SciPy (Scientific Python):** Builds on NumPy and offers a vast collection of algorithms and tools for tasks such as optimization, integration, statistics, signal processing, and more.
  - **Matplotlib:** A comprehensive library for creating static, animated, and interactive visualizations in Python. Essential for plotting results and understanding data.
  - **Pandas:** Provides high-performance, easy-to-use data structures (like DataFrames) and data analysis tools, particularly useful for working with tabular data.
- **Large and Active Community:** Python has a vast global community of users and developers. This translates to abundant online resources, tutorials, forums for help, and a continuous stream of new packages and improvements.
- **Interpreted Language:** Python is an interpreted language, which allows for interactive coding and rapid prototyping. This is particularly beneficial when exploring new ideas or debugging code.
- **Versatility and Integration:** Python can be used for a wide range of tasks beyond numerical computation, including data processing, web development, machine learning, and automation. It can also interface with code written in other languages like C++ or Fortran.
- **Open Source and Free:** Python and its core scientific libraries are free to use, distribute, and modify, making them accessible to everyone.

In the Earth sciences, Python is increasingly used for data analysis (e.g., processing satellite imagery, geochemical datasets, or time series from sensors), numerical modeling (e.g., simulating climate dynamics, volcanic processes, or groundwater flow), and a wide array of visualization tasks (e.g., creating maps, cross-sections, or plots of model output).

2.2.2 Jupyter Notebooks: An Interactive Coding Environment

Throughout this volume, we will primarily interact with Python code using **Jupyter Notebooks**. A Jupyter Notebook is a web-based interactive computing environment that allows you to create and share documents containing live code, equations, visualizations, and narrative text.

Key features and benefits of Jupyter Notebooks include:

- **Combining Code, Text, and Output:** Notebooks are organized into "cells." Code cells allow you to write and execute Python code, and the output (text, plots, etc.) is displayed directly below the cell. Markdown cells allow you to write formatted text, include mathematical equations (using LaTeX syntax), images, and links, providing context and explanation for your code.

- **Interactive Exploration:** You can execute code cells individually and in any order, making it easy to experiment, test ideas, and debug step-by-step.

- **Documentation and Storytelling:** Notebooks are excellent for creating a complete record of an analysis or modeling workflow, documenting not just the code but also the reasoning, methodology, and interpretation of results.

- **Reproducibility and Sharing:** A saved notebook (.ipynb file) captures the code, output, and narrative. Notebooks can be easily shared with collaborators or published online, allowing others to understand and reproduce your work.

- **Support for Many Languages:** While we will use Python, Jupyter Notebooks can support kernels for many other programming languages.

You can run Jupyter Notebooks locally on your computer after installing Python and the Jupyter package (often included with distributions like Anaconda), or you can use cloud-based services like Google Colaboratory (Colab) or run them within Integrated Development Environments (IDEs) like Visual Studio Code.

2.2.3 Python Syntax Basics

Let's briefly review some fundamental aspects of Python syntax, many of which you might have encountered if you've used Python before.

Variables and Assignment

Variables are used to store data values in a program. Think of them as named containers or labels for information that your program needs to remember and work with. In Python, you assign a value to a variable using the equals sign (‘=‘).

A key characteristic of Python is that it is **dynamically typed**. This means you do not need to declare the type of a variable (e.g., whether it will hold an integer, a floating-point number, or text) before you use it. Python automatically infers the variable’s type from the value assigned to it at runtime. Furthermore, a variable in Python can even change its type if it is reassigned a value of a different type later in the program.

This contrasts with **statically typed** languages like C++ or Fortran, which are also commonly used in scientific computing. In those languages, you typically must explicitly declare the type of a variable before you can assign a value to it. For example:

- **In C++ (conceptual example):**

```
// You must declare the type first
int count;          // Declares 'count' as an integer
double temperature; // Declares 'temperature' as a
                    // double-precision float

// Then you can assign values
count = 10;
temperature = 25.5;

// temperature = "hot"; // This would cause a compile-time
                        // error in C++ because "hot" (a
                        // string) cannot be assigned
                        // to a variable declared as
                        // double.
```

Listing 2.1. Variable declaration and assignment in C++.

- **In Fortran 90/95 (conceptual example):**

```
! You must declare the type first
INTEGER :: count
REAL :: temperature

! Then you can assign values
count = 10
temperature = 25.5

! This would also cause a compile-time error
temperature = "hot"
```

Listing 2.2. Variable declaration and assignment in Fortran.

In these statically typed languages, the variable ‘temperature‘ is fixed to hold only floating-point numbers once declared. Attempting to assign a string to it would result in an error, typically caught by the compiler before the program even runs.

Python’s dynamic typing, on the other hand, offers more flexibility during development, as demonstrated below.

```
# Assigning different data types to variables
x = 3.0           # x is now a floating-point number (float)
name = "basalt"   # name is now a string (str)
is_volcano = True # is_volcano is now a boolean (bool)
count = 10        # count is now an integer (int)

# You can print the value and type of a variable
print(x, type(x))
print(name, type(name))
```

Listing 2.3. Variable assignment in Python

Indentation

Unlike many other programming languages that use curly braces or keywords to define blocks of code (e.g., for loops, function definitions, conditional statements), Python uses **indentation**. Consistent indentation (typically 4 spaces per level) is crucial and is part of the syntax. Incorrect indentation will lead to an ‘IndentationError‘.

```
# Example of a for loop and an if statement
for i in range(3):  # The colon indicates the start of a block
    print("Outer loop, i =", i) # This line is indented, part
    of the for loop
    if i % 2 == 0: # Another block starts here
        print("   i is even") # Further indented, part of the
    if block
    else:
        print("   i is odd")  # Also part of the else block
print("Loop finished") # This line is not indented, so it's
    outside the for loop
```

Listing 2.4. Python indentation example

A common source of indentation issues, especially when moving code between different text editors or operating systems, or when collaborating with others, is the mixing of tabs and spaces. While a tab character might *visually* appear as a certain number of spaces in one editor (e.g., 4 spaces or 8 spaces), this visual representation is not standardized. Python 3 disallows mixing tabs and spaces for indentation in a way that is ambiguous. However, the best practice, strongly recommended by the official Python style guide (PEP 8), is to use only spaces for indentation. The most common convention is to use **4 spaces per indentation level**.

Using spaces exclusively ensures that the code’s structure is interpreted consistently across all editors and platforms. Most modern text editors and Integrated Development Environments (IDEs) designed for Python can be configured to automatically insert 4 spaces when you press the Tab key, making it easy to adhere to

this convention. Mixing tabs and spaces can lead to subtle 'IndentationError's or, worse, code that runs but has an incorrect logical structure because Python interprets the indentation differently than what you see visually. Therefore, it is highly advisable to configure your editor to use spaces for indentation and to be consistent.

Comments

Comments are used to explain your code and make it more readable. Python ignores comments during execution.

- Single-line comments start with the hash symbol (#). Anything after # on that line is a comment.
- Multi-line comments can be created by using multiple # symbols, or by enclosing the comment block in triple quotes (`"""..."""` or `'''...'''`). Triple-quoted strings are also used for "docstrings," which are special comments used to document functions, classes, and modules.

```
# This is a single-line comment explaining the next line
gravity_g = 9.81  # m/s^2, standard gravity

"""
This is a
multi-line comment using triple quotes.
It can span several lines.
"""

def calculate_area(radius):
    """This is a docstring. It explains what the function does
    ."""
    return 3.14159 * radius**2
```

Listing 2.5. Python comments

Good commenting habits are essential for writing understandable and maintainable code.

Line Length and Line Continuation

Readability is a core tenet of Python programming. While Python does not strictly enforce a maximum line length, the official Python style guide, PEP 8, recommends limiting all lines to a maximum of 79 characters for code (and 72 for docstrings/-comments). This convention helps ensure that code is easily readable on a wide variety of screen sizes and when multiple files are open side-by-side. Long lines can be difficult to read and comprehend.

When a logical line of code is longer than the recommended length, Python provides ways to split it across multiple physical lines. This is known as **line continuation**.

There are two main ways to achieve line continuation:

1. **Implicit Line Continuation (Preferred):** Python automatically allows line breaks inside parentheses `()`, square brackets `[]`, and curly braces `{}`. This is the preferred method as it is generally more readable and less prone to errors.

```
# Long list definition
my_long_list = [
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, # Comments can also be on
    continued lines
    ]

# Long function call
result = some_function_with_many_arguments(
    argument1, argument2, argument3,
    keyword_arg1="value1", keyword_arg2="value2"
    )

# Long arithmetic expression within parentheses
total_value = (first_value + second_value + third_value
               - fourth_value + fifth_value)

# Long dictionary definition
my_dictionary = {
    "key1": "
    very_long_value_that_makes_the_line_exceed_limit",
    "key2": "another_value",
    }
```

Listing 2.6. Implicit line continuation in Python.

When using implicit line continuation, it's good practice to align the continued part of the line either vertically with the opening delimiter (parenthesis, bracket, brace) or by using a "hanging indent" (typically an extra 4 spaces relative to the previous line).

2. **Explicit Line Continuation (Backslash):** You can also use a backslash character (\) at the end of a line to indicate that the logical line continues on the next physical line.

```
# Long assignment statement using backslash
x = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
print(x)

# Long if condition
a_variable = 10
another_variable = 20
yet_another_variable = 30
if a_variable > 5 and \
   another_variable < 30 and \
   yet_another_variable == 30:
    print("All conditions met!")
```

Listing 2.7. Explicit line continuation using backslash.

While the backslash method works, it is generally less preferred than implicit continuation because it can be more fragile (e.g., an accidental space after the backslash will break the continuation) and sometimes less readable. Implicit continuation within delimiters is the cleaner and more Pythonic approach when possible.

Adhering to reasonable line length limits and using line continuation appropriately significantly improves the readability and maintainability of your Python code.

Defining Reusable Blocks of Code: Functions

As programs become more complex, it is often necessary to execute a particular sequence of operations multiple times, or to organize code into logical, manageable units. **Functions** provide the mechanism for doing this. A function is a named block of code that performs a specific task and can be "called" (executed) from other parts of the program as needed. Using functions promotes code reusability, readability, and modularity.

Defining a Function In Python, you define a function using the `def` keyword, followed by the function name, a pair of parentheses `()` which may enclose one or more parameters (input values for the function), and a colon `:`. The block of code that constitutes the body of the function must be indented. The basic syntax is:

```
def functionName(parameter1, parameter2, ...):
    """
    Optional: This is a docstring.
    It describes what the function does, its parameters,
    and what it returns.
    """
    # Body of the function: statements to perform the task
    # ... calculations ...
    # result = ...
    return result # Optional: returns a value back to the caller
```

- `def`: Keyword indicating a function definition.
- `functionName`: The name you give to your function (should follow standard variable naming conventions, typically `snake_case` or `camelCase` for functions).
- `parameter1, parameter2, ...`: These are the input arguments that the function accepts. They act as local variables within the function. This is a crucial concept: it means these variables exist only within the function's self-contained "workspace". They are created when the function is called and are destroyed once the function finishes, and they will not conflict with variables of the same name that might exist elsewhere in your program. A function can have zero or more parameters.

- `"""Docstring"""`: An optional string literal that serves as documentation for the function. It's good practice to include a docstring explaining the function's purpose, arguments, and what it returns.

- **Indented Body:** All statements that are part of the function must be indented relative to the `def` line.

- `return result`: The `return` statement is used to send a value (or values) back to the part of the code that called the function. A function can return one value, multiple values (as a tuple), or no value (in which case it implicitly returns `None`). If there is no `return` statement, or just `return` without a value, the function also returns `None`.

Calling a Function Once a function is defined, you can execute it by "calling" it by its name, followed by parentheses containing any required arguments.

```
# Define a function to calculate the area of a circle
def calculateCircleArea(radius):
    """Calculates the area of a circle given its radius."""
    piApprox = 3.14159 # Using camelCase for local variable
    area = piApprox * radius**2
    return area

# Call the function with different arguments
radiusOne = 5.0 # Using camelCase
areaOne = calculateCircleArea(radiusOne)
# Using print with multiple arguments, separated by commas
print("The area of a circle with radius", radiusOne, "is",
    areaOne)
# Note: This will print with default spacing. We'll see f-
    strings for better formatting later.

radiusTwo = 2.5 # Using camelCase
areaTwo = calculateCircleArea(radiusTwo)
# Alternatively, converting numbers to strings and
    concatenating
# (This is more verbose than using f-strings or multiple print
     arguments)
print("The area of a circle with radius " + str(radiusTwo) + "
     is " + str(areaTwo))

# Example of a function without a specific return value (
    implicitly returns None)
def greet(name):
    """Prints a greeting message."""
    print("Hello,", name, "!") # Using multiple arguments for
    print

greet("World")
returnedValue = greet("Student") # greet() is called, message
    printed
```

```
print("Value returned by greet():", returnedValue) # Will
    print: None
```

Listing 2.8. Defining and calling simple Python functions. The example uses `camelCase` for some variable names (e.g., `radiusOne`, `piApprox`), a common naming convention where the first word is lowercase and each subsequent word starts with a capital letter. The official Python style guide (PEP 8) recommends an alternative called `snake_case`, which uses all lowercase letters separated by underscores (e.g., `radius_one`).

Functions are a cornerstone of structured programming. We will use them extensively to encapsulate calculations for our numerical methods, such as defining the function $G(X)$ whose root we seek with an iterative method, or to define the right-hand side of an ODE. This makes our main simulation scripts cleaner and the specific mathematical operations more reusable.

2.2.4 Fundamental Python Data Structures: Lists

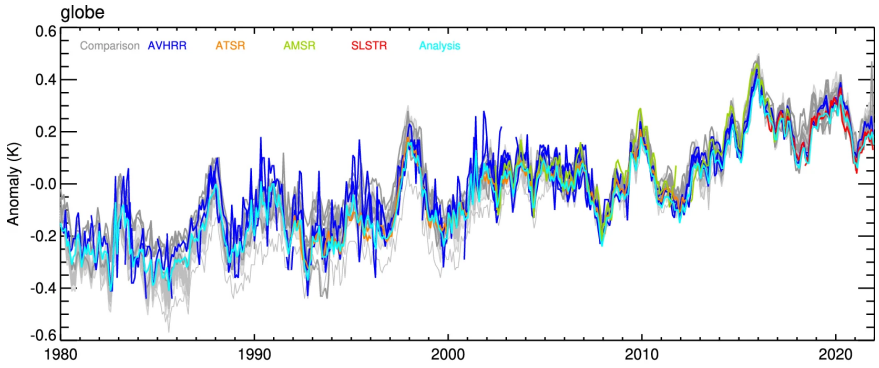


Figure 2.5. Time series of global sea surface temperature anomalies (in Kelvin) from 1980 onwards, showcasing interannual variability and longer-term trends. Multiple datasets (indicated by different colors) are often compared to assess uncertainty and robustness of climate signals. Such data collections are typically managed using array-like structures in programming.

While variables allow us to store individual pieces of data, many scientific and computational tasks require us to work with collections of related items. For instance, we might have a series of temperature measurements taken over time, a list of rock sample names collected from a field site, coordinates (x, y, z) of multiple seismic events, or porosity values measured from different core samples. Storing each such value in a separate, individually named variable (e.g., `temp1`, `temp2`, `temp3`, ...) would be highly impractical, difficult to manage, and inefficient for performing operations on the entire collection. Python, like most programming languages, provides data structures designed to hold and organize multiple items. One of the most versatile and commonly used is the **list**.

Introducing Python Lists

A Python list is an ordered, mutable (i.e., changeable) collection of items. This means that the items in a list are stored in a specific sequence, and you can change, add, or remove items after the list has been created. Lists are incredibly flexible as they can contain items of different data types, including numbers, strings, booleans, and even other lists (creating nested lists).

Creating Lists Lists are created by enclosing a comma-separated sequence of items within square brackets `[]`.

```
# A list of floating-point numbers (temperatures)
temperatures = [15.5, 16.1, 15.8, 17.0]
print(temperatures)

# A list of strings (rock types)
rock_types = ["basalt", "granite", "shale", "sandstone"]
print(rock_types)

# A list containing mixed data types
mixed_data = [10, "andesite", 25.3, True]
print(mixed_data)

# An empty list (can be populated later)
empty_list = []
print(empty_list)

# A list containing another list (nested list)
nested_list = [1, 2, ["a", "b"], 3]
print(nested_list)
```

Listing 2.9. Creating lists in Python.

Accessing List Elements: Indexing

Each item in a Python list has a specific position, known as its **index**. A crucial point to remember is that Python uses **0-based indexing**. This means:

- The first item in the list is at index `0`.
- The second item is at index `1`.
- And so on, up to index `length - 1` for a list of a given `length`.

You can access an individual element in a list by using its index in square brackets after the list's variable name: `listName[index]`.

```
rock_samples = ["granite", "basalt", "shale", "sandstone", "
    limestone"]
print("Original list:", rock_samples)

# Accessing the first element (index 0)
first_sample = rock_samples[0]
print("First sample (index 0):", first_sample)  # Output:
    granite

# Accessing the third element (index 2)
```

```
third_sample = rock_samples[2]
print("Third sample (index 2):", third_sample)  # Output:
    shale

# Attempting to access an element beyond the list's range will
    result in an IndexError
# print(rock_samples[5]) # This would cause an IndexError
```

Listing 2.10. Accessing list elements using positive indices.

Python also supports **negative indexing**, which is a very convenient feature for accessing elements from the end of the list without needing to know the list's length explicitly:

- Index `-1` refers to the *last* item in the list.
- Index `-2` refers to the *second-to-last* item, and so on.

```
elements = ["Oxygen", "Silicon", "Aluminum", "Iron", "Calcium"
    ]
print("Original list of elements:", elements)

# Accessing the last element
last_element = elements[-1]
print("Last element (index -1):", last_element)  # Output:
    Calcium

# Accessing the second-to-last element
second_to_last = elements[-2]
print("Second-to-last element (index -2):", second_to_last)  #
    Output: Iron
```

Listing 2.11. Accessing list elements using negative indices.

Attempting to access an index that is out of bounds (either too large positively or too small negatively, e.g., `elements[-6]` for the list above) will result in an `IndexError`.

Extracting Sub-Lists: Slicing

Often, you will want to work not with a single element, but with a portion of a list, known as a *sub-list* or a *slice*. To do this, Python provides a powerful slicing mechanism. It is crucial to distinguish this from indexing.

- **Indexing** (`listName[index]`) accesses a *single item* from the list and returns that item itself, preserving its original data type.
- **Slicing** (`listName[start:stop]`) extracts a *sequence of items* and always returns a **new list** containing those items. The key syntactic difference is the use of a colon (`:`).

Let's see the difference in practice:

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

# Indexing: returns the element itself (a float)
first_element = measurements[0]
print(f"Indexing with measurements[0]: {first_element}, type:
    {type(first_element)}")
```

```
# Slicing: returns a new list containing the element
first_slice = measurements[0:1]
print(f"Slicing with measurements[0:1]: {first_slice}, type: {
    type(first_slice)}")
```

Listing 2.12. Distinguishing between indexing and slicing.

The basic syntax for slicing is `listName[start:stop]`, which extracts elements starting from the index `start` up to, but *not including*, the index `stop`. The result of a slicing operation is always a new list, even if that list contains only one element or is empty.

- If `start` is omitted, slicing starts from the beginning of the list (index 0).
- If `stop` is omitted, slicing goes up to the very end of the list.

```
measurements = [10.1, 12.5, 11.3, 13.0, 12.8, 10.9, 11.5,
    14.2]
print("Original measurements:", measurements)

# 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]

# Get elements from the beginning up to (not including) index
    3
sub_list2 = measurements[:3]
print("measurements[:3]:", sub_list2)  # Output: [10.1, 12.5,
    11.3]

# Get elements from index 3 to the end of the list
sub_list3 = measurements[3:]
print("measurements[3:]:", sub_list3)  # Output: [13.0, 12.8,
    10.9, 11.5, 14.2]

# Creating a (shallow) copy of the entire list
full_copy = measurements[:]
print("measurements[:]:", full_copy)
print("Is full_copy the same object as measurements?",
    full_copy is measurements) # Output: False
```

Listing 2.13. Basic list slicing.

Slicing can also include a third parameter, `step`, using the syntax `listName[start:stop:step]`. The `step` value determines the increment between indices.

- If `step` is omitted, it defaults to 1.
- A positive `step` selects items moving forward.
- A negative `step` selects items moving backward, which can be used to reverse a list.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("Original numbers:", numbers)

# Get every second element from the beginning to the end
every_other = numbers[::2]  # start and stop are omitted
print("numbers[::2]:", every_other)  # Output: [0, 2, 4, 6, 8]
```

```
# Get elements from index 1 up to (not including) index 8,
    with a step of 3
stepped_slice = numbers[1:8:3]
print("numbers[1:8:3]:", stepped_slice)  # Output: [1, 4, 7]

# A common idiom to reverse a list
reversed_list = numbers[::-1] # start and stop omitted, step
    is -1
print("numbers[::-1]:", reversed_list)  # Output: [9, 8, 7, 6,
     5, 4, 3, 2, 1, 0]
```

Listing 2.14. List slicing with a step.

Mutability: Lists Can Be Changed

An important characteristic of Python lists is that they are **mutable**, meaning their contents can be modified after they are created. You can change individual elements, add new elements, or remove existing ones.

```
rock_types = ["basalt", "granite", "shale"]
print("Initial rock_types:", rock_types)

# Change an element at a specific index
rock_types[1] = "rhyolite" # Replace "granite" with "rhyolite"
print("After changing index 1:", rock_types)

# Add an element to the end of the list using the append()
    method
rock_types.append("marble")
print("After appending 'marble':", rock_types)

# Remove an element by its value using the remove() method
rock_types.remove("shale")
print("After removing 'shale':", rock_types)

# Remove an element by its index using the del statement or
    pop() method
del rock_types[0] # Removes "basalt"
print("After deleting element at index 0:", rock_types)

popped_element = rock_types.pop() # Removes and returns the
    last element ("marble")
print("After popping the last element:", rock_types)
print("Popped element:", popped_element)
```

Listing 2.15. Modifying list elements.

There are many other list methods available for common operations like inserting elements at specific positions (`insert()`), counting occurrences of an element (`count()`), finding the index of an element (`index()`), sorting the list (`sort()`), and reversing the list in place (`reverse()`).

Variables, References, and Copying Lists

When you assign a list (or any mutable object) to a variable, the variable does not store the list itself, but rather a *reference* (or a pointer) to the location in memory where the list object is stored. If you then assign this variable to another variable, both variables will point to the *same list object* in memory.

```
list_a = [10, 20, 30]
list_b = list_a  # list_b now refers to the SAME list object
    as list_a

print("list_a:", list_a)
print("list_b:", list_b)

# Modify the list using list_b
list_b[0] = 99
print("After modifying list_b[0]:")
print("list_a:", list_a)  # Output: [99, 20, 30] - list_a is
    also changed!
print("list_b:", list_b)  # Output: [99, 20, 30]

# Check if they are the same object in memory
print("Are list_a and list_b the same object?", list_a is
    list_b) # Output: True
```

Listing 2.16. List assignment creates references, not copies.

As seen in the example, modifying `list_b` also affected `list_a` because they both refer to the exact same list in memory. This behavior is different for immutable objects like numbers or strings, where assignment typically creates a new copy of the value if the original is then reassigned.

If you need to create an actual independent copy of a list, so that modifications to one do not affect the other, you must do so explicitly. There are several ways to create a **shallow copy**:

- Using slicing: `new_list = old_list[:]`
- Using the `list()` constructor: `new_list = list(old_list)`
- Using the list's `copy()` method: `new_list = old_list.copy()`

```
original_list = [1, 2, 3, [40, 50]] # Contains a nested list
print("Original list:", original_list)

# Method 1: Slicing
copied_list_slice = original_list[:]
# Method 2: list() constructor
copied_list_constructor = list(original_list)
# Method 3: copy() method
copied_list_method = original_list.copy()

# Modify the copied list (e.g., the slice copy)
copied_list_slice[0] = 100
print("After modifying copied_list_slice[0]:")
print("Original list:", original_list)          # Output: [1,
    2, 3, [40, 50]] (unchanged at top level)
print("Copied list (slice):", copied_list_slice)  # Output:
    [100, 2, 3, [40, 50]]
```

```
# Check object identity
print("original_list is copied_list_slice:", original_list is
    copied_list_slice) # False

# Important: For shallow copies, if the list contains other
    mutable objects (like a nested list),
# the nested objects are still references to the same objects
    in both lists.
copied_list_slice[3][0] = 999 # Modify the nested list via the
     copied list
print("After modifying nested list in copied_list_slice:")
print("Original list:", original_list) # Output: [1, 2, 3,
    [999, 50]] - nested list is changed!
print("Copied list (slice):", copied_list_slice)
```

Listing 2.17. Creating shallow copies of lists.

As the example shows, modifying a top-level element of a shallow copy does not affect the original. However, if the list contains mutable elements (like the sub-list `[40, 50]`), the shallow copy contains a reference to that *same* sub-list. Modifying this sub-list through either the original or the shallow copy will affect both.

For situations where you need a completely independent copy of a list and all its nested mutable elements, Python provides **deep copy** functionality through the 'copy' module.

```
import copy # Import the copy module

original_nested_list = [1, [10, 20], 3]
# Create a deep copy
deep_copied_list = copy.deepcopy(original_nested_list)

# Modify the nested list in the deep copy
deep_copied_list[1][0] = 99
print("Original nested list:", original_nested_list) # Output:
     [1, [10, 20], 3]
print("Deep copied list:", deep_copied_list)       # Output:
    [1, [99, 20], 3]

# Check object identity of nested lists
print("original_nested_list[1] is deep_copied_list[1]:", \
      original_nested_list[1] is deep_copied_list[1]) # Output
    : False
```

Listing 2.18. Creating deep copies of lists.

Understanding the distinction between references, shallow copies, and deep copies is crucial for avoiding subtle bugs when working with mutable data structures in Python.

Concept Check

Why is it useful to work with lists or arrays instead of many separate variables?

2.2.5 Repeating Actions: Loops and the range() Function

A fundamental concept in programming is the ability to repeat a block of code multiple times. Python provides `for` loops and `while` loops for this purpose. We will primarily focus

on `for` loops in conjunction with iterating over sequences (like lists) or number ranges.

The for Loop

A `for` loop in Python iterates over the items of any sequence (a list, a string, a tuple, etc.) or other iterable objects, in the order that they appear in the sequence. The basic syntax is:

```
for item_variable in sequence:
    # code block to be executed for each item
    # use item_variable to access the current item
```

```
rock_types = ["basalt", "granite", "shale"]

# Iterate through the list and print each rock type
for rock in rock_types:
    print("Current rock type:", rock)
print("Finished iterating through rock_types.")

# Iterate through a string (which is a sequence of characters)
magma_type = "Rhyolite"
for char in magma_type:
    print(char, end=" ") # Print characters on the same line,
    separated by a space
print("\nFinished iterating through magma_type string.")
```

Listing 2.19. Iterating over a list with a for loop.

Notice the indentation in the example above. The line `print("Current rock type:", rock)` is indented relative to the `for rock in rockTypes:` line, indicating that it is part of the code block that gets executed for each `rock` in the `rockTypes` list. The subsequent line, `print("Finished iterating through rockTypes.")`, is **not indented** to the same level as the `for` loop's body; it is aligned with the `for` statement itself. This signifies that it is the first line of code to be executed *after* the `for` loop has completed all its iterations. This precise use of indentation to define code structure is a fundamental characteristic of Python. Similarly, in the second loop, the `print(char, end=" ")` is part of the loop, while the final `print` statement is executed once the loop over `magmaType` is finished.

The range() Function

Often, we need to execute a loop a specific number of times or iterate over a sequence of numbers. The built-in `range()` function is perfect for this. The `range()` function generates an immutable sequence of numbers. It does not create a list directly in memory (which is efficient for large ranges), but rather an "iterable" object that can be used in a `for` loop.

The `range()` function can be called in a few ways:

- `range(stop)`: Generates numbers from 0 up to (but not including) `stop`.
- `range(start, stop)`: Generates numbers from `start` up to (but not including) `stop`.
- `range(start, stop, step)`: Generates numbers from `start` up to (but not including) `stop`, with an increment (or decrement if negative) of `step`.

```
# Loop 5 times (indices 0, 1, 2, 3, 4)
print("Looping with range(5):")
for i in range(5):
    print(i)

# Loop from 2 up to (not including) 6
print("\nLooping with range(2, 6):")
for num in range(2, 6):
    print(num)

# Loop from 10 down to (not including) 0, with a step of -2
print("\nLooping with range(10, 0, -2):")
for k in range(10, 0, -2):
    print(k)

# Using range to access list elements by index (though direct
    iteration is often more Pythonic)
values = [100, 200, 300, 400]
print("\nAccessing list elements using range(len(list)):")
for index in range(len(values)): # len(values) is 4, so range
    (4) gives 0, 1, 2, 3
    print(f"Element at index {index} is {values[index]}")
```

Listing 2.20. Using the range() function in for loops.

The combination of `for` loops and `range()` is extremely common for performing repetitive tasks, such as iterating through simulation time steps or processing elements in a numerical grid, as will be seen extensively in later chapters.

2.3 Essential Libraries for Scientific Computing in Python

While Python's core language provides fundamental building blocks like lists and loops, its true power for scientific and numerical work comes from a rich ecosystem of specialized libraries. These libraries offer optimized data structures and pre-built functions for common mathematical operations, data analysis, and visualization, saving researchers significant time and effort. In this section, we introduce two of the most foundational libraries for any scientific Python user: NumPy and Matplotlib.

2.3.1 NumPy: Numerical Computing with Arrays

NumPy (short for Numerical Python) is the cornerstone library for numerical computing in Python. It provides:

- A powerful N-dimensional **array object**, called `ndarray`, which is the central data structure.
- Functions for performing efficient operations on these arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, basic linear algebra, basic statistical operations, random simulation, and much more.
- Tools for integrating code from C, C++, and Fortran.

NumPy arrays are significantly more efficient for numerical computations than standard Python lists, especially when dealing with large datasets. Many other scientific libraries in Python, such as SciPy (Scientific Python), Pandas (for data analysis), and Matplotlib (for plotting), are built upon NumPy and use its arrays as their primary data exchange format.

By convention, NumPy is typically imported into a Python script or session using the alias `np`:

```
import numpy as np
```

Listing 2.21. Importing the NumPy library.

This line makes all of NumPy's functions and objects available under the `np.` prefix (e.g., `np.array()`, `np.sin()`).

Python Lists vs. NumPy Arrays: Key Differences

While Python lists are versatile general-purpose containers, NumPy arrays are specifically designed for numerical efficiency. Understanding their differences is key to using them effectively:

1. **Type Homogeneity:**
    - *Python Lists:* Can store elements of different data types within the same list (e.g., `my_list = [1, "rock", 3.14, True]`).
    - *NumPy Arrays:* Typically store elements of the **same data type** (e.g., all integers, or all floating-point numbers). This homogeneity allows NumPy to store and operate on data much more efficiently in memory, as the size and type of each element are known and consistent. If you create an array from a list of mixed types, NumPy will try to upcast them to a common, more general type (e.g., integers and floats might all become floats; numbers and strings might all become strings).
2. **Performance for Numerical Operations:**
    - *Python Lists:* Performing mathematical operations on numerical data stored in lists often requires explicit Python loops. For example, to add two lists element-wise, you would write a `for` loop. These Python-level loops can be slow for large datasets due to the overhead of interpreting each step.
    - *NumPy Arrays:* Support **vectorized operations** (also called element-wise operations). This means you can apply operations and functions directly to entire arrays, or between arrays, without writing explicit loops in Python. These operations are implemented in compiled C or Fortran code "under the hood," making them significantly faster than their list-based loop equivalents.
3. **Memory Usage:**
    - *Python Lists:* Tend to have a higher memory overhead because each element in a list is a separate Python object with its own type information and other metadata.
    - *NumPy Arrays:* Are more memory-efficient for numerical data because they store elements in a contiguous block of memory with minimal overhead, especially for primitive types like integers and floats.

4. **Functionality:**

   - *Python Lists:* Offer basic sequence operations (appending, inserting, removing, sorting, etc.).
   - *NumPy Arrays:* Provide a vast array of mathematical functions (trigonometric, exponential, logarithmic, etc.), linear algebra routines (matrix multiplication, decompositions, solving linear systems), random number generation capabilities, Fourier transforms, and much more, all optimized to operate directly on arrays.

When to Use Which?

- Use **Python lists** for general-purpose collections, especially if you need to store items of mixed data types or require the dynamic flexibility of easily appending or inserting elements of varying types.
- Use **NumPy arrays** whenever you are working with numerical data, particularly for mathematical computations, handling large datasets, and when performance is a critical factor. In scientific computing and numerical modelling, NumPy arrays are almost always the preferred choice for representing and manipulating numerical data.

Creating NumPy Arrays

There are several ways to create NumPy arrays:

From Python Lists or Tuples The most straightforward way to create an array is using the `np.array()` function, passing it a Python list or tuple.

```
import numpy as np

# From a 1D Python list
py_list_1d = [1, 2, 3, 4, 5]
np_array_1d = np.array(py_list_1d)
print("1D NumPy array:", np_array_1d)
print("Type of array:", type(np_array_1d)) # Output: <class '
    numpy.ndarray'>
print("Data type of elements:", np_array_1d.dtype) # Output:
    int64 (depends on system)

# From a 2D Python list (list of lists)
py_list_2d = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
np_array_2d = np.array(py_list_2d)
print("\n2D NumPy array:\n", np_array_2d)
print("Data type of elements:", np_array_2d.dtype) # Output:
    float64
```

Listing 2.22. Creating NumPy arrays from Python lists.

NumPy infers the data type (`dtype`) of the array elements from the input data. You can also explicitly specify the data type using the `dtype` argument in `np.array()`.

Using Built-in NumPy Functions NumPy provides several functions to create arrays with initial placeholder content, which is often more efficient than creating a Python list first, especially for large arrays:

- `np.zeros(shape, dtype=float)`: Creates an array of the given `shape` filled with zeros. `shape` can be an integer for a 1D array (e.g., `np.zeros(5)`) or a tuple for a multi-dimensional array (e.g., `np.zeros((2, 3))` for a 2x3 array).
- `np.ones(shape, dtype=float)`: Creates an array filled with ones.
- `np.full(shape, fill_value, dtype=None)`: Creates an array of the given `shape` filled with `fill_value`.
- `np.arange(start, stop, step, dtype=None)`: Similar to Python's `range()`, but returns a NumPy array. It creates an array with evenly spaced values within a given interval. The 'stop' value is not included.
- `np.linspace(start, stop, num=50, endpoint=True, dtype=None)`: Creates an array with 'num' evenly spaced values over a specified interval [`start`, `stop`]. The `stop` value *is included* by default. This is very useful for creating coordinate arrays for plotting or for defining spatial grids.
- `np.random.rand(d0, d1, ..., dn)` or `np.random.randn(d0, d1, ..., dn)`: Functions for creating arrays filled with random numbers (e.g., from a uniform distribution between 0 and 1 with 'rand', or from a standard normal distribution with 'randn').
- `np.empty(shape, dtype=float)`: Creates an "empty" array of the given shape and dtype, without initializing entries. This can be slightly faster if you plan to fill all elements immediately, but the initial values will be arbitrary (whatever was in that memory location).

```
import numpy as np

# Array of zeros
zeros_array = np.zeros(5)
print("Zeros array:", zeros_array)

# Array of ones, 2x3 shape, integer type
ones_array_int = np.ones((2, 3), dtype=int)
print("\nOnes array (2x3, int):\n", ones_array_int)

# Array using arange (0, 1, 2, 3, 4)
arange_array = np.arange(0, 5, 1)
print("\nArange array:", arange_array)

# Array using linspace (5 points from 0 to 1, inclusive)
linspace_array = np.linspace(0, 1, 5)
print("\nLinspace array:", linspace_array)

# 2x2 array of random numbers (uniform distribution [0,1))
random_array = np.random.rand(2, 2)
print("\nRandom array (2x2):\n", random_array)
```

Listing 2.23. Creating NumPy arrays using built-in functions.

NumPy Array Attributes

NumPy `ndarray` objects have several useful attributes that provide information about the array (without needing to call a function):

- `ndarray.ndim`: The number of dimensions (or axes) of the array. A 1D array has `ndim=1`, a 2D array (matrix) has `ndim=2`, etc.
- `ndarray.shape`: A tuple of integers indicating the size of the array in each dimension. For a 2x3 matrix, `shape` would be `(2, 3)`. For a 1D array of length 5, `shape` would be `(5,)`.
- `ndarray.size`: The total number of elements in the array. This is the product of the elements in `shape`.
- `ndarray.dtype`: An object describing the data type of the elements in the array (e.g., `int64`, `float64`, `bool`).
- `ndarray.itemsize`: The size in bytes of each element of the array.
- `ndarray.nbytes`: The total number of bytes consumed by the elements of the array (`itemsize * size`).

```
import numpy as np

# Create a 2D array (3 rows, 4 columns)
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]], dtype=np.int16)

print("Array arr:\n", arr)
print("Number of dimensions (ndim):", arr.ndim)    # Output: 2
print("Shape of array (shape):", arr.shape)      # Output: (3,
    4)
print("Total number of elements (size):", arr.size)    #
    Output: 12
print("Data type of elements (dtype):", arr.dtype)  # Output:
    int16
print("Size of each element in bytes (itemsize):", arr.
    itemsize) # Output: 2 (for int16)
print("Total bytes consumed by elements (nbytes):", arr.nbytes
    ) # Output: 24
```

Listing 2.24. Accessing NumPy array attributes.

Basic Operations: Vectorization

One of the most powerful features of NumPy is its support for **vectorized operations**. This means that standard arithmetic operations (+, -, *, /, ** for power, etc.) can be applied directly to arrays, and the operation is performed element-wise. This is not only convenient to write but also much faster than using explicit Python loops.

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
```

```
print("Array a:", a)
print("Array b:", b)

# Element-wise addition
sum_ab = a + b
print("a + b =", sum_ab)  # Output: [11 22 33 44]

# Element-wise multiplication
prod_ab = a * b
print("a * b =", prod_ab) # Output: [ 10  40  90 160]

# Scalar multiplication (broadcasts the scalar to all elements
    )
mult_by_scalar = a * 2
print("a * 2 =", mult_by_scalar) # Output: [2 4 6 8]

# Element-wise squaring
squared_a = a**2
print("a ** 2 =", squared_a) # Output: [ 1  4  9 16]

# Universal functions (ufuncs) like np.sin, np.exp also
    operate element-wise
angles = np.array([0, np.pi/2, np.pi])
sines = np.sin(angles)
print("\nsin(angles):", sines) # Output: [0.0000000e+00
    1.0000000e+00 1.2246468e-16] (approx 0, 1, 0)
```

Listing 2.25. Vectorized arithmetic operations with NumPy arrays.

NumPy provides a vast collection of these "universal functions" (ufuncs) that operate element-wise on arrays, covering a wide range of mathematical operations. This vectorization is key to writing efficient and concise numerical code in Python.

A Quick Comparison Summary (Table) To summarize the main differences between Python lists and NumPy arrays, Table 2.1 provides a detailed comparison. This table is presented on a separate, rotated page for better readability due to its width.

2.3.2 Matplotlib: Visualizing Data

Visualizing data and model results is a critical part of any scientific workflow. It helps in understanding trends, identifying errors, and communicating findings. **Matplotlib** is the most widely used and comprehensive library for creating static, animated, and interactive visualizations in Python.

While Matplotlib is a large library with many modules, for most common 2D plotting tasks, we primarily use its `pyplot` module. By strong convention, `pyplot` is imported as `plt`:

```
import matplotlib.pyplot as plt
```

Listing 2.26. Importing the Matplotlib pyplot module.

Table 2.1. Detailed Comparison of Python Lists and NumPy Arrays (‘ndarray‘).

Feature	Python List	NumPy Array (‘ndarray‘)
Data Types	Can store elements of *different* data types within the same list (heterogeneous). For example, ‘[1, "rock", 3.14, True]‘. Each element is a full Python object.	Typically stores elements of the *same* data type (homogeneous), e.g., all integers or all floats. This allows for optimized storage and operations. If mixed types are provided, NumPy will upcast to a common, more general type.
Memory Usage	Generally higher memory overhead per element, as each item is a distinct Python object with associated metadata.	More memory-efficient for numerical data, storing elements compactly in a contiguous block of memory, especially for primitive types.
Performance for Numerical Operations	Operations on numerical data often require explicit Python ‘for‘ loops, which can be slow for large lists due to Python’s interpretation overhead at each step.	Supports highly optimized **vectorized operations** (element-wise operations). These are implemented in compiled C or Fortran and are significantly faster than Python loops for numerical tasks.
Functionality	Offers basic built-in sequence operations like ‘append()‘, ‘insert()‘, ‘remove()‘, ‘sort()‘, indexing, and slicing.	Provides a vast array of mathematical functions (trigonometric, exponential, linear algebra routines, random number generation, Fourier transforms, etc.) that operate efficiently on entire arrays.
Flexibility & Size	Highly flexible; lists can be easily resized by adding or removing elements dynamically.	Generally fixed-size once created. While functions exist to resize or concatenate arrays (e.g., ‘np.resize‘, ‘np.append‘, ‘np.concatenate‘), these operations often create new arrays in memory and copy data, which can be less efficient for frequent modifications.
Primary Use Case	General-purpose collections of items, especially when mixed data types are needed or when the size needs to change frequently and unpredictably.	Numerical computation, scientific computing, data analysis involving large numerical datasets, and any task where performance of mathematical operations on arrays is critical. Forms the basis for many other scientific Python libraries.

Note: The fixed-size nature of NumPy arrays is a key aspect of their performance and memory efficiency for numerical tasks. Operations that appear to "change" the size of an array often result in a new array object being created.

Basic Plotting with Pyplot

The `pyplot` interface provides a MATLAB-like way of plotting. Let's create a simple line plot.

```
import matplotlib.pyplot as plt
import numpy as np # Often used with Matplotlib for data
    generation

# Data for plotting
x_values = np.array([0, 1, 2, 3, 4, 5])
y_values = np.array([0, 1, 4, 9, 16, 25]) # y = x^2

# Create the plot
plt.plot(x_values, y_values)

# Add labels and a title
plt.xlabel("X-axis Label (e.g., Time)")
plt.ylabel("Y-axis Label (e.g., Value)")
plt.title("Simple Plot: y = x^2")

# Add a grid (optional, but often helpful)
plt.grid(True)

# Display the plot
plt.show() # This command renders and displays the plot window
```

Listing 2.27. A basic line plot using Matplotlib pyplot.

Executing this code will generate a window displaying a line plot of $y = x^2$. In a Jupyter Notebook, the plot will typically appear directly below the code cell after execution.

Key Elements of a Matplotlib Plot

A typical Matplotlib plot consists of several key elements, many of which can be customized:

- **Figure and Axes:** A `Figure` is the top-level container for all plot elements. An `Axes` (note: plural, not 'axis') is the region of the figure where data is plotted; a figure can contain one or more Axes (subplots). Most `pyplot` functions operate on the "current" Axes.
- `plt.plot(x, y, ...)`: The primary function for creating line plots. It can take many optional arguments to control line style, color, markers, etc.
- `plt.xlabel("text")`, `plt.ylabel("text")`: Set the labels for the x-axis and y-axis, respectively.
- `plt.title("text")`: Sets the title for the plot (or current Axes).
- `plt.grid(True/False)`: Toggles the display of a grid on the plot.
- `plt.legend()`: Displays a legend if multiple lines have been plotted with labels (e.g., `plt.plot(x, y1, label="Data 1")`).
- `plt.xlim(min, max)`, `plt.ylim(min, max)`: Set the limits for the x and y axes.
- `plt.show()`: Displays all open figures. In scripts, this is essential. In interactive environments like Jupyter, plots might display automatically after the plotting commands, but `plt.show()` can still be used to explicitly render a figure at a certain point.

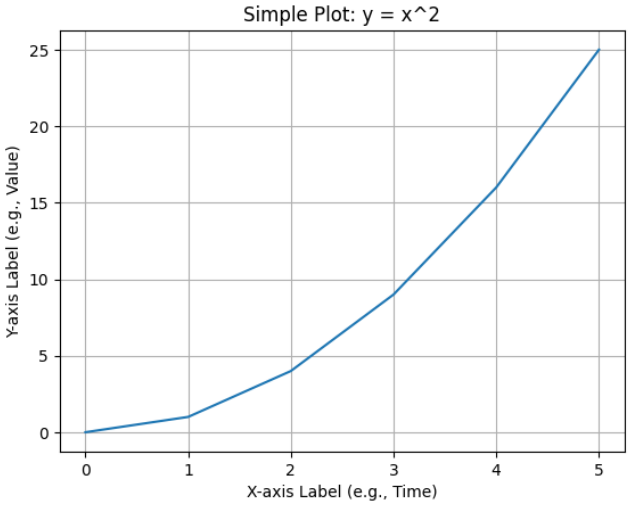


Figure 2.6. Output of a basic Matplotlib plot generated by the Python code in Listing 2.27. The plot displays the relationship $y = x^2$ for a few integer values of x, with axis labels, a title, and a grid.

- Other plot types: `pyplot` also supports scatter plots (`plt.scatter()`), bar charts (`plt.bar()`), histograms (`plt.hist()`), image display (`plt.imshow()`), and more.

We will use Matplotlib extensively throughout this volume to visualize the results of our numerical simulations.

Plotting an Analytical ODE Solution

Let's revisit Newton's Law of Cooling and plot its analytical solution, which we derived earlier: $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})e^{-kt}$.

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

# Parameters for Newton's cooling law
T_env = 20.0  # Ambient temperature (deg C)
T0 = 90.0     # Initial temperature of the body (deg C)
k_coeff = 0.05  # Cooling coefficient (e.g., 1/min)

# Time array for plotting (e.g., from 0 to 60 minutes)
time_array = np.linspace(0, 60, 300) # 300 points for a smooth
    curve

# Calculate the temperature using the analytical solution
Temperature_analytical = T_env + (T0 - T_env) * np.exp(-
    k_coeff * time_array)

# Create the plot
plt.figure(figsize=(8, 5)) # Optional: set figure size
```

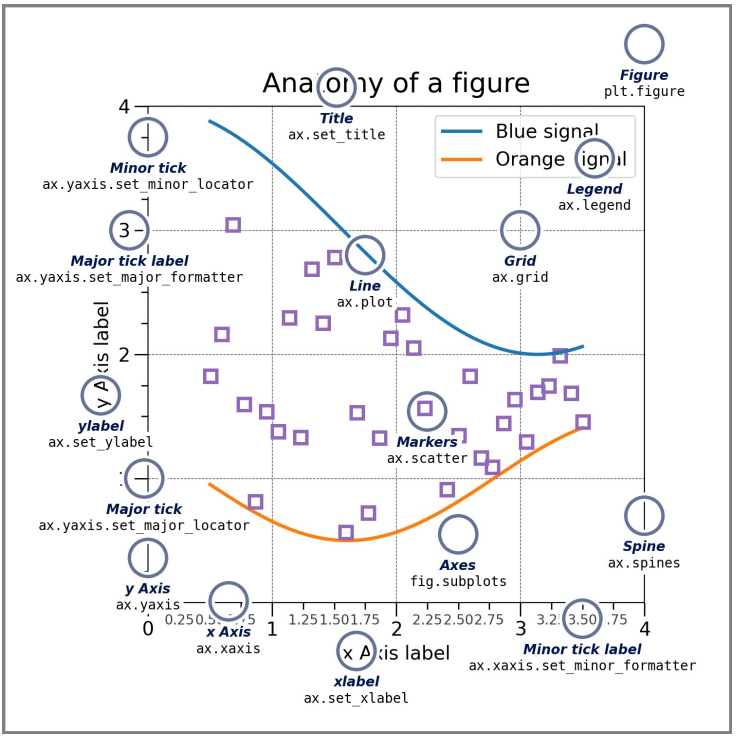


Figure 2.7. Anatomy of a Matplotlib figure, illustrating key components and their corresponding typical `pyplot` commands or object-oriented Axes methods. Understanding these elements, such as the Figure, Axes, Title, Labels (xlabel, ylabel), Ticks (major, minor), Legend, Grid, Lines, and Markers, is essential for creating customized and informative scientific visualizations. (Image based on Matplotlib documentation examples).

```
plt.plot(time_array, Temperature_analytical, label="Analytical
    Solution", color="blue")

# Add labels, title, legend, and grid
plt.xlabel("Time (minutes)")
plt.ylabel("Temperature (deg C)")
plt.title("Newton's Law of Cooling: Analytical Solution")
plt.legend()
plt.grid(True)

# Display the plot
plt.show()
```

Listing 2.28. Plotting the analytical solution of Newton's cooling law.

This example demonstrates how NumPy can be used to generate the data points for the independent variable (time) and to perform the vectorized calculation of the analytical solution, while Matplotlib handles the visualization. This combination is a powerful workflow for scientific computation and analysis in Python.

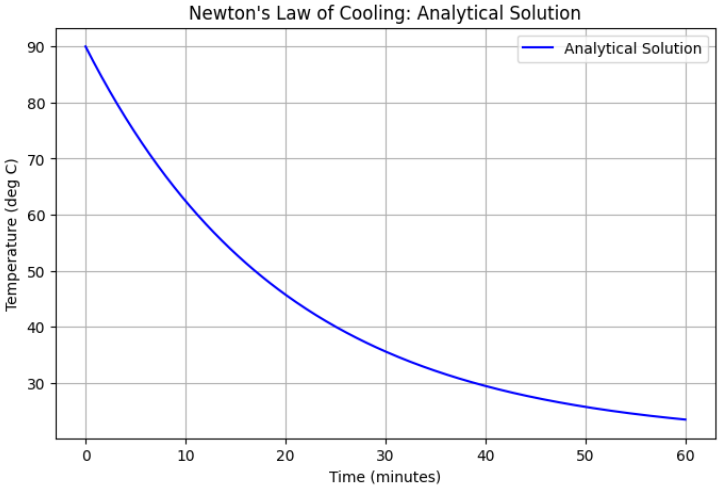


Figure 2.8. Graphical representation of the analytical solution $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})e^{-kt}$ for Newton's Law of Cooling, generated by the Python code in Listing 2.28. The plot shows the exponential decay of an object's temperature from an initial $T_0 = 90°\text{C}$ towards an ambient temperature $T_{\text{env}} = 20°\text{C}$, with a cooling coefficient $k = 0.05\ \text{min}^{-1}$.

2.4 Chapter Summary: Foundations and Tools

This chapter has laid the essential groundwork for our journey into numerical modelling in the geosciences. We began by exploring the fundamental role of **differential equations** as the mathematical language used to describe the dynamic processes that shape our planet.

Key concepts and skills covered include:

- **Motivation for Differential Equations:**
  - Understanding how physical laws governing Earth systems (e.g., conservation principles) translate into differential equations.
  - Illustrated with examples like Newton's Law of Cooling and radioactive decay, leading to first-order Ordinary Differential Equations (ODEs).
  - Introduction to the distinction between ODEs (single independent variable, typically time for our initial examples) and Partial Differential Equations (PDEs, multiple independent variables like space and time), which will be central to later chapters.
  - Acknowledged the limitations of analytical solutions for complex, real-world geoscience problems, thereby motivating the need for numerical methods.
- **Introduction to Python for Scientific Computing:**
  - Justification for using Python due to its readability, extensive scientific libraries, and active community.
  - Introduction to the Jupyter Notebook environment for interactive coding and documentation.

  - Review of Python syntax basics: variables, dynamic typing (contrasted with static typing in languages like C++ or Fortran), the critical role of indentation, line length conventions, line continuation, and commenting.

- **Fundamental Python Data Structures and Control Flow:**
  - Detailed exploration of Python **lists**: creation, 0-based indexing (positive and negative), slicing (with and without step), mutability, and the important distinction between variable references, shallow copies, and deep copies.
  - Introduction to 'for' loops for iteration and the 'range()' function for generating sequences of numbers, essential for repetitive computational tasks.
- **Core Scientific Libraries:**
  - **NumPy:** Introduced as the foundational package for numerical computing, focusing on its powerful N-dimensional array object ('ndarray'). We discussed its advantages over Python lists for numerical work (type homogeneity, performance via vectorization, memory efficiency) and covered basic array creation, attributes, and vectorized arithmetic operations.
  - **Matplotlib:** Presented as the primary library for data visualization, with a focus on the 'pyplot' interface. We covered how to create simple line plots, add essential elements like labels, titles, grids, and legends, and how to display these plots. This was demonstrated by plotting the analytical solution to Newton's Law of Cooling.

By mastering these foundational programming concepts and tools in Python, and by understanding the nature of the differential equations we aim to solve, we are now well-equipped to begin exploring the numerical methods themselves. The next chapter will transition from understanding and visualizing analytical solutions to developing our first numerical schemes for solving ODEs.

Chapter Exercises

Test your understanding of the concepts covered in this chapter with the following exercises.

E1: Classifying Differential Equations

For each of the following differential equations, classify it by stating:

1. Whether it is an Ordinary Differential Equation (ODE) or a Partial Differential Equation (PDE).
2. Its order.
3. Whether it is linear or nonlinear. Justify your answer for linearity/nonlinearity briefly.

(a) $\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + y = \sin(x)$

(b) $\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$ (Burgers' equation, ν is constant)

(c) $\frac{dT}{dt} = -k(T^4 - T_s^4)$ (Stefan-Boltzmann law for radiative cooling, k, T_s are constants)

(d) $\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = F(x, y)$ (Poisson's equation for pressure P, $F(x, y)$ is a known source term)

(e) $L\frac{dI}{dt}+RI = V(t)$ (RL circuit equation for current I, L, R constants, $V(t)$ known voltage)

(f) $\frac{\partial N}{\partial t} = D\frac{\partial^2 N}{\partial x^2} - \lambda N^2$ (Reaction-diffusion equation with a nonlinear reaction term, D, λ constants)

E2: Python List Manipulations and Loops

Consider the following list representing depths (in meters) where rock samples were collected: `depths = [10.5, 22.1, 15.0, 33.5, 8.2, 25.0, 19.7]`

Write Python code snippets to:

1. Print the depth of the second sample collected.
2. Print the last three depths using slicing with negative indices.
3. Create a new list called `shallowSamples` containing only depths less than 20 meters. (Hint: You might use a `for` loop and an `if` statement).
4. Calculate and print the average depth of all samples. (Hint: Sum all elements and divide by the number of elements. You can use a `for` loop to sum).
5. Create a new list where each depth is converted from meters to feet (1 meter ≈ 3.28084 feet). Print the new list. Use a `for` loop.

E3: NumPy Array Operations

Perform the following tasks using NumPy:

1. Create a NumPy array named `porosityValues` from the Python list `[0.12, 0.15, 0.08, 0.21, 0.17]`. Print the array, its shape, and its data type.
2. Create a 1D NumPy array named `xCoordinates` containing 50 equally spaced points from -5.0 to 5.0 (inclusive).
3. Create a 4x2 NumPy array named `matrixB` filled with the constant value 7.5.
4. Given the arrays `vec1 = np.array([0.1, 0.2, 0.3, 0.4])` and `vec2 = np.array([10.0, 20.0, 30.0, 40.0])`:
   - Calculate and print `vec1` multiplied by `vec2` (element-wise).
   - Calculate and print the sine of each element in `vec1` (i.e., `np.sin(vec1)`).
   - Calculate and print `vec2` minus 5 (element-wise subtraction of a scalar).
5. Create a 1D NumPy array of 10 random integers between 1 and 100 (inclusive). (Hint: look into `np.random.randint()`).

E4: Basic Plotting with Matplotlib

1. **Plotting a Geothermal Gradient:** Assume a simplified linear geothermal gradient where temperature T increases with depth z according to $T(z) = T_{surface} + G \cdot z$, where $T_{surface} = 15°C$ and the geothermal gradient $G = 0.025°C/meter$.
   - Create a NumPy array for depths `zValues` from 0 to 2000 meters (e.g., with 100 points).
   - Calculate the corresponding temperatures `Tvalues` using the formula.

- Use Matplotlib to create a line plot of Temperature (y-axis) vs. Depth (x-axis).
- Label your axes ("Depth (m)", "Temperature (°C)").
- Add a title to your plot (e.g., "Simplified Linear Geothermal Gradient").
- Add a grid to the plot.

2. **Multiple Functions on One Plot (using a loop):** You want to plot the functions $y = x^p$ for $p = 1$, $p = 2$, and $p = 3$ on the same graph, for x ranging from 0 to 4.
   - Create a NumPy array `xVals` for the x-axis (e.g., 50 points from 0 to 4).
   - Create a list of powers: `powers = [1, 2, 3]`.
   - Use a `for` loop to iterate through the ‘powers‘ list. Inside the loop:
     - Calculate `yVals = xVals**p` for the current power `p`.
     - Plot `xVals` vs. `yVals`, adding a label for each curve (e.g., ‘f"y $= x^p$"‘).
   - Add a title ("Plot of $y = x^p$"), axis labels ("x", "y"), a legend, and a grid to your figure.

Chapter 3

Numerical Solution of Ordinary Differential Equations: The Euler Methods

In the preceding chapter, we established the fundamental role of differential equations in describing Earth science phenomena and equipped ourselves with essential Python programming tools, including the NumPy library for numerical computations and Matplotlib for visualization. We saw how, for certain simple Ordinary Differential Equations (ODEs) like Newton's Law of Cooling or radioactive decay, exact analytical solutions can be derived and plotted.

However, the reality of geoscience research is that most systems of interest are too complex for such analytical treatment. Nonlinearities, intricate geometries, or an inability to isolate variables often render the quest for an exact formula futile. It is in these prevalent scenarios that **numerical methods** become indispensable, providing a pathway to approximate the solutions to these ODEs.

This chapter marks our first foray into the practicalities of developing and implementing these numerical methods. We will begin with one of the simplest yet most foundational techniques: the **Euler method**. Using the radioactive decay ODE as our test case—an equation for which we know the analytical solution, allowing for direct comparison and error assessment—we will:

- Derive the **Explicit Euler method** by approximating the time derivative with a finite difference.
- Implement this scheme in Python to generate a numerical solution.
- Investigate the concept of **numerical stability** by observing how the solution behaves when the time step, Δt, is varied, and understand the limitations of explicit schemes.
- Introduce an alternative, the **Implicit Euler (or Backward Euler) method**, and discuss its contrasting stability properties and implementation challenges, particularly for nonlinear problems.
- Explore how to quantify the **accuracy** of our numerical solutions by defining and calculating numerical errors (absolute and relative) in comparison to known exact

solutions.

- Discuss the crucial interplay between **accuracy, stability, and computational cost**, which are central considerations in all numerical modelling endeavors.

By the end of this chapter, you will have a practical understanding of how to take a simple ODE, discretize it, implement a numerical solver, and begin to critically evaluate the results – core skills for any numerical modeller. We will start by revisiting the radioactive decay problem, this time from a numerical perspective.

3.1 Numerical Solutions for Ordinary Differential Equations

In the previous chapter, we have seen how ordinary differential equations (ODEs) arise naturally from physical principles to describe processes like cooling or radioactive decay. For simple linear ODEs such as these, we were fortunate enough to be able to find *analytical solutions* – exact mathematical formulas like $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})e^{-kt}$ or $N(t) = N_0 e^{-\lambda t}$ that describe the system's behavior for all times t.

However, as alluded to earlier, many real-world problems in the Earth sciences lead to ODEs (or systems of ODEs) that are far more complex. These complexities can stem from:

- **Nonlinearities:** The rate of change might depend on the unknown function in a nonlinear way (e.g., $dy/dt = y^2 - \sin(y)$).
- **Forcing Terms:** The system might be driven by external influences that vary with time in a complicated manner.
- **Coupled Equations:** Multiple interdependent quantities might evolve simultaneously, described by a system of several ODEs.

In such cases, finding an analytical solution can be exceedingly difficult or outright impossible. Even when an analytical solution technically exists, it might be so convoluted that it offers little practical insight or is cumbersome for computations.

This is where **numerical methods** for solving ODEs become indispensable. Numerical methods do not aim to find an exact formula for the solution. Instead, they provide a way to *approximate* the solution at a series of discrete points. The core idea is to start from a known initial condition and then "march" forward, step by step, calculating an approximate value for the solution at each new point. Our immediate goal is to develop a practical method to approximate the solution of the radioactive decay ODE, which, despite having a known analytical solution, serves as an excellent introductory example for numerical techniques.

3.1.1 Numerical Solution of Radioactive Decay: The Explicit Euler Method

Let's return to the ODE for radioactive decay:

$$\frac{dN}{dt} = -\lambda N(t) \tag{3.1}$$

with an initial condition $N(0) = N_0$. We want to find an approximate numerical solution for $N(t)$ at discrete time points: $t_0 = 0, t_1 = t_0 + \Delta t, t_2 = t_1 + \Delta t, \ldots, t_k = t_0 + k\Delta t$, where

Δt is a chosen small time step. Let N^k denote the numerical approximation of $N(t_k)$. So, $N^0 = N_0$.

The fundamental idea behind many simple numerical methods is to replace the derivative $\frac{dN}{dt}$ with a **finite difference approximation**. Recall the definition of the derivative:

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

If Δt is small (but not zero), we can approximate the derivative at time t_k using the values at t_k and t_{k+1}:

$$\left.\frac{dN}{dt}\right|_{t_k} \approx \frac{N(t_{k+1}) - N(t_k)}{\Delta t} \approx \frac{N^{k+1} - N^k}{\Delta t} \tag{3.2}$$

This is a *forward difference* approximation because it uses the current point N^k and a future point N^{k+1} to approximate the derivative at t_k.

Now, we substitute this approximation into the ODE (Equation 3.1). In the **Explicit Euler method** (also known as the Forward Euler method), the right-hand side of the ODE, $-\lambda N(t)$, is evaluated at the current time t_k using the known value N^k:

$$\frac{N^{k+1} - N^k}{\Delta t} = -\lambda N^k \tag{3.3}$$

Our goal is to find N^{k+1}, the approximate solution at the next time step. We can rearrange Equation 3.3 to solve for N^{k+1}:

$$N^{k+1} - N^k = -\lambda N^k \Delta t \tag{3.4}$$

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

$$N^{k+1} = N^k(1 - \lambda \Delta t) \tag{3.5}$$

This is the **update formula for the Explicit Euler method** applied to the radioactive decay equation.

Summary of the Explicit Euler Algorithm: To solve $dN/dt = -\lambda N$ with $N(0) = N_0$ up to a final time T_{final}:

1. Choose a time step Δt. This determines the number of steps $M = T_{final}/\Delta t$.
2. Initialize: Set $N^0 = N_0$. Set current time $t = 0$.
3. Loop for $k = 0, 1, 2, \ldots, M-1$:
   (a) Calculate the next value N^{k+1} using the update formula: $N^{k+1} = N^k(1 - \lambda \Delta t)$.
   (b) Update time: $t \leftarrow t + \Delta t$.
   (c) Store or process N^{k+1} and t.

This iterative process allows us to generate a sequence of approximate values $N^0, N^1, N^2, \ldots, N^M$ at times $t_0, t_1, t_2, \ldots, t_M$.

Python Implementation of the Explicit Euler Method

Let's implement the Explicit Euler method in Python to solve the radioactive decay problem. We will use NumPy for array operations and Matplotlib for plotting.

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

# Parameters for the decay process and numerical solution
lambda_decay_const = 0.1  # Decay constant (lambda)
N0_initial = 100          # Initial number of atoms
T_final_time = 50         # Total simulation time
dt_time_step = 1.0          # Time step size

# Create an array to store time points
time_points = np.arange(0, T_final_time + dt_time_step,
    dt_time_step)
num_steps = len(time_points)

# Initialize an array to store the numerical solution N at
    each time point
N_euler_solution = np.zeros(num_steps)
N_euler_solution[0] = N0_initial  # Set the initial condition

# Time integration loop using the Explicit Euler formula
for k in range(num_steps - 1): # Loop from 0 to num_steps-2
    N_euler_solution[k+1] = N_euler_solution[k] * (1 -
    lambda_decay_const * dt_time_step)

# Plotting the numerical solution
plt.figure(figsize=(8, 5))
plt.plot(time_points, N_euler_solution, 'o--', label=f'Euler (
    dt={dt_time_step})', color='blue')
plt.xlabel("Time (arbitrary units)")
plt.ylabel("N(t) - Number of Atoms")
plt.title("Radioactive Decay: Explicit Euler Numerical
    Solution")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.1. Python code for solving radioactive decay with Explicit Euler.

Executing this Python script will generate a plot showing the exponential decay of $N(t)$ as approximated by the Explicit Euler method. This is purely a numerical result; we have not yet compared it to the known analytical solution or explored the impact of changing Δt.

Interpreting the Explicit Euler Discretization

It is insightful to consider what the Explicit Euler discretization implies physically or geometrically. The ODE $dN/dt = -\lambda N(t)$ states that the rate of change of N at any time t is proportional to the value of N at that exact time.

The Explicit Euler formula $N^{k+1} = N^k - (\lambda N^k)\Delta t$ can be interpreted as follows:

- We estimate the change in N over the interval $[t_k, t_{k+1}]$ by assuming that the rate of change, dN/dt, remains **constant** throughout this entire interval.

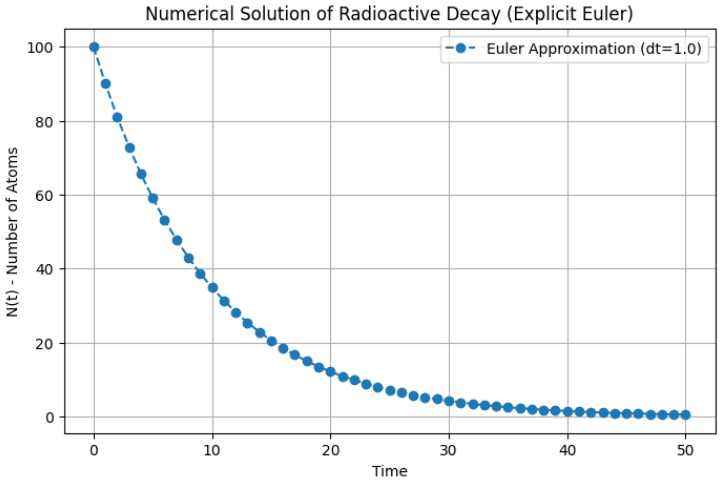


Figure 3.1. Numerical solution for radioactive decay obtained using the Explicit Euler method, as implemented in Listing 3.1. The plot shows the approximated number of atoms $N(t)$ over time. Parameters used: decay constant $\lambda = 0.1$, initial atoms $N_0 = 100$, and time step $\Delta t = 1.0$. The markers indicate the discrete time points at which the solution is calculated. The dashed line connecting the points helps visualize the overall trend, which qualitatively resembles the expected exponential decay.

- Crucially, this constant rate is taken to be the rate calculated at the **beginning of the interval**, i.e., $-\lambda N^k$.

Geometrically, if we think of the solution $N(t)$ as a curve, the Explicit Euler method approximates the value $N(t_{k+1})$ by taking a step of length Δt along the tangent to the solution curve at $N(t_k)$. This is a first-order approximation, as it only uses information from the start of the interval to project forward. As we will see, this simple approach has implications for both the accuracy and stability of the numerical solution. The question arises: is this the only way to approximate the rate over the interval? This leads to other types of numerical schemes, such as implicit methods, which we will explore later.

Concept Check

Recall the analytical solution for radioactive decay, $N(t) = N_0 e^{-\lambda t}$, which was discussed in Chapter 1. This function $N(t)$ is **decreasing** and **convex** (it curves upwards).
The Explicit Euler method approximates the solution at the next time step by taking a step along the tangent to the solution curve at the current time point. Given that the true solution curve $N(t)$ is convex, do you expect the Explicit Euler method to generally **overestimate** or **underestimate** the true solution for $N(t)$ at each step (assuming Δt is small enough for stability but not infinitesimally small)?

3.1.2 Numerical Stability of the Explicit Euler Method

The Explicit Euler method, due to its simplicity, is often a first choice for numerically solving ODEs. However, its ease of implementation comes with a critical caveat: it is not always stable. Let's investigate this by modifying the time step Δt in our Python simulation of radioactive decay.

Experimenting with a Larger Time Step

Consider the same radioactive decay problem ($\lambda = 0.1, N_0 = 100$) previously solved with $\Delta t = 1.0$. What happens if we significantly increase the time step, for example, to $\Delta t = 15.0$?

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

# Parameters (lambda_decay_const and N0_initial from previous
    example)
lambda_decay_const = 0.1
N0_initial = 100
T_final_time = 50
dt_large = 15.0  # Significantly larger time step

time_points_large_dt = np.arange(0, T_final_time + dt_large,
    dt_large)
num_steps_large_dt = len(time_points_large_dt)

N_euler_large_dt = np.zeros(num_steps_large_dt)
N_euler_large_dt[0] = N0_initial

for k in range(num_steps_large_dt - 1):
    N_euler_large_dt[k+1] = N_euler_large_dt[k] * \
                            (1 - lambda_decay_const * dt_large
    )

# Plotting the result
plt.figure(figsize=(8, 5))
plt.plot(time_points_large_dt, N_euler_large_dt, 'o--',
         label=f'Euler (dt={dt_large})', color='orange')
plt.axhline(0, color='black', lw=0.5, linestyle='--') # Zero
    line for reference
plt.xlabel("Time (arbitrary units)")
plt.ylabel("N(t) - Number of Atoms")
plt.title(f"Radioactive Decay: Explicit Euler with Large $\
    Delta t = {dt_large}$")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.2. Explicit Euler for radioactive decay with a large time step, $\Delta t = 15.0$.

Running this code (Figure 3.2) reveals a drastically different and problematic behavior compared to the solution with $\Delta t = 1.0$.

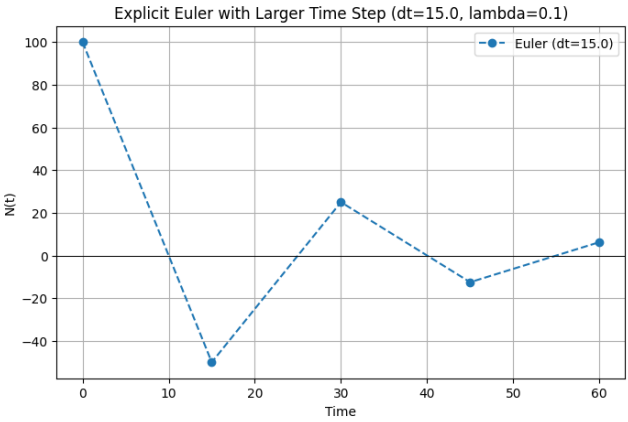


Figure 3.2. Numerical solution of radioactive decay using Explicit Euler with a large time step ($\Delta t = 15.0$, for $\lambda = 0.1$). The solution exhibits unphysical oscillations and negative values, indicating numerical instability.

Understanding Numerical Instability

The output in Figure 3.2 shows that with $\Delta t = 15.0$, the calculated number of atoms $N(t)$ oscillates and takes on negative values. This is clearly unphysical: the amount of a radioactive substance cannot be negative, nor should it oscillate in this manner during simple decay. This behavior is a hallmark of **numerical instability**.

A numerical method is considered **stable** if small errors introduced at one step of the computation (due to truncation or round-off) do not grow in subsequent steps, leading to a bounded and qualitatively correct solution (assuming the true solution is bounded). If these errors are amplified and grow uncontrollably, the method is **unstable**, and the numerical solution becomes meaningless.

For the Explicit Euler method applied to $dN/dt = -\lambda N$, the update formula is $N^{k+1} = N^k(1-\lambda\Delta t)$.

- If $(1-\lambda\Delta t)$ is between 0 and 1 (i.e., $0 < \lambda\Delta t < 1$), then N^{k+1} will be smaller than N^k and positive, correctly representing decay.
- If $(1-\lambda\Delta t)$ is between -1 and 0 (i.e., $1 < \lambda\Delta t < 2$), then N^{k+1} will have the opposite sign of N^k but a smaller magnitude, leading to damped oscillations around zero. While perhaps not explosive, this is still unphysical for decay, as seen in our experiment where $1 - 0.1 \times 15.0 = 1 - 1.5 = -0.5$.
- If $(1-\lambda\Delta t)$ is less than -1 (i.e., $\lambda\Delta t > 2$), then N^{k+1} will have the opposite sign of N^k and a larger magnitude. This leads to growing oscillations and a completely unstable solution.

The condition for the Explicit Euler method to produce a non-negative and non-oscillatory (monotonically decreasing for $N_0 > 0$) solution for this specific decay problem is $1-\lambda\Delta t \geq 0$, which implies:

$$\Delta t \leq \frac{1}{\lambda} \tag{3.6}$$

This type of stability, which depends on the choice of Δt (and other problem parameters like λ), is called **conditional stability**. The Explicit Euler method is conditionally stable for many problems.

This observation leads to a natural question: are there methods that do not suffer from such restrictive stability conditions?

3.1.3 The Implicit Euler (Backward Euler) Method

The Explicit Euler method discretizes $dN/dt = -\lambda N(t)$ by evaluating the right-hand side $(-\lambda N)$ at the current (known) time level n. An alternative approach is to evaluate the right-hand side at the *next* (unknown) time level $n+1$. This leads to an **implicit method**.

Using the same forward difference for the time derivative, $\frac{N^{n+1}-N^n}{\Delta t}$, but evaluating the term $-\lambda N$ at time t_{n+1} (using N^{n+1}), we get:

$$\frac{N^{n+1}-N^n}{\Delta t} = -\lambda N^{n+1} \tag{3.7}$$

This is known as the **Backward Euler** or **Implicit Euler method**. The key difference is that the unknown, N^{n+1}, now appears on both sides of the equation. The term "**implicit**" arises precisely because the value we are trying to find, N^{n+1}, is not given *explicitly* as a direct calculation from known quantities at time n. Instead, N^{n+1} is defined *implicitly* through an equation that involves N^{n+1} itself. To find its value, we must typically perform some algebraic manipulation or, for more complex equations, employ numerical solution techniques to solve for the unknown N^{n+1}. In our simple case, We need to rearrange Eq. 3.7 to solve for N^{n+1}:

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

This yields the update formula for the Implicit Euler method:

$$N^{n+1} = \frac{N^n}{1+\lambda \Delta t} \tag{3.8}$$

Key Features of the Implicit Euler Method for this Problem:

- The unknown N^{n+1} appears on both sides of Equation 3.7 before rearrangement, making it an "implicit" definition. For this simple linear ODE, we could easily solve for N^{n+1} algebraically. For more complex or nonlinear ODEs, solving for N^{n+1} in an implicit scheme might require iterative numerical techniques (like root-finding methods, which we will discuss later in Section 3.2).
- **Stability**: Look at the update term $1/(1+\lambda \Delta t)$. Since $\lambda > 0$ and $\Delta t > 0$, the denominator $(1+\lambda \Delta t)$ is always greater than 1. Thus, $0 < 1/(1+\lambda \Delta t) < 1$. This means N^{n+1} will always be smaller in magnitude than N^n and will retain the same sign. The Implicit Euler method, when applied to this decay equation, is **unconditionally stable** – it will produce a stable, non-oscillatory, and non-negative solution for *any* choice of $\Delta t > 0$.

The unconditional stability of implicit methods is a significant advantage, especially for "stiff" problems (problems involving vastly different time scales) where explicit methods would require prohibitively small time steps. However, this often comes at the cost of increased computational effort per time step if an algebraic system or nonlinear equation needs to be solved.

Python Implementation of the Implicit Euler Method

Implementing the Implicit Euler method for the radioactive decay problem is straightforward due to the simple algebraic solution for N^{n+1}.

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

# Parameters (can reuse from previous examples or define new)
lambda_val_imp = 0.1 # Decay constant
dt_implicit = 15.0   # Time step (can be large, e.g., same as
    unstable explicit case)
T_final_imp = 60     # Total time
N0_imp = 100         # Initial quantity

times_imp = np.arange(0, T_final_imp + dt_implicit,
    dt_implicit)
num_steps_imp = len(times_imp)

N_implicit_solution = np.zeros(num_steps_imp)
N_implicit_solution[0] = N0_imp

# Time integration loop using the Implicit Euler formula
for k in range(num_steps_imp - 1):
    N_implicit_solution[k+1] = N_implicit_solution[k] / \
                               (1 + lambda_val_imp *
    dt_implicit)

# Plotting the implicit solution
plt.figure(figsize=(8, 5))
plt.plot(times_imp, N_implicit_solution, 'o-',
         label=f'Implicit Euler (dt={dt_implicit})', color='
    green')
plt.xlabel("Time (arbitrary units)")
plt.ylabel("N(t) - Number of Atoms")
plt.title(f"Radioactive Decay: Implicit Euler Solution (dt={
    dt_implicit})")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.3. Python code for Implicit Euler solution of radioactive decay.

Running this code (Figure 3.3) with a large Δt (e.g., $\Delta t = 15.0$, which caused instability for the explicit method if $\lambda = 0.1$) will demonstrate the stability of the implicit scheme. The solution will decay smoothly towards zero without oscillations.

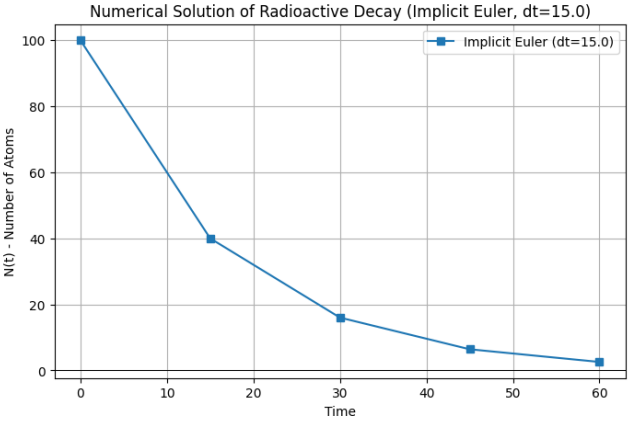


Figure 3.3. Numerical solution of radioactive decay using the Implicit Euler method with a large time step ($\Delta t = 15.0$, for $\lambda = 0.1$). The solution remains stable and decays smoothly, in contrast to the Explicit Euler method with the same large time step.

Concept Check

For the radioactive decay ODE $\frac{dN}{dt} = -\lambda N$:

- The Explicit Euler update is $N^{k+1} = N^k(1 - \lambda\Delta t)$.
- The Implicit Euler update is $N^{k+1} = N^k/(1 + \lambda\Delta t)$.

Explain, conceptually, why the term $(1-\lambda\Delta t)$ in the explicit scheme can lead to unphysical (e.g., negative or oscillating) results if Δt is too large, while the term $1/(1+\lambda\Delta t)$ in the implicit scheme generally prevents this for any positive Δt.

3.1.4 Assessing Numerical Solutions: Accuracy and Error

We have now seen two methods for obtaining approximate numerical solutions to an ODE: the Explicit Euler and the Implicit Euler methods. While the Implicit Euler method demonstrated superior stability for the radioactive decay problem, especially with larger time steps, stability is not the only desirable property of a numerical scheme. We also need to assess its **accuracy** – that is, how close the numerical solution is to the true, exact solution of the differential equation.

For problems where an analytical solution is known (like our radioactive decay example, $N(t) = N_0 e^{-\lambda t}$), we can directly compare our numerical approximation with this exact solution to quantify the error.

Defining Numerical Error

Let $N(t_k)$ be the exact analytical solution at a specific time point t_k, and let N^k_{num} be the numerical solution obtained by our scheme at that same time point.

The **pointwise absolute error** (or simply absolute error) at time t_k is defined as the

absolute difference between the exact and numerical solutions:

$$\varepsilon_{\text{abs}}(t_k) = |N(t_k) - N_{\text{num}}^k| \tag{3.9}$$

Alternatively, one might consider the signed error $\varepsilon(t_k) = N_{\text{num}}^k - N(t_k)$ to see if the numerical method consistently overestimates or underestimates the solution. For a general measure of discrepancy, the absolute error is commonly used.

If the absolute error is small across the simulation domain, our numerical solution is considered a good approximation. If the error is large or grows significantly over time (even for a stable scheme), it might indicate that the time step Δt is too large for the desired accuracy, or that the chosen numerical method has inherent limitations.

The numerical error generally depends on several factors:

- The **time step size** Δt: Smaller time steps usually lead to smaller errors (up to a point where machine precision effects, or round-off errors, might become dominant, though this is less of a concern for the methods and step sizes we are currently considering).
- The **numerical method** itself: Different methods have different intrinsic error characteristics.
- The **regularity (smoothness)** of the true solution: Methods often perform better for smoother solutions.
- The **specific problem** being solved (e.g., the value of λ in the decay equation).

Visualizing Numerical vs. Analytical Solutions

A primary way to assess the quality of a numerical solution is to plot it alongside the exact analytical solution. Let's do this for the Explicit Euler method applied to the radioactive decay problem.

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

# Parameters
lambda_coeff = 0.5  # A slightly larger lambda to see effects
    more quickly
N0_val = 100
t_max_comp = 10.0
dtComp = 0.5      # Time step for comparison
# Students can be encouraged to try dtComp = 0.1, 1.0, 1.9 (
    stable limit for lambda=0.5 is dt=2/0.5=4 for non-osc)

time_vals_comp = np.arange(0, t_max_comp + dtComp, dtComp)
num_steps_comp = len(time_vals_comp)

# Analytical solution
N_analytical = N0_val * np.exp(-lambda_coeff * time_vals_comp)

# Explicit Euler numerical solution
N_explicit_comp = np.zeros(num_steps_comp)
N_explicit_comp[0] = N0_val
for k in range(num_steps_comp - 1):
    N_explicit_comp[k+1] = N_explicit_comp[k] * \
                           (1 - lambda_coeff * dtComp)
```

```
# Plotting both solutions
plt.figure(figsize=(9, 6))
plt.plot(time_vals_comp, N_analytical, 'b-', lw=2, label='
    Analytical Solution')
plt.plot(time_vals_comp, N_explicit_comp, 'ro--', markersize
    =5,
         label=f'Explicit Euler (dt={dtComp})')
plt.xlabel("Time")
plt.ylabel("N(t) - Number of Atoms")
plt.title(f"Radioactive Decay: Analytical vs. Explicit Euler (
    lambda={lambda_coeff})")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.4. Comparing Explicit Euler solution with the analytical solution for radioactive decay.

Executing this code will produce a plot similar to Figure 3.4, where the discrepancy between the numerical and analytical solutions can be visually assessed. By experimenting with different values of `dtComp`, one can observe how the numerical solution converges towards the analytical solution as the time step is reduced (provided the stability condition is met).

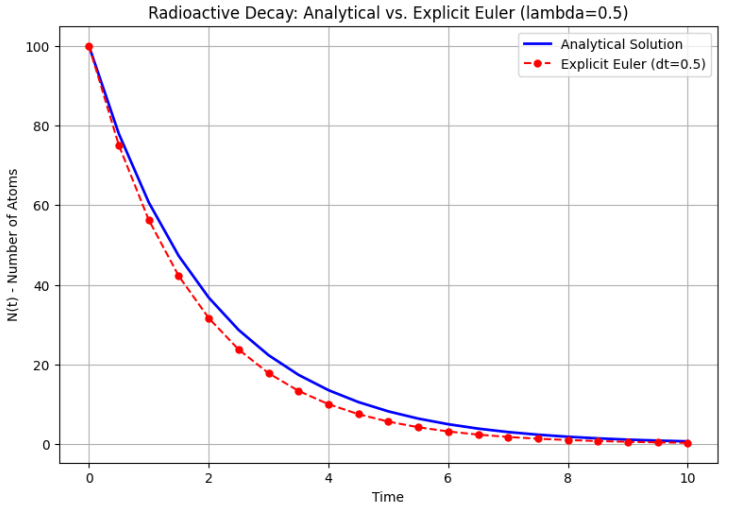


Figure 3.4. Comparison of the analytical solution (blue solid line) and the numerical solution obtained by the Explicit Euler method (red dashed line with markers) for the radioactive decay problem. Parameters: $\lambda = 0.5, N_0 = 100, \Delta t = 0.5$. The numerical solution approximates the true decay but exhibits some deviation.

Plotting the Absolute Error

To quantify the deviation more directly, we can compute and plot the absolute error $\varepsilon_{\text{abs}}(t_k)$ at each time step.

```
# Assuming N_analytical and N_explicit_comp are available from
    the previous code cell
# and were computed on the same time_vals_comp grid.

# Compute absolute error
absolute_error_euler = np.abs(N_analytical - N_explicit_comp)

# Plotting the absolute error
plt.figure(figsize=(9, 6))
plt.plot(time_vals_comp, absolute_error_euler, 'g-o',
    markersize=5,
         label=f'Absolute Error (Euler, dt={dtComp})')
plt.xlabel("Time")
plt.ylabel("Absolute Error |N_exact - N_numerical|")
plt.title(f"Absolute Error of Explicit Euler for Radioactive
    Decay (lambda={lambda_coeff}, dt={dtComp})")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.5. Calculating and plotting the absolute error of the Explicit Euler method.

The resulting plot (an example is shown in Figure 3.5) typically shows how the error evolves over time. For many methods and problems, the error might grow initially and then decrease as the solution itself decays, or it might accumulate over time.

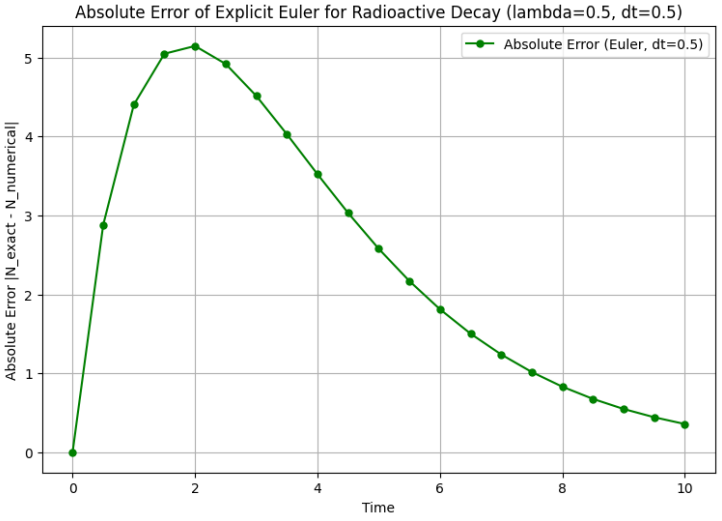


Figure 3.5. Evolution of the absolute error between the analytical solution and the Explicit Euler numerical solution for radioactive decay. Parameters: $\lambda = 0.5, N_0 = 100, \Delta t = 1.0$ (Note: dt might differ from the code generating image6 if this is a generic image). The error behavior can depend on both the method and the problem characteristics.

Relative Error

While absolute error gives a direct measure of the difference, it might not always be the most informative metric, especially when the true solution $N(t)$ varies over a large range of magnitudes or approaches zero. For instance, an absolute error of 0.1 might be insignificant if the true solution value is 100, but quite significant if the true value is 0.2.

The **relative error** normalizes the absolute error by the magnitude of the true solution:

$$\varepsilon_{\text{rel}}(t_k) = \frac{|N(t_k) - N_{\text{num}}^k|}{|N(t_k)|}, \quad \text{for } N(t_k) \neq 0 \tag{3.10}$$

Relative error is often expressed as a percentage if multiplied by 100. It is particularly useful for:

- Assessing error proportionally to the expected value.
- Situations where the quantity of interest becomes very small (e.g., in radioactive decay, as $N(t) \to 0$, the relative error can highlight persistent proportional discrepancies even if the absolute error also becomes small).

When computing relative error, care must be taken to avoid division by zero if the exact solution can be zero at any point.

```
# Assuming N_analytical and N_explicit_comp are available from
    previous cells
# and were computed on the same time_vals_comp grid.

# Compute relative error, avoiding division by zero if
    N_analytical can be zero
# (For exponential decay N_analytical is always > 0 for finite
    N0 > 0)
# A small epsilon can be added to the denominator for general
    robustness.
epsilon = 1e-15 # To prevent division by zero if N_analytical
    is truly zero
relative_error_euler = np.abs(N_analytical - N_explicit_comp)
    / (np.abs(N_analytical) + epsilon)

# Plotting the relative error
plt.figure(figsize=(9, 6))
plt.plot(time_vals_comp, relative_error_euler, 'm-s',
    markersize=5,
         label=f'Relative Error (Euler, dt={dtComp})')
plt.xlabel("Time")
plt.ylabel("Relative Error")
plt.title(f"Relative Error of Explicit Euler for Radioactive
    Decay (lambda={lambda_coeff}, dt={dtComp})")
plt.legend()
plt.grid(True)
# plt.ylim(0, max(0.1, np.max(relative_error_euler)*1.1)) #
    Optional: adjust y-limits
plt.show()
```

Listing 3.6. Calculating and plotting the relative error.

A plot of the relative error (e.g., Figure 3.6) can provide different insights into the performance of the numerical scheme compared to the absolute error plot.

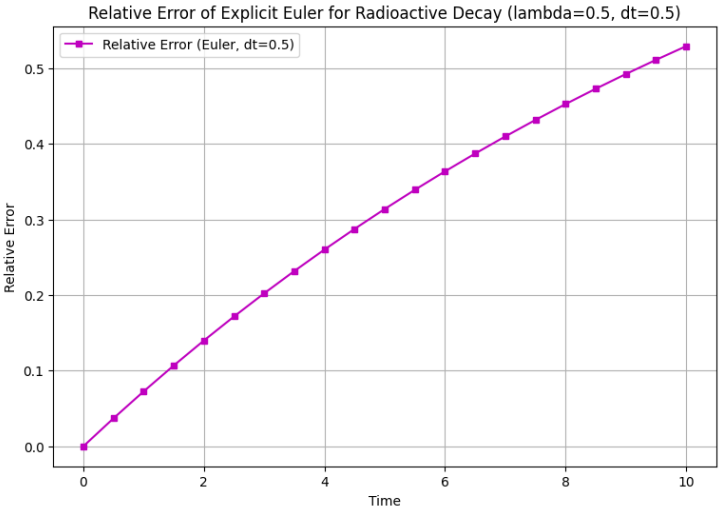


Figure 3.6. Evolution of the relative error for the Explicit Euler method applied to radioactive decay. Parameters may vary from other plots; this illustrates the typical shape. For problems where the true solution approaches zero, the relative error can become large even if the absolute error is small.

Concept Check

Imagine you are modeling the concentration of a rare trace element in a geological process. Its concentration starts very low, increases significantly, and then decays back to very low levels. If you want to assess the accuracy of your numerical solution throughout this process, would absolute error or relative error be more consistently informative? Why?

Numerical Accuracy and Order of a Method

Numerical accuracy refers to how close the numerical solution is to the exact analytical solution. As we have seen, reducing the time step Δt generally improves the accuracy of the Explicit Euler method (provided it remains stable), making the numerical curve approach the exact solution more closely.

The rate at which the error decreases as Δt (or, for PDEs, spatial step Δx) decreases is related to the **order of accuracy** of the numerical method.

- A method is said to be **p-th order accurate** if its *global truncation error* (the accumulated error after many steps, up to a fixed final time T) is proportional to $(\Delta t)^p$.
- The Explicit Euler method has a global truncation error that is $\mathcal{O}(\Delta t)$, meaning it is a **first-order accurate** method in time. This implies that if you halve the time step Δt, you should expect the global error (roughly) to also halve.
- Higher-order methods (e.g., second-order, $\mathcal{O}(\Delta t^2)$) generally achieve a desired level of accuracy with larger time steps (and thus less computational effort) than lower-

order methods, because their error decreases more rapidly as Δt is reduced (e.g., halving Δt would quarter the error for a second-order method).

Improving accuracy by using smaller Δt comes at the cost of more computations (more time steps are needed to reach the same final time). The choice of method and step size often involves a trade-off between desired accuracy, computational resources, and stability considerations. We will delve deeper into the concept of order of accuracy when we analyze spatial discretization schemes.

Concept Check

Why do we often need numerical methods to solve differential equations in Earth sciences, even when we understand the physical laws governing a process? Provide one example of a complexity that might make an analytical solution difficult or impossible to find.

3.2 Challenges with Nonlinear ODEs: Implicit Methods and Root-Finding

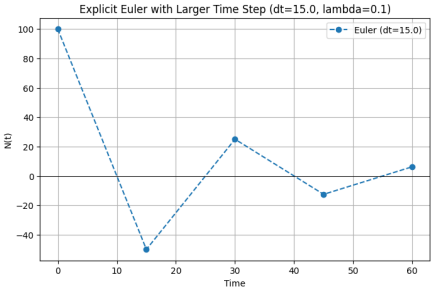
Thus far, our exploration of numerical methods for ODEs has centered on a simple linear equation: radioactive decay. For this problem, both the Explicit Euler and Implicit Euler methods yielded straightforward update formulas. In particular, for the Implicit Euler method, $N^{k+1} = N^k/(1 + \lambda\Delta t)$, we could algebraically isolate the unknown N^{k+1} on one side of the equation. In such linear cases where the implicit formulation can be easily solved algebraically, the Implicit Euler method offers a significant advantage: **unconditional stability**. As we saw, this means we can choose the time step Δt based on accuracy requirements rather than being strictly limited by a stability constraint tied to the problem's parameters (like $\Delta t \leq 2/\lambda$ for the explicit scheme to avoid growing oscillations). With relatively little extra implementational effort for this linear problem, we gain considerable robustness, allowing for potentially larger time steps without the risk of the numerical solution diverging (Figure 3.7 serves as a reminder of this contrast).

However, many important processes in the Earth sciences are described by **nonlinear ODEs**. In such cases, applying an implicit numerical scheme often leads to an algebraic equation where the unknown future value cannot be easily isolated. This necessitates the use of numerical **root-finding methods** to solve for the unknown at each time step.

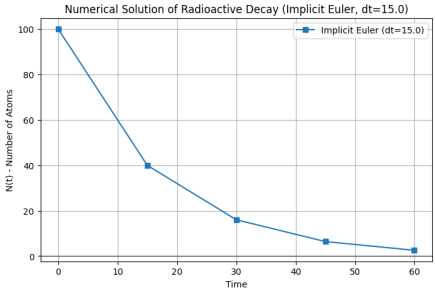
Implicit schemes are often favored for their superior stability properties, allowing for larger time steps than explicit schemes, especially for "stiff" problems. But if the ODE is nonlinear, this stability comes at the cost of needing to solve a (potentially nonlinear) algebraic equation at every iteration. Let's explore this with a relevant geoscience example.

3.2.1 Example: Water Infiltration into Soil – A Nonlinear ODE

A common process in hydrology and soil physics is the infiltration of rainwater into unsaturated soil, or the subsequent drainage of water from a soil layer. While a full description often involves Richards' equation (a nonlinear **PDE**), a simplified, lumped-parameter model can be formulated as an ODE if we consider the spatially averaged volumetric water content $\theta(t)$ in a soil column as it decreases over time due to percolation or drainage.



Explicit Euler (large Δt, potentially unstable)



Implicit Euler (large Δt, stable for decay eq.)

Figure 3.7. Recap from stability discussions: Explicit schemes (left, showing potential instability with large Δt for radioactive decay) are conditionally stable, while implicit schemes (right, showing stability for radioactive decay even with large Δt) can offer unconditional stability, a significant advantage when implementable.

Assume that water is lost from the soil due to gravity-driven flow, and the hydraulic conductivity K is a function of the current water content θ. A simplified ODE can be written as:

$$\frac{d\theta}{dt} = -K(\theta) \tag{3.11}$$

A commonly used empirical expression for the hydraulic conductivity in unsaturated soils is a power-law relationship (e.g., related to Brooks-Corey or van Genuchten models under certain simplifications):

$$K(\theta) = K_s \cdot \left(\frac{\theta - \theta_r}{\theta_s - \theta_r} \right)^m \approx K_s \cdot \tilde{\theta}^m \tag{3.12}$$

where K_s is the saturated hydraulic conductivity, θ_s is the saturated water content (often close to porosity), θ_r is the residual water content, and m is a nonlinearity parameter (typically $m > 1$, e.g., $m = 3$ or 4). For simplicity in our ODE example, let's assume $\theta_r = 0$ and $\theta_s = 1$ (if θ is normalized effective saturation), or simply use θ directly if it represents effective water content, so $K(\theta) = K_s \cdot \theta^m$. Our ODE becomes:

$$\frac{d\theta}{dt} = -K_s \theta^m \tag{3.13}$$

This is a **nonlinear ODE** because the right-hand side depends on θ^m, where the exponent $m \neq 1$.

Explicit Euler Scheme for the Infiltration ODE

Applying the Explicit (Forward) Euler method to Equation 3.13 is straightforward. Let θ^k be the water content at time t_k. The update formula for θ^{k+1} is:

$$\theta^{k+1} = \theta^k - \Delta t \cdot K_s \cdot (\theta^k)^m \tag{3.14}$$

All terms on the right-hand side are known from time level k, so θ^{k+1} can be calculated directly. An example implementation snippet is shown in Listing 3.7.

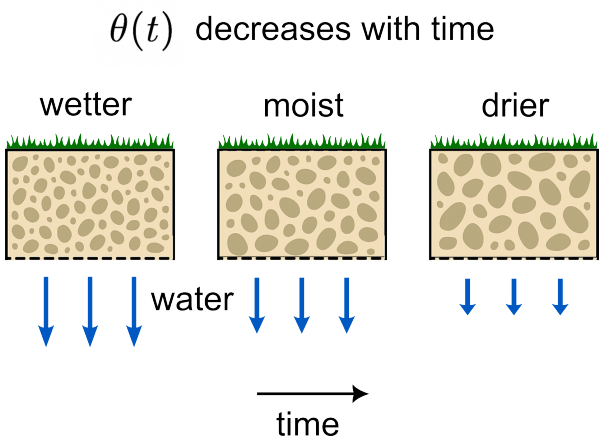


Figure 3.8. A simplified model of gravity-driven drainage in a soil column, a process governed by a nonlinear ODE. The figure shows three snapshots in time: **(Left)** In a 'wetter' state, the high volumetric water content, $\theta(t)$, leads to a high rate of water drainage (many arrows). **(Center)** As water drains, the soil becomes 'moist', and the drainage rate decreases. **(Right)** In a 'drier' state, both the water content and the drainage rate are significantly lower. This visualizes the core principle that the rate of change, $\frac{d\theta}{dt}$, depends on the current value of $\theta(t)$.

```
# Parameters assumed defined: Ks, m_exponent, dt, theta0
# t_array = np.arange(0, Tmax + dt, dt)
# theta_explicit_sol = np.zeros_like(t_array)
# theta_explicit_sol[0] = theta0

# Time integration loop (core part)
for k_step in range(len(t_array) - 1):
    theta_prev = theta_explicit_sol[k_step]
    theta_explicit_sol[k_step+1] = theta_prev - \
                                   dt * Ks * (theta_prev **
    m_exponent)
```

Listing 3.7. Core loop for Explicit Euler applied to the nonlinear infiltration ODE.

Implicit Euler Scheme for the Infiltration ODE: The Challenge

Now, let's apply the Implicit (Backward) Euler method to Equation 3.13. We evaluate the right-hand side at the unknown future time level $k+1$:

$$\frac{\theta^{k+1} - \theta^k}{\Delta t} = -K_s(\theta^{k+1})^m \tag{3.15}$$

Rearranging to try and solve for θ^{k+1}:

$$\theta^{k+1} + \Delta t \cdot K_s \cdot (\theta^{k+1})^m - \theta^k = 0 \tag{3.16}$$

This is a **nonlinear algebraic equation** for the unknown θ^{k+1}. Unlike the linear radioactive decay problem, we cannot simply isolate θ^{k+1} with basic algebra due to the $(\theta^{k+1})^m$ term (assuming $m \neq 1$).

To find θ^{k+1} at each time step, we need to find the root $X = \theta^{k+1}$ of the function:

$$G(X) = X + \Delta t \cdot K_s \cdot X^m - \theta^k \tag{3.17}$$

Finding a "root" of a function $G(X)$ means finding the specific value (or values) of X for which the function evaluates to zero. In our case, this value of X will be our desired θ^{k+1}. This requires a numerical root-finding technique.

3.2.2 Root-Finding: The Bisection Method

One of the simplest and most robust (though not always the fastest) numerical methods for finding the root of a continuous function $G(X) = 0$ is the **bisection method**.

The Core Idea of Bisection

The bisection method relies on the *Intermediate Value Theorem*, which states that if a continuous function $G(X)$ has values of opposite sign at the endpoints of an interval $[a, b]$ (i.e., $G(a) \cdot G(b) < 0$), then there must be at least one root X^* of $G(X)$ within the open interval (a, b).

The bisection method works as follows:

1. **Bracket the Root:** Start with an interval $[a, b]$ such that $G(a)$ and $G(b)$ have opposite signs.
2. **Halve the Interval:** Calculate the midpoint of the interval, $c = (a + b)/2$.
3. **Evaluate $G(c)$:**
    - If $G(c)$ is very close to zero (within a specified tolerance), then c is considered an approximate root.
    - Otherwise, determine which sub-interval, $[a, c]$ or $[c, b]$, now contains the root by checking the sign of $G(c)$ relative to $G(a)$ and $G(b)$.
        - If $G(a) \cdot G(c) < 0$, the root lies in $[a, c]$. The new interval becomes $[a, c]$.
        - Else (meaning $G(c) \cdot G(b) < 0$, assuming $G(c) \neq 0$), the root lies in $[c, b]$. The new interval becomes $[c, b]$.
4. **Repeat:** Repeat step 2 and 3 with the new, smaller interval until the interval becomes sufficiently small or $G(c)$ is sufficiently close to zero.

Each iteration halves the width of the interval bracketing the root, so the method is guaranteed to converge to a root if one is bracketed initially.

Algorithm for the Bisection Method

Given a function $G(X)$, an initial interval $[a_{init}, b_{init}]$ such that $G(a_{init})G(b_{init}) < 0$, a tolerance for the function value ϵ_F, a tolerance for the interval width ϵ_X, and a maximum number of iterations max_iter:

1. Set $a = a_{init}$, $b = b_{init}$.
2. Calculate $G_a = G(a)$.

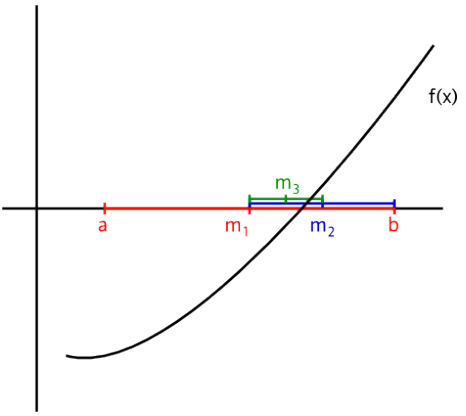


Figure 3.9. The bisection method iteratively refines an interval $[a, b]$ known to contain a root of $f(x)$ (where $f(a)f(b) < 0$). At each step, the interval is halved by choosing the midpoint m_k. The subinterval that retains the sign change is selected for the next iteration ($[a, m_1] \to [m_1, b] \to [m_1, m_2] \to \ldots$). This guarantees convergence to the root where $f(x) = 0$.

3. For $iter = 1$ to max_iter:
   (a) Calculate midpoint: $c = (a + b)/2$.
   (b) Calculate $G_c = G(c)$.
   (c) **Check for convergence:**
      - If $|G_c| < \epsilon_F$ (function value is close to zero), OR
      - If $(b - a)/2 < \epsilon_X$ (interval is small enough),
      - then c is the approximate root. **Return c and stop**.
   (d) **Update interval:**
      - If $G_a \cdot G_c < 0$ (sign change between a and c), then set $b = c$.
      - Else (sign change must be between c and b), set $a = c$ and $G_a = G_c$ (to reuse $G(a)$ in the next iteration).
4. If the loop finishes (max iterations reached without convergence), report failure or return the best estimate c.

The bisection method's convergence is linear, meaning the error is roughly halved at each step. While robust, it can be slower than methods like Newton-Raphson (which exhibits quadratic convergence when close to a root but requires the derivative of $G(X)$ and a good initial guess).

A Simple Bisection Example in Python

To make the bisection method more concrete, let's apply it to find a root of a simple function using Python. Consider the quadratic function:

$$f(x) = x^2 - 4x + 3 \tag{3.18}$$

This function has roots at $x = 1$ and $x = 3$, as it can be factored into $(x-1)(x-3)$. Let's use the bisection method to find the root $x = 1$. We need an initial interval $[a, b]$ such that $f(a)$ and $f(b)$ have opposite signs and bracket this root. For example, we can choose $a = 0$ (where $f(0) = 3$) and $b = 2$ (where $f(2) = 4 - 8 + 3 = -1$). Since $f(0) > 0$ and $f(2) < 0$, a root must exist between 0 and 2.

The Python script below implements the bisection algorithm for this specific function and visualizes the first few iterations.

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

# Define the function whose root we want to find
def example_function(x):
    return x**2 - 4*x + 3

# Bisection method parameters
a_init = -0.5  # Left endpoint of the initial interval
b_init = 1.5  # Right endpoint of the initial interval
tolerance = 1e-5 # Desired tolerance for the root
max_iterations = 20 # Maximum number of iterations

# Check if the initial interval brackets a root
f_a_init = example_function(a_init)
f_b_init = example_function(b_init)

if f_a_init * f_b_init >= 0:
    print("Initial interval does not bracket a root, or a root
     is at an endpoint.")
    if abs(f_a_init) < tolerance: print(f"Root found at a_init
    : {a_init}")
    elif abs(f_b_init) < tolerance: print(f"Root found at
    b_init: {b_init}")
    # exit() # Or handle error appropriately

# Variables for the bisection algorithm
a = a_init
b = b_init
iterations = 0
midpoints_history = [] # To store midpoints for plotting

print(f"Bisection for f(x) = x^2 - 4x + 3, interval [{a:.2f},
    {b:.2f}]")
print("Iter |    a    |    b    |    c    |   f(c)   |
    Interval Width")
print("-" * 60)

for k_iter in range(max_iterations):
    iterations += 1
    c = (a + b) / 2.0
    f_c = example_function(c)
    midpoints_history.append(c)
    interval_width = b - a

    print(f"{iterations:4d} | {a:7.4f} | {b:7.4f} | {c:7.4f} |
     {f_c:8.2e} | {interval_width:14.4e}")
```

```
    if abs(f_c) < tolerance or interval_width / 2.0 <
    tolerance:
        print(f"\nConverged to root: {c:.5f} after {iterations
    } iterations.")
        break

    # Update the interval
    if example_function(a) * f_c < 0: # Root is in [a, c]
        b = c
    else: # Root is in [c, b]
        a = c
else: # Executed if the loop finishes without a 'break'
    print(f"\nMax iterations ({max_iterations}) reached. Best
    estimate: {c:.5f}")

# Plotting the function and the bisection steps
x_plot = np.linspace(min(a_init, b_init) - 0.5, max(a_init,
    b_init) + 0.5, 200)
y_plot = example_function(x_plot)

plt.figure(figsize=(10, 6))
plt.plot(x_plot, y_plot, label='$f(x) = x^2 - 4x + 3$')
plt.axhline(0, color='black', lw=0.5) # x-axis

# Plot initial interval points
plt.plot([a_init, b_init], [example_function(a_init),
    example_function(b_init)],
         'ko', markersize=8, label='Initial Interval [a,b]')

# Plot midpoints from iterations
midpoints_array = np.array(midpoints_history)
plt.plot(midpoints_array, example_function(midpoints_array),
         'r.', markersize=10, label='Midpoints (c_k)')
for i, xc in enumerate(midpoints_array):
    plt.text(xc, example_function(xc) + 0.2, f'$m_{i+1}$',
    color='red', fontsize=9)


plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method Example')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.8. Python implementation of the bisection method to find a root of $f(x) = x^2 - 4x + 3$.

This script first defines the function $f(x)$. It then initializes the interval $[a, b]$ and iteratively applies the bisection logic. The `midpoints_history` list stores each calculated midpoint c for later visualization. The loop continues until the value of $f(c)$ is close enough to zero or the interval width $(b - a)$ becomes smaller than the specified tolerance. Finally, it plots the original function, the initial interval, and the sequence of midpoints calculated, visually demonstrating how the method "homes in" on the root. Figure 3.10 shows an example of such a plot.

This simple example illustrates the core mechanics of the bisection method. We will

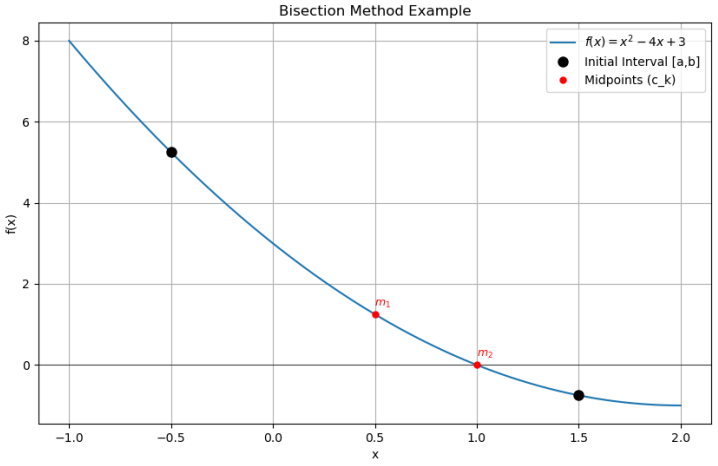


Figure 3.10. Visualization of the bisection method applied to $f(x) = x^2 - 4x + 3$. The blue curve is $f(x)$. The black dots mark the initial interval $[-0.5, 1.5]$. The red dots (labeled $m_1, m_2, \ldots$) show successive midpoints converging towards the root at $x = 1$.

now adapt this logic to solve the nonlinear algebraic equation that arises from the implicit discretization of our soil water infiltration problem.

Concept Check

The bisection method requires an initial interval $[a, b]$ such that $G(a)$ and $G(b)$ have opposite signs to guarantee finding a root of $G(X) = 0$. If you mistakenly choose an initial interval where $G(a)$ and $G(b)$ have the same sign, what would happen when you try to apply the bisection algorithm? Would it still converge? If so, to what? If not, why not?

Implementing the Implicit Infiltration Solver with Bisection

We now return to the problem of solving the nonlinear algebraic equation that arose from applying the Implicit Euler scheme to our simplified soil water infiltration/drainage model. Recall from Equation 3.16 that at each time step k, we need to find the value of θ^{k+1} that satisfies:

$$\theta^{k+1} + \Delta t \cdot K_s \cdot (\theta^{k+1})^m - \theta^k = 0$$

To apply a root-finding method like bisection, we define $X = \theta^{k+1}$ as our unknown and seek the root of the function $G(X)$ where:

$$G(X) = X + \Delta t \cdot K_s \cdot X^m - \theta^k = 0 \tag{3.19}$$

Here, θ^k (the water content from the previous time step) is known, as are Δt, K_s, and m. Our goal is to find the value of X that makes $G(X)$ equal to zero; this X will then be our θ^{k+1}.

To implement the bisection algorithm for this, we first need a Python function that evaluates $G(X)$ for given inputs.

```
def G_infiltration(X_current, theta_prev, dt, Ks, m_exponent):
    """
    Calculates G(X) = X + dt*Ks*X^m - 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.
    m_exponent: Nonlinearity exponent (m in the equation).
    """
    term_nonlinear = dt * Ks * (X_current ** m_exponent)
    return X_current + term_nonlinear - theta_prev
```

Listing 3.9. Python function for $G(X)$ in the infiltration problem.

Now, we can write the main time-stepping loop, incorporating the bisection logic directly to solve for `thetaNext` (which is $X = \theta^{k+1}$) at each step.

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

# (G_infiltration function defined as above)

# Simulation Parameters
Ks_val = 1e-5           # Saturated hydraulic conductivity
m_val = 3               # Nonlinearity exponent
dt_val = 10000.0         # Time step (s) - can be larger for
    implicit
Tmax_val = 500000.0      # Total time
theta0_val = 0.4        # Initial water content

tol_bisect = 1e-7       # Tolerance for bisection
max_iter_bisect = 100   # Max iterations for bisection

# Time array and solution initialization
t_points_impl = np.arange(0, Tmax_val + dt_val, dt_val)
theta_implicit_sol = np.zeros_like(t_points_impl)
theta_implicit_sol[0] = theta0_val

# --- Time integration (Implicit Euler with Bisection Inlined)
    ---
for k_idx in range(len(t_points_impl) - 1):
    theta_k_prev = theta_implicit_sol[k_idx] # This is theta^k

    if theta_k_prev < tol_bisect: # If effectively zero, next
    step is also zero
        theta_implicit_sol[k_idx+1] = 0.0
        continue

    # Bisection method to find X = theta^{k+1}
    # For X + dt*Ks*X^m - theta_k_prev = 0
    # Physical bounds for X = theta^{k+1} are [0, theta_k_prev
    ]
    a_interval = 0.0
    b_interval = theta_k_prev
```

```
    Ga = G_infiltration(a_interval, theta_k_prev, dt_val,
    Ks_val, m_val)
    # Gb = G_infiltration(b_interval, theta_k_prev, dt_val,
    Ks_val, m_val)
    # We expect G(0) = -theta_k_prev <= 0
    # We expect G(theta_k_prev) = dt*Ks*(theta_k_prev)^m >= 0
    # So if theta_k_prev > 0, they should have opposite signs
    or one is zero.

    # Safety check for bracketing (optional, but good practice
    )
    # if Ga * Gb > 0 and not (abs(Ga) < tol_bisect or abs(Gb)
    < tol_bisect):
    #     print(f"Bisection bracketing issue at t={
    t_points_impl[k_idx+1]}")
    #     theta_implicit_sol[k_idx+1] = theta_k_prev # Fallback
    #     continue

    root_found_in_step = False
    for iter_num in range(max_iter_bisect):
        c_midpoint = (a_interval + b_interval) / 2.0

        if (b_interval - a_interval) / 2.0 < tol_bisect: #
    Interval width check
            theta_implicit_sol[k_idx+1] = c_midpoint
            root_found_in_step = True
            break

        Gc = G_infiltration(c_midpoint, theta_k_prev, dt_val,
    Ks_val, m_val)

        if abs(Gc) < tol_bisect: # Function value check
            theta_implicit_sol[k_idx+1] = c_midpoint
            root_found_in_step = True
            break

        if Ga * Gc < 0: # Root is in [a, c]
            b_interval = c_midpoint
            # Gb = Gc # No need to update Gb explicitly if not
     checking Ga*Gb
        else: # Root is in [c, b]
            a_interval = c_midpoint
            Ga = Gc # Update Ga for the new 'a'

    if not root_found_in_step: # If max_iter reached
        # print(f"Bisection max_iter at t={t_points_impl[k_idx
    +1]}, using midpoint.")
        theta_implicit_sol[k_idx+1] = (a_interval + b_interval
    ) / 2.0

# --- Plotting ---
plt.figure(figsize=(9,6))
plt.plot(t_points_impl, theta_implicit_sol, 'o-',
         label=f'Implicit Euler (Bisection, dt={dt_val})')
# (Can also plot explicit solution here for comparison if
    computed)
```

```
plt.xlabel("Time (s)")
plt.ylabel("Volumetric Water Content $\\theta$")
plt.title("Nonlinear Soil Water Infiltration/Drainage (
    Implicit Euler with Bisection)")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3.10. Implicit Euler with inlined bisection for nonlinear infiltration (core logic).

This example illustrates the increased computational effort per time step for implicit methods when dealing with nonlinear ODEs: each step requires an iterative root-finding procedure. However, the potential gain in stability often justifies this extra work, especially if it allows for significantly larger time steps Δt than an explicit method would permit.

Executing the script in Listing 3.10 produces the plot shown in Figure 3.11. The result demonstrates the expected physical behavior: the rate of drainage (the slope of the curve) is steepest at the beginning when the soil is wettest (θ is high) and becomes progressively shallower as the soil dries out. Unlike the linear radioactive decay where the rate was simply proportional to the current value ($dN/dt \propto N$), here the rate is proportional to a power of the current value ($d\theta/dt \propto \theta^m$ with $m \neq 1$). This nonlinear relationship causes the drainage to slow down more significantly as the soil dries compared to a simple exponential decay, and it is this specific behavior that our numerical model has successfully captured.

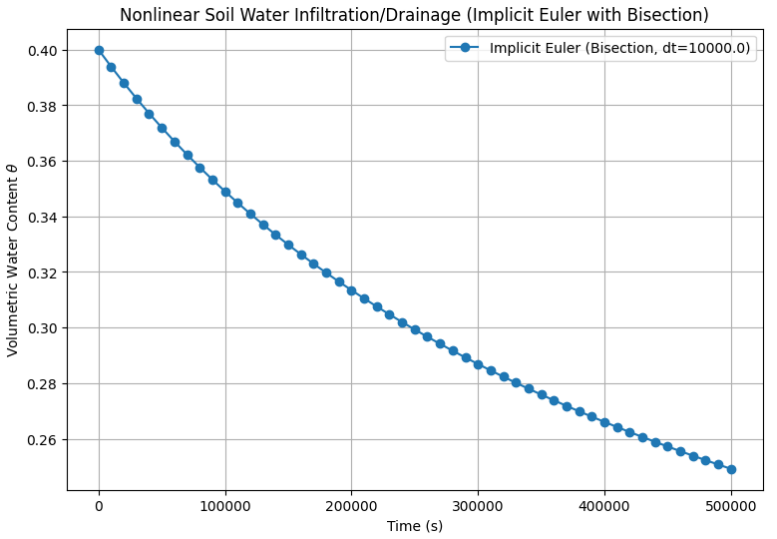


Figure 3.11. Numerical solution for the nonlinear soil water drainage model, obtained using the Implicit Euler method with an embedded bisection root-finder, as implemented in Listing 3.10. The plot shows the decrease in volumetric water content θ over time. The curve's steepness decreases as θ decreases, correctly capturing the nonlinear relationship where hydraulic conductivity, and thus the drainage rate, diminishes as the soil becomes drier.

Chapter Summary: First Steps in Solving ODEs Numerically

This chapter marked our transition from understanding Ordinary Differential Equations (ODEs) analytically to developing and implementing our first numerical methods for their solution. We focused on foundational techniques that form the basis for more advanced approaches in numerical modelling.

The key concepts and methodologies covered in this chapter include:

- **The Need for Numerical Solutions:**
  - Acknowledged that while analytical solutions are ideal, many real-world ODEs (especially nonlinear or complex systems) do not admit them, necessitating numerical approximation.
- **The Explicit Euler Method:**
  - Derived from a forward finite difference approximation of the time derivative, evaluating the ODE's right-hand side at the current time level.
  - Implemented in Python for the radioactive decay problem, $dN/dt = -\lambda N$.
  - Its simplicity in formulation and implementation was highlighted.
- **Numerical Stability:**
  - Introduced the concept of numerical stability, where a scheme is stable if small errors do not grow uncontrollably.
  - Demonstrated that the Explicit Euler method is **conditionally stable**; for the decay problem, stability requires $\Delta t \leq 1/\lambda$ (for non-oscillatory, non-negative results, or $\Delta t \leq 2/\lambda$ to avoid growing oscillations). Using too large a Δt leads to unphysical, unstable solutions.
- **The Implicit Euler (Backward Euler) Method:**
  - Derived by evaluating the ODE's right-hand side at the future (unknown) time level.
  - For the linear decay ODE, this resulted in an algebraic equation for N^{k+1} that could be solved directly: $N^{k+1} = N^k/(1+\lambda\Delta t)$.
  - Demonstrated its **unconditional stability** for the decay problem, producing stable solutions for any $\Delta t > 0$.
- **Assessing Numerical Solutions:**
  - Defined **absolute error** ($|N_{exact} - N_{num}|$) and **relative error** ($|N_{exact} - N_{num}|/|N_{exact}|$) as metrics to quantify the discrepancy between numerical and analytical solutions.
  - Implemented Python scripts to visualize numerical solutions alongside analytical solutions and to plot these error metrics over time.
  - Discussed **numerical accuracy** (how close the numerical solution is to the true solution) and the concept of the **order of a method**. The Euler methods were identified as first-order accurate in time ($\mathcal{O}(\Delta t)$).
- **Nonlinear ODEs and Root-Finding:**

- Highlighted that applying implicit methods to nonlinear ODEs (e.g., a simplified soil water infiltration model, $d\theta/dt = -K_s\theta^m$) results in nonlinear algebraic equations for the unknown future value, which cannot be solved by simple rearrangement.
- Introduced the **bisection method** as a robust numerical technique for finding the root X of an equation $G(X) = 0$.
- Explained the core idea (bracketing a root and iteratively halving the interval) and the algorithm for bisection.
- Demonstrated a simple Python implementation of bisection for a quadratic function, and then showed how to integrate this logic to solve the nonlinear algebraic equation arising from the implicit discretization of the infiltration ODE.

This chapter has equipped us with the fundamental tools to numerically approximate solutions to ODEs using basic explicit and implicit schemes. We have also gained an appreciation for the critical concepts of stability and accuracy, and an initial understanding of the challenges and solution strategies associated with nonlinear problems. These foundational elements will be built upon as we move towards tackling Partial Differential Equations in subsequent chapters. A set of exercises at the end of this chapter provides opportunities to practice these concepts.

Chapter 3 Exercises

Apply your understanding of the concepts and methods discussed in this chapter to solve the following problems.

E3.1: Explicit Euler for Isotope Geochemistry - Sr Evolution in Magma

In a simplified model of magma mixing or crustal assimilation, the concentration of a trace element or an isotopic ratio in a magma body can change over time due to interaction with a contaminant. Consider the evolution of the $^{87}Sr/^{86}Sr$ isotopic ratio (denoted as R) in a magma chamber of mass M_m. The chamber is being contaminated by wall rock with an isotopic ratio R_c at a constant assimilation rate $\dot{m}_a$ (mass per unit time). The magma initially has an isotopic ratio R_0.

If we assume perfect mixing and that the mass of the magma chamber M_m remains approximately constant (e.g., assimilation is balanced by some crystallization, or $\dot{m}_a \ll M_m$ over the timescale of interest), a simplified ODE for the rate of change of R in the magma can be:

$$\frac{dR}{dt} = \frac{\dot{m}_a}{M_m}(R_c - R)$$

Let $k = \frac{\dot{m}_a}{M_m}$ be the effective contamination rate constant. The ODE becomes:

$$\frac{dR}{dt} = k(R_c - R)$$

This equation is linear and describes how R in the magma evolves from R_0 towards R_c.

Tasks:

1. **Analytical Solution (Optional Bonus):** This ODE is similar in form to Newton's Law of Cooling. Can you derive or write down its analytical solution $R(t)$ given $R(0) = R_0$?
2. **Numerical Solution with Explicit Euler:** Given the following parameters:
   - Initial magma $^{87}Sr/^{86}Sr$, $R_0 = 0.704$
   - Contaminant $^{87}Sr/^{86}Sr$, $R_c = 0.710$
   - Effective contamination rate, $k = 5 \times 10^{-4}$ (time units)$^{-1}$ (e.g., if time is in years, this is per year)
   - Total simulation time, $T_{final} = 5000$ time units.

   Write a Python script to solve this ODE for $R(t)$ using the Explicit Euler method.
3. **Time Step Exploration:**
   - Solve the ODE using a "reasonable" time step Δt (e.g., $\Delta t = 100$ time units).
   - Investigate what happens if you use a very large time step, e.g., $\Delta t = 2500$ time units. (Hint: Analyze the stability factor $(1 - k\Delta t)$ or $(1 + k\Delta t)$ depending on how you rearrange for R_c term). The equation can be rewritten as $\frac{dR}{dt} = -k(R - R_c)$.
4. **Plotting:** Plot your numerical solutions for $R(t)$ vs. time for both time steps. If you found the analytical solution, include it on the plot for comparison with the $\Delta t = 100$ case.

(Hint: Refer to the Python implementation of Explicit Euler for radioactive decay. This problem is structurally similar.)

E3.2: Implicit Euler for Sr Isotope Evolution

Continuing with the Sr isotope evolution problem from Exercise E3.1 ($\frac{dR}{dt} = k(R_c - R)$):

Tasks:

1. **Derive Implicit Euler Update Formula:** Starting from $\frac{R^{n+1} - R^n}{\Delta t} = k(R_c - R^{n+1})$, derive the explicit algebraic formula for R^{n+1} in terms of R^n, R_c, k, and Δt.
2. **Python Implementation:** Implement the Implicit Euler method in Python to solve the ODE using the same parameters as in Exercise E3.1 ($R_0 = 0.704, R_c = 0.710, k = 5 \times 10^{-4}, T_{final} = 5000$).
3. **Test with Large Time Step:** Use the same large time step that caused issues for the Explicit Euler method (e.g., $\Delta t = 2500$ time units).
4. **Plotting and Comparison:** On a new plot (or subplot), show:
   - The analytical solution (if you derived it).
   - The Implicit Euler solution with $\Delta t = 100$.
   - The Implicit Euler solution with $\Delta t = 2500$.

   Comment on the stability and accuracy of the Implicit Euler method compared to the Explicit Euler method for this problem, especially with the large time step.

E3.3: Bisection Method - Finding a Bracket and Root

Consider the function $f(x) = x^3 - x - 2$. We want to find a root of this function.

Tasks:

1. **Plot the Function:** Write a short Python script to plot $f(x)$ for x in the range, say, $[-2, 3]$. This will help you visually identify approximate locations of roots.
2. **Find a Bracketing Interval [a,b]:** By observing the plot or by testing a few values, find an interval $[a, b]$ of width at least 0.5 (i.e., $|b - a| \geq 0.5$) such that $f(a)$ and $f(b)$ have opposite signs, thus bracketing a root. State your chosen a and b, and the values of $f(a)$ and $f(b)$. (Hint: Test integer or half-integer values for x.)
3. **Manual Bisection (First 3 Iterations):** Starting with your chosen interval $[a, b]$, manually perform the first three iterations of the bisection method. For each iteration, calculate the midpoint c, $f(c)$, and state the new interval that brackets the root.
4. **Python Implementation for Bracketing (Challenge):** (Optional Challenge) Can you think of a simple strategy to write a Python function that tries to automatically find a bracketing interval $[a, b]$ for a root of a general function 'func(x)', given an initial guess 'xGuess' and an initial step 'stepSize'? The function could, for example, check 'func(xGuess)' and 'func(xGuess + stepSize)', and if they don't have opposite signs, try expanding the interval (e.g., by increasing 'stepSize' or checking further points) until a sign change is found or a maximum search range is exceeded. This is a non-trivial problem for general functions! For this exercise, focus on describing a simple strategy; a full robust implementation is complex.

E3.4: Nonlinear ODE - Simplified Population Logistics with Harvesting

Consider a simplified model for a fish population $P(t)$ in a lake that grows logistically but is also subject to constant rate harvesting H:

$$\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right) - H$$

where:

- r is the intrinsic growth rate.
- K is the carrying capacity of the lake.
- H is the constant harvesting rate.

This is a nonlinear ODE due to the P^2 term (from $P \cdot P/K$) and potentially the P term.

Let $P_0 = 500$ fish, $r = 0.2$ (year)$^{-1}$, $K = 1000$ fish, and $H = 70$ fish/year. We want to find the fish population $P(t)$ over 50 years using the Implicit Euler method and the bisection method to solve the resulting nonlinear algebraic equation at each time step.

Tasks:

1. **Formulate $G(P^{n+1}) = 0$:** Apply the Implicit Euler scheme to the ODE:

$$\frac{P^{n+1} - P^n}{\Delta t} = rP^{n+1}\left(1 - \frac{P^{n+1}}{K}\right) - H$$

Rearrange this to get a nonlinear algebraic equation of the form $G(P^{n+1}) = 0$, where P^{n+1} is the unknown. Write down your function $G(X)$ where $X = P^{n+1}$.

2. **Python Implementation:** Write a Python script to solve this problem from $t = 0$ to $t = 50$ years with a time step $\Delta t = 1$ year.
    - Define a Python function for $G(X)$ (similar to 'G_infiltration' from the chapter text).
    - In your time-stepping loop, for each step:
        - Determine a suitable bracketing interval $[a, b]$ for P^{n+1}. (Hint: The population is unlikely to go much above K or below 0 if it starts positive. Consider $a = 0$. For b, perhaps P^n or K could be starting points for your search, but you need to ensure $G(a)G(b) < 0$. You might need to test a few values or use a small search if $G(K)$ or $G(P^n)$ doesn't have the opposite sign to $G(0)$.)
        - Use an inlined bisection method (similar to the one shown for the infiltration example, Listing 3.10) to find P^{n+1} by finding the root of $G(X) = 0$. Use a tolerance like 10^{-6} and max 100 iterations for bisection.
3. **Plotting:** Plot the fish population $P(t)$ versus time.
4. **Discussion (Conceptual):** What might happen if the harvesting rate H is very high (e.g., $H = 150$ fish/year)? How would this affect your choice of bracketing interval for the bisection method, or the existence of a positive solution? (You don't need to code this part, just discuss).

(Hint: When setting up the bisection for P^{n+1}, P^n is known from the previous step. The equation $G(X) = 0$ will be a quadratic in $X = P^{n+1}$ after rearranging. The bisection method works even if you don't explicitly solve the quadratic, as long as you can evaluate $G(X)$.)

Chapter 4

Approximating Spatial Derivatives: The Finite Difference Method

In the preceding chapters, we focused on Ordinary Differential Equations (ODEs), where the quantities of interest varied with a single independent variable, typically time. We developed numerical methods, such as Euler's explicit and implicit schemes, by approximating the time derivative using values at discrete time steps. However, many, if not most, phenomena in the Earth sciences are not only time-dependent but also vary significantly in space. Understanding the temperature distribution within a cooling magmatic dike, the flow of groundwater through an aquifer, or the propagation of seismic waves requires us to consider these spatial variations. This leads us to the realm of Partial Differential Equations (PDEs).

A crucial step in numerically solving PDEs is the ability to approximate spatial derivatives. Just as we replaced $\frac{dN}{dt}$ with a ratio of differences in N over a time interval Δt, we will now explore how to approximate terms like $\frac{\partial \phi}{\partial x}$ (first spatial derivative) or $\frac{\partial^2 \phi}{\partial x^2}$ (second spatial derivative) using values of ϕ at discrete points along a spatial grid. This chapter introduces the foundational concepts of the **Finite Difference Method (FDM)** for spatial derivatives, focusing on their derivation via Taylor series expansions, their implementation in Python, and the analysis of their numerical accuracy.

4.1 From Time Derivatives to Spatial Derivatives: A Unified Concept

We have already encountered the core idea of finite differences when discretizing the time derivative $\frac{dy}{dt}$ in our ODE solvers. For instance, the Explicit (Forward) Euler method for $\frac{dy}{dt} = f(y,t)$ uses the approximation:

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

Here, $\frac{y^{k+1}-y^k}{\Delta t}$ is a forward difference approximation of $\frac{dy}{dt}$ at time t^k, utilizing values at two distinct time points, t^k and t^{k+1}, separated by the time step Δt.

When dealing with Partial Differential Equations, such as the Heat Equation $\left(\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}\right)$ or the Advection Equation $\left(\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0\right)$, we encounter derivatives with respect to spatial variables (e.g., x, y, or z) in addition to the time derivative. The approach to approximating these spatial derivatives is entirely analogous to how we handled time derivatives:

- We first establish a **spatial grid**, which is a set of discrete points (or nodes) within our spatial domain of interest. For a one-dimensional domain, these points might be $x_0, x_1, x_2, \ldots, x_i, \ldots, x_{N_x-1}$, typically separated by a uniform grid spacing $\Delta x = x_{i+1} - x_i$.
- We then replace the continuous spatial derivatives (e.g., $\frac{\partial \phi}{\partial x}$ at point x_i) with ratios of finite differences, using the values of the function ϕ at these discrete grid points (e.g., $\phi_i = \phi(x_i)$, $\phi_{i+1} = \phi(x_{i+1})$, $\phi_{i-1} = \phi(x_{i-1})$).

This process of replacing continuous derivatives with discrete difference formulas is the essence of the Finite Difference Method.

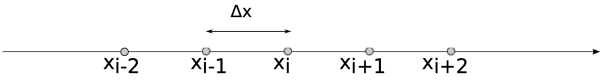


Figure 4.1. A one-dimensional numerical grid with equally spaced nodes x_i. The value of a function $\phi(x)$ is known or sought only at these discrete locations. The spacing between nodes is Δx.

4.2 The Finite Difference Method (FDM) for First Derivatives

Let us consider a function $q(x)$ whose values $q_i = q(x_i)$ are known or sought at the nodes x_i of a uniform 1D grid with spacing Δx. Our goal is to approximate the first derivative, $\left(\frac{dq}{dx}\right)_{x_i}$, at a generic grid point x_i. The Finite Difference Method achieves this by replacing the continuous derivative with an approximation based on the nodal values of the function. The underlying idea comes directly from the definition of a derivative:

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

By not taking the limit and instead using a finite (small) Δx corresponding to our grid spacing, we can derive various finite difference formulas.

Forward Difference Scheme

The **forward difference** approximation for $\frac{dq}{dx}$ at x_i uses the value of the function at the current node x_i and at the next node "downstream" (in the direction of increasing x), $x_{i+1} = x_i + \Delta x$. The formula is:

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_i + \Delta x) - q(x_i)}{\Delta x} = \frac{q_{i+1} - q_i}{\Delta x} \tag{4.1}$$

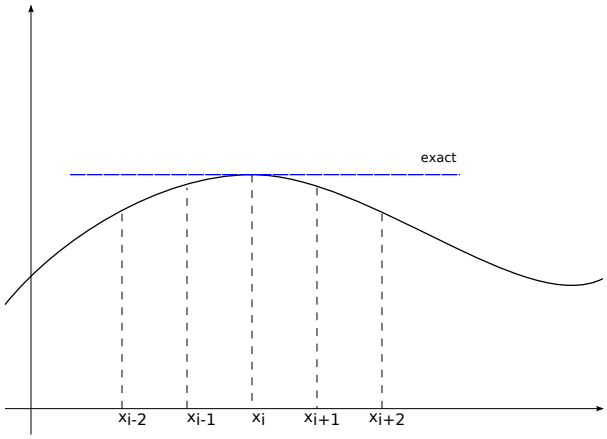


Figure 4.2. Conceptual illustration of approximating the first derivative (slope) of a function $q(x)$ (black curve) at a grid point x_i, which in this specific example coincides with a local maximum where the exact derivative is zero (indicated by the dashed blue line labeled "exact"). Finite difference schemes will use values at x_i and its neighbors (e.g., x_{i-1}, x_{i+1}) to approximate this slope.

Visually, this approximates the slope at x_i using the secant line connecting (x_i, q_i) and (x_{i+1}, q_{i+1}), as depicted in Figure 4.3.

Backward Difference Scheme

The **backward difference** approximation for $\frac{dq}{dx}$ at x_i uses the value at the current node x_i and at the previous node "upstream", $x_{i-1} = x_i - \Delta x$. The formula is:

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_i) - q(x_i - \Delta x)}{\Delta x} = \frac{q_i - q_{i-1}}{\Delta x} \tag{4.2}$$

This approximates the slope at x_i using the secant line connecting (x_{i-1}, q_{i-1}) and (x_i, q_i), as shown in Figure 4.4.

Central Difference Scheme

The **central difference** (or centered difference) approximation for $\frac{dq}{dx}$ at x_i uses values from nodes on both sides of x_i, i.e., x_{i-1} and x_{i+1}. The formula is:

$$\left(\frac{dq}{dx}\right)_{x_i} \approx \frac{q(x_{i+1}) - q(x_{i-1})}{2\Delta x} = \frac{q_{i+1} - q_{i-1}}{2\Delta x} \tag{4.3}$$

This approximates the slope at x_i using the slope of the secant line connecting (x_{i-1}, q_{i-1}) and (x_{i+1}, q_{i+1}). This scheme typically offers higher accuracy for smooth functions, as illustrated in Figure 4.5.

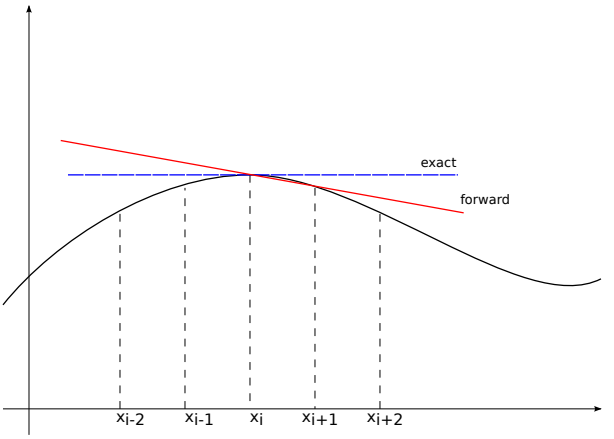


Figure 4.3. Geometric interpretation of the forward difference approximation (red line) for the derivative of the function $q(x)$ (black curve) at the grid point x_i. The scheme uses the values $q(x_i)$ and $q(x_{i+1})$ to calculate the slope of the secant line. In this example, where x_i is at a local maximum, the exact derivative (slope of the tangent) is zero (dashed blue line "exact"), while the forward difference yields a negative slope, underestimating the true derivative at this specific point.

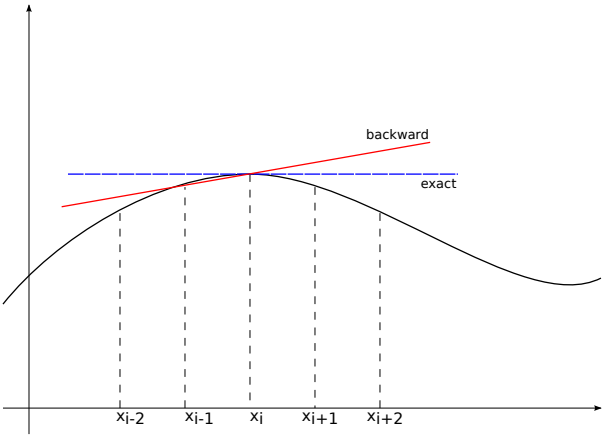


Figure 4.4. Geometric interpretation of the backward difference approximation (red line) for the derivative of the function $q(x)$ (black curve) at the grid point x_i. The scheme utilizes the values $q(x_{i-1})$ and $q(x_i)$ to determine the slope of the secant line. In this example, with x_i at a local maximum, the exact derivative is zero (dashed blue line "exact"). The backward difference yields a positive slope, overestimating the true derivative at this particular point.

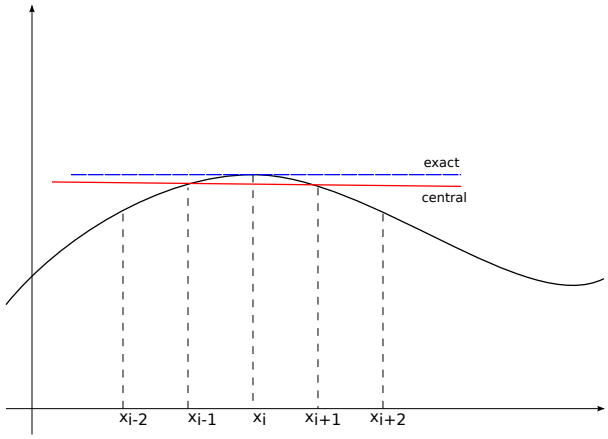


Figure 4.5. Geometric interpretation of the central difference approximation (red line) for the derivative of the function $q(x)$ (black curve) at the grid point x_i. The scheme uses values from both sides of x_i, namely $q(x_{i-1})$ and $q(x_{i+1})$, to calculate the slope. In this example, where x_i is at a local maximum, the exact derivative is zero (dashed blue line "exact"). The central difference approximation, due to the symmetry of the function around x_i in this idealized case, also yields a zero slope, perfectly matching the exact derivative at this specific point. This highlights the higher accuracy often achieved by central differencing for smooth functions.

4.3 Accuracy of Finite Difference Schemes: Truncation Error

The finite difference formulas presented above are approximations to the true derivative. The difference between the true derivative and its finite difference approximation is known as the **truncation error**. This error arises because, in deriving these formulas, we are essentially truncating an infinite Taylor series expansion. The magnitude of the truncation error is directly related to the grid spacing Δx and the first higher-order derivative that was neglected in the Taylor series expansion. Specifically, the error is proportional to the product of this derivative and a power of Δx.

Taylor Series Expansion: The Foundation for Error Analysis

To understand where the truncation error in our finite difference formulas comes from, and to determine their order of accuracy, we need a way to relate the value of a function at one point (e.g., $x_i + \Delta x$) to its value and its derivatives at a nearby point (e.g., x_i). The mathematical tool for this is the **Taylor series expansion**.

The core idea behind a Taylor series is that if a function $q(x)$ is "sufficiently smooth" (meaning it has enough continuous derivatives) around a point x_i, we can approximate the function's value at a nearby point $x_i + h$ (where h is a small displacement, like our Δx) using a polynomial whose terms involve the value of the function $q(x_i)$ and its derivatives ($q'(x_i), q''(x_i)$, etc.) evaluated at x_i. The more terms we include in this polynomial (i.e., the higher the order of derivatives considered), the better the approximation of $q(x_i + h)$ generally becomes, especially for small h.

Figure 4.6 provides a visual intuition. It shows a function (blue curve, in this case, qualitatively similar to e^x) and how it can be progressively better approximated around a point (here, $x_i = 0$) by adding more terms to its Taylor polynomial (red curves):

- A zero-order approximation ($n = 0$ terms in the polynomial beyond the constant $q(x_i)$) is simply a constant value, $q(x_i)$.
- A first-order approximation ($n = 1$, including the term with the first derivative) is a straight line – the tangent to the curve at x_i.
- A second-order approximation ($n = 2$, including the term with the second derivative) is a parabola that matches the function's value, slope, and curvature at x_i.
- As we include terms with higher-order derivatives (e.g., $n = 3$), the polynomial approximation generally hugs the true function more closely over a wider interval around x_i.

Finite difference methods are, in essence, derived by taking these Taylor series expansions and truncating them after a few terms, leading to an approximation and an associated error.

Now, let's write down the formal expressions. Assuming $q(x)$ is sufficiently smooth (possesses enough continuous derivatives as needed for the expansion), the Taylor series expansion for $q(x)$ around a point x_i allows us to express $q(x_i + h)$ and $q(x_i - h)$ as follows:

$$q(x_i + h) = q(x_i) + h\left(\frac{dq}{dx}\right)_{x_i} + \frac{h^2}{2!}\left(\frac{d^2q}{dx^2}\right)_{x_i} + \frac{h^3}{3!}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \cdots + \frac{h^N}{N!}\left(\frac{d^Nq}{dx^N}\right)_{x_i} + \mathscr{R}_{N+1} \tag{4.4}$$

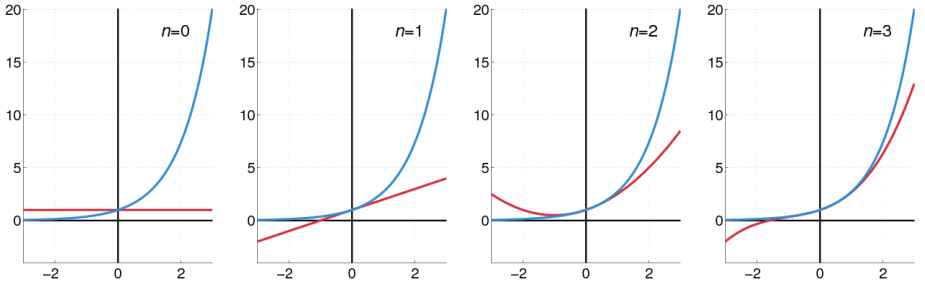


Figure 4.6. Progressive approximation of a function (blue curve, e.g., e^x) around $x_i = 0$ using its Taylor series polynomial (red curve). From left to right: Taylor polynomial including terms up to order $n = 0$ (constant value $q(x_i)$), $n = 1$ (tangent line), $n = 2$ (matching curvature), and $n = 3$. As more terms are added, the polynomial approximation becomes more accurate over a wider range around the expansion point x_i.

$$q(x_i - h) = q(x_i) - h\left(\frac{dq}{dx}\right)_{x_i} + \frac{h^2}{2!}\left(\frac{d^2q}{dx^2}\right)_{x_i} - \frac{h^3}{3!}\left(\frac{d^3q}{dx^3}\right)_{x_i} + \cdots + (-1)^N \frac{h^N}{N!}\left(\frac{d^Nq}{dx^N}\right)_{x_i} + \mathscr{R}'_{N+1} \tag{4.5}$$

Here, h is a small displacement from x_i (in our case, h will typically be our grid spacing Δx). The notation $(\cdot)_{x_i}$ signifies that the derivative is evaluated at the point x_i. The terms $\mathscr{R}_{N+1}$ and $\mathscr{R}'_{N+1}$ are remainder terms, which contain all higher-order terms. For our purposes, we often express these remainder terms using Big O notation, for example, as $\mathscr{O}(h^{N+1})$, indicating that the neglected terms are proportional to h^{N+1} or higher powers of h. For our derivations of finite difference schemes, we will typically expand up to the terms needed to isolate the derivative we want to approximate and identify the leading term of the truncation error.

Concept Check

Consider the Taylor series expansion for $q(x_i + h)$ around the point x_i (as shown in Equation 4.4). This series expresses $q(x_i + h)$ as a polynomial in the variable h, where the coefficients of the powers of h involve the derivatives of $q(x)$ evaluated at x_i.

1. If you were to compute the derivative of the polynomial with respect to h (treating x_i and derivatives of q at x_i as constants with respect to h), what would be the first term of the resulting new polynomial (i.e., the term not multiplied by h)?
2. If you then evaluate this new polynomial at $h = 0$, what is the result?
3. Can you generalize this? What would you obtain if you differentiate the Taylor series for $q(x_i + h)$ k times with respect to h, and then set $h = 0$?

(Thinking through this helps to see how the derivatives of $q(x)$ at x_i are fundamentally encoded within its Taylor expansion.)

Truncation Error of the Forward Difference Scheme

Using Equation 4.4 with $h = \Delta x$, and denoting $q(x_i + \Delta x)$ with q_{i+1}, we have:

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

Rearranging to match the forward difference formula (Equation 4.1):

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

This equation is very insightful. It shows that the **forward difference approximation** (the term on the left-hand side) is equal to the **exact value of the first derivative** at x_i (the first term on the right-hand side) *plus* a series of additional terms. These additional terms, grouped by the curly bracket, represent the **truncation error** of the forward difference scheme.

This truncation error is precisely the discrepancy, or the error we commit, when we use the simple forward difference formula $\frac{q_{i+1}-q_i}{\Delta x}$ as an approximation for the true derivative $\left(\frac{dq}{dx}\right)_{x_i}$. It arises because our finite difference formula is derived from a Taylor series that has been *truncated* by neglecting these higher-order terms. The magnitude of this truncation error depends on the step size Δx and the higher-order derivatives of the function $q(x)$ at the point x_i.

The **leading term of the truncation error** (the term in the series with the lowest power of Δx that does not cancel out) is $\frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i}$. Since this term is proportional to Δx^1, the forward difference scheme is said to be **first-order accurate**, denoted as $\mathcal{O}(\Delta x)$. This means that if we halve Δx, the truncation error should, in principle, also roughly halve (assuming Δx is small enough that the leading term dominates the error).

Truncation Error of the Backward Difference Scheme

Using Equation 4.5 with $h = \Delta x$, we have $q(x_i - \Delta x) = q_{i-1}$:

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

Rearranging to match the backward difference formula (Equation 4.2):

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

The leading term of the truncation error is $-\frac{\Delta x}{2}\left(\frac{d^2q}{dx^2}\right)_{x_i}$. This scheme is also **first-order accurate**, $\mathcal{O}(\Delta x)$.

Truncation Error of the Central Difference Scheme

To find the error for the central difference scheme, we subtract Equation 4.5 (for q_{i-1}) from Equation 4.4 (for q_{i+1}):

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

Many terms cancel out:

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

(Note: The $\mathcal{O}(\Delta x^4)$ terms actually cancel due to symmetry in this case, leaving $\mathcal{O}(\Delta x^5)$ as the next higher-order error term from the original expansion). Rearranging to match the central difference formula (Equation 4.3):

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

The leading term of the truncation error is $\frac{\Delta x^2}{6} \left(\frac{d^3q}{dx^3} \right)_{x_i}$. Since this term is proportional to Δx^2, the central difference scheme is **second-order accurate**, $\mathcal{O}(\Delta x^2)$. This implies that if Δx is halved, the error should decrease by a factor of approximately four, making it generally more accurate (i.e. with a smaller error) than first-order schemes for sufficiently smooth functions and small Δx.

Summary of Truncation Error and Order of Accuracy

- The terms neglected in the Taylor series when deriving a finite difference formula constitute the **truncation error**.
- This error measures the *local* discrepancy between the true derivative and its numerical approximation at a point.
- The **order of accuracy** of a scheme is determined by the lowest power of Δx (or Δt for time derivatives) in the leading term of the truncation error.
- Higher-order schemes generally converge to the true solution more rapidly as the grid spacing is refined, but they may involve more complex formulas or wider stencils (using more grid points).

4.4 Practical Examples and Numerical Verification of Order

Theoretical analysis of truncation error provides insight into the expected accuracy of finite difference schemes. We can also numerically verify these orders of accuracy by applying the schemes to functions with known analytical derivatives and observing how the actual error changes as we vary the grid spacing Δx.

Printing Formatted Output in Python: f-strings

Before diving into the Python examples for derivative approximation, let's revisit and expand on how to print formatted output in Python, as this will be useful for displaying results clearly. While basic `print()` works, Python (version 3.6+) offers a convenient and powerful way to format strings called **f-strings** (formatted string literals).

F-strings allow you to embed expressions, including variable values, directly inside string literals. You create an f-string by prefixing the string with an `f` or `F` and placing expressions inside curly braces `{ }`.

```
rockType = "Granite"
sampleNumber = 12
porosity = 0.1537

# Embedding variables directly in the string
print(f"Sample {sampleNumber} is {rockType} with porosity {
    porosity}.")
# Output: Sample 12 is Granite with porosity 0.1537.
```

Listing 4.1. Basic f-string usage in Python.

More importantly for numerical results, f-strings allow for precise control over the formatting of numbers:

- `{variableName:.Nf}`: Displays a floating-point number with `N` digits after the decimal point (e.g., `:.2f` for two decimal places).
- `{variableName:.Ne}` : Displays a floating-point number in scientific (exponential) notation with `N` digits after the decimal point (e.g., `:.4e` for four decimal places in scientific notation).

For example:

```
dxValue = 0.1
fPrimeApprox = 1.90001234
errorValue = 0.09998765

print(f"For dx = {dxValue:.2f}: Approximation = {fPrimeApprox
    :.4f}, Error = {errorValue:.3e}")
# Expected Output:
# For dx = 0.10: Approximation = 1.9000, Error = 9.999e-02
```

Listing 4.2. Formatting numbers with f-strings.

We will use f-strings to present our numerical results in a readable format.

Example 1: Approximating the Derivative of $f(x) = x^2$

Let's start with a simple polynomial function, $f(x) = x^2$. The analytical (exact) derivative of this function is $f'(x) = 2x$. We will choose to approximate the derivative at a specific point, $x_0 = 1.0$. At this point, the exact derivative is $f'(1.0) = 2 \times 1.0 = 2.0$.

We will use a step size Δx (denoted `dxValue` in the code) to calculate the function values needed for our finite difference schemes:

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

The Python code in Listing 4.3 defines the function and its derivative, then calculates these approximations and their absolute errors for a chosen Δx.

```
import numpy as np
# import matplotlib.pyplot as plt # Not strictly needed for
    this specific listing

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

def derivativeXSquared(x):
    return 2*x

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

# Exact derivative value at x0
fPrimeExact = derivativeXSquared(x0)
print(f"Function: f(x) = x^2")
print(f"Exact derivative f'(x) = 2x")
print(f"Exact derivative at x = {x0}: {fPrimeExact:.4f}\n")

# Calculate function values needed for differences
fAtX0 = functionXSquared(x0)
fAtX0PlusDx = functionXSquared(x0 + dxValue)
fAtX0MinusDx = functionXSquared(x0 - dxValue)

# Forward Difference Approximation
fPrimeForward = (fAtX0PlusDx - fAtX0) / dxValue
errorForward = abs(fPrimeForward - fPrimeExact)
print(f"Forward Difference (dx = {dxValue:.2f}):")
print(f"  Approximation = {fPrimeForward:.4f}, Absolute Error
    = {errorForward:.4e}")

# Backward Difference Approximation
fPrimeBackward = (fAtX0 - fAtX0MinusDx) / dxValue
errorBackward = abs(fPrimeBackward - fPrimeExact)
print(f"\nBackward Difference (dx = {dxValue:.2f}):")
print(f"  Approximation = {fPrimeBackward:.4f}, Absolute Error
     = {errorBackward:.4e}")

# Central Difference Approximation
fPrimeCentral = (fAtX0PlusDx - fAtX0MinusDx) / (2 * dxValue)
errorCentral = abs(fPrimeCentral - fPrimeExact)
print(f"\nCentral Difference (dx = {dxValue:.2f}):")
print(f"  Approximation = {fPrimeCentral:.4f}, Absolute Error
    = {errorCentral:.4e}")
```

Listing 4.3. Approximating the derivative of $f(x) = x^2$ at $x_0 = 1.0$ using finite differences.

Running this script produces the output summarized in Table 4.1. The results highlight several key points:

- Both the forward and backward difference schemes produce an error of the same magnitude but with opposite signs (the forward difference overestimates the derivative,

while the backward difference underestimates it).

- The central difference scheme, for this specific quadratic function, produces a result that is remarkably accurate. Its error is extremely small, on the order of machine precision.

The exceptional accuracy of the central difference for $f(x) = x^2$ is not a coincidence. As discussed in Section 4.3, the leading term of the truncation error for this method is proportional to the third derivative of the function, $f'''(x)$. For $f(x) = x^2$, we have $f'(x) = 2x$, $f''(x) = 2$, and $f'''(x) = 0$. Since the third derivative is identically zero, the leading error term vanishes entirely. The remaining error is due to higher-order terms in the Taylor expansion (which are also zero for a quadratic) and the finite precision of floating-point arithmetic in the computer. This special case perfectly illustrates why central differencing is a higher-order, and generally more accurate, method.

Table 4.1. Finite difference approximations and absolute errors for the derivative of $f(x) = x^2$ at $x_0 = 1.0$ (exact derivative is 2.0), using a step size of $\Delta x = 0.1$.

Difference Scheme	Approximation	Absolute Error
Forward Difference	2.1000	1.00×10^{-1}
Backward Difference	1.9000	1.00×10^{-1}
Central Difference	2.0000	≈ 0 (e.g., $< 10^{-15}$)

Example 2: Approximating the Derivative of $f(x) = \sin(x)$

Now, let's consider a transcendental function, $f(x) = \sin(x)$. Its analytical derivative is $f'(x) = \cos(x)$. This relationship is visualized in Figure 4.7. Understanding the behavior of the derivative (slope) of $\sin(x)$ can help in interpreting the results of our numerical approximations. For instance, where $\sin(x)$ has a peak or a trough, its derivative $\cos(x)$ is zero. Where $\sin(x)$ is increasing most rapidly, $\cos(x)$ is at its positive peak, and where $\sin(x)$ is decreasing most rapidly, $\cos(x)$ is at its negative peak (trough).

We will approximate the derivative of $f(x) = \sin(x)$ at $x_0 = \pi/4$. The exact derivative at this point is $f'(\pi/4) = \cos(\pi/4) = \sqrt{2}/2 \approx 0.70710678$. We will again use a step size Δx (e.g., $\Delta x = 0.1$) and apply the same three finite difference schemes.

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

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

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

# Point at which to evaluate the derivative
x0Sin = np.pi / 4.0
# Step size
dxSin = 0.1

# Exact derivative value at x0Sin
```

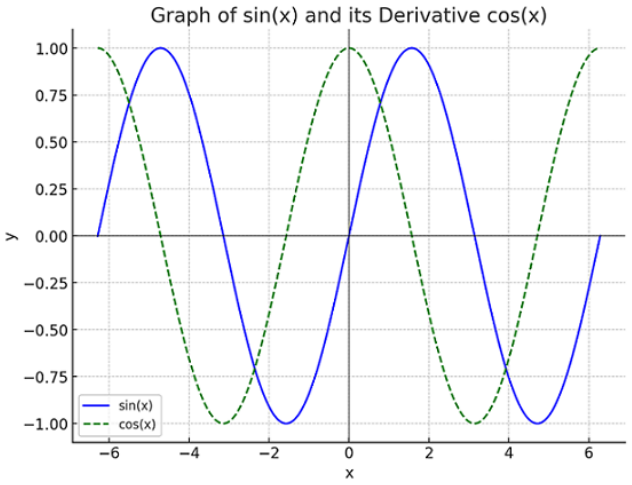


Figure 4.7. The function $f(x) = \sin(x)$ (solid blue line) and its analytical derivative $f'(x) = \cos(x)$ (dashed green line). Note how the value of $\cos(x)$ corresponds to the slope of $\sin(x)$ at each point x.

```
fPrimeExactSin = derivativeSinX(x0Sin)
print(f"\nFunction: f(x) = sin(x)")
print(f"Exact derivative f'(x) = cos(x)")
print(f"Exact derivative at x = {x0Sin:.4f} (pi/4): {
    fPrimeExactSin:.6f}\n")

# Calculate function values
fAtX0Sin = functionSinX(x0Sin)
fAtX0PlusDxSin = functionSinX(x0Sin + dxSin)
fAtX0MinusDxSin = functionSinX(x0Sin - dxSin)

# Forward Difference
fPrimeForwardSin = (fAtX0PlusDxSin - fAtX0Sin) / dxSin
errorForwardSin = abs(fPrimeForwardSin - fPrimeExactSin)
print(f"Forward Difference (dx = {dxSin:.2f}):")
print(f"  Approximation = {fPrimeForwardSin:.6f}, Absolute
    Error = {errorForwardSin:.4e}")

# Backward Difference
fPrimeBackwardSin = (fAtX0Sin - fAtX0MinusDxSin) / dxSin
errorBackwardSin = abs(fPrimeBackwardSin - fPrimeExactSin)
print(f"\nBackward Difference (dx = {dxSin:.2f}):")
print(f"  Approximation = {fPrimeBackwardSin:.6f}, Absolute
    Error = {errorBackwardSin:.4e}")

# Central Difference
fPrimeCentralSin = (fAtX0PlusDxSin - fAtX0MinusDxSin) / (2 *
    dxSin)
errorCentralSin = abs(fPrimeCentralSin - fPrimeExactSin)
print(f"\nCentral Difference (dx = {dxSin:.2f}):")
```

```
print(f"  Approximation = {fPrimeCentralSin:.6f}, Absolute
    Error = {errorCentralSin:.4e}")
```

Listing 4.4. Approximating the derivative of $f(x) = \sin(x)$ at $x_0 = \pi/4$.

Running this script provides the numerical approximations for the derivative of $f(x) = \sin(x)$ at $x_0 = \pi/4$, with the results summarized in Table 4.2. This example, using a non-polynomial function, offers a more general illustration of the performance of these schemes:

- As before, the forward and backward difference schemes produce errors of a similar magnitude. For this specific function and point, the backward difference happens to be slightly more accurate than the forward difference, but both are clearly first-order approximations.
- The central difference scheme is once again significantly more accurate. Its absolute error is more than an order of magnitude smaller than the errors from the one-sided schemes.
- Unlike the unique case with the quadratic function, the central difference error is no longer zero. This is because the third derivative of $\sin(x)$ (which is $-\cos(x)$) is non-zero, and therefore the leading term of the truncation error, $\frac{\Delta x^2}{6} f'''(x)$, does not vanish.

This outcome reinforces the general principle: for sufficiently smooth functions, the second-order central difference scheme will typically provide a much better approximation of the first derivative than the first-order forward or backward schemes.

Table 4.2. Finite difference approximations and absolute errors for the derivative of $f(x) = \sin(x)$ at $x_0 = \pi/4 \approx 0.7854$ (exact derivative is $\cos(\pi/4) \approx 0.7071$), using a step size of $\Delta x = 0.1$.

Difference Scheme	Approximation	Absolute Error
Forward Difference	0.6706	3.65×10^{-2}
Backward Difference	0.7413	3.41×10^{-2}
Central Difference	0.7059	1.18×10^{-3}

Numerically Verifying the Order of Accuracy

The theoretical order of accuracy (e.g., $\mathcal{O}(\Delta x)$ or $\mathcal{O}(\Delta x^2)$) predicts how the truncation error should decrease as the step size Δx is reduced. Specifically, if the error E is proportional to $(\Delta x)^p$, where p is the order of accuracy, then:

$$E \approx C \cdot (\Delta x)^p$$

Here, C is a constant that depends on the higher-order derivatives of the function being approximated but not on Δx. Taking the logarithm of both sides gives:

$$\log(E) \approx \log(C) + p \cdot \log(\Delta x)$$

This equation is in the form of a straight line, $Y = A + pX$, where $Y = \log(E)$, $X = \log(\Delta x)$, and the slope of the line is p, the order of accuracy.

Therefore, by calculating the error for a range of decreasing Δx values and plotting $\log(E)$ against $\log(\Delta x)$ (a "log-log plot"), we should observe points that lie approximately on a straight line. The slope of this line will give us the numerically observed order of accuracy, which can then be compared to the theoretical order.

Understanding Log-Log Plots for Error Analysis Log-log plots are powerful tools for visualizing relationships that span several orders of magnitude or follow power-law behavior, like our error relationship $E \approx C(\Delta x)^p$.

- **Visualizing Convergence Rate:** As Δx decreases (typically moving from right to left on the x-axis if Δx is plotted directly, or left to right if $\log(\Delta x)$ is plotted and $\Delta x < 1$), the error E also decreases.
- A first-order scheme ($p = 1$) means that if Δx is halved, the error E is also roughly halved. On a log-log plot, this will produce a line with a slope of 1.
- A second-order scheme ($p = 2$) means that if Δx is halved, the error E is roughly quartered (reduced by a factor of $2^2 = 4$). On a log-log plot, this will produce a steeper line with a slope of 2.
- Generally, a line with slope p on a log-log plot of error versus step size indicates p-th order accuracy. A steeper slope (larger p) signifies faster convergence of the error to zero as Δx is reduced.

Figure 4.8 provides a schematic illustration.

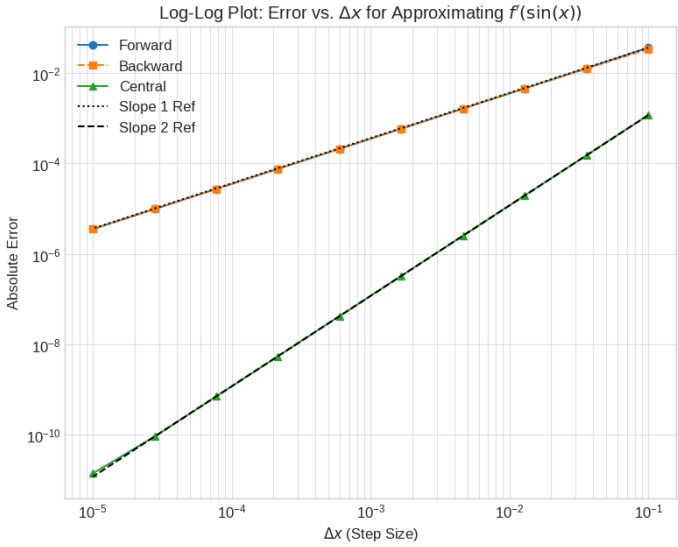


Figure 4.8. Schematic log-log plot of error versus step size Δx. A line with slope 1 indicates first-order accuracy, while a line with slope 2 indicates second-order accuracy. Steeper slopes correspond to higher orders of accuracy and faster error reduction as Δx decreases.

Let's perform this numerical verification for the derivative of $f(x) = \sin(x)$ at $x_0 = \pi/4$.

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

# Function and its derivative (defined in Listing LST:
    py_approx_deriv_sin_x)
# def functionSinX(x): return np.sin(x)
# def derivativeSinX(x): return np.cos(x)

x0Order = np.pi / 4.0
fPrimeExactOrder = derivativeSinX(x0Order)

# Range of dx values, decreasing logarithmically
# Example: 10 points from 10^-1 down to 10^-4
dxValues = np.logspace(-1, -4, 10)

errorsForwardOrder = []
errorsBackwardOrder = []
errorsCentralOrder = []

for dxCurr in dxValues:
    f0 = functionSinX(x0Order)
    fPlus = functionSinX(x0Order + dxCurr)
    fMinus = functionSinX(x0Order - dxCurr)

    # Forward difference error
    approxFwd = (fPlus - f0) / dxCurr
    errorsForwardOrder.append(abs(approxFwd - fPrimeExactOrder
    ))

    # Backward difference error
    approxBwd = (f0 - fMinus) / dxCurr
    errorsBackwardOrder.append(abs(approxBwd -
    fPrimeExactOrder))

    # Central difference error
    approxCtrl = (fPlus - fMinus) / (2 * dxCurr)
    errorsCentralOrder.append(abs(approxCtrl -
    fPrimeExactOrder))

# Convert lists to NumPy arrays for plotting and calculations
errorsForwardOrder = np.array(errorsForwardOrder)
errorsBackwardOrder = np.array(errorsBackwardOrder)
errorsCentralOrder = np.array(errorsCentralOrder)

# Create the log-log plot
plt.figure(figsize=(9, 7))
plt.loglog(dxValues, errorsForwardOrder, 'o-', label='Forward
    Diff. (Expected Order 1)')
plt.loglog(dxValues, errorsBackwardOrder, 's--', label='
    Backward Diff. (Expected Order 1)')
plt.loglog(dxValues, errorsCentralOrder, '^-', label='Central
    Diff. (Expected Order 2)')

# Add reference lines for slopes to guide the eye
# For Order 1: Error = C * dx^1. Choose C such that C*dxValues
    [0] approx errors[0]
if len(dxValues) > 0 and len(errorsForwardOrder) > 0:
```

```
    C1 = errorsForwardOrder[0] / dxValues[0]
    plt.loglog(dxValues, C1 * dxValues**1, 'k:', lw=1.5, label
    ='Reference Slope 1')
# For Order 2: Error = C * dx^2
if len(dxValues) > 0 and len(errorsCentralOrder) > 0:
    C2 = errorsCentralOrder[0] / (dxValues[0]**2)
    plt.loglog(dxValues, C2 * dxValues**2, 'k--', lw=1.5,
    label='Reference Slope 2')

plt.xlabel('$\Delta x$ (Step Size)')
plt.ylabel('Absolute Error')
plt.title('Log-Log Plot: Error vs. $\Delta x$ for
    Approximating $f\'(\sin(x))$ at $x=\pi/4$')
plt.legend(loc='best')
plt.grid(True, which="both", ls="-", alpha=0.7) # Grid for
    both major and minor ticks
# plt.gca().invert_xaxis() # Optional: if you prefer dx
    decreasing to the right
plt.show()
```

Listing 4.5. Numerically verifying the order of accuracy for finite difference schemes applied to $f(x) = \sin(x)$.

When this code is executed, the resulting log-log plot (an example of which would look similar to Figure 4.8, but with actual data points) should show that the errors for the forward and backward difference schemes align with a line of slope 1, confirming their first-order accuracy. The errors for the central difference scheme should align with a line of slope 2, confirming its second-order accuracy.

Interpreting the Log-Log Plot Results On such a plot, for the range of Δx where the truncation error dominates:

- The data points for the forward and backward difference methods should indeed form lines with a slope approximately equal to 1.
- The data points for the central difference method should form a line with a slope approximately equal to 2. This steeper slope visually confirms its faster rate of convergence.
- If Δx becomes *extremely* small (e.g., approaching machine precision, typically 10^{-8} to 10^{-16} for standard double precision), the plot might deviate from these straight lines. At this stage, **round-off errors** – small errors introduced by the finite precision of computer arithmetic – can begin to dominate the truncation error. For instance, when calculating $(q_{i+1} - q_i)/\Delta x$, if Δx is extremely small, q_{i+1} and q_i become very nearly equal. The subtraction $q_{i+1} - q_i$ might then suffer from a "loss of significance" or "catastrophic cancellation", where most of the significant digits are lost, and the result is dominated by the small errors in representing q_{i+1} and q_i. This can cause the error to increase or behave erratically for very tiny Δx. However, for the range of Δx values typically used in these illustrative examples (e.g., 10^{-1} down to 10^{-4} or 10^{-5}), the truncation error is usually the dominant source of error, and the expected convergence rates are clearly observable.

This numerical experiment provides strong evidence supporting our theoretical analysis of the order of accuracy for these fundamental finite difference schemes.

Concept Check

Imagine you are comparing two finite difference schemes, Scheme A and Scheme B, for approximating a derivative. On a log-log plot of error versus step size (Δx), Scheme A shows a line with a slope of 1, while Scheme B shows a line with a slope of 2 (assuming error decreases as Δx decreases). If your goal is to achieve a very small error, which scheme would likely require you to make Δx much smaller? Why does the difference in slope on this type of plot tell you that?

Chapter Summary: Approximating Spatial Change with Finite Differences

This chapter marked a crucial transition in our study of numerical methods, extending the concept of finite difference approximations from time derivatives (used for ODEs) to **spatial derivatives**. This skill is fundamental for numerically solving Partial Differential Equations (PDEs), where quantities vary in both space and time. We focused on understanding, deriving, and practically implementing basic finite difference schemes for the first spatial derivative.

Key concepts and methodologies covered in this chapter include:

- **Motivation for Spatial Discretization:**
  - Established the need to approximate spatial derivatives to model phenomena described by PDEs, where spatial variations are inherent.
  - Introduced the concept of a **spatial grid** (or mesh) – a set of discrete points x_i with spacing Δx – at which the solution is approximated.
- **The Finite Difference Method (FDM) for First Spatial Derivatives:**
  - Defined three fundamental schemes for approximating $\frac{dq}{dx}$ at a grid point x_i:
    - **Forward Difference:** Uses q_i and q_{i+1}. Formula: $\frac{q_{i+1}-q_i}{\Delta x}$.
    - **Backward Difference:** Uses q_i and q_{i-1}. Formula: $\frac{q_i-q_{i-1}}{\Delta x}$.
    - **Central Difference:** Uses q_{i-1} and q_{i+1}. Formula: $\frac{q_{i+1}-q_{i-1}}{2\Delta x}$.
  - Illustrated the geometric interpretation of these schemes as approximations of the slope.
- **Accuracy and Truncation Error Analysis via Taylor Series:**
  - Re-emphasized the use of **Taylor series expansions** as the theoretical tool for analyzing the accuracy of finite difference approximations.
  - Derived the **truncation error** for each of the three schemes:
    - Forward and Backward differences were shown to be **first-order accurate** in space, with a leading error term proportional to $\mathcal{O}(\Delta x)$.
    - The Central difference scheme was shown to be **second-order accurate** in space, with a leading error term proportional to $\mathcal{O}(\Delta x^2)$, generally making it more accurate for smooth functions.
  - Discussed the meaning of "order of accuracy" in terms of how rapidly the error decreases as Δx is reduced.

- **Practical Python Implementation and Error Calculation:**
  - Introduced Python's **f-strings** for clear and formatted output of numerical results.
  - Provided Python code examples to calculate the forward, backward, and central difference approximations for the derivatives of known functions ($f(x) = x^2$ and $f(x) = \sin(x)$).
  - Demonstrated how to compute the **absolute error** by comparing these numerical approximations with the exact analytical derivatives.
  - Highlighted the special case where the central difference for $f(x) = x^2$ is theoretically exact due to its third derivative being zero.
- **Numerical Verification of Order of Accuracy:**
  - Explained the rationale behind using **log-log plots** of error versus step size (Δx) to numerically determine the order of accuracy of a scheme.
  - Developed a Python script to perform this analysis for the $\sin(x)$ example, calculating errors for a range of Δx values.
  - Showed how the slopes of the lines on the log-log plot correspond to the theoretical orders of accuracy (slope ≈ 1 for first-order, slope ≈ 2 for second-order).
  - Briefly discussed the potential impact of **round-off errors** when Δx becomes extremely small, causing deviations from the expected convergence rates.

This chapter has equipped you with the ability to derive, implement, and analyze the accuracy of basic finite difference approximations for first-order spatial derivatives. These are essential building blocks that we will use extensively in the following chapters when we begin to construct full numerical schemes for solving Partial Differential Equations, starting with the heat (diffusion) equation and then the advection equation.

Chapter 4 Exercises

Apply your understanding of spatial finite differences and error analysis to complete the following exercises.

E4.1: Finite Difference Approximations for a Cubic Function

Consider the function $f(x) = x^3 - 2x^2 + x - 5$.

1. **Analytical Derivatives:** Calculate the exact first derivative $f'(x)$ and second derivative $f''(x)$ of this function.
2. **Approximations at a Point:** Choose $x_0 = 2.0$ and a step size $\Delta x = 0.2$. Calculate the exact value of $f'(x_0)$. Then, compute the following numerical approximations for $f'(x_0)$:
   - Forward difference
   - Backward difference
   - Central difference
3. **Errors:** For each approximation in part (b), calculate the absolute error by comparing it to the exact value of $f'(x_0)$. Which scheme is most accurate for this Δx?

E4.2: Truncation Error Analysis for $f(x) = x^3$

Let's analyze the function $f(x) = x^3$.

1. **Theoretical Truncation Errors:** Using the Taylor series expansions derived in this chapter, write out the *leading term* of the truncation error for:
   - The Forward Difference scheme applied to $f(x) = x^3$.
   - The Backward Difference scheme applied to $f(x) = x^3$.
   - The Central Difference scheme applied to $f(x) = x^3$.

   (Hint: You will need the second and third derivatives of $f(x) = x^3$.)
2. **Evaluate at $x_0 = 1.0, \Delta x = 0.1$:** Using your results from part (1), calculate the *estimated* error from the leading truncation error term for each scheme at $x_0 = 1.0$ with $\Delta x = 0.1$.
3. **Compare with Actual Errors:** Write a Python script to calculate the actual absolute errors for the forward, backward, and central difference approximations of $f'(1.0)$ for $f(x) = x^3$ using $\Delta x = 0.1$. Compare these actual errors with the estimated errors from part (2). How well does the leading truncation error term predict the actual error for this function and step size?

E4.3: Numerical Order of Accuracy for $f(x) = e^x$

Consider the function $f(x) = e^x$. Its analytical derivative is also $f'(x) = e^x$. We want to numerically verify the order of accuracy of the Forward, Backward, and Central difference schemes when approximating $f'(x)$ at $x_0 = 0.5$.

Tasks:

1. **Python Script Setup:** Write a Python script that:
   - Defines a Python function for $f(x) = e^x$ (use 'np.exp()').
   - Defines a Python function for its analytical derivative $f'(x) = e^x$.
   - Sets $x_0 = 0.5$ and calculates the exact value $f'(x_0)$.
   - Creates a list or NumPy array of decreasing Δx values (e.g., 'dxValues = np.logspace(-1, -5, 10)' which gives 10 points from 10^{-1} down to 10^{-5}).
2. **Calculate Errors:** Inside a loop that iterates through your 'dxValues':
   - For each 'dxCurrent' in 'dxValues', calculate the Forward, Backward, and Central difference approximations of $f'(x_0)$.
   - Calculate the absolute error for each approximation.
   - Store these errors in separate lists or NumPy arrays (e.g., 'errorsForward', 'errorsBackward', 'errorsCentral').
3. **Log-Log Plot:** Create a log-log plot of the absolute errors versus Δx for all three schemes (similar to the plot in Listing 4.5 for the $\sin(x)$ example). Include reference lines for slopes 1 and 2 to guide the eye.
4. **Analyze the Slopes:** Visually inspect the slopes of your plotted data. Do they match the theoretical orders of accuracy for each scheme (1 for Forward/Backward, 2 for Central) in the region where truncation error dominates?
5. **(Optional) Fit Slopes:** If you are familiar with it, you could try to fit a line to the log-log data (e.g., using 'np.polyfit' on $\log(\Delta x)$ and log(Error)) to get a numerical estimate of the slope for each scheme.

E4.4: Impact of Function Smoothness (Conceptual)

Consider two functions:

- $f_1(x) = \sin(x)$ (which is infinitely smooth, meaning all its derivatives exist and are continuous).
- $f_2(x) = |x|$ (the absolute value function, which has a "kink" or discontinuity in its first derivative at $x = 0$).

Suppose you try to approximate the first derivative of $f_2(x)$ at $x_0 = 0$ using the central difference scheme: $\frac{f_2(0+\Delta x)-f_2(0-\Delta x)}{2\Delta x} = \frac{|\Delta x|-|-\Delta x|}{2\Delta x}$.

1. What value does this central difference approximation give for $f_2'(0)$ for any $\Delta x > 0$?
2. Does the true derivative $f_2'(0)$ exist in the mathematical sense (i.e., is $f_2(x)$ differentiable at $x = 0$)?
3. The central difference scheme is formally $\mathcal{O}(\Delta x^2)$. This formal derivation relies on the function having a continuous third derivative, which appears in its leading truncation error term. What might happen to the observed convergence rate if you tried to make a log-log plot of error vs. Δx for approximating $f_2'(0.001)$ (a point very near the kink, but where the derivative is defined on either side)? What if you attempted to approximate a "derivative" at the kink itself using the formula, even if the true derivative doesn't exist there?

(This is a conceptual question designed to make you think about the assumptions behind the Taylor series-based error analysis. You don't need to write extensive code, but consider how the "smoothness" of a function impacts the performance and applicability of finite difference approximations.)

Chapter 5

Partial Differential Equations and Conservation Laws

Having established a foundation in solving Ordinary Differential Equations (ODEs) both analytically and numerically, and having explored the approximation of spatial derivatives using finite differences, we now transition to the broader and often more complex realm of **Partial Differential Equations (PDEs)**. This chapter aims to provide an understanding of how PDEs generalize ODEs by incorporating spatial variation, delve into the physical and mathematical underpinnings of conservation laws from which many PDEs are derived, and see how these concepts manifest in various Earth science processes. We will then focus on a cornerstone PDE, the Heat (or Diffusion) Equation, explore the critical role of boundary conditions, and develop a numerical scheme for its solution.

5.1 From Ordinary to Partial Differential Equations

In previous chapters, an Ordinary Differential Equation (ODE) was characterized as describing how a quantity u changes with respect to a *single* independent variable. Most commonly, this was time t, leading to solutions like $u(t)$, as seen in Newton's Law of Cooling (Equation 2.4) or radioactive decay (Equation 2.7). For example, in Newton's cooling law:

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

the temperature T is assumed to be spatially uniform within the object and varies only with time. This is a *lumped-parameter model*, suitable for small or well-mixed objects where internal spatial gradients are negligible. The analytical solution, $T(t) = T_{\text{env}} + (T_0 - T_{\text{env}})e^{-kt}$, gives the temperature as a function of time alone.

We also briefly noted that an ODE could describe purely spatial variation under steady-state conditions, such as the temperature profile $T(x)$ through a rock layer, which, for constant thermal conductivity κ and no heat sources, simplifies to $\kappa \frac{d^2T(x)}{dx^2} = 0$.

However, many, if not most, processes in the Earth sciences involve quantities that vary simultaneously in *both* space and time. Consider the cooling of a large magmatic dike intruding into cooler crustal rocks. The temperature within the dike and the surrounding host

rock will not only change as time progresses but will also be different at different locations (e.g., near the contact versus deep within the dike, or far into the host rock). In such scenarios, the temperature T is a function of at least one spatial coordinate (say, x, the distance from the contact) and time t, i.e., $T(x,t)$.

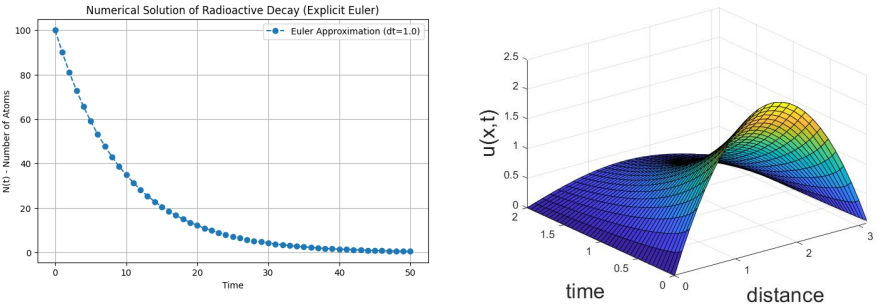


Figure 5.1. Comparison of solution types. Left: An Ordinary Differential Equation (ODE) solution, like the numerically approximated radioactive decay $N(t)$, depends on a single independent variable (time). Right: A Partial Differential Equation (PDE) solution, such as $u(x,t)$ for a diffusion process, is a surface or field that varies across multiple independent variables (here, a spatial dimension 'distance' and 'time').

A **Partial Differential Equation (PDE)** is an equation that involves an unknown function of two or more independent variables and its partial derivatives with respect to these variables. For quantities $u(x,t)$ or $u(x,y,t)$, etc., a PDE relates how u changes in time (e.g., $\frac{\partial u}{\partial t}$) to how it varies in space (e.g., $\frac{\partial u}{\partial x}$, $\frac{\partial^2 u}{\partial x^2}$). The "solution" to a PDE is a function (a field, distribution, or pattern) that describes the quantity of interest across the entire domain of independent variables.

Numerous Earth science phenomena are inherently spatially distributed and dynamic, thus requiring PDEs for their quantitative description:

- The temperature distribution within a cooling lava flow or the Earth's lithosphere.
- The volumetric water content $\theta(z,t)$ varying with depth z and time t during soil infiltration.
- The concentration $c(x,y,z,t)$ of volcanic ash dispersing in the atmosphere.
- The hydraulic head $h(x,y,t)$ in a groundwater aquifer.
- The propagation of seismic waves $u(x,y,z,t)$ through the Earth.

This chapter, and subsequent ones, will focus on understanding and numerically solving some of the fundamental PDEs encountered in these contexts.

5.1.1 Geoscience Examples of PDEs and Their Significance

Let's look more closely at some common PDEs that arise in Earth science applications.

The Heat (or Diffusion) Equation

A classic example where spatial temperature gradients are crucial is the cooling of a magmatic dike after its emplacement into cooler host rock. Initially hot, the dike loses heat to

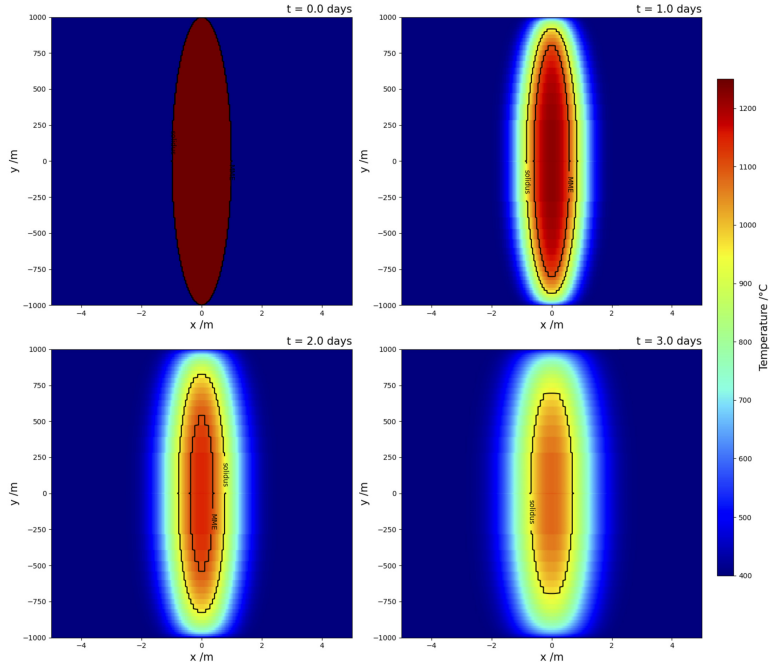


Figure 5.2. Numerical simulation of the cooling of a vertical magmatic dike emplaced in host rock. The four panels show the temperature distribution (color scale in °C) at different times: $t = 0.0$ days (initial hot dike), $t = 1.0$ day, $t = 2.0$ days, and $t = 3.0$ days. The black contours may represent specific isotherms like the solidus (temperature below which magma is completely solid) or other critical temperatures. The model captures the outward spread of heat and the progressive cooling of the dike's interior. (Image based on Loncart and Huppert, 2022, EPSL, adapted).

its surroundings, and the temperature distribution within both the dike and the host rock evolves over time. This process is governed by the heat equation. Figure 5.2 illustrates this temporal evolution.

Another important geoscience application involving heat transfer and PDEs is the study of permafrost dynamics, particularly its response to seasonal temperature variations and long-term climate change. Figure 5.3 illustrates key features of a permafrost environment.

When heat transfer is primarily by conduction, the evolution of temperature $T(x,t)$ (in a simplified 1D case, like through the thickness of a dike or a soil layer) is governed by the **Heat Equation** (also known as the Diffusion Equation):

$$\underbrace{\frac{\partial T}{\partial t}}_{\substack{\text{Rate of change}\\ \text{of T at a point}}} = \underbrace{\kappa}_{\substack{\text{Thermal}\\ \text{Diffusivity}}} \cdot \underbrace{\frac{\partial^2 T}{\partial x^2}}_{\substack{\text{Curvature of}\\ \text{T profile (Laplacian)}}} \tag{5.1}$$

Here:

- $\frac{\partial T}{\partial t}$ is the local rate of temperature change.

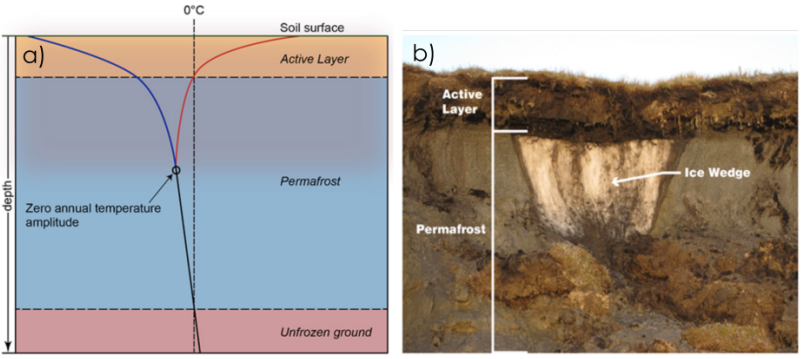


Figure 5.3. Characteristics of a permafrost environment. (a) Schematic annual temperature profile with depth (from "Nunavik and Nunatsiavut: From science to policy. An Integrated Regional Impact Study (IRIS) of climate change and modernization"): the blue curve shows the minimum winter temperatures, and the red curve the maximum summer temperatures. The "active layer" at the surface thaws seasonally, while the permafrost below remains frozen year-round. The depth of zero annual temperature amplitude marks a transition. (b) Photograph showing an exposed ice wedge within permafrost, a common feature in these environments (Photo credit: Benjamin Jones, USGS.). Modelling the thermal regime and thaw depth in permafrost requires solving the heat equation with appropriate surface boundary conditions reflecting seasonal and climatic forcing.

- κ is the thermal diffusivity (units: m^2/s), a material property indicating how quickly heat diffuses. It is defined as $\kappa = k_c/(\rho c_p)$, where k_c is the thermal conductivity, ρ is density, and c_p is the specific heat capacity.
- $\frac{\partial^2 T}{\partial x^2}$ is the second spatial derivative, representing the curvature of the temperature profile. It drives heat flow from regions of higher "average" temperature (concave down profile, $\partial^2 T/\partial x^2 < 0$, leading to cooling) to regions of lower "average" temperature (concave up profile, $\partial^2 T/\partial x^2 > 0$, leading to heating), thus smoothing out temperature differences over time.

The Heat Equation is a classic example of a **parabolic PDE**. Such equations typically model diffusive processes that evolve towards a smoother, more uniform state or equilibrium.

Richards' Equation for Unsaturated Flow

The movement of water in unsaturated soils is a highly complex, nonlinear process. A common model is Richards' Equation. In 1D vertical form, describing the change in volumetric water content $\theta(z,t)$ with depth z and time t, it can be written as:

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[D(\theta)\frac{\partial \theta}{\partial z}\right] - \frac{\partial K(\theta)}{\partial z} \tag{5.2}$$

Here:

- $\frac{\partial \theta}{\partial t}$ is the rate of change of water content.
- $D(\theta)$ is the hydraulic diffusivity, which depends strongly on θ. The term $\frac{\partial}{\partial z}[D(\theta)\frac{\partial \theta}{\partial z}]$ represents water movement driven by capillary forces (a diffusive-like process).

- $K(\theta)$ is the hydraulic conductivity, also highly dependent on θ. The term $\frac{\partial K(\theta)}{\partial z}$ represents water movement driven by gravity (and also pressure gradients if K is defined appropriately).

The strong dependence of D and K on θ makes Richards' equation highly **nonlinear** and challenging to solve both analytically and numerically.

The Advection-Diffusion Equation

Many transport processes in geosciences involve both the bulk movement (advection) of a substance by a flowing fluid and its simultaneous spreading or mixing due to diffusion or dispersion. The **Advection-Diffusion Equation** describes such phenomena. For the concentration $c(x, y, z, t)$ of a substance, it is:

$$\frac{\partial c}{\partial t} + \mathbf{u} \cdot \nabla c = \nabla \cdot (D_m \nabla c) + S_{source} \tag{5.3}$$

Where:

- $\frac{\partial c}{\partial t}$ is the local rate of change of concentration.
- $\mathbf{u} \cdot \nabla c$ is the **advective transport** term, representing the movement of c with the mean fluid velocity $\mathbf{u}$ (e.g., wind, river current).
- $\nabla \cdot (D_m \nabla c)$ is the **diffusive/dispersive transport** term, representing the spreading of c due to molecular diffusion or larger-scale turbulent mixing, characterized by the diffusion/dispersion coefficient D_m.
- S_{source} represents sources or sinks of the substance c (e.g., an erupting volcano emitting ash, or chemical reactions consuming a species).

This equation is fundamental for modeling, for example, the dispersion of volcanic ash plumes, the spread of pollutants in rivers or groundwater, or the transport of chemical species in geological fluids. It combines characteristics of hyperbolic PDEs (from the advection term) and parabolic PDEs (from the diffusion term).

All these examples underscore that PDEs are essential for capturing the dynamic interplay of processes that vary in both space and time across diverse Earth systems.

5.2 Physical Foundations: Conservation Laws and Frames of Reference

Many of the Partial Differential Equations (PDEs) encountered in the Earth sciences are not arbitrary mathematical constructs but are derived from fundamental physical principles, most notably **conservation laws**. To properly formulate these equations and understand their meaning, we also need to consider the framework, or frame of reference, from which we observe and describe fluid motion and transport processes. Before deriving the general transport equation, we will briefly discuss two important concepts: the continuum hypothesis and the distinction between Eulerian and Lagrangian descriptions of motion.

5.2.1 The Continuum Hypothesis in Geosciences

Fluid dynamics, which underpins the study of atmospheric circulation, ocean currents, magma flow, groundwater movement, and many other geoscience processes, typically treats fluids as **continua**. The word "fluid" itself often refers to liquids or gases, but in a broader sense, a fluid is any material that deforms continuously and irrecoverably under an applied shear stress over a sufficiently long timescale. This definition is particularly relevant in geosciences. For example, the Earth's mantle, while behaving as a solid elastic material on short timescales (transmitting seismic shear waves), deforms like an extremely viscous fluid over geological timescales (millions of years), driving plate tectonics. Even glaciers, composed of solid ice, flow and deform like highly viscous fluids over decades to millennia.

The **continuum hypothesis** assumes that a fluid can be treated as a continuous medium, ignoring its discrete molecular structure. This means that macroscopic properties like density (ρ), pressure (P), temperature (T), and velocity ($\mathbf{u}$) are considered to be well-defined, continuous functions of space and time, $P(x,y,z,t)$, $\mathbf{u}(x,y,z,t)$, etc., at any point within the fluid. This is an approximation, valid when the characteristic length scale of the phenomenon being studied (e.g., the size of a convection cell, the width of a river) is much larger than the mean free path of the molecules.

While the continuum hypothesis breaks down at very small scales (e.g., flow in nanopores) or in very rarefied gases (e.g., the uppermost atmosphere), it provides an extremely accurate and effective framework for the vast majority of fluid dynamics and transport problems encountered in Earth sciences. The PDEs we derive and solve are based on this fundamental assumption.

Concept Check

The continuum hypothesis allows us to define properties like density $\rho(x,y,z,t)$ and velocity $\mathbf{u}(x,y,z,t)$ as continuous fields. Why is this hypothesis a crucial first step before we can even write down conservation laws in the form of Partial Differential Equations? What might be a geoscience example where the continuum hypothesis could start to break down?

5.2.2 Describing Motion: Eulerian and Lagrangian Viewpoints

There are two classical ways to describe the motion of a fluid or the transport of quantities within it: the Lagrangian viewpoint and the Eulerian viewpoint.

The Lagrangian Description

In the **Lagrangian description**, we select and follow individual fluid parcels (or discrete particles) as they move through space and time. The properties of each parcel (e.g., its position, velocity, temperature) are tracked as functions of time.

- The independent variables are typically time t and an identifier for the initial position of the parcel, $\mathbf{r}_0 = (x_0, y_0, z_0)$.
- The position of a parcel that was at $\mathbf{r}_0$ at time $t = 0$ is given by $\mathbf{x}_{\mathbf{r}_0}(t)$.
- Other properties, like velocity $\mathbf{u}_{\mathbf{r}_0}(t)$ or temperature $T_{\mathbf{r}_0}(t)$, are also functions of time for that specific parcel.

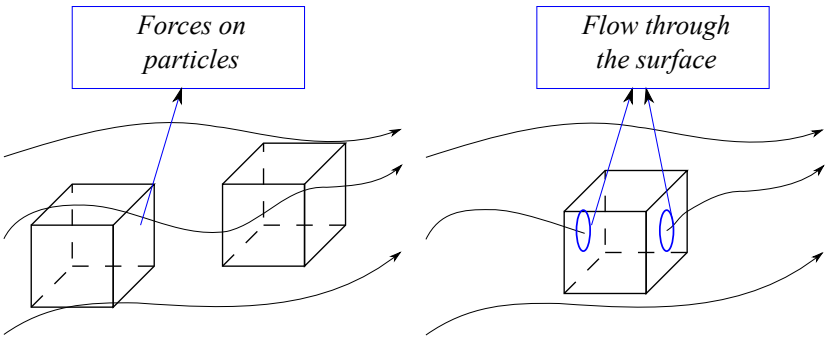


Figure 5.4. Conceptual difference between the Lagrangian and Eulerian viewpoints. Left (Lagrangian-like): Focus is on individual fluid parcels or control volumes that move and deform with the flow, often analyzing forces acting *on* these moving particles/volumes. Right (Eulerian-like): Focus is on a fixed control volume in space, analyzing the flux of quantities *through* its stationary surfaces.

This approach is akin to attaching a sensor to a small float in a river and recording its path and the water properties it encounters. It is naturally suited for problems involving particle tracking (e.g., transport of individual ash particles, sediment grains, or pollutants) or the deformation of specific material bodies. Newton's laws of motion are inherently Lagrangian, as they apply to individual masses.

Figure 5.5. Lagrangian viewpoint: Tracking an individual fluid parcel (represented as a deforming volume) as it moves along a streamline within a flow field over time.

The Eulerian Description

In the **Eulerian description**, we focus on fixed points or fixed regions (control volumes) in space and observe the fluid properties as different fluid parcels pass through these locations.

- The independent variables are the spatial coordinates (x, y, z) of a fixed point and time t.

- Fluid properties like velocity $\mathbf{u}(x,y,z,t)$, pressure $P(x,y,z,t)$, and temperature $T(x,y,z,t)$ are defined as fields, i.e., functions of fixed spatial position and time.

This approach is like standing on a bridge and observing the water velocity and temperature at a fixed point below, or using a fixed weather station to measure wind speed and air temperature. Most PDEs in fluid dynamics and transport phenomena (like the Heat Equation or Advection Equation) are formulated in the Eulerian frame because it is often more convenient for describing fields and deriving conservation laws based on fluxes through fixed boundaries.

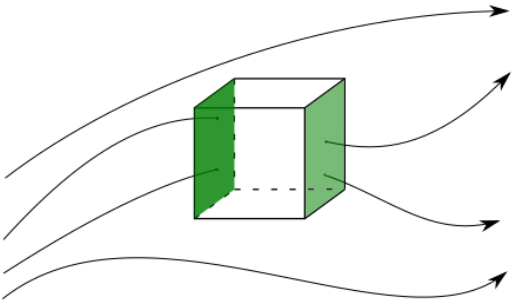


Figure 5.6. Eulerian viewpoint: Observing fluid flow through a fixed control volume in space. Fluid properties are measured at fixed locations or as fluxes across the boundaries of this volume.

Concept Check

Consider a plume of volcanic ash dispersing in the atmosphere.

1. If you were to model the concentration of ash using a fixed grid of monitoring stations on the ground, would this primarily represent an Eulerian or a Lagrangian approach to measuring concentration? Why?
2. If, instead, you released a weather balloon designed to stay within the ash cloud and measure its properties as it travels, which descriptive framework would this represent?
3. Can you think of one advantage and one disadvantage for each approach (Eulerian and Lagrangian) when trying to understand the overall dispersion of the ash plume?

The Material (or Substantial) Derivative: Connecting Lagrangian and Eulerian Views

Consider a fluid parcel moving through a flow field where a property ϕ (e.g., temperature) varies in both space and time, i.e., $\phi = \phi(x,y,z,t)$ in an Eulerian sense. If we follow this specific parcel (Lagrangian view), the rate of change of ϕ *experienced by that parcel* is

given by the **material derivative** (also known as substantial, total, or Lagrangian derivative), denoted as $\frac{D\phi}{Dt}$.

This change has two contributions:

1. The *local rate of change* of ϕ at a fixed point in space: $\frac{\partial \phi}{\partial t}$. This is the change one would observe if staying stationary.

2. The *advective rate of change* due to the parcel moving to a new location where ϕ has a different value: $\mathbf{u} \cdot \nabla \phi$, where $\mathbf{u}$ is the velocity of the parcel and $\nabla \phi$ is the spatial gradient of ϕ. This term can be expanded as $u_x \frac{\partial \phi}{\partial x} + u_y \frac{\partial \phi}{\partial y} + u_z \frac{\partial \phi}{\partial z}$.

Combining these, the material derivative is:

$$\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi \tag{5.4}$$

This crucial operator links the Eulerian and Lagrangian perspectives. It represents the time rate of change of ϕ following a moving fluid element.

For example, consider a river where the water temperature $\phi = T$ is steady in an Eulerian sense ($\partial T / \partial t = 0$) but varies spatially (e.g., cooler upstream, warmer downstream). A raft (fluid parcel) moving with the current $\mathbf{u}$ will still experience a change in its temperature ($\frac{DT}{Dt} = \mathbf{u} \cdot \nabla T \neq 0$) as it moves from cooler to warmer regions.

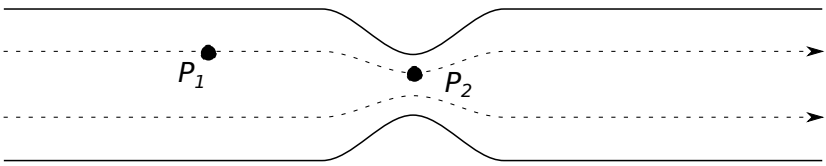


Figure 5.7. Illustration of how a fluid parcel can experience a change in a property even in a steady Eulerian flow field. In a river with varying width, if the flow is steady (Eulerian $\partial / \partial t = 0$), the velocity at any fixed point is constant. However, a fluid parcel (e.g., P_1 moving to P_2) accelerates as it passes through a narrower section, so its velocity changes in a Lagrangian sense ($Dv/Dt \neq 0$).

While the Lagrangian approach is intuitive for particle motion, the Eulerian framework is generally more convenient for deriving conservation laws for continua based on fluxes through fixed control volumes, which is the approach we will primarily adopt.

Concept Check

The material derivative is given by $\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi$. Imagine you are in a boat (a "fluid parcel") drifting down a river where the water depth $\phi = h(x, t)$ changes both along the river (x) and possibly over time t due to tides.

1. If the river flow is steady (i.e., $\frac{\partial h}{\partial t} = 0$ at any fixed location), but the riverbed profile changes (e.g., it gets shallower downstream, so $\frac{\partial h}{\partial x} < 0$), will you, in your boat, experience a change in water depth as you drift? Which term in the material derivative would describe this change?
2. Now, imagine the riverbed is perfectly flat ($\frac{\partial h}{\partial x} = 0$), but a tidal bore is moving up the river, causing the local depth to increase over time ($\frac{\partial h}{\partial t} > 0$). If your boat is drifting with the main river current (assume $\mathbf{u}$ is still downstream), what change in depth do you experience?

5.2.3 Conservation Laws as the Foundation of Transport PDEs

As previously mentioned, many of the Partial Differential Equations (PDEs) that describe physical phenomena in the Earth sciences are mathematical expressions of fundamental **conservation laws**. These laws state that certain physical quantities (like mass, momentum, or energy) are conserved within a defined system or control volume, meaning they can change only due to fluxes across the system's boundaries or due to internal sources or sinks.

To derive the governing PDEs, we typically follow a consistent modeling philosophy:

1. **Choose a Physical Principle:** Identify the relevant conservation law (e.g., conservation of mass, conservation of energy, Newton's second law for conservation of momentum).
2. **Define a Control Volume (CV):** Select a representative, finite (though often considered infinitesimally small in the limit) volume of space through which the quantity of interest can flow and within which it can be generated or consumed. We will primarily use an Eulerian approach, where the control volume is fixed in space.
3. **Apply the Principle to the Control Volume:** Formulate a balance equation for the chosen quantity within the control volume. This balance typically takes the generic form:

$$\begin{gathered} \textit{Rate of Accumulation of Quantity within CV} \\ = \\ \textit{Net Rate of Influx of Quantity into CV across its Boundaries} \\ + \\ \textit{Net Rate of Generation of Quantity within CV} \end{gathered}$$

4. **Derive the Mathematical Equation:** By expressing each term in the balance equation mathematically (often involving integrals of fluxes over surfaces and rates over volumes) and then taking the limit as the control volume shrinks to an infinitesimal point, we arrive at a PDE.

This systematic approach allows us to build mathematical models from first principles.

The primary physical principles we will rely on are:

- **Conservation of Mass:** States that the mass of a closed (isolated) system remains constant. For an open system (like our control volume), the rate of change of mass within the volume equals the net rate at which mass flows across its boundaries.

- **Conservation of Momentum (Newton's Second Law):** States that the rate of change of momentum of a body is equal to the net force acting on it. For a fluid, this relates changes in flow velocity to pressure gradients, viscous stresses, and body forces like gravity.
- **Conservation of Energy (First Law of Thermodynamics):** States that energy cannot be created or destroyed, only transformed from one form to another. For a control volume, the rate of change of total energy equals the net rate of heat transfer into the volume plus the net rate of work done on the volume.

Let us now formalize the derivation of a general scalar transport equation. Consider a **specific quantity** ϕ. This ϕ is typically a quantity expressed per unit mass (e.g., velocity which is momentum per unit mass, or internal energy per unit mass, or mass fraction of a chemical species). We are interested in how the total amount of this quantity, $\rho\phi$ (where ρ is the density, so $\rho\phi$ is the quantity per unit volume), changes within a fixed Cartesian control volume $V = \Delta x \Delta y \Delta z$, as depicted in Figure 5.8.

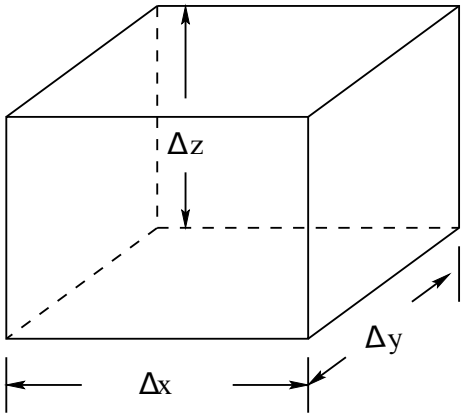


Figure 5.8. A finite Cartesian control volume V with dimensions $\Delta x, \Delta y, \Delta z$, fixed in space (Eulerian frame), used for deriving conservation equations.

The general conservation principle for the quantity ϕ within this control volume V over a time interval Δt can be stated as:

> *Rate of accumulation of $(\rho\phi)$ in V = Net rate of influx of $(\rho\phi)$ into V across its surface S + Net rate of generation of $(\rho\phi)$ inside V.*

Let's examine each term.

Accumulation Term

The "accumulation of $\rho\phi$ in the control volume over time Δt" is the change in the total amount of $\rho\phi$ within V from time t to time $t + \Delta t$. If we approximate the average value of $\rho\phi$ within the volume at these times, this term can be written as:

$$\text{Accumulation} = \left[(\rho\phi)_{\text{avg}} \Delta V\right]_{t+\Delta t} - \left[(\rho\phi)_{\text{avg}} \Delta V\right]_{t} \tag{5.5}$$

where $\Delta V = \Delta x \Delta y \Delta z$. The rate of accumulation is this quantity divided by Δt. As $\Delta t \to 0$, and assuming $(\rho\phi)_{\text{avg}}$ can be represented by the value at the center of the (now infinitesimal)

control volume, this rate becomes:

$$\text{Rate of Accumulation} = \frac{\partial(\rho\phi)}{\partial t}\Delta V \tag{5.6}$$

Generation Term

The "net generation of $\rho\phi$ inside the control volume V" accounts for any processes that create or destroy the quantity ϕ within the volume itself (e.g., chemical reactions producing a species, radioactive heat production). Let S_ϕ be the volumetric rate of generation of $\rho\phi$ (quantity per unit volume per unit time). The total amount generated in volume ΔV over time Δt is approximately $(S_\phi)_{\text{avg}}\Delta V\Delta t$. The rate of generation is then:

$$\text{Rate of Generation} = (S_\phi)_{\text{avg}}\Delta V \tag{5.7}$$

As the control volume becomes infinitesimal, this becomes $S_\phi\Delta V$, where S_ϕ is the local volumetric source rate.

Net Influx Term

The "net influx of $\rho\phi$ into the control volume V through its surface S" represents the transport of the quantity across the six faces of our Cartesian control volume (Figure 5.9). Let $\mathbf{J}_\phi$ be the **flux vector** of the quantity $\rho\phi$. The component $J_{\phi,x}$ represents the rate at which $\rho\phi$ flows per unit area in the x-direction.

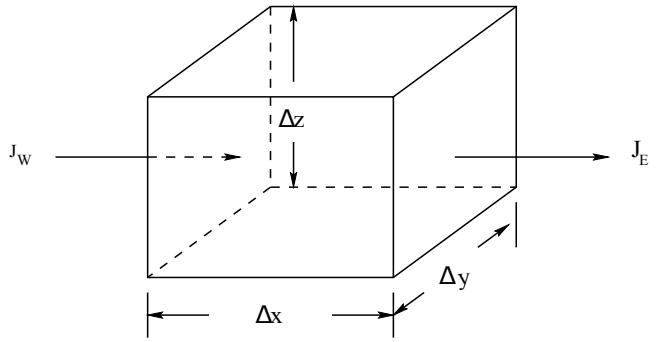


Figure 5.9. Fluxes of a quantity ϕ through the faces of the control volume. For example, J_W is the flux entering through the west face (at x) and J_E is the flux leaving through the east face (at $x+\Delta x$). Similar fluxes occur for the north/south (y-direction) and top/bottom (z-direction) faces.

Consider the faces perpendicular to the x-axis:

- Influx through the west face (at x, area $\Delta y\Delta z$): $(J_{\phi,x})_{\text{at } x}\cdot(\Delta y\Delta z)$
- Outflux through the east face (at $x+\Delta x$, area $\Delta y\Delta z$): $(J_{\phi,x})_{\text{at } x+\Delta x}\cdot(\Delta y\Delta z)$

The net influx in the x-direction is therefore $[(J_{\phi,x})_{\text{at } x} - (J_{\phi,x})_{\text{at } x+\Delta x}]\Delta y\Delta z$. Similar expressions apply for the y and z directions:

- Net influx in y-direction: $[(J_{\phi,y})_{\text{at } y} - (J_{\phi,y})_{\text{at } y+\Delta y}]\Delta x\Delta z$

- Net influx in z-direction: $[(J_{\phi,z})_{\text{at } z} - (J_{\phi,z})_{\text{at } z+\Delta z}]\Delta x \Delta y$

The total net rate of influx is the sum of these contributions. Dividing the net influx in the x-direction by the volume $\Delta V = \Delta x \Delta y \Delta z$ gives:

$$\frac{(J_{\phi,x})_{\text{at } x} - (J_{\phi,x})_{\text{at } x+\Delta x}}{\Delta x}$$

In the limit as $\Delta x \to 0$, this becomes $-\frac{\partial J_{\phi,x}}{\partial x}$. Summing for all directions, the net rate of influx per unit volume becomes $-\left(\frac{\partial J_{\phi,x}}{\partial x} + \frac{\partial J_{\phi,y}}{\partial y} + \frac{\partial J_{\phi,z}}{\partial z}\right) = -\nabla \cdot \mathbf{J}_\phi$. The total net rate of influx for the volume ΔV is thus $(-\nabla \cdot \mathbf{J}_\phi)\Delta V$.

The flux $\mathbf{J}_\phi$ itself can be due to different physical mechanisms, primarily **advection** (transport by bulk fluid motion) and **diffusion** (transport due to gradients, e.g., molecular motion or turbulent eddies). We will detail these flux components shortly.

Advective and Diffusive Flux Components

Advective (or Convective) Flux The **advective flux** (often termed convective flux in a broader sense) represents the transport of the quantity ϕ due to the bulk motion of the fluid itself. If the fluid is moving with a velocity $\mathbf{u} = (u_x, u_y, u_z)$, then the quantity $\rho\phi$ is carried along with it. The advective flux component in the x-direction, for example, is given by:

$$J_{\phi,adv,x} = (\rho u_x)\phi \tag{5.8}$$

In general vector form, the advective flux is $\mathbf{J}_{\phi,adv} = \rho \mathbf{u} \phi$. This term describes how the mean flow field transports the property ϕ. As noted previously, "convection" is sometimes reserved for buoyancy-driven flows, while "advection" refers to transport by an imposed velocity field. For our purposes here, we consider them to represent the same mechanism of bulk transport.

Diffusive Flux The **diffusive flux** represents the transport of ϕ due to random molecular motion (molecular diffusion) or, on larger scales, due to turbulent eddies (turbulent diffusion or dispersion). This transport mechanism always acts to move the quantity ϕ from regions of higher concentration (or higher values of ϕ) to regions of lower concentration, effectively smoothing out gradients.

A common model for diffusive flux is **Fick's First Law** (or an analogous law for other quantities like heat, e.g., Fourier's Law). For a scalar quantity ϕ, Fick's Law states that the diffusive flux is proportional to the negative of the gradient of ϕ. The constant of proportionality is the diffusion coefficient. For the x-component of the diffusive flux, this is:

$$J_{\phi,diff,x} = -\Gamma \frac{\partial \phi}{\partial x} \tag{5.9}$$

where Γ (Gamma) is the **diffusion coefficient** (or diffusivity) for the quantity ϕ in the medium (units, e.g., m^2/s). A larger Γ means faster diffusion. In general vector form, the diffusive flux is $\mathbf{J}_{\phi,diff} = -\Gamma \nabla \phi$.

Figure 5.10 illustrates the concept of diffusion from a microscopic and macroscopic perspective. Initially, particles (representing molecules or a tracer) are confined to one side of a domain. When a barrier is removed, random motion causes the particles to spread out, occupying the available space and leading to a smoother concentration profile over time.

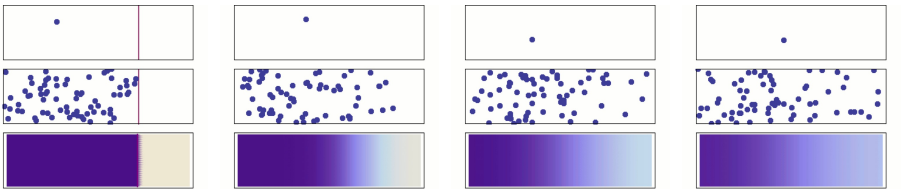


Figure 5.10. Schematic illustration of the diffusion process over time (progressing from left to right). Top panels: A single particle's random walk. Middle panels: Many particles spreading out from an initially confined region (left) after a barrier (purple line) is notionally removed. Bottom panels: The corresponding macroscopic concentration profile, which starts as a sharp step and gradually smooths out due to diffusion. This illustrates how random microscopic motions lead to a net transport from high to low concentration regions.

Total Flux The total flux of $\rho\phi$ is the sum of the advective and diffusive components. For example, in the x-direction:

$$J_{\phi,x} = J_{\phi,adv,x} + J_{\phi,diff,x} = (\rho u_x)\phi - \Gamma \frac{\partial \phi}{\partial x} \tag{5.10}$$

Similar expressions hold for the y and z directions:

$$J_{\phi,y} = (\rho u_y)\phi - \Gamma \frac{\partial \phi}{\partial y} \tag{5.11}$$

$$J_{\phi,z} = (\rho u_z)\phi - \Gamma \frac{\partial \phi}{\partial z} \tag{5.12}$$

In vector form, the total flux is $\mathbf{J}_\phi = \rho \mathbf{u}\phi - \Gamma \nabla \phi$.

5.2.4 The Roles of Advection and Diffusion in Mixing: Stirring vs. Homogenization

Understanding the distinct roles of advection (bulk transport by flow) and diffusion (transport down gradients) is crucial when considering how substances or properties are **mixed** or homogenized within a fluid system, such as a magma chamber.

Imagine a magma chamber initially containing resident magma. A new batch of magma with different properties (e.g., different chemical composition or temperature, which we can represent with a scalar ϕ) is then injected into this chamber. Figure 5.11 illustrates such a scenario: the left panel depicts a schematic convective flow field within the chamber, and the right panel shows an initial distribution where a distinct region of "new magma" (e.g., $\phi = 1$) is present within the "resident magma" (e.g., $\phi = 0$).

Over time, the convective flow field (advection) will act to deform and transport this new magma. As shown in the left panel of Figure 5.12, if only advection is considered (i.e., diffusion is negligible, $\Gamma = 0$), the complex flow will stretch and fold the initial blob of new magma into intricate, thin filaments. This process, called **stirring**, significantly increases the interfacial area between the two magma types. However, advection alone does not change the local values of ϕ; the filaments of new magma will still have $\phi = 1$, and the surrounding resident magma will still have $\phi = 0$. No intermediate values are created.

True **homogenization**, where intermediate values of ϕ (between 0 and 1) are produced, requires **diffusion** to act across these greatly increased interfaces. The right panel of Figure

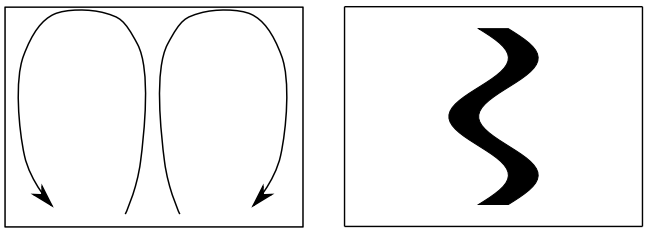


Figure 5.11. Initial state for a mixing scenario in a magma chamber (conceptual). (Left) A schematic representation of a convective flow field (arrows indicating circulation). (Right) An initial distribution of a scalar property ϕ, showing a distinct region of "new" magma (e.g., high ϕ) within the resident magma (low ϕ).

5.12 illustrates the state after a long time when both advection and diffusion are active. The initial sharp boundaries have been blurred, and a more homogeneous mixture with intermediate values of ϕ has developed. Diffusion effectively smooths out the fine-scale heterogeneities created by advective stirring.

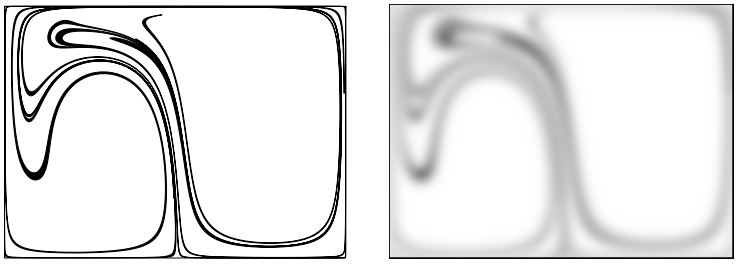


Figure 5.12. Comparison of the effects of advection alone versus advection with diffusion after a long time, starting from an initial heterogeneity. (Left) **Stirring by Advection Only ($\Gamma = 0$):** The initial heterogeneity is stretched and folded into complex filaments by the convective flow, but local values remain distinct (e.g., only 0 or 1). (Right) **Mixing (Advection + Diffusion, $\Gamma > 0$):** Diffusion acts across the interfaces of the stretched filaments, leading to true homogenization and the creation of intermediate values of the scalar property ϕ.

This interplay between large-scale advective stirring and small-scale molecular or turbulent diffusion is fundamental to mixing processes in many Earth systems. For instance, the chemical hybridization observed in magmas often results from the mechanical mingling of different magma batches by convection (stirring), followed by chemical diffusion across the stretched interfaces to produce a more compositionally homogeneous melt. The efficiency of this overall mixing process depends on the vigor of the convection (which creates interfacial area) and the magnitude of the diffusion coefficient (which governs the rate of homogenization at the interfaces). This understanding of mixing is crucial for interpreting geochemical signatures in volcanic rocks.

In many other natural Earth systems, both advection and diffusion are present. For exam-

ple, in the atmosphere, large-scale winds advect volcanic ash or pollutants, while smaller-scale turbulence (a form of effective diffusion or dispersion) causes them to spread and dilute. The relative importance of advection versus diffusion is often characterized by dimensionless numbers like the Péclet number ($Pe = UL/\Gamma$, where U is a characteristic velocity and L is a characteristic length scale).

5.3 Deriving the General Scalar Transport Equation

Having established the fundamental concepts of conservation, the Eulerian frame of reference, and the nature of advective and diffusive fluxes, we are now prepared to derive the general mathematical equation that describes how a scalar quantity ϕ is transported within a fluid. As before, ϕ represents a specific quantity (quantity per unit mass), and we will track the evolution of $\rho\phi$ (quantity per unit volume).

We return to the fundamental conservation principle applied to a fixed Cartesian control volume $V = \Delta x \Delta y \Delta z$ (Figure 5.8) over a time interval Δt:

> *Accumulation of $(\rho\phi)$ in V over Δt = Net influx of $(\rho\phi)$ into V through its surface S over Δt + Net generation of $(\rho\phi)$ in V over Δt.*

Let's express each part of this balance mathematically, building upon our previous definitions.

The total amount of the quantity $\rho\phi$ within the control volume at time t is $(\rho\phi)_{avg,t}\Delta V$. The change in this amount over Δt is:

$$\text{Change in stored quantity} = \left[(\rho\phi)_{avg}\Delta V\right]_{t+\Delta t} - \left[(\rho\phi)_{avg}\Delta V\right]_t \tag{5.13}$$

The net influx of $\rho\phi$ into the control volume across its six faces over the time interval Δt was derived in Section 5.2.3. Using the components of the total flux vector $\mathbf{J}_\phi = (J_{\phi,x}, J_{\phi,y}, J_{\phi,z})$, where each component includes both advective and diffusive parts (e.g., $J_{\phi,x} = \rho u_x \phi - \Gamma \frac{\partial \phi}{\partial x}$ as in Equation 5.10), the net influx is:

$$\begin{aligned}\text{Net Influx over } \Delta t = \\ &\left[(J_{\phi,x})_{\text{at } x} - (J_{\phi,x})_{\text{at } x+\Delta x}\right]\Delta y \Delta z \Delta t \\ +&\left[(J_{\phi,y})_{\text{at } y} - (J_{\phi,y})_{\text{at } y+\Delta y}\right]\Delta x \Delta z \Delta t \\ &+\left[(J_{\phi,z})_{\text{at } z} - (J_{\phi,z})_{\text{at } z+\Delta z}\right]\Delta x \Delta y \Delta t\end{aligned} \tag{5.14}$$

The net generation of $\rho\phi$ within the control volume over time Δt, due to sources or sinks, is given by:

$$\text{Net Generation over } \Delta t = (S_\phi)_{avg}\Delta V \Delta t \tag{5.15}$$

where $(S_\phi)_{avg}$ is the average volumetric source rate of $\rho\phi$.

Equating the change in stored quantity (Equation 5.13) to the sum of net influx (Equation 5.14) and net generation (Equation 5.15):

$$\begin{aligned}&[(\rho\phi)\Delta V]_{t+\Delta t} - [(\rho\phi)\Delta V]_t = \\ &\Big\{\left[(J_{\phi,x})_{\text{at } x} - (J_{\phi,x})_{\text{at } x+\Delta x}\right]\Delta y \Delta z \\ &+\left[(J_{\phi,y})_{\text{at } y} - (J_{\phi,y})_{\text{at } y+\Delta y}\right]\Delta x \Delta z \\ &+\left[(J_{\phi,z})_{\text{at } z} - (J_{\phi,z})_{\text{at } z+\Delta z}\right]\Delta x \Delta y\Big\}\,\Delta t \\ &+(S_\phi)\Delta V \Delta t\end{aligned} \tag{5.16}$$

(Here, for brevity, we use $(\rho\phi)$ and S_ϕ to imply their average values over the control volume and time step where appropriate).

Now, we divide the entire equation by $\Delta V \Delta t = \Delta x \Delta y \Delta z \Delta t$ to get the balance on a per unit volume, per unit time basis:

$$\frac{[(\rho\phi)]_{t+\Delta t} - [(\rho\phi)]_t}{\Delta t} = - \frac{(J_{\phi,x})_{x+\Delta x} - (J_{\phi,x})_x}{\Delta x} - \frac{(J_{\phi,y})_{y+\Delta y} - (J_{\phi,y})_y}{\Delta y} - \frac{(J_{\phi,z})_{z+\Delta z} - (J_{\phi,z})_z}{\Delta z} + S_\phi \quad (5.17)$$

In the limit as $\Delta t \to 0$, $\Delta x \to 0$, $\Delta y \to 0$, and $\Delta z \to 0$:

- The left-hand side becomes the partial derivative with respect to time: $\frac{\partial(\rho\phi)}{\partial t}$.
- Each term like $-\frac{(J_{\phi,x})_{x+\Delta x} - (J_{\phi,x})_x}{\Delta x}$ becomes, by definition of the partial derivative, $-\frac{\partial J_{\phi,x}}{\partial x}$.

Thus, we arrive at the differential form of the conservation equation in terms of flux components:

$$\frac{\partial(\rho\phi)}{\partial t} = -\left(\frac{\partial J_{\phi,x}}{\partial x} + \frac{\partial J_{\phi,y}}{\partial y} + \frac{\partial J_{\phi,z}}{\partial z} \right) + S_\phi \quad (5.18)$$

This can be written more compactly using the divergence operator $(\nabla\cdot)$:

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot \mathbf{J}_\phi = S_\phi \quad (5.19)$$

where $\mathbf{J}_\phi = (J_{\phi,x}, J_{\phi,y}, J_{\phi,z})$ is the total flux vector.

Finally, substituting the expression for the total flux vector, $\mathbf{J}_\phi = \rho\mathbf{u}\phi - \Gamma\nabla\phi$ (which combines the advective flux $\rho\mathbf{u}\phi$ and the diffusive flux $-\Gamma\nabla\phi$), into Equation 5.19, we obtain the **general scalar transport equation** (also known as the advection-diffusion-reaction equation) in its conservative (or divergence) form:

$$\boxed{\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho\mathbf{u}\phi) = \nabla \cdot (\Gamma\nabla\phi) + S_\phi} \quad (5.20)$$

This fundamental equation describes how the quantity $\rho\phi$ (representing the concentration of ϕ per unit volume) changes at a point in space and time. It clearly delineates the contributing physical processes:

- $\frac{\partial(\rho\phi)}{\partial t}$: The **rate of accumulation** (or unsteady term), representing the local rate of change of $\rho\phi$ over time.
- $\nabla \cdot (\rho\mathbf{u}\phi)$: The **advective (or convective) transport term**. The divergence of the advective flux $\rho\mathbf{u}\phi$ represents the net rate at which $\rho\phi$ is transported out of (if positive) or into (if negative) an infinitesimal volume due to the bulk motion of the fluid $\mathbf{u}$.
- $\nabla \cdot (\Gamma\nabla\phi)$: The **diffusive transport term**. The divergence of the diffusive flux $\mathbf{J}_{\text{diff}} = -\Gamma\nabla\phi$ represents the net rate at which $\rho\phi$ is transported out of or into an infinitesimal volume due to diffusion. (Note: $\nabla \cdot (\Gamma\nabla\phi)$ can be written as $\Gamma\nabla^2\phi + \nabla\Gamma \cdot \nabla\phi$; if Γ is constant, it simplifies to $\Gamma\nabla^2\phi$, where ∇^2 is the Laplacian operator).
- S_ϕ: The **source (or sink) term**, representing the volumetric rate of generation or consumption of $\rho\phi$ due to other processes (e.g., chemical reactions, radioactive decay producing heat).

As previously noted, this derivation, while using a Cartesian control volume for simplicity, yields a general differential equation valid in any coordinate system. The specific form of the divergence ($\nabla\cdot$) and gradient (∇) operators will, however, depend on the chosen coordinate system (e.g., cylindrical, spherical).

A very illustrative example of applying these control volume concepts in a geoscience context is the modeling of a **volcanic eruption plume**, as depicted schematically in Figure 5.13. An eruption plume is a complex, multiphase flow where a mixture of hot volcanic gases and solid particles (ash, lapilli) is ejected from a vent and rises buoyantly into the atmosphere, spreading laterally at a level of neutral buoyancy.

If we consider a horizontal slice of this plume at a certain height z with thickness Δz as our control volume, the conservation laws we have discussed can be applied to various quantities:

- **Conservation of Mass (Particles):** The rate of change of particle mass within the slice depends on the advective fluxes of particles at the bottom and top of the slice, and the "Particle Loss" (as shown in red) due to sedimentation from the plume's margins. This particle loss acts as a negative source term (a sink) for particle mass within that control volume slice.

- **Conservation of Mass (Total):** The rate of change of total mass (gas + particles) within the slice depends on the mass flux entering from below, the mass flux exiting above, the particle loss from the plume's margins, and critically, the net mass entrained from the surrounding atmosphere through the lateral boundaries of the plume segment. "Air Entrainment" (as shown in blue) is a significant source of mass for the plume.

- **Conservation of Momentum:** The change in momentum of the material in the slice is governed by the momentum fluxes, pressure forces, buoyancy forces (due to density differences with the ambient air), and drag forces from entrained air.

- **Conservation of Energy (Thermal):** The change in thermal energy within the slice is affected by the advection of enthalpy, heat exchange with entrained air, and radiative heat loss.

Each of these balances, when formulated for an infinitesimal control volume and considering the relevant advective and diffusive fluxes (e.g., turbulent diffusion/dispersion for momentum and heat), leads to a set of coupled PDEs that describe the plume's dynamics and its interaction with the atmosphere.

Concept Check

The general scalar transport equation is:

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla\cdot(\rho\mathbf{u}\phi) = \nabla\cdot(\Gamma\nabla\phi) + S_\phi$$

Consider the transport of dissolved salt (as ϕ) in an estuary. For each term in the equation, provide a brief physical interpretation relevant to this specific scenario. For instance, what could S_ϕ represent? What processes contribute to the advective flux and the diffusive flux of salt?

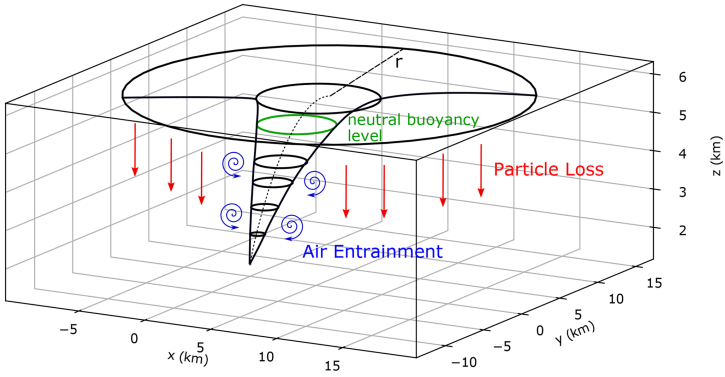


Figure 5.13. Schematic of a volcanic eruption plume, illustrating key processes relevant to control volume analysis. A segment of the rising, widening plume can be considered a control volume. The evolution of mass, momentum, and energy within this volume is governed by vertical fluxes (inflow/outflow, not explicitly shown but implied by upward motion), lateral fluxes due to "Air Entrainment" (a mass source, momentum and energy sink/source depending on relative properties) and "Particle Loss" (a mass and momentum sink), and internal buoyancy forces. The plume spreads laterally at the "neutral buoyancy level." The radius r and height z are key geometric parameters.

5.4 Governing Equations: Special Cases of the Transport Equation

The general scalar transport equation (Equation 5.20) provides a unified framework for describing many different physical processes. By selecting appropriate choices for the scalar quantity ϕ, the diffusion coefficient Γ, and the source term S_ϕ, we can derive specific governing equations for various phenomena.

5.4.1 The Mass Conservation Equation (Continuity Equation)

One of the most fundamental conservation principles is the conservation of mass. We can obtain the mathematical expression for mass conservation by applying the general scalar transport equation.

Consider the scalar quantity ϕ to be simply 1 (representing mass per unit mass). If $\phi = 1$, then $\rho\phi = \rho$ (which is mass per unit volume, i.e., density). For pure mass conservation, there is no "diffusion of mass" in the Fickian sense (mass doesn't diffuse relative to itself due to a "mass gradient" in the same way heat or chemical species do), so we can set the diffusion coefficient $\Gamma = 0$. Furthermore, in the absence of processes that convert energy into mass (which are not relevant for typical fluid dynamics or geoscience problems), the source term for mass S_ϕ (or S_ρ in this case) is zero.

Substituting $\phi = 1$, $\Gamma = 0$, and $S_\phi = 0$ into the general scalar transport equation:

$$\boxed{\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0} \tag{5.21}$$

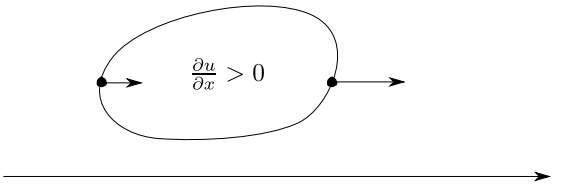


Figure 5.14. Physical interpretation of velocity divergence $\nabla \cdot \mathbf{u}$ and its effect on fluid density within a parcel. (Left) Convergent flow ($\nabla \cdot \mathbf{u} < 0$) compresses the fluid parcel, increasing its density. (Right) Divergent flow ($\nabla \cdot \mathbf{u} > 0$) expands the fluid parcel, decreasing its density. The image depicts a fluid parcel (the ovoid shape) moving along streamlines (horizontal axis not explicitly shown but implied by the arrows representing velocity vectors at the parcel's entry and exit points).

This is the **continuity equation**, also known as the mass conservation equation, in its Eulerian (conservative) form. It states that the local rate of change of density $\frac{\partial \rho}{\partial t}$ at a point is balanced by the net rate at which mass is advected out of or into that point, represented by the divergence of the mass flux $\rho \mathbf{u}$ (i.e., $\nabla \cdot (\rho \mathbf{u})$).

It is crucial to note that these simplifications ($\Gamma = 0$ and $S_\phi = 0$) apply only when we are considering the total mass of the system. If we were to write a conservation equation for a single chemical species *within* a mixture (e.g., dissolved salt in water), that species *would* have a diffusive flux relative to the bulk fluid ($\Gamma > 0$) and could have a source/sink term if it participates in chemical reactions ($S_\phi \neq 0$). However, for the total mass, these terms are zero. The statement "mass is not created or destroyed" is a fundamental principle in physics known as the Law of Conservation of Mass.

Lagrangian Form of the Continuity Equation and the Role of Divergence

We can gain further insight into the meaning of the continuity equation by transforming it into a Lagrangian perspective, which describes the rate of change of density *following a moving fluid parcel*. Starting with the Eulerian form (Equation 5.21) and expanding the divergence term using the vector identity $\nabla \cdot (\rho \mathbf{u}) = \mathbf{u} \cdot \nabla \rho + \rho (\nabla \cdot \mathbf{u})$, we get:

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho (\nabla \cdot \mathbf{u}) = 0$$

The first two terms, $\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho$, constitute the material derivative of density, $\frac{D\rho}{Dt}$ (see Equation 5.4). Therefore, the continuity equation can be rewritten in Lagrangian form as:

$$\frac{D\rho}{Dt} = -\rho (\nabla \cdot \mathbf{u}) \tag{5.22}$$

This equation provides a clear physical interpretation:

- The term $\nabla \cdot \mathbf{u}$ is the **divergence of the velocity field**. It measures the rate at which the fluid is expanding (if $\nabla \cdot \mathbf{u} > 0$) or contracting/compressing (if $\nabla \cdot \mathbf{u} < 0$) per unit volume at a point.
- Equation 5.22 states that the density of a moving fluid parcel decreases ($\frac{D\rho}{Dt} < 0$) if the flow is divergent (expanding, $\nabla \cdot \mathbf{u} > 0$), because the same mass now occupies a larger volume.

- Conversely, the density of a fluid parcel increases ($\frac{D\rho}{Dt} > 0$) if the flow is convergent (compressing, $\nabla \cdot \mathbf{u} < 0$), as the same mass is squeezed into a smaller volume.

Figure 5.14 illustrates how the divergence or convergence of the velocity field directly impacts the volume, and thus the density, of a deforming fluid parcel.

We can visualize these principles with a common example: a soap bubble drifting and deforming in the air (Figure 5.15). The thin film of the bubble encloses a specific parcel of air, which for our purposes can be considered a closed system in terms of mass (i.e., the mass of air inside the bubble remains constant, assuming no leaks). As this bubble—our Lagrangian air parcel—moves through regions where the external air flow might cause it to expand or compress, its volume changes. If the surrounding flow field is divergent ($\nabla \cdot \mathbf{u} > 0$) locally around the bubble, the bubble will tend to expand. Since the mass of air inside is fixed, this increase in volume directly leads to a decrease in the air's density within the bubble, precisely as described by Equation 5.22 ($\frac{D\rho}{Dt} < 0$). Conversely, if the bubble is advected into a region of convergent flow ($\nabla \cdot \mathbf{u} < 0$), it will be squeezed, its volume will decrease, and the density of the air parcel inside will increase ($\frac{D\rho}{Dt} > 0$). This is a direct manifestation of Equation 5.22: the change in density of the air parcel (the air inside the bubble) is directly related to the divergence or convergence of the flow field that deforms its volume.

Figure 5.15. A soap bubble moving and deforming in the air. The bubble contains a fixed mass of air. If the bubble's volume changes due to interactions with the surrounding air flow (e.g., due to velocity divergence or convergence of the external flow field, or pressure changes), the density of the air inside the bubble will change according to the principle of mass conservation as expressed by the Lagrangian form of the continuity equation ($\frac{D\rho}{Dt} = -\rho(\nabla \cdot \mathbf{u})$).

We will now pay close attention to the terminology, distinguishing between an *incompressible fluid* and an *incompressible flow*. In common usage, compressibility is seen as a physical property of a material, but the term "incompressible" is more precisely associated with a flow regime. Many texts do not distinguish between these concepts because the resulting mathematical conditions are often similar, but the physical distinction is important for a deeper understanding.

An **incompressible fluid** is a fluid whose density is assumed to be a constant value, unaffected by changes in pressure or temperature:

$$\rho(\mathbf{x}, t) = \text{const.} \tag{5.23}$$

This is a strong assumption about the material itself.

A **flow** is said to be **incompressible** (or isochoric) when the density of each fluid parcel remains constant as it moves with the flow. This is equivalent to stating that the material derivative of density vanishes:

$$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho = 0 \tag{5.24}$$

This distinction is crucial. A compressible fluid like air can give rise to an incompressible flow if its velocity is much smaller than the speed of sound (i.e., Mach number $Ma < 0.3$), as the pressure variations are too small to cause significant density changes.

By substituting the incompressible flow condition (Equation 5.24) into the expanded form of the continuity equation, we can see its main consequence. Recall the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

Using the vector identity $\nabla \cdot (\rho \mathbf{u}) = \mathbf{u} \cdot \nabla \rho + \rho(\nabla \cdot \mathbf{u})$, this becomes:

$$\underbrace{\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho}_{\text{This is } D\rho/Dt} + \rho(\nabla \cdot \mathbf{u}) = 0$$

Applying the incompressible flow condition, $\frac{D\rho}{Dt} = 0$, the equation simplifies dramatically to:

$$\rho(\nabla \cdot \mathbf{u}) = 0$$

For a fluid with non-zero density ($\rho \neq 0$), this implies that the velocity field must be divergence-free:

$$\nabla \cdot \mathbf{u} = 0 \tag{5.25}$$

This is the most common mathematical form of the continuity equation for an incompressible flow. It is a kinematic constraint on the velocity field, stating that the volume of any fluid parcel is conserved as it moves.

Crucially, the incompressible flow condition ($\frac{D\rho}{Dt} = 0$) does not require the density to be uniform in space. A flow can be incompressible but also **heterogeneous** ($\nabla \rho \neq 0$). A classic geoscience example is a river carrying suspended sediment. If we consider both the water and the sediment particles to be incompressible, any parcel of the mixture maintains its density as it moves, so the flow satisfies $\nabla \cdot \mathbf{u} = 0$. However, due to settling, the concentration of sediment, and thus the bulk density of the mixture ρ_{mixture}, is higher near the riverbed than near the surface. In this case, the density field is not homogeneous, even though the flow is incompressible. The Earth's oceans, where temperature and salinity variations create a density-stratified fluid, provide another prominent example of an incompressible, heterogeneous flow.

Finally, let us consider the stricter case of an **incompressible fluid**, where ρ is a true constant everywhere (Equation 5.23). This trivially implies that its partial derivatives in time and space are zero: $\frac{\partial \rho}{\partial t} = 0$ and $\nabla \rho = 0$. In this situation, the incompressible flow condition $\frac{D\rho}{Dt} = 0$ is automatically satisfied. The continuity equation, $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$, with

ρ being a constant, simplifies to $\rho(\nabla \cdot \mathbf{u}) = 0$, which once again yields the divergence-free condition $\nabla \cdot \mathbf{u} = 0$.

In summary, while a constant-density fluid always leads to an incompressible flow, the reverse is not true. An incompressible flow does not necessarily imply a constant and homogeneous density field.

Concept Check

The Lagrangian form of the continuity equation is $\frac{D\rho}{Dt} = -\rho(\nabla \cdot \mathbf{u})$.

1. If a parcel of magma is rising in a volcanic conduit that narrows upwards, would you expect $\nabla \cdot \mathbf{u}$ within that parcel to be generally positive or negative (assuming the magma is forced through the narrowing)? How would this affect the density of that rising magma parcel if it were compressible?
2. If we consider the magma incompressible, what does the condition $\nabla \cdot \mathbf{u} = 0$ imply about the volume of a magma parcel as it rises?

The principle of mass conservation in a steady, incompressible flow has direct and observable consequences in natural systems. Consider, for example, an active lava flow advancing across the landscape, as shown in Figure 5.16. Within such a flow, lava often becomes channelized. If we observe a segment of a well-defined channel where the flow can be considered approximately steady (i.e., the flow rate at any given point in the channel does not change significantly over the observation time) and the lava is behaving as an incompressible fluid (its density ρ is constant), then the principle of mass conservation dictates a clear relationship between the channel's cross-sectional area A and the average flow velocity u.

Specifically, the volumetric flow rate $Q = A \cdot u$ must remain constant along the channel segment if no lava is being added or lost from the sides. Therefore, if the channel narrows (decreasing A), the lava must accelerate (increasing u) to maintain the same volumetric flow rate. Conversely, if the channel widens (increasing A), the lava will decelerate (decreasing u). The channel visible on the right side of Figure 5.16 appears to exhibit such variations in width, which would directly translate to changes in flow velocity if the discharge is steady. This inverse relationship between cross-sectional area and velocity is a direct consequence of conserving mass in a steady, incompressible flow. While real lava flows are complex (temperature-dependent viscosity, potential for outgassing affecting density, unsteady supply rates), this simplified application of mass conservation provides valuable first-order insights into their dynamics.

5.4.2 The Heat (Diffusion) Equation

Another crucial conservation law is the conservation of energy. When applied to thermal processes where heat transfer by conduction is dominant, it leads to the **Heat Equation**, also commonly referred to as the **Diffusion Equation** due to its mathematical similarity to other diffusive processes (like chemical diffusion).

We can derive the Heat Equation as a special case of the general scalar transport equation:

$$\boxed{\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{u}\phi) = \nabla \cdot (\Gamma \nabla \phi) + S_\phi} \tag{5.26}$$

To do this, we need to define what our scalar quantity ϕ, the diffusivity Γ, and the source term S_ϕ represent in the context of thermal energy.

Figure 5.16. Active lava channels from an eruption in Kīlauea's lower East Rift Zone, Hawai'i (May 23, 2018). Note the variations in channel width, particularly in the prominent channel on the right. Assuming steady, incompressible flow, conservation of mass implies that lava velocity must increase where the channel narrows and decrease where it widens to maintain a constant volumetric flow rate. The image also shows overflows and haze from sulfur dioxide gas. (Image courtesy of J. Ozbolt, Hilo Civil Air Patrol, via USGS.)

1. **The Scalar Quantity ϕ:** For heat transfer, the conserved quantity related to thermal energy per unit mass is often taken as the **internal energy** e or, under common simplifying assumptions, it can be related directly to **temperature** T. If we consider a material with constant density ρ and constant specific heat capacity at constant volume c_v (or constant pressure c_p, often approximated as equal for solids and liquids), then the change in thermal energy per unit volume is $\rho c_v dT$. We can let $\phi = c_v T$ (so $\rho\phi = \rho c_v T$ is volumetric heat content relative to a reference). For simplicity in deriving the common form, many texts effectively set $\phi = T$ and absorb ρc_v into other coefficients or assume they are unity for non-dimensionalization. Let's consider $\phi = T$ for now, and the term ρc_p will be grouped with the diffusivity.
2. **Velocity u:** For pure heat conduction in a stationary medium (or a solid), there is no bulk advective transport of heat by fluid motion, so we set the velocity $\mathbf{u} = \mathbf{0}$.
3. **The Diffusion Coefficient Γ:** For heat conduction, the "diffusion coefficient" for temperature is related to the material's ability to conduct heat. This is the **thermal diffusivity**, denoted by κ (kappa). As mentioned in Section 5.1.1, $\kappa = k_c/(\rho c_p)$, where k_c is the thermal conductivity. So, we set $\Gamma = \kappa$. We will assume κ is constant for simplicity here.
4. **The Source Term S_ϕ:** This term, S_T, represents the volumetric rate of heat generation (or consumption) within the material, for example, due to radioactive decay within rocks, exothermic chemical reactions, or viscous dissipation (though the latter is often linked to the momentum equation).

Substituting these into the general transport equation (Equation 5.26), and assuming constant ρ and c_p (so they can be divided out or absorbed into κ and S_T if we define $\phi = T$ and S_T as rate of temperature change due to sources): If we set $\rho = 1$ and $\phi = T$ for simplicity in the unsteady and diffusive terms (effectively working with temperature directly, and S_T becomes a source term for temperature):

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{0} \cdot T) = \nabla \cdot (\kappa \nabla T) + S_T'$$

where S_T' would be $S_T/(\rho c_p)$. The advection term $\nabla \cdot (\mathbf{0} \cdot T)$ vanishes. If we assume κ is spatially constant, $\nabla \cdot (\kappa \nabla T) = \kappa \nabla^2 T$. This simplifies to the **Heat Equation with a source term**:

$$\boxed{\frac{\partial T}{\partial t} = \kappa \nabla^2 T + S_T'} \tag{5.27}$$

In one spatial dimension x, and if there are no internal heat sources ($S_T' = 0$), this reduces to the familiar 1D Heat Equation:

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \tag{5.28}$$

This parabolic PDE describes how temperature at a point changes over time due to the net conduction of heat to or from its surroundings, driven by the local curvature of the temperature profile. It is a fundamental equation for modelling a vast array of thermal processes in the Earth sciences, from the cooling of igneous bodies to the propagation of seasonal temperature waves into the ground.

Steady-State Solutions and the Emergence of ODEs from PDEs It is also interesting to note that sometimes an Ordinary Differential Equation (ODE) can emerge as a

description of a **steady-state** (or equilibrium) condition of a system that is more generally described by a Partial Differential Equation (PDE) like the Heat Equation.

A "steady state" implies that the temperature at any given point in space is no longer changing with time. Mathematically, this means the time derivative term in the Heat Equation (Equation 5.27 or 5.28) is zero: $\frac{\partial T}{\partial t} = 0$.

If we consider the 1D Heat Equation with a source term $S'_T(x)$ (representing volumetric heat production rate per unit ρc_p):

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} + S'_T(x)$$

Setting $\frac{\partial T}{\partial t} = 0$ for steady-state, the equation simplifies to:

$$0 = \kappa \frac{d^2 T}{dx^2} + S'_T(x)$$

Rearranging, and noting that T now only depends on x (since it's steady in time, $T(x)$), we have:

$$\kappa \frac{d^2 T}{dx^2} = -S'_T(x)$$

This is now a second-order **Ordinary Differential Equation** for the steady-state temperature profile $T(x)$, where the independent variable is the spatial coordinate x.

For instance, if there are no internal heat sources ($S'_T(x) = 0$), the ODE becomes $\kappa \frac{d^2 T}{dx^2} = 0$. If this describes a rock layer between $x = 0$ and $x = L$ with fixed boundary temperatures $T(0) = T_A$ and $T(L) = T_B$, the solution to this simple ODE is a linear temperature profile:

$$T(x) = T_A + (T_B - T_A)\frac{x}{L}$$

This linear profile represents the final, time-invariant temperature distribution that the system reaches after a sufficiently long time, as governed by the original time-dependent PDE and the applied boundary conditions.

This relationship highlights how ODEs can describe specific limiting cases or simplified aspects (like the steady-state spatial distribution) of more complex, time-evolving systems governed by PDEs. Understanding this connection can be very useful in analyzing and interpreting both analytical and numerical solutions to PDEs.

Having derived the transient Heat Equation, our next challenge is to solve it. As with many PDEs describing real-world phenomena, analytical solutions are often limited to simplified geometries and conditions. Therefore, in the next chapter, we will focus on developing a **numerical method** to approximate its solution. This will involve two key new steps beyond what we have covered for ODEs:

1. Devising a **finite difference approximation for the second spatial derivative**, $\frac{\partial^2 T}{\partial x^2}$, as the Heat Equation involves this term (we have previously developed schemes only for first derivatives).
2. Understanding and implementing **boundary conditions**, which are essential for defining a unique solution to a PDE over a finite spatial domain.

We will then combine these with a time-stepping scheme to simulate how temperature diffuses over space and time.

Chapter Summary: PDEs, Conservation, and Fundamental Equations

This chapter served as an essential bridge from the world of Ordinary Differential Equations (ODEs) to the more complex domain of Partial Differential Equations (PDEs), which are indispensable for describing the spatially and temporally varying systems prevalent in the Earth sciences. We laid the conceptual groundwork for understanding how these equations arise from fundamental physical principles.

Key takeaways from this chapter include:

- **Transition from ODEs to PDEs:**
  - We highlighted that while ODEs describe changes with respect to a single independent variable (typically time), PDEs are necessary when quantities of interest (e.g., temperature, concentration, pressure) vary simultaneously in space and time.
  - Numerous geoscience examples, such as heat flow in dikes, soil water movement, and ash dispersion, were presented to motivate the need for PDEs.
- **Physical Foundations for PDEs:**
  - The **continuum hypothesis** was introduced as a foundational assumption, allowing us to treat geological materials like fluids or solids as continuous media with well-defined properties at every point.
  - The distinction between **Eulerian** (fixed-point observation) and **Lagrangian** (following-particle observation) frames of reference was explained, with the material derivative $\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi$ providing the link between them.
- **Derivation of the General Scalar Transport Equation:**
  - The core of the chapter focused on deriving the general scalar transport equation from a **conservation law** applied to a fixed control volume (Eulerian approach).
  - The balance equation (Accumulation = Net Influx + Net Generation) was broken down into its constituent parts:
    - **Accumulation term**: $\frac{\partial(\rho\phi)}{\partial t}$
    - **Generation term**: S_ϕ
    - **Net Influx term**: $-\nabla \cdot \mathbf{J}_\phi$, where the total flux $\mathbf{J}_\phi$ was shown to comprise:
      - **Advective (Convective) Flux**: $\rho \mathbf{u} \phi$ (transport due to bulk motion).
      - **Diffusive Flux**: $-\Gamma \nabla \phi$ (transport down gradients, e.g., Fick's Law).
  - This culminated in the general scalar transport equation: $\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{u} \phi) = \nabla \cdot (\Gamma \nabla \phi) + S_\phi$.
  - The distinct roles of advection (stirring) and diffusion (homogenization) in mixing processes were also discussed.
- **Special Cases - Fundamental Governing Equations:**
  - **Continuity Equation (Mass Conservation):** Derived by setting $\phi = 1$, $\Gamma = 0$, $S_\phi = 0$ in the general transport equation, yielding $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$. Its Lagrangian form, $\frac{D\rho}{Dt} = -\rho(\nabla \cdot \mathbf{u})$, and the concept of velocity divergence were explored, along with the incompressibility condition $\nabla \cdot \mathbf{u} = 0$.

- **Heat (Diffusion) Equation:** Derived by setting $\phi = T$ (temperature), $\mathbf{u} = \mathbf{0}$ (no advection), $\Gamma = \kappa$ (thermal diffusivity), and considering a source term S_T', leading to $\frac{\partial T}{\partial t} = \kappa \nabla^2 T + S_T'$.

This chapter has established how fundamental conservation principles are mathematically formulated into PDEs that govern a wide array of Earth science processes. Understanding the origin and meaning of these equations is the first critical step before attempting their numerical solution, which will be the focus of subsequent chapters for specific PDEs like the Heat Equation and the Advection Equation.

Chapter 5 Exercises

Test your understanding of the physical principles, conservation laws, and fundamental equations discussed in this chapter.

E5.1: Magma Flow in a Volcanic Conduit - Applying Mass Conservation

Consider a steady, incompressible flow of magma up a volcanic conduit. The conduit has a circular cross-section, but its radius R can vary with vertical position z. Let $u(z)$ be the average vertical velocity of the magma at a given depth z.

Tasks:

1. **Mass Flux**: Write down an expression for the mass flux (mass flowing per unit time) through a cross-section of the conduit at depth z, assuming the magma has a constant density ρ. (Hint: Mass flux = density × volumetric flow rate. Volumetric flow rate = area × average velocity.)

2. **Conservation of Mass (Steady State)**: For a steady flow, the mass flux must be constant along the conduit (i.e., what flows in at one depth must flow out at another, assuming no addition or loss of magma along the conduit walls). Using this principle, relate the average magma velocity u_1 at a location where the conduit radius is R_1 to the average magma velocity u_2 at another location where the conduit radius is R_2.

3. **Velocity Change**: If the conduit narrows (i.e., $R_2 < R_1$), how does u_2 compare to u_1? Explain this physically. What if the conduit widens?

4. **Incompressibility and Divergence**: Recall the continuity equation for an incompressible fluid ($\nabla \cdot \mathbf{u} = 0$). In our 1D conduit flow, if we primarily consider the vertical velocity component $u_z(z)$ (and assume radial/tangential components are negligible for the average flow), how does $\frac{du_z}{dz}$ relate to $\nabla \cdot \mathbf{u}$? If the velocity $u_z(z)$ increases with z (e.g., as magma rises and decompresses, though here we assumed incompressible for simplicity, or if the conduit narrows), what does this imply about the other velocity components or the strict 1D assumption if the flow is truly incompressible? (Hint: For a strictly 1D flow $\mathbf{u} = (0, 0, u_z(z))$, $\nabla \cdot \mathbf{u} = \frac{\partial u_z}{\partial z}$.)

This exercise helps connect the abstract concept of mass conservation to observable velocity changes in a common geological setting.

E5.2: Towards the Momentum Conservation Equation (Conceptual)

The general scalar transport equation (Equation 5.20) is:

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{u}\phi) = \nabla \cdot (\Gamma \nabla \phi) + S_\phi$$

This equation was derived for a generic scalar quantity ϕ (expressed per unit mass). Momentum, however, is a vector quantity. The conservation of linear momentum is essentially Newton's Second Law: the rate of change of momentum of a fluid parcel equals the sum of forces acting on it.

Let's consider the x-component of momentum. The momentum per unit mass in the x-direction is simply the x-component of velocity, u_x. So, we can set $\phi = u_x$.

Tasks (Conceptual Discussion - No full derivation required):

1. What is $\rho\phi$ in this case? What physical quantity does ρu_x represent?
2. The Source Term S_ϕ (or S_{u_x}): Newton's Second Law states $\sum F = ma = m\frac{Du}{Dt}$. For a control volume, the "source" term in the momentum equation corresponds to the sum of forces acting on the fluid within that volume. What types of forces typically act on a fluid element in a geological context (e.g., magma flow, groundwater flow, atmospheric flow)? (Hint: Think about pressure, gravity, and viscous stresses.)
3. The Advection Term $\nabla \cdot (\rho \mathbf{u} u_x)$: This term describes the advection of x-momentum by the flow field $\mathbf{u}$. If we expand part of this for 1D flow where $\mathbf{u} = (u_x, 0, 0)$, one of the terms might look like $u_x \frac{\partial u_x}{\partial x}$ (after some manipulation involving the continuity equation for constant density). Why is a term like $u_x \frac{\partial u_x}{\partial x}$ (or more generally $\mathbf{u} \cdot \nabla u_x$) considered a **nonlinear term**? How does this differ from the advection term $u\frac{\partial \phi}{\partial x}$ in the linear advection equation we will study later (where ϕ is a passive scalar and u is a specified constant velocity)?
4. The Diffusion Term $\nabla \cdot (\Gamma \nabla u_x)$: For momentum, the "diffusion coefficient" Γ is related to the fluid's viscosity μ (dynamic viscosity). The term $\nabla \cdot (\mu \nabla u_x)$ (or similar forms involving stress tensors) represents the net rate of momentum transfer due to viscous forces (friction within the fluid). How does viscosity act to smooth out velocity differences, similar to how thermal diffusivity smooths out temperature differences?

This exercise is intended to make you think about how the general transport equation framework can be adapted to describe vector quantities like momentum, and to identify key physical terms and potential nonlinearities that arise in more complex fluid dynamics equations like the Navier-Stokes equations. A full derivation of the momentum equation is more involved as it requires careful consideration of surface and body forces and the stress tensor.

Chapter 6

Numerical Solution of the 1D Heat (Diffusion) Equation

In the previous chapter, we established the physical and mathematical foundations for many Partial Differential Equations (PDEs) encountered in the Earth sciences and based on conservation laws. We saw how the Heat (or Diffusion) Equation arises from the principle of energy conservation. For a one-dimensional domain, assuming constant thermal diffusivity κ and no internal heat sources, this equation is:

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \tag{6.1}$$

where $T(x,t)$ is the temperature as a function of position x and time t.

This chapter is dedicated to developing a numerical method to solve this fundamental PDE. The process will require us to approximate both the time derivative $\frac{\partial T}{\partial t}$ and, crucially, the second spatial derivative $\frac{\partial^2 T}{\partial x^2}$. This involves discretizing both the time and spatial domains. Furthermore, a key aspect of solving PDEs on finite domains, which distinguishes them from many initial value ODE problems, is the necessity of specifying **boundary conditions** to ensure a unique and physically relevant solution.

6.1 The Crucial Role of Boundary Conditions in PDE Problems

When we solve a PDE like the Heat Equation over a specific region in space (e.g., the thickness of a rock layer or a computational domain representing a part of a geoscience system), the behavior of the solution $T(x,t)$ within that region is profoundly influenced by what happens at its edges or boundaries. These interactions with the "outside world" or adjacent parts of the system are mathematically expressed as **boundary conditions (BCs)**.

- **Physical Meaning:** Boundary conditions represent the physical constraints or processes occurring at the extremities of the domain. For example, one end of a rod might be held at a constant temperature, while the other might be insulated (no heat flow), or lose heat to the environment via convection.

- **Mathematical Necessity:** For a PDE like the Heat Equation (which is second-order in space), we typically need to specify two boundary conditions in space (e.g., one at each end of a 1D domain) in addition to an initial condition in time (the temperature distribution at $t = 0$) to ensure a unique solution. Without BCs, an infinite number of mathematical solutions could satisfy the PDE itself.
- **Impact on Solution:** The choice of boundary conditions dramatically affects the resulting temperature distribution and its evolution over time. Incorrect or poorly chosen BCs can lead to physically unrealistic or numerically unstable solutions.

Consider, for instance, modeling the cooling of a vertical magmatic dike. If we exploit symmetry and model only half of the dike, the plane of symmetry would have a zero heat flux condition. The outer contact with the host rock might be represented by a fixed temperature or a more complex heat transfer condition. These choices are critical.

Common Types of Boundary Conditions

We generally encounter three main types of boundary conditions for problems like heat conduction:

1. **Dirichlet Condition (Type I or Fixed Value):** The value of the dependent variable (e.g., temperature T) is explicitly specified at the boundary.
    - Example for a 1D domain $x \in [0, L]$: $T(0, t) = T_{left}(t)$ and/or $T(L, t) = T_{right}(t)$.
    - *Geoscience Context:* The surface of a lava flow in contact with cold air (fixed surface temperature), or the temperature at a known depth in a geothermal system maintained by a large heat reservoir.
2. **Neumann Condition (Type II or Fixed Gradient/Flux):** The spatial derivative (gradient) of the dependent variable is specified at the boundary. For heat conduction, this is related to specifying the heat flux via Fourier's Law ($q = -k_c \frac{\partial T}{\partial x}$, where k_c is thermal conductivity).
    - Example: $\frac{\partial T}{\partial x}|_{x=0} = g_0(t)$. A common special case is the **zero-flux** or **insulated boundary** condition: $\frac{\partial T}{\partial x}|_{x_{boundary}} = 0$.
    - *Geoscience Context:* An insulated boundary (e.g., the base of a thick lithospheric plate), a symmetry plane in a thermal model, or a specified geothermal heat flux.
3. **Robin Condition (Type III or Mixed/Convective):** A linear combination of the variable's value and its derivative is specified at the boundary, often representing convective heat transfer: $-k_c \frac{\partial T}{\partial x}|_{x_{boundary}} = h_{coeff}(T_{boundary} - T_\infty)$. *Geoscience Context:* The surface of a lava flow cooling by convection to the atmosphere.

Choosing appropriate and physically realistic boundary conditions is a critical step. In our numerical examples, we will primarily explore Dirichlet and Neumann conditions.

Concept Check

Consider modeling the temperature evolution within a thick, laterally extensive lava flow cooling primarily by losing heat from its top surface to the atmosphere and from its bottom surface to the underlying ground.
What type of boundary condition (Dirichlet, Neumann, or Robin) might be most appropriate for the top surface if we assume a strong wind keeps the air temperature constant and promotes convective cooling? Briefly justify your choice.

6.2 Numerical Discretization of the 1D Heat Equation

To solve the 1D Heat Equation (Equation 6.1) numerically, we discretize both the spatial and time domains and approximate the partial derivatives using finite differences.

Spatial and Temporal Grids

As introduced in Chapter 4, we discretize the spatial domain, say $x \in [0, L]$, into N_x equally spaced grid points (or nodes) $x_i = i\Delta x$, for $i = 0, 1, \ldots, N_x - 1$. The spatial step size is $\Delta x = L/(N_x - 1)$.

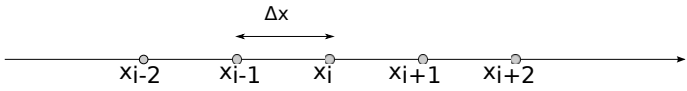


Figure 6.1. A one-dimensional spatial grid with nodes x_i and uniform spacing Δx.

Similarly, we discretize time into steps $t^n = n\Delta t$ (see Figure 6.2). The numerical solution for temperature at grid point x_i and time t^n is denoted by $T_i^n \approx T(x_i, t^n)$. Our goal is to advance the solution from time level n to $n+1$.

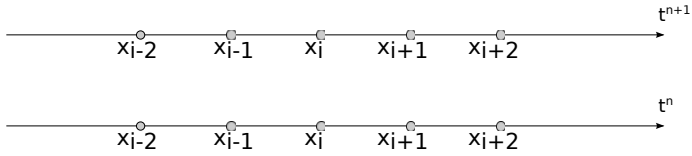


Figure 6.2. Space-time discretization grid: T_i^n is the solution at node x_i and time t^n.

Concept Check

In our discretized space-time grid, T_i^n represents the temperature at spatial node x_i and time level t^n. If you are implementing a numerical scheme and need to refer to the temperature at the same spatial location x_i but at the previous time step, what notation would you use? If you need the temperature at the spatial node to the right of x_i (i.e., x_{i+1}) at the current time step t^n, what notation would that be?

Finite Difference Approximations for the Derivatives

Time Derivative $\frac{\partial T}{\partial t}$. We use the **Forward Difference** in time, evaluated at (x_i, t^n):

$$\left.\frac{\partial T}{\partial t}\right|_i^n \approx \frac{T_i^{n+1} - T_i^n}{\Delta t} \quad \text{(Truncation Error } \mathcal{O}(\Delta t)) \tag{6.2}$$

Second Spatial Derivative $\frac{\partial^2 T}{\partial x^2}$. The Heat Equation requires an approximation for the second spatial derivative. We will use a **Central Difference** scheme. To derive this, we again turn to Taylor series expansions. Consider the function $T(x,t)$. We are interested in its spatial variation at a fixed time t. Let $T_i = T(x_i, t)$, $T_{i+1} = T(x_i + \Delta x, t)$, and $T_{i-1} = T(x_i - \Delta x, t)$. The Taylor expansions around (x_i, t) for T_{i+1} and T_{i-1} are:

$$T_{i+1} = T_i + \Delta x \left.\frac{\partial T}{\partial x}\right|_i + \frac{(\Delta x)^2}{2!}\left.\frac{\partial^2 T}{\partial x^2}\right|_i + \frac{(\Delta x)^3}{3!}\left.\frac{\partial^3 T}{\partial x^3}\right|_i + \mathcal{O}(\Delta x^4) \qquad (Eq.A)$$

$$T_{i-1} = T_i - \Delta x \left.\frac{\partial T}{\partial x}\right|_i + \frac{(-\Delta x)^2}{2!}\left.\frac{\partial^2 T}{\partial x^2}\right|_i + \frac{(-\Delta x)^3}{3!}\left.\frac{\partial^3 T}{\partial x^3}\right|_i + \mathcal{O}(\Delta x^4) \qquad (Eq.B)$$

Adding Equation A and Equation B:

$$\begin{aligned} T_{i+1} + T_{i-1} = (T_i + T_i) &+ (\Delta x - \Delta x)\left.\frac{\partial T}{\partial x}\right|_i \\ &+ \left(\frac{(\Delta x)^2}{2} + \frac{(-\Delta x)^2}{2}\right)\left.\frac{\partial^2 T}{\partial x^2}\right|_i \\ &+ \left(\frac{(\Delta x)^3}{6} + \frac{(-\Delta x)^3}{6}\right)\left.\frac{\partial^3 T}{\partial x^3}\right|_i + \mathcal{O}(\Delta x^4) \end{aligned}$$

The terms with odd powers of Δx (first and third derivatives) cancel out. This simplifies to:

$$T_{i+1} + T_{i-1} = 2T_i + (\Delta x)^2 \left.\frac{\partial^2 T}{\partial x^2}\right|_i + \mathcal{O}(\Delta x^4)$$

Rearranging to solve for the second derivative and neglecting terms of $\mathcal{O}(\Delta x^4)$ and higher (which constitute the truncation error), we get the central difference approximation for the second spatial derivative:

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i^n \approx \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} \tag{6.3}$$

The leading term of the truncation error for this approximation (i.e. the neglected term) is proportional to $(\Delta x)^2$ (specifically, it involves $\mathcal{O}(\Delta x^4)/(\Delta x)^2$), so this scheme is **second-order accurate** in space, $\mathcal{O}(\Delta x^2)$. Thirs approximation is evaluated at time level n for use in an explicit time-stepping scheme.

The Forward-Time Central-Space (FTCS) Scheme: Formulation

Having established the finite difference approximations for both the time derivative and the second spatial derivative, we can now substitute them into the 1D Heat Equation, $\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$. Using the forward difference in time (Equation 6.2) and the central difference in space for the second derivative (Equation 6.3), we get:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \kappa \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} \right) \tag{6.4}$$

Our objective is to determine the temperature T_i^{n+1} at the new time level t^{n+1} for each spatial node x_i, given the known temperatures T_j^n at the current time level t^n. Since all terms on the right-hand side of Equation 6.4 are evaluated at time level n, this scheme is **explicit**. We can solve for T_i^{n+1} by simple algebraic rearrangement:

$$T_i^{n+1} = T_i^n + \kappa \frac{\Delta t}{(\Delta x)^2} \left(T_{i+1}^n - 2T_i^n + T_{i-1}^n \right) \tag{6.5}$$

This is the update formula for the **Forward-Time Central-Space (FTCS)** scheme. It is often convenient to define a dimensionless parameter α (alpha), known as the numerical diffusion number or, in some contexts, the Courant number for diffusion problems:

$$\alpha = \kappa \frac{\Delta t}{(\Delta x)^2} \tag{6.6}$$

Using this parameter, the FTCS update formula takes the more compact form:

$$\boxed{T_i^{n+1} = T_i^n + \alpha \left(T_{i+1}^n - 2T_i^n + T_{i-1}^n \right)} \tag{6.7}$$

This equation is applied to all *interior* grid points, typically $i = 1, 2, \ldots, N_x - 2$ (if the grid nodes are indexed from 0 to $N_x - 1$). The temperatures at the boundary points, T_0^{n+1} and $T_{N_x-1}^{n+1}$, are determined by the specific boundary conditions of the problem.

Concept Check

The FTCS update formula for the 1D heat equation is given by Eq. 6.7.
If, at time t^n, the temperature at node i is $T_i^n = 50°\text{C}$, and its neighbors are $T_{i-1}^n = 60°\text{C}$ and $T_{i+1}^n = 60°\text{C}$, would you expect T_i^{n+1} to be higher or lower than T_i^n (assuming $\alpha > 0$)? Explain your reasoning based on the physics of heat diffusion and how it's represented in the formula.

Physical Interpretation of the FTCS Scheme. The FTCS update formula can be rewritten to provide insight into its physical meaning:

$$T_i^{n+1} = \alpha T_{i-1}^n + (1 - 2\alpha) T_i^n + \alpha T_{i+1}^n$$

This form clearly shows that the new temperature T_i^{n+1} at node i is a weighted average of the temperatures at node i itself and its immediate neighbors ($i-1$ and $i+1$) at the previous time step n.

- The terms αT_{i-1}^n and αT_{i+1}^n represent the influence of the neighboring temperatures. Heat effectively "flows" from hotter neighbors to cooler ones.

- The term $(1-2\alpha)T_i^n$ represents the contribution of the current temperature at node i.
- The combination $(T_{i+1}^n - 2T_i^n + T_{i-1}^n)$ is proportional to the discrete Laplacian, approximating the second order derivative of the temperature profile. If T_i^n is a local minimum (profile is concave up), this derivative is positive, leading to an increase in T_i^{n+1} (heating). If T_i^n is a local maximum (profile is concave down), this derivative is negative, leading to a decrease in T_i^{n+1} (cooling).

The scheme thus explicitly models the diffusion of heat, driven by temperature differences, which acts to smooth out the temperature profile over time.

6.3 Python Implementation and Results for the 1D Heat Equation (FTCS)

We will now implement the FTCS scheme in Python to simulate 1D heat diffusion. This will allow us to observe its behavior under different conditions, particularly concerning stability and the application of boundary conditions.

Scenario 1: Dirichlet Boundary Conditions

Problem Setup. We consider the following setup for our first numerical experiment:

- **Spatial Domain:** A 1D rod of length $L = 1.0$ meter. The domain is thus $0 \le x \le L$.
- **Grid Discretization:** $N_x = 51$ spatial grid points, leading to a spatial step $\Delta x = L/(N_x - 1) = 1.0/50 = 0.02$ m. The grid points are $x_i = i\Delta x$.
- **Material Property:** Thermal diffusivity $\kappa = 1.0 \times 10^{-6}$ m^2/s.
- **Time Discretization:** The time step Δt will be chosen carefully in relation to the stability of the scheme. We will simulate up to a final time t_{final}.
- **Initial Condition ($t = 0$):** A "hot pulse" is defined in the center of the domain:

$$T(x,0) = \begin{cases} 100^\circ\text{C} & \text{if } 0.4L \le x \le 0.6L \\ 20^\circ\text{C} & \text{elsewhere} \end{cases}$$

- **Boundary Conditions (for $t > 0$):** Fixed temperatures (Dirichlet conditions) are applied at both ends of the domain: $T(0,t) = 20^\circ$C and $T(L,t) = 20^\circ$C.

Python Implementation. Listing 6.1 provides the Python code to simulate this scenario. It initializes the parameters and the temperature field, then iterates through time, applying the FTCS update formula for interior points and enforcing the Dirichlet boundary conditions at each step. Temperature profiles are stored at regular intervals for later plotting.

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

# --- Parameters ---
lengthDomain = 1.0  # m
numXPoints = 51     # Number of spatial grid points
dxStep = lengthDomain / (numXPoints - 1)
xGrid = np.linspace(0, lengthDomain, numXPoints)
```

```
kappaDiffusivity = 1.0e-6  # Thermal diffusivity (m^2/s)

# Time parameters and Stability
# alpha = kappa * dt / dx^2. For stability, alpha <= 0.5
alphaTarget = 0.45 # Choose a value for alpha (e.g., 0.45 for
    stable, 0.55 for unstable)
# alphaTarget = 0.55 # Uncomment to test instability

dtStep = alphaTarget * dxStep**2 / kappaDiffusivity #
    Calculate dt based on target alpha
tFinal = 100000.0  # s (e.g., ~27 hours)
numTSteps = int(tFinal / dtStep)

print(f"Domain Length (L): {lengthDomain} m")
print(f"Number of X Points (numXPoints): {numXPoints}")
print(f"Spatial Step (dxStep): {dxStep:.4f} m")
print(f"Thermal Diffusivity (kappaDiffusivity): {
    kappaDiffusivity:.1e} m^2/s")
print(f"Target Alpha (alphaTarget): {alphaTarget:.2f}")
print(f"Time Step (dtStep): {dtStep:.2f} s")
print(f"Final Time (tFinal): {tFinal:.1f} s")
print(f"Number of Time Steps (numTSteps): {numTSteps}")
print(f"Calculated alpha = {kappaDiffusivity * dtStep / dxStep
    **2:.3f}") # Verify actual alpha

# --- Initial Condition ---
temperatureCurrent = np.ones(numXPoints) * 20.0  # Ambient
    temperature
pulseStartIndex = int(0.4 * lengthDomain / dxStep)
pulseEndIndex = int(0.6 * lengthDomain / dxStep)
temperatureCurrent[pulseStartIndex : pulseEndIndex + 1] =
    100.0

# Store temperature profiles for plotting
plotIntervalRatio = 0.1 # Plot approx 10 intermediate profiles
plotIntervalSteps = max(1, int(numTSteps * plotIntervalRatio))
timePointsToPlot = [0.0]
tempProfilesToPlot = [temperatureCurrent.copy()]

# --- Time-stepping loop (FTCS) ---
temperatureOld = temperatureCurrent.copy() # For T^n values

for n in range(1, numTSteps + 1):
    # Update interior points using values from T_old (time
    level n)
    for i in range(1, numXPoints - 1):
        temperatureCurrent[i] = temperatureOld[i] + \
            alphaTarget * (temperatureOld[i+1] - 2*
    temperatureOld[i] + temperatureOld[i-1])

    # Apply Dirichlet Boundary Conditions (temperature remains
     T^n+1 after update)
    temperatureCurrent[0] = 20.0   # Fixed temperature at x=0
    temperatureCurrent[numXPoints-1] = 20.0 # Fixed
    temperature at x=L
```

```
    # Update temperatureOld for the next iteration
    temperatureOld = temperatureCurrent.copy()

    # Store profile for plotting at specified intervals
    if n % plotIntervalSteps == 0 or n == numTSteps:
        timePointsToPlot.append(n * dtStep)
        tempProfilesToPlot.append(temperatureCurrent.copy())

# --- Plotting Results ---
plt.figure(figsize=(10, 7))
for iPlot in range(len(tempProfilesToPlot)):
    plt.plot(xGrid, tempProfilesToPlot[iPlot],
             label=f't = {timePointsToPlot[iPlot]:.0f} s')

plt.xlabel("Position x [m]")
plt.ylabel("Temperature [deg C]")
plt.title(f"1D Heat Diffusion (FTCS, Dirichlet, $\kappa$={
    kappaDiffusivity:.1e}, $\\alpha$={alphaTarget:.2f})")
plt.legend(loc='center left', bbox_to_anchor=(1.01, 0.5))
plt.grid(True)
plt.tight_layout(rect=[0, 0, 0.85, 1]) # Adjust layout for
    legend outside
plt.show()
```

Listing 6.1. Python FTCS scheme for 1D Heat Equation with Dirichlet BCs.

Executing the Python script in Listing 6.1 allows us to simulate the diffusion of the initial hot pulse. The behavior of this numerical solution is critically dependent on the choice of the dimensionless parameter $\alpha = \kappa\Delta t/(\Delta x)^2$. The solution for $\alpha = 0.45$ is illustrated in Figure 6.3

6.3.1 Stability Analysis of the FTCS Scheme for the Heat Equation

The Explicit Forward-Time Central-Space (FTCS) scheme, while straightforward to implement, is not unconditionally stable. It is **conditionally stable**, meaning that for the numerical solution to remain bounded and physically realistic (i.e., free from spurious, growing oscillations), a specific condition relating the time step Δt, the spatial step Δx, and the thermal diffusivity κ must be met.

This stability condition for the 1D FTCS heat equation is:

$$\alpha = \kappa\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2} \tag{6.8}$$

This implies that the time step Δt must be chosen such that:

$$\Delta t \leq \frac{(\Delta x)^2}{2\kappa} \tag{6.9}$$

If this condition is violated (i.e., if $\alpha > 0.5$), the numerical solution will become unstable, typically manifesting as oscillations that grow exponentially in amplitude with each time step, eventually overwhelming the true solution.

Intuitive Explanation for the Stability Condition. As discussed when interpreting the FTCS formula (Equation 6.7), the new temperature T_i^{n+1} can be expressed as a weighted average of temperatures at the previous time step:

$$T_i^{n+1} = \alpha T_{i-1}^n + (1-2\alpha)T_i^n + \alpha T_{i+1}^n$$

For this to represent a physically meaningful averaging process where no new local maxima or minima are spontaneously generated (a condition for avoiding oscillations in diffusion problems, related to the "maximum principle"), all coefficients acting on the T^n terms should ideally be non-negative.

- Since $\alpha = \kappa\Delta t/(\Delta x)^2$ is always positive (as $\kappa, \Delta t, \Delta x > 0$), the coefficients for T_{i-1}^n and T_{i+1}^n are positive.
- The critical coefficient is $(1-2\alpha)$, which multiplies T_i^n. If $1-2\alpha < 0$, meaning $\alpha > 0.5$, then the contribution of T_i^n to T_i^{n+1} becomes negative. If T_i^n is positive, this negative contribution can cause T_i^{n+1} to "overshoot" and become smaller than its neighbors, or even negative, initiating or amplifying oscillations.

Thus, the condition $1-2\alpha \geq 0$, or $\alpha \leq 0.5$, emerges as necessary to prevent this unphysical behavior and maintain stability. A rigorous proof typically involves Von Neumann stability analysis, which examines how different Fourier modes of the solution are amplified by the numerical scheme.

Concept Check

The stability condition for the FTCS scheme for the 1D heat equation is $\alpha = \kappa\frac{\Delta t}{(\Delta x)^2} \leq 0.5$. Suppose you have a stable simulation running with certain values of $\kappa, \Delta t$, and Δx.
If you decide to halve your spatial step size ($\Delta x \to \Delta x/2$) to get better spatial resolution, how must you adjust your time step Δt to ensure the simulation remains stable with the same α value (or to maintain $\alpha \leq 0.5$)?

Practical Implications of the Stability Constraint. The quadratic dependence on Δx in the stability condition ($\Delta t \propto (\Delta x)^2$) is a significant limitation of the FTCS scheme:

- To achieve higher spatial resolution (i.e., to make Δx smaller), Δt must be reduced by the square of that factor. For example, halving Δx requires reducing Δt by a factor of four to keep α constant (and stable).
- This can lead to a very large number of time steps and consequently long computation times, especially for problems requiring fine spatial grids or simulations over extended periods.

Visualizing Stable and Unstable FTCS Solutions

The Python script in Listing 6.1 can be used to demonstrate these stability characteristics by simply changing the value of ‘alphaTarget‘.

Stable Solution ($\alpha \leq 0.5$). If we set `alphaTarget = 0.45` in the script (satisfying the stability condition), the numerical solution evolves smoothly, showing the expected diffusion of the initial hot pulse. The temperature profiles at different time steps gradually flatten and spread, approaching the boundary temperatures, as illustrated in Figure 6.3.

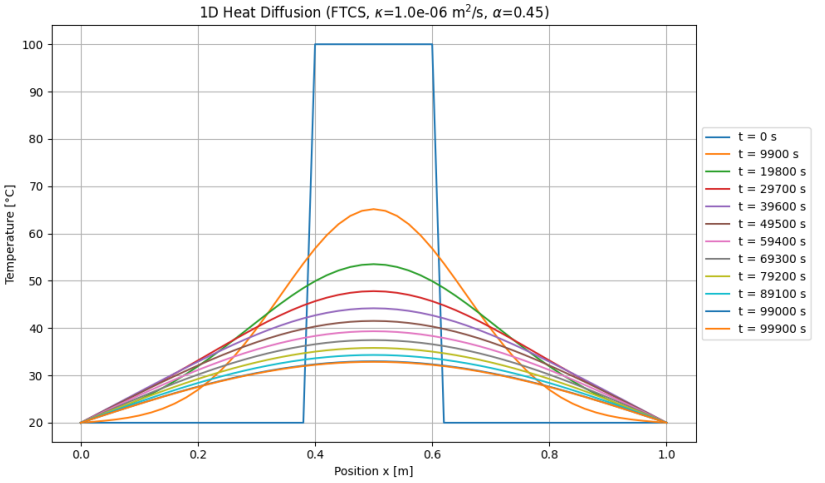


Figure 6.3. FTCS solution for the 1D Heat Equation with Dirichlet BCs ($T = 20°C$ at ends) and a stability parameter $\alpha = 0.45$. The initial hot pulse diffuses and cools over time. This output can be generated by running the code in Listing 6.1 with `alphaTarget = 0.45`.

Unstable Solution ($\alpha > 0.5$). If we modify the script to use `alphaTarget = 0.55` (thus violating the stability condition, as $0.55 > 0.5$), the behavior of the numerical solution changes dramatically. As shown in Figure 6.4, the solution quickly develops unphysical oscillations. These oscillations grow in amplitude with each time step, eventually leading to extremely large positive and negative temperature values that bear no resemblance to the true physical process.

These examples clearly illustrate the conditional stability of the FTCS scheme and the critical importance of choosing an appropriate time step Δt relative to the spatial step Δx and the thermal diffusivity κ.

6.3.2 Practical Guidance: Selecting Step Sizes for FTCS

The conditional stability of the explicit FTCS scheme, $\alpha = \kappa \Delta t/(\Delta x)^2 \leq 0.5$, underscores the critical importance of carefully selecting the spatial step size Δx and the temporal step size Δt. These choices impact not only the stability of the numerical solution but also its accuracy and the overall computational cost of the simulation.

Considerations for Choosing Spatial Step Size (Δx)

The selection of Δx is primarily driven by the need to **resolve the physical features** of the temperature field you are modeling:

- **Sharp Gradients and Fine Details:** If your problem involves regions with very sharp temperature gradients (e.g., near a chilled boundary, at the interface between different materials in more complex models, or around localized heat sources/sinks), Δx must be small enough to capture these variations adequately. A coarse grid (large

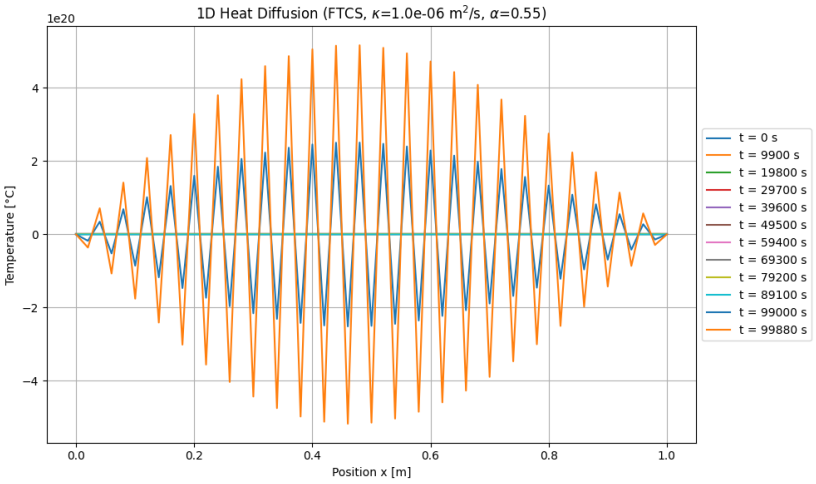


Figure 6.4. FTCS solution with the same setup as Figure 6.3, but with $\alpha = 0.55$ (unstable). The solution is dominated by rapidly growing numerical oscillations. This output can be generated by modifying `alphaTarget = 0.55` in the code from Listing 6.1.

Δx) might "smear out" these features or miss them entirely, leading to an inaccurate representation of the physics, even if the scheme is stable.

- **Length Scales of the Problem:** Consider the characteristic length scales of the phenomena you are studying. For example, if you are modeling heat penetration into a rock over a certain depth, Δx should be significantly smaller than that penetration depth to resolve the profile.
- **Computational Cost:** While a smaller Δx generally leads to a more accurate spatial representation, it also increases the total number of grid points ($N_x \approx L/\Delta x$). This directly increases the number of calculations per time step. Furthermore, as we have already partially seen and as we will better detail in the sext section, it dramatically constrains Δt.

A common practice is to perform a *grid convergence study* (or mesh refinement study): you run simulations with progressively smaller Δx values and observe how the solution changes. When further reductions in Δx lead to only minor changes in the solution features of interest, the grid is often considered sufficiently refined for spatial accuracy.

Considerations for Choosing Time Step Size (Δt)

Once Δx (and the material property κ) are determined, the choice of Δt for the FTCS scheme is heavily constrained by stability but also influenced by accuracy:

- **Stability First (The CFL Constraint for Diffusion):** The absolute priority is to satisfy the stability condition:

$$\Delta t \leq \frac{(\Delta x)^2}{2\kappa} \quad \text{or equivalently} \quad \alpha = \kappa \frac{\Delta t}{(\Delta x)^2} \leq 0.5$$

Violating this condition leads to unusable, divergent solutions, as demonstrated in Figure 6.4. In practice, it is wise to choose Δt such that α is safely below the limit, for example, $\alpha = 0.45$ or even 0.25, to provide a margin of safety and sometimes to reduce certain numerical artifacts. Setting α exactly to 0.5 can sometimes be borderline stable and may still exhibit minor issues in some cases due to round-off errors or the way boundary conditions are handled.

- **Impact of Δx on Δt:** The quadratic dependence on Δx is crucial. If you halve Δx to improve spatial resolution, you must reduce Δt by a factor of four to maintain the same α value (and thus stability). This rapid increase in the number of required time steps ($N_t = t_{final}/\Delta t$) is a major drawback of the explicit FTCS scheme for problems requiring high spatial resolution or long simulation times.
- **Temporal Accuracy:** The FTCS scheme has a local truncation error in time of $\mathcal{O}(\Delta t)$ (it is first-order accurate in time). This means that even if Δt satisfies the stability condition, a larger stable Δt (e.g., corresponding to $\alpha = 0.5$) will generally result in a larger temporal truncation error compared to a smaller stable Δt (e.g., corresponding to $\alpha = 0.25$). Therefore, if high temporal accuracy is critical, you might need to choose a Δt (and thus an α) significantly smaller than the stability limit, further increasing computational cost.
- **Balancing Act:** The selection of Δt involves balancing the strict stability requirement against the desired temporal accuracy and the acceptable computational runtime. There is no single "best" $\alpha < 0.5$; it often depends on the problem and what aspects of the solution are most important.

In summary, a typical workflow is:

1. Determine Δx based on the spatial scales you need to resolve.
2. Calculate the maximum allowable $\Delta t_{stable} = (\Delta x)^2/(2\kappa)$.
3. Choose an operating $\Delta t < \Delta t_{stable}$ (e.g., $0.9 \times \Delta t_{stable}$ corresponding to $\alpha = 0.45$, or smaller if higher temporal accuracy is desired and computationally feasible).

Always verify your numerical solutions, for instance, by checking if results change significantly upon further reduction of Δt and Δx (convergence testing).

Scenario 2: Implementing a Neumann Boundary Condition

We now adapt our Python simulation to handle a different, common type of boundary condition: a **Neumann condition**, specifically a zero-flux (or insulated) boundary. This type of condition is crucial for modeling systems with symmetry or boundaries where no heat (or other quantity) is exchanged with the surroundings.

Problem Setup for Neumann BC Test. We modify the previous problem setup (from Section 6.3) as follows:

- **Spatial Domain, Grid, and Material Property κ:** Remain the same ($L = 1.0$ m, $N_x = 51$ points, $\kappa = 1.0 \times 10^{-6}$ m^2/s).
- **Initial Condition ($t = 0$):** To clearly observe the effect of an insulated boundary at $x = 0$, we place the hot pulse adjacent to this boundary:

$$T(x,0) = \begin{cases} 100^\circ\text{C} & \text{if } 0 \le x \le 0.2L \\ 20^\circ\text{C} & \text{elsewhere} \end{cases}$$

- **Boundary Conditions (for $t > 0$):**
  - At the **left boundary ($x = 0$, node $i = 0$):** Zero heat flux (insulated). This is a Neumann condition:
$$\left.\frac{\partial T}{\partial x}\right|_{x=0} = 0$$
  - At the **right boundary ($x = L$, node $i = N_x - 1$):** Fixed temperature (Dirichlet condition), maintained at the ambient temperature:
$$T(L,t) = 20^\circ\text{C}$$
- **Time Discretization:** We will use a time step Δt that ensures stability for the FTCS scheme (i.e., $\alpha = \kappa\Delta t/(\Delta x)^2 \leq 0.5$).

This setup models a scenario where one end of our 1D domain is perfectly insulated, preventing any heat from escaping or entering through it, while the other end is kept at a constant temperature.

Numerical Implementation of the Zero-Flux Neumann Condition. To incorporate the Neumann condition $\frac{\partial T}{\partial x}|_{x=0} = 0$ into our FTCS scheme at the boundary node $i = 0$, we need to approximate this first derivative. A common and second-order accurate approach (consistent with the central difference used for the interior $\frac{\partial^2 T}{\partial x^2}$) is to use a *central difference for the first derivative centered at x_0*. This requires imagining a "ghost point" $x_{-1} = x_0 - \Delta x$ located outside the physical domain. The central difference approximation for the first derivative at x_0 at time level n is:

$$\left.\frac{\partial T}{\partial x}\right|_{x_0}^{n} \approx \frac{T(x_1,t^n) - T(x_{-1},t^n)}{2\Delta x} = \frac{T_1^n - T_{-1}^n}{2\Delta x}$$

Setting this approximation to zero for our no-flux condition gives:

$$\frac{T_1^n - T_{-1}^n}{2\Delta x} = 0 \quad \Longrightarrow \quad T_1^n = T_{-1}^n$$

This means that the temperature at the ghost point T_{-1}^n is set equal to the temperature at the first interior physical point T_1^n.

Now, we apply the standard FTCS update formula (Equation 6.7) at the boundary node $i = 0$:

$$T_0^{n+1} = T_0^n + \alpha(T_1^n - 2T_0^n + T_{-1}^n)$$

Substituting $T_{-1}^n = T_1^n$ (from our zero-flux condition) into this FTCS formula, we obtain the specific update equation for the temperature at the insulated boundary node $i = 0$:

$$\begin{aligned} T_0^{n+1} &= T_0^n + \alpha(T_1^n - 2T_0^n + T_1^n) \\ T_0^{n+1} &= T_0^n + 2\alpha(T_1^n - T_0^n) \end{aligned} \tag{6.10}$$

This formula allows us to calculate T_0^{n+1} using only values from within the physical domain at time level n. The right boundary ($i = N_x - 1$) is still handled by the Dirichlet condition $T_{N_x-1}^{n+1} = 20^\circ$C. The interior points ($i = 1, \ldots, N_x - 2$) are updated using the standard FTCS formula (Equation 6.7).

Python Code Modifications for Neumann BC. The Python script shown in Listing 6.1 needs to be modified to implement this new initial condition and the different boundary condition update for node $i = 0$. The core changes within the time-stepping loop are highlighted in Listing 6.2.

```
# --- Within the main time-stepping loop ---
# (Assumes temperatureOld holds T^n, temperatureCurrent will
    hold T^{n+1})
# (Assumes alphaTarget, numXPoints are defined)
# (Initial condition should be set to the pulse near x=0
    before the loop)

# Loop over time steps:
for nStep in range(1, numTSteps + 1):
    # Update interior points (iPoint = 1 to numXPoints - 2)
    for iPoint in range(1, numXPoints - 1):
        temperatureCurrent[iPoint] = temperatureOld[iPoint] +
    \
            alphaTarget * (temperatureOld[iPoint+1] - \
                           2*temperatureOld[iPoint] +
    temperatureOld[iPoint-1])

    # Apply Neumann BC at x=0 (node iPoint=0) using Equation \
    ref{eq:chap6_ftcs_neumann_update_i0_v2}
    temperatureCurrent[0] = temperatureOld[0] + \
                          2 * alphaTarget * (temperatureOld[1]
    - temperatureOld[0])

    # Apply Dirichlet BC at x=L (node iPoint=numXPoints-1)
    temperatureCurrent[numXPoints-1] = 20.0 # Example fixed
    temperature

    # Update temperatureOld for the next iteration
    temperatureOld = temperatureCurrent.copy()
    # (Store for plotting as before)
```

Listing 6.2. Key modifications within the FTCS time-stepping loop to implement a zero-flux Neumann BC at $x = 0$ (node `iPoint=0`) and a Dirichlet BC at $x = L$.

Note. *A full runnable script for this Neumann case would involve setting the appropriate initial condition (pulse at $x = 0$) and integrating these boundary updates into the main loop structure similar to Listing 6.1).*

Numerical Result with Neumann BC. Figure 6.5 shows an example of the temperature evolution for this setup, using a stable $\alpha = 0.45$. As expected, the zero-flux condition at $x = 0$ prevents heat from escaping through this boundary. Consequently, the temperature gradient $\frac{\partial T}{\partial x}$ at $x = 0$ becomes (and remains close to) zero, meaning the temperature profile becomes flat at this insulated end. Heat from the initial pulse diffuses only towards the right, interacting with the fixed-temperature Dirichlet boundary at $x = L$. This behavior is distinctly different from the Dirichlet-Dirichlet case, where heat would diffuse in both directions away from a central pulse.

These examples illustrate the flexibility of the finite difference method in accommodating different physical scenarios through the appropriate formulation and implementation of boundary conditions. The choice of boundary conditions, along with the initial condition and the governing PDE itself, completely defines the mathematical problem to be solved.

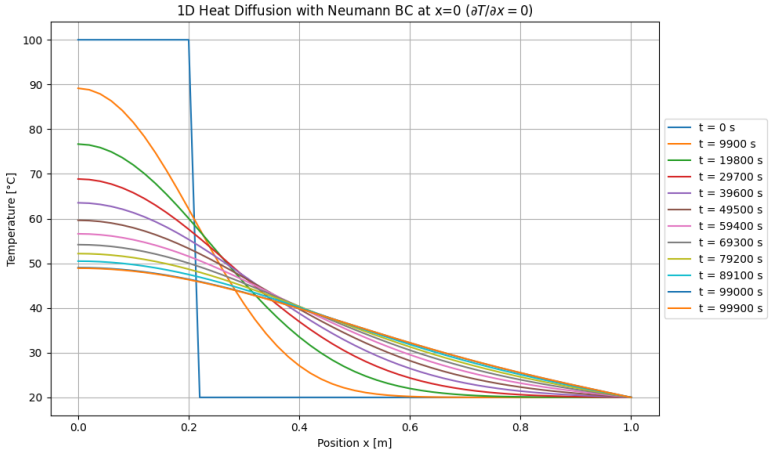


Figure 6.5. FTCS solution for the 1D Heat Equation with a zero-flux Neumann boundary condition at $x = 0$ (insulated) and a fixed temperature (Dirichlet) condition at $x = L$. The stability parameter is $\alpha = 0.45$. An initial hot pulse, placed near $x = 0$ (blue line at $t = 0$), diffuses primarily to the right. The temperature profile becomes characteristically flat (zero gradient) at the insulated left boundary.

These examples with Dirichlet and Neumann boundary conditions illustrate the adaptability of the finite difference method. However, the FTCS scheme, being explicit, always carries the burden of its conditional stability, $\Delta t \leq (\Delta x)^2/(2\kappa)$. This often motivates the exploration of alternative numerical approaches, particularly for problems requiring very fine spatial resolution or long-term simulations.

Concept Check

To implement a zero-flux Neumann boundary condition ($\partial T/\partial x = 0$) at x_0 (node $i = 0$) using the FTCS scheme, we derived the update $T_0^{n+1} = T_0^n + 2\alpha(T_1^n - T_0^n)$. This was based on introducing a "ghost point" x_{-1} and setting $T_{-1}^n = T_1^n$. Explain conceptually why setting $T_{-1}^n = T_1^n$ corresponds to a zero gradient (or zero flux, assuming constant κ) at the boundary x_0.

6.3.3 Beyond Explicit Schemes: An Introduction to Implicit Methods for Diffusion

The restrictive stability condition of the explicit FTCS scheme, $\Delta t \propto (\Delta x)^2$, can be a significant computational bottleneck for many practical geoscience problems. If a fine spatial grid (small Δx) is needed to resolve important features, the maximum permissible time step Δt becomes exceedingly small, leading to a very large number of time steps and consequently long simulation runtimes. This is especially problematic in two or three spatial dimensions, where the stability constraints for explicit schemes become even more severe.

To overcome this limitation, **implicit numerical schemes** can be employed for the heat (or diffusion) equation, similar in spirit to the Implicit Euler method we encountered for

ODEs in Chapter 3. Implicit schemes evaluate some or all of the spatial derivative terms at the *future* (unknown) time level $n+1$.

Common implicit schemes for the 1D heat equation include:

1. **Backward Time, Central Space (BTCS) Scheme:** This scheme is the direct implicit counterpart to FTCS. It uses a forward difference for the time derivative (as before) but approximates the central difference for the spatial derivative using temperatures at the *new* time level $n+1$:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \kappa \left(\frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{(\Delta x)^2} \right) \tag{6.11}$$

   In this equation, the unknown temperatures T_{i-1}^{n+1}, T_i^{n+1}, and T_{i+1}^{n+1} all appear. Rearranging this equation for all interior grid points i results in a *system of simultaneous linear algebraic equations* that must be solved at each time step to find the vector of temperatures $\mathbf{T}^{n+1}$. For 1D problems, this system is often tridiagonal, for which efficient solvers (like the Thomas algorithm) exist.

2. **Crank-Nicolson Scheme:** This widely used scheme aims for higher accuracy by averaging the central space difference at both the current time level n and the new time level $n+1$. It is often described as being "centered" in both space and time:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{\kappa}{2} \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} + \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{(\Delta x)^2} \right) \tag{6.12}$$

   Like the BTCS scheme, this also results in a system of linear algebraic equations for the T_i^{n+1} values at each time step. A key advantage of the Crank-Nicolson scheme is its second-order accuracy in both time and space, i.e., $\mathcal{O}(\Delta t^2, (\Delta x)^2)$, making it potentially more accurate than FTCS or BTCS (which is $\mathcal{O}(\Delta t, (\Delta x)^2)$) for a given step size.

Key Advantages of Implicit Schemes for Diffusion:

- **Unconditional Stability:** The primary advantage of schemes like BTCS and Crank-Nicolson is that they are generally **unconditionally stable** for the linear heat equation. This means that, from a stability perspective, there is no upper limit on the time step Δt relative to Δx and κ. The choice of Δt can therefore be based primarily on considerations of temporal accuracy (i.e., how well the scheme resolves the time evolution of the solution) rather than being dictated by a strict stability criterion. This can allow for significantly larger time steps than explicit methods, especially when Δx is small or κ is large.

Key Disadvantages of Implicit Schemes for Diffusion:

- **Increased Computational Cost per Time Step:** Although larger time steps can be taken, each individual time step is more computationally expensive. This is because a system of $N_x - 2$ (for 1D with Dirichlet BCs) coupled linear algebraic equations must be solved to find the temperatures at all interior grid points simultaneously. While efficient direct solvers (like the Thomas algorithm for tridiagonal systems) or iterative solvers can be used, the work per time step is still greater than the simple explicit updates of FTCS.

- **Implementation Complexity:** Implementing implicit solvers, especially for 2D or 3D problems where the resulting linear systems are larger and more complex (e.g., banded but not tridiagonal), is generally more involved than coding explicit schemes.

The decision to use an explicit versus an implicit scheme for a diffusion problem often involves a trade-off. If the stability constraint of an explicit scheme forces Δt to be comparable to or smaller than what would be chosen for accuracy anyway, an explicit scheme might be more efficient overall due to its lower cost per step. However, if stability demands a Δt that is much smaller than what accuracy alone would require (a "stiff" situation), an implicit scheme, despite its higher cost per step, might be more efficient because it can take far fewer, much larger, time steps.

For the remainder of our introductory exploration, we will continue to focus on understanding and utilizing explicit methods, but it is crucial to be aware that these more robust (in terms of stability) implicit alternatives exist and are widely used in advanced numerical modelling.

Chapter Summary: Numerical Solution of the 1D Heat Equation

This chapter provided our first in-depth look at numerically solving a Partial Differential Equation (PDE), focusing on the fundamental 1D Heat (or Diffusion) Equation, $\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$. We built upon concepts from previous chapters to develop, implement, and analyze an explicit finite difference scheme.

The key topics and skills developed in this chapter include:

- **The Importance of Boundary Conditions (BCs) for PDEs:**
  - Emphasized that for PDEs on finite domains, initial conditions (for time) must be supplemented by boundary conditions (for space) to obtain unique and physically meaningful solutions.
  - Reviewed common types of BCs:
    - **Dirichlet (Type I):** Specifies the value of the variable (e.g., temperature) at the boundary.
    - **Neumann (Type II):** Specifies the gradient (and thus flux) of the variable at the boundary (e.g., zero-flux for an insulated boundary).
    - **Robin (Type III):** Specifies a mixed condition, often representing convective heat transfer.
  - Highlighted how BCs represent the system's interaction with its surroundings and are crucial for accurate modeling.
- **Numerical Discretization of the 1D Heat Equation:**
  - Detailed the setup of **spatial and temporal grids** (x_i, t^n) with respective step sizes $\Delta x, \Delta t$.
  - Reaffirmed the use of a **Forward Difference** in time for $\frac{\partial T}{\partial t}$, leading to an $\mathcal{O}(\Delta t)$ approximation.
  - Formally derived the **Central Difference** approximation for the second spatial derivative $\frac{\partial^2 T}{\partial x^2}$ using Taylor series expansions, showing it to be $\mathcal{O}(\Delta x^2)$ accurate.

- **The Forward-Time Central-Space (FTCS) Scheme:**
  - Assembled the derivative approximations to derive the explicit FTCS update formula: $T_i^{n+1} = T_i^n + \alpha(T_{i+1}^n - 2T_i^n + T_{i-1}^n)$, where $\alpha = \kappa\Delta t/(\Delta x)^2$.
  - Discussed the physical interpretation of this scheme as a weighted average representing heat diffusion.
- **Stability of the FTCS Scheme:**
  - Introduced the concept of **conditional stability** for explicit schemes.
  - Stated and intuitively explained the stability condition for the FTCS heat equation: $\alpha \leq 0.5$, or $\Delta t \leq (\Delta x)^2/(2\kappa)$.
  - Discussed the practical implication: the time step Δt is severely restricted by the spatial step Δx (scaling with $(\Delta x)^2$), which can make simulations computationally expensive for fine grids.
- **Python Implementation and Visualization:**
  - Developed a Python script using NumPy and Matplotlib to implement the FTCS scheme for the 1D heat equation.
  - Demonstrated the implementation of **Dirichlet boundary conditions** and visualized both stable ($\alpha \leq 0.5$) and unstable ($\alpha > 0.5$) solutions, highlighting the growth of numerical oscillations when the stability condition is violated.
  - Showed how to implement a **zero-flux Neumann boundary condition** by modifying the update formula at the boundary node (using a ghost point concept) and visualized its effect on the temperature profile.
- **Practical Considerations and Outlook:**
  - Discussed factors influencing the choice of Δx (spatial resolution) and Δt (stability and temporal accuracy).
  - Briefly introduced **implicit schemes** (like BTCS and Crank-Nicolson) as alternatives that are unconditionally stable for the linear heat equation, allowing larger time steps but requiring the solution of a linear system at each step.

This chapter has provided a comprehensive introduction to numerically solving a parabolic PDE. You have learned to discretize the equation, implement an explicit finite difference scheme, manage boundary conditions, analyze stability, and interpret the results. These skills form a critical foundation for tackling more complex PDE problems in the Earth sciences. The exercises that follow will help reinforce these concepts.

Chapter 6 Exercises

These exercises will help you solidify your understanding of the FTCS scheme for the 1D Heat Equation, stability considerations, and the implementation of boundary conditions using Python. You should base your work on the Python scripts presented in this chapter (e.g., Listing **??** and the modifications for Neumann conditions).

E6.1: Exploring the Stability Parameter α

Using the Python script for the 1D Heat Equation with Dirichlet boundary conditions (Listing **??**), perform the following numerical experiments. Keep the physical parameters ($L = 1.0$ m, $N_x = 51$ points, $\kappa = 1.0 \times 10^{-6}$ m^2/s), initial condition (central hot pulse), and $t_{final} = 100,000$ s the same as in the chapter example.

1. **Stable Case (Reference):** Run the simulation with `alphaTarget = 0.45`. Observe and save the plot of temperature profiles over time. This will be your reference stable solution.
2. **Marginally Stable Case:** Run the simulation with `alphaTarget = 0.50`. How does the solution behave compared to the $\alpha = 0.45$ case? Do you notice any subtle differences or artifacts, even if it doesn't "explode"?
3. **Mildly Unstable Case:** Run the simulation with `alphaTarget = 0.51`. Describe what happens to the solution. Does it become unstable immediately, or does the instability take some time to become apparent?
4. **Very Unstable Case:** Run the simulation with `alphaTarget = 0.60`. How does this compare to the $\alpha = 0.51$ case?
5. **Discussion:** Based on your observations, comment on the practical importance of choosing an `alphaTarget` safely below the theoretical stability limit of 0.5.

E6.2: Impact of Thermal Diffusivity κ

Using the Python script for Dirichlet BCs (Listing 6.1) and a stable `alphaTarget` (e.g., 0.45), investigate the effect of thermal diffusivity κ. Keep $L = 1.0$ m, $N_x = 51$, and the central hot pulse initial condition. Simulate up to $t_{final} = 100,000$ s.

1. Set $\kappa = 1.0 \times 10^{-7}$ m^2/s (slower diffusion, e.g., like some dry rocks). Run the simulation.
2. Set $\kappa = 5.0 \times 10^{-6}$ m^2/s (faster diffusion, e.g., like some water-saturated sediments or a different material). Run the simulation.
3. **Plot and Compare:** For each case, plot the temperature profiles at selected times (e.g., initial, a few intermediate, and final). How does the rate at which the initial heat pulse diffuses and cools down change with different κ values?
4. **Time Step Consideration:** When you change κ, how does the `dtStep` calculated by the script (to maintain the same `alphaTarget`) change? Explain why.

E6.3: Implementing a Different Neumann Boundary Condition

Modify the Python script that handles the Neumann BC at $x = 0$ and Dirichlet at $x = L$. Instead of a zero-flux condition at $x = 0$, implement a **constant positive heat flux** into the domain at $x = 0$. The physical condition is $q_0 = -k_c \frac{\partial T}{\partial x}\big|_{x=0}$, where q_0 is the specified inward heat flux (e.g., $q_0 = 10$ W/m^2). The numerical condition for $\frac{\partial T}{\partial x}$ at $x = 0$ will then be $\frac{\partial T}{\partial x}\big|_{x=0} = -q_0/k_c$. (Assume $\rho c_p = 2 \times 10^6$ J m^{-3}K^{-1}, so $k_c = \kappa \rho c_p = (10^{-6}$ m^2/s$) \times (2 \times 10^6$ J m^{-3}K$^{-1}) = 2$ W m^{-1}K^{-1}).

1. **Derive the Update Formula for T_0^{n+1}:** Using the central difference approximation $\frac{T_1^n - T_{-1}^n}{2\Delta x} = -q_0/k_c$ to find T_{-1}^n, substitute this into the FTCS formula for T_0^{n+1}. Show your derived formula. (Hint: $T_{-1}^n = T_1^n + 2\Delta x(q_0/k_c)$).

2. **Implement and Simulate:** Modify your Python script to use this new update for T_0^{n+1}. Use an initial condition of $T(x,0) = 20°C$ everywhere. Keep the Dirichlet condition $T(L,t) = 20°C$ at the right boundary. Use a stable α (e.g., 0.45) and simulate for a sufficiently long time (e.g., $t_{final} = 200,000$ s or more) to observe the temperature profile evolution.
3. **Plot and Discuss:** Plot the temperature profiles at different times. How does the temperature at $x = 0$ and within the domain change over time due to the constant heat influx? Does the system reach a steady state? If so, what does the steady-state profile look like?

E6.4: Grid Refinement Study (Qualitative)

Using the original Dirichlet BC scenario (Listing 6.1) with the central hot pulse and stable `alphaTarget = 0.45`:

1. Run the simulation with $N_x = 21$ points. Note the Δx and the Δt calculated. Plot the temperature profile at t_{final}.
2. Run the simulation with $N_x = 51$ points (as in the chapter example). Note the new Δx and Δt. Plot the temperature profile at t_{final}.
3. Run the simulation with $N_x = 101$ points. Note the new Δx and Δt. Plot the temperature profile at t_{final}.
4. **Compare and Discuss:**
   - How do the temperature profiles at t_{final} compare as N_x increases (and Δx decreases)? Do they seem to converge to a particular shape?
   - How does the required Δt (and thus the total number of time steps for a fixed t_{final}) change as N_x increases?
   - This exercise provides a qualitative feel for grid convergence. A more rigorous study would involve quantifying the error against a highly resolved solution or an analytical solution if available for a simpler IC.

Chapter 7

Numerical Solution of the Linear Advection Equation

The previous chapters have laid a crucial foundation, moving from the conceptual understanding of Ordinary and Partial Differential Equations (ODEs and PDEs) derived from physical principles, to the practicalities of numerical approximation using Python, NumPy, and Matplotlib. We have explored basic time-stepping schemes like the Euler methods for ODEs, investigated the approximation of spatial derivatives via finite differences, and developed a numerical solver for a fundamental parabolic PDE, the Heat (or Diffusion) Equation.

With this background, we now turn our attention to another cornerstone of transport phenomena: **advection**. This process, where substances or properties are carried along by the bulk movement of a fluid, is ubiquitous in Earth systems. From the grand scale of atmospheric and oceanic currents shaping global climate, to the localized dispersion of volcanic emissions or the transport of nutrients in a river, advection plays a pivotal role. This chapter is dedicated to understanding the simplest mathematical representation of this process—the 1D Linear Advection Equation—and to developing and analyzing numerical schemes for its solution.

Our journey through this chapter will involve:

- Defining advection and appreciating its significance across various Earth science disciplines.
- Formally deriving the 1D linear advection equation as a specific case of the general scalar transport equation, highlighting its characteristics as a first-order, linear, hyperbolic PDE.
- Examining its analytical solution and the insightful concept of characteristic curves, which describe how information propagates in advective systems.
- Discussing the critical role of boundary conditions, with a special focus on periodic conditions often used for testing advection schemes.
- Developing several explicit finite difference schemes, including the Forward-Time Centered-Space (FTCS) method and the upwind approach.
- Implementing these schemes in Python to visualize their behavior, paying close attention to how well they replicate the exact solution.

- A thorough investigation of numerical stability, centered around the Courant-Friedrichs-Lewy (CFL) condition, and an exploration of common numerical artifacts such as numerical diffusion and dispersion that can affect solution accuracy.

By the end of this chapter, the aim is to provide a solid understanding of both the physical process of linear advection and the fundamental numerical techniques used to simulate it, along with an appreciation for the challenges and considerations involved in obtaining reliable numerical solutions.

7.1 The Linear Advection Equation: Physical and Mathematical Basis

7.1.1 What is Advection? The "Carrying Along" Process in Earth Systems

At its core, **advection** describes the transport of a scalar quantity—be it mass (like a dissolved chemical, suspended sediment, or volcanic ash), energy (like heat entrained in a moving fluid), or momentum—by the bulk motion or flow of the medium in which this quantity is embedded. Imagine a puff of volcanic ash released into the atmosphere, as depicted in Figure 7.1; its subsequent movement over potentially vast distances is primarily governed by the prevailing wind patterns. The ash particles themselves do not possess an intrinsic ability to travel these distances through the air; rather, they are *passively carried along*, or advected, by the moving air mass. Similarly, if a soluble pollutant is introduced into a river, its downstream journey is largely a result of being advected by the river's current.

This "carrying along" mechanism is a fundamental mode of transport in countless natural systems and is distinct from **diffusion**. Diffusion, as we saw in the context of the Heat Equation (Chapter 6), describes the transport of a quantity down its concentration (or temperature, etc.) gradient, driven by random molecular motion or small-scale turbulent eddies. Diffusion tends to smooth out variations and spread a substance from regions of high concentration to low concentration in all available directions. Advection, on the other hand, is inherently directional, dictated by the velocity field of the carrying fluid. A plume of volcanic ash, for example, will primarily travel downwind, and its shape will be elongated in the direction of the wind. However, it is important to note that if advection were the only process, the plume would travel as a narrow, non-spreading filament. In reality, turbulent eddies in the atmosphere act as a powerful diffusive mechanism. This turbulence causes the plume to spread out vertically and horizontally, mixing it with the surrounding air and diluting the ash concentration with distance from the volcano. The final shape and concentration profile of the ash cloud is therefore a result of both directional advection and multi-directional diffusion.

The significance of advection in shaping our planet and its environment cannot be overstated. It is a central process in:

- **Atmospheric Sciences:** The global and regional dispersion of natural aerosols (volcanic ash, dust, sea salt) and anthropogenic pollutants (soot, industrial emissions) profoundly affects air quality, atmospheric chemistry, cloud formation, and the Earth's radiative balance. The large-scale circulation of the atmosphere itself, which drives weather systems and transports heat and moisture across latitudes, is a manifestation of advection.

Figure 7.1. An ash plume from the Shiveluch Volcano, located on Russia's Kamchatka Peninsula, vividly illustrates large-scale advective transport. This image was captured by the Moderate Resolution Imaging Spectroradiometer (MODIS) instrument aboard NASA's Aqua satellite at approximately noon local time (00:00 Universal Time) on November 6, 2012. The plume can be seen extending about 90 kilometers south-southeast over the Kamchatka Gulf (Kamchatskiy Zaliv), where a shift in wind direction begins to push it eastward towards the Bering Sea. The shape and trajectory of such plumes are primarily governed by advection within the atmospheric wind field. (Image credit: Jeff Schmaltz, LANCE MODIS Rapid Response Team, NASA Goddard Space Flight Center).

- **Hydrology and Oceanography:** Rivers act as primary conduits for the advection of sediments from terrestrial sources to coastal environments and the deep sea, building deltas and shaping coastlines. The movement of dissolved nutrients, salts, and contaminants within river networks, lakes, estuaries, and groundwater aquifers is largely governed by advection with the water flow. Similarly, major ocean currents like the Gulf Stream or the Antarctic Circumpolar Current are massive advective systems, transporting vast quantities of heat, salt, dissolved gases (such as CO_2), and marine organisms (plankton, larvae) across ocean basins, thereby playing a critical role in global climate regulation and marine ecosystem dynamics.
- **Solid Earth Geosciences:** Within magmatic systems, the flow of magma in volcanic conduits, dikes, or larger chambers can advect suspended crystals, entrained gas bubbles, or chemically distinct parcels of melt (xenoliths, enclaves from magma mingling processes). On the grandest scale, mantle convection involves the slow, thermally driven advection of mantle rock, which is the primary engine for plate tectonics and dictates the long-term thermal and chemical evolution of our planet.

Given its pervasive importance, developing the ability to mathematically describe and numerically simulate advective transport is an essential skill for quantitative Earth scientists. This chapter will focus on the simplest, yet foundational, mathematical representation of

this process: the 1D linear advection equation with a constant velocity.

7.1.2 The 1D Linear Advection Equation: Formulation and Interpretation

The 1D Linear Advection Equation is derived from the general scalar transport equation (Equation 5.20 from Chapter 5) by applying a set of simplifying assumptions: constant density of the carrying fluid (ρ), negligible diffusion ($\Gamma = 0$), no internal sources or sinks of the advected quantity ($S_\phi = 0$), and a one-dimensional flow field with a constant velocity u. Under these conditions, the general equation simplifies to:

$$\boxed{\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0} \tag{7.1}$$

Here, $\phi(x,t)$ represents the scalar quantity being advected (e.g., concentration, temperature anomaly) as a function of spatial position x and time t, and u is the constant advection velocity in the x-direction.

Key Mathematical Properties. This equation is characterized by being:

- **First-Order:** It contains only first-order partial derivatives with respect to both time (t) and space (x).
- **Linear:** The unknown function ϕ and its derivatives appear only to the first power and are not multiplied by each other. The coefficient u is constant.
- **Hyperbolic:** This is a mathematical classification that has profound implications for the behavior of its solutions. For hyperbolic PDEs, information or disturbances in the field ϕ propagate at a finite speed through the domain along specific paths known as characteristic curves. The direction of this propagation is well-defined. This is fundamentally different from parabolic (diffusive) equations, such as the Heat Equation, where disturbances, in principle, influence the entire domain instantaneously (though the effect diminishes rapidly with distance), and the primary process is one of smoothing and equilibration rather than directed propagation.

Physical Interpretation of the Terms. The equation $\frac{\partial \phi}{\partial t} = -u\frac{\partial \phi}{\partial x}$ provides insight into how local spatial variations in ϕ drive its temporal evolution at a fixed point. Let's consider an initial spatial profile of ϕ, for instance, a pulse or wave-like shape, as conceptually shown in Figure 7.2.

The term $\frac{\partial \phi}{\partial t}$ represents the rate at which ϕ changes at a fixed location x. The term $-u\frac{\partial \phi}{\partial x}$ dictates this rate of change.

- If $u > 0$ (advection to the right):
  - On the left (rising) flank of a pulse moving to the right (Figure 7.3a, where $\frac{\partial \phi}{\partial x} > 0$), the term $-u\frac{\partial \phi}{\partial x}$ is negative. Thus, $\frac{\partial \phi}{\partial t} < 0$, meaning the value of ϕ at a fixed point on this flank will decrease as the higher values of the pulse move past it.
  - On the right (falling) flank of this pulse (Figure 7.3b, where $\frac{\partial \phi}{\partial x} < 0$), the term $-u\frac{\partial \phi}{\partial x}$ is positive. Thus, $\frac{\partial \phi}{\partial t} > 0$, meaning ϕ increases as the tail of the pulse (or lower values preceding a new pulse) passes by and the main body of the pulse arrives.

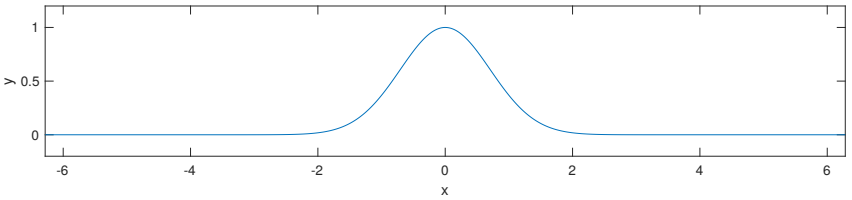
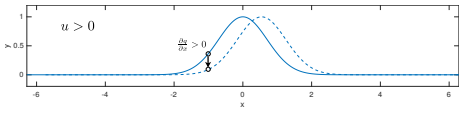


Figure 7.2. An example initial profile $\phi_0(x)$ for an advection problem. The local value of the spatial derivative $\partial\phi/\partial x$ (the slope) varies along this profile. For example, at the peak of this symmetric pulse, the slope would be zero.

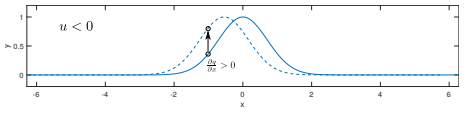
- If $u < 0$ (advection to the left):
  - On the right (rising, relative to direction of motion) flank of a pulse moving to the left (Figure 7.3c, where $\frac{\partial\phi}{\partial x} > 0$), the term $-u\frac{\partial\phi}{\partial x}$ becomes positive (since u is negative). Thus, $\frac{\partial\phi}{\partial t} > 0$, and ϕ increases.
  - On the left (falling, relative to direction of motion) flank of this pulse (Figure 7.3d, where $\frac{\partial\phi}{\partial x} < 0$), the term $-u\frac{\partial\phi}{\partial x}$ becomes negative. Thus, $\frac{\partial\phi}{\partial t} < 0$, and ϕ decreases.

These local changes observed at fixed points are entirely consistent with the entire initial profile of ϕ being transported or advected rigidly with velocity u without changing its shape. The phrase "for a short time" often accompanies such local interpretations because, as the profile moves, the value of $\frac{\partial\phi}{\partial x}$ at a fixed point x_i will itself change.



(a) $u > 0, \partial\phi/\partial x > 0 \Rightarrow \partial\phi/\partial t < 0$

(b) $u > 0, \partial\phi/\partial x < 0 \Rightarrow \partial\phi/\partial t > 0$



(c) $u < 0, \partial\phi/\partial x > 0 \Rightarrow \partial\phi/\partial t > 0$

(d) $u < 0, \partial\phi/\partial x < 0 \Rightarrow \partial\phi/\partial t < 0$

Figure 7.3. Local rates of change for ϕ at fixed points on different flanks of a translating pulse, illustrating how the advection equation $\frac{\partial\phi}{\partial t} = -u\frac{\partial\phi}{\partial x}$ describes this movement.

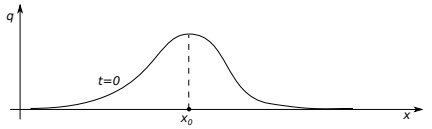
7.1.3 Analytical Solution: Pure Translation and Characteristic Curves

The 1D linear advection equation with constant velocity u (Equation 7.1) is remarkable for its simple and elegant analytical solution. If the initial spatial distribution of ϕ at time $t = 0$ is given by a known function $\phi_0(x)$, i.e., $\phi(x, t = 0) = \phi_0(x)$, then the solution at any

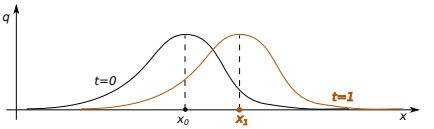
subsequent time t is:

$$\phi(x,t) = \phi_0(x-ut) \tag{7.2}$$

This solution conveys a clear physical meaning: the entire initial profile $\phi_0(x)$ is simply **translated** or shifted along the x-axis by a distance ut after a time t has passed. Most importantly, for pure linear advection, this translation occurs **without any change in the shape or amplitude** of the profile. It is a rigid displacement, as illustrated in Figure 7.4.



An initial profile $\phi_0(x)$ at time $t = 0$. The peak (or any feature) is located at some position x_0.



The same profile at a later time t_1. If $u > 0$, it has shifted to the right. The peak is now at $x_1 = x_0 + ut_1$.

Figure 7.4. Illustration of the analytical solution for linear advection. The initial spatial distribution of ϕ (left panel) is transported by a constant velocity u (assumed positive here, so transport is to the right) without any distortion in shape or reduction in amplitude, resulting in a simply shifted profile at a later time (right panel).

This behavior is intimately linked to the concept of **characteristic curves**. As briefly mentioned, these are paths in the $x-t$ plane along which the solution ϕ remains constant. For the equation $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$, if we consider an observer moving with velocity $dx/dt = u$, the rate of change of ϕ experienced by this observer (the material derivative) is $D\phi/Dt = 0$. This implies ϕ is constant along these paths $x(t) = ut + x_{initial}$. These straight lines are the characteristic curves. The value of ϕ at any point (x,t) is found by tracing the unique characteristic curve passing through (x,t) back to its intersection with the x-axis at $t = 0$, say at $x_0 = x - ut$. Then, $\phi(x,t) = \phi_0(x_0)$. This is the essence of the method of characteristics.

So far, we have often visualized the solution as a profile $\phi(x)$ at different, discrete moments in time (e.g., Figure 7.4). However, the full solution $\phi(x,t)$ is a surface defined over the two-dimensional $x-t$ plane. Figure 7.5 provides a conceptual view of this solution surface, where profiles at different times are "stacked" along the time axis.

The "information" of the solution—for instance, a particular value of ϕ or the location of a peak—travels through the domain along these characteristic curves with a finite speed. For a constant-coefficient first-order linear equation like ours, the solution ϕ is actually **constant** along these curves.

Let's prove this. Consider the advection equation:

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0 \quad (*)$$

A characteristic curve is a path in the $x-t$ plane, described by a function $x(t)$, along which an observer would need to move to see a constant value of ϕ. The velocity of this observer, $\frac{dx}{dt}$, must be chosen carefully. The total rate of change of ϕ experienced by an observer moving along a path $x(t)$ is given by the material (or total) derivative:

$$\frac{d}{dt}\phi(x(t),t) = \frac{\partial \phi}{\partial t} + \frac{\partial \phi}{\partial x}\frac{dx}{dt}$$

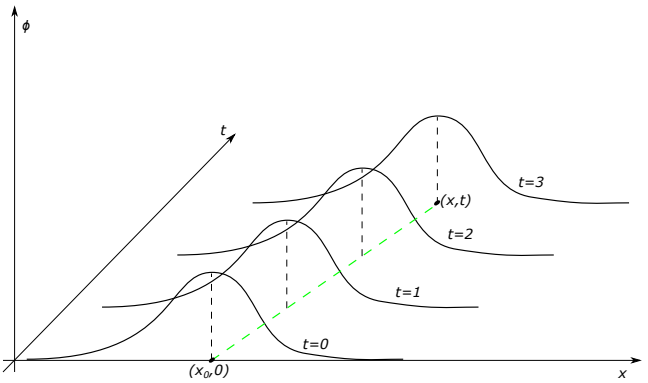


Figure 7.5. A three-dimensional representation of the solution $\phi(x,t)$ to the advection equation. The profiles $\phi(x)$ at different times ($t = 0, 1, 2, 3$) are shown. The "information" of the profile, such as its peak, travels along specific paths in the $x-t$ plane. The green dashed line connecting the peaks is an example of a **characteristic curve**.

If we choose the path such that the observer's velocity is exactly the advection velocity of the fluid, i.e.,

$$\frac{dx}{dt} = u \tag{7.3}$$

then the material derivative becomes:

$$\frac{d}{dt}\phi(x(t),t) = \frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x}$$

But the right-hand side is exactly the left-hand side of our advection equation (*), which is equal to zero. Therefore, we have found that:

$$\frac{d}{dt}\phi(x(t),t) = 0 \tag{7.4}$$

This confirms that the solution ϕ is indeed constant along the characteristic curves defined by the Ordinary Differential Equation (ODE) $\frac{dx}{dt} = u$. For constant u, the solution to this ODE is simply $x(t) = ut + x_0$, where x_0 is the starting position of the characteristic at $t = 0$. These are straight lines in the $x-t$ plane.

This principle forms the basis of the **Method of Characteristics** for finding the solution to an initial value problem (a "Cauchy problem"). To find the value of the solution at any point $P = (x,t)$:

1. **Find the Characteristic Curve:** Determine the equation of the characteristic curve that passes through the point P. For our problem, this is the line $x(t') = ut' + x_0$. We find x_0 by substituting our point (x,t): $x = ut + x_0 \implies x_0 = x - ut$.
2. **Trace Back to Initial Condition:** Find the point where this characteristic curve intersects the initial time axis ($t = 0$). This occurs at position $x_0 = x - ut$.
3. **Apply Constancy of Solution:** Since the solution ϕ is constant along the characteristic, the value at (x,t) must be the same as the value at the initial point $(x_0, 0)$. The value at $(x_0, 0)$ is given by the initial condition $\phi_0(x_0)$. Therefore:

$$\phi(x,t) = \phi(x_0,0) = \phi_0(x_0) = \phi_0(x-ut)$$

This elegant method recovers the same analytical solution we stated earlier and provides a powerful geometric interpretation of how hyperbolic PDEs propagate information. The value at a point P depends only on the initial data within its "domain of dependence," which is the single point x_0 on the initial line.

Visualizing the Analytical Solution with Python

To gain a practical appreciation for this shape-preserving translation associated with the 1D linear advection equation, and to establish a clear benchmark against which our future numerical solutions can be compared, it is instructive to plot the analytical solution using Python. We will define a specific initial condition—for this example, a square wave (often called a "top-hat" function)—and then plot this initial profile at $t = 0$ alongside its advected position at a later time $t = t_{\text{final}}$.

The Python script in Listing 7.1 (using the exact code you provided) demonstrates how to generate these plots using NumPy for array calculations and Matplotlib for visualization.

```
# --- Analytical Solution Visualization ---
import numpy as np
import matplotlib.pyplot as plt

def initialConditionGaussian(x, x0=0.5, sigma=0.1): # Note:
    camelCase for consistency
    """A Gaussian pulse as an initial condition."""
    return np.exp(-0.5 * ((x - x0) / sigma)**2)

def initialConditionSquareWave(x, xStart=0.25, xEnd=0.75,
    phiLow=0.0, phiHigh=1.0): # camelCase
    """A square wave (top-hat) as an initial condition."""
    phi0 = np.zeros_like(x) # Using np.zeros_like for
    efficiency
    phi0[(x >= xStart) & (x <= xEnd)] = phiHigh
    return phi0

# Domain and parameters for analytical solution
lengthDomain = 2.0
nxAnalytical = 201 # Using camelCase for variable name
xAnalytical = np.linspace(0, lengthDomain, nxAnalytical)
advectionVelocity = 1.0    # Advection velocity

# Choose initial condition type
# icFunction = initialConditionGaussian # Using camelCase
icFunction = initialConditionSquareWave # Using camelCase

# Initial profile
# For Square Wave:
phi0Analytical = icFunction(xAnalytical, xStart=0.25, xEnd
    =0.75) # camelCase
# For Gaussian specific parameters:
# phi0Analytical = icFunction(xAnalytical, x0=0.5, sigma=0.08)


# Time points for plotting
timeInitial = 0.0
timeFinalAnalytical = 0.5
```

```
# Analytical solution at tFinal
xShiftedAnalytical = xAnalytical - advectionVelocity *
    timeFinalAnalytical
# For Square Wave:
phiTFinalAnalytical = icFunction(xShiftedAnalytical, xStart
    =0.25, xEnd=0.75) # camelCase
# For Gaussian:
# phiTFinalAnalytical = icFunction(xShiftedAnalytical, x0=0.5,
     sigma=0.08)


# Plotting
plt.figure(figsize=(10, 6))
plt.plot(xAnalytical, phi0Analytical, 'b-',
         label=f'Initial Condition $\phi_0(x)$ (t={timeInitial
    :.1f})')
plt.plot(xAnalytical, phiTFinalAnalytical, 'r--',
         label=f'Exact Solution $\phi(x,t)$ (t={
    timeFinalAnalytical:.1f}, u={advectionVelocity})')
plt.xlabel('Position x')
plt.ylabel('$\phi$')
plt.title('Exact Solution of 1D Linear Advection')
plt.legend()
plt.grid(True)
plt.ylim(-0.1, 1.1)
plt.show()
```

Listing 7.1. Python script to compute and plot the analytical solution of the 1D linear advection equation for an initial square wave profile. This script uses the exact code provided by the student.

Executing this script will produce a plot similar to that shown in Figure 7.6. The figure clearly demonstrates the pure translation of the initial square wave: its sharp edges are perfectly maintained, and its amplitude remains constant; only its position changes due to the advection process.

This exact, distortion-free translation is the ideal behavior we expect from a pure advection process. It will serve as a critical reference: any deviations observed in our numerical solutions—such as the smearing of the sharp edges of the square wave (numerical diffusion), the appearance of spurious oscillations (numerical dispersion), or an incorrect propagation speed—will be indicative of errors and artifacts introduced by the chosen numerical discretization scheme. Understanding these potential numerical issues is a key theme of this chapter.

7.2 Boundary Conditions for Advection Problems

The analytical solution $\phi(x,t) = \phi_0(x-ut)$ for the 1D linear advection equation, which describes a perfect, shape-preserving translation of an initial profile, is strictly valid for an infinitely long spatial domain ($x \in (-\infty, \infty)$). In this idealized scenario, the profile $\phi_0(x)$ is known for all x, and there are no "ends" to the domain to influence the solution. However, when we perform numerical simulations, we are invariably restricted to a **finite computational domain**, for instance, a segment of a river, a specific region of the atmosphere, or simply a defined interval $x \in [0, L]$ for our numerical grid.

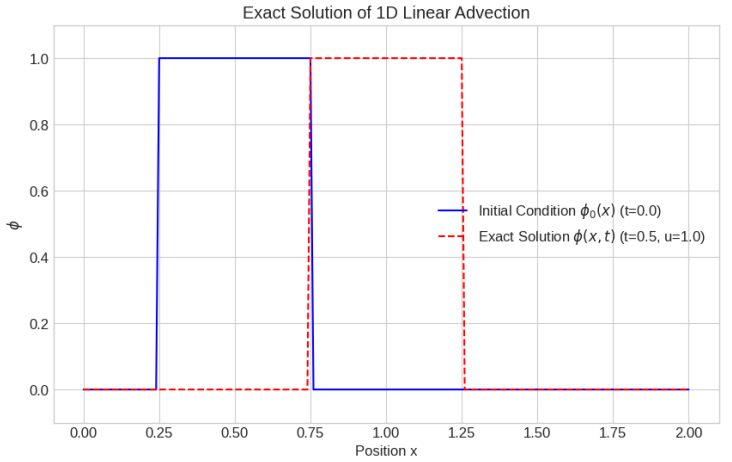


Figure 7.6. Analytical solution of the 1D linear advection equation for an initial square wave profile. The initial condition (solid blue line, representing $\phi(x, t = 0.0)$) is advected with velocity $u = 1.0$. The dashed red line shows the profile at $t = 0.5$, perfectly translated to the right without any change in shape or amplitude. This plot is generated by the script in Listing 7.1.

When the domain is finite, the Partial Differential Equation (PDE) itself, even when coupled with an initial condition $\phi(x, 0)$, is not sufficient to yield a unique solution. As done for the heat equation in the previous chapter, we need to specify the **boundary conditions (BCs)**, i.e. the additional information that dictates how the quantity ϕ behaves, or how it interacts with the "outside world," at the **boundaries** of this finite domain (e.g., at $x = 0$ and $x = L$).

As already stated, the proper formulation of boundary conditions is a critical step in setting up any numerical model based on PDEs. They serve several vital roles:

- **Mathematical Well-Posedness:** BCs provide the necessary mathematical constraints to ensure that the problem we are trying to solve has a single, unique solution. For first-order hyperbolic PDEs like the advection equation, the number and type of BCs needed at each boundary depend critically on the direction of information flow, which is determined by the sign of the advection velocity u relative to the boundary normal.
- **Physical Realism:** BCs are our way of representing the physical reality or assumptions about what happens at the edges of our modeled system. Does material flow into the domain at a known rate or concentration? Does it flow out freely? Is the boundary a solid wall, or does the system repeat itself? The answers to these questions translate into specific mathematical forms for the BCs.
- **Numerical Implementation:** BCs directly impact how the numerical scheme is applied to the grid points located at or near the boundaries of the computational mesh. These boundary nodes often require special treatment compared to interior nodes.

The choice of boundary conditions must be carefully considered, as inappropriate or poorly implemented BCs can lead to physically unrealistic solutions, introduce spurious numerical artifacts (like artificial reflections of waves), or even cause numerical instability.

7.2.1 Recap of Common Boundary Condition Types

In our discussion of the Heat Equation in Chapter 6, we introduced two fundamental types of boundary conditions:

- **Dirichlet (or Type I) Condition:** This condition specifies the *value* of the unknown variable ϕ directly at the boundary. For our 1D domain $x \in [0, L]$, a Dirichlet BC would take the form $\phi(0, t) = g_0(t)$ or $\phi(L, t) = g_L(t)$, where $g_0(t)$ and $g_L(t)$ are known functions that can vary with time (or be constant).
  - *Advection Context:* A common application is an **inflow boundary**. If fluid carrying a tracer with a known concentration ϕ_{inflow} enters the domain at $x = 0$ (and the advection velocity u is positive, pointing into the domain from this boundary), then a Dirichlet condition $\phi(0, t) = \phi_{inflow}(t)$ would be appropriate to specify this incoming concentration.
- **Neumann (or Type II) Condition:** This condition specifies the *gradient* (the first spatial derivative) of ϕ normal to the boundary. In 1D, this would be $\frac{\partial \phi}{\partial x}\Big|_{boundary} = h(t)$, where $h(t)$ is a specified function.
  - *Advection Context:* For an **outflow boundary** (e.g., at $x = L$ if $u > 0$), a frequently used, though sometimes problematic, Neumann condition is the **zero-gradient condition**: $\frac{\partial \phi}{\partial x}\Big|_{x=L} = 0$. The intention is often to allow the advected profile to leave the domain with minimal distortion, by assuming that the profile becomes flat as it exits. However, it's important to note that for pure advection, the physical flux is $F = u\phi$. A zero-gradient condition does not directly enforce zero flux unless ϕ itself happens to be zero at the boundary or $u = 0$. If not chosen carefully, Neumann conditions at outflow boundaries of advection problems can sometimes lead to artificial reflections of the solution back into the domain. More sophisticated "non-reflecting" or "radiation" boundary conditions are often used in advanced applications but are beyond our current scope.

7.2.2 Inflow and Outflow Boundaries for Hyperbolic Problems

For hyperbolic PDEs like the advection equation, the directionality of information flow along characteristic curves has a profound impact on how boundary conditions must be posed. Unlike parabolic (diffusive) problems where every boundary point influences the interior (and vice-versa), in hyperbolic problems, a strict distinction must be made between **inflow** and **outflow** boundaries.

The guiding principle is: *Boundary conditions must be specified at inflow boundaries, where characteristics enter the computational domain, but should not be specified at outflow boundaries, where characteristics leave the domain.*

Let's illustrate this for the 1D advection equation, $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$, on a finite domain $x \in [0, L]$. We assume a constant, positive advection velocity, $u > 0$.

- **At the left boundary ($x = 0$):** The characteristic curves, which have a positive slope $dx/dt = u$, are "coming from" outside the domain on the left and are continuously *entering* the domain at $x = 0$ for all times $t > 0$. This is an **inflow boundary**. The values of ϕ along these incoming characteristics are not determined by the initial condition that was defined inside the domain ($x \in [0, L]$ at $t = 0$). Therefore, to have a well-posed problem, we *must* specify a boundary condition at $x = 0$. This

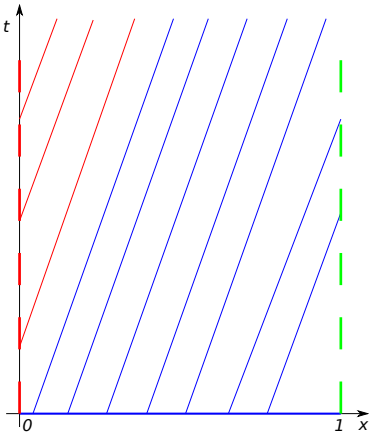


Figure 7.7. The role of characteristics in determining where to apply boundary conditions for a 1D advection problem on the domain $x \in [0, 1]$ with a constant positive velocity $u > 0$. **Red lines** are characteristics that enter the domain from the left (inflow) boundary, requiring a specified boundary condition. **Blue lines** are characteristics that originate from the bottom (initial) boundary, carrying the initial condition. The **green dashed lines** indicate that characteristics are leaving the domain at the right (outflow) boundary, where no boundary condition is needed or should be imposed.

condition, for example $\phi(0, t) = g_0(t)$, provides the necessary information that is "fed into" the domain over time.

- **At the right boundary ($x = L$):** The characteristic curves are continuously *exiting* the domain at $x = L$. This is an **outflow boundary**. The value of ϕ at $x = L$ is entirely determined by the information that has propagated from within the domain (originating from either the initial condition or the inflow boundary condition at $x = 0$). Imposing a condition at this boundary (e.g., forcing $\phi(L, t)$ to a specific value) would over-constrain the problem and would likely conflict with the solution that is naturally arriving at that point. This can lead to mathematical inconsistencies and severe numerical errors, such as artificial reflections of the solution back into the domain. Therefore, at a pure outflow boundary, no condition should be imposed; the numerical scheme should be designed to allow the solution to pass through freely.

Figure 7.7 provides a clear visual representation of this principle in the $x-t$ plane. The solution at any point (x, t) within the domain is found by tracing its characteristic curve backward in time.

- Characteristics that trace back to the initial line $t = 0$ (shown in blue) carry information from the **initial condition**.
- Characteristics that trace back to the inflow boundary $x = 0$ (shown in red) carry information from the **left boundary condition**.
- No information originates from the outflow boundary at $x = L$; characteristics simply pass through it (indicated by the green dashed lines).

This characteristic-based reasoning is fundamental for correctly setting up numerical simulations of hyperbolic systems. If the velocity u were negative, the roles of the boundaries would be reversed: $x = L$ would become the inflow boundary, and $x = 0$ would be the outflow boundary. For more complex systems with multiple characteristic speeds (like in gas dynamics) or with velocities that change sign within the domain, the analysis of inflow/outflow conditions at each boundary becomes more intricate but follows the same

core principle. An important special case that handles boundaries differently is the periodic boundary condition, which we will discuss next.

7.2.3 Periodic Boundary Conditions: Closing the Loop

A distinct and highly useful type of boundary condition, especially for testing numerical schemes for advection and for modeling certain types of physical systems, is the **Periodic Boundary Condition (PBC)**.

Definition and Concept. For a one-dimensional domain defined over an interval $x \in [0, L]$, periodic boundary conditions mathematically connect the two ends of the domain as if they were joined together to form a continuous loop or a circle. The primary condition is that the value of the advected quantity ϕ at one boundary is identical to its value at the other boundary at all times:

$$\phi(0, t) = \phi(L, t) \tag{7.5}$$

To ensure smoothness of the solution across this "join" and for consistency with numerical schemes that might use wider stencils (i.e., more than just the immediate neighbors) or that involve higher-order spatial derivatives, it is often also required that the spatial derivatives of ϕ match at the boundaries:

$$\frac{\partial \phi}{\partial x}(0, t) = \frac{\partial \phi}{\partial x}(L, t), \quad \frac{\partial^2 \phi}{\partial x^2}(0, t) = \frac{\partial^2 \phi}{\partial x^2}(L, t), \quad \text{and so on if needed.} \tag{7.6}$$

The intuitive picture is that any "signal" or feature in ϕ that is advected out of the domain at one end (say, $x = L$ if $u > 0$) immediately re-enters the domain at the other end ($x = 0$) with the same properties, as illustrated in Figure 7.8. In such a system, there are no "true" external boundaries that introduce or remove ϕ; the system is self-contained and its total content of ϕ (if u is constant or its divergence is zero over the domain) should be conserved.

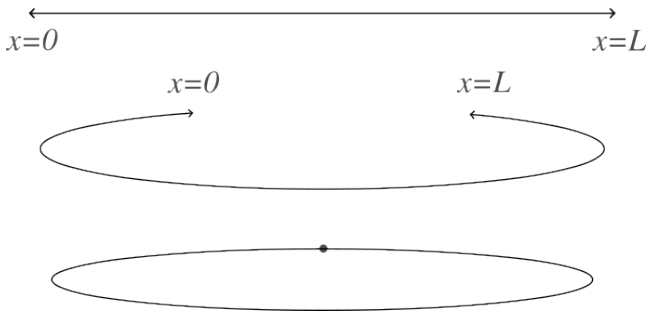


Figure 7.8. Conceptual illustration of periodic boundary conditions. The 1D domain from $x = 0$ to $x = L$ is effectively "wrapped" into a circle. A feature (e.g., a pulse of ϕ) exiting at $x = L$ (if $u > 0$) re-enters at $x = 0$, continuing its propagation through the domain cyclically.

Relevance in Geosciences and Numerical Testing

While few natural Earth systems are perfectly periodic in a simple 1D sense, periodic boundary conditions are a valuable tool and a reasonable approximation in several contexts:

- **Large-Scale Atmospheric and Oceanic Circulation.** When modeling phenomena that extend zonally around the Earth, such as the mean westerly winds, planetary-scale waves (e.g., Rossby waves), or major ocean currents like the Antarctic Circumpolar Current (Figure 7.9), the system naturally "reconnects" after 360° of longitude. Periodic BCs in the longitudinal direction are a standard choice for such global or large-scale regional models.
- **Idealized Laboratory Experiments.** Physical model experiments in fluid dynamics, such as studies of convection or sediment transport, are sometimes conducted in annular (ring-shaped) tanks or flumes. These experimental setups are designed to mimic periodic conditions in one dimension.
- **Modeling of Intrinsically Periodic Phenomena.** If the physical system being studied exhibits an inherent spatial periodicity (e.g., regularly spaced geological structures that influence flow, or some types of wave fields), PBCs might be used to represent a single repeating unit of this system.
- **Testing and Analysis of Numerical Schemes (Primary Use in this Context).** Perhaps the most significant application of PBCs in the context of learning numerical methods is for rigorously testing the intrinsic properties of different numerical schemes for advection. By creating a closed loop:
  - **Boundary Effects are Eliminated.** We can observe the behavior of a numerical scheme as it advects a feature (like a Gaussian pulse or a sharp square wave) over long distances or for many "laps" around the domain, without the solution being contaminated by artificial reflections or damping that can be introduced by fixed-value (Dirichlet) or fixed-gradient (Neumann) conditions at inflow/outflow boundaries.
  - **Focus on Scheme Properties.** This "clean" environment allows for a clearer assessment of the scheme's inherent numerical diffusion (artificial smearing of sharp features), numerical dispersion (generation of spurious oscillations or wiggles), phase speed errors (whether different wave components travel at the correct speed), and long-term stability.
  - **Conservation Studies.** For an advection equation where ϕ represents a conserved quantity (like mass), PBCs make it straightforward to check if the numerical scheme conserves the total amount of ϕ within the domain over time, as there are no net fluxes into or out of the "overall" system.

For these reasons, many standard benchmark problems used to evaluate and compare advection algorithms employ periodic boundary conditions.

Concept Check

Periodic Boundary Conditions (PBCs) are often used when testing numerical schemes for advection.

1. Explain in your own words the core idea behind PBCs for a 1D domain $x \in [0, L]$.
2. Why are PBCs particularly advantageous for observing numerical errors like diffusion or dispersion inherent to a scheme, compared to, say, fixed value (Dirichlet) inflow/outflow conditions?

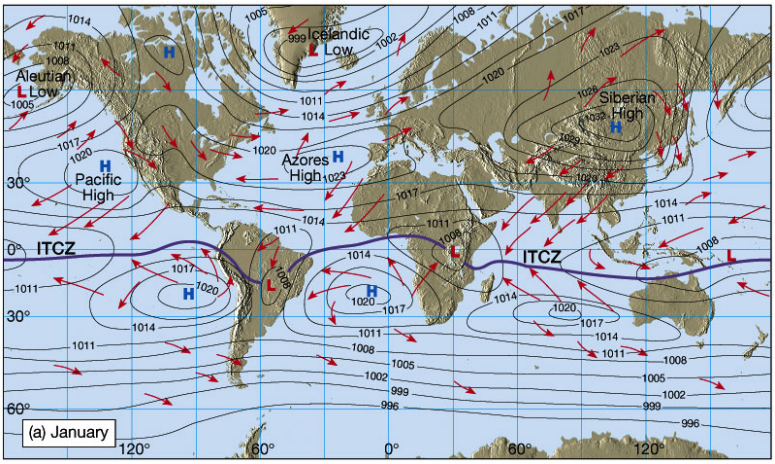


Figure 7.9. Global patterns of mean sea-level pressure and surface winds (example for January). Large-scale atmospheric phenomena, such as zonal jets or planetary waves that encircle the Earth, are often modeled using periodic boundary conditions in the longitudinal direction to represent this cyclical nature.

Numerical Implementation of Periodic Boundary Conditions

Implementing periodic boundary conditions within a finite difference scheme involves ensuring that the grid points at one end of the computational domain effectively "see" the grid points from the opposite end as their neighbors, whenever the numerical stencil (the set of points used in the difference formula) extends beyond the physical boundary.

Consider our standard 1D spatial grid with N_x points, indexed $i = 0, 1, \ldots, N_x - 1$. The physical domain extends from x_0 (node 0) to x_{N_x-1} (node $N_x - 1$). The spacing is Δx.

- When calculating the solution ϕ_0^{n+1} at the **leftmost physical node** ($i = 0$), some finite difference schemes (like a backward difference for $\partial\phi/\partial x$, or a central difference for $\partial^2\phi/\partial x^2$) might require a value from its "left" neighbor, which would conceptually be at index $i = -1$. With periodic BCs, this "ghost" node x_{-1} takes its value from the *last physical node* of the domain, x_{N_x-1}. So, we effectively set $\phi_{-1}^n \equiv \phi_{N_x-1}^n$.
- Similarly, when calculating $\phi_{N_x-1}^{n+1}$ at the **rightmost physical node** ($i = N_x - 1$), if the stencil needs a value from its "right" neighbor (conceptually at x_{N_x}), periodicity implies this value is taken from the *first physical node* of the domain, x_0. So, we effectively set $\phi_{N_x}^n \equiv \phi_0^n$.

This "wrapping around" of indices can be managed in Python implementations in a couple of common ways, often involving the concept of "ghost cells" or by directly using **modulo arithmetic**.

Modulo Arithmetic for Periodic Indexing. A very convenient and common technique for handling periodic boundaries directly within array indexing, especially when using NumPy arrays, is to employ **modulo arithmetic**. The **modulo operator** in Python is represented by the percent sign (`%`). For two integers `a` and `b` (where `b` is non-zero), the expression `a % b` calculates the *remainder* of the division of `a` by `b`.

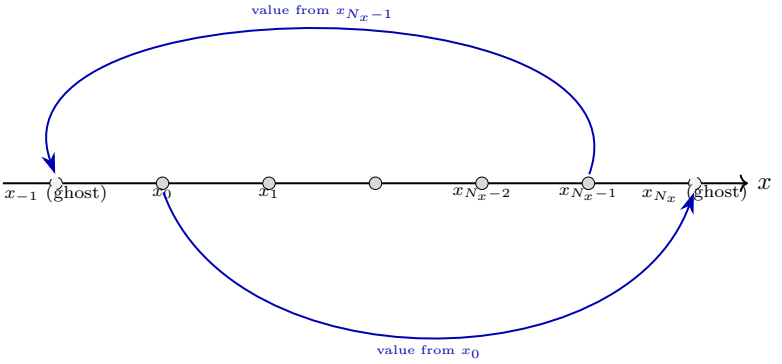


Figure 7.10. Numerical implementation of periodic boundary conditions on a 1D grid of N_x points (indexed 0 to N_x-1). For computations at node 0 that require a point to its left (stencil extending to index -1), the value from node N_x-1 is used. Similarly, for computations at node N_x-1 requiring a point to its right (stencil extending to index N_x), the value from node 0 is used. This creates a cyclical connection, as if the array is wrapped into a loop.

Let's see some examples:

- `7 % 3` evaluates to `1` (because $7 = 2\times 3+1$).
- `10 % 5` evaluates to `0` (because $10 = 2\times 5+0$).
- `0 % 5` evaluates to `0`.
- Importantly for negative numbers in Python: `-1 % 5` evaluates to `4` (Python's `%` operator ensures the result has the same sign as the divisor; so, $-1 = (-1)\times 5+4$).
- `-2 % 5` evaluates to `3`.

How does this help with periodic indexing on a grid with N_x points, indexed from 0 to N_x-1? For a loop iterating `iNode` from 0 to `numXPoints - 1`:

- The upwind neighbor (from the left, for $u > 0$) can be found using `iMinus1 = (iNode - 1) % numXPoints`.
- The downwind neighbor (to the right, for $u > 0$) can be found using `iPlus1 = (iNode + 1) % numXPoints`.

This use of the modulo operator provides an elegant way to implement the "wrapping around" logic for periodic boundaries directly within the indexing of our solution arrays, as we will see in upcoming Python examples.

Ghost Cells. Another common technique, particularly in more complex codes or when dealing with higher-order schemes that might require points further out (e.g., $i-2, i+2$), is the use of **ghost cells** (or ghost nodes). These are additional cells/nodes added computationally at each end of the physical domain. Before each time step (or each stage of a time-stepping method), the values in these ghost cells are filled according to the specified boundary condition.

- For periodic BCs, the ghost cell at x_{-1} would be filled with the value from x_{N_x-1}, and the ghost cell at x_{N_x} would be filled with the value from x_0.
- The main computational loop then often iterates only over the physical grid points, but the finite difference stencils at the boundaries can safely access the ghost cells, which already contain the correct periodic values.

While ghost cells add a little more setup, they can sometimes simplify the logic within the main computational loop, as every physical point can then use the same stencil formula. For the simple schemes we are considering, direct modulo arithmetic is often sufficient and very efficient.

By implementing periodic boundary conditions, we create a computational domain that has no "ends" in the traditional sense, allowing advected features to pass through the nominal boundaries and re-emerge on the opposite side. This is extremely useful for studying the long-term behavior of numerical schemes and their fidelity in transporting various profiles.

7.3 Finite Difference Schemes for Linear Advection: Formulation, Implementation, and Initial Observations

Having established the 1D linear advection equation and discussed the importance of boundary conditions, particularly periodic ones for testing purposes, we now turn to developing **finite difference schemes** to approximate its solution numerically. Our goal is to transform the continuous PDE into a set of algebraic equations that can be solved iteratively on a computer.

7.3.1 Recap: Discretizing Derivatives

As a foundation for constructing our numerical schemes, let us briefly recall the key finite difference approximations we will employ. For a function $\phi(x,t)$, discretized on a grid with spatial step Δx and time step Δt, such that $\phi_i^n \approx \phi(x_i, t^n)$:

Time Derivative $\frac{\partial \phi}{\partial t}$. For the explicit schemes we will consider in this chapter, the time derivative at grid point (i,n) is approximated using a **Forward Difference in Time (FDT)**, also known as a Forward Euler step:

$$\left.\frac{\partial \phi}{\partial t}\right|_i^n \approx \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} \tag{7.7}$$

This approximation has a local truncation error of $\mathcal{O}(\Delta t)$, making it first-order accurate in time.

First Spatial Derivative $\frac{\partial \phi}{\partial x}$. For the spatial derivative term $u\frac{\partial \phi}{\partial x}$ in the advection equation, we have several choices for approximating $\frac{\partial \phi}{\partial x}$ at grid point (i,n) using values from time level n:

- **Forward Difference in Space (FDS):**

$$\left.\frac{\partial \phi}{\partial x}\right|_i^n \approx \frac{\phi_{i+1}^n - \phi_i^n}{\Delta x} \quad \text{(Order } \mathcal{O}(\Delta x)\text{)}$$

- **Backward Difference in Space (BDS):**

$$\left.\frac{\partial \phi}{\partial x}\right|_i^n \approx \frac{\phi_i^n - \phi_{i-1}^n}{\Delta x} \quad (\text{Order } \mathcal{O}(\Delta x))$$

- **Central Difference in Space (CDS):**

$$\left.\frac{\partial \phi}{\partial x}\right|_i^n \approx \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} \quad (\text{Order } \mathcal{O}(\Delta x^2))$$

The combination of the Forward Euler time discretization with one of these spatial discretizations will define a specific numerical scheme. We will start by examining the scheme that uses a central difference for the spatial derivative.

7.3.2 Scheme 1: Forward Time, Centered Space (FTCS)

The **Forward-Time Central-Space (FTCS)** scheme is constructed by applying the Forward Euler method for the time derivative (Equation 7.7) and the Central Difference approximation for the spatial derivative to the 1D linear advection equation, $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$. Substituting these approximations, we get:

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + u\left(\frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x}\right) = 0 \tag{7.8}$$

This equation relates the unknown value ϕ_i^{n+1} at the new time level $n+1$ to known values at the current time level n. Since ϕ_i^{n+1} can be isolated directly, this is an explicit scheme. Solving for ϕ_i^{n+1}:

$$\phi_i^{n+1} = \phi_i^n - u\frac{\Delta t}{2\Delta x}(\phi_{i+1}^n - \phi_{i-1}^n) \tag{7.9}$$

It is common to introduce the dimensionless **Courant number** (or Courant-Friedrichs-Lewy number, CFL number), C, defined as:

$$C = \frac{u\Delta t}{\Delta x} \tag{7.10}$$

(Note: sometimes $|u|$ is used if u can be negative, but for deriving the scheme with a given u, this form is common). Using the Courant number, the FTCS update formula becomes:

$$\boxed{\phi_i^{n+1} = \phi_i^n - \frac{C}{2}(\phi_{i+1}^n - \phi_{i-1}^n)} \tag{7.11}$$

This formula is applied to update the solution at all interior grid points i. The values at the boundary points are handled by the chosen boundary conditions.

Stencil and Formal Accuracy. The FTCS scheme uses values from three points at time level n ($\phi_{i-1}^n, \phi_i^n, \phi_{i+1}^n$) to compute the value at one point at time level $n+1$ (ϕ_i^{n+1}), as illustrated by its computational stencil in Figure 7.11.

The local truncation error of the FTCS scheme, found by substituting the Taylor series expansions for ϕ_i^{n+1}, ϕ_{i+1}^n, and ϕ_{i-1}^n back into Equation 7.8 and comparing with the original PDE, can be shown to be $\mathcal{O}(\Delta t, (\Delta x)^2)$. This means the scheme is formally first-order accurate in time and second-order accurate in space. The second-order spatial accuracy might initially seem appealing, as it suggests that the spatial error will decrease rapidly with refinement of the spatial grid. However, as we will see, this formal accuracy is overshadowed by a critical issue with this scheme when applied to the pure advection equation.

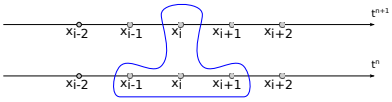


Figure 7.11. Computational stencil for the FTCS scheme applied to the 1D advection equation. The value at node i at time level $n+1$ (top point) depends on the values at nodes $i-1, i$, and $i+1$ at time level n (bottom points).

Python Implementation and Initial Observation of FTCS

Let's implement the FTCS scheme in Python to advect an initial profile, for example, a square wave, using periodic boundary conditions. The full script, including the setup of parameters, grid, initial condition, the time-stepping loop with FTCS updates, and plotting, is provided in Listing 7.2. We will use the analytical solution function developed in Section 7.1.3 for comparison.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# print("--- FTCS Scheme for 1D Linear Advection ---") #
    Optional

# Assume initialConditionSquareWave and
    analyticalSolutionPeriodic
# are defined as in Listing \ref{lst:
    py_advection_analytical_solution_v6_full}

def initialConditionSquareWave(x, xStart=0.25, xEnd=0.75,
    phiLow=0.0, phiHigh=1.0): # Re-define if not global
    phi0 = np.full_like(x, phiLow)
    phi0[(x >= xStart) & (x <= xEnd)] = phiHigh
    return phi0

def analyticalSolutionPeriodic(xGrid, t, u, L, phi0Func,
    xStartIC, xEndIC): # Added IC params
    xEffectiveSource = (xGrid - u * t) % L
    return phi0Func(xEffectiveSource, xStart=xStartIC, xEnd=
    xEndIC) # Pass IC params

# --- 1. Simulation Parameters ---
lengthDomain = 1.0
numXPoints = 101 # Changed from 201 for potentially faster
    instability with fewer points
dxStep = lengthDomain / (numXPoints -1) if numXPoints > 1 else
     lengthDomain
advectionVelocity = 1.0

courantNumber = 0.40 # Courant number (FTCS is unstable, but
    we test a value)
# For FTCS, even small CFL leads to instability for pure
    advection
```

```
deltaTime = courantNumber * dxStep / abs(advectionVelocity) if
     abs(advectionVelocity) > 1e-9 else 1e-3
timeFinal = 0.050  # Short time to see instability develop
numTimeSteps = int(timeFinal / deltaTime) if deltaTime > 0
    else 0

print(f"FTCS - Domain: L={lengthDomain}, N_x={numXPoints}, dx
    ={dxStep:.4f}")
print(f"FTCS - Velocity: u={advectionVelocity}, CFL={
    courantNumber:.2f}, dt={deltaTime:.4e}, Time={timeFinal:.3f
    }, Steps={numTimeSteps}")

# --- 2. Grid and Initial Condition ---
xGridFTCS = np.linspace(0, lengthDomain, numXPoints, endpoint=
    True)
# Define initial position of the square wave
xInitialStartFTCS = 0.15 # Adjusted for a domain [0,1] to see
    movement
xInitialEndFTCS = 0.35

phiInitialFTCS = initialConditionSquareWave(xGridFTCS, xStart=
    xInitialStartFTCS, xEnd=xInitialEndFTCS)

# Analytical solution at tFinal for comparison
phiAnalyticalFinalFTCS = analyticalSolutionPeriodic(xGridFTCS,
     timeFinal, advectionVelocity,
                                                    lengthDomain,
     initialConditionSquareWave,
                                                    xStartIC=
    xInitialStartFTCS, xEndIC=xInitialEndFTCS)

# --- 3. Initialization for FTCS ---
phiCurrent = phiInitialFTCS.copy()    # Current solution array
phiNew = np.zeros_like(phiCurrent)    # Array for the solution
     at the next time step

# --- 4. Time-stepping Loop (FTCS with Periodic BCs) ---
if numTimeSteps > 0:
    for nLoopIndex in range(numTimeSteps):
        phiOldStep = phiCurrent.copy() # Values from time
    level 'n' for RHS

        for iNode in range(numXPoints): # Loop over ALL points
     0 to nx-1 for periodic
            # Determine periodic indices for neighbors
            iMinus1 = (iNode - 1 + numXPoints) % numXPoints #
    Handles iNode=0
            iPlus1 = (iNode + 1) % numXPoints      # Handles
    iNode=nx-1

            # FTCS formula using values from phiOldStep
            phiNew[iNode] = phiOldStep[iNode] - \
                            courantNumber / 2.0 * \
                            (phiOldStep[iPlus1] - phiOldStep[
    iMinus1])
            # Note: courantNumber = advectionVelocity *
    deltaTime / dxStep
```

```
        phiCurrent = phiNew.copy() # Update solution for the
    next time step
else:
    phiCurrent = phiInitialFTCS.copy()

# --- 5. Plotting Results ---
plt.figure(figsize=(10, 6))
plt.plot(xGridFTCS, phiInitialFTCS, label='Initial Condition (
    t=0)', linestyle=':', color='gray', lw=2)
plt.plot(xGridFTCS, phiCurrent, label=f'FTCS Numerical (t={
    timeFinal:.3f}, CFL={courantNumber:.2f})',
         marker='.', markersize=5, linestyle='-')
plt.plot(xGridFTCS, phiAnalyticalFinalFTCS, label=f'Analytical
     (t={timeFinal:.3f})',
         linestyle='--', color='red', lw=2)

plt.title(f'1D Linear Advection with FTCS Scheme (Periodic BCs
    )')
plt.xlabel('Spatial domain x')
plt.ylabel('$\phi(x,t)$')
plt.legend()
plt.grid(True)
plt.ylim(-1.2, 2.2) # Adjust ylim to better see instability
plt.show()

if numTimeSteps > 0 and (np.any(np.isnan(phiCurrent)) or np.
    any(np.abs(phiCurrent) > 2.5*np.max(np.abs(phiInitialFTCS))
     if np.max(np.abs(phiInitialFTCS)) > 1e-9 else 1.0 )):
    print("Instability strongly suggested by FTCS solution (
    NaNs or large growth)!")
else:
    print("FTCS simulation completed. Visually inspect plot
    for stability.")
```

Listing 7.2. Python implementation of the FTCS scheme for 1D linear advection with periodic boundary conditions, using a square wave initial condition.

When this Python script is executed, even with a Courant number C that might seem reasonable for other schemes (e.g., $C = 0.4$), the FTCS method typically demonstrates problematic behavior for the pure advection equation. Figure 7.12 shows an example of the output.

Initial Observation: As clearly visible in Figure 7.12, the FTCS scheme, when applied to the 1D linear advection equation, produces a solution riddled with rapidly growing oscillations. These oscillations are not physical and quickly render the numerical result useless. This behavior is a classic manifestation of **numerical instability**. Despite its second-order spatial accuracy on paper, the FTCS scheme is, in fact, unconditionally unstable for the pure advection problem. This dramatic failure motivates the search for alternative, stable numerical schemes.

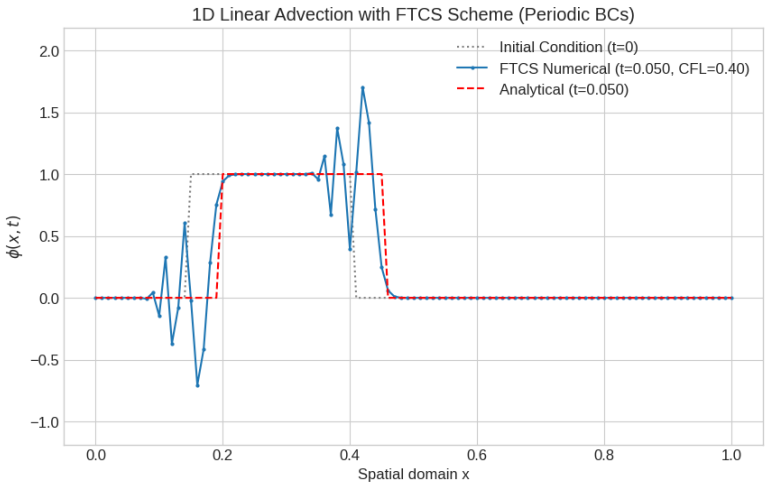


Figure 7.12. Numerical solution of 1D linear advection of an initial square wave (dotted gray line) using the FTCS scheme with periodic boundary conditions, shown after a short time ($t = 0.050$). The analytical solution is the dashed red line. The FTCS solution (blue line with markers) exhibits severe, unphysical oscillations that are growing in amplitude, indicative of numerical instability. Parameters: $L = 1.0, N_x = 101, u = 1.0, CFL = 0.40$.

Concept Check

We observed that the FTCS scheme, $\phi_i^{n+1} = \phi_i^n - \frac{C}{2}(\phi_{i+1}^n - \phi_{i-1}^n)$, is unconditionally unstable for the pure linear advection equation. This scheme is, however, conditionally stable for the Heat (Diffusion) Equation (recall Chapter 6). Qualitatively, why might a scheme that uses a centered spatial difference be problematic for a first-order hyperbolic (advective) PDE but potentially suitable for a second-order parabolic (diffusive) PDE when combined with a forward time step? (Hint: Think about the directionality of information flow in advection versus diffusion.)

7.3.3 Scheme 2: Upwind Schemes - Aligning with the Flow Direction

The dramatic failure of the FTCS scheme for pure advection, despite its seemingly reasonable second-order spatial accuracy, underscores a critical point: for hyperbolic problems where information propagates in a well-defined direction, numerical schemes that do not respect this directionality can suffer severe stability issues. This motivates the development of **upwind schemes**.

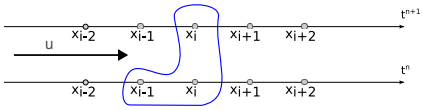
The Upwind Principle. The core idea behind upwind differencing is to approximate the spatial derivative $\frac{\partial \phi}{\partial x}$ using information taken from the "upwind" (or "upstream") direction relative to the flow velocity u. This means:

- If the flow is from left to right ($u > 0$), the value of ϕ at a grid point x_i is primarily

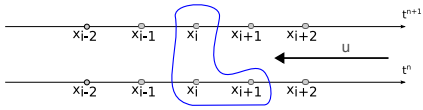
influenced by the value of ϕ at x_{i-1} (its left neighbor). Thus, a backward difference in space is appropriate.

- If the flow is from right to left ($u < 0$), the value of ϕ at x_i is primarily influenced by the value of ϕ at x_{i+1} (its right neighbor). In this case, a forward difference in space is used.

This approach attempts to mimic the physical process where the quantity ϕ is carried *from* the upwind direction *to* the point of interest. Figure 7.13 illustrates the stencils for these two cases.



(a) Upwind for $u > 0$: Backward Difference in Space (FTBS). Value at x_i^{n+1} depends on x_i^n and x_{i-1}^n.



(b) Upwind for $u < 0$: Forward Difference in Space (FTFS). Value at x_i^{n+1} depends on x_i^n and x_{i+1}^n.

Figure 7.13. Computational stencils for first-order upwind schemes. The spatial difference is taken in the direction opposite to the flow velocity u.

Forward Time, Backward Space (FTBS) Scheme (for $u > 0$)

Let's first consider the case where the advection velocity u is positive ($u > 0$), meaning the flow is directed towards increasing x (from left to right). We combine the Forward Euler approximation for the time derivative (Equation 7.7) with a **Backward Difference in Space (BDS)** for the spatial derivative $\frac{\partial \phi}{\partial x}$, evaluated at (x_i, t^n):

$$\left.\frac{\partial \phi}{\partial x}\right|_i^n \approx \frac{\phi_i^n - \phi_{i-1}^n}{\Delta x}$$

Substituting these into the linear advection equation, $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$, gives:

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + u\left(\frac{\phi_i^n - \phi_{i-1}^n}{\Delta x}\right) = 0 \tag{7.12}$$

Solving for ϕ_i^{n+1} yields the update formula for the FTBS scheme:

$$\begin{aligned} \phi_i^{n+1} &= \phi_i^n - u\frac{\Delta t}{\Delta x}(\phi_i^n - \phi_{i-1}^n) \\ \phi_i^{n+1} &= \phi_i^n - C(\phi_i^n - \phi_{i-1}^n) \end{aligned} \tag{7.13}$$

where $C = u\Delta t/\Delta x$ is the Courant number (positive since $u > 0$). This can also be rewritten as:

$$\boxed{\phi_i^{n+1} = (1-C)\phi_i^n + C\phi_{i-1}^n} \tag{7.14}$$

This form highlights that ϕ_i^{n+1} is a linear combination of ϕ_i^n and its upwind neighbor ϕ_{i-1}^n.

Properties of FTBS:

- **Explicit:** ϕ_i^{n+1} is calculated directly from known values at time level n.
- **Stencil:** Uses values from two points at time level n (ϕ_i^n and ϕ_{i-1}^n) to compute ϕ_i^{n+1} (as shown in Figure 7.13a).
- **Truncation Error:** The local truncation error for the FTBS scheme is $\mathcal{O}(\Delta t, \Delta x)$. It is first-order accurate in both time and space. This lower formal spatial accuracy compared to FTCS ($\mathcal{O}(\Delta x^2)$) might seem like a disadvantage, but we will see that its stability properties are far superior for advection.

Python Implementation and Initial Observation of FTBS (Upwind $u > 0$)

We now implement the FTBS scheme in Python to advect the same initial square wave profile as used for the FTCS test, again employing periodic boundary conditions. The Courant number C will be a key parameter to observe. Listing 7.3 provides the Python code.

```
# --- Upwind (FTBS for u > 0) Scheme Implementation and Test
    ---
import numpy as np
import matplotlib.pyplot as plt

# Assume initialConditionSquareWave and
    analyticalSolutionPeriodic are defined
# as in Listing \ref{lst:
    py_advection_analytical_solution_v6_full}

def initialConditionSquareWave(x, xStart=0.25, xEnd=0.75,
    phiLow=0.0, phiHigh=1.0): # Re-define if not global
    phi0 = np.full_like(x, phiLow)
    phi0[(x >= xStart) & (x <= xEnd)] = phiHigh
    return phi0

def analyticalSolutionPeriodic(xGrid, t, u, L, phi0Func,
    xStartIC, xEndIC): # Added IC params
    xEffectiveSource = (xGrid - u * t) % L
    return phi0Func(xEffectiveSource, xStart=xStartIC, xEnd=
    xEndIC) # Pass IC params

# --- 1. Simulation Parameters ---
lengthDomainUpwind = 1.0
numXPointsUpwind = 101
dxStepUpwind = lengthDomainUpwind / (numXPointsUpwind - 1)
xGridUpwind = np.linspace(0, lengthDomainUpwind,
    numXPointsUpwind)

advectionVelocityUpwind = 1.0 # Advection velocity (MUST BE >
    0 for this FTBS)
courantNumberUpwind = 0.8     # Courant number (should be <= 1
    for stability)
deltaTimeUpwind = courantNumberUpwind * dxStepUpwind /
    advectionVelocityUpwind # u is positive
timeFinalUpwind = 0.5
numTimeStepsUpwind = int(timeFinalUpwind / deltaTimeUpwind)
```

```
print(f"\nUpwind (FTBS) - Domain: L={lengthDomainUpwind}, Nx={
    numXPointsUpwind}, dx={dxStepUpwind:.4f}")
print(f"Upwind (FTBS) - Velocity: u={advectionVelocityUpwind},
     CFL={courantNumberUpwind:.2f}, dt={deltaTimeUpwind:.4e},
    Time={timeFinalUpwind:.3f}, Steps={numTimeStepsUpwind}")

# --- 2. Initial Condition ---
xInitialStartUpwind = 0.1 # Adjusted IC position
xInitialEndUpwind = 0.4
phiCurrentUpwind = initialConditionSquareWave(xGridUpwind,
    xStart=xInitialStartUpwind, xEnd=xInitialEndUpwind)
phiInitialUpwind = phiCurrentUpwind.copy()

# --- 3. Time-stepping Loop for Upwind (FTBS) ---
phiNewUpwind = np.zeros_like(phiCurrentUpwind) # Array for
    solution at n+1

for nLoopIndex in range(numTimeStepsUpwind):
    phiOldStepUpwind = phiCurrentUpwind.copy() # Values from
    time level 'n'

    # Apply FTBS scheme with Periodic Boundary Conditions
    for iNode in range(numXPointsUpwind): # Loop through all
    physical points
        # Determine index for phi[i-1] with periodicity
        iMinus1 = (iNode - 1 + numXPointsUpwind) %
    numXPointsUpwind # Modulo for wrap-around

        phiNewUpwind[iNode] = phiOldStepUpwind[iNode] - \
                              courantNumberUpwind * (
    phiOldStepUpwind[iNode] - phiOldStepUpwind[iMinus1])

    phiCurrentUpwind = phiNewUpwind.copy() # Update solution
    for the next time step

# --- 4. Analytical Solution for Comparison ---
phiAnalyticalFinalUpwind = analyticalSolutionPeriodic(
    xGridUpwind, timeFinalUpwind,

    advectionVelocityUpwind, lengthDomainUpwind,

    initialConditionSquareWave,
                                                    xStartIC
    =xInitialStartUpwind, xEndIC=xInitialEndUpwind)

# --- 5. Plotting Upwind (FTBS) Result ---
plt.figure(figsize=(10, 6))
plt.plot(xGridUpwind, phiInitialUpwind, 'k-', lw=1.5, label='
    Initial Condition (t=0)')
plt.plot(xGridUpwind, phiAnalyticalFinalUpwind, 'g--', lw=2,
    label='Analytical Solution')
plt.plot(xGridUpwind, phiCurrentUpwind, 'b.-', markersize=5,
         label=f'Upwind (FTBS, CFL={courantNumberUpwind:.2f})'
    )
plt.xlabel('Position x')
plt.ylabel('$\phi$')
```

```
plt.title(f'Upwind (FTBS) Scheme for Advection (t={
    timeFinalUpwind:.2f})')
plt.legend()
plt.grid(True)
plt.ylim(-0.1, 1.1)
plt.show()
```

Listing 7.3. Python implementation of the Upwind (FTBS, $u > 0$) scheme for 1D linear advection with periodic BCs, using a square wave initial condition.

When the Python script in Listing 7.3 is executed with a Courant number $C \leq 1$ (e.g., $C = 0.8$), the numerical solution behaves very differently from the FTCS scheme. As shown in Figure 7.14, the upwind scheme is stable.

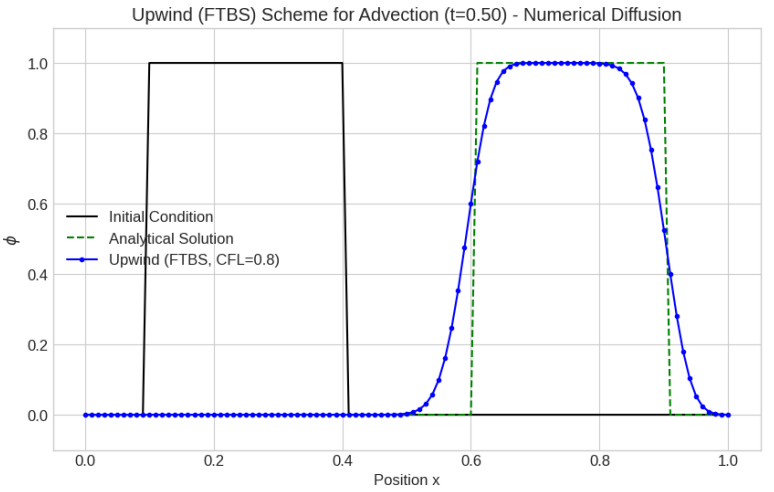


Figure 7.14. Numerical solution of 1D linear advection of an initial square wave (black solid line) using the first-order Upwind (FTBS) scheme with periodic boundary conditions ($u = 1.0$, CFL=0.8, $t = 0.5$). The analytical solution is the dashed green line. The Upwind solution (blue line with markers) is stable and advects the profile correctly, but exhibits significant numerical diffusion, causing the sharp edges of the square wave to be smeared out and the peak amplitude to be slightly reduced.

Initial Observation: The Upwind (FTBS) scheme successfully advects the initial profile across the domain without the catastrophic instabilities seen with FTCS (provided the Courant number condition, $0 \leq C \leq 1$, is met, which we will formalize later). However, a new numerical artifact is evident: the sharp edges of the square wave have been significantly smeared out, and the "corners" are rounded. This phenomenon is known as **numerical diffusion** or **artificial dissipation**. While the scheme is stable, its first-order accuracy in space introduces an error term that behaves much like a physical diffusion term, causing a loss of sharpness in the advected profile. This is a common characteristic and a primary drawback of simple first-order upwind schemes.

Upwind Scheme for $u < 0$: Forward Time, Forward Space (FTFS)

For completeness, if the advection velocity u is negative ($u < 0$, meaning flow is to the left), the upwind principle dictates that information for node x_i comes from its right neighbor, x_{i+1}. In this case, the appropriate one-sided spatial difference is a **Forward Difference in Space (FDS)**:

$$\left.\frac{\partial \phi}{\partial x}\right|_i^n \approx \frac{\phi_{i+1}^n - \phi_i^n}{\Delta x}$$

Combining this with Forward Euler in time gives the **FTFS (or 1st Order Upwind for $u < 0$)** scheme:

$$\phi_i^{n+1} = \phi_i^n - u\frac{\Delta t}{\Delta x}(\phi_{i+1}^n - \phi_i^n)$$
$$\phi_i^{n+1} = \phi_i^n - C(\phi_{i+1}^n - \phi_i^n) \tag{7.15}$$

where $C = u\Delta t/\Delta x$ will now be negative (since $u < 0$). This scheme can also be rewritten as:

$$\boxed{\phi_i^{n+1} = (1+C)\phi_i^n - C\phi_{i+1}^n} \tag{7.16}$$

The FTFS scheme shares similar properties with FTBS: it is first-order accurate ($\mathcal{O}(\Delta t, \Delta x)$), conditionally stable (requiring $0 \le -C \le 1$, or equivalently $|C| \le 1$), and exhibits numerical diffusion. The choice between FTBS and FTFS depends solely on the sign of the advection velocity u to ensure that the spatial differencing is always taken from the upwind direction.

7.4 Stability Analysis and Courant-Friedrichs-Lewy (CFL) Condition

Our initial numerical experiments in Section 7.3 provided a stark contrast in the behavior of different finite difference schemes applied to the linear advection equation. The Forward-Time Centered-Space (FTCS) scheme, despite its seemingly higher formal accuracy in space, produced wildly unstable and physically meaningless results (Figure 7.12). In contrast, the first-order Upwind scheme, while introducing some smearing of the profile (numerical diffusion, as seen in Figure 7.14), yielded a stable solution that correctly propagated the initial feature. This dramatic difference underscores the paramount importance of **numerical stability** in any computational modeling endeavor.

7.4.1 What is Numerical Stability? A Critical Requirement

In the context of numerical methods for solving differential equations, **stability** refers to the property of a scheme whereby small errors introduced at any stage of the computation do not grow uncontrollably as the simulation progresses. These initial errors can arise from various sources:

- **Truncation errors:** These are inherent to the finite difference approximations themselves, as we are truncating Taylor series.
- **Round-off errors:** These are due to the finite precision with which computers represent real numbers.
- **Perturbations in initial or boundary data:** Small uncertainties in the input conditions.

A **stable numerical scheme** ensures that such errors remain bounded, or ideally, decay over time. If a scheme is stable, then as the step sizes (Δt and Δx) approach zero, the numerical solution should converge to the true solution of the PDE (this is related to the Lax Equivalence Theorem, which states that for consistent linear schemes, stability is equivalent to convergence, though a formal discussion is beyond our current scope).

Conversely, an **unstable numerical scheme** will amplify these small errors. Even if the initial error is tiny, it can grow exponentially with each time step, leading to a numerical solution dominated by spurious oscillations or absurdly large (or small) values that bear no resemblance to the physical reality being modeled. The FTCS scheme applied to the advection equation is a classic example of an unconditionally unstable scheme; no matter how small we make Δt and Δx (while keeping their ratio finite), errors will eventually grow and destroy the solution. Therefore, ensuring the stability of a chosen numerical method is a non-negotiable prerequisite for obtaining reliable and meaningful simulation results.

7.4.2 The Courant Number: Linking Space, Time, and Velocity

For many explicit finite difference schemes applied to hyperbolic PDEs like the advection equation, stability is intimately linked to a dimensionless parameter known as the **Courant-Friedrichs-Lewy (CFL) number**, or more commonly, the **Courant number**. For the 1D linear advection equation with constant velocity u, the Courant number, typically denoted by C or ν (nu), is defined as:

$$C = \frac{|u|\Delta t}{\Delta x} \tag{7.17}$$

Here, $|u|$ is the magnitude of the advection velocity, Δt is the time step, and Δx is the spatial grid spacing.

Physical Interpretation of the Courant Number. The Courant number has a very intuitive physical interpretation:

- The term $|u|\Delta t$ represents the **physical distance** that a piece of information (or a point on the profile ϕ) travels due to advection in a single time step Δt.
- The Courant number C is therefore the ratio of this physical travel distance to the spatial grid spacing Δx.
- In other words, C tells us **how many grid cells (Δx) the physical signal or information traverses during one time step (Δt)**.

For example (see Figure 7.15):

- If $C = 0.5$, the information travels a distance equivalent to half a grid cell in one time step.
- If $C = 1.0$, the information travels exactly one full grid cell in one time step.
- If $C = 1.2$, the information travels a distance of 1.2 grid cells in one time step.

As we will see, the requirement that C be less than or equal to some maximum value (often 1) is central to the stability of explicit advection schemes. This makes intuitive sense: if the information physically "jumps" over more than one grid cell in a single time step ($C > 1$), an explicit numerical scheme that typically only uses information from immediately adjacent grid cells to compute the solution at the next time step might "miss" this information, leading to instability.

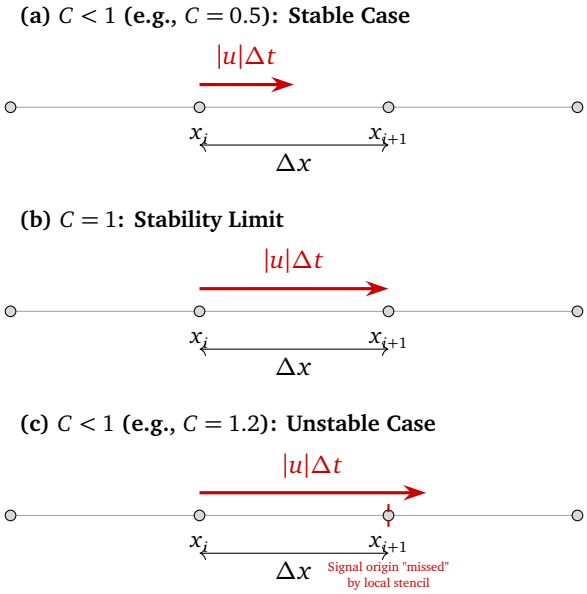


Figure 7.15. Conceptual illustration of the Courant number $C = |u|\Delta t/\Delta x$. It represents the distance traveled by a physical signal in one time step Δt, measured in units of grid cells Δx. (a) For $C < 1$, the signal's origin (the start of the red arrow) lies within the stencil of a local explicit scheme centered at x_i. (b) At the limit $C = 1$, the signal originates exactly at the upwind node x_i. (c) For $C > 1$, the signal originates from beyond the adjacent grid cell, and a numerical scheme using only immediate neighbors will "miss" this information, leading to instability.

7.4.3 The Courant-Friedrichs-Lewy (CFL) Condition

The **Courant-Friedrichs-Lewy (CFL) condition** is a fundamental principle that governs the stability of many explicit numerical schemes for hyperbolic PDEs. In its general form, it states that *for a numerical scheme to be stable, its numerical domain of dependence must contain the physical (or true) domain of dependence of the PDE.*

Let's break down what this means in our context:

- The **physical domain of dependence** for the solution at a point (x_i, t^{n+1}) refers to the set of points at the earlier time t^n that have a direct physical influence on the solution at (x_i, t^{n+1}). For the 1D linear advection equation, we know from the method of characteristics that the solution at (x_i, t^{n+1}) is determined *only* by the value of the initial profile at a single point: $x_i - u\Delta t$. This single point is the physical domain of dependence.
- The **numerical domain of dependence** refers to the set of grid points at time t^n that are actually used by the numerical scheme to compute the new value ϕ_i^{n+1}. This is simply the scheme's **computational stencil** (see Figures 7.11 and 7.13). For instance, for the FTCS scheme, the numerical domain of dependence for ϕ_i^{n+1} consists of the points $\{x_{i-1}, x_i, x_{i+1}\}$ at time t^n. For the first-order Upwind (FTBS, $u > 0$) scheme, it consists of $\{x_{i-1}, x_i\}$.

The CFL condition, therefore, essentially requires that the physical origin of the information, $x_i - u\Delta t$, must lie within the span of the computational stencil being used at time t^n.

Let's look at this visually for an upwind scheme (Figure 7.13). The scheme for ϕ_i^{n+1} uses information from the interval $[x_{i-1}, x_i]$. The true information comes from the point $x_i - u\Delta t$.

- If $C = u\Delta t/\Delta x \leq 1$, then $u\Delta t \leq \Delta x$. This means $x_i - u\Delta t \geq x_i - \Delta x = x_{i-1}$. The true origin point lies within the numerical stencil $[x_{i-1}, x_i]$, and the scheme has access to the correct information (even if it approximates it).
- If $C > 1$, then $u\Delta t > \Delta x$. This means $x_i - u\Delta t < x_{i-1}$. The true origin point lies *outside* the numerical stencil. The scheme is trying to compute a new value at x_i without using any information from where that value physically came from. It is "blind" to the true source of the signal, which intuitively leads to instability.

For many common explicit schemes for the 1D linear advection equation, this geometric condition translates directly to a constraint on the Courant number:

$$C = \frac{|u|\Delta t}{\Delta x} \leq C_{max} \tag{7.18}$$

where C_{max} is the maximum allowable Courant number for that specific scheme to be stable. For many simple advection schemes like first-order Upwind, $C_{max} = 1$. Violating this condition typically leads to instability, as the numerical method loses the ability to correctly track the physical propagation of information. However, satisfying it is not always sufficient for good accuracy (as numerical diffusion might still be large) or even for stability in more complex problems or for all schemes (as we saw with FTCS for advection).

7.4.4 Stability of Specific Advection Schemes

Having introduced the general principle of the CFL condition, let's revisit the explicit schemes we have discussed and analyze their stability properties more formally, though still aiming for an intuitive understanding rather than a rigorous mathematical proof for every case.

FTCS (Forward Time, Centered Space) Scheme

The FTCS update formula is $\phi_i^{n+1} = \phi_i^n - \frac{C}{2}(\phi_{i+1}^n - \phi_{i-1}^n)$. Despite its appealing $\mathcal{O}((\Delta x)^2)$ spatial accuracy, a rigorous stability analysis (like Von Neumann analysis, which examines the amplification of Fourier modes of the solution) shows that the FTCS scheme is **unconditionally unstable** for the pure 1D linear advection equation when $u \neq 0$. It will diverge for any choice of $\Delta t > 0$ and $\Delta x > 0$, regardless of the Courant number C.

One heuristic way to understand why this scheme is prone to amplifying errors is to consider its behavior with a high-frequency "sawtooth" wave initial condition, as shown in Figure 7.16. Such waves, with alternating signs at adjacent grid points, represent the shortest wavelength component that the grid can resolve and are often where instabilities first manifest.

Let's consider an initial condition of the form $\phi_i^n = (-1)^i A_i$, where $A_i > 0$ and, for simplicity, let's assume the amplitude A_i is locally increasing, i.e., $A_{i+1} > A_i > A_{i-1}$.

- **For an odd node** i, $\phi_i^n = -A_i < 0$. Its neighbors are $\phi_{i+1}^n = A_{i+1}$ and $\phi_{i-1}^n = A_{i-1}$. The FTCS update gives:

$$\phi_i^{n+1} = -A_i - \frac{C}{2}(A_{i+1} - A_{i-1})$$

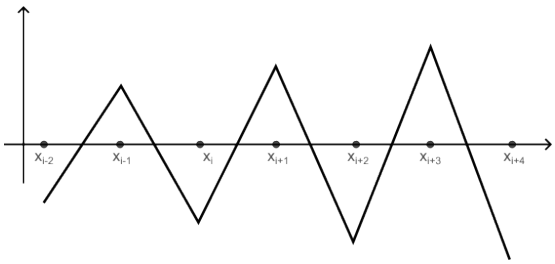


Figure 7.16. A "sawtooth" wave, representing a high-frequency component of the numerical solution. Such alternating patterns are often amplified by unstable numerical schemes.

Since $A_{i+1} > A_{i-1}$, the term in the parenthesis is positive. Therefore, ϕ_i^{n+1} will be even more negative than $-A_i$. The magnitude of the negative peak, $|\phi_i^{n+1}|$, is greater than its original magnitude, $|\phi_i^n| = A_i$. The negative peak is amplified.

- **For an even node i**, $\phi_i^n = A_i > 0$. Its neighbors are $\phi_{i+1}^n = -A_{i+1}$ and $\phi_{i-1}^n = -A_{i-1}$. The FTCS update gives:

$$\phi_i^{n+1} = A_i - \frac{C}{2}(-A_{i+1} - (-A_{i-1})) = A_i + \frac{C}{2}(A_{i+1} - A_{i-1})$$

Again, since $A_{i+1} > A_{i-1}$, the term in the parenthesis is positive. Therefore, ϕ_i^{n+1} will be greater than A_i. The positive peak is also amplified.

This heuristic argument shows how the FTCS scheme can amplify the amplitude of high-frequency components at each time step if $C \neq 0$, leading to unbounded growth and instability. More simply, for a purely directional advective process, the centered difference in FTCS "looks" both upwind and downwind, averaging information from the "wrong" direction (downwind). When combined with a simple forward Euler time step, this can lead to a feedback loop that amplifies errors.

Upwind Schemes (e.g., FTBS for $u > 0$)

Let's now analyze the stability of the first-order Upwind scheme (FTBS for $u > 0$), whose update formula is:

$$\phi_i^{n+1} = \phi_i^n - C(\phi_i^n - \phi_{i-1}^n)$$

where $C = u\Delta t/\Delta x$ is the Courant number (and $C \geq 0$ since we assume $u > 0$).

As a first step, we can rewrite this equation by grouping the terms involving ϕ^n:

$$\begin{aligned} \phi_i^{n+1} &= \phi_i^n - C\phi_i^n + C\phi_{i-1}^n \\ \phi_i^{n+1} &= (1-C)\phi_i^n + C\phi_{i-1}^n \end{aligned} \tag{7.19}$$

This form is very revealing: it shows that the new value ϕ_i^{n+1} is a **linear combination** (specifically, a weighted average) of two values from the previous time step: ϕ_i^n and its upwind neighbor ϕ_{i-1}^n.

Our intuition for stability (related to the concept of boundedness) is that the numerical scheme should not create new, spurious maxima or minima. If the solution ϕ^n is bounded at time n (e.g., $m \leq \phi_j^n \leq M$ for all grid points j), we want the solution ϕ^{n+1} at the next

time step to remain within the same bounds. Let's examine the coefficients in Equation 7.19 to see when this holds.

For ϕ_i^{n+1} to be a well-behaved "average" of ϕ_i^n and ϕ_{i-1}^n, we require the weighting coefficients to be non-negative.

1. **The coefficient of ϕ_{i-1}^n is C**: Since we assume $u > 0$, $\Delta t > 0$, and $\Delta x > 0$, the Courant number $C = \frac{u\Delta t}{\Delta x}$ is always greater than or equal to zero. This condition, $C \geq 0$, is naturally satisfied.
2. **The coefficient of ϕ_i^n is $(1 - C)$**: For this coefficient to be non-negative, we require $1 - C \geq 0$, which implies $C \leq 1$.

Combining these two requirements, we find that for both coefficients to be non-negative, we need:

$$0 \leq C = \frac{u\Delta t}{\Delta x} \leq 1 \tag{7.20}$$

This is precisely the **CFL condition for the stability of the first-order Upwind scheme** (when $u > 0$).

If $C > 1$, the coefficient $(1 - C)$ becomes negative. This means ϕ_i^n contributes negatively to ϕ_i^{n+1}, which can cause the solution to overshoot and develop oscillations, leading to instability. Figure 7.17 illustrates the onset of this unstable behavior when the Courant number slightly exceeds one. A similar analysis for the FTFS scheme (for $u < 0$) leads to the condition $0 \leq -C \leq 1$, or more generally $|C| \leq 1$.

Interpretation of the Upwind Stability Condition If the CFL condition $0 \leq C \leq 1$ is satisfied, the coefficients $(1 - C)$ and C are both non-negative and their sum is $(1 - C) + C = 1$. This means that ϕ_i^{n+1} is a **convex combination** of ϕ_i^n and ϕ_{i-1}^n. This has a powerful consequence: the new value ϕ_i^{n+1} will always be bounded by the minimum and maximum of the values used to compute it (ϕ_i^n and ϕ_{i-1}^n). This property, often related to satisfying a "maximum principle" or ensuring "monotonicity" for certain initial data, prevents the scheme from amplifying errors or creating new, unphysical oscillations. This argument, while not a full formal stability analysis, provides a strong intuitive justification for why the CFL condition $C \leq 1$ is crucial for the good behavior of this scheme. A similar analysis holds for the FTFS scheme when $u < 0$, leading to the general stability condition $|C| \leq 1$ for first-order upwind methods.

7.4.5 Practical Implications of the CFL Condition and Choosing Δt

The CFL condition $C = \frac{|u|\Delta t}{\Delta x} \leq C_{max}$ (where $C_{max} = 1$ for first-order Upwind) has significant practical consequences for explicit advection schemes:

1. **Selection of Δt is Constrained:** Once the advection velocity $|u|$ is known (from the physics of the problem) and the spatial resolution Δx is chosen (to resolve the features of interest), the CFL condition dictates an upper limit on the time step:

$$\Delta t \leq \frac{C_{max}\Delta x}{|u|}$$

Attempting to use a larger Δt will likely result in an unstable simulation.

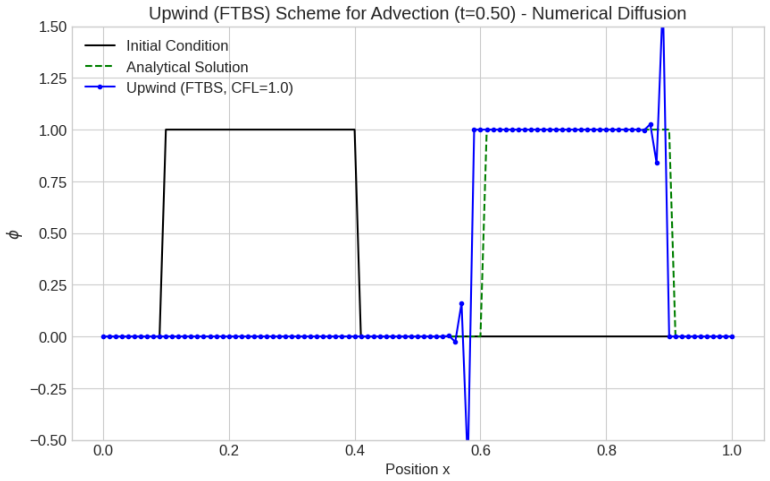


Figure 7.17. Numerical solution using the Upwind (FTBS) scheme for the advection of an initial square wave (black solid line) with a Courant number $C = 1.01$, slightly violating the stability condition $C \leq 1$. The analytical solution is the dashed green line. The Upwind solution (blue line with markers) exhibits significant unphysical oscillations (overshoots and undershoots) and distortion, demonstrating the onset of numerical instability when the CFL condition is not met. Further increasing C would lead to even more pronounced and rapidly growing instabilities.

2. **Computational Cost:** If high spatial resolution (small Δx) is required, Δt must also be proportionally small. This means that a large number of time steps will be needed to simulate a given physical duration t_{final}, increasing the overall computational cost. Unlike the FTCS scheme for diffusion where $\Delta t \propto (\Delta x)^2$, for advection with these explicit schemes, $\Delta t \propto \Delta x$.

3. **Variable Velocities or Grids:** If u or Δx vary within the computational domain, Δt must be chosen based on the *most restrictive local condition* throughout the grid to ensure stability everywhere. This means using the maximum expected $|u|$ and the minimum Δx in the CFL formula.

4. **Safety Margin:** In practice, it is common to choose a Courant number C somewhat less than the theoretical maximum C_{max} (e.g., $C = 0.8$ or 0.9 if $C_{max} = 1$) to provide a safety margin and sometimes to reduce numerical artifacts like diffusion or dispersion, which can be more pronounced when operating very close to the stability limit.

Always remember that satisfying the CFL condition is necessary for stability with these explicit schemes, but it does not guarantee high accuracy. Other numerical errors, such as numerical diffusion, may still be present and significant, as we observed with the Upwind scheme.

Concept Check

The CFL condition for the first-order Upwind scheme is $C = \frac{|u|\Delta t}{\Delta x} \leq 1$. Suppose you are modeling the transport of volcanic ash by a strong wind ($u = 20$ m/s). You want to use a spatial resolution $\Delta x = 100$ m to capture details of the plume near the vent.

1. What is the maximum permissible time step Δt you can use with the first-order Upwind scheme to ensure numerical stability?
2. If, for accuracy reasons related to other rapid processes, you needed to use a Δt that is half of this maximum stable value, what would be your new Δt and the corresponding Courant number C?

7.5 Numerical Artifacts in Advection Schemes: Diffusion and Dispersion

Our exploration of the FTCS and Upwind schemes for the linear advection equation has revealed that numerical solutions can deviate from the exact analytical solution in ways that go beyond simple instability. Even when a scheme is stable, like the first-order Upwind method under the CFL condition $C \leq 1$, the computed solution may not perfectly preserve the shape of the advected profile. These deviations, which are inherent to the discretization process and the specific approximations made by the numerical scheme, are termed **numerical artifacts**. Understanding these artifacts is crucial for correctly interpreting simulation results and for appreciating the limitations and strengths of different numerical methods.

For advection problems, two of the most prominent numerical artifacts are *numerical diffusion* (also known as artificial dissipation) and *numerical dispersion* (related to phase errors).

7.5.1 Numerical Diffusion (Artificial Dissipation)

Numerical diffusion manifests as an excessive **smearing or smoothing** of sharp gradients and high-frequency components in the solution. When a profile with sharp edges (like a square wave) or a narrow peak (like a Gaussian pulse) is advected using a scheme with significant numerical diffusion, these sharp features tend to become rounded, peaks become lower and wider, and the overall profile appears more "spread out" than it should be according to the pure advection equation (which predicts no change in shape). This effect is clearly visible in the results from the first-order Upwind scheme (Figure 7.14).

It is as if an artificial, unphysical diffusion process, with an "effective numerical diffusivity," has been added to the advection equation. Indeed, a more detailed mathematical analysis (Taylor series expansion of the discretized equation, often called modified equation analysis) for the first-order Upwind scheme reveals that its leading truncation error term is proportional to $\frac{\partial^2 \phi}{\partial x^2}$. Specifically, the modified equation approximated by the FTBS scheme ($u > 0, C = u\Delta t/\Delta x$) can be shown to be approximately:

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = \underbrace{\frac{u\Delta x}{2}(1-C)}_{D_{num}} \frac{\partial^2 \phi}{\partial x^2} + \mathcal{O}(\Delta x^2, \Delta t^2) \tag{7.21}$$

The term $D_{num} = \frac{u\Delta x}{2}(1-C)$ acts as a **numerical diffusion coefficient**.

- This numerical diffusion is always present when $0 < C < 1$.
- It is minimized when $C = 1$ (in which case the Upwind scheme becomes exact for constant u if the initial condition is advected exactly by one grid cell per time step). However, operating exactly at $C = 1$ is often impractical due to variations in u or complex geometries.
- It is maximized for a given Δx when C is small (e.g., $C \approx 0$), meaning very small time steps for a fixed Δx can paradoxically increase numerical diffusion relative to the advective step, although the absolute error might still decrease due to smaller Δt in the temporal truncation error. The key is that D_{num} is proportional to Δx.

Numerical diffusion is a common characteristic of many first-order accurate schemes, especially those designed for stability in advection problems.

Consequences in Geoscience Applications. Excessive numerical diffusion can have significant implications when modeling Earth science processes:

- **Transport of Chemical Species or Tracers:** If we are modeling the advection of a distinct parcel of magma with a specific chemical signature, or a plume of contaminant in groundwater, numerical diffusion will cause the boundaries of this parcel/plume to appear more mixed with the surroundings than they physically are. Peak concentrations will be underestimated, and the spatial extent of low concentrations will be overestimated. For example, if an initial condition consists only of values $\phi = 0$ (resident magma) and $\phi = 1$ (intruding magma), numerical diffusion will create a zone of artificial mixing with intermediate ϕ values, even if no physical diffusion is occurring.
- **Front Tracking:** When modeling the movement of sharp fronts (e.g., a thermal front, a saturation front in porous media, or the leading edge of a pyroclastic density current), numerical diffusion will smear out the front, making it appear less steep and more spread out than in reality. This can affect predictions of arrival times or the intensity of the front.
- **Preservation of Physical Constraints:** If ϕ represents a quantity that must be physically bounded (e.g., a volumetric fraction that must remain between 0 and 1), numerical diffusion (if not coupled with dispersion causing overshoots) generally helps keep the solution within these bounds by smoothing, but it does so at the cost of accuracy in representing sharp transitions.

While numerical diffusion can sometimes be beneficial by damping unphysical oscillations that might arise from other numerical errors, it is generally an undesirable artifact that reduces the accuracy of the simulation.

7.5.2 Numerical Dispersion

Numerical dispersion is another common artifact, particularly in schemes that are not sufficiently dissipative or are of higher order but not carefully designed for sharp gradients. It arises because different wavelength components (or Fourier modes) of the numerical solution propagate at slightly different (and incorrect) speeds compared to the true physical advection velocity u.

This differential propagation speed for different wavelengths leads to a "dispersal" of the wave packet and can manifest as:

- **Spurious Oscillations or "Wiggles":** These often appear near sharp changes in the solution profile (e.g., the edges of a square wave or a steep front). These oscillations are non-physical and can make the solution appear noisy or unrepresentative.
- **Phase Errors:** The main features of the profile (like peaks or troughs) might arrive too early or too late compared to the exact solution.

The FTCS scheme, if it were stable for advection, is known to be dispersive. Some higher-order centered schemes can also exhibit significant dispersion if not properly formulated (e.g., with added artificial diffusion or limiters). While first-order upwind schemes are dominated by numerical diffusion, which tends to damp oscillations, some numerical dispersion can still be present.

Consequences in Geoscience Applications:

- **Wave Propagation (e.g., Tsunami Modeling - Conceptual):** While the linear advection equation is a gross simplification for tsunamis (which are governed by nonlinear shallow water equations), the concept of numerical dispersion is highly relevant. If different parts of a numerically modeled wave (e.g., its leading edge, peak, and tail) travel at slightly incorrect speeds due to numerical dispersion, this can lead to errors in predicting the wave's arrival time at a coastline, its shape upon arrival, and its run-up height. Spurious oscillations might be misinterpreted as real wave features.
- **Transport of Sharp Interfaces:** When advecting a sharp interface between two distinct fluid compositions or contaminant levels, numerical dispersion can create non-physical undershoots or overshoots. For example, if ϕ represents a concentration that must physically remain between 0 and 1, dispersive oscillations could cause the numerical solution to predict $\phi < 0$ or $\phi > 1$ in some regions, violating physical constraints. This is a serious issue, especially if these values are then used in other coupled equations (e.g., reaction rates that depend on concentration).

Minimizing both numerical diffusion and dispersion while maintaining stability is a central challenge in the development of numerical schemes for advection-dominated problems. This often leads to the design of higher-order schemes or methods that incorporate "limiters" to control oscillations near discontinuities. For our introductory purposes, recognizing their existence and qualitative behavior in simpler schemes is the primary goal.

7.6 Reflections on Advection Schemes and Model Selection

Our exploration of the Forward-Time Centered-Space (FTCS) and first-order Upwind schemes for the 1D linear advection equation has provided valuable, if sometimes cautionary, lessons about numerical modeling. While the advection equation itself appears simple, its numerical solution highlights fundamental challenges and trade-offs inherent in discretizing PDEs.

7.6.1 Performance Recap: FTCS vs. First-Order Upwind

Let's briefly summarize the contrasting behaviors we observed:

- **FTCS Scheme:**
  - *Formulation:* Combines a forward step in time with a central difference in space.

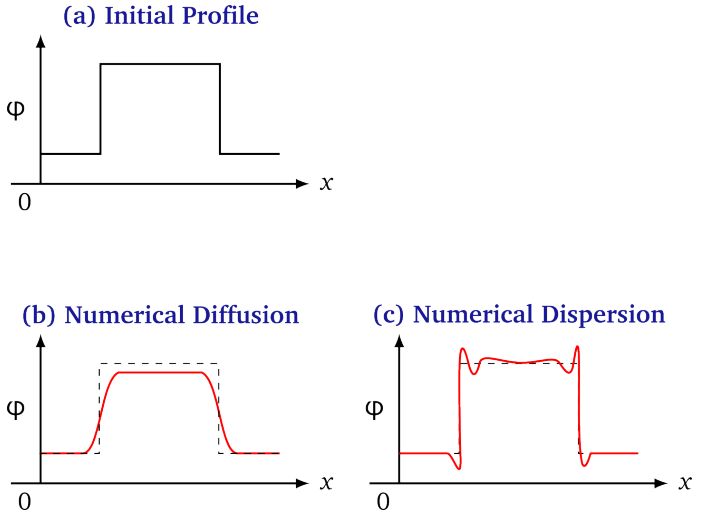


Figure 7.18. Conceptual illustration of common numerical artifacts when advecting a sharp profile (e.g., a square wave). (a) The initial, undistorted profile. (b) Effect of numerical diffusion: the sharp corners are rounded, and the profile is smeared out. (c) Effect of numerical dispersion: spurious oscillations (wiggles) appear, particularly near the sharp edges, and the main pulse might be distorted or show phase errors.

  - *Formal Accuracy:* $\mathcal{O}(\Delta t, (\Delta x)^2)$ – first-order in time, second-order in space. This higher spatial order might initially suggest superior accuracy.
  - *Observed Behavior for Advection:* Unconditionally unstable for $u \neq 0$. The numerical solution rapidly develops growing oscillations that render it physically meaningless (as seen in Figure 7.12).
  - *Verdict for Pure Advection:* Unsuitable. Its instability outweighs any formal accuracy advantages.

- **First-Order Upwind Scheme (e.g., FTBS for $u > 0$):**
  - *Formulation:* Combines a forward step in time with a one-sided spatial difference taken in the "upwind" direction (against the flow).
  - *Formal Accuracy:* $\mathcal{O}(\Delta t, \Delta x)$ – first-order in both time and space.
  - *Observed Behavior for Advection:* Conditionally stable, requiring the Courant number $C = |u|\Delta t/\Delta x \leq 1$. When stable, it correctly propagates the advected feature. However, it introduces significant **numerical diffusion**, which smears sharp gradients and reduces peak amplitudes (as seen in Figure 7.14).
  - *Verdict for Pure Advection:* A robust and simple starting point if stability is prioritized and some smearing of the solution is acceptable. Its primary drawback is low accuracy due to numerical diffusion.

This comparison underscores that formal order of accuracy is not the sole determinant of a scheme's utility; stability is a non-negotiable prerequisite, and other numerical artifacts must also be considered.

7.6.2 The Inescapable Trade-off: Stability, Accuracy, and Numerical Artifacts

The contrasting behaviors of FTCS and Upwind highlight a common theme in numerical methods for PDEs: there are often trade-offs between desirable properties.

- **Stability vs. Formal Accuracy:** The FTCS scheme aimed for higher spatial accuracy with its centered difference but sacrificed stability. The Upwind scheme achieved stability (under the CFL condition) by using a one-sided difference, which, however, reduced its formal spatial accuracy to first order and introduced numerical diffusion.
- **Minimizing One Artifact Can Exacerbate Another:** Schemes that are very good at suppressing numerical diffusion (e.g., many higher-order centered schemes) can sometimes be more prone to numerical dispersion (generating spurious oscillations), especially near sharp fronts, unless special techniques are employed. Conversely, highly diffusive schemes (like first-order Upwind) are excellent at damping oscillations but at the cost of smearing the solution.
- **Simplicity vs. Performance:** Simpler schemes like FTCS and Upwind are easy to understand and implement. Achieving higher accuracy and better control over numerical artifacts typically requires more complex schemes (e.g., higher-order methods, flux limiters, non-linear schemes) that are more challenging to derive, analyze, and code.

The "perfect" numerical scheme that is unconditionally stable, highly accurate for all types of solutions (smooth and discontinuous), free of numerical artifacts, and computationally inexpensive generally does not exist. The choice of scheme is therefore almost always a compromise, guided by the specific requirements of the problem at hand.

7.6.3 Choosing an Advection Scheme: Problem-Dependent Considerations

Given these trade-offs, how does one approach the selection of a numerical scheme for an advection problem in the Earth sciences? The answer is highly dependent on the specific application and the characteristics of the physical system being modeled.

- **Nature of the Solution Expected:**
  - If the solution is expected to be very smooth and diffusion-like processes are also physically present (i.e., you are solving an advection-diffusion equation), the numerical diffusion from a simple upwind scheme might be less problematic or even masked by the physical diffusion.
  - However, if you need to model the transport of sharp fronts, interfaces, or discontinuities (e.g., the contact between two distinct magma batches, the leading edge of a contaminant plume, or an initially square wave representing a distinct parcel of material), the smearing effect of first-order upwind schemes can be a severe limitation. In such cases, its use might lead to qualitatively incorrect conclusions about mixing zones or arrival times of sharp features.
- **Accuracy Requirements vs. Computational Resources:**
  - For preliminary investigations or qualitative models where only the general direction and approximate magnitude of transport are needed, a robust but diffusive scheme like Upwind might suffice, especially given its simplicity.

  - For quantitative predictions requiring high fidelity in representing the shape and amplitude of advected features, more sophisticated, higher-order schemes are generally necessary, even if they come with increased computational cost or implementation complexity.

- **Physical Constraints:**
  - If the advected quantity ϕ represents something that must remain physically bounded (e.g., a mass fraction between 0 and 1, a positive concentration), schemes prone to strong dispersive oscillations that can produce unphysical overshoots or undershoots must be used with caution or be equipped with techniques (like flux limiters in more advanced methods) to prevent such violations. First-order upwind schemes are often "monotonic" (do not create new extrema) under their stability condition, which can be an advantage in these situations, despite their diffusivity.

- **Verification and Validation are Key:** Regardless of the scheme chosen, it is always crucial to perform verification tests (e.g., comparing with analytical solutions for simplified cases, checking for grid convergence) and, where possible, validation against experimental or field data. This helps to understand the extent to which numerical artifacts might be influencing the simulation results and to build confidence in the model's predictions.

The simple schemes discussed in this chapter provide a starting point. Recognizing their limitations is the first step towards appreciating why a vast array of more advanced numerical methods for advection-dominated problems have been developed.

7.6.4 A Brief Outlook: Pathways to Improving Advection Schemes

The challenges posed by numerical diffusion and dispersion in simple schemes have driven extensive research into more advanced numerical methods for hyperbolic conservation laws. While a detailed exploration is beyond the scope of this introductory volume, it is useful to be aware of some of the directions taken:

- **Higher-Order Schemes:** Schemes with higher formal orders of accuracy in space and/or time (e.g., Lax-Wendroff, Beam-Warming, Fromm's scheme) can significantly reduce numerical diffusion compared to first-order upwind. However, linear higher-order schemes often introduce more pronounced numerical dispersion (oscillations) near sharp gradients.
- **Flux Limiter Methods (High-Resolution Schemes):** A major breakthrough came with the development of "high-resolution" schemes that aim to achieve at least second-order accuracy in smooth regions of the solution while controlling or eliminating spurious oscillations near discontinuities. These methods, such as Total Variation Diminishing (TVD) schemes, Flux-Corrected Transport (FCT), Essentially Non-Oscillatory (ENO), and Weighted Essentially Non-Oscillatory (WENO) schemes, often use nonlinear "flux limiters" or adaptive stencils to achieve these desirable properties. They are considerably more complex than the basic schemes discussed here but are the standard in many fields requiring accurate shock-capturing or front-tracking.
- **Implicit Schemes for Advection:** While less common for pure linear advection than for diffusion (due to the CFL condition being linear in Δx for advection, not quadratic),

implicit time-stepping can also be applied to advection equations. This might be beneficial for certain coupled problems or when very long time integrations are needed, but they still require careful spatial discretization to manage diffusion and dispersion.

The journey into advanced numerical methods for advection is rich and ongoing, driven by the need for ever more accurate and robust simulations of complex physical systems. The foundational understanding of basic schemes, stability, and numerical artifacts gained in this chapter is essential preparation for engaging with these more sophisticated techniques.

Chapter Summary: Numerical Advection and Its Challenges

This chapter focused on the 1D linear advection equation, $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$, a fundamental hyperbolic PDE describing the transport of a scalar quantity ϕ by a constant velocity u. We explored its physical basis, analytical solution, and the development and analysis of explicit finite difference schemes for its numerical approximation.

Key concepts and methodologies covered include:

- **The Linear Advection Equation:**
  - Its derivation from the general scalar transport equation under simplifying assumptions.
  - Its properties as a first-order, linear, hyperbolic PDE, signifying directional information propagation at finite speed.
  - The analytical solution $\phi(x,t) = \phi_0(x-ut)$, representing a shape-preserving translation of the initial profile $\phi_0(x)$, illustrated with Python examples.
  - The concept of characteristic curves along which ϕ is constant.
- **Boundary Conditions for Advection:**
  - Recap of Dirichlet and Neumann conditions in the context of advection (inflow/outflow).
  - Detailed introduction to **Periodic Boundary Conditions (PBCs)**: their definition, conceptual meaning (a looped domain), relevance in geoscience models, and numerical implementation using ghost cells or modulo indexing. Their utility for testing numerical schemes without boundary interference was highlighted.
- **Explicit Finite Difference Schemes:**
  - **Forward-Time Centered-Space (FTCS):** Derived as $\phi_i^{n+1} = \phi_i^n - \frac{C}{2}(\phi_{i+1}^n - \phi_{i-1}^n)$. Formally $\mathcal{O}(\Delta t, (\Delta x)^2)$.
  - **First-Order Upwind Schemes (FTBS for $u > 0$, FTFS for $u < 0$):** Derived by using one-sided spatial differences that respect the flow direction, e.g., $\phi_i^{n+1} = (1-C)\phi_i^n + C\phi_{i-1}^n$ for FTBS. Formally $\mathcal{O}(\Delta t, \Delta x)$.
  - Python implementations (core loops and examples) were discussed for FTCS and Upwind with periodic BCs.
- **Numerical Stability and the CFL Condition:**

  - The critical importance of numerical stability was emphasized.
  - The **Courant number**, $C = |u|\Delta t/\Delta x$, was defined and its physical interpretation discussed.
  - The **Courant-Friedrichs-Lewy (CFL) condition**, $C \leq C_{max}$, was introduced as a necessary condition for the stability of explicit hyperbolic schemes.
  - Specific stability properties: FTCS is unconditionally unstable for pure advection; first-order Upwind schemes are stable if $0 \leq C \leq 1$.
  - Practical implications of the CFL condition on the choice of Δt relative to Δx and u were explored.
- **Numerical Artifacts:**
  - **Numerical Diffusion (Artificial Dissipation):** Explained as the smearing of sharp profiles, characteristic of first-order upwind schemes, with its connection to the truncation error term.
  - **Numerical Dispersion (Phase Error):** Described as the cause of spurious oscillations or wiggles, particularly problematic for schemes that are not sufficiently dissipative when handling sharp gradients.
  - The consequences of these artifacts for geoscience applications, such as modeling tracer transport or front propagation, were discussed.
- **Scheme Selection and Outlook:**
  - The chapter concluded with reflections on the trade-offs between stability, accuracy, and numerical artifacts, emphasizing that the "best" scheme is problem-dependent.
  - A brief outlook on more advanced techniques (higher-order schemes, flux limiters) for improving advection simulations was provided.

The study of the linear advection equation provides fundamental insights into the challenges of numerically solving hyperbolic PDEs, particularly concerning stability and the accurate representation of transported features. The exercises that follow will allow for further practical exploration of these important concepts.

Chapter 7 Exercises

These exercises are designed to help you implement and test the finite difference schemes for the 1D linear advection equation, explore the CFL condition, and observe common numerical artifacts. Base your work on the Python scripts and concepts presented in this chapter. For all simulations, unless otherwise specified, use periodic boundary conditions.

E7.1: FTCS Scheme - Detailed Instability Exploration

Refer to the Python script for the FTCS scheme (e.g., based on Listing 7.2). Use an initial square wave profile (e.g., $\phi = 1$ for $0.1L \leq x \leq 0.4L$ and $\phi = 0$ elsewhere on a domain $L = 1.0$). Set advection velocity $u = 1.0$.

1. **Varying Courant Number (C):** Set $N_x = 101$ (number of spatial points). Calculate Δx. Then, run simulations up to $t_{final} = 0.05$ (a short time, as FTCS is unstable) for the following Courant numbers: $C = 0.1, 0.5, 1.0$.

- For each C, calculate the required $\Delta t = C\Delta x/u$.
- Plot the numerical solution at t_{final} alongside the initial condition and the analytical solution.
- Describe how the instability manifests for different C values. Does it always look the same? Does it appear faster or slower?

2. **Effect of Spatial Resolution (Δx):** Fix the Courant number at a value you found to be quickly unstable, e.g., $C = 0.5$. Now, vary the spatial resolution:
   - Run with $N_x = 51$. Calculate the new Δx and Δt (to keep $C = 0.5$). Simulate up to $t_{final} = 0.05$.
   - Run with $N_x = 201$. Calculate the new Δx and Δt. Simulate up to $t_{final} = 0.05$.

   Plot the results. Does changing Δx (while keeping C constant) prevent the instability of the FTCS scheme for pure advection?

E7.2: Upwind Scheme - Impact of Courant Number and Numerical Diffusion

Refer to the Python script for the first-order Upwind (FTBS for $u > 0$) scheme (e.g., based on Listing 7.3). Use an initial square wave profile (e.g., $\phi = 1$ for $0.1L \le x \le 0.4L$, $\phi = 0$ elsewhere, $L = 1.0$) and $u = 1.0$. Simulate up to $t_{final} = 0.5$ (long enough for the wave to pass the initial position if $L = 1.0$).

1. **Varying Courant Number (C):** Set $N_x = 101$. Run simulations for $C = 0.2, 0.5, 0.8, 1.0$.
   - Plot the numerical solution at t_{final} for each C value on the same graph, along with the initial condition and the analytical solution.
   - How does the amount of numerical diffusion (smearing of the square wave) change as C varies within the stable range ($0 < C \le 1$)? Which value of C gives the "sharpest" (least diffusive) result for the Upwind scheme?
2. **Testing Stability Limit:** Now try $C = 1.01$ (or $C = 1.1$). Plot the result. What happens when the CFL condition $C \le 1$ is violated for the Upwind scheme? Compare this to the FTCS instability.
3. **Numerical Diffusion Coefficient (Conceptual):** The modified equation for the first-order upwind scheme includes a numerical diffusion term $D_{num}\frac{\partial^2\phi}{\partial x^2}$ where $D_{num} = \frac{u\Delta x}{2}(1-C)$.
   - For a fixed Δx and u, how does D_{num} change as C goes from near 0 to 1? Does this match your observations of smearing from part (1)?
   - What happens to D_{num} if $C = 1$? What does this imply for the accuracy of the Upwind scheme at $C = 1$?

E7.3: Advecting Different Initial Profiles with Upwind Scheme

Using the Upwind (FTBS, $u > 0$) scheme with $N_x = 101$, $L = 1.0$, $u = 1.0$, and a stable Courant number (e.g., $C = 0.8$), simulate the advection up to $t_{final} = 0.5$ for the following initial conditions:

1. A narrow Gaussian pulse (e.g., use 'initialConditionGaussian' from Listing **??**, perhaps with 'x0=0.25', 'sigma=0.05').
2. A sine wave, e.g., $\phi_0(x) = 0.5 + 0.5\sin(2\pi x/L)$.
3. A triangular pulse.

For each case, plot the initial condition, the analytical solution at t_{final}, and the numerical solution at t_{final}. Discuss how well the Upwind scheme preserves the shape of these different profiles. Does numerical diffusion affect smooth profiles differently from sharp ones?

E7.4: Mass Conservation with Periodic Boundary Conditions

For the 1D linear advection equation $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$ with periodic boundary conditions, the total amount of ϕ in the domain, $M(t) = \int_0^L \phi(x,t)dx$, should be conserved over time if u is constant (or if the equation is written in conservative form $\frac{\partial \phi}{\partial t} + \frac{\partial (u\phi)}{\partial x} = 0$ and $u(0) = u(L)$ for periodic u).

Numerically, we can approximate this integral as $M_n \approx \sum_{i=0}^{N_x-1} \phi_i^n \Delta x$.

Tasks:

1. Modify your Upwind (FTBS) script (using the square wave IC and $C = 0.8$) to calculate and store this sum M_n at each time step n.
2. At the end of the simulation, plot M_n (or the percentage change from M_0) versus time.
3. Does the first-order Upwind scheme, as implemented with periodic BCs for the form $\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0$, conserve the total mass M_n exactly? If not, is the loss/gain significant?
4. **Conceptual:** Why might a scheme derived directly from the non-conservative form of the advection equation sometimes have issues with perfect discrete conservation, even with periodic BCs? (Hint: Think about how the sum $\sum(\phi_i^n - \phi_{i-1}^n)$ behaves over a periodic domain.)

(Note: For constant u, the FTBS scheme applied to $\phi_t + u\phi_x = 0$ should conserve mass well with periodic BCs. This exercise helps verify it.)

Chapter 8

Concluding Remarks and the Path Forward in Numerical Modelling

Throughout this volume, we have embarked on a journey from the fundamental physical principles governing Earth science phenomena to the practical implementation of numerical methods for solving the differential equations that describe them. We began by understanding why differential equations are the natural language for many dynamic systems, explored essential Python tools for scientific computation, and then focused on the numerical solution of both Ordinary Differential Equations (ODEs) and two cornerstone Partial Differential Equations (PDEs): the Heat (Diffusion) Equation and the Linear Advection Equation.

Our focus has consistently been on building not just a toolkit of numerical recipes, but also a conceptual understanding of *why* these methods work, their inherent assumptions, their strengths, and, crucially, their limitations. We have seen that numerical modelling is a powerful discipline, but one that requires careful thought, critical assessment, and a constant awareness of the interplay between the physical problem, its mathematical representation, and the chosen numerical approximation.

8.1 Revisiting the Numerical Modeller's Workflow

The "Numerical Modeller's Workflow," introduced conceptually in earlier discussions, can now be appreciated with greater depth given the practical experience gained through the preceding chapters. Let's revisit and expand upon its key stages:

1. **Understand the Physics and Define the Problem:** This remains the paramount first step. Before any equations are written or code is developed, a clear understanding of the geoscience process is essential. What are the dominant physical laws? What are the relevant scales in space and time? What quantities are we trying to predict or understand? What simplifications are reasonable and justifiable? For example, is a 1D model sufficient, or are 2D/3D effects crucial? Is the process diffusion-dominated, advection-dominated, or a mix? Is it linear or inherently nonlinear?

2. **Formulate the Mathematical Model (Governing Equations and Conditions):** This involves translating the physical understanding into a precise mathematical problem:
    - **Governing Equations:** Deriving or selecting the appropriate ODEs or PDEs (e.g., from conservation principles as seen in Chapter **??**).
    - **Initial Conditions (ICs):** Specifying the state of the system at the beginning of the simulation (e.g., $T(x,0)$, $\phi(x,0)$).
    - **Boundary Conditions (BCs):** Specifying how the system interacts with its surroundings at the edges of the computational domain (e.g., Dirichlet, Neumann, Periodic, as discussed in Chapters 6 and **??**). The choice of BCs must be physically appropriate for the problem.

    This stage often involves idealizations and assumptions that simplify the real-world complexity into a tractable mathematical form.
3. **Select and Implement a Numerical Method:** This is where much of our focus has been. The choice involves:
    - **Discretization:** Choosing how to represent the continuous domain (spatial and temporal grids, $\Delta x, \Delta t$) and approximate derivatives (e.g., finite differences like Forward Euler, Backward Euler, FTCS, Upwind).
    - **Algorithm Properties:** Considering the method's formal order of accuracy, its stability properties (e.g., CFL condition for explicit advection/diffusion schemes), and its tendency to introduce numerical artifacts (diffusion, dispersion).
    - **Complexity vs. Benefit:** Balancing the simplicity of a scheme (e.g., first-order Upwind) against the potential for higher accuracy or better stability from more complex methods (e.g., Crank-Nicolson for diffusion, or higher-order advection schemes hinted at). For nonlinear problems solved implicitly, this also includes choosing a robust root-finding algorithm (like the bisection method).
4. **Verification and Validation (V&V): Crucial Steps for Trust** These two terms are distinct and both essential for building confidence in a numerical model:
    - **Verification ("Solving the equations right"):** This process aims to ensure that the numerical code correctly solves the chosen mathematical equations (the discretized form of the PDEs/ODEs). Techniques include:
        - Comparing numerical solutions to known analytical solutions for simplified test cases (as we did for decay, heat, and advection).
        - The Method of Manufactured Solutions: Designing a problem with a known (manufactured) solution and checking if the code reproduces it to the expected order of accuracy.
        - Code-to-code comparison: Comparing results with those from other, independently developed codes for the same problem.
        - Grid convergence studies: Systematically refining Δx and Δt to ensure the numerical solution converges at the theoretically predicted rate.
    - **Validation ("Solving the right equations"):** This process aims to determine the degree to which the mathematical model (and its numerical solution) is an accurate representation of the real-world physical system it is intended to simulate. Techniques include:

  - Comparing model predictions against experimental data or field observations from the actual geoscience system.
  - Assessing the sensitivity of model outputs to uncertainties in input parameters and assumptions.

  A code can be perfectly verified (correctly solving the equations it was programmed to solve) but still produce invalid results if the underlying mathematical model or its assumptions are flawed representations of reality.

5. **Simulation, Analysis, and Interpretation of Results:** Once confidence in the model is established through V&V, simulations can be run to explore the problem of interest. This involves:
   - Designing numerical experiments to address specific scientific questions.
   - Visualizing and analyzing the output data (as practiced with Matplotlib).
   - Critically interpreting the results in the context of the physical system, being mindful of the model's assumptions and the potential influence of numerical artifacts.
   - Quantifying uncertainties in model predictions.
6. **Documentation and Communication:** Clearly documenting the model (equations, assumptions, numerical methods, parameters) and effectively communicating the results and their limitations are vital parts of the scientific process.

This workflow is often iterative. Insights gained from analyzing results or from V&V might lead back to refining the physical understanding, the mathematical model, or the numerical method.

8.2 Key Lessons and Broader Implications from Our Journey

Our exploration through the principles of numerical modelling has illuminated several overarching themes that extend beyond the specific equations and algorithms we have discussed. Perhaps the most fundamental realization is that numerical solutions are, by their very nature, **approximations** to the true behavior of continuous physical systems. This inherent characteristic means that a deep understanding of the sources and nature of error—be it truncation error arising from our discretization choices, or round-off error from the finite precision of computers—is essential for any modeller.

We have consistently seen that **numerical stability** is not merely a desirable feature but a paramount requirement. An unstable scheme, regardless of its theoretical accuracy or elegance, yields results that are devoid of physical meaning, as dramatically illustrated by the FTCS scheme for advection. The stability conditions we encountered, such as the CFL criterion for explicit advection and diffusion schemes, highlight the delicate balance that must often be struck between the chosen time step, spatial resolution, and the physical parameters of the system. These conditions often dictate very practical limits on our computational approach.

Furthermore, **accuracy** itself has revealed itself to be a multi-faceted concept. While the formal order of accuracy provides a theoretical measure of how rapidly errors should decrease with grid refinement, we observed that numerical artifacts like artificial diffusion

(seen with first-order upwind schemes) or numerical dispersion (which can plague less dissipative or improperly designed higher-order schemes) can significantly impact the true fidelity of a solution. The smearing of sharp fronts or the appearance of spurious oscillations are not just mathematical curiosities; they can lead to misinterpretations of the physical processes being modelled, for instance, by overestimating mixing zones or predicting unphysical extrema in concentrations or temperatures.

This leads to another critical lesson: there is **no universally "best" numerical method**. The FTCS scheme, conditionally stable and useful for the heat equation, proved entirely unsuitable for pure advection. The first-order upwind scheme, while robustly stable for advection under the CFL condition, paid a price in accuracy through numerical diffusion. The "optimal" approach is invariably problem-dependent, requiring a thoughtful consideration of the underlying physics, the specific characteristics of the PDE being solved (e.g., its hyperbolic or parabolic nature, its linearity or nonlinearity), the features of the expected solution (smooth versus sharp or discontinuous), the desired level of precision, and the available computational resources.

Finally, this journey has underscored the empowering role of **computational tools**. Python, augmented by its powerful scientific libraries like NumPy for efficient array computations and Matplotlib for insightful visualization, provides an accessible yet remarkably capable environment. It allows for the relatively straightforward implementation of numerical schemes, facilitates rapid prototyping and testing of ideas, and enables a deep, interactive exploration of how model parameters and numerical choices influence simulation outcomes. The ability to not only formulate a model but also to implement it, visualize its behavior, and critically assess its results is a cornerstone of modern quantitative science.

The principles of discretization, stability analysis, error assessment, and careful implementation discussed throughout this volume provide a foundational skill set. As you venture into more complex modelling challenges in your specific areas of Earth science research, these core concepts will serve as a reliable guide. The path of a numerical modeller is one of continuous learning and refinement, but it is a journey that opens up profound new ways to understand and interact with the dynamic world around us.