Finite Difference Computing with PDEs

A Devito Approach

Hans Petter Langtangen Svein Linge

2026-01-29

Table of contents

Welcome

This book teaches finite difference methods for solving partial differential equations, featuring Devito for high-performance PDE solvers.

About this Edition

This is an adaptation of *Finite Difference Computing with PDEs: A Modern Software Approach* by Hans Petter Langtangen and Svein Linge (Springer, 2017). This Devito edition features:

- **Devito** - A domain-specific language for symbolic PDE specification and automatic code generation
- **Quarto** - Modern scientific publishing for web and PDF output
- **Modern Python** - Type hints, testing, and CI/CD practices

Adapted by Gerard J. Gorman (Imperial College London).

License

What is Devito?

Devito allows you to write PDEs symbolically and automatically generates optimized finite difference code:

```
from devito import Grid, TimeFunction, Eq, Operator, solve, Constant

grid = Grid(shape=(101,), extent=(1.0,))
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
c = Constant(name='c')  # wave speed

# Write the wave equation symbolically: u_tt = c^2 * u_xx
pde = Eq(u.dt2, c**2 * u.dx2)

# Solve for u at the next time step
update = Eq(u.forward, solve(pde, u.forward))
```

```
# Devito generates optimized C code
op = Operator([update])
op.apply(time_M=100, dt=0.001, c=1.0)
```

Book Structure

The book covers:

1. **Introduction to Devito** - Grid, Function, TimeFunction, Operator, and boundary conditions
2. **Wave Equations** - 1D/2D wave propagation, sources, absorbing boundaries
3. **Diffusion Equations** - Heat equation, stability analysis, 2D extension
4. **Advection Equations** - Upwind schemes, Lax-Wendroff, CFL condition
5. **Nonlinear Problems** - Operator splitting, Burgers' equation, Picard iteration

Plus appendices on finite difference formulas, truncation error analysis, and software engineering.

Getting Started

```
git clone https://github.com/devitocodes/devito_book.git
cd devito_book
pip install -e ".[devito]"
```

See the GitHub repository for full installation instructions.

Preface

About This Adaptation

This book is an adaptation of *Finite Difference Computing with PDEs: A Modern Software Approach* by Hans Petter Langtangen and Svein Linge, originally published by Springer in 2017 under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Original Work:

> Langtangen, H.P., Linge, S. (2017). *Finite Difference Computing with PDEs: A Modern Software Approach*. Texts in Computational Science and Engineering, vol 16. Springer, Cham. https://doi.org/10.1007/978-3-319-55456-3

What Has Changed

This edition has been substantially adapted to feature Devito, a domain-specific language for symbolic PDE specification and automatic code generation.

New Content:

- Introduction to Devito chapter covering Grid, Function, TimeFunction, and Operator
- Devito solver implementations for wave, diffusion, advection, and nonlinear equations
- Comprehensive exercises using Devito
- Test suite with verification of all numerical methods

Technical Updates:

- Conversion from DocOnce to Quarto publishing format
- Modern Python practices
- Continuous integration and testing infrastructure
- Updated external links and references

Preserved Content:

- Mathematical derivations and theoretical foundations
- Pedagogical structure and learning philosophy
- Appendices on truncation errors and finite difference formulas

Acknowledgment

This adaptation was prepared by Gerard J. Gorman (Imperial College London) in collaboration with the Devito development team.

Professor Hans Petter Langtangen passed away in October 2016. His profound contributions to computational science education continue to benefit students and practitioners worldwide. This adaptation aims to honor his legacy by bringing his pedagogical approach to modern tools.

Original Preface

The following preface is from the original work by Langtangen and Linge.

There are so many excellent books on finite difference methods for ordinary and partial differential equations that writing yet another one requires a different view on the topic. The present book is not so concerned with the traditional academic presentation of the topic, but is focused at teaching the practitioner how to obtain reliable computations involving finite difference methods. This focus is based on a set of learning outcomes:

1. understanding of the ideas behind finite difference methods,
2. understanding how to transform an algorithm to a well-designed computer code,
3. understanding how to test (verify) the code,
4. understanding potential artifacts in simulation results.

Compared to other textbooks, the present one has a particularly strong emphasis on computer implementation and verification. It also has a strong emphasis on an intuitive understanding of constructing finite difference methods. To learn about the potential non-physical artifacts of various methods, we study exact solutions of finite difference schemes as these give deeper insight into the physical behavior of the numerical methods than the traditional (and more general) asymptotic error analysis. However, asymptotic results regarding convergence rates, typically truncation errors, are crucial for testing implementations, so an extensive appendix is devoted to the computation of truncation errors.

Why finite differences?

One may ask why we do finite differences when finite element and finite volume methods have been developed to greater generality and sophistication than finite differences and can cover more problems. The finite element and finite volume methods are also the industry standard nowadays. Why not just those methods? The reason for finite differences is the method's simplicity, both from a mathematical and coding perspective. Especially in academia, where simple model problems are used a lot for teaching and in research (e.g., for verification of advanced implementations), there is a constant need to solve the model problems from scratch with easy-to-verify computer codes. Here, finite differences are ideal. A simple 1D heat equation can of course be solved by a finite element package, but a 20-line code with a difference scheme is just right to the point and provides an understanding of all details involved in the model and the solution method. Everybody nowadays

has a laptop and the natural method to attack a 1D heat equation is a simple Python or Matlab program with a difference scheme. The conclusion goes for other fundamental PDEs like the wave equation and Poisson equation as long as the geometry of the domain is a hypercube. The present book contains all the practical information needed to use the finite difference tool in a safe way.

Various pedagogical elements are utilized to reach the learning outcomes, and these are commented upon next.

Simplify, understand, generalize

The book's overall pedagogical philosophy is the three-step process of first *simplifying* the problem to something we can *understand* in detail, and when that understanding is in place, we can *generalize* and hopefully address real-world applications with a sound scientific problem-solving approach. For example, in the chapter on a particular family of equations we first simplify the problem in question to a 1D, constant-coefficient equation with simple boundary conditions. We learn how to construct a finite difference method, how to implement it, and how to understand the behavior of the numerical solution. Then we can generalize to higher dimensions, variable coefficients, a source term, and more complicated boundary conditions. The solution of a compound problem is in this way an assembly of elements that are well understood in simpler settings.

Constructive mathematics

This text favors a constructive approach to mathematics. Instead of a set of definitions followed by popping up a method, we emphasize how to think about the construction of a method. The aim is to obtain a good intuitive understanding of the mathematical methods.

The text is written in an easy-to-read style much inspired by the following quote.

> *Some people think that stiff challenges are the best device to induce learning, but I am not one of them. The natural way to learn something is by spending vast amounts of easy, enjoyable time at it. This goes whether you want to speak German, sight-read at the piano, type, or do mathematics. Give me the German storybook for fifth graders that I feel like reading in bed, not Goethe and a dictionary. The latter will bring rapid progress at first, then exhaustion and failure to resolve.*
>
> *The main thing to be said for stiff challenges is that inevitably we will encounter them, so we had better learn to face them boldly. Putting them in the curriculum can help teach us to do so. But for teaching the skill or subject matter itself, they are overrated.*
>
> — Lloyd N. Trefethen, Applied Mathematician, 1955-.

This book assumes some basic knowledge of finite difference approximations, differential equations, and scientific Python or MATLAB programming, as often met in an introductory numerical methods course. Readers without this background may start with the light companion book "Finite Difference Computing with Exponential Decay Models" (Langtangen 2016b). That book will in particular be a useful resource for the programming parts of the present book. Since the present book deals with partial differential equations, the reader is assumed to master multi-variable calculus and linear algebra.

Fundamental ideas and their associated scientific details are first introduced in the simplest possible differential equation setting, often an ordinary differential equation, but in a way that easily allows reuse in more complex settings with partial differential equations. With this approach, new concepts are introduced with a minimum of mathematical details. The text should therefore have a potential for use early in undergraduate student programs.

All nuts and bolts

Many have experienced that "vast amounts of easy, enjoyable time", as stated in the quote above, arises when mathematics is implemented on a computer. The implementation process triggers understanding, creativity, and curiosity, but many students find the transition from a mathematical algorithm to a working code difficult and spend a lot of time on"programming issues".

Most books on numerical methods concentrate on the mathematics of the subject while details on going from the mathematics to a computer implementation are less in focus. A major purpose of this text is therefore to help the practitioner by providing *all nuts and bolts* necessary for safely going from the mathematics to a well-designed and well-tested computer code. A significant portion of the text is consequently devoted to programming details.

Python as programming language

While MATLAB enjoys widespread popularity in books on numerical methods, we have chosen to use the Python programming language. Python is very similar to MATLAB, but contains a lot of modern software engineering tools that have become standard in the software industry and that should be adopted also for numerical computing projects. Python is at present also experiencing an exponential growth in popularity within the scientific computing community. One of the book's goals is to present an up-to-date Python eco system for implementing finite difference methods.

Program verification

Program testing, called *verification*, is a key topic of the book. Good verification techniques are indispensable when debugging computer code, but also fundamental for achieving reliable simulations. Two verification techniques saturate the book: exact solution of discrete equations (where the approximation error vanishes) and empirical estimation of convergence rates in problems with exact (analytical or manufactured) solutions of the differential equation(s).

Vectorized code

Finite difference methods lead to code with loops over large arrays. Such code in plain Python is known to run slowly. We demonstrate, especially in Appendix Chapter 8, how to port loops to fast, compiled code in C or Fortran. However, an alternative is to vectorize the code to get rid of explicit Python loops, and this technique is met throughout the book. Vectorization becomes closely connected to the underlying array library, here `numpy`, and is often thought of as a difficult subject by students. Through numerous examples in different contexts, we hope that the present book provides a substantial contribution to explaining how algorithms can be vectorized. Not

only will this speed up serial code, but with a library that can produce parallel code from `numpy` commands (such as Numba), vectorized code can be automatically turned into parallel code and utilize multi-core processors and GPUs. Also when creating tailored parallel code for today's supercomputers, vectorization is useful as it emphasizes splitting up an algorithm into plain and simple array operations, where each operation is trivial to parallelize efficiently, rather than trying to develop a "smart" overall parallelization strategy.

Analysis via exact solutions of discrete equations

Traditional asymptotic analysis of errors is important for verification of code using convergence rates, but gives a limited understanding of how and why a correctly implemented numerical method may give non-physical results. By developing exact solutions, usually based on Fourier methods, of the discrete equations, one can obtain a physical understanding of the behavior of a numerical method. This approach is favored for analysis of methods in this book.

Code-inspired mathematical notation

Our primary aim is to have a clean and easy-to-read computer code, and we want a close one-to-one relationship between the computer code and mathematical description of the algorithm. This principle calls for a mathematical notation that is governed by the natural notation in the computer code. The unknown is mostly called u, but the meaning of the symbol u in the mathematical description changes as we go from the exact solution fulfilling the differential equation to the symbol `u` that is naturally used for the associated data structure in the code.

Limited scope

The aim of this book is not to give an overview of a lot of methods for a wide range of mathematical models. Such information can be found in numerous existing, more advanced books. The aim is rather to introduce basic concepts and a thorough understanding of how to think about computing with finite difference methods. We therefore go in depth with only the most fundamental methods and equations. However, we have a multi-disciplinary scope and address the interplay of mathematics, numerics, computer science, and physics.

Focus on wave phenomena

Most books on finite difference methods, or books on theory with computer examples, have their emphasis on diffusion phenomena. Half of this book (Chapters Chapter 1, Chapter 2, and Appendix Chapter 8) is devoted to wave phenomena. Extended material on this topic is not so easy find in the literature, so the book should be a valuable contribution in this respect. Wave phenomena is also a good topic in general for choosing the finite difference method over other discretization methods since one quickly needs fine resolution over the entire mesh and uniform meshes are most natural.

Instead of introducing the finite difference method for diffusion problems, where one soon ends up with matrix systems, we do the introduction in a wave phenomena setting where explicit schemes are most relevant. This slows down the learning curve since we can introduce a lot of theory for

differences and for software aspects in a context with simple, explicit stencils for updating the solution.

Independent chapters

Most book authors are careful with avoiding repetitions of material. The chapters in this book, however, contain some overlap, because we want the chapters to appear meaningful on their own. Modern publishing technology makes it easy to take selected chapters from different books to make a new book tailored to a specific course. The more a chapter builds on details in other chapters, the more difficult it is to reuse chapters in new contexts. Also, most readers find it convenient that important information is explicitly stated, even if it was already met in another chapter.

Supplementary materials

All program and data files referred to in this book are available from the book's primary web site: URL: https://github.com/devitocodes/devito_book/.

Acknowledgments

Professor Kent-Andre Mardal at the University of Oslo has kindly contributed to enlightening discussions on several issues. Many students have provided lots of useful feedback on the exposition and found many errors in the text. Special efforts in this regard were made by Imran Ali, Shirin Fallahi, Anders Hafreager, Daniel Alexander Mo Søreide Houshmand, Kristian Gregorius Hustad, Mathilde Nygaard Kamperud, and Fatemeh Miri. The collaboration with the Springer team, with Dr.~Martin Peters, Thanh-Ha Le Thi, and their production staff has always been a great pleasure and a very efficient process.

Finally, want really appreciate the strong push from the COE of Simula Research Laboratory, Aslak Tveito, for publishing and financing books in open access format, including this one. We are grateful for the laboratory's financial contribution as well as to the financial contribution from the Department of Process, Energy and Environmental Technology at the University College of Southeast Norway.

Oslo, July 2016 — Hans Petter Langtangen, Svein Linge

Part I.

Main Chapters

1. Introduction to Devito

This chapter introduces Devito, a domain-specific language (DSL) for solving partial differential equations using finite differences. We begin with the motivation for symbolic PDE specification, then work through a complete example using the 1D wave equation.

1.1. What is Devito?

Devito is a Python-based domain-specific language (DSL) for expressing and solving partial differential equations using finite difference methods. Rather than writing low-level loops that update arrays at each time step, you write the mathematical equations symbolically and let Devito generate optimized code automatically.

1.1.1. The Traditional Approach

Consider solving the 1D diffusion equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

A traditional NumPy implementation might look like:

```
import numpy as np

# Parameters
Nx, Nt = 100, 1000
dx, dt = 0.01, 0.0001
alpha = 1.0
F = alpha * dt / dx**2  # Fourier number

# Initialize
u = np.zeros(Nx + 1)
u_new = np.zeros(Nx + 1)
u[Nx//2] = 1.0  # Initial impulse

# Time stepping loop
for n in range(Nt):
    for i in range(1, Nx):
        u_new[i] = u[i] + F * (u[i+1] - 2*u[i] + u[i-1])
    u, u_new = u_new, u  # Swap arrays
```

This approach has several limitations:

1. **Error-prone**: Manual index arithmetic is easy to get wrong
2. **Hard to optimize**: Achieving good performance requires expertise in vectorization, parallelization, and cache optimization
3. **Dimension-specific**: The code must be rewritten for 2D or 3D problems
4. **Not portable**: Optimizations for one architecture don't transfer to others

1.1.2. The Devito Approach

With Devito, the same problem becomes:

```
from devito import Grid, TimeFunction, Eq, Operator, solve, Constant

# Problem parameters
Nx = 100
L = 1.0
alpha = 1.0  # diffusion coefficient
F = 0.5      # Fourier number (for stability, F <= 0.5)

# Compute dt from stability condition: F = alpha * dt / dx^2
dx = L / Nx
dt = F * dx**2 / alpha

# Create computational grid
grid = Grid(shape=(Nx + 1,), extent=(L,))

# Define the unknown field
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# Set initial condition
u.data[0, Nx // 2] = 1.0

# Define the PDE symbolically and solve for u.forward
a = Constant(name='a')
pde = u.dt - a * u.dx2
update = Eq(u.forward, solve(pde, u.forward))

# Create and run the operator
op = Operator([update])
op(time=1000, dt=dt, a=alpha)
```

This approach offers significant advantages:

1. **Mathematical clarity**: The PDE `u.dt - a * u.dx2 = 0` is written symbolically, and Devito derives the update formula automatically using `solve()`

2. **Automatic optimization**: Devito generates C code with loop tiling, SIMD vectorization, and OpenMP parallelization
3. **Dimension-agnostic**: The same code structure works for 1D, 2D, or 3D
4. **Portable performance**: Generated code adapts to the target architecture

1.1.3. How Devito Works

Devito's workflow consists of three stages:

```
Python DSL → Symbolic Processing → C Code Generation → Compilation → Execution
```

1. **Symbolic representation**: Your Python code creates SymPy expressions that represent the PDE and its discretization
2. **Code generation**: Devito analyzes the expressions and generates optimized C code with appropriate loop structures
3. **Just-in-time compilation**: The C code is compiled (and cached) the first time the operator runs
4. **Execution**: Subsequent runs use the cached compiled code for maximum performance

1.1.4. When to Use Devito

Devito excels at:

- **Explicit time-stepping schemes**: Forward Euler, leapfrog, Runge-Kutta
- **Structured grids**: Regular Cartesian meshes in 1D, 2D, or 3D
- **Stencil computations**: Any PDE discretized with finite differences
- **Large-scale problems**: Where performance optimization matters

Common applications include:

- Wave propagation (acoustic, elastic, electromagnetic)
- Heat conduction and diffusion
- Computational fluid dynamics
- Seismic imaging (reverse time migration, full waveform inversion)

1.1.5. Installation

Devito can be installed via pip:

```
pip install devito
```

For this book, we recommend installing the optional dependencies as well:

```
pip install devito[extras]
```

This includes visualization tools and additional solvers that we'll use in later chapters.

1.1.6. What You'll Learn

In this chapter, you will:

1. Solve your first PDE (the 1D wave equation) using Devito
2. Understand the core abstractions: `Grid`, `Function`, `TimeFunction`, `Eq`, and `Operator`
3. Implement boundary conditions in Devito
4. Verify your numerical solutions using convergence testing

1.2. Your First PDE: The 1D Wave Equation

We begin our exploration of Devito with the one-dimensional wave equation, a fundamental PDE that describes vibrations in strings, sound waves in tubes, and many other physical phenomena.

1.2.1. The Mathematical Model

The 1D wave equation is:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{1.1}$$

where:

- $u(x, t)$ is the displacement at position x and time t
- c is the wave speed (a constant)

We solve this on a domain $x \in [0, L]$ for $t \in [0, T]$ with:

- **Initial conditions**: $u(x, 0) = I(x)$ and $\frac{\partial u}{\partial t}(x, 0) = 0$
- **Boundary conditions**: $u(0, t) = u(L, t) = 0$ (fixed ends)

1.2.2. Finite Difference Discretization

Using central differences in both space and time, we approximate:

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}$$
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Substituting into 1.1 and solving for u_i^{n+1}:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{1.2}$$

where $C = c\Delta t/\Delta x$ is the Courant number. The scheme is stable for $C \leq 1$.

1.2.3. The Devito Implementation

Let's implement this step by step:

```
from devito import Grid, TimeFunction, Eq, Operator
import numpy as np

# Problem parameters
L = 1.0       # Domain length
c = 1.0       # Wave speed
T = 1.0       # Final time
Nx = 100      # Number of grid points
C = 0.5       # Courant number (for stability)

# Derived parameters
dx = L / Nx
dt = C * dx / c
Nt = int(T / dt)

# Create the computational grid
grid = Grid(shape=(Nx + 1,), extent=(L,))

# Create a time-varying field
# time_order=2 because we have second derivative in time
# space_order=2 for standard second-order accuracy
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

# Set initial condition: a Gaussian pulse
x = grid.dimensions[0]
x_coord = 0.5 * L  # Center of domain
sigma = 0.1        # Width of pulse
u.data[0, :] = np.exp(-((np.linspace(0, L, Nx+1) - x_coord)**2) / (2*sigma**2))
u.data[1, :] = u.data[0, :]  # Zero initial velocity

# Define the update equation
# u.forward is u at time n+1, u is at time n, u.backward is at time n-1
# u.dx2 is the second spatial derivative
eq = Eq(u.forward, 2*u - u.backward + (c*dt)**2 * u.dx2)

# Create the operator
op = Operator([eq])

# Run the simulation
op(time=Nt, dt=dt)

# The solution is now in u.data
print(f"Simulation complete: {Nt} time steps")
print(f"Max amplitude at t={T}: {np.max(np.abs(u.data[0, :])):.6f}")
```

1.2.4. Understanding the Code

Let's examine each component:

Grid creation:

```
grid = Grid(shape=(Nx + 1,), extent=(L,))
```

This creates a 1D grid with `Nx + 1` points spanning a domain of length `L`. The grid spacing is automatically computed as `dx = L / Nx`.

TimeFunction:

```
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
```

- `name='u'`: The symbolic name for this field
- `time_order=2`: We need values at three time levels ($n-1$, n, $n+1$) for the second time derivative
- `space_order=2`: Use second-order accurate spatial stencils

Initial conditions:

```
u.data[0, :] = ...  # u at t=0
u.data[1, :] = ...  # u at t=dt (for zero initial velocity, same as t=0)
```

The `data` attribute provides direct access to the underlying NumPy arrays. Index 0 and 1 represent the two most recent time levels.

Update equation:

```
eq = Eq(u.forward, 2*u - u.backward + (c*dt)**2 * u.dx2)
```

- `u.forward`: The solution at the next time step (u^{n+1})
- `u`: The solution at the current time step (u^n)
- `u.backward`: The solution at the previous time step (u^{n-1})
- `u.dx2`: The second spatial derivative, computed using finite differences

Operator and execution:

```
op = Operator([eq])
op(time=Nt, dt=dt)
```

The `Operator` compiles the equations into optimized C code. Calling it runs the time-stepping loop for `Nt` steps with time increment `dt`.

1.2.5. Visualizing the Solution

```
import matplotlib.pyplot as plt

# Get spatial coordinates
x_vals = np.linspace(0, L, Nx + 1)

# Plot the solution at the final time
plt.figure(figsize=(10, 4))
plt.plot(x_vals, u.data[0, :], 'b-', linewidth=2)
plt.xlabel('x')
plt.ylabel('u')
plt.title(f'Wave equation solution at t = {T}')
plt.grid(True)
plt.show()
```

1.2.6. The CFL Condition

The Courant-Friedrichs-Lewy (CFL) condition states that for stability:

$$C = \frac{c\Delta t}{\Delta x} \leq 1$$

Physically, this means information cannot travel more than one grid cell per time step. If $C > 1$, the numerical solution will grow without bound.

Exercise: Try running the code with `C = 1.5` and observe what happens.

1.2.7. What Devito Does Behind the Scenes

When you create the `Operator`, Devito:

1. Analyzes the symbolic equations
2. Determines the stencil pattern and data dependencies
3. Generates optimized C code with:
    - Proper loop ordering for cache efficiency
    - SIMD vectorization where possible
    - OpenMP parallelization for multi-core execution
4. Compiles the code and caches the result

You can inspect the generated code:

```
print(op.ccode)
```

This reveals the low-level implementation that Devito creates automatically.

1.3. Core Devito Abstractions

Devito provides a small set of powerful abstractions for expressing PDEs. Understanding these building blocks is essential for writing effective Devito code.

1.3.1. Grid: The Computational Domain

The `Grid` defines the discrete domain on which we solve our PDE:

```
from devito import Grid

# 1D grid: 101 points over [0, 1]
grid_1d = Grid(shape=(101,), extent=(1.0,))

# 2D grid: 101x101 points over [0, 1] x [0, 1]
grid_2d = Grid(shape=(101, 101), extent=(1.0, 1.0))

# 3D grid: 51x51x51 points over [0, 2] x [0, 2] x [0, 2]
grid_3d = Grid(shape=(51, 51, 51), extent=(2.0, 2.0, 2.0))
```

Key properties:

- `shape`: Number of grid points in each dimension
- `extent`: Physical size of the domain
- `dimensions`: Symbolic dimension objects (x, y, z)
- `spacing`: Grid spacing in each dimension (computed automatically)

```
grid = Grid(shape=(101, 101), extent=(1.0, 1.0))
x, y = grid.dimensions      # Symbolic dimensions
dx, dy = grid.spacing       # Symbolic spacing (h_x, h_y)
print(f"Grid spacing: dx={float(dx)}, dy={float(dy)}")
```

1.3.2. Function: Static Fields

A `Function` represents a field that does not change during time-stepping. Use it for material properties, source terms, or any spatially-varying coefficient:

```
from devito import Function

grid = Grid(shape=(101,), extent=(1.0,))

# Wave velocity field
c = Function(name='c', grid=grid)
c.data[:] = 1500.0  # Constant velocity (m/s)
```

```
# Spatially varying velocity
import numpy as np
x_vals = np.linspace(0, 1, 101)
c.data[:] = 1500 + 500 * x_vals  # Linear velocity gradient
```

The `space_order` parameter controls the stencil width for derivatives:

```
# Higher-order derivatives need wider stencils
c = Function(name='c', grid=grid, space_order=4)
```

1.3.3. TimeFunction: Time-Varying Fields

A `TimeFunction` represents the solution field that evolves in time:

```
from devito import TimeFunction

grid = Grid(shape=(101,), extent=(1.0,))

# For first-order time derivatives (diffusion equation)
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# For second-order time derivatives (wave equation)
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
```

Key parameters:

- `time_order`: Number of time levels needed (1 for first derivative, 2 for second)
- `space_order`: Accuracy order for spatial derivatives

Time indexing shortcuts:

Syntax	Meaning	Mathematical notation
`u`	Current time level	u^n
`u.forward`	Next time level	u^{n+1}
`u.backward`	Previous time level	u^{n-1}
`u.dt`	First time derivative	$\partial u/\partial t$
`u.dt2`	Second time derivative	$\partial^2 u/\partial t^2$

1.3.4. Derivative Notation

Devito provides intuitive notation for spatial derivatives:

Syntax	Meaning	Stencil
`u.dx`	$\partial u/\partial x$	Centered difference
`u.dy`	$\partial u/\partial y$	Centered difference
`u.dx2`	$\partial^2 u/\partial x^2$	Second derivative
`u.dy2`	$\partial^2 u/\partial y^2$	Second derivative
`u.laplace`	$\nabla^2 u$	Laplacian (dimension-agnostic)

The `laplace` operator is particularly useful because it works in any number of dimensions:

```
# These are equivalent for 2D:
laplacian_explicit = u.dx2 + u.dy2
laplacian_auto = u.laplace

# In 3D, u.laplace automatically becomes u.dx2 + u.dy2 + u.dz2
```

1.3.5. Eq: Defining Equations

The `Eq` class creates symbolic equations:

```
from devito import Eq

# Explicit update: u^{n+1} = expression
update = Eq(u.forward, 2*u - u.backward + dt**2 * c**2 * u.laplace)

# Using the solve() helper for implicit forms
from devito import solve

pde = u.dt2 - c**2 * u.laplace  # The PDE residual
update = Eq(u.forward, solve(pde, u.forward))
```

The `solve()` function is useful when the update formula is complex. It symbolically solves for the target variable.

1.3.6. Operator: Compilation and Execution

The `Operator` takes a list of equations and generates executable code:

```
from devito import Operator

# Single equation
op = Operator([update])

# Multiple equations (e.g., with boundary conditions)
op = Operator([update, bc_left, bc_right])
```

```
# Run for Nt time steps
op(time=Nt, dt=dt)
```

The operator compiles the equations into optimized C code on first execution. Subsequent calls reuse the cached compiled code.

1.3.7. Complete Example: 2D Diffusion

Let's put these abstractions together for a 2D diffusion problem:

```
from devito import Grid, TimeFunction, Eq, Operator
import numpy as np

# Create a 2D grid
grid = Grid(shape=(101, 101), extent=(1.0, 1.0))

# Time-varying field (first-order in time for diffusion)
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# Parameters
alpha = 0.1  # Diffusion coefficient
dx = 1.0 / 100
F = 0.25     # Fourier number (for stability)
dt = F * dx**2 / alpha

# Initial condition: hot spot in the center
u.data[0, 45:55, 45:55] = 1.0

# The diffusion equation: u_t = alpha * (u_xx + u_yy)
# Using .laplace for dimension-agnostic code
eq = Eq(u.forward, u + alpha * dt * u.laplace)

# Create and run
op = Operator([eq])
op(time=500, dt=dt)

# Visualize
import matplotlib.pyplot as plt
plt.imshow(u.data[0, :, :], origin='lower', cmap='hot')
plt.colorbar(label='Temperature')
plt.title('2D Diffusion')
plt.show()
```

1.3.8. Summary of Core Abstractions

Abstraction	Purpose	Key Parameters
`Grid`	Define computational domain	`shape`, `extent`
`Function`	Static fields (coefficients)	`name`, `grid`, `space_order`
`TimeFunction`	Time-varying fields	`name`, `grid`, `time_order`, `space_order`
`Eq`	Define equations	LHS, RHS
`Operator`	Compile and execute	List of equations

These five abstractions form the foundation of all Devito programs. In the following sections, we'll see how to handle boundary conditions and verify our numerical solutions.

1.4. Boundary Conditions in Devito

Properly implementing boundary conditions is crucial for accurate PDE solutions. Devito provides several approaches, each suited to different situations.

1.4.1. Dirichlet Boundary Conditions

Dirichlet conditions specify the solution value at the boundary:

$$u(0,t) = g_0(t), \quad u(L,t) = g_L(t)$$

Method 1: Explicit equations

The most direct approach adds equations that set boundary values:

```
from devito import Grid, TimeFunction, Eq, Operator

grid = Grid(shape=(101,), extent=(1.0,))
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

# Get the time dimension for indexing
t = grid.stepping_dim

# Interior update (wave equation)
update = Eq(u.forward, 2*u - u.backward + dt**2 * c**2 * u.dx2)

# Boundary conditions: u = 0 at both ends
bc_left = Eq(u[t+1, 0], 0)
bc_right = Eq(u[t+1, 100], 0)

# Include all equations in the operator
op = Operator([update, bc_left, bc_right])
```

Method 2: Using subdomain

For interior-only updates, use `subdomain=grid.interior`:

```
# Update only interior points (automatically excludes boundaries)
update = Eq(u.forward, 2*u - u.backward + dt**2 * c**2 * u.dx2,
            subdomain=grid.interior)

# Set boundaries explicitly
bc_left = Eq(u[t+1, 0], 0)
bc_right = Eq(u[t+1, 100], 0)

op = Operator([update, bc_left, bc_right])
```

The `subdomain=grid.interior` approach is often cleaner because it explicitly separates the physics (interior PDE) from the boundary treatment.

1.4.2. Neumann Boundary Conditions

Neumann conditions specify the derivative at the boundary:

$$\frac{\partial u}{\partial x}(0,t) = h_0(t), \quad \frac{\partial u}{\partial x}(L,t) = h_L(t)$$

For a zero-flux condition ($\partial u/\partial x = 0$), we use the ghost point method. The central difference at the boundary requires a point outside the domain:

$$\left.\frac{\partial u}{\partial x}\right|_{i=0} \approx \frac{u_1 - u_{-1}}{2\Delta x} = 0$$

This gives $u_{-1} = u_1$, which we substitute into the interior equation:

```
grid = Grid(shape=(101,), extent=(1.0,))
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)
x = grid.dimensions[0]
t = grid.stepping_dim

# Interior update (diffusion equation)
update = Eq(u.forward, u + alpha * dt * u.dx2, subdomain=grid.interior)

# Neumann BC at left (du/dx = 0): use one-sided update
# u_new[0] = u[0] + alpha*dt * 2*(u[1] - u[0])/dx^2
dx = grid.spacing[0]
bc_left = Eq(u[t+1, 0], u[t, 0] + alpha * dt * 2 * (u[t, 1] - u[t, 0]) / dx**2)

# Neumann BC at right (du/dx = 0)
bc_right = Eq(u[t+1, 100], u[t, 100] + alpha * dt * 2 * (u[t, 99] - u[t, 100]) / dx**2)

op = Operator([update, bc_left, bc_right])
```

1.4.3. Mixed Boundary Conditions

Often we have different conditions on different boundaries:

```
# Dirichlet on left, Neumann on right
bc_left = Eq(u[t+1, 0], 0)  # u(0,t) = 0
bc_right = Eq(u[t+1, 100], u[t+1, 99])  # du/dx(L,t) = 0 (copy from interior)

op = Operator([update, bc_left, bc_right])
```

1.4.4. 2D Boundary Conditions

For 2D problems, boundary conditions apply to all four edges:

```
grid = Grid(shape=(101, 101), extent=(1.0, 1.0))
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

x, y = grid.dimensions
t = grid.stepping_dim
Nx, Ny = 100, 100

# Interior update
update = Eq(u.forward, 2*u - u.backward + dt**2 * c**2 * u.laplace,
            subdomain=grid.interior)

# Dirichlet BCs on all four edges
bc_left = Eq(u[t+1, 0, y], 0)
bc_right = Eq(u[t+1, Nx, y], 0)
bc_bottom = Eq(u[t+1, x, 0], 0)
bc_top = Eq(u[t+1, x, Ny], 0)

op = Operator([update, bc_left, bc_right, bc_bottom, bc_top])
```

1.4.5. Time-Dependent Boundary Conditions

For boundaries that vary in time, use the time index:

```
from devito import Constant

# Time-varying amplitude
A = Constant(name='A')

# Sinusoidal forcing at left boundary
# u(0, t) = A * sin(omega * t)
import sympy as sp
omega = 2 * sp.pi  # Angular frequency
```

```
# The time value at step n
t_val = t * dt  # Symbolic time value

bc_left = Eq(u[t+1, 0], A * sp.sin(omega * t_val))

# Set the amplitude before running
op = Operator([update, bc_left, bc_right])
op(time=Nt, dt=dt, A=1.0)  # Pass A as keyword argument
```

1.4.6. Absorbing Boundary Conditions

For wave equations, we often want waves to exit the domain without reflection. A simple first-order absorbing condition is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \quad \text{at } x = L$$

This can be discretized as:

```
# Absorbing BC at right boundary (waves traveling right)
dx = grid.spacing[0]
bc_right_absorbing = Eq(
    u[t+1, Nx],
    u[t, Nx] - c * dt / dx * (u[t, Nx] - u[t, Nx-1])
)
```

More sophisticated absorbing conditions use damping layers (sponges) near the boundaries. This is covered in detail in Section 2.13.10.

1.4.7. Periodic Boundary Conditions

For periodic domains, the solution wraps around:

$$u(0, t) = u(L, t)$$

Devito doesn't directly support periodic BCs, but they can be implemented by copying values:

```
# Periodic BCs: u[0] = u[Nx-1], u[Nx] = u[1]
bc_periodic_left = Eq(u[t+1, 0], u[t+1, Nx-1])
bc_periodic_right = Eq(u[t+1, Nx], u[t+1, 1])
```

Note: The order of equations matters. Update the interior first, then copy for periodicity.

1.4.8. Best Practices

1. **Use `subdomain=grid.interior`** for interior updates to clearly separate physics from boundary treatment
2. **Check boundary equation order**: Boundary equations should typically come after interior updates in the operator
3. **Verify boundary values**: After running, check that boundaries have the expected values
4. **Test with known solutions**: Use problems with analytical solutions to verify boundary condition implementation

1.4.9. Example: Complete Wave Equation Solver

Here's a complete example combining interior updates with boundary conditions:

```
from devito import Grid, TimeFunction, Eq, Operator
import numpy as np

# Setup
L, c, T = 1.0, 1.0, 2.0
Nx = 100
C = 0.9  # Courant number
dx = L / Nx
dt = C * dx / c
Nt = int(T / dt)

# Grid and field
grid = Grid(shape=(Nx + 1,), extent=(L,))
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
t = grid.stepping_dim

# Initial condition: plucked string
x_vals = np.linspace(0, L, Nx + 1)
u.data[0, :] = np.sin(np.pi * x_vals)
u.data[1, :] = u.data[0, :]  # Zero initial velocity

# Equations
update = Eq(u.forward, 2*u - u.backward + (c*dt)**2 * u.dx2,
            subdomain=grid.interior)
bc_left = Eq(u[t+1, 0], 0)
bc_right = Eq(u[t+1, Nx], 0)

# Solve
op = Operator([update, bc_left, bc_right])
op(time=Nt, dt=dt)
```

```
# Verify: solution should return to initial shape at t = 2L/c
print(f"Initial max: {np.max(u.data[1, :]):.6f}")
print(f"Final max: {np.max(u.data[0, :]):.6f}")
```

For a string with fixed ends and initial shape $\sin(\pi x)$, the solution oscillates with period $2L/c$. After one period, it should return to the initial configuration.

1.5. Verification and Convergence Testing

How do we know our numerical solution is correct? Verification is the process of confirming that our code correctly solves the mathematical equations we intended. This section introduces key verification techniques.

1.5.1. The Importance of Verification

Numerical codes can produce plausible-looking but incorrect results due to:

- Programming errors (typos, off-by-one errors)
- Incorrect boundary condition implementation
- Stability violations
- Insufficient resolution

Systematic verification catches these problems before they corrupt scientific results.

1.5.2. Convergence Rate Testing

The most powerful verification technique is convergence rate testing. For a scheme with truncation error $O(\Delta x^p)$, the error should decrease as:

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

By measuring errors at different resolutions, we can estimate p:

$$p \approx \frac{\log(E_1/E_2)}{\log(\Delta x_1/\Delta x_2)}$$

If the measured rate matches the theoretical order, we have strong evidence the implementation is correct.

1.5.3. Implementing a Convergence Test

```python
import numpy as np
from devito import Grid, TimeFunction, Eq, Operator

def solve_wave_equation(Nx, L=1.0, T=0.5, c=1.0, C=0.5):
    """Solve 1D wave equation and return error vs exact solution."""

    dx = L / Nx
    dt = C * dx / c
    Nt = int(T / dt)

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
    t_dim = grid.stepping_dim

    # Initial condition: sin(pi*x)
    x_vals = np.linspace(0, L, Nx + 1)
    u.data[0, :] = np.sin(np.pi * x_vals)
    u.data[1, :] = np.sin(np.pi * x_vals) * np.cos(np.pi * c * dt)

    # Wave equation
    update = Eq(u.forward, 2*u - u.backward + (c*dt)**2 * u.dx2,
                subdomain=grid.interior)
    bc_left = Eq(u[t_dim+1, 0], 0)
    bc_right = Eq(u[t_dim+1, Nx], 0)

    op = Operator([update, bc_left, bc_right])
    op(time=Nt, dt=dt)

    # Exact solution: u(x,t) = sin(pi*x)*cos(pi*c*t)
    t_final = Nt * dt
    u_exact = np.sin(np.pi * x_vals) * np.cos(np.pi * c * t_final)

    # Return max error
    error = np.max(np.abs(u.data[0, :] - u_exact))
    return error, dx


def convergence_test(grid_sizes):
    """Run convergence test and compute rates."""

    errors = []
    dx_values = []

    for Nx in grid_sizes:
        error, dx = solve_wave_equation(Nx)
        errors.append(error)
        dx_values.append(dx)
```

```
        print(f"Nx = {Nx:4d}, dx = {dx:.6f}, error = {error:.6e}")

    # Compute convergence rates
    rates = []
    for i in range(len(errors) - 1):
        rate = np.log(errors[i] / errors[i+1]) / np.log(dx_values[i] / dx_values[i+1])
        rates.append(rate)

    print("\nConvergence rates:")
    for i, rate in enumerate(rates):
        print(f"  {grid_sizes[i]} -> {grid_sizes[i+1]}: rate = {rate:.2f}")

    return errors, dx_values, rates


# Run the test
grid_sizes = [20, 40, 80, 160, 320]
errors, dx_values, rates = convergence_test(grid_sizes)

# Check: rates should be close to 2 for second-order scheme
expected_rate = 2.0
assert all(abs(r - expected_rate) < 0.2 for r in rates), \
    f"Convergence rates {rates} differ from expected {expected_rate}"
```

1.5.4. Method of Manufactured Solutions (MMS)

For problems without analytical solutions, we use the Method of Manufactured Solutions:

1. **Choose** a solution $u_{\text{mms}}(x, t)$ (any smooth function)
2. **Compute** the source term by substituting into the PDE
3. **Solve** the modified PDE with the computed source
4. **Compare** the numerical solution to u_{mms}

Example: Diffusion equation

Let's verify a diffusion solver using MMS:

```
import sympy as sp

# Symbolic variables
x_sym, t_sym = sp.symbols('x t')
alpha_sym = sp.Symbol('alpha')

# Manufactured solution (arbitrary smooth function)
u_mms = sp.sin(sp.pi * x_sym) * sp.exp(-t_sym)

# Compute required source term: f = u_t - alpha * u_xx
```

```
u_t = sp.diff(u_mms, t_sym)
u_xx = sp.diff(u_mms, x_sym, 2)
f_mms = u_t - alpha_sym * u_xx

print("Manufactured solution:")
print(f"  u_mms = {u_mms}")
print(f"Required source term:")
print(f"  f = {sp.simplify(f_mms)}")
```

Now implement the solver with this source term:

```
from devito import Grid, TimeFunction, Function, Eq, Operator
import numpy as np

def solve_diffusion_mms(Nx, alpha=1.0, T=0.5, F=0.4):
    """Solve diffusion with MMS source term."""

    L = 1.0
    dx = L / Nx
    dt = F * dx**2 / alpha
    Nt = int(T / dt)

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)
    t_dim = grid.stepping_dim

    # Spatial coordinates for evaluation
    x_vals = np.linspace(0, L, Nx + 1)

    # MMS: u = sin(pi*x) * exp(-t)
    # Source: f = sin(pi*x) * exp(-t) * (alpha*pi^2 - 1)
    def u_exact(x, t):
        return np.sin(np.pi * x) * np.exp(-t)

    def f_source(x, t):
        return np.sin(np.pi * x) * np.exp(-t) * (alpha * np.pi**2 - 1)

    # Initial condition from MMS
    u.data[0, :] = u_exact(x_vals, 0)

    # We need to add source term at each time step
    # For simplicity, use time-lagged source
    f = Function(name='f', grid=grid)

    # Update equation with source
    update = Eq(u.forward, u + alpha * dt * u.dx2 + dt * f,
                subdomain=grid.interior)
```

```
    bc_left = Eq(u[t_dim+1, 0], 0)  # u_mms(0,t) = 0
    bc_right = Eq(u[t_dim+1, Nx], 0)  # u_mms(1,t) = 0

    op = Operator([update, bc_left, bc_right])

    # Time stepping with source update
    for n in range(Nt):
        t_current = n * dt
        f.data[:] = f_source(x_vals, t_current)
        op(time=1, dt=dt)

    # Compare to exact solution
    t_final = Nt * dt
    u_exact_final = u_exact(x_vals, t_final)
    error = np.max(np.abs(u.data[0, :] - u_exact_final))

    return error, dx


# Convergence test with MMS
print("MMS Convergence Test for Diffusion Equation:")
grid_sizes = [20, 40, 80, 160]
errors = []
dx_vals = []

for Nx in grid_sizes:
    error, dx = solve_diffusion_mms(Nx)
    errors.append(error)
    dx_vals.append(dx)
    print(f"Nx = {Nx:4d}, error = {error:.6e}")

# Compute rates
for i in range(len(errors) - 1):
    rate = np.log(errors[i] / errors[i+1]) / np.log(2)
    print(f"Rate {grid_sizes[i]}->{grid_sizes[i+1]}: {rate:.2f}")
```

1.5.5. Quick Verification Checks

Before running full convergence tests, use these quick checks:

1. Conservation properties

For problems that should conserve mass or energy:

```
# Check mass conservation for diffusion with Neumann BCs
mass_initial = np.sum(u.data[1, :]) * dx
```

```
mass_final = np.sum(u.data[0, :]) * dx
print(f"Mass change: {abs(mass_final - mass_initial):.2e}")
```

2. Symmetry

For symmetric initial conditions and domains:

```
# Check symmetry is preserved
u_left = u.data[0, :Nx//2]
u_right = u.data[0, Nx//2+1:][::-1]  # Reversed
symmetry_error = np.max(np.abs(u_left - u_right))
print(f"Symmetry error: {symmetry_error:.2e}")
```

3. Steady state

For problems with known steady states:

```
# Run to steady state and check
u_steady_numerical = u.data[0, :]
u_steady_exact = ...  # Known analytical steady state
error = np.max(np.abs(u_steady_numerical - u_steady_exact))
```

1.5.6. Debugging Tips

When convergence tests fail:

1. **Check boundary conditions**: Are they correctly implemented? Plot the solution near boundaries.
2. **Check stability**: Is the CFL/Fourier number within limits? Try smaller time steps.
3. **Check initial conditions**: Are they set correctly? Verify `u.data[0, :]` and `u.data[1, :]`.
4. **Inspect generated code**: Use `print(op.ccode)` to see what Devito actually computes.
5. **Test components separately**: Verify spatial derivatives work on known functions before testing full PDE.

1.5.7. Summary

Verification is essential for trustworthy numerical results:

Technique	When to Use	What It Checks
Convergence testing	Always	Correct order of accuracy
MMS	No analytical solution	Correct PDE implementation
Conservation	Physics requires it	No spurious sources/sinks

Technique	When to Use	What It Checks
Symmetry	Symmetric problems	Consistent treatment

A well-verified code gives confidence that results represent the physics, not numerical artifacts.

2. Wave Equations

Computational algorithms: Can they be briefly stated in words and then shown directly in Python code rather than in an Algorithm box? Think so, for wave1D we can say dx, dt, etc what they are and then first show the core of the algorithm. Thereafter the complete function and sample call.

Examples:

- Debugging: constant solution when we have Neumann conditions.
- Verification: convergence test, example with $h = \Delta x = \Delta t$.
- Make file database for solutions, 1D, 2D, 3D.
- guitar string, triangular, C=1
- a different C, ok solution
- $C > 1$ instability
- moving left
- plug, C=1
- plug, C=0.95
- spherical waves
- Software: put spatial update in a separate function, could introduce a version with a class for Mesh, Function (w/interpolation)
- Develop study guides for each file or module

2D: lots of implementations (Fortran, Instant C++, Cython, vectorized)

A very wide range of physical processes lead to wave motion, where signals are propagated through a medium in space and time, normally with little or no permanent movement of the medium itself. The shape of the signals may undergo changes as they travel through matter, but usually not so much that the signals cannot be recognized at some later point in space and time. Many types of wave motion can be described by the equation $u_{tt} = \nabla \cdot (c^2 \nabla u) + f$, which we will solve in the forthcoming text by finite difference methods.

2.1. Simulation of waves on a string

We begin our study of wave equations by simulating one-dimensional waves on a string, say on a guitar or violin. Let the string in the undeformed state coincide with the interval $[0, L]$ on the x axis, and let $u(x, t)$ be the displacement at time t in the y direction of a point initially at x. The displacement function u is governed by the mathematical model

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L),\ t \in (0, T] \tag{2.1}$$

$$u(x, 0) = I(x), \quad x \in [0, L] \tag{2.2}$$

$$\frac{\partial}{\partial t}u(x,0) = 0, \quad x \in [0,L] \tag{2.3}$$

$$u(0,t) = 0, \quad t \in (0,T] \tag{2.4}$$

$$u(L,t) = 0, \quad t \in (0,T] \tag{2.5}$$

The constant c and the function $I(x)$ must be prescribed.

Equation (2.1) is known as the one-dimensional *wave equation*. Since this PDE contains a second-order derivative in time, we need *two initial conditions*. The condition (2.2) specifies the initial shape of the string, $I(x)$, and (2.3) expresses that the initial velocity of the string is zero. In addition, PDEs need *boundary conditions*, given here as (2.4) and (2.5). These two conditions specify that the string is fixed at the ends, i.e., that the displacement u is zero.

The solution $u(x,t)$ varies in space and time and describes waves that move with velocity c to the left and right.

Sometimes we will use a more compact notation for the partial derivatives to save space:

$$u_t = \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2},$$

and similar expressions for derivatives with respect to other variables. Then the wave equation can be written compactly as $u_{tt} = c^2 u_{xx}$.

The PDE problem (2.1)-(2.5) will now be discretized in space and time by a finite difference method.

2.2. Discretizing the domain

The temporal domain $[0,T]$ is represented by a finite number of mesh points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,.$$

Similarly, the spatial domain $[0,L]$ is replaced by a set of mesh points

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L\,.$$

One may view the mesh as two-dimensional in the x,t plane, consisting of points (x_i, t_n), with $i = 0,\ldots,N_x$ and $n = 0,\ldots,N_t$.

2.2.1. Uniform meshes

For uniformly distributed mesh points we can introduce the constant mesh spacings Δt and Δx. We have that

$$x_i = i\Delta x,\ i = 0,\ldots,N_x, \quad t_n = n\Delta t,\ n = 0,\ldots,N_t\,.$$

We also have that $\Delta x = x_i - x_{i-1}$, $i = 1,\ldots,N_x$, and $\Delta t = t_n - t_{n-1}$, $n = 1,\ldots,N_t$. Figure Figure 2.1 displays a mesh in the x,t plane with $N_t = 5$, $N_x = 5$, and constant mesh spacings.

2.3. The discrete solution

The solution $u(x,t)$ is sought at the mesh points. We introduce the mesh function u_i^n, which approximates the exact solution at the mesh point (x_i, t_n) for $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$. Using the finite difference method, we shall develop algebraic equations for computing the mesh function.

2.4. Fulfilling the equation at the mesh points

In the finite difference method, we relax the condition that (2.1) holds at all points in the space-time domain $(0, L) \times (0, T]$ to the requirement that the PDE is fulfilled at the *interior* mesh points only:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \tag{2.6}$$

for $i = 1, \ldots, N_x - 1$ and $n = 1, \ldots, N_t - 1$. For $n = 0$ we have the initial conditions $u = I(x)$ and $u_t = 0$, and at the boundaries $i = 0, N_x$ we have the boundary condition $u = 0$.

2.5. Replacing derivatives by finite differences

The second-order derivatives can be replaced by central differences. The most widely used difference approximation of the second-order derivative is

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}.$$

It is convenient to introduce the finite difference operator notation

$$[D_t D_t u]_i^n = \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}.$$

A similar approximation of the second-order derivative in the x direction reads

$$\frac{\partial^2}{\partial x^2} u(x_i, t_n) \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} = [D_x D_x u]_i^n.$$

Algebraic version of the PDE We can now replace the derivatives in (2.6) and get

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{2.7}$$

or written more compactly using the operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n. \tag{2.8}$$

2.5.1. Interpretation of the equation as a stencil

A characteristic feature of (2.7) is that it involves u values from neighboring points only: u_i^{n+1}, $u_{i\pm1}^n$, u_i^n, and u_i^{n-1}. The circles in Figure Figure 2.1 illustrate such neighboring mesh points that contribute to an algebraic equation. In this particular case, we have sampled the PDE at the point $(2, 2)$ and constructed (2.7), which then involves a coupling of u_1^2, u_2^3, u_2^2, u_2^1, and u_3^2. The term *stencil* is often used about the algebraic equation at a mesh point, and the geometry of a typical stencil is illustrated in Figure Figure 2.1. One also often refers to the algebraic equations as *discrete equations*, *(finite) difference equations* or a *finite difference scheme*.

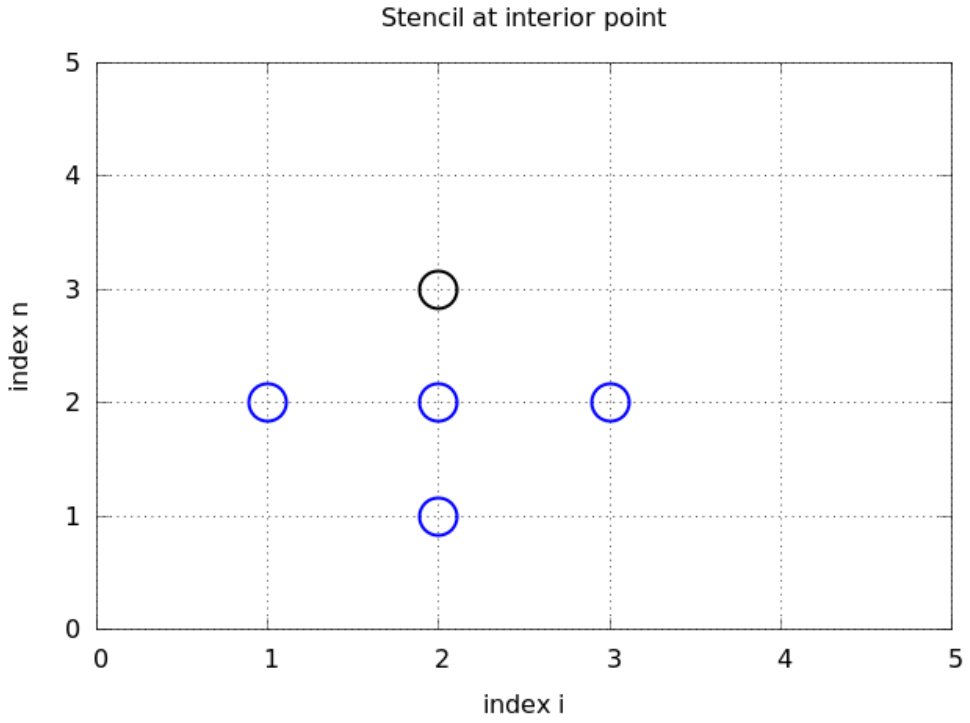


Figure 2.1.: Mesh in space and time. The circles show points connected in a finite difference equation.

2.5.2. Algebraic version of the initial conditions

We also need to replace the derivative in the initial condition (2.3) by a finite difference approximation. A centered difference of the type

$$\frac{\partial}{\partial t}u(x_i, t_0) \approx \frac{u_i^1 - u^{-1} * *i}{2\Delta t} = [D * *2tu]_i^0,$$

seems appropriate. Writing out this equation and ordering the terms give

$$u_i^{-1} = u_i^1, \quad i = 0, \ldots, N_x\,. \tag{2.9}$$

The other initial condition can be computed by

$$u_i^0 = I(x_i), \quad i = 0, \ldots, N_x\,.$$

2.6. Formulating a recursive algorithm

We assume that u_i^n and u_i^{n-1} are available for $i = 0, \ldots, N_x$. The only unknown quantity in (2.7) is therefore u_i^{n+1}, which we now can solve for:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u^n **i + 1 - 2u^n **i + u_{i-1}^n\right) . \tag{2.10}$$

We have here introduced the parameter

$$C = c\frac{\Delta t}{\Delta x},$$

known as the *Courant number*.

> **i** C is the key parameter in the discrete wave equation
>
> We see that the discrete version of the PDE features only one parameter, C, which is therefore the key parameter, together with N_x, that governs the quality of the numerical solution (see Section Section 2.61 for details). Both the primary physical parameter c and the numerical parameters Δx and Δt are lumped together in C. Note that C is a dimensionless parameter.

Given that u_i^{n-1} and u_i^n are known for $i = 0, \ldots, N_x$, we find new values at the next time level by applying the formula (2.10) for $i = 1, \ldots, N_x - 1$. Figure Figure 2.1 illustrates the points that are used to compute u_2^3. For the boundary points, $i = 0$ and $i = N_x$, we apply the boundary conditions $u_i^{n+1} = 0$.

Even though sound reasoning leads up to (2.10), there is still a minor challenge with it that needs to be resolved. Think of the very first computational step to be made. The scheme (2.10) is supposed to start at $n = 1$, which means that we compute u^2 from u^1 and u^0. Unfortunately, we do not know the value of u^1, so how to proceed? A standard procedure in such cases is to apply (2.10) also for $n = 0$. This immediately seems strange, since it involves u_i^{-1}, which is an undefined quantity outside the time mesh (and the time domain). However, we can use the initial condition (2.9) in combination with (2.10) when $n = 0$ to eliminate u_i^{-1} and arrive at a special formula for u_i^1:

$$u_i^1 = u_i^0 - \frac{1}{2}C^2 \left(u^0 **i + 1 - 2u^0 **i + u_{i-1}^0\right) . \tag{2.11}$$

Figure Figure 2.2 illustrates how (2.11) connects four instead of five points: u_2^1, u_1^0, u_2^0, and u_3^0.

We can now summarize the computational algorithm:

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. Compute u_i^1 by (2.11) for $i = 1, 2, \ldots, N_x - 1$ and set $u_i^1 = 0$ for the boundary points given by $i = 0$ and $i = N_x$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
4. apply (2.10) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
5. set $u_i^{n+1} = 0$ for the boundary points having $i = 0$, $i = N_x$.

The algorithm essentially consists of moving a finite difference stencil through all the mesh points.

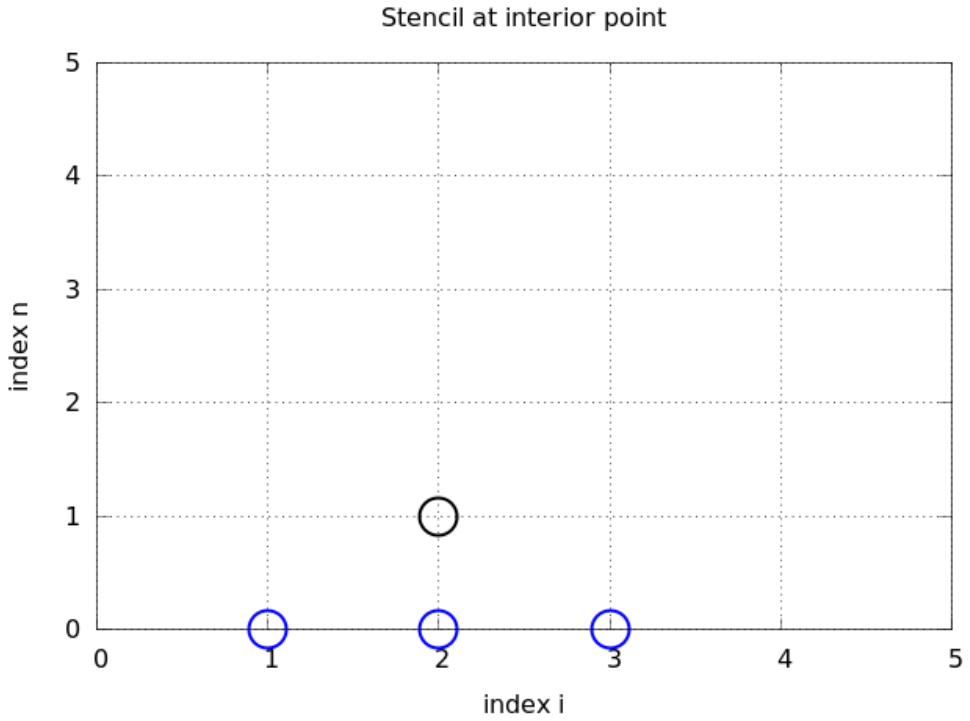


Figure 2.2.: Modified stencil for the first time step.

2.7. Sketch of an implementation

The algorithm only involves the three most recent time levels, so we need only three arrays for u_i^{n+1}, u_i^n, and u_i^{n-1}, $i = 0, \ldots, N_x$. Storing all the solutions in a two-dimensional array of size $(N_x + 1) \times (N_t + 1)$ would be possible in this simple one-dimensional PDE problem, but is normally out of the question in three-dimensional (3D) and large two-dimensional (2D) problems. We shall therefore, in all our PDE solving programs, have the unknown in memory at as few time levels as possible.

In a Python implementation of this algorithm, we use the array elements `u[i]` to store u_i^{n+1}, `u_n[i]` to store u_i^n, and `u_nm1[i]` to store u_i^{n-1}.

The following Python snippet realizes the steps in the computational algorithm.

```
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx             # Courant number
Nt = len(t)-1
C2 = C**2               # Help variable in the scheme

for i in range(0, Nx+1):
    u_n[i] = I(x[i])

for i in range(1, Nx):
```

```
    u[i] = u_n[i] - \
           0.5*C**2(u_n[i+1] - 2*u_n[i] + u_n[i-1])
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

u_nm1[:], u_n[:] = u_n, u

for n in range(1, Nt):
    for i in range(1, Nx):
        u[i] = 2u_n[i] - u_nm1[i] - \
               C**2(u_n[i+1] - 2*u_n[i] + u_n[i-1])

    u[0] = 0;  u[Nx] = 0

    u_nm1[:], u_n[:] = u_n, u
```

Before implementing the algorithm, it is convenient to add a source term to the PDE (2.1), since that gives us more freedom in finding test problems for verification. Physically, a source term acts as a generator for waves in the interior of the domain.

2.8. A slightly generalized model problem

We now address the following extended initial-boundary value problem for one-dimensional wave phenomena:

$$u_{tt} = c^2 u_{xx} + f(x,t), \quad x \in (0,L),\ t \in (0,T] \tag{2.12}$$

$$u(x,0) = I(x), \quad x \in [0,L] \tag{2.13}$$

$$u_t(x,0) = V(x), \quad x \in [0,L] \tag{2.14}$$

$$u(0,t) = 0, \quad t > 0 \tag{2.15}$$

$$u(L,t) = 0, \quad t > 0 \tag{2.16}$$

Sampling the PDE at (x_i, t_n) and using the same finite difference approximations as above, yields

$$[D_t D_t u = c^2 D_x D_x u + f]_i^n \,. \tag{2.17}$$

Writing this out and solving for the unknown u_i^{n+1} results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u^n **i + 1 - 2u^n **i + u_{i-1}^n) + \Delta t^2 f_i^n \,. \tag{2.18}$$

The equation for the first time step must be rederived. The discretization of the initial condition $u_t = V(x)$ at $t = 0$ becomes

$$[D_{2t} u = V]_i^0 \quad \Rightarrow \quad u_i^{-1} = u_i^1 - 2\Delta t V_i,$$

which, when inserted in (2.18) for $n = 0$, gives the special formula

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2} C^2 \left(u^0 **i + 1 - 2u^0 **i + u_{i-1}^0 \right) + \frac{1}{2} \Delta t^2 f_i^0 \,. \tag{2.19}$$

2.9. Using an analytical solution of physical significance

Many wave problems feature sinusoidal oscillations in time and space. For example, the original PDE problem (2.1)-(2.5) allows an exact solution

$$u_{\mathrm{e}}(x,t) = A\sin\left(\frac{\pi}{L}x\right)\cos\left(\frac{\pi}{L}ct\right)\,. \tag{2.20}$$

This u_{e} fulfills the PDE with $f=0$, boundary conditions $u_{\mathrm{e}}(0,t)=u_{\mathrm{e}}(L,t)=0$, as well as initial conditions $I(x)=A\sin\left(\frac{\pi}{L}x\right)$ and $V=0$.

How to use exact solutions for verification

It is common to use such exact solutions of physical interest to verify implementations. However, the numerical solution u_i^n will only be an approximation to $u_{\mathrm{e}}(x_i,t_n)$. We have no knowledge of the precise size of the error in this approximation, and therefore we can never know if discrepancies between u_i^n and $u_{\mathrm{e}}(x_i,t_n)$ are caused by mathematical approximations or programming errors. In particular, if plots of the computed solution u_i^n and the exact one (2.20) look similar, many are tempted to claim that the implementation works. However, even if color plots look nice and the accuracy is "deemed good", there can still be serious programming errors present!
The only way to use exact physical solutions like (2.20) for serious and thorough verification is to run a series of simulations on finer and finer meshes, measure the integrated error in each mesh, and from this information estimate the empirical convergence rate of the method.

An introduction to the computing of convergence rates is given in Section 3.1.6 in (Langtangen 2016b). There is also a detailed example on computing convergence rates in Section 1.5.

In the present problem, one expects the method to have a convergence rate of 2 (see Section Section 2.61), so if the computed rates are close to 2 on a sufficiently fine mesh, we have good evidence that the implementation is free of programming mistakes.

2.10. Manufactured solution and estimation of convergence rates

2.10.1. Specifying the solution and computing corresponding data

One problem with the exact solution (2.20) is that it requires a simplification ($V=0, f=0$) of the implemented problem (2.12)-(2.16). An advantage of using a *manufactured solution* is that we can test all terms in the PDE problem. The idea of this approach is to set up some chosen solution and fit the source term, boundary conditions, and initial conditions to be compatible with the chosen solution. Given that our boundary conditions in the implementation are $u(0,t)=u(L,t)=0$, we must choose a solution that fulfills these conditions. One example is

$$u_{\mathrm{e}}(x,t) = x(L-x)\sin t\,.$$

Inserted in the PDE $u_{tt}=c^2u_{xx}+f$ we get

$$-x(L-x)\sin t = -c^2 2\sin t + f \quad \Rightarrow f = (2c^2 - x(L-x))\sin t\,.$$

The initial conditions become

$$\begin{aligned} u(x,0) =& I(x) = 0, \\ & u_t(x,0) = V(x) = x(L-x)\,. \end{aligned}$$

2.10.2. Defining a single discretization parameter

To verify the code, we compute the convergence rates in a series of simulations, letting each simulation use a finer mesh than the previous one. Such empirical estimation of convergence rates relies on an assumption that some measure E of the numerical error is related to the discretization parameters through

$$E = C_t\Delta t^r + C_x\Delta x^p,$$

where C_t, C_x, r, and p are constants. The constants r and p are known as the *convergence rates* in time and space, respectively. From the accuracy in the finite difference approximations, we expect $r = p = 2$, since the error terms are of order Δt^2 and Δx^2. This is confirmed by truncation error analysis and other types of analysis.

By using an exact solution of the PDE problem, we will next compute the error measure E on a sequence of refined meshes and see if the rates $r = p = 2$ are obtained. We will not be concerned with estimating the constants C_t and C_x, simply because we are not interested in their values.

It is advantageous to introduce a single discretization parameter $h = \Delta t = \hat{c}\Delta x$ for some constant $\hat{c}$. Since Δt and Δx are related through the Courant number, $\Delta t = C\Delta x/c$, we set $h = \Delta t$, and then $\Delta x = hc/C$. Now the expression for the error measure is greatly simplified:

$$E = C_t\Delta t^r + C_x\Delta x^r = C_t h^r + C_x\left(\frac{c}{C}\right)^r h^r = Dh^r, \quad D = C_t + C_x\left(\frac{c}{C}\right)^r.$$

Computing errors We choose an initial discretization parameter h_0 and run experiments with decreasing h: $h_i = 2^{-i}h_0$, $i = 1, 2, \ldots, m$. Halving h in each experiment is not necessary, but it is a common choice. For each experiment we must record E and h. Standard choices of error measure are the ℓ^2 and ℓ^∞ norms of the error mesh function e_i^n:

$$E = ||e_i^n||_{\ell^2} = \left(\Delta t\Delta x\sum_{n=0}^{N_t}\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i,t_n) - u_i^n, \tag{2.21}$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{i,n}|e_i^n|\,. \tag{2.22}$$

In Python, one can compute $\sum_i(e_i^n)^2$ at each time step and accumulate the value in some sum variable, say `e2_sum`. At the final time step one can do `sqrt(dt*dx*e2_sum)`. For the ℓ^∞ norm one must compare the maximum error at a time level (`e.max()`) with the global maximum over the time domain: `e_max = max(e_max, e.max())`.

An alternative error measure is to use a spatial norm at one time step only, e.g., the end time T ($n = N_t$):

$$E = ||e_i^n|| * *\ell^2 = \left(\Delta x \sum * * i = 0^{N_x} (e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n, \tag{2.23}$$

$$E = ||e_i^n|| * *\ell^\infty = \max * * 0 \leq i \leq N_x |e_i^n| \,. \tag{2.24}$$

The important point is that the error measure (E) for the simulation is represented by a single number.

2.10.3. Computing rates

Let E_i be the error measure in experiment (mesh) number i (not to be confused with the spatial index i) and let h_i be the corresponding discretization parameter (h). With the error model $E_i = Dh_i^r$, we can estimate r by comparing two consecutive experiments:

$$E_{i+1} = Dh_{i+1}^r,$$
$$E_i = Dh_i^r \,.$$

Dividing the two equations eliminates the (uninteresting) constant D. Thereafter, solving for r yields

$$r = \frac{\ln E_{i+1}/E_i}{\ln h_{i+1}/h_i} \,.$$

Since r depends on i, i.e., which simulations we compare, we add an index to r: r_i, where $i = 0, \ldots, m-2$, if we have m experiments: $(h_0, E_0), \ldots, (h_{m-1}, E_{m-1})$.

In our present discretization of the wave equation we expect $r = 2$, and hence the r_i values should converge to 2 as i increases.

2.11. Constructing an exact solution of the discrete equations

With a manufactured or known analytical solution, as outlined above, we can estimate convergence rates and see if they have the correct asymptotic behavior. Experience shows that this is a quite good verification technique in that many common bugs will destroy the convergence rates. A significantly better test though, would be to check that the numerical solution is exactly what it should be. This will in general require exact knowledge of the numerical error, which we do not normally have (although we in Section Section 2.61 establish such knowledge in simple cases). However, it is possible to look for solutions where we can show that the numerical error vanishes, i.e., the solution of the original continuous PDE problem is also a solution of the discrete equations. This property often arises if the exact solution of the PDE is a lower-order polynomial. (Truncation error analysis leads to error measures that involve derivatives of the exact solution. In the present problem, the truncation error involves 4th-order derivatives of u in space and time. Choosing u as a polynomial of degree three or less will therefore lead to vanishing error.)

We shall now illustrate the construction of an exact solution to both the PDE itself and the discrete equations. Our chosen manufactured solution is quadratic in space and linear in time. More specifically, we set

$$u_{\mathrm{e}}(x,t) = x(L-x)(1+\frac{1}{2}t), \tag{2.25}$$

which by insertion in the PDE leads to $f(x,t) = 2(1+t)c^2$. This u_e fulfills the boundary conditions $u = 0$ and demands $I(x) = x(L-x)$ and $V(x) = \frac{1}{2}x(L-x)$.

To realize that the chosen u_e is also an exact solution of the discrete equations, we first remind ourselves that $t_n = n\Delta t$ so that

$$[D_t D_t t^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - 2n^2 + (n-1)^2 = 2, \tag{2.26}$$

$$[D_t D_t t]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - 2n + (n-1))\Delta t}{\Delta t^2} = 0 \tag{2.27}$$

.Hence,

$$[D_t D_t u_\mathrm{e}]_i^n = x_i(L-x_i)[D_t D_t(1+\frac{1}{2}t)]^n = x_i(L-x_i)\frac{1}{2}[D_t D_t t]^n = 0\,.$$

Similarly, we get that

$$\begin{aligned}
[D_x D_x u_\mathrm{e}]_i^n &= (1+\frac{1}{2}t_n)[D_x D_x(xL - x^2)]_i \\
&= (1+\frac{1}{2}t_n)[LD_x D_x x - D_x D_x x^2]_i \\
&= -2(1+\frac{1}{2}t_n)
\end{aligned}$$

.Now, $f_i^n = 2(1+\frac{1}{2}t_n)c^2$, which results in

$$[D_t D_t u_\mathrm{e} - c^2 D_x D_x u_\mathrm{e} - f]_i^n = 0 + c^2 2(1+\frac{1}{2}t_n) + 2(1+\frac{1}{2}t_n)c^2 = 0\,.$$

Moreover, $u_\mathrm{e}(x_i,0) = I(x_i)$, $\partial u_\mathrm{e}/\partial t = V(x_i)$ at $t = 0$, and $u_\mathrm{e}(x_0,t) = u_\mathrm{e}(x_{N_x},0) = 0$. Also the modified scheme for the first time step is fulfilled by $u_\mathrm{e}(x_i,t_n)$.

Therefore, the exact solution $u_\mathrm{e}(x,t) = x(L-x)(1+t/2)$ of the PDE problem is also an exact solution of the discrete problem. This means that we know beforehand what numbers the numerical algorithm should produce. We can use this fact to check that the computed u_i^n values from an implementation equals $u_\mathrm{e}(x_i,t_n)$, within machine precision. This result is valid *regardless of the mesh spacings* Δx and Δt! Nevertheless, there might be stability restrictions on Δx and Δt, so the test can only be run for a mesh that is compatible with the stability criterion (which in the present case is $C \leq 1$, to be derived later).

i A product of quadratic or linear expressions in the various

independent variables, as shown above, will often fulfill both the PDE problem and the discrete equations, and can therefore be very useful solutions for verifying implementations.
However, for 1D wave equations of the type $u_{tt} = c^2 u_{xx}$ we shall see that there is always another much more powerful way of generating exact solutions (which consists in just setting $C = 1$ (!), as shown in Section Section 2.61).

2.12. Solving the Wave Equation with Devito

In this section we demonstrate how to solve the wave equation using the Devito domain-specific language (DSL). Devito allows us to write the PDE symbolically and generates optimized C code automatically.

2.12.1. From Mathematics to Devito Code

Recall the 1D wave equation from Section 2.1:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L),\ t \in (0, T] \tag{2.28}$$

with initial conditions $u(x, 0) = I(x)$ and $\partial u/\partial t|_{t=0} = V(x)$, and boundary conditions $u(0, t) = u(L, t) = 0$.

In Devito, we express this PDE directly using symbolic derivatives. The key abstractions are:

- **Grid**: Defines the discrete domain
- **TimeFunction**: A field that varies in both space and time
- **Eq**: An equation relating symbolic expressions
- **Operator**: Compiles equations to optimized C code

2.12.2. The Devito Grid

A Devito `Grid` defines the discrete spatial domain:

```
from devito import Grid

L = 1.0      # Domain length
Nx = 100     # Number of grid intervals

grid = Grid(shape=(Nx + 1,), extent=(L,))
```

The `shape` is the number of grid points (including boundaries), and `extent` is the physical size of the domain.

2.12.3. TimeFunction for the Wave Field

The solution $u(x, t)$ is represented by a `TimeFunction`:

```
from devito import TimeFunction

u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
```

The key parameters are:

- `time_order=2`: We need u^{n+1}, u^n, u^{n-1} for the wave equation
- `space_order=2`: Central difference with second-order accuracy

2.12.4. Symbolic Derivatives

Devito provides symbolic access to derivatives through attribute notation:

Derivative	Devito syntax	Mathematical meaning
First time	`u.dt`	$\partial u/\partial t$
Second time	`u.dt2`	$\partial^2 u/\partial t^2$
First space	`u.dx`	$\partial u/\partial x$
Second space	`u.dx2`	$\partial^2 u/\partial x^2$

2.12.5. Formulating the PDE

We express the wave equation as a residual that should be zero:

```
from devito import Eq, solve, Constant

c_sq = Constant(name='c_sq')  # Wave speed squared

# PDE: u_tt - c^2 * u_xx = 0
pde = u.dt2 - c_sq * u.dx2
```

The `solve` function isolates the unknown u^{n+1}:

```
stencil = Eq(u.forward, solve(pde, u.forward))
```

Here `u.forward` represents u^{n+1}, the solution at the next time level.

2.12.6. Boundary Conditions

For Dirichlet conditions $u(0,t) = u(L,t) = 0$, we add explicit equations:

```
t_dim = grid.stepping_dim  # Time index dimension

bc_left = Eq(u[t_dim + 1, 0], 0)
bc_right = Eq(u[t_dim + 1, Nx], 0)
```

2.12.7. Creating and Running the Operator

The `Operator` compiles all equations into optimized code:

```
from devito import Operator

op = Operator([stencil, bc_left, bc_right])
```

To execute a time step, we call:

```
op.apply(time_m=1, time_M=1, dt=dt, c_sq=c**2)
```

2.12.8. Complete Solver Implementation

The module `src.wave` provides a complete solver that handles:

- Initial conditions with velocity ($u_t(x,0) = V(x)$)
- CFL stability checking
- Optional history storage

```
from src.wave import solve_wave_1d
import numpy as np

# Define initial condition: plucked string
def I(x):
    return np.sin(np.pi * x)

# Solve
result = solve_wave_1d(
    L=1.0,           # Domain length
    c=1.0,           # Wave speed
    Nx=100,          # Grid points
    T=1.0,           # Final time
    C=0.9,           # Courant number
    I=I,             # Initial displacement
)

# Access results
u_final = result.u   # Solution at final time
x = result.x         # Spatial grid
```

2.12.9. The Courant Number and Stability

The Courant number $C = c\Delta t/\Delta x$ determines stability. For the explicit wave equation solver, we require $C \leq 1$.

When $C = 1$ (the magic value), the numerical solution is **exact** for waves traveling in either direction. This is because the domain of dependence of the numerical scheme exactly matches the physical domain of dependence.

2.12.10. Handling Initial Velocity

The first time step requires special treatment when $V(x) \neq 0$. Using the Taylor expansion:

$$u^1 = u^0 + \Delta t \cdot V(x) + \frac{1}{2}\Delta t^2 c^2 u^0_{xx}$$

The solver implements this as:

```
u0 = I(x_coords)
v0 = V(x_coords)
u_xx_0 = np.zeros_like(u0)
u_xx_0[1:-1] = (u0[2:] - 2*u0[1:-1] + u0[:-2]) / dx**2

u1 = u0 + dt * v0 + 0.5 * dt**2 * c**2 * u_xx_0
```

2.12.11. Verification: Standing Wave Solution

The standing wave with $I(x) = A\sin(\pi x/L)$ and $V = 0$ has the exact solution:

$$u(x,t) = A\sin\left(\frac{\pi x}{L}\right)\cos\left(\frac{\pi ct}{L}\right)$$

We can verify our implementation converges at the expected rate:

```
from src.wave import convergence_test_wave_1d

grid_sizes, errors, rate = convergence_test_wave_1d(
    grid_sizes=[20, 40, 80, 160],
    T=0.5,
    C=0.9,
)

print(f"Observed convergence rate: {rate:.2f}")  # Should be ~2.0
```

2.12.12. Visualization

For time-dependent problems, animation is essential. With the history saved, we can create animations:

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

result = solve_wave_1d(
    L=1.0, c=1.0, Nx=100, T=2.0, C=0.9,
    save_history=True,
```

```
)

fig, ax = plt.subplots()
line, = ax.plot(result.x, result.u_history[0])
ax.set_ylim(-1.2, 1.2)
ax.set_xlabel('x')
ax.set_ylabel('u')

def update(frame):
    line.set_ydata(result.u_history[frame])
    ax.set_title(f't = {result.t_history[frame]:.3f}')
    return line,

anim = FuncAnimation(fig, update, frames=len(result.t_history),
                     interval=50, blit=True)
```

2.12.13. Summary: Devito vs. NumPy

The key advantages of using Devito for wave equations:

1. **Symbolic PDEs**: Write the math, not the stencils
2. **Automatic optimization**: Cache-efficient loops generated automatically
3. **Parallelization**: OpenMP/MPI/GPU support without code changes
4. **Dimension-agnostic**: Same code pattern works for 1D, 2D, 3D

The explicit time-stepping loop remains visible to the user for educational purposes, but Devito handles the spatial discretization and can generate highly optimized code for the inner loop.

2.13. Source Terms and Variable Coefficients

Real-world wave propagation often involves source terms and spatially varying wave speeds. This section extends the Devito wave solver to handle these features.

2.13.1. Adding a Source Term

The wave equation with a source term is:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} + f(x,t) \tag{2.29}$$

In seismic applications, $f(x,t)$ often represents an impulsive source at a specific location.

2.13.2. Source Wavelets

The `src.wave` module provides common source wavelets used in seismic modeling:

```
from src.wave import ricker_wavelet, gaussian_pulse
import numpy as np

t = np.linspace(0, 0.5, 501)  # Time array

# Ricker wavelet with 25 Hz peak frequency
src_ricker = ricker_wavelet(t, f0=25.0)

# Gaussian pulse
src_gauss = gaussian_pulse(t, t0=0.1, sigma=0.02)
```

2.13.3. The Ricker Wavelet

The Ricker wavelet (Mexican hat wavelet) is the negative normalized second derivative of a Gaussian:

$$r(t) = A\left(1 - 2\pi^2 f_0^2 (t - t_0)^2\right) e^{-\pi^2 f_0^2 (t-t_0)^2}$$

where f_0 is the peak frequency and t_0 is the time shift.

```
import matplotlib.pyplot as plt
from src.wave import ricker_wavelet, get_source_spectrum

t = np.linspace(0, 0.3, 301)
dt = t[1] - t[0]

# Create wavelet
wavelet = ricker_wavelet(t, f0=25.0)

# Compute spectrum
freq, amp = get_source_spectrum(wavelet, dt)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.plot(t, wavelet)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Ricker Wavelet (f0 = 25 Hz)')

ax2.plot(freq[:100], amp[:100])
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Amplitude')
ax2.set_title('Frequency Spectrum')
ax2.axvline(25, color='r', linestyle='--', label='Peak freq')
ax2.legend()
```

2.13.4. Point Sources in Devito

For seismic modeling, sources are often located at specific points in space. Devito provides `SparseTimeFunction` for this:

```
from devito import SparseTimeFunction

# Point source at x = 0.5
src = SparseTimeFunction(
    name='src', grid=grid,
    npoint=1, nt=Nt,
    coordinates=np.array([[0.5]])
)

# Set source wavelet
src.data[:] = ricker_wavelet(t, f0=25.0).reshape(-1, 1)

# Inject into the wave equation
src_term = src.inject(field=u.forward, expr=src * dt**2)
```

2.13.5. Variable Wave Speed

In heterogeneous media, the wave speed varies in space:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (c^2(x) \nabla u)$$

In 1D, this simplifies to:

$$u_{tt} = (c^2 u_x)_x = c^2 u_{xx} + 2cc_x u_x$$

For smoothly varying $c(x)$, we can approximate this as:

$$u_{tt} \approx c^2(x) u_{xx}$$

2.13.6. Implementing Variable Velocity in Devito

We use a `Function` (not `TimeFunction`) for the velocity field:

```
from devito import Function

# Velocity field
c = Function(name='c', grid=grid)

# Set velocity values (e.g., layer model)
x_coords = np.linspace(0, L, Nx + 1)
c.data[:] = np.where(x_coords < 0.5, 1.0, 2.0)  # Two layers
```

The PDE uses this spatially varying velocity:

```
pde = u.dt2 - c**2 * u.dx2
stencil = Eq(u.forward, solve(pde, u.forward))
```

2.13.7. CFL Condition with Variable Velocity

When velocity varies, the CFL condition must use the maximum velocity:

$$\Delta t \leq \frac{\Delta x}{c_{\max}}$$

```
c_max = np.max(c.data)
dt_stable = dx / c_max
```

2.13.8. Example: Wave Propagation in Layered Medium

Consider a domain with two layers of different wave speeds:

```
from devito import Grid, TimeFunction, Function, Eq, solve, Operator

# Setup
L = 2.0
Nx = 200
grid = Grid(shape=(Nx + 1,), extent=(L,))

# Velocity: slow layer (c=1) then fast layer (c=2)
c = Function(name='c', grid=grid)
x_coords = np.linspace(0, L, Nx + 1)
c.data[:] = np.where(x_coords < 1.0, 1.0, 2.0)

# Wave field
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

# Initial condition: Gaussian pulse in slow region
sigma = 0.1
x0 = 0.3
u.data[0, :] = np.exp(-((x_coords - x0) / sigma)**2)
u.data[1, :] = u.data[0, :]

# Wave equation with variable velocity
pde = u.dt2 - c**2 * u.dx2
stencil = Eq(u.forward, solve(pde, u.forward))

# Boundary conditions
bc_left = Eq(u[grid.stepping_dim + 1, 0], 0)
```

```
bc_right = Eq(u[grid.stepping_dim + 1, Nx], 0)

# Operator
op = Operator([stencil, bc_left, bc_right])
```

When the pulse reaches the interface at $x = 1$:

1. Part of the wave is **reflected** back into the slow medium
2. Part of the wave is **transmitted** into the fast medium
3. The transmitted wave travels faster and has a different wavelength

2.13.9. Reflection and Transmission Coefficients

At an interface between media with velocities c_1 and c_2, the reflection coefficient is:

$$R = \frac{c_2 - c_1}{c_2 + c_1}$$

And the transmission coefficient is:

$$T = \frac{2c_2}{c_2 + c_1}$$

For our example with $c_1 = 1$ and $c_2 = 2$:

- $R = (2 - 1)/(2 + 1) = 1/3$
- $T = 2 \cdot 2/(2 + 1) = 4/3$

The transmitted wave has larger amplitude but carries the same energy (accounting for the velocity change).

2.13.10. Absorbing Boundary Conditions

For open-domain problems, we want waves to leave without reflecting from artificial boundaries. A simple approach is a **sponge layer** that gradually damps the solution near boundaries:

```
from devito import Function

# Damping coefficient (zero in interior, increasing at boundaries)
damp = Function(name='damp', grid=grid)

pad = 20  # Width of sponge layer
damp_profile = np.zeros(Nx + 1)
damp_profile[:pad] = 0.1 * (1 - np.linspace(0, 1, pad))
damp_profile[-pad:] = 0.1 * np.linspace(0, 1, pad)
damp.data[:] = damp_profile

# Modified PDE with damping term
pde_damped = u.dt2 + damp * u.dt - c**2 * u.dx2
```

The damping term γu_t removes energy from the wave as it enters the sponge layer.

2.13.11. Summary

Devito makes it straightforward to extend the basic wave solver to handle:

- **Source terms**: Point sources and wavelets for seismic modeling
- **Variable velocity**: Layered or smooth velocity variations
- **Absorbing boundaries**: Sponge layers to reduce reflections

The key is that Devito handles the discretization automatically once we express the PDE symbolically. This allows us to focus on the physics rather than implementation details.

2.14. Implementation

This section presents the complete computational algorithm, its implementation in Python code, animation of the solution, and verification of the implementation.

A real implementation of the basic computational algorithm from Sections Section 2.6 and Section 2.7 can be encapsulated in a function, taking all the input data for the problem as arguments. The physical input data consists of c, $I(x)$, $V(x)$, $f(x,t)$, L, and T. The numerical input is the mesh parameters Δt and Δx.

Instead of specifying Δt *and* Δx, we can specify one of them and the Courant number C instead, since having explicit control of the Courant number is convenient when investigating the numerical method. Many find it natural to prescribe the resolution of the spatial grid and set N_x. The solver function can then compute $\Delta t = CL/(cN_x)$. However, for comparing $u(x,t)$ curves (as functions of x) for various Courant numbers it is more convenient to keep Δt fixed for all C and let Δx vary according to $\Delta x = c\Delta t/C$. With Δt fixed, all frames correspond to the same time t, and this simplifies animations that compare simulations with different mesh resolutions. Plotting functions of x with different spatial resolution is trivial, so it is easier to let Δx vary in the simulations than Δt.

2.15. Callback function for user-specific actions

The solution at all spatial points at a new time level is stored in an array `u` of length $N_x + 1$. We need to decide what to do with this solution, e.g., visualize the curve, analyze the values, or write the array to file for later use. The decision about what to do is left to the user in the form of a user-supplied function

```
user_action(u, x, t, n)
```

where `u` is the solution at the spatial points `x` at time `t[n]`. The `user_action` function is called from the solver at each time level `n`.

If the user wants to plot the solution or store the solution at a time point, she needs to write such a function and take appropriate actions inside it. We will show examples on many such `user_action` functions.

Since the solver function makes calls back to the user's code via such a function, this type of function is called a *callback function*. When writing general software, like our solver function, which also needs to carry out special problem- or solution-dependent actions (like visualization), it is a common technique to leave those actions to user-supplied callback functions.

The callback function can be used to terminate the solution process if the user returns `True`. For example,

```
def my_user_action_function(u, x, t, n):
    return np.abs(u).max() > 10
```

is a callback function that will terminate the solver function (given below) of the amplitude of the waves exceed 10, which is here considered as a numerical instability.

2.16. The solver function

A first attempt at a solver function is listed below.

```
import numpy as np


def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T / dt))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = dt * c / float(C)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    C2 = C**2  # Help variable in the scheme
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    if f is None or f == 0:
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    u = np.zeros(Nx + 1)  # Solution array at new time level
    u_n = np.zeros(Nx + 1)  # Solution at 1 time level back
    u_nm1 = np.zeros(Nx + 1)  # Solution at 2 time levels back

    import time
```

```
    t0 = time.perf_counter()  # Measure CPU time

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    n = 0
    for i in range(1, Nx):
        u[i] = (
            u_n[i]
            + dt * V(x[i])
            + 0.5 * C2 * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
            + 0.5 * dt**2 * f(x[i], t[n])
        )
    u[0] = 0
    u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    u_nm1[:] = u_n
    u_n[:] = u

    for n in range(1, Nt):
        for i in range(1, Nx):
            u[i] = (
                -u_nm1[i]
                + 2 * u_n[i]
                + C2 * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
                + dt**2 * f(x[i], t[n])
            )

        u[0] = 0
        u[Nx] = 0
        if user_action is not None:
            if user_action(u, x, t, n + 1):
                break

        u_nm1[:] = u_n
        u_n[:] = u

    cpu_time = time.perf_counter() - t0
    return u, x, t, cpu_time
```

A couple of remarks about the above code is perhaps necessary:

- Although we give `dt` and compute `dx` via `C` and `c`, the resulting `t` and `x` meshes do not necessarily correspond exactly to these values because of rounding errors. To explicitly ensure that `dx` and `dt` correspond to the cell sizes in `x` and `t`, we recompute the values.
- According to the particular choice made in Section Section 2.15, a true value returned from `user_action` should terminate the simulation. This is here implemented by a `break` statement inside the for loop in the solver.

2.17. Verification: exact quadratic solution

We use the test problem derived in Section Section 2.8 for verification. Below is a unit test based on this test problem and realized as a proper *test function* compatible with the unit test frameworks nose or pytest.

```
def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""

    def u_exact(x, t):
        return x * (L - x) * (1 + 0.5 * t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5 * u_exact(x, 0)

    def f(x, t):
        return 2 * (1 + 0.5 * t) * c**2

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 6  # Very coarse mesh for this exact test
    dt = C * (L / Nx) / c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(u - u_e).max()
        tol = 1e-13
        assert diff < tol

    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error)
```

When this function resides in the file `wave1D_u0.py`, one can run pytest to check that all test functions with names `test_*()` in this file work:

```
Terminal> py.test -s -v wave1D_u0.py
```

2.18. Verification: convergence rates

A more general method, but not so reliable as a verification method, is to compute the convergence rates and see if they coincide with theoretical estimates. Here we expect a rate of 2 according to the various results in Section Section 2.61. A general function for computing convergence rates can be written like this:

```
"""
1D wave equation with u=0 at the boundary.
Simplest possible implementation.

The key function is::

  u, x, t, cpu = (I, V, f, c, L, dt, C, T, user_action)

which solves the wave equation u_tt = c**2*u_xx on (0,L) with u=0
on x=0,L, for t in (0,T].  Initial conditions: u=I(x), u_t=V(x).

T is the stop time for the simulation.
dt is the desired time step.
C is the Courant number (=c*dt/dx), which specifies dx.
f(x,t) is a function for the source term (can be 0 or None).
I and V are functions of x.

user_action is a function of (u, x, t, n) where the calling
code can add visualization, error computations, etc.
"""

import numpy as np


def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T / dt))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = dt * c / float(C)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    C2 = C**2  # Help variable in the scheme
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    if f is None or f == 0:
        f = lambda x, t: 0
```

```
    if V is None or V == 0:
        V = lambda x: 0

    u = np.zeros(Nx + 1)  # Solution array at new time level
    u_n = np.zeros(Nx + 1)  # Solution at 1 time level back
    u_nm1 = np.zeros(Nx + 1)  # Solution at 2 time levels back

    import time

    t0 = time.perf_counter()  # Measure CPU time

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    n = 0
    for i in range(1, Nx):
        u[i] = (
            u_n[i]
            + dt * V(x[i])
            + 0.5 * C2 * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
            + 0.5 * dt**2 * f(x[i], t[n])
        )
    u[0] = 0
    u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    u_nm1[:] = u_n
    u_n[:] = u

    for n in range(1, Nt):
        for i in range(1, Nx):
            u[i] = (
                -u_nm1[i]
                + 2 * u_n[i]
                + C2 * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
                + dt**2 * f(x[i], t[n])
            )

        u[0] = 0
        u[Nx] = 0
        if user_action is not None:
            if user_action(u, x, t, n + 1):
```

```
                break

            u_nm1[:] = u_n
            u_n[:] = u

        cpu_time = time.perf_counter() - t0
        return u, x, t, cpu_time


def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""

    def u_exact(x, t):
        return x * (L - x) * (1 + 0.5 * t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5 * u_exact(x, 0)

    def f(x, t):
        return 2 * (1 + 0.5 * t) * c**2

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 6  # Very coarse mesh for this exact test
    dt = C * (L / Nx) / c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(u - u_e).max()
        tol = 1e-13
        assert diff < tol

    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error)


def test_constant():
    """Check that u(x,t)=Q=0 is exactly reproduced."""
    u_const = 0  # Require 0 because of the boundary conditions
    C = 0.75
    dt = C  # Very coarse mesh
    u, x, t, cpu = solver(I=lambda x: 0, V=0, f=0, c=1.5, L=2.5, dt=dt, C=C, T=18)
    tol = 1e-14
```

```
    assert np.abs(u - u_const).max() < tol


def viz(
    I,
    V,
    f,
    c,
    L,
    dt,
    C,
    T,  # PDE parameters
    umin,
    umax,  # Interval for u in plots
    animate=True,  # Simulation with animation?
    solver_function=solver,  # Function with numerical algorithm
):
    """Run solver and visualize u at each time level."""
    import glob
    import os
    import time

    import matplotlib.pyplot as plt

    class PlotMatplotlib:
        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if n == 0:
                plt.ion()
                self.lines = plt.plot(x, u, "r-")
                plt.xlabel("x")
                plt.ylabel("u")
                plt.axis([0, L, umin, umax])
                plt.legend(["t=%f" % t[n]], loc="lower left")
            else:
                self.lines[0].set_ydata(u)
                plt.legend(["t=%f" % t[n]], loc="lower left")
                plt.draw()
            time.sleep(2) if t[n] == 0 else time.sleep(0.2)
            plt.savefig("tmp_%04d.png" % n)  # for movie making

    plot_u = PlotMatplotlib()

    for filename in glob.glob("tmp_*.png"):
        os.remove(filename)

    user_action = plot_u if animate else None
```

```
    u, x, t, cpu = solver_function(I, V, f, c, L, dt, C, T, user_action)

    fps = 4  # frames per second
    codec2ext = dict(
        flv="flv", libx264="mp4", libvpx="webm", libtheora="ogg"
    )  # video formats
    filespec = "tmp_%04d.png"
    movie_program = "ffmpeg"
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = (
            "%(movie_program)s -r %(fps)d -i %(filespec)s "
            "-vcodec %(codec)s movie.%(ext)s" % vars()
        )
        os.system(cmd)

    return cpu


def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8 * L
    a = 0.005
    freq = 440
    wavelength = 2 * L
    c = freq * wavelength
    omega = 2 * np.pi * freq
    num_periods = 1
    T = 2 * np.pi / omega * num_periods
    dt = L / 50.0 / c

    def I(x):
        return a * x / x0 if x < x0 else a / (L - x0) * (L - x)

    umin = -1.2 * a
    umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)


def convergence_rates(
    u_exact,  # Python function for exact solution
    I,
    V,
    f,
    c,
    L,  # physical parameters
```

```
    dt0,
    num_meshes,
    C,
    T,
):  # numerical parameters
    """
    Half the time step and estimate convergence rates for
    for num_meshes simulations.
    """
    global error
    error = 0  # error computed in the user action function

    def compute_error(u, x, t, n):
        global error  # must be global to be altered here
        if n == 0:
            error = 0
        else:
            error = max(error, np.abs(u - u_exact(x, t[n])).max())

    E = []
    h = []  # dt, solver adjusts dx such that C=dt*c/dx
    dt = dt0
    for i in range(num_meshes):
        solver(I, V, f, c, L, dt, C, T, user_action=compute_error)
        E.append(error)
        h.append(dt)
        dt /= 2  # halve the time step for next simulation
    print("E:", E)
    print("h:", h)
    r = [np.log(E[i] / E[i - 1]) / np.log(h[i] / h[i - 1]) for i in range(1, num_meshes)]
    return r
```

Using the analytical solution from Section Section 2.9, we can call `convergence_rates` to see if we get a convergence rate that approaches 2 and use the final estimate of the rate in an `assert` statement such that this function becomes a proper test function:

```
def test_convrate_sincos():
    n = m = 2
    L = 1.0
    u_exact = lambda x, t: np.cos(m * np.pi / L * t) * np.sin(m * np.pi / L * x)

    r = convergence_rates(
        u_exact=u_exact,
        I=lambda x: u_exact(x, 0),
        V=lambda x: 0,
        f=0,
        c=1,
```

```
        L=L,
        dt0=0.1,
        num_meshes=6,
        C=0.9,
        T=1,
    )
    print("rates sin(x)*cos(t) solution:", [round(r_, 2) for r_ in r])
    assert abs(r[-1] - 2) < 0.002
```

Doing `py.test -s -v wave1D_u0.py` will run also this test function and show the rates 2.05, 1.98, 2.00, 2.00, and 2.00 (to two decimals).

2.19. Visualization: animating the solution

Now that we have verified the implementation it is time to do a real computation where we also display evolution of the waves on the screen. Since the `solver` function knows nothing about what type of visualizations we may want, it calls the callback function `user_action(u, x, t, n)`. We must therefore write this function and find the proper statements for plotting the solution.

2.19.1. Function for administering the simulation

The following `viz` function

1. defines a `user_action` callback function for plotting the solution at each time level,
2. calls the `solver` function, and
3. combines all the plots (in files) to video in different formats.

```
def viz(
    I,
    V,
    f,
    c,
    L,
    dt,
    C,
    T,  # PDE parameters
    umin,
    umax,  # Interval for u in plots
    animate=True,  # Simulation with animation?
    solver_function=solver,  # Function with numerical algorithm
):
    """Run solver and visualize u at each time level."""
    import glob
    import os
    import time
```

```
    import matplotlib.pyplot as plt

    class PlotMatplotlib:
        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if n == 0:
                plt.ion()
                self.lines = plt.plot(x, u, "r-")
                plt.xlabel("x")
                plt.ylabel("u")
                plt.axis([0, L, umin, umax])
                plt.legend(["t=%f" % t[n]], loc="lower left")
            else:
                self.lines[0].set_ydata(u)
                plt.legend(["t=%f" % t[n]], loc="lower left")
                plt.draw()
            time.sleep(2) if t[n] == 0 else time.sleep(0.2)
            plt.savefig("tmp_%04d.png" % n)  # for movie making

    plot_u = PlotMatplotlib()

    for filename in glob.glob("tmp_*.png"):
        os.remove(filename)

    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(I, V, f, c, L, dt, C, T, user_action)

    fps = 4  # frames per second
    codec2ext = dict(
        flv="flv", libx264="mp4", libvpx="webm", libtheora="ogg"
    )  # video formats
    filespec = "tmp_%04d.png"
    movie_program = "ffmpeg"
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = (
            "%(movie_program)s -r %(fps)d -i %(filespec)s "
            "-vcodec %(codec)s movie.%(ext)s" % vars()
        )
        os.system(cmd)

    return cpu
```

2.19.2. Dissection of the code

The `viz` function uses Matplotlib for visualizing the solution. The `user_action` function is realized as a class and needs statements that differ from those for making static plots.

With Matplotlib, one has to make the first plot the standard way, and then update the y data in the plot at every time level. The update requires active use of the returned value from `plt.plot` in the first plot. This value would need to be stored in a local variable if we were to use a closure for the `user_action` function when doing the animation with Matplotlib. It is much easier to store the variable as a class attribute `self.lines`. Since the class is essentially a function, we implement the function as the special method `__call__` such that the instance `plot_u(u, x, t, n)` can be called as a standard callback function from `solver`.

To achieve a smooth animation, we want to save each frame in the animation to file. We then need a filename where the frame number is padded with zeros, here `tmp_0000.png`, `tmp_0001.png`, and so on. The proper printf construction is then `tmp_%04d.png`.

2.19.3. Making movie files

From the `frame_*.png` files containing the frames in the animation we can make video files using the `ffmpeg` (or `avconv`) program to produce videos in modern formats: Flash, MP4, Webm, and Ogg.

The `viz` function creates an `ffmpeg` or `avconv` command with the proper arguments for each of the formats Flash, MP4, WebM, and Ogg. The task is greatly simplified by having a `codec2ext` dictionary for mapping video codec names to filename extensions. In practice, only two formats are needed to ensure that all browsers can successfully play the video: MP4 and WebM.

Some animations having a large number of plot files may not be properly combined into a video using `ffmpeg` or `avconv`. One alternative is to play the PNG files directly in an image viewer or create an animated GIF using ImageMagick's `convert` command:

```
Terminal> convert -delay 25 tmp_*.png animation.gif
```

The `-delay` option specifies the delay between frames in hundredths of a second.

2.19.4. Skipping frames for animation speed

Sometimes the time step is small and T is large, leading to an inconveniently large number of plot files and a slow animation on the screen. The solution to such a problem is to decide on a total number of frames in the animation, `num_frames`, and plot the solution only for every `skip_frame` frames. For example, setting `skip_frame=5` leads to plots of every 5 frames. The default value `skip_frame=1` plots every frame. The total number of time levels (i.e., maximum possible number of frames) is the length of `t`, `t.size` (or `len(t)`), so if we want `num_frames` frames in the animation, we need to plot every `t.size/num_frames` frames:

```
skip_frame = int(t.size/float(num_frames))
if n % skip_frame == 0 or n == t.size-1:
    st.plot(x, u, 'r-', ...)
```

The initial condition (`n=0`) is included by `n % skip_frame == 0`, as well as every `skip_frame`-th frame. As `n % skip_frame == 0` will very seldom be true for the very final frame, we must also check if `n == t.size-1` to get the final frame included.

A simple choice of numbers may illustrate the formulas: say we have 801 frames in total (`t.size`) and we allow only 60 frames to be plotted. As `n` then runs from 801 to 0, we need to plot every 801/60 frame, which with integer division yields 13 as `skip_frame`. Using the mod function, `n % skip_frame`, this operation is zero every time `n` can be divided by 13 without a remainder. That is, the `if` test is true when `n` equals $0, 13, 26, 39, ..., 780, 801$. The associated code is included in the `plot_u` function, inside the `viz` function, in the file wave1D_u0.py.

2.20. Running a case

The first demo of our 1D wave equation solver concerns vibrations of a string that is initially deformed to a triangular shape, like when picking a guitar string:

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{2.30}$$

We choose $L = 75$ cm, $x_0 = 0.8L$, $a = 5$ mm, and a time frequency $\nu = 440$ Hz. The relation between the wave speed c and ν is $c = \nu\lambda$, where λ is the wavelength, taken as $2L$ because the longest wave on the string forms half a wavelength. There is no external force, so $f = 0$ (meaning we can neglect gravity), and the string is at rest initially, implying $V = 0$.

Regarding numerical parameters, we need to specify a Δt. Sometimes it is more natural to think of a spatial resolution instead of a time step. A natural semi-coarse spatial resolution in the present problem is $N_x = 50$. We can then choose the associated Δt (as required by the `viz` and `solver` functions) as the stability limit: $\Delta t = L/(N_x c)$. This is the Δt to be specified, but notice that if $C < 1$, the actual Δx computed in `solver` gets larger than L/N_x: $\Delta x = c\Delta t/C = L/(N_x C)$. (The reason is that we fix Δt and adjust Δx, so if C gets smaller, the code implements this effect in terms of a larger Δx.)

A function for setting the physical and numerical parameters and calling `viz` in this application goes as follows:

```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8 * L
    a = 0.005
    freq = 440
    wavelength = 2 * L
    c = freq * wavelength
```

```
    omega = 2 * np.pi * freq
    num_periods = 1
    T = 2 * np.pi / omega * num_periods
    dt = L / 50.0 / c

    def I(x):
        return a * x / x0 if x < x0 else a / (L - x0) * (L - x)

    umin = -1.2 * a
    umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)
```

The associated program has the name `wave1D_u0.py`. Run the program and watch the movie of the vibrating string. The string should ideally consist of straight segments, but these are somewhat wavy due to numerical approximation. Run the case with the `wave1D_u0.py` code and $C = 1$ to see the exact solution.

2.21. Working with a scaled PDE model

Depending on the model, it may be a substantial job to establish consistent and relevant physical parameter values for a case. The guitar string example illustrates the point. However, by *scaling* the mathematical problem we can often reduce the need to estimate physical parameters dramatically. The scaling technique consists of introducing new independent and dependent variables, with the aim that the absolute values of these lie in $[0, 1]$. We introduce the dimensionless variables (details are found in Section 3.1.1 in (Langtangen and Pedersen 2016))

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{c}{L}t, \quad \bar{u} = \frac{u}{a}.$$

Here, L is a typical length scale, e.g., the length of the domain, and a is a typical size of u, e.g., determined from the initial condition: $a = \max_x |I(x)|$.

We get by the chain rule that

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial \bar{t}}(a\bar{u})\frac{d\bar{t}}{dt} = \frac{ac}{L}\frac{\partial \bar{u}}{\partial \bar{t}}.$$

Similarly,

$$\frac{\partial u}{\partial x} = \frac{a}{L}\frac{\partial \bar{u}}{\partial \bar{x}}.$$

Inserting the dimensionless variables in the PDE gives, in case $f = 0$,

$$\frac{a^2c^2}{L^2}\frac{\partial^2 \bar{u}}{\partial \bar{t}^2} = \frac{a^2c^2}{L^2}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2}.$$

Dropping the bars, we arrive at the scaled PDE

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \quad x \in (0,1),\ t \in (0, cT/L),$$

which has no parameter c^2 anymore. The initial conditions are scaled as

$$a\bar{u}(\bar{x},0) = I(L\bar{x})$$

and

$$\frac{a}{L/c}\frac{\partial \bar{u}}{\partial \bar{t}}(\bar{x},0) = V(L\bar{x}),$$

resulting in

$$\bar{u}(\bar{x},0) = \frac{I(L\bar{x})}{\max_x |I(x)|}, \quad \frac{\partial \bar{u}}{\partial \bar{t}}(\bar{x},0) = \frac{L}{ac}V(L\bar{x})\,.$$

In the common case $V = 0$ we see that there are no physical parameters to be estimated in the PDE model!

If we have a program implemented for the physical wave equation with dimensions, we can obtain the dimensionless, scaled version by setting $c = 1$. The initial condition of a guitar string, given in (2.30), gets its scaled form by choosing $a = 1$, $L = 1$, and $x_0 \in [0, 1]$. This means that we only need to decide on the x_0 value as a fraction of unity, because the scaled problem corresponds to setting all other parameters to unity. In the code we can just set `a=c=L=1`, `x0=0.8`, and there is no need to calculate with wavelengths and frequencies to estimate c!

The only non-trivial parameter to estimate in the scaled problem is the final end time of the simulation, or more precisely, how it relates to periods in periodic solutions in time, since we often want to express the end time as a certain number of periods. The period in the dimensionless problem is 2, so the end time can be set to the desired number of periods times 2.

Why the dimensionless period is 2 can be explained by the following reasoning. Suppose that u behaves as $\cos(\omega t)$ in time in the original problem with dimensions. The corresponding period is then $P = 2\pi/\omega$, but we need to estimate ω. A typical solution of the wave equation is $u(x,t) = A\cos(kx)\cos(\omega t)$, where A is an amplitude and k is related to the wave length λ in space: $\lambda = 2\pi/k$. Both λ and A will be given by the initial condition $I(x)$. Inserting this $u(x,t)$ in the PDE yields $-\omega^2 = -c^2k^2$, i.e., $\omega = kc$. The period is therefore $P = 2\pi/(kc)$. If the boundary conditions are $u(0,t) = u(L,t)$, we need to have $kL = n\pi$ for integer n. The period becomes $P = 2L/nc$. The longest period is $P = 2L/c$. The dimensionless period $\tilde{P}$ is obtained by dividing P by the time scale L/c, which results in $\tilde{P} = 2$. Shorter waves in the initial condition will have a dimensionless shorter period $\tilde{P} = 2/n$ ($n > 1$).

2.22. Vectorized computations

The computational algorithm for solving the wave equation visits one mesh point at a time and evaluates a formula for the new value u_i^{n+1} at that point. Technically, this is implemented by a loop over array elements in a program. Such loops may run slowly in Python (and similar interpreted languages such as R and MATLAB). One technique for speeding up loops is to perform operations on entire arrays instead of working with one element at a time. This is referred to as *vectorization*, *vector computing*, or *array computing*. Operations on whole arrays are possible if the computations involving each element is independent of each other and therefore can, at least in principle, be performed simultaneously. That is, vectorization not only speeds up the code on serial computers, but also makes it easy to exploit parallel computing. Actually, there are Python tools like Numba that can automatically turn vectorized code into parallel code.

2.23. Operations on slices of arrays

Efficient computing with `numpy` arrays demands that we avoid loops and compute with entire arrays at once (or at least large portions of them). Consider this calculation of differences $d_i = u_{i+1} - u_i$:

```
n = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

All the differences here are independent of each other. The computation of `d` can therefore alternatively be done by subtracting the array $(u_0, u_1, \ldots, u_{n-1})$ from the array where the elements are shifted one index upwards: $(u_1, u_2, \ldots, u_n)$, see Figure Figure 2.3. The former subset of the array can be expressed by `u[0:n-1]`, `u[0:-1]`, or just `u[:-1]`, meaning from index 0 up to, but not including, the last element (`-1`). The latter subset is obtained by `u[1:n]` or `u[1:]`, meaning from index 1 and the rest of the array. The computation of `d` can now be done without an explicit Python loop:

```
d = u[1:] - u[:-1]
```

or with explicit limits if desired:

```
d = u[1:n] - u[0:n-1]
```

Indices with a colon, going from an index to (but not including) another index are called *slices*. With `numpy` arrays, the computations are still done by loops, but in efficient, compiled, highly optimized C or Fortran code. Such loops are sometimes referred to as *vectorized loops*. Such loops can also easily be distributed among many processors on parallel computers. We say that the *scalar code* above, working on an element (a scalar) at a time, has been replaced by an equivalent *vectorized code*. The process of vectorizing code is called *vectorization*.

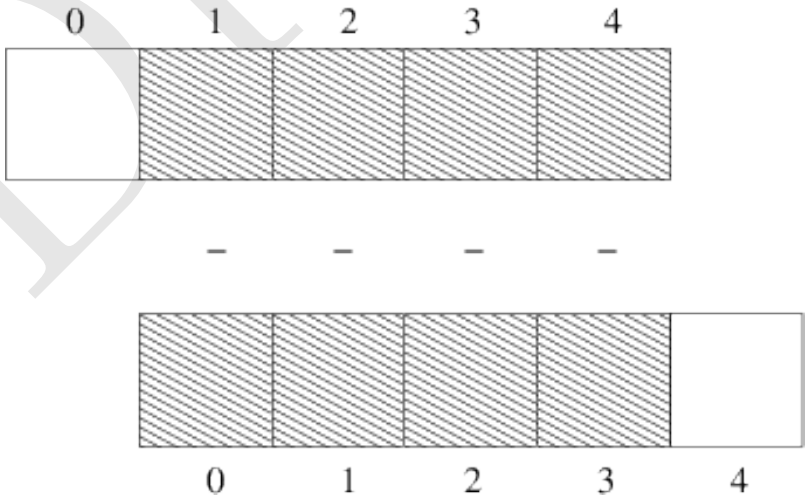


Figure 2.3.: Illustration of subtracting two slices of two arrays.

Test your understanding

Newcomers to vectorization are encouraged to choose a small array `u`, say with five elements, and simulate with pen and paper both the loop version and the vectorized version above.

Finite difference schemes basically contain differences between array elements with shifted indices. As an example, consider the updating formula

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

The vectorization consists of replacing the loop by arithmetics on slices of arrays of length `n-2`:

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

Note that the length of `u2` becomes `n-2`. If `u2` is already an array of length `n` and we want to use the formula to update all the "inner" elements of `u2`, as we will when solving a 1D wave equation, we can write

```
u2[1:-1]  = u[:-2] - 2*u[1:-1] + u[2:]
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

The first expression's right-hand side is realized by the following steps, involving temporary arrays with intermediate results, since each array operation can only involve one or two arrays. The `numpy` package performs (behind the scenes) the first line above in four steps:

```
temp1 = 2*u[1:-1]
temp2 = u[:-2] - temp1
temp3 = temp2 + u[2:]
u2[1:-1] = temp3
```

We need three temporary arrays, but a user does not need to worry about such temporary arrays.

Common mistakes with array slices

Array expressions with slices demand that the slices have the same shape. It easy to make a mistake in, e.g.,

```
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]
```

and write

```
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[1:n]
```

Now `u[1:n]` has wrong length (`n-1`) compared to the other array slices, causing a `ValueError` and the message `could not broadcast input array from shape 103 into shape 104` (if

n is 105). When such errors occur one must closely examine all the slices. Usually, it is easier to get upper limits of slices right when they use -1 or -2 or empty limit rather than expressions involving the length.
Another common mistake, when u2 has length n, is to forget the slice in the array on the left-hand side,

```
u2 = u[0:n-2] - 2*u[1:n-1] + u[1:n]
```

This is really crucial: now u2 becomes a *new* array of length n-2, which is the wrong length as we have no entries for the boundary values. We meant to insert the right-hand side array *into* the original u2 array for the entries that correspond to the internal points in the mesh (1:n-1 or 1:-1).

Vectorization may also work nicely with functions. To illustrate, we may extend the previous example as follows:

```
def f(x):
    return x**2 + 1

for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])
```

Assuming u2, u, and x all have length n, the vectorized version becomes

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:] + f(x[1:-1])
```

Obviously, f must be able to take an array as argument for f(x[1:-1]) to make sense.

2.24. Finite difference schemes expressed as slices

We now have the necessary tools to vectorize the wave equation algorithm as described mathematically in Section Section 2.6 and through code in Section Section 2.16. There are three loops: one for the initial condition, one for the first time step, and finally the loop that is repeated for all subsequent time levels. Since only the latter is repeated a potentially large number of times, we limit our vectorization efforts to this loop. Within the time loop, the space loop reads:

```
for i in range(1, Nx):
    u[i] = 2*u_n[i] - u_nm1[i] + \
           C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
```

The vectorized version becomes

```
u[1:-1] = - u_nm1[1:-1] + 2*u_n[1:-1] + \
          C2*(u_n[:-2] - 2*u_n[1:-1] + u_n[2:])
```

or

```
u[1:Nx] = 2*u_n[1:Nx]- u_nm1[1:Nx] + \
          C2*(u_n[0:Nx-1] - 2*u_n[1:Nx] + u_n[2:Nx+1])
```

The program wave1D_u0v.py contains a new version of the function `solver` where both the scalar and the vectorized loops are included (the argument `version` is set to `scalar` or `vectorized`, respectively).

2.25. Verification

We may reuse the quadratic solution $u_e(x,t) = x(L-x)(1+\frac{1}{2}t)$ for verifying also the vectorized code. A test function can now verify both the scalar and the vectorized version. Moreover, we may use a `user_action` function that compares the computed and exact solution at each time level and performs a test:

```
def test_quadratic():
    """
    Check the scalar and vectorized versions for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    u_exact = lambda x, t: x * (L - x) * (1 + 0.5 * t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5 * u_exact(x, 0)
    f = lambda x, t: np.zeros_like(x) + 2 * c**2 * (1 + 0.5 * t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C * (L / Nx) / c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        tol = 1e-13
        diff = np.abs(u - u_e).max()
        assert diff < tol

    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error, version="scalar")
    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error, version="vectorized")
```

Lambda functions

The code segment above demonstrates how to achieve very compact code, without degraded readability, by use of lambda functions for the various input parameters that require a Python function. In essence,

```
f = lambda x, t: L*(x-t)**2
```

is equivalent to

```
def f(x, t):
    return L(x-t)**2
```

Note that lambda functions can just contain a single expression and no statements.
One advantage with lambda functions is that they can be used directly in calls:

```
solver(I=lambda x: sin(pi*x/L), V=0, f=0, ...)
```

2.26. Efficiency measurements

The `wave1D_u0v.py` contains our new `solver` function with both scalar and vectorized code. For comparing the efficiency of scalar versus vectorized code, we need a `viz` function as discussed in Section Section 2.19. All of this `viz` function can be reused, except the call to `solver_function`. This call lacks the parameter `version`, which we want to set to `vectorized` and `scalar` for our efficiency measurements.

One solution is to copy the `viz` code from `wave1D_u0` into `wave1D_u0v.py` and add a `version` argument to the `solver_function` call. Taking into account how much animation code we then duplicate, this is not a good idea. Alternatively, introducing the `version` argument in `wave1D_u0.viz`, so that this function can be imported into `wave1D_u0v.py`, is not a good solution either, since `version` has no meaning in that file. We need better ideas!

2.26.1. Solution 1

Calling `viz` in `wave1D_u0` with `solver_function` as our new solver in `wave1D_u0v` works fine, since this solver has `version='vectorized'` as default value. The problem arises when we want to test `version='scalar'`. The simplest solution is then to use `wave1D_u0.solver` instead. We make a new `viz` function in `wave1D_u0v.py` that has a `version` argument and that just calls `wave1D_u0.viz`:

```
def viz(
    I,
    V,
    f,
    c,
```

```
    L,
    dt,
    C,
    T,  # PDE parameters
    umin,
    umax,  # Interval for u in plots
    animate=True,  # Simulation with animation?
    solver_function=solver,  # Function with numerical algorithm
    version="vectorized",  # 'scalar' or 'vectorized'
):
    import wave1D_u0

    if version == "vectorized":
        cpu = wave1D_u0.viz(
            I, V, f, c, L, dt, C, T, umin, umax, animate, solver_function=solver
        )
    elif version == "scalar":
        cpu = wave1D_u0.viz(
            I,
            V,
            f,
            c,
            L,
            dt,
            C,
            T,
            umin,
            umax,
            animate,
            solver_function=wave1D_u0.solver,
        )
    return cpu

def test_quadratic():
    """
    Check the scalar and vectorized versions for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    u_exact = lambda x, t: x * (L - x) * (1 + 0.5 * t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5 * u_exact(x, 0)
    f = lambda x, t: np.zeros_like(x) + 2 * c**2 * (1 + 0.5 * t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
```

```
    dt = C * (L / Nx) / c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        tol = 1e-13
        diff = np.abs(u - u_e).max()
        assert diff < tol

    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error, version="scalar")
    solver(I, V, f, c, L, dt, C, T, user_action=assert_no_error, version="vectorized")

def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8 * L
    a = 0.005
    freq = 440
    wavelength = 2 * L
    c = freq * wavelength
    omega = 2 * pi * freq
    num_periods = 1
    T = 2 * pi / omega * num_periods
    dt = L / 50.0 / c

    def I(x):
        return a * x / x0 if x < x0 else a / (L - x0) * (L - x)

    umin = -1.2 * a
    umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)

def run_efficiency_experiments():
    L = 1
    x0 = 0.8 * L
    a = 1
    c = 2
    T = 8
    C = 0.9
    umin = -1.2 * a
    umax = -umin

    def I(x):
        return a * x / x0 if x < x0 else a / (L - x0) * (L - x)

    intervals = []
    speedup = []
```

```
    for Nx in [50, 100, 200, 400, 800]:
        dx = float(L) / Nx
        dt = C / c * dx
        print("solving scalar Nx=%d" % Nx, end=" ")
        cpu_s = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=False, version="scalar")
        print(cpu_s)
        print("solving vectorized Nx=%d" % Nx, end=" ")
        cpu_v = viz(
            I, 0, 0, c, L, dt, C, T, umin, umax, animate=False, version="vectorized"
        )
        print(cpu_v)
        intervals.append(Nx)
        speedup.append(cpu_s / float(cpu_v))
        print("Nx=%3d: cpu_v/cpu_s: %.3f" % (Nx, 1.0 / speedup[-1]))
    print("Nx:", intervals)
    print("Speed-up:", speedup)

if __name__ == "__main__":
    test_quadratic()  # verify
    import sys

    try:
        C = float(sys.argv[1])
        print("C=%g" % C)
    except IndexError:
        C = 0.85
    guitar(C)
```

2.26.2. Solution 2

There is a more advanced and fancier solution featuring a very useful trick: we can make a new function that will always call `wave1D_u0v.solver` with `version='scalar'`. The `functools.partial` function from standard Python takes a function `func` as argument and a series of positional and keyword arguments and returns a new function that will call `func` with the supplied arguments, while the user can control all the other arguments in `func`. Consider a trivial example,

```
def f(a, b, c=2):
    return a + b + c
```

We want to ensure that `f` is always called with `c=3`, i.e., `f` has only two "free" arguments `a` and `b`. This functionality is obtained by

```
import functools
f2 = functools.partial(f, c=3)

print f2(1, 2)  # results in 1+2+3=6
```

Now `f2` calls `f` with whatever the user supplies as `a` and `b`, but `c` is always 3.

Back to our `viz` code, we can do

```
import functools
scalar_solver = functools.partial(wave1D_u0.solver, version='scalar')
cpu = wave1D_u0.viz(
        I, V, f, c, L, dt, C, T, umin, umax,
        animate, tool, solver_function=scalar_solver)
```

The new `scalar_solver` takes the same arguments as `wave1D_u0.scalar` and calls `wave1D_u0v.scalar`, but always supplies the extra argument `version='scalar'`. When sending this `solver_function` to `wave1D_u0.viz`, the latter will call `wave1D_u0v.solver` with all the `I`, `V`, `f`, etc., arguments we supply, plus `version='scalar'`.

2.26.3. Efficiency experiments

We now have a `viz` function that can call our solver function both in scalar and vectorized mode. The function `run_efficiency_experiments` in `wave1D_u0v.py` performs a set of experiments and reports the CPU time spent in the scalar and vectorized solver for the previous string vibration example with spatial mesh resolutions $N_x = 50, 100, 200, 400, 800$. Running this function reveals that the vectorized code runs substantially faster: the vectorized code runs approximately $N_x/10$ times as fast as the scalar code!

2.27. Remark on the updating of arrays

At the end of each time step we need to update the `u_nm1` and `u_n` arrays such that they have the right content for the next time step:

```
u_nm1[:] = u_n
u_n[:] = u
```

The order here is important: updating `u_n` first, makes `u_nm1` equal to `u`, which is wrong!

The assignment `u_n[:] = u` copies the content of the `u` array into the elements of the `u_n` array. Such copying takes time, but that time is negligible compared to the time needed for computing `u` from the finite difference formula, even when the formula has a vectorized implementation. However, efficiency of program code is a key topic when solving PDEs numerically (particularly when there are two or three space dimensions), so it must be mentioned that there exists a much more efficient way of making the arrays `u_nm1` and `u_n` ready for the next time step. The idea is based on *switching references* and explained as follows.

A Python variable is actually a reference to some object (C programmers may think of pointers). Instead of copying data, we can let `u_nm1` refer to the `u_n` object and `u_n` refer to the `u` object. This is a very efficient operation (like switching pointers in C). A naive implementation like

```
u_nm1 = u_n
u_n = u
```

will fail, however, because now `u_nm1` refers to the `u_n` object, but then the name `u_n` refers to `u`, so that this `u` object has two references, `u_n` and `u`, while our third array, originally referred to by `u_nm1`, has no more references and is lost. This means that the variables `u`, `u_n`, and `u_nm1` refer to two arrays and not three. Consequently, the computations at the next time level will be messed up, since updating the elements in `u` will imply updating the elements in `u_n` too, thereby destroying the solution at the previous time step.

While `u_nm1 = u_n` is fine, `u_n = u` is problematic, so the solution to this problem is to ensure that `u` points to the `u_nm1` array. This is mathematically wrong, but new correct values will be filled into `u` at the next time step and make it right.

The correct switch of references is

```
tmp = u_nm1
u_nm1 = u_n
u_n = u
u = tmp
```

We can get rid of the temporary reference `tmp` by writing

```
u_nm1, u_n, u = u_n, u, u_nm1
```

This switching of references for updating our arrays will be used in later implementations.

> **Caution:**
>
> The update `u_nm1, u_n, u = u_n, u, u_nm1` leaves wrong content in `u` at the final time step. This means that if we return `u`, as we do in the example codes here, we actually return `u_nm1`, which is obviously wrong. It is therefore important to adjust the content of `u` to `u = u_n` before returning `u`. (Note that the `user_action` function reduces the need to return the solution from the solver.)

2.28. Making Movies

We could also add making a hardcopy of the plot for later production of a movie file. The hardcopies must be numbered consecutively, say `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so forth. The filename construction can be based on the `n` counter supplied to the user action function:

```
filename = 'tmp_%04d.png' % n
```

The `04d` format implies formatting of an integer in a field of width 4 characters and padded with zeros from the left. An animated GIF file `movie.gif` can be made from these individual frames by using the `convert` program from the ImageMagick suite:

```
Unix> convert -delay 50 tmp_*.png movie.gif
Unix> animate movie.gif
```

The delay is measured in units of 1/100 s. The `animate` program, also in the ImageMagick suite, can play the movie file. Alternatively, the `display` program can be used to walk through each frame, i.e., solution curve, by pressing the space bar.

2.29. Exercise: Simulate a standing wave

The purpose of this exercise is to simulate standing waves on $[0, L]$ and illustrate the error in the simulation. Standing waves arise from an initial condition

$$u(x,0) = A\sin\left(\frac{\pi}{L}mx\right),$$

where m is an integer and A is a freely chosen amplitude. The corresponding exact solution can be computed and reads

$$u_{\mathrm{e}}(x,t) = A\sin\left(\frac{\pi}{L}mx\right)\cos\left(\frac{\pi}{L}mct\right).$$

a)

Explain that for a function $\sin kx\cos\omega t$ the wave length in space is $\lambda = 2\pi/k$ and the period in time is $P = 2\pi/\omega$. Use these expressions to find the wave length in space and period in time of u_{e} above.

Solution

Since the sin and cos functions depend on x and t, respectively, the sin function will run through one period as x increases by $\frac{2\pi}{k}$, while the cos function starts repeating as t increases by $\frac{2\pi}{\omega}$.
The wave length in space becomes

$$\lambda = \frac{2\pi}{\frac{\pi}{L}m} = \frac{2L}{m}.$$

The period in time becomes

$$P = \frac{2\pi}{\frac{\pi}{L}mc} = \frac{2L}{mc}.$$

b)

Import the `solver` function from `wave1D_u0.py` into a new file where the `viz` function is reimplemented such that it plots either the numerical *and* the exact solution, *or* the error.

Solution

See code below.

c)

Make animations where you illustrate how the error $e_i^n = u_e(x_i, t_n) - u_i^n$ develops and increases in time. Also make animations of u and u_e simultaneously.

Quite long time simulations are needed in order to display significant

discrepancies between the numerical and exact solution.

A possible set of parameters is $L = 12$, $m = 9$, $c = 2$, $A = 1$, $N_x = 80$,

$C = 0.8$. The error mesh function e^n can be simulated for 10 periods, while 20-30 periods are needed to show significant differences between the curves for the numerical and exact solution.

Solution

The code:

```
import os
import sys

sys.path.insert(0, os.path.join(os.pardir, os.pardir, "src-wave", "wave1D"))

import numpy as np
from wave1D_u0 import solver


def viz(
    I, V, f, c, L, dt, C, T,
    ymax,  # y axis: [-ymax, ymax]
    u_exact,  # u_exact(x, t)
    animate="u and u_exact",  # or 'error'
    movie_filename="movie",
):
    """Run solver and visualize u at each time level."""
    import glob
    import os

    import matplotlib.pyplot as plt

    class Plot:
        def __init__(self, ymax, frame_name="frame"):
            self.max_error = []  # hold max amplitude errors
            self.max_error_t = []  # time points corresp. to max_error
            self.frame_name = frame_name
            self.ymax = ymax

        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if animate == "u and u_exact":
                plt.clf()
                plt.plot(x, u, "r-", x, u_exact(x, t[n]), "b--")
                plt.xlabel("x")
                plt.ylabel("u")
                plt.axis([0, L, -self.ymax, self.ymax])
                plt.title(f"t={t[n]:f}")
                plt.draw()
                plt.pause(0.001)
            else:
                error = u_exact(x, t[n]) - u
                local_max_error = np.abs(error).max()
                if self.max_error == [] or local_max_error > max(self.max_error):
                    self.max_error.append(local_max_error)
                    self.max_error_t.append(t[n])
                self.ymax = max(self.ymax, max(self.max_error))
                plt.clf()
                plt.plot(x, error, "r-")
                plt.xlabel("x")
                plt.ylabel("error")
                plt.axis([0, L, -self.ymax, self.ymax])
                plt.title(f"t={t[n]:f}")
```

Remarks

The important parameters for numerical quality are C and $k\Delta x$, where $C = c\Delta t/\Delta x$ is the Courant number and k is defined above ($k\Delta x$ is proportional to how many mesh points we have per wave length in space, see Section Section 2.64 for explanation).

2.30. Exercise: Add storage of solution in a user action function

Extend the `plot_u` function in the file `wave1D_u0.py` to also store the solutions `u` in a list. To this end, declare `all_u` as an empty list in the `viz` function, outside `plot_u`, and perform an append operation inside the `plot_u` function. Note that a function, like `plot_u`, inside another function, like `viz`, remembers all local variables in `viz` function, including `all_u`, even when `plot_u` is called (as `user_action`) in the `solver` function. Test both `all_u.append(u)` and `all_u.append(u.copy())`. Why does one of these constructions fail to store the solution correctly? Let the `viz` function return the `all_u` list converted to a two-dimensional `numpy` array.

Solution

We have to explicitly use a copy of u, i.e. as `all_u.append(u.copy())`, otherwise we just get a reference to `u`, which goes on changing with the computations.

```
def viz(
    I, V, f, c, L, dt, C, T,
    umin, umax,
    animate=True,
    solver_function=solver,
):
    """Run solver, store and visualize u at each time level."""
    import glob
    import os
    import time

    import matplotlib.pyplot as plt

    all_u = []  # store solutions

    def plot_u(u, x, t, n):
        """user_action function for solver."""
        if n == 0:
            plt.ion()
            lines = plt.plot(x, u, "r-")
            plt.xlabel("x")
            plt.ylabel("u")
            plt.axis([0, L, umin, umax])
            plt.legend([f"t={t[n]:f}"], loc="lower left")
        else:
            lines[0].set_ydata(u)
            plt.legend([f"t={t[n]:f}"], loc="lower left")
            plt.draw()
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig("tmp_%04d.png" % n)
        all_u.append(u.copy())  # must use copy!

    # Clean up old movie frames
    for filename in glob.glob("tmp_*.png"):
        os.remove(filename)

    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(I, V, f, c, L, dt, C, T, user_action)
    return cpu, np.array(all_u)
```

2.31. Exercise: Use a class for the user action function

Redo Exercise Section 2.30 using a class for the user action function. Let the `all_u` list be an attribute in this class and implement the user action function as a method (the special method `__call__` is a natural choice). The class versions avoid that the user action function depends on

parameters defined outside the function (such as `all_u` in Exercise Section 2.30).

Solution

Using a class, we get

```
class PlotMatplotlib:
    def __init__(self):
        self.all_u = []

    def __call__(self, u, x, t, n):
        """user_action function for solver."""
        if n == 0:
            plt.ion()
            self.lines = plt.plot(x, u, "r-")
            plt.xlabel("x")
            plt.ylabel("u")
            plt.axis([0, L, umin, umax])
            plt.legend([f"t={t[n]:f}"], loc="lower left")
        else:
            self.lines[0].set_ydata(u)
            plt.legend([f"t={t[n]:f}"], loc="lower left")
            plt.draw()
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig("tmp_%04d.png" % n)  # for movie making
        self.all_u.append(u.copy())


def viz(I, V, f, c, L, dt, C, T, umin, umax,
        animate=True, solver_function=solver):
    """Run solver, store and visualize u at each time level."""
    import glob
    import os

    plot_u = PlotMatplotlib()

    # Clean up old movie frames
    for filename in glob.glob("tmp_*.png"):
        os.remove(filename)

    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(I, V, f, c, L, dt, C, T, user_action)
    return cpu, np.array(plot_u.all_u)
```

2.32. Exercise: Compare several Courant numbers in one movie

The goal of this exercise is to make movies where several curves, corresponding to different Courant numbers, are visualized. Write a program that resembles `wave1D_u0_s2c.py` in Exercise Section 2.31, but with a `viz` function that can take a list of `C` values as argument and create a movie with solutions corresponding to the given `C` values. The `plot_u` function must be changed to store the solution in an array (see Exercise Section 2.30 or Section 2.31 for details), `solver` must be computed for each value of the Courant number, and finally one must run through each time step and plot all the spatial solution curves in one figure and store it in a file.

The challenge in such a visualization is to ensure that the curves in one plot correspond to the same time point. The easiest remedy is to keep the time resolution constant and change the space resolution to change the Courant number. Note that each spatial grid is needed for the final plotting, so it is an option to store those grids too.

Solution

Modifying the code to store all solutions for each C value and also each corresponding spatial grid (needed for final plotting), we get

```
class PlotMatplotlib:
    def __init__(self):
        self.all_u = []
        self.all_u_for_all_C = []
        self.x_mesh = []  # need each mesh for final plots

    def __call__(self, u, x, t, n):
        """user_action function for solver."""
        self.all_u.append(u.copy())
        if t[n] == T:  # i.e., whole time interv. done for this C
            self.x_mesh.append(x.copy())
            self.all_u_for_all_C.append(self.all_u)
            self.all_u = []  # reset to empty list

            if len(self.all_u_for_all_C) == len(C):  # all C done
                print("Finished all C. Proceed with plots...")
                plt.ion()
                for n_ in range(0, n + 1):  # for each tn
                    plt.clf()
                    for j in range(len(C)):
                        plt.plot(self.x_mesh[j], self.all_u_for_all_C[j][n_])
                    plt.axis([0, L, umin, umax])
                    plt.xlabel("x")
                    plt.ylabel("u")
                    plt.title(f"Solutions for all C at t={t[n_]:f}")
                    plt.draw()
                    time.sleep(2) if t[n_] == 0 else time.sleep(0.2)
                    plt.savefig("tmp_%04d.png" % n_)  # for movie


def viz(I, V, f, c, L, dt, C, T, umin, umax,
        animate=True, solver_function=solver):
    """Run solver, store and viz. u at each time level with all C values."""
    import glob
    import os

    plot_u = PlotMatplotlib()

    # Clean up old movie frames
    for filename in glob.glob("tmp_*.png"):
        os.remove(filename)

    user_action = plot_u if animate else None
    for C_value in C:
        print("C_value:", C_value)
        u, x, t, cpu = solver_function(I, V, f, c, L, dt, C_value, T, user_action)

    return cpu
```

2.33. Exercise: Implementing the solver function as a generator

The callback function `user_action(u, x, t, n)` is called from the `solver` function (in, e.g., `wave1D_u0.py`) at every time level and lets the user work perform desired actions with the solution, like plotting it on the screen. We have implemented the callback function in the typical way it would have been done in C and Fortran. Specifically, the code looks like

```
if user_action is not None:
    if user_action(u, x, t, n):
        break
```

Many Python programmers, however, may claim that `solver` is an iterative process, and that iterative processes with callbacks to the user code is more elegantly implemented as *generators*. The rest of the text has little meaning unless you are familiar with Python generators and the `yield` statement.

Instead of calling `user_action`, the `solver` function issues a `yield` statement, which is a kind of `return` statement:

```
yield u, x, t, n
```

The program control is directed back to the calling code:

```
for u, x, t, n in solver(...):
```

When the block is done, `solver` continues with the statement after `yield`. Note that the functionality of terminating the solution process if `user_action` returns a `True` value is not possible to implement in the generator case.

Implement the `solver` function as a generator, and plot the solution at each time step.

Solution

2.34. Project: Calculus with 1D mesh functions

This project explores integration and differentiation of mesh functions, both with scalar and vectorized implementations. We are given a mesh function f_i on a spatial one-dimensional mesh $x_i = i\Delta x$, $i = 0, \ldots, N_x$, over the interval $[a, b]$.

a)

Define the discrete derivative of f_i by using centered differences at internal mesh points and one-sided differences at the end points. Implement a scalar version of the computation in a Python function and write an associated unit test for the linear case $f(x) = 4x - 2.5$ where the discrete derivative should be exact.

Solution

See code below.

b)

Vectorize the implementation of the discrete derivative. Extend the unit test to check the validity of the implementation.

Solution

See code below.

c)

To compute the discrete integral F_i of f_i, we assume that the mesh function f_i varies linearly between the mesh points. Let $f(x)$ be such a linear interpolant of f_i. We then have

$$F_i = \int_{x_0}^{x_i} f(x)dx\,.$$

The exact integral of a piecewise linear function $f(x)$ is given by the Trapezoidal rule. Show that if F_i is already computed, we can find F_{i+1} from

$$F_{i+1} = F_i + \frac{1}{2}(f_i + f_{i+1})\Delta x\,.$$

Make a function for the scalar implementation of the discrete integral as a mesh function. That is, the function should return F_i for $i = 0, \ldots, N_x$. For a unit test one can use the fact that the above defined discrete integral of a linear function (say $f(x) = 4x - 2.5$) is exact.

Solution

We know that the difference $F_{i+1} - F_i$ must amount to the area of a trapezoid, which is exactly what $\frac{1}{2}(f_i + f_{i+1})\Delta x$ is. To show the relation above, we may start with the Trapezoidal rule:

$$F_{i+1} = \Delta x \left[\frac{1}{2}f(x_0) + \sum_{j=1}^{n-1} f(x_j) + \frac{1}{2}f(x_n)\right]\,.$$

Since $n = i + 1$, and since the final term in the sum may be separated out from the sum and split in two, this may be written as

$$F_{i+1} = \Delta x \left[\frac{1}{2}f(x_0) + \sum_{j=1}^{i-1} f(x_j) + \frac{1}{2}f(x_i) + \frac{1}{2}f(x_i) + \frac{1}{2}f(x_{i+1})\right]\,.$$

This may further be written as

$$F_{i+1} = \Delta x \left[\frac{1}{2}f(x_0) + \sum_{j=1}^{i-1} f(x_j) + \frac{1}{2}f(x_i)\right] + \Delta x \left[\frac{1}{2}f(x_i) + \frac{1}{2}f(x_{i+1})\right]\,.$$

Finally, this gives

$$F_{i+1} = F_i + \frac{1}{2}(f_i + f_{i+1})\Delta x\,.$$

See code below for implementation.

d)

Vectorize the implementation of the discrete integral. Extend the unit test to check the validity of the implementation.

Interpret the recursive formula for F_{i+1} as a sum.

Make an array with each element of the sum and use the "cumsum" (`numpy.cumsum`) operation to compute the accumulative sum: `numpy.cumsum([1,3,5])` is `[1,4,9]`.

Solution

See code below.

e)

Create a class `MeshCalculus` that can integrate and differentiate mesh functions. The class can just define some methods that call the previously implemented Python functions. Here is an example on the usage:

```
import numpy as np
calc = MeshCalculus(vectorized=True)
x = np.linspace(0, 1, 11)        # mesh
f = np.exp(x)                    # mesh function
df = calc.differentiate(f, x)    # discrete derivative
F = calc.integrate(f, x)         # discrete anti-derivative
```

Solution

See code below.

Solution

The final version of the code reads

```
"""
Calculus with a 1D mesh function.
"""

import numpy as np


class MeshCalculus:
    def __init__(self, vectorized=True):
        self.vectorized = vectorized

    def differentiate(self, f, x):
        """
        Computes the derivative of f by centered differences, but
        forw and back difference at the start and end, respectively.
        """
        dx = x[1] - x[0]
        Nx = len(x) - 1  # number of spatial steps
        num_dfdx = np.zeros(Nx + 1)
        # Compute approximate derivatives at end-points first
        num_dfdx[0] = (f(x[1]) - f(x[0])) / dx  # FD approx.
        num_dfdx[Nx] = (f(x[Nx]) - f(x[Nx - 1])) / dx  # BD approx.
        # proceed with approximate derivatives for inner mesh points
        if self.vectorized:
            num_dfdx[1:-1] = (f(x[2:]) - f(x[:-2])) / (2 * dx)
        else:  # scalar version
            for i in range(1, Nx):
                num_dfdx[i] = (f(x[i + 1]) - f(x[i - 1])) / (2 * dx)
        return num_dfdx

    def integrate(self, f, x):
        """
        Computes the integral of f(x) over the interval
        covered by x.
        """
        dx = x[1] - x[0]
        F = np.zeros(len(x))
        F[0] = 0  # starting value for iterative scheme
        if self.vectorized:
            all_trapezoids = np.zeros(len(x) - 1)
            all_trapezoids[:] = 0.5 * (f(x[:-1]) + f(x[1:])) * dx
            F[1:] = np.cumsum(all_trapezoids)
        else:  # scalar version
            for i in range(0, len(x) - 1):
                F[i + 1] = F[i] + 0.5 * (f(x[i]) + f(x[i + 1])) * dx
        return F


def test_differentiate():
    def f(x):
        return 4 * x - 2.5
```

2.35. Neumann boundary conditions

The boundary condition $u = 0$ in a wave equation reflects the wave, but u changes sign at the boundary, while the condition $u_x = 0$ reflects the wave as a mirror and preserves the sign.

Why is it so? Consider the boundary $x = 0$ and the condition $u = 0$. How will two values $u(0,t)$ and $u(\Delta x, t$ change from time t to $t+\Delta t$? Since $u(0,t) = 0$, $u(\Delta x, t)$ and $u(\Delta x, t+\Delta t)$ will be close to zero too. Their average in time must also be close to zero, especially in the limit $\Delta x, \Delta t \to 0$:

$$\frac{1}{2}(u(\Delta x, t) + u(\Delta x, t+\Delta t)) \approx 0 \quad \Rightarrow \quad u(\Delta x, t+\Delta t) = -u(\Delta x, t)\,.$$

This tells that u changes sign in time close to the boundary (otherwise the average would be larger than the u values and this is not compatible with keeping neighboring value $u(0,t)$ fixed at zero).

For a Neumann condition $u_x = 0$ at $x = 0$ we consider the values $u(0,t)$, $u(\Delta x, t)$, $u(0, t+\Delta t)$ and $u(\Delta x, t+\Delta t)$. Now the boundary condition demands $u(0,t) \approx u(\Delta x, t)$ and $u(0, t+\Delta t) \approx u(\Delta x, t+\Delta t)$ to always get a flat spatial derivative.

Our next task is to explain how to implement the boundary condition $u_x = 0$, which is more complicated to express numerically and also to implement than a given value of u. We shall present two methods for implementing $u_x = 0$ in a finite difference scheme, one based on deriving a modified stencil at the boundary, and another one based on extending the mesh with ghost cells and ghost points.

2.36. Neumann boundary condition

When a wave hits a boundary and is to be reflected back, one applies the condition

$$\frac{\partial u}{\partial n} \equiv \boldsymbol{n}\cdot\nabla u = 0\,. \tag{2.31}$$

The derivative $\partial/\partial n$ is in the outward normal direction from a general boundary. For a 1D domain $[0, L]$, we have that

$$\left.\frac{\partial}{\partial n}\right|_{x=L} = \left.\frac{\partial}{\partial x}\right|_{x=L}, \quad \left.\frac{\partial}{\partial n}\right|_{x=0} = -\left.\frac{\partial}{\partial x}\right|_{x=0}\,.$$

Boundary condition terminology

Boundary conditions that specify the value of $\partial u/\partial n$ (or shorter u_n) are known as Neumann conditions, while Dirichlet conditions refer to specifications of u. When the values are zero ($\partial u/\partial n = 0$ or $u = 0$) we speak about *homogeneous* Neumann or Dirichlet conditions.

2.37. Discretization of derivatives at the boundary

How can we incorporate the condition (2.31) in the finite difference scheme? Since we have used central differences in all the other approximations to derivatives in the scheme, it is tempting to implement (2.31) at $x = 0$ and $t = t_n$ by the difference

$$[D_{2x}u]_0^n = \frac{u_{-1}^n - u_1^n}{2\Delta x} = 0\,. \tag{2.32}$$

The problem is that u_{-1}^n is not a u value that is being computed since the point is outside the mesh. However, if we combine (2.32) with the scheme

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u^n **i + 1 - 2u^n **i + u_{i-1}^n\right), \tag{2.33}$$

for $i = 0$, we can eliminate the fictitious value u_{-1}^n. We see that $u_{-1}^n = u_1^n$ from (2.32), which can be used in (2.33) to arrive at a modified scheme for the boundary point u_0^{n+1}:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u^n **i + 1 - u^n **i\right), \quad i = 0\,.$$

Figure Figure 2.4 visualizes this equation for computing u_0^3 in terms of u_0^2, u_0^1, and u_1^2.

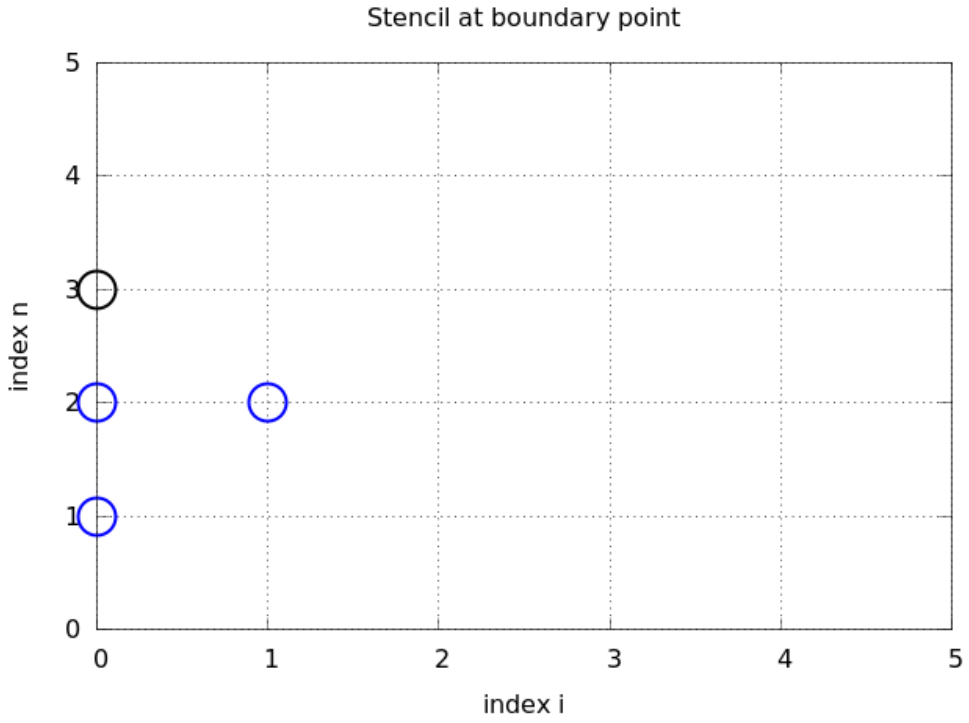


Figure 2.4.: Modified stencil at a boundary with a Neumann condition.

Similarly, (2.31) applied at $x = L$ is discretized by a central difference

$$\frac{u_{N_x+1}^n - u_{N_x-1}^n}{2\Delta x} = 0\,. \tag{2.34}$$

Combined with the scheme for $i = N_x$ we get a modified scheme for the boundary value $u_{N_x}^{n+1}$:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u^n **i - 1 - u^n **i\right), \quad i = N_x\,.$$

The modification of the scheme at the boundary is also required for the special formula for the first time step.

2.38. Implementation of Neumann conditions

We have seen in the preceding section that the special formulas for the boundary points arise from replacing u^n_{i-1} by u^n_{i+1} when computing u^{n+1}_i from the stencil formula for $i = 0$. Similarly, we replace u^n_{i+1} by u^n_{i-1} in the stencil formula for $i = N_x$. This observation can conveniently be used in the coding: we just work with the general stencil formula, but write the code such that it is easy to replace `u[i-1]` by `u[i+1]` and vice versa. This is achieved by having the indices `i+1` and `i-1` as variables `ip1` (`i` plus 1) and `im1` (`i` minus 1), respectively. At the boundary we can easily define `im1=i+1` while we use `im1=i-1` in the internal parts of the mesh. Here are the details of the implementation (note that the updating formula for `u[i]` is the general stencil formula):

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])
```

We can in fact create one loop over both the internal and boundary points and use only one updating formula:

```
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])
```

The program wave1D_n0.py contains a complete implementation of the 1D wave equation with boundary conditions $u_x = 0$ at $x = 0$ and $x = L$.

It would be nice to modify the `test_quadratic` test case from the `wave1D_u0.py` with Dirichlet conditions, described in Section Section 2.25. However, the Neumann conditions require the polynomial variation in the x direction to be of third degree, which causes challenging problems when designing a test where the numerical solution is known exactly. Exercise Section 2.60 outlines ideas and code for this purpose. The only test in `wave1D_n0.py` is to start with a plug wave at rest and see that the initial condition is reached again perfectly after one period of motion, but such a test requires $C = 1$ (so the numerical solution coincides with the exact solution of the PDE, see Section Section 2.64).

2.39. Index set notation

To improve our mathematical writing and our implementations, it is wise to introduce a special notation for index sets. This means that we write x_i, followed by $i \in \mathcal{I}_x$, instead of $i = 0, \ldots, N_x$. Obviously, $\mathcal{I}_x$ must be the index set $\mathcal{I}_x = \{0, \ldots, N_x\}$, but it is often advantageous to have a symbol for this set rather than specifying all its elements (all the time, as we have done up to now). This new notation saves writing and makes specifications of algorithms and their implementation as computer code simpler.

The first index in the set will be denoted $\mathcal{I}_x^0$ and the last $\mathcal{I}_x^{-1}$. When we need to skip the first element of the set, we use $\mathcal{I}_x^+$ for the remaining subset $\mathcal{I}_x^+ = \{1, \ldots, N_x\}$. Similarly, if the last element is to be dropped, we write $\mathcal{I}_x^- = \{0, \ldots, N_x - 1\}$ for the remaining indices. All the indices corresponding to inner grid points are specified by $\mathcal{I}_x^i = \{1, \ldots, N_x - 1\}$. For the time domain we find it natural to explicitly use 0 as the first index, so we will usually write $n = 0$ and t_0 rather than $n = \mathcal{I}_t^0$. We also avoid notation like $x_{\mathcal{I}_x^{-1}}$ and will instead use x_i, $i = \mathcal{I}_x^{-1}$.

The Python code associated with index sets applies the following conventions:

Notation	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}_x^0$	`Ix[0]`
$\mathcal{I}_x^{-1}$	`Ix[-1]`
$\mathcal{I}_x^-$	`Ix[:-1]`
$\mathcal{I}_x^+$	`Ix[1:]`
$\mathcal{I}_x^i$	`Ix[1:-1]`

Why index sets are useful

An important feature of the index set notation is that it keeps our formulas and code independent of how we count mesh points. For example, the notation $i \in \mathcal{I}_x$ or $i = \mathcal{I}_x^0$ remains the same whether $\mathcal{I}_x$ is defined as above or as starting at 1, i.e., $\mathcal{I}_x = \{1, \ldots, Q\}$. Similarly, we can in the code define `Ix=range(Nx+1)` or `Ix=range(1,Q)`, and expressions like `Ix[0]` and `Ix[1:-1]` remain correct. One application where the index set notation is convenient is conversion of code from a language where arrays has base index 0 (e.g., Python and C) to languages where the base index is 1 (e.g., MATLAB and Fortran). Another important application is implementation of Neumann conditions via ghost points (see next section).

For the current problem setting in the x, t plane, we work with the index sets

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\},$$

defined in Python as

```
Ix = range(0, Nx+1)
It = range(0, Nt+1)
```

A finite difference scheme can with the index set notation be specified as

$$
\begin{aligned}
u_i^{n+1} &= u_i^n - \frac{1}{2}C^2\left(u^n**i+1 - 2u^n**i + u_{i-1}^n\right), \quad , i \in \mathcal{I}_x^i,\ n=0,\\
u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + C^2\left(u^n**i+1 - 2u^n**i + u_{i-1}^n\right), \quad i \in \mathcal{I}_x^i,\ n \in \mathcal{I}_t^i,\\
u_i^{n+1} &= 0, \quad i = \mathcal{I}_x^0,\ n \in \mathcal{I}_t^-,\\
u_i^{n+1} &= 0, \quad i = \mathcal{I}_x^{-1},\ n \in \mathcal{I}_t^- .
\end{aligned}
$$

The corresponding implementation becomes

```
for i in Ix[1:-1]:
    u[i] = u_n[i] - 0.5*C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])

for n in It[1:-1]:
    for i in Ix[1:-1]:
        u[i] = - u_nm1[i] + 2*u_n[i] + \
               C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

i The program `wave1D_dn.py`

applies the index set notation and solves the 1D wave equation $u_{tt} = c^2u_{xx} + f(x,t)$ with quite general boundary and initial conditions:

- $x = 0$: $u = U_0(t)$ or $u_x = 0$
- $x = L$: $u = U_L(t)$ or $u_x = 0$
- $t = 0$: $u = I(x)$
- $t = 0$: $u_t = V(x)$

The program combines Dirichlet and Neumann conditions, scalar and vectorized implementation of schemes, and the index set notation into one piece of code. A lot of test examples are also included in the program:

- A rectangular plug-shaped initial condition. (For $C = 1$ the solution will be a rectangle that jumps one cell per time step, making the case well suited for verification.)
- A Gaussian function as initial condition.
- A triangular profile as initial condition, which resembles the typical initial shape of a guitar string.
- A sinusoidal variation of u at $x = 0$ and either $u = 0$ or $u_x = 0$ at $x = L$.
- An analytical solution $u(x,t) = \cos(m\pi t/L)\sin(\frac{1}{2}m\pi x/L)$, which can be used for convergence rate tests.

2.40. Verifying the implementation of Neumann conditions

How can we test that the Neumann conditions are correctly implemented? The `solver` function in the `wave1D_dn.py` program described in the box above accepts Dirichlet or Neumann conditions at

$x = 0$ and $x = L$. It is tempting to apply a quadratic solution as described in Sections Section 2.8 and Section 2.17, but it turns out that this solution is no longer an exact solution of the discrete equations if a Neumann condition is implemented on the boundary. A linear solution does not help since we only have homogeneous Neumann conditions in `wave1D_dn.py`, and we are consequently left with testing just a constant solution: $u = \text{const}$.

```
def test_constant():
    """
    Check the scalar and vectorized versions for
    a constant u(x,t). We simulate in [0, L] and apply
    Neumann and Dirichlet conditions at both ends.
    """
    u_const = 0.45
    u_exact = lambda x, t: u_const
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0
    f = lambda x, t: 0

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(u - u_e).max()
        msg = "diff=%E, t_%d=%g" % (diff, n, t[n])
        tol = 1e-13
        assert diff < tol, msg

    for U_0 in (None, lambda t: u_const):
        for U_L in (None, lambda t: u_const):
            L = 2.5
            c = 1.5
            C = 0.75
            Nx = 3  # Very coarse mesh for this exact test
            dt = C * (L / Nx) / c
            T = 18  # long time integration

            solver(
                I,
                V,
                f,
                c,
                U_0,
                U_L,
                L,
                dt,
                C,
                T,
                user_action=assert_no_error,
                version="scalar",
            )
```

```
            solver(
                I,
                V,
                f,
                c,
                U_0,
                U_L,
                L,
                dt,
                C,
                T,
                user_action=assert_no_error,
                version="vectorized",
            )
            print(U_0, U_L)
```

The quadratic solution is very useful for testing, but it requires Dirichlet conditions at both ends.

Another test may utilize the fact that the approximation error vanishes when the Courant number is unity. We can, for example, start with a plug profile as initial condition, let this wave split into two plug waves, one in each direction, and check that the two plug waves come back and form the initial condition again after "one period" of the solution process. Neumann conditions can be applied at both ends. A proper test function reads

```
def test_plug():
    """Check that an initial plug is correct back after one period."""
    L = 1.0
    c = 0.5
    dt = (L / 10) / c  # Nx=10
    I = lambda x: 0 if abs(x - L / 2.0) > 0.1 else 1

    u_s, x, t, cpu = solver(
        I=I,
        V=None,
        f=None,
        c=0.5,
        U_0=None,
        U_L=None,
        L=L,
        dt=dt,
        C=1,
        T=4,
        user_action=None,
        version="scalar",
    )
    u_v, x, t, cpu = solver(
        I=I,
```

```
        V=None,
        f=None,
        c=0.5,
        U_0=None,
        U_L=None,
        L=L,
        dt=dt,
        C=1,
        T=4,
        user_action=None,
        version="vectorized",
    )
    tol = 1e-13
    diff = abs(u_s - u_v).max()
    assert diff < tol
    u_0 = np.array([I(x_) for x_ in x])
    diff = np.abs(u_s - u_0).max()
    assert diff < tol
```

Other tests must rely on an unknown approximation error, so effectively we are left with tests on the convergence rate.

2.41. Alternative implementation via ghost cells

2.41.1. Idea

Instead of modifying the scheme at the boundary, we can introduce extra points outside the domain such that the fictitious values u^n_{-1} and $u^n_{N_x+1}$ are defined in the mesh. Adding the intervals $[-\Delta x, 0]$ and $[L, L+\Delta x]$, known as *ghost cells*, to the mesh gives us all the needed mesh points, corresponding to $i = -1, 0, \ldots, N_x, N_x+1$. The extra points with $i = -1$ and $i = N_x + 1$ are known as *ghost points*, and values at these points, u^n_{-1} and $u^n_{N_x+1}$, are called *ghost values*.

The important idea is to ensure that we always have

$$u^n_{-1} = u^n_1 \text{ and } u^n_{N_x+1} = u^n_{N_x-1},$$

because then the application of the standard scheme at a boundary point $i = 0$ or $i = N_x$ will be correct and guarantee that the solution is compatible with the boundary condition $u_x = 0$.

Some readers may find it strange to just extend the domain with ghost cells as a general technique, because in some problems there is a completely different medium with different physics and equations right outside of a boundary. Nevertheless, one should view the ghost cell technique as a purely mathematical technique, which is valid in the limit $\Delta x \to 0$ and helps us to implement derivatives.

2.41.2. Implementation

The `u` array now needs extra elements corresponding to the ghost points. Two new point values are needed:

```
u   = zeros(Nx+3)
```

The arrays `u_n` and `u_nm1` must be defined accordingly.

Unfortunately, a major indexing problem arises with ghost cells. The reason is that Python indices *must* start at 0 and `u[-1]` will always mean the last element in `u`. This fact gives, apparently, a mismatch between the mathematical indices $i = -1, 0, \ldots, N_x + 1$ and the Python indices running over `u`: `0,..,Nx+2`. One remedy is to change the mathematical indexing of i in the scheme and write

$$u_i^{n+1} = \cdots, \quad i = 1, \ldots, N_x + 1,$$

instead of $i = 0, \ldots, N_x$ as we have previously used. The ghost points now correspond to $i = 0$ and $i = N_x + 1$. A better solution is to use the ideas of Section Section 2.39: we hide the specific index value in an index set and operate with inner and boundary points using the index set notation.

To this end, we define `u` with proper length and `Ix` to be the corresponding indices for the real physical mesh points $(1, 2, \ldots, N_x + 1)$:

```
u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)
```

That is, the boundary points have indices `Ix[0]` and `Ix[-1]` (as before). We first update the solution at all physical mesh points (i.e., interior points in the mesh):

```
for i in Ix:
    u[i] = - u_nm1[i] + 2*u_n[i] + \
           C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
```

The indexing becomes a bit more complicated when we call functions like `V(x)` and `f(x, t)`, as we must remember that the appropriate x coordinate is given as `x[i-Ix[0]]`:

```
for i in Ix:
    u[i] = u_n[i] + dt*V(x[i-Ix[0]]) + \
           0.5*C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
           0.5*dt2*f(x[i-Ix[0]], t[0])
```

It remains to update the solution at ghost points, i.e., `u[0]` and `u[-1]` (or `u[Nx+2]`). For a boundary condition $u_x = 0$, the ghost value must equal the value at the associated inner mesh point. Computer code makes this statement precise:

```
i = Ix[0]           # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1]          # x=L boundary
u[i+1] = u[i-1]
```

The physical solution to be plotted is now in `u[1:-1]`, or equivalently `u[Ix[0]:Ix[-1]+1]`, so this slice is the quantity to be returned from a solver function. A complete implementation appears in the program `wave1D_n0_ghost.py`.

> ⚠ We have to be careful with how the spatial and temporal mesh
>
> points are stored. Say we let `x` be the physical mesh points,
>
> ```
> x = linspace(0, L, Nx+1)
> ```
>
> "Standard coding" of the initial condition,
>
> ```
> for i in Ix:
>     u_n[i] = I(x[i])
> ```
>
> becomes wrong, since `u_n` and `x` have different lengths and the index `i` corresponds to two different mesh points. In fact, `x[i]` corresponds to `u[1+i]`. A correct implementation is
>
> ```
> for i in Ix:
>     u_n[i] = I(x[i-Ix[0]])
> ```
>
> Similarly, a source term usually coded as `f(x[i], t[n])` is incorrect if `x` is defined to be the physical points, so `x[i]` must be replaced by `x[i-Ix[0]]`.
> An alternative remedy is to let `x` also cover the ghost points such that `u[i]` is the value at `x[i]`.

The ghost cell is only added to the boundary where we have a Neumann condition. Suppose we have a Dirichlet condition at $x = L$ and a homogeneous Neumann condition at $x = 0$. One ghost cell $[-\Delta x, 0]$ is added to the mesh, so the index set for the physical points becomes $\{1, \ldots, N_x + 1\}$. A relevant implementation is

```
u = zeros(Nx+2)
Ix = range(1, u.shape[0])
...
for i in Ix[:-1]:
    u[i] = - u_nm1[i] + 2*u_n[i] + \
           C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
           dt2*f(x[i-Ix[0]], t[n])
i = Ix[-1]
u[i] = U_0       # set Dirichlet value
i = Ix[0]
u[i-1] = u[i+1]  # update ghost value
```

The physical solution to be plotted is now in `u[1:]` or (as always) `u[Ix[0]:Ix[-1]+1]`.

2.42. Variable wave velocity

Our next generalization of the 1D wave equation (2.1) or (2.12) is to allow for a variable wave velocity c: $c = c(x)$, usually motivated by wave motion in a domain composed of different physical media. When the media differ in physical properties like density or porosity, the wave velocity c is affected and will depend on the position in space. Figure Figure 2.5 shows a wave propagating in one medium $[0, 0.7] \cup [0.9, 1]$ with wave velocity c_1 (left) before it enters a second medium $(0.7, 0.9)$ with wave velocity c_2 (right). When the wave meets the boundary where c jumps from c_1 to c_2, a part of the wave is reflected back into the first medium (the *reflected* wave), while one part is transmitted through the second medium (the *transmitted* wave).

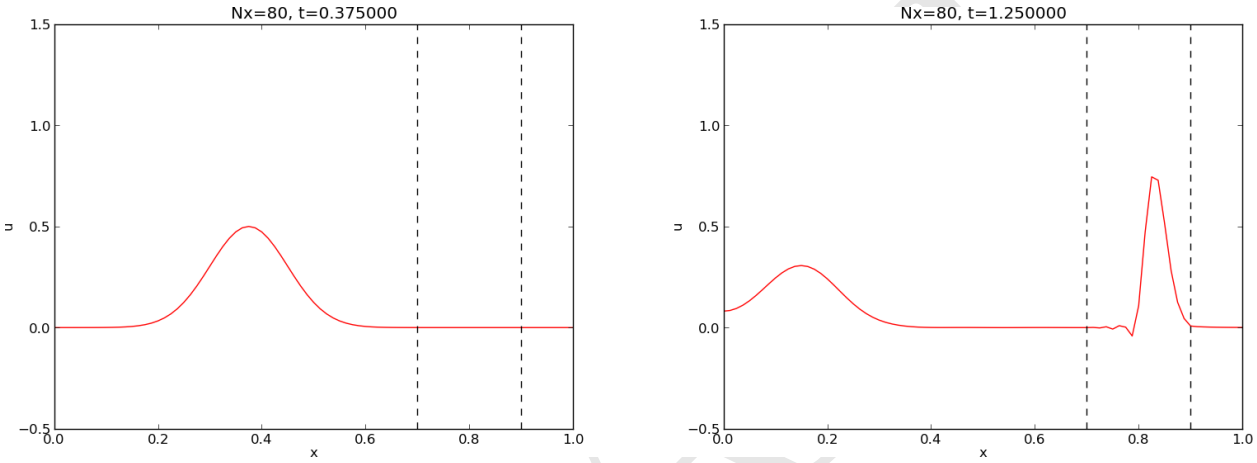


Figure 2.5.: Left: wave entering another medium; right: transmitted and reflected wave.

2.43. The model PDE with a variable coefficient

Instead of working with the squared quantity $c^2(x)$, we shall for notational convenience introduce $q(x) = c^2(x)$. A 1D wave equation with variable wave velocity often takes the form

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{2.35}$$

This is the most frequent form of a wave equation with variable wave velocity, but other forms also appear, see Section Section 2.81 and equation (2.118).

As usual, we sample (2.35) at a mesh point,

$$\frac{\partial^2}{\partial t^2}u(x_i,t_n) = \frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) + f(x_i,t_n),$$

where the only new term to discretize is

$$\frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) = \left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n.$$

Discretizing the variable coefficient {#sec-wave-pde2-var-c-ideas}

The principal idea is to first discretize the outer derivative. Define

$$\phi = q(x)\frac{\partial u}{\partial x},$$

and use a centered derivative around $x = x_i$ for the derivative of ϕ:

$$\left[\frac{\partial \phi}{\partial x}\right]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x\phi]_i^n .$$

Then discretize

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}}\frac{u_{i+1}^n - u_i^n}{\Delta x} = [qD_x u]_{i+\frac{1}{2}}^n .$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}}\frac{u_i^n - u_{i-1}^n}{\Delta x} = [qD_x u]_{i-\frac{1}{2}}^n .$$

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right) . \tag{2.36}$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_x(\overline{q}^x D_x u)]_i^n . \tag{2.37}$$

⚠ Do not use the chain rule on the spatial derivative term!

Many are tempted to use the chain rule on the term $\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)$, but this is not a good idea when discretizing such a term.
The term with a variable coefficient expresses the net flux qu_x into a small volume (i.e., interval in 1D):

$$\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Delta x}(q(x+\Delta x)u_x(x+\Delta x) - q(x)u_x(x)) .$$

Our discretization reflects this principle directly: qu_x at the right end of the cell minus qu_x at the left end, because this follows from the formula (2.36) or $[D_x(qD_x u)]_i^n$.
When using the chain rule, we get two terms $qu_{xx} + q_x u_x$. The typical discretization is

$$[D_x q D_x u + D_{2x} q D_{2x} u]_i^n, \tag{2.38}$$

Writing this out shows that it is different from $[D_x(qD_x u)]_i^n$ and lacks the physical interpretation of net flux into a cell. With a smooth and slowly varying $q(x)$ the differences between the two discretizations are not substantial. However, when q exhibits (potentially large) jumps, $[D_x(qD_x u)]_i^n$ with harmonic averaging of q yields a better solution than arithmetic averaging or (2.38). In the literature, the discretization $[D_x(qD_x u)]_i^n$ totally dominates and very few mention the alternative in (2.38).

2.44. Computing the coefficient between mesh points

If q is a known function of x, we can easily evaluate $q_{i+\frac{1}{2}}$ simply as $q(x_{i+\frac{1}{2}})$ with $x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$. However, in many cases c, and hence q, is only known as a discrete function, often at the mesh points x_i. Evaluating q between two mesh points x_i and x_{i+1} must then be done by *interpolation* techniques, of which three are of particular interest in this context:

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}\left(q_i + q_{i+1}\right) = [\overline{q}^x]_i \quad \text{(arithmetic mean)} \tag{2.39}$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1} \quad \text{(harmonic mean)} \tag{2.40}$$

$$q_{i+\frac{1}{2}} \approx \left(q_i q_{i+1}\right)^{1/2} \quad \text{(geometric mean)} \tag{2.41}$$

The arithmetic mean is by far the most commonly used averaging technique and is well suited for smooth $q(x)$ functions. The harmonic mean is often preferred when $q(x)$ exhibits large jumps (which is typical for geological media). The geometric mean is less used, but popular in discretizations to linearize quadratic nonlinearities.

With the operator notation for the arithmetic mean we can specify the discretization of the complete variable-coefficient wave equation in a compact way:

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n \,. \tag{2.42}$$

Strictly speaking, $[D_x \overline{q}^x D_x u]_i^n = [D_x(\overline{q}^x D_x u)]_i^n$.

From the compact difference notation we immediately see what kind of differences that each term is approximated with. The notation $\overline{q}^x$ also specifies that the variable coefficient is approximated by an arithmetic mean, the definition being $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i + q_{i+1})/2$.

Before implementing, it remains to solve (2.42) with respect to u_i^{n+1}:

$$\begin{aligned} u_i^{n+1} = &- u_i^{n-1} + 2u_i^n + \\ &\left(\frac{\Delta t}{\Delta x}\right)^2 \left(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\right) + \\ &\Delta t^2 f_i^n \,. \end{aligned} \tag{2.43}$$

2.45. How a variable coefficient affects the stability

The stability criterion derived later (Section Section 2.63) reads $\Delta t \leq \Delta x/c$. If $c = c(x)$, the criterion will depend on the spatial location. We must therefore choose a Δt that is small enough such that no mesh cell has $\Delta t > \Delta x/c(x)$. That is, we must use the largest c value in the criterion:

$$\Delta t \leq \beta \frac{\Delta x}{\max_{x\in[0,L]} c(x)} \,.$$

The parameter β is included as a safety factor: in some problems with a significantly varying c it turns out that one must choose $\beta < 1$ to have stable solutions ($\beta = 0.9$ may act as an all-round value).

A different strategy to handle the stability criterion with variable wave velocity is to use a spatially varying Δt. While the idea is mathematically attractive at first sight, the implementation quickly becomes very complicated, so we stick to a constant Δt and a worst case value of $c(x)$ (with a safety factor β).

2.46. Neumann condition and a variable coefficient

Consider a Neumann condition $\partial u/\partial x = 0$ at $x = L = N_x\Delta x$, discretized as

$$[D_{2x}u]_i^n = \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0 \quad \Rightarrow \quad u_{i+1}^n = u_{i-1}^n,$$

for $i = N_x$. Using the scheme (2.43) at the end point $i = N_x$ with $u_{i+1}^n = u_{i-1}^n$ results in

$$\begin{aligned} u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + \\ &\quad \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \\ &= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \\ &\approx -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_i(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n\,. \end{aligned} \tag{2.44}$$

Here we used the approximation

$$\begin{aligned} q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} &= q_i + \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots + \\ &\quad q_i - \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots \\ &= 2q_i + 2\left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \mathcal{O}(\Delta x^4) \\ &\approx 2q_i \end{aligned} \tag{2.45}$$

.

An alternative derivation may apply the arithmetic mean of $q_{n-\frac{1}{2}}$ and $q_{n+\frac{1}{2}}$ in (2.44), leading to the term

$$(q_i + \frac{1}{2}(q_{i+1} + q_{i-1}))(u_{i-1}^n - u_i^n)\,.$$

Since $\frac{1}{2}(q_{i+1} + q_{i-1}) = q_i + \mathcal{O}(\Delta x^2)$, we can approximate with $2q_i(u_{i-1}^n - u_i^n)$ for $i = N_x$ and get the same term as we did above.

A common technique when implementing $\partial u/\partial x = 0$ boundary conditions, is to assume $dq/dx = 0$ as well. This implies $q_{i+1} = q_{i-1}$ and $q_{i+1/2} = q_{i-1/2}$ for $i = N_x$. The implications for the scheme are

$$
\begin{aligned}
u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + \\
&\quad \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \\
&\quad \Delta t^2 f_i^n \\
&= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \,.
\end{aligned}
\tag{2.46}
$$

2.47. Implementation of variable coefficients

The implementation of the scheme with a variable wave velocity $q(x) = c^2(x)$ may assume that q is available as an array `q[i]` at the spatial mesh points. The following loop is a straightforward implementation of the scheme (2.43):

```
for i in range(1, Nx):
    u[i] = - u_nm1[i] + 2*u_n[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_n[i+1] - u_n[i])  - \
               0.5*(q[i] + q[i-1])*(u_n[i] - u_n[i-1])) + \
           dt2*f(x[i], t[n])
```

The coefficient `C2` is now defined as `(dt/dx)**2`, i.e., *not* as the squared Courant number, since the wave velocity is variable and appears inside the parenthesis.

With Neumann conditions $u_x = 0$ at the boundary, we need to combine this scheme with the discrete version of the boundary condition, as shown in Section Section 2.46. Nevertheless, it would be convenient to reuse the formula for the interior points and just modify the indices `ip1=i+1` and `im1=i-1` as we did in Section Section 2.38. Assuming $dq/dx = 0$ at the boundaries, we can implement the scheme at the boundary with the following code.

```
i = 0
ip1 = i+1
im1 = ip1
u[i] = - u_nm1[i] + 2*u_n[i] + \
       C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
           0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
       dt2*f(x[i], t[n])
```

With ghost cells we can just reuse the formula for the interior points also at the boundary, provided that the ghost values of both u and q are correctly updated to ensure $u_x = 0$ and $q_x = 0$.

A vectorized version of the scheme with a variable coefficient at internal mesh points becomes

```
u[1:-1] = - u_nm1[1:-1] + 2*u_n[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_n[2:] - u_n[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_n[1:-1] - u_n[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

2.48. A more general PDE model with variable coefficients

Sometimes a wave PDE has a variable coefficient in front of the time-derivative term:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{2.47}$$

One example appears when modeling elastic waves in a rod with varying density, cf.~(Section 2.81) with $\varrho(x)$.

A natural scheme for (2.47) is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n\,.$$

We realize that the ϱ coefficient poses no particular difficulty, since ϱ enters the formula just as a simple factor in front of a derivative. There is hence no need for any averaging of ϱ. Often, ϱ will be moved to the right-hand side, also without any difficulty:

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]_i^n\,.$$

Generalization: damping

Waves die out by two mechanisms. In 2D and 3D the energy of the wave spreads out in space, and energy conservation then requires the amplitude to decrease. This effect is not present in 1D. Damping is another cause of amplitude reduction. For example, the vibrations of a string die out because of damping due to air resistance and non-elastic effects in the string.

The simplest way of including damping is to add a first-order derivative to the equation (in the same way as friction forces enter a vibrating mechanical system):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} - f(x,t), \tag{2.48}$$

where $b \geq 0$ is a prescribed damping coefficient.

A typical discretization of (2.48) in terms of centered differences reads

$$[D_t D_t u + bD_{2t} u = c^2 D_x D_x u + f]_i^n\,. \tag{2.49}$$

Writing out the equation and solving for the unknown u_i^{n+1} gives the scheme

$$u_i^{n+1} = (1 + \frac{1}{2}b\Delta t)^{-1}((\frac{1}{2}b\Delta t - 1)u_i^{n-1} + 2u_i^n + C^2\left(u^n * *i + 1 - 2u^n * *i + u_{i-1}^n\right) + \Delta t^2 f_i^n), \tag{2.50}$$

for $i \in \mathcal{I}_x^i$ and $n \geq 1$. New equations must be derived for u_i^1, and for boundary points in case of Neumann conditions.

The damping is very small in many wave phenomena and thus only evident for very long time simulations. This makes the standard wave equation without damping relevant for a lot of applications.

2.49. Building a general 1D wave equation solver

The program wave1D_dn_vc.py is a fairly general code for 1D wave propagation problems that targets the following initial-boundary value problem

$$
\begin{aligned}
u_{tt} &= (c^2(x)u_x)_x + f(x,t), \quad x \in (0,L),\ t \in (0,T] \\
u(x,0) &= I(x), \quad x \in [0,L] \\
u_t(x,0) &= V(t), \quad x \in [0,L] \\
u(0,t) &= U_0(t) \text{ or } u_x(0,t) = 0, \quad t \in (0,T] \\
u(L,t) &= U_L(t) \text{ or } u_x(L,t) = 0, \quad t \in (0,T]
\end{aligned}
\tag{2.51}
$$

The only new feature here is the time-dependent Dirichlet conditions, but they are trivial to implement:

```
i = Ix[0]  # x=0
u[i] = U_0(t[n+1])

i = Ix[-1] # x=L
u[i] = U_L(t[n+1])
```

The `solver` function is a natural extension of the simplest `solver` function in the initial `wave1D_u0.py` program, extended with Neumann boundary conditions ($u_x = 0$), time-varying Dirichlet conditions, as well as a variable wave velocity. The different code segments needed to make these extensions have been shown and commented upon in the preceding text. We refer to the `solver` function in the `wave1D_dn_vc.py` file for all the details. Note in that `solver` function, however, that the technique of "hashing" is used to check whether a certain simulation has been run before, or not. This technique is further explained in Section Section 8.7.

The vectorization is only applied inside the time loop, not for the initial condition or the first time steps, since this initial work is negligible for long time simulations in 1D problems.

The following sections explain various more advanced programming techniques applied in the general 1D wave equation solver.

2.50. User action function as a class

A useful feature in the `wave1D_dn_vc.py` program is the specification of the `user_action` function as a class. This part of the program may need some motivation and explanation. Although the `plot_u_st` function (and the `PlotMatplotlib` class) in the `wave1D_u0.viz` function remembers the local variables in the `viz` function, it is a cleaner solution to store the needed variables together with the function, which is exactly what a class offers.

2.50.1. The code

A class for flexible plotting, cleaning up files, making movie files, like the function `wave1D_u0.viz` did, can be coded as follows:

```
class PlotAndStoreSolution:
    """
    Class for the user_action function in solver.
    Visualizes the solution only.
    """
    def __init__(
        self,
        casename='tmp',    # Prefix in filenames
        umin=-1, umax=1,   # Fixed range of y axis
        pause_between_frames=None,  # Movie speed
        screen_movie=True, # Show movie on screen?
        title='',          # Extra message in title
        skip_frame=1,      # Skip every skip_frame frame
        filename=None):    # Name of file with solutions
        self.casename = casename
        self.yaxis = [umin, umax]
        self.pause = pause_between_frames
        import matplotlib.pyplot as plt
        self.plt = plt
        self.screen_movie = screen_movie
        self.title = title
        self.skip_frame = skip_frame
        self.filename = filename
        if filename is not None:
            self.t = []
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            for filename in filenames:
                os.remove(filename)

        for filename in glob.glob('frame_*.png'):
            os.remove(filename)

    def __call__(self, u, x, t, n):
        """
        Callback function user_action, call by solver:
        Store solution, plot on screen and save to file.
        """
        if self.filename is not None:
            name = 'u%04d' % n  # array name
            kwargs = {name: u}
            fname = '.' + self.filename + '_' + name + '.dat'
            np.savez(fname, **kwargs)
            self.t.append(t[n])  # store corresponding time value
```

```
            if n == 0:           # save x once
                np.savez('.' + self.filename + '_x.dat', x=x)

        if n % self.skip_frame != 0:
            return
        title = 't=%.3f' % t[n]
        if self.title:
            title = self.title + ' ' + title

        if n == 0:
            self.plt.ion()
            self.lines = self.plt.plot(x, u, 'r-')
            self.plt.axis([x[0], x[-1],
                           self.yaxis[0], self.yaxis[1]])
            self.plt.xlabel('x')
            self.plt.ylabel('u')
            self.plt.title(title)
            self.plt.legend(['t=%.3f' % t[n]])
        else:
            self.lines[0].set_ydata(u)
            self.plt.legend(['t=%.3f' % t[n]])
            self.plt.draw()

        if t[n] == 0:
            time.sleep(2)  # let initial condition stay 2 s
        else:
            if self.pause is None:
                pause = 0.2 if u.size < 100 else 0
            time.sleep(pause)

        self.plt.savefig('frame_%04d.png' % (n))
```

2.50.2. Dissection

Understanding this class requires quite some familiarity with Python in general and class programming in particular. The class supports plotting with Matplotlib for visualization.

With the `screen_movie` parameter we can suppress displaying each movie frame on the screen. Alternatively, for slow movies associated with fine meshes, one can set `skip_frame=10`, causing every 10 frames to be shown.

The `__call__` method makes `PlotAndStoreSolution` instances behave like functions, so we can just pass an instance, say `p`, as the `user_action` argument in the `solver` function, and any call to `user_action` will be a call to `p.__call__`. The `__call__` method plots the solution on the screen, saves the plot to file, and stores the solution in a file for later retrieval.

More details on storing the solution in files appear in Section 8.4.

2.51. Pulse propagation in two media

The function `pulse` in `wave1D_dn_vc.py` demonstrates wave motion in heterogeneous media where c varies. One can specify an interval where the wave velocity is decreased by a factor `slowness_factor` (or increased by making this factor less than one). Figure Figure 2.5 shows a typical simulation scenario.

Four types of initial conditions are available:

1. a rectangular pulse (`plug`),
2. a Gaussian function (`gaussian`),
3. a "cosine hat" consisting of one period of the cosine function (`cosinehat`),
4. half a period of a "cosine hat" (`half-cosinehat`)

These peak-shaped initial conditions can be placed in the middle (`loc='center'`) or at the left end (`loc='left'`) of the domain. With the pulse in the middle, it splits in two parts, each with half the initial amplitude, traveling in opposite directions. With the pulse at the left end, centered at $x = 0$, and using the symmetry condition $\partial u/\partial x = 0$, only a right-going pulse is generated. There is also a left-going pulse, but it travels from $x = 0$ in negative x direction and is not visible in the domain $[0, L]$.

The `pulse` function is a flexible tool for playing around with various wave shapes and jumps in the wave velocity (i.e., discontinuous media). The code is shown to demonstrate how easy it is to reach this flexibility with the building blocks we have already developed:

```
def pulse(
    C=1,            # Maximum Courant number
    Nx=200,         # spatial resolution
    animate=True,
    version='vectorized',
    T=2,            # end time
    loc='left',     # location of initial condition
    pulse_tp='gaussian',  # pulse/init.cond. type
    slowness_factor=2, # inverse of wave vel. in right medium
    medium=[0.7, 0.9], # interval for right medium
    skip_frame=1,      # skip frames in animations
    sigma=0.05         # width measure of the pulse
    ):
    """
    Various peaked-shaped initial conditions on [0,1].
    Wave velocity is decreased by the slowness_factor inside
    medium. The loc parameter can be 'center' or 'left',
    depending on where the initial pulse is to be located.
    The sigma parameter governs the width of the pulse.
    """
    L = 1.0
    c_0 = 1.0
    if loc == 'center':
        xc = L/2
```

```
    elif loc == 'left':
        xc = 0

    if pulse_tp in ('gaussian','Gaussian'):
        def I(x):
            return np.exp(-0.5*((x-xc)/sigma)**2)
    elif pulse_tp == 'plug':
        def I(x):
            return 0 if abs(x-xc) > sigma else 1
    elif pulse_tp == 'cosinehat':
        def I(x):
            w = 2
            a = w*sigma
            return 0.5*(1 + np.cos(np.pi*(x-xc)/a)) \
                   if xc - a <= x <= xc + a else 0

    elif pulse_tp == 'half-cosinehat':
        def I(x):
            w = 4
            a = w*sigma
            return np.cos(np.pi*(x-xc)/a) \
                   if xc - 0.5*a <= x <= xc + 0.5*a else 0
    else:
        raise ValueError('Wrong pulse_tp="%s"' % pulse_tp)

    def c(x):
        return c_0/slowness_factor \
               if medium[0] <= x <= medium[1] else c_0

    umin=-0.5; umax=1.5*I(xc)
    casename = '%s_Nx%s_sf%s' % \
               (pulse_tp, Nx, slowness_factor)
    action = PlotMediumAndSolution(
        medium, casename=casename, umin=umin, umax=umax,
        skip_frame=skip_frame, screen_movie=animate,
        backend=None, filename='tmpdata')

    dt = (L/Nx)/c_0
    cpu, hashed_input = solver(
        I=I, V=None, f=None, c=c,
        U_0=None, U_L=None,
        L=L, dt=dt, C=C, T=T,
        user_action=action,
        version=version,
        stability_safety_factor=1)

    if cpu > 0:  # did we generate new data?
```

```
        action.close_file(hashed_input)
        action.make_movie_file()
    print('cpu (-1 means no new data generated):', cpu)

def convergence_rates(
    u_exact,
    I, V, f, c, U_0, U_L, L,
    dt0, num_meshes,
    C, T, version='scalar',
    stability_safety_factor=1.0):
    """
    Half the time step and estimate convergence rates for
    for num_meshes simulations.
    """
    class ComputeError:
        def __init__(self, norm_type):
            self.error = 0

        def __call__(self, u, x, t, n):
            """Store norm of the error in self.E."""
            error = np.abs(u - u_exact(x, t[n])).max()
            self.error = max(self.error, error)

    E = []
    h = []  # dt, solver adjusts dx such that C=dt*c/dx
    dt = dt0
    for i in range(num_meshes):
        error_calculator = ComputeError('Linf')
        solver(I, V, f, c, U_0, U_L, L, dt, C, T,
               user_action=error_calculator,
               version='scalar',
               stability_safety_factor=1.0)
        E.append(error_calculator.error)
        h.append(dt)
        dt /= 2  # halve the time step for next simulation
    print('E:', E)
    print('h:', h)
    r = [np.log(E[i]/E[i-1])/np.log(h[i]/h[i-1])
         for i in range(1,num_meshes)]
    return r

def test_convrate_sincos():
    n = m = 2
    L = 1.0
    u_exact = lambda x, t: np.cos(m*np.pi/L*t)*np.sin(m*np.pi/L*x)

    r = convergence_rates(
```

```
        u_exact=u_exact,
        I=lambda x: u_exact(x, 0),
        V=lambda x: 0,
        f=0,
        c=1,
        U_0=0,
        U_L=0,
        L=L,
        dt0=0.1,
        num_meshes=6,
        C=0.9,
        T=1,
        version='scalar',
        stability_safety_factor=1.0)
    print('rates sin(x)*cos(t) solution:',
          [round(r_,2) for r_ in r])
    assert abs(r[-1] - 2) < 0.002
```

The `PlotMediumAndSolution` class used here is a subclass of `PlotAndStoreSolution` where the medium with reduced c value, as specified by the `medium` interval, is visualized in the plots.

Comment on the choices of discretization parameters

The argument N_x in the `pulse` function does not correspond to the actual spatial resolution of $C < 1$, since the `solver` function takes a fixed Δt and C, and adjusts Δx accordingly. As seen in the `pulse` function, the specified Δt is chosen according to the limit $C = 1$, so if $C < 1$, Δt remains the same, but the `solver` function operates with a larger Δx and smaller N_x than was specified in the call to `pulse`. The practical reason is that we always want to keep Δt fixed such that plot frames and movies are synchronized in time regardless of the value of C (i.e., Δx is varied when the Courant number varies).

The reader is encouraged to play around with the `pulse` function:

```
>>> import wave1D_dn_vc as w
>>> w.pulse(Nx=50, loc='left', pulse_tp='cosinehat', slowness_factor=2)
```

To easily kill the graphics by Ctrl-C and restart a new simulation it might be easier to run the above two statements from the command line with

```
Terminal> python -c 'import wave1D_dn_vc as w; w.pulse(...)'
```

2.52. Exercise: Find the analytical solution to a damped wave equation

Consider the wave equation with damping (2.48). The goal is to find an exact solution to a wave problem with damping and zero source term. A starting point is the standing wave solution from

Exercise Section 2.29. It becomes necessary to include a damping term $e^{-\beta t}$ and also have both a sine and cosine component in time:

$$u_{\mathrm{e}}(x,t) = e^{-\beta t}\sin kx\,(A\cos\omega t - B\sin\omega t)\ .$$

Find k from the boundary conditions $u(0,t) = u(L,t) = 0$. Then use the PDE to find constraints on β, ω, A, and B. Set up a complete initial-boundary value problem and its solution.

Solution

Mathematical model:

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2},$$

$b \geq 0$ is a prescribed damping coefficient.
Ansatz:

$$u(x,t) = e^{-\beta t}\sin kx\,(A\cos\omega t - B\sin\omega t)$$

Boundary condition: $u = 0$ for $x = 0, L$. Fulfilled for $x = 0$. Requirement at $x = L$ gives

$$kL = m\pi,$$

for an arbitrary integer m. Hence, $k = m\pi/L$.
Inserting the ansatz in the PDE and dividing by $e^{-\beta t}$ results in

$$\begin{aligned}(\beta^2 sinkx - \omega^2 sinkx - b\beta sinkx)(A\cos\omega t + B\sin\omega t) + \\ (b\omega sinkx - 2\beta\omega sinkx)(-A\sin\omega t + B\cos\omega t) = -(A\cos\omega t + B\sin\omega t)k^2c^2\end{aligned}$$

This gives us two requirements:

$$\beta^2 - \omega^2 + b\beta + k^2c^2 = 0$$

and

$$-2\beta\omega + b\omega = 0$$

Since b, c and k are to be given in advance, we may solve these two equations to get

$$\beta = \frac{b}{2}$$
$$\omega = \sqrt{c^2k^2 - \frac{b^2}{4}}$$

From the initial condition on the derivative, i.e. $\frac{\partial u_e}{\partial t} = 0$, we find that

$$B\omega = \beta A$$

Inserting the expression for ω, we find that

$$B = \frac{b}{2\sqrt{c^2k^2 - \frac{b^2}{4}}}A$$

for A prescribed.

Using $t = 0$ in the expression for u_e gives us the initial condition as

$$I(x) = Asinkx$$

Summarizing, the PDE problem can then be states as

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} &= c^2\frac{\partial^2 u}{\partial x^2}, && x \in (0, L),\ t \in (0, T] \\
u(x, 0) &= I(x), && x \in [0, L] \\
\frac{\partial}{\partial t}u(x, 0) &= 0, && x \in [0, L] \\
u(0, t) &= 0, && t \in (0, T] \\
u(L, t) &= 0, && t \in (0, T]
\end{aligned}$$

where constants c, A, b and k, as well as $I(x)$, are prescribed.
The solution to the problem is then given as

$$u_{\mathrm{e}}(x, t) = e^{-\beta t}\sin kx\left(A\cos\omega t - B\sin\omega t\right).$$

with $k = m\pi/L$ for arbitrary integer m, $\beta = \frac{b}{2}$, $\omega = \sqrt{c^2k^2 - \frac{b^2}{4}}$, $B = \frac{b}{2\sqrt{c^2k^2 - \frac{b^2}{4}}}A$ and $I(x) = Asinkx$.

2.53. Problem: Explore symmetry boundary conditions

Consider the simple "plug" wave where $\Omega = [-L, L]$ and

$$I(x) = \begin{cases} 1, & x \in [-\delta, \delta], \\ 0, & \text{otherwise} \end{cases}$$

for some number $0 < \delta < L$. The other initial condition is $u_t(x, 0) = 0$ and there is no source term f. The boundary conditions can be set to $u = 0$. The solution to this problem is symmetric around $x = 0$. This means that we can simulate the wave process in only half of the domain $[0, L]$.

a)

Argue why the symmetry boundary condition is $u_x = 0$ at $x = 0$.

Symmetry of a function about $x = x_0$ means that

$f(x_0 + h) = f(x_0 - h)$.

Solution

A symmetric u around $x = 0$ means that $u(-x, t) = u(x, t)$. Let $x_0 = 0$ and $x = x_0 + h$. Then

we can use a *centered* finite difference definition of the derivative:

$$\frac{\partial}{\partial x}u(x_0,t) = \lim_{h\to 0}\frac{u(x_0+h,t)-u(x_0-h)}{2h} = \lim_{h\to 0}\frac{u(h,t)-u(-h,t)}{2h} = 0,$$

since $u(h,t) = u(-h,t)$ for any h. Symmetry around a point $x = x_0$ therefore always implies $u_x(x_0,t) = 0$.

b)

Perform simulations of the complete wave problem on $[-L,L]$. Thereafter, utilize the symmetry of the solution and run a simulation in half of the domain $[0,L]$, using a boundary condition at $x = 0$. Compare plots from the two solutions and confirm that they are the same.

Solution

We can utilize the `wave1D_dn.py` code which allows Dirichlet and Neumann conditions. The `solver` and `viz` functions must take x_0 and x_L as parameters instead of just L such that we can solve the wave equation in $[x_0,x_L]$. The we can call up `solver` for the two problems on $[-L,L]$ and $[0,L]$ with boundary conditions $u(-L,t) = u(L,t) = 0$ and $u_x(0,t) = u(L,t) = 0$, respectively.
The original `wave1D_dn.py` code makes a movie by playing all the `.png` files in a browser. It can then be wise to let the `viz` function create a movie directory and place all the frames and HTML player file in that directory. Alternatively, one can just make some ordinary movie file (Ogg, WebM, MP4, Flash) with `avconv` or `ffmpeg` and give it a name. It is a point that the name is transferred to `viz` so it is easy to call `viz` twice and get two separate movie files or movie directories.
The plots produced by the code (below) shows that the solutions indeed are the same.

c)

Prove the symmetry property of the solution by setting up the complete initial-boundary value problem and showing that if $u(x,t)$ is a solution, then also $u(-x,t)$ is a solution.

Solution

The plan in this proof is to introduce $v(x,t) = u(-x,t)$ and show that v fulfills the same initial-boundary value problem as u. If the problem has a unique solution, then $v = u$. Or, in other words, the solution is symmetric: $u(-x,t) = u(x,t)$.
We can work with a general initial-boundary value problem on the form

$$u_tt(x,t) = c^2u_{xx}(x,t) + f(x,t) \tag{2.52}$$
$$u(x,0) = I(x) \tag{2.53}$$
$$u_t(x,0) = V(x) \tag{2.54}$$
$$u(-L,0) = 0 \tag{2.55}$$
$$u(L,0) = 0 \tag{2.56}$$

Introduce a new coordinate $\bar{x} = -x$. We have that

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \bar{x}}\frac{\partial \bar{x}}{\partial x}\right) = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \bar{x}}(-1)\right) = (-1)^2\frac{\partial^2 u}{\partial \bar{x}^2}$$

The derivatives in time are unchanged.
Substituting x by $-\bar{x}$ leads to

$$u_{tt}(-\bar{x},t) = c^2 u_{\bar{x}\bar{x}}(-\bar{x},t) + f(-\bar{x},t) \tag{2.57}$$
$$u(-\bar{x},0) = I(-\bar{x}) \tag{2.58}$$
$$u_t(-\bar{x},0) = V(-\bar{x}) \tag{2.59}$$
$$u(L,0) = 0 \tag{2.60}$$
$$u(-L,0) = 0 \tag{2.61}$$

Now, dropping the bars and introducing $v(x,t) = u(-x,t)$, we find that

$$v_{tt}(x,t) = c^2 v_{xx}(x,t) + f(-x,t) \tag{2.62}$$
$$v(x,0) = I(-x) \tag{2.63}$$
$$v_t(x,0) = V(-x) \tag{2.64}$$
$$v(-L,0) = 0 \tag{2.65}$$
$$v(L,0) = 0 \tag{2.66}$$

Provided that I, f, and V are all symmetric around $x = 0$ such that $I(x) = I(-x)$, $V(x) = V(-x)$, and $f(x,t) = f(-x,t)$, we can express the initial-boundary value problem as

$$v_{tt}(x,t) = c^2 v_{xx}(x,t) + f(x,t) \tag{2.67}$$
$$v(x,0) = I(x) \tag{2.68}$$
$$v_t(x,0) = V(x) \tag{2.69}$$
$$v(-L,0) = 0 \tag{2.70}$$
$$v(L,0) = 0 \tag{2.71}$$

This is the same problem as the one that u fulfills. If the solution is unique, which can be proven, then $v = u$, and $u(-x,t) = u(x,t)$.
To summarize, the necessary conditions for symmetry are that

- all involved functions I, V, and f must be symmetric, and
- the boundary conditions are symmetric in the sense that they can be flipped (the condition at $x = -L$ can be applied at $x = L$ and vice versa).

d)

If the code works correctly, the solution $u(x,t) = x(L-x)(1+\frac{t}{2})$ should be reproduced exactly. Write a test function `test_quadratic` that checks whether this is the case. Simulate for x in $[0, \frac{L}{2}]$ with a symmetry condition at the end $x = \frac{L}{2}$.

Solution

Running the code below, shows that the test case indeed is reproduced exactly.

```
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    We simulate in [0, L/2] and apply a symmetry condition
    at the end x=L/2.
    """
    exact_solution = lambda x, t: x * (L - x) * (1 + 0.5 * t)
    I = lambda x: exact_solution(x, 0)
    V = lambda x: 0.5 * exact_solution(x, 0)
    f = lambda x, t: 2 * (1 + 0.5 * t) * c**2
    U_0 = lambda t: exact_solution(0, t)
    U_L = None
    L = 2.5
    c = 1.5
    Nx = 3  # very coarse mesh
    C = 1
    T = 18  # long time integration

    def assert_no_error(u, x, t, n):
        u_e = exact_solution(x, t[n])
        diff = abs(u - u_e).max()
        assert diff < 1e-13, f"Max error: {diff}"

    solver(
        I, V, f, c, U_0, U_L, 0, L / 2, Nx, C, T,
        user_action=assert_no_error, version="scalar",
    )
    solver(
        I, V, f, c, U_0, U_L, 0, L / 2, Nx, C, T,
        user_action=assert_no_error, version="vectorized",
    )
```

2.54. Exercise: Send pulse waves through a layered medium

Use the `pulse` function in `wave1D_dn_vc.py` to investigate sending a pulse, located with its peak at $x = 0$, through two media with different wave velocities. The (scaled) velocity in the left medium is 1 while it is $\frac{1}{s_f}$ in the right medium. Report what happens with a Gaussian pulse, a "cosine hat" pulse, half a "cosine hat" pulse, and a plug pulse for resolutions $N_x = 40, 80, 160$, and $s_f = 2, 4$. Simulate until $T = 2$.

Solution

In all cases, the change in velocity causes some of the wave to be reflected back (while the rest is let through). When the waves go from higher to lower velocity, the amplitude builds, and vice versa.

```
import os
import sys

path = os.path.join(
    os.pardir, os.pardir, os.pardir, os.pardir, "wave", "src-wave", "wave1D"
)
sys.path.insert(0, path)
from wave1D_dn_vc import pulse

pulse_tp = sys.argv[1]
C = float(sys.argv[2])
pulse(pulse_tp=pulse_tp, C=C, Nx=100, animate=False, slowness_factor=4)
```

2.55. Exercise: Explain why numerical noise occurs

The experiments performed in Exercise Section 2.54 shows considerable numerical noise in the form of non-physical waves, especially for $s_f = 4$ and the plug pulse or the half a "cosinehat" pulse. The noise is much less visible for a Gaussian pulse. Run the case with the plug and half a "cosinehat" pulse for $s_f = 1$, $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion relation to explain the observations.

2.56. Exercise: Investigate harmonic averaging in a 1D model

Harmonic means are often used if the wave velocity is non-smooth or discontinuous. Will harmonic averaging of the wave velocity give less numerical noise for the case $s_f = 4$ in Exercise Section 2.54?

2.57. Problem: Implement open boundary conditions

To enable a wave to leave the computational domain and travel undisturbed through the boundary $x = L$, one can in a one-dimensional problem impose the following condition, called a *radiation condition* or *open boundary condition*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0\,. \tag{2.72}$$

The parameter c is the wave velocity.

Show that (2.72) accepts a solution $u = g_R(x - ct)$ (right-going wave), but not $u = g_L(x + ct)$ (left-going wave). This means that (2.72) will allow any right-going wave $g_R(x - ct)$ to pass through the boundary undisturbed.

A corresponding open boundary condition for a left-going wave through $x = 0$ is

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0\,. \tag{2.73}$$

a)

A natural idea for discretizing the condition (2.72) at the spatial end point $i = N_x$ is to apply centered differences in time and space:

$$[D_{2t}u + cD_{2x}u = 0]_i^n, \quad i = N_x\,. \tag{2.74}$$

Eliminate the fictitious value $u^n_{N_x+1}$ by using the discrete equation at the same point.

The equation for the first step, u_i^1, is in principle also affected, but we can then use the condition $u_{N_x} = 0$ since the wave has not yet reached the right boundary.

b)

A much more convenient implementation of the open boundary condition at $x = L$ can be based on an explicit discretization

$$[D_t^+u + cD_x^-u = 0]_i^n, \quad i = N_x\,. \tag{2.75}$$

From this equation, one can solve for $u^{n+1}_{N_x}$ and apply the formula as a Dirichlet condition at the boundary point. However, the finite difference approximations involved are of first order.

Implement this scheme for a wave equation $u_{tt} = c^2u_{xx}$ in a domain $[0, L]$, where you have $u_x = 0$ at $x = 0$, the condition (2.72) at $x = L$, and an initial disturbance in the middle of the domain, e.g., a plug profile like

$$u(x, 0) = \begin{cases} 1, & L/2 - \ell \leq x \leq L/2 + \ell, \\ 0, \text{otherwise} \end{cases}$$

Observe that the initial wave is split in two, the left-going wave is reflected at $x = 0$, and both waves travel out of $x = L$, leaving the solution as $u = 0$ in $[0, L]$. Use a unit Courant number such that the numerical solution is exact. Make a movie to illustrate what happens.

Because this simplified implementation of the open boundary condition works, there is no need to pursue the more complicated discretization in a).

Modify the solver function in

wave1D_dn.py.

c)

Add the possibility to have either $u_x = 0$ or an open boundary condition at the left boundary. The latter condition is discretized as

$$[D_t^+u - cD_x^+u = 0]_i^n, \quad i = 0, \tag{2.76}$$

leading to an explicit update of the boundary value u_0^{n+1}.

The implementation can be tested with a Gaussian function as initial condition:

$$g(x; m, s) = \frac{1}{\sqrt{2\pi}s} e^{-\frac{(x-m)^2}{2s^2}} .$$

Run two tests:

1. Disturbance in the middle of the domain, $I(x) = g(x; L/2, s)$, and open boundary condition at the left end.
2. Disturbance at the left end, $I(x) = g(x; 0, s)$, and $u_x = 0$ as symmetry boundary condition at this end.

Make test functions for both cases, testing that the solution is zero after the waves have left the domain.

d)

In 2D and 3D it is difficult to compute the correct wave velocity normal to the boundary, which is needed in generalizations of the open boundary conditions in higher dimensions. Test the effect of having a slightly wrong wave velocity in (2.75). Make movies to illustrate what happens.

Remarks

The condition (2.72) works perfectly in 1D when c is known. In 2D and 3D, however, the condition reads $u_t + c_x u_x + c_y u_y = 0$, where c_x and c_y are the wave speeds in the x and y directions. Estimating these components (i.e., the direction of the wave) is often challenging. Other methods are normally used in 2D and 3D to let waves move out of a computational domain.

2.58. Exercise: Implement periodic boundary conditions

It is frequently of interest to follow wave motion over large distances and long times. A straightforward approach is to work with a very large domain, but that might lead to a lot of computations in areas of the domain where the waves cannot be noticed. A more efficient approach is to let a right-going wave out of the domain and at the same time let it enter the domain on the left. This is called a *periodic boundary condition*.

The boundary condition at the right end $x = L$ is an open boundary condition (see Exercise Section 2.57) to let a right-going wave out of the domain. At the left end, $x = 0$, we apply, in the beginning of the simulation, either a symmetry boundary condition (see Exercise Section 2.53) $u_x = 0$, or an open boundary condition.

This initial wave will split in two and either be reflected or transported out of the domain at $x = 0$. The purpose of the exercise is to follow the right-going wave. We can do that with a *periodic boundary condition*. This means that when the right-going wave hits the boundary $x = L$, the open boundary condition lets the wave out of the domain, but at the same time we use a boundary condition on the left end $x = 0$ that feeds the outgoing wave into the domain again. This periodic condition is simply $u(0) = u(L)$. The switch from $u_x = 0$ or an open boundary condition at the left end to a periodic condition can happen when $u(L, t) > \epsilon$, where $\epsilon = 10^{-4}$ might be an appropriate value for determining when the right-going wave hits the boundary $x = L$.

The open boundary conditions can conveniently be discretized as explained in Exercise Section 2.57. Implement the described type of boundary conditions and test them on two different initial shapes: a plug $u(x,0) = 1$ for $x \leq 0.1$, $u(x,0) = 0$ for $x > 0.1$, and a Gaussian function in the middle of the domain: $u(x,0) = \exp\left(-\frac{1}{2}(x-0.5)^2/0.05\right)$. The domain is the unit interval $[0,1]$. Run these two shapes for Courant numbers 1 and 0.5. Assume constant wave velocity. Make movies of the four cases. Reason why the solutions are correct.

2.59. Exercise: Compare discretizations of a Neumann condition

We have a 1D wave equation with variable wave velocity: $u_{tt} = (qu_x)_x$. A Neumann condition u_x at $x = 0, L$ can be discretized as shown in (2.44) and (2.46).

The aim of this exercise is to examine the rate of the numerical error when using different ways of discretizing the Neumann condition.

a)

As a test problem, $q = 1 + (x - L/2)^4$ can be used, with $f(x,t)$ adapted such that the solution has a simple form, say $u(x,t) = \cos(\pi x/L)\cos(\omega t)$ for, e.g., $\omega = 1$. Perform numerical experiments and find the convergence rate of the error using the approximation (2.44).

b)

Switch to $q(x) = 1 + \cos(\pi x/L)$, which is symmetric at $x = 0, L$, and check the convergence rate of the scheme (2.46). Now, $q_{i-1/2}$ is a 2nd-order approximation to q_i, $q_{i-1/2} = q_i + 0.25q_i''\Delta x^2 + \cdots$, because $q_i' = 0$ for $i = N_x$ (a similar argument can be applied to the case $i = 0$).

c)

A third discretization can be based on a simple and convenient, but less accurate, one-sided difference: $u_i - u_{i-1} = 0$ at $i = N_x$ and $u_{i+1} - u_i = 0$ at $i = 0$. Derive the resulting scheme in detail and implement it. Run experiments with q from a) or b) to establish the rate of convergence of the scheme.

d)

A fourth technique is to view the scheme as

$$[D_tD_tu]_i^n = \frac{1}{\Delta x}\left([qD_xu]_{i+\frac{1}{2}}^n - [qD_xu]_{i-\frac{1}{2}}^n\right) - [f]_i^n,$$

and place the boundary at $x_{i+\frac{1}{2}}$, $i = N_x$, instead of exactly at the physical boundary. With this idea of approximating (moving) the boundary, we can just set $[qD_xu]_{i+\frac{1}{2}}^n = 0$. Derive the complete scheme using this technique. The implementation of the boundary condition at $L - \Delta x/2$ is $\mathcal{O}(\Delta x^2)$ accurate, but the interesting question is what impact the movement of the boundary has on the convergence rate. Compute the errors as usual over the entire mesh and use q from a) or b).

2.60. Exercise: Verification by a cubic polynomial in space

The purpose of this exercise is to verify the implementation of the `solver` function in the program `wave1D_n0.py` by using an exact numerical solution for the wave equation $u_{tt} = c^2 u_{xx} + f$ with Neumann boundary conditions $u_x(0,t) = u_x(L,t) = 0$.

A similar verification is used in the file `wave1D_u0.py`, which solves the same PDE, but with Dirichlet boundary conditions $u(0,t) = u(L,t) = 0$. The idea of the verification test in function `test_quadratic` in `wave1D_u0.py` is to produce a solution that is a lower-order polynomial such that both the PDE problem, the boundary conditions, and all the discrete equations are exactly fulfilled. Then the `solver` function should reproduce this exact solution to machine precision. More precisely, we seek $u = X(x)T(t)$, with $T(t)$ as a linear function and $X(x)$ as a parabola that fulfills the boundary conditions. Inserting this u in the PDE determines f. It turns out that u also fulfills the discrete equations, because the truncation error of the discretized PDE has derivatives in x and t of order four and higher. These derivatives all vanish for a quadratic $X(x)$ and linear $T(t)$.

It would be attractive to use a similar approach in the case of Neumann conditions. We set $u = X(x)T(t)$ and seek lower-order polynomials X and T. To force u_x to vanish at the boundary, we let X_x be a parabola. Then X is a cubic polynomial. The fourth-order derivative of a cubic polynomial vanishes, so $u = X(x)T(t)$ will fulfill the discretized PDE also in this case, if f is adjusted such that u fulfills the PDE.

However, the discrete boundary condition is not exactly fulfilled by this choice of u. The reason is that

$$[D_{2x}u]_i^n = u_x(x_i,t_n) + \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2 - \mathcal{O}(\Delta x^4)\,. \tag{2.77}$$

At the two boundary points, we must demand that the derivative $X_x(x) = 0$ such that $u_x = 0$. However, u_{xxx} is a constant and not zero when $X(x)$ is a cubic polynomial. Therefore, our $u = X(x)T(t)$ fulfills

$$[D_{2x}u]_i^n = \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2,$$

and not

$$[D_{2x}u]_i^n = 0, \quad i = 0, N_x,$$

as it should. (Note that all the higher-order terms $\mathcal{O}(\Delta x^4)$ also have higher-order derivatives that vanish for a cubic polynomial.) So to summarize, the fundamental problem is that u as a product of a cubic polynomial and a linear or quadratic polynomial in time is not an exact solution of the discrete boundary conditions.

To make progress, we assume that $u = X(x)T(t)$, where T for simplicity is taken as a prescribed linear function $1 + \frac{1}{2}t$, and $X(x)$ is taken as an *unknown* cubic polynomial $\sum_{j=0}^{3} a_j x^j$. There are two different ways of determining the coefficients $a_0, \ldots, a_3$ such that both the discretized PDE and the discretized boundary conditions are fulfilled, under the constraint that we can specify a function $f(x,t)$ for the PDE to feed to the `solver` function in `wave1D_n0.py`. Both approaches are explained in the subexercises.

a)

One can insert u in the discretized PDE and find the corresponding f. Then one can insert u in the discretized boundary conditions. This yields two equations for the four coefficients $a_0, \ldots, a_3$. To

find the coefficients, one can set $a_0 = 0$ and $a_1 = 1$ for simplicity and then determine a_2 and a_3. This approach will make a_2 and a_3 depend on Δx and f will depend on both Δx and Δt.

Use `sympy` to perform analytical computations. A starting point is to define u as follows:

```
def test_cubic1():
    import sympy as sm
    x, t, c, L, dx, dt = sm.symbols('x t c L dx dt')
    i, n = sm.symbols('i n', integer=True)

    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    a = sm.symbols('a_0 a_1 a_2 a_3')
    X = lambda x: sum(a[q]*x**q for q in range(4))  # Spatial term
    u = lambda x, t: X(x)*T(t)
```

The symbolic expression for u is reached by calling `u(x,t)` with `x` and `t` as `sympy` symbols.

Define `DxDx(u, i, n)`, `DtDt(u, i, n)`, and `D2x(u, i, n)` as Python functions for returning the difference approximations $[D_xD_xu]_i^n$, $[D_tD_tu]_i^n$, and $[D_{2x}u]_i^n$. The next step is to set up the residuals for the equations $[D_{2x}u]_0^n = 0$ and $[D_{2x}u]_{N_x}^n = 0$, where $N_x = L/\Delta x$. Call the residuals `R_0` and `R_L`. Substitute a_0 and a_1 by 0 and 1, respectively, in `R_0`, `R_L`, and `a`:

```
R_0 = R_0.subs(a[0], 0).subs(a[1], 1)
R_L = R_L.subs(a[0], 0).subs(a[1], 1)
a = list(a)  # enable in-place assignment
a[0:2] = 0, 1
```

Determining a_2 and a_3 from the discretized boundary conditions is then about solving two equations with respect to a_2 and a_3, i.e., `a[2:]`:

```
s = sm.solve([R_0, R_L], a[2:])
a[2:] = s[a[2]], s[a[3]]
```

Now, `a` contains computed values and `u` will automatically use these new values since `X` accesses `a`.

Compute the source term f from the discretized PDE: $f_i^n = [D_tD_tu - c^2D_xD_xu]_i^n$. Turn u, the time derivative u_t (needed for the initial condition $V(x)$), and f into Python functions. Set numerical values for L, N_x, C, and c. Prescribe the time interval as $\Delta t = CL/(N_xc)$, which imply $\Delta x = c\Delta t/C = L/N_x$. Define new functions `I(x)`, `V(x)`, and `f(x,t)` as wrappers of the ones made above, where fixed values of L, c, Δx, and Δt are inserted, such that `I`, `V`, and `f` can be passed on to the `solver` function. Finally, call `solver` with a `user_action` function that compares the numerical solution to this exact solution u of the discrete PDE problem.

> To turn a `sympy` expression `e`, depending on a series of
>
> symbols, say `x`, `t`, `dx`, `dt`, `L`, and `c`, into a plain Python function `e_exact(x,t,L,dx,dt,c)`, one can write

```
e_exact = sm.lambdify([x,t,L,dx,dt,c], e, 'numpy')
```

The `'numpy'` argument is a good habit as the `e_exact` function will then work with array arguments if it contains mathematical functions (but here we only do plain arithmetics, which automatically work with arrays).

b)

An alternative way of determining $a_0, \ldots, a_3$ is to reason as follows. We first construct $X(x)$ such that the boundary conditions are fulfilled: $X = x(L-x)$. However, to compensate for the fact that this choice of X does not fulfill the discrete boundary condition, we seek u such that

$$u_x = \frac{\partial}{\partial x} x(L-x)T(t) - \frac{1}{6}u_{xxx}\Delta x^2,$$

since this u will fit the discrete boundary condition. Assuming $u = T(t)\sum_{j=0}^{3} a_j x^j$, we can use the above equation to determine the coefficients a_1, a_2, a_3. A value, e.g., 1 can be used for a_0. The following `sympy` code computes this u:

```
def test_cubic2():
    import sympy as sm
    x, t, c, L, dx = sm.symbols('x t c L dx')
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    X = lambda x: sum(a[i]*x**i for i in range(4))
    a = sm.symbols('a_0 a_1 a_2 a_3')
    u = lambda x, t: X(x)*T(t)
    R = sm.diff(u(x,t), x) - (
        x*(L-x) - sm.Rational(1,6)*sm.diff(u(x,t), x, x, x)*dx**2)
    R = sm.poly(R, x)
    coeff = R.all_coeffs()
    s = sm.solve(coeff, a[1:])  # a[0] is not present in R
    s[a[0]] = 1
    X = lambda x: sm.simplify(sum(s[a[i]]*x**i for i in range(4)))
    u = lambda x, t: X(x)*T(t)
    print 'u:', u(x,t)
```

The next step is to find the source term `f_e` by inserting `u_e` in the PDE. Thereafter, turn `u`, `f`, and the time derivative of `u` into plain Python functions as in a), and then wrap these functions in new functions `I`, `V`, and `f`, with the right signature as required by the `solver` function. Set parameters as in a) and check that the solution is exact to machine precision at each time level using an appropriate `user_action` function.

2.61. Analysis of the wave equation

2.61.1. Properties of the solution

The wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

has solutions of the form

$$u(x,t) = g_R(x - ct) + g_L(x + ct), \tag{2.78}$$

for any functions g_R and g_L sufficiently smooth to be differentiated twice. The result follows from inserting (2.78) in the wave equation. A function of the form $g_R(x - ct)$ represents a signal moving to the right in time with constant velocity c. This feature can be explained as follows. At time $t = 0$ the signal looks like $g_R(x)$. Introducing a moving horizontal coordinate $\xi = x - ct$, we see the function $g_R(\xi)$ is "at rest" in the ξ coordinate system, and the shape is always the same. Say the $g_R(\xi)$ function has a peak at $\xi = 0$. This peak is located at $x = ct$, which means that it moves with the velocity $dx/dt = c$ in the x coordinate system. Similarly, $g_L(x + ct)$ is a function, initially with shape $g_L(x)$, that moves in the negative x direction with constant velocity c (introduce $\xi = x + ct$, look at the point $\xi = 0$, $x = -ct$, which has velocity $dx/dt = -c$).

With the particular initial conditions

$$u(x,0) = I(x), \quad \frac{\partial}{\partial t} u(x,0) = 0,$$

we get, with u as in (2.78),

$$g_R(x) + g_L(x) = I(x), \quad -cg_R'(x) + cg_L'(x) = 0\,.$$

The former suggests $g_R = g_L$, and the former then leads to $g_R = g_L = I/2$. Consequently,

$$u(x,t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct)\,. \tag{2.79}$$

The interpretation of (2.79) is that the initial shape of u is split into two parts, each with the same shape as I but half of the initial amplitude. One part is traveling to the left and the other one to the right.

The solution has two important physical features: constant amplitude of the left and right wave, and constant velocity of these two waves. It turns out that the numerical solution will also preserve the constant amplitude, but the velocity depends on the mesh parameters Δt and Δx.

The solution (2.79) will be influenced by boundary conditions when the parts $\frac{1}{2} I(x - ct)$ and $\frac{1}{2} I(x + ct)$ hit the boundaries and get, e.g., reflected back into the domain. However, when $I(x)$ is nonzero only in a small part in the middle of the spatial domain $[0, L]$, which means that the boundaries are placed far away from the initial disturbance of u, the solution (2.79) is very clearly observed in a simulation.

A useful representation of solutions of wave equations is a linear combination of sine and/or cosine waves. Such a sum of waves is a solution if the governing PDE is linear and each sine or cosine wave fulfills the equation. To ease analytical calculations by hand we shall work with complex exponential functions instead of real-valued sine or cosine functions. The real part of complex expressions will

typically be taken as the physical relevant quantity (whenever a physical relevant quantity is strictly needed). The idea now is to build $I(x)$ of complex wave components e^{ikx}:

$$I(x) \approx \sum_{k\in K} b_k e^{ikx}\,. \tag{2.80}$$

Here, k is the frequency of a component, K is some set of all the discrete k values needed to approximate $I(x)$ well, and b_k are constants that must be determined. We will very seldom need to compute the b_k coefficients: most of the insight we look for, and the understanding of the numerical methods we want to establish, come from investigating how the PDE and the scheme treat a single component e^{ikx} wave.

Letting the number of k values in K tend to infinity, makes the sum (2.80) converge to $I(x)$. This sum is known as a *Fourier series* representation of $I(x)$. Looking at (2.79), we see that the solution $u(x,t)$, when $I(x)$ is represented as in (2.80), is also built of basic complex exponential wave components of the form $e^{ik(x\pm ct)}$ according to

$$u(x,t) = \frac{1}{2}\sum_{k\in K} b_k e^{ik(x-ct)} + \frac{1}{2}\sum_{k\in K} b_k e^{ik(x+ct)}\,. \tag{2.81}$$

It is common to introduce the frequency in time $\omega = kc$ and assume that $u(x,t)$ is a sum of basic wave components written as $e^{ikx-\omega t}$. (Observe that inserting such a wave component in the governing PDE reveals that $\omega^2 = k^2c^2$, or $\omega = \pm kc$, reflecting the two solutions: one ($+kc$) traveling to the right and the other ($-kc$) traveling to the left.)

2.62. More precise definition of Fourier representations

The above introduction to function representation by sine and cosine waves was quick and intuitive, but will suffice as background knowledge for the following material of single wave component analysis. However, to understand all details of how different wave components sum up to the analytical and numerical solutions, a more precise mathematical treatment is helpful and therefore summarized below.

It is well known that periodic functions can be represented by Fourier series. A generalization of the Fourier series idea to non-periodic functions defined on the real line is the *Fourier transform*:

$$I(x) = \int_{-\infty}^{\infty} A(k)e^{ikx}dk, \tag{2.82}$$

$$A(k) = \int_{-\infty}^{\infty} I(x)e^{-ikx}dx\,. \tag{2.83}$$

The function $A(k)$ reflects the weight of each wave component e^{ikx} in an infinite sum of such wave components. That is, $A(k)$ reflects the frequency content in the function $I(x)$. Fourier transforms are particularly fundamental for analyzing and understanding time-varying signals.

The solution of the linear 1D wave PDE can be expressed as

$$u(x,t) = \int_{-\infty}^{\infty} A(k)e^{i(kx-\omega(k)t)}dx\,.$$

In a finite difference method, we represent u by a mesh function u_q^n, where n counts temporal mesh points and q counts the spatial ones (the usual counter for spatial points, i, is here already used as imaginary unit). Similarly, $I(x)$ is approximated by the mesh function I_q, $q = 0, \ldots, N_x$. On a mesh, it does not make sense to work with wave components e^{ikx} for very large k, because the shortest possible sine or cosine wave that can be represented uniquely on a mesh with spacing Δx is the wave with wavelength $2\Delta x$. This wave has its peaks and throughs at every two mesh points. That is, the wave "jumps up and down" between the mesh points.

The corresponding k value for the shortest possible wave in the mesh is $k = 2\pi/(2\Delta x) = \pi/\Delta x$. This maximum frequency is known as the *Nyquist frequency*. Within the range of relevant frequencies $(0, \pi/\Delta x]$ one defines the discrete Fourier transform, using $N_x + 1$ discrete frequencies:

$$I_q = \frac{1}{N_x + 1} \sum_{k=0}^{N_x} A_k e^{i2\pi kq/(N_x+1)}, \quad q = 0, \ldots, N_x, \tag{2.84}$$

$$A_k = \sum_{q=0}^{N_x} I_q e^{-i2\pi kq/(N_x+1)}, \quad k = 0, \ldots, N_x\,. \tag{2.85}$$

The A_k values represent the discrete Fourier transform of the I_q values, which themselves are the inverse discrete Fourier transform of the A_k values.

The discrete Fourier transform is efficiently computed by the *Fast Fourier transform* algorithm. For a real function $I(x)$, the relevant Python code for computing and plotting the discrete Fourier transform appears in the example below.

```
import numpy as np
from numpy import pi, sin

def I(x):
    return sin(2 * pi * x) + 0.5 * sin(4 * pi * x) + 0.1 * sin(6 * pi * x)

L = 10
Nx = 100
x = np.linspace(0, L, Nx + 1)
dx = L / float(Nx)

A = np.fft.rfft(I(x))
A_amplitude = np.abs(A)

freqs = np.linspace(0, pi / dx, A_amplitude.size)

import matplotlib.pyplot as plt

plt.plot(freqs, A_amplitude)
plt.show()
```

2.63. Stability

The scheme

$$[D_t D_t u = c^2 D_x D_x u]_q^n \tag{2.86}$$

for the wave equation $u_{tt} = c^2 u_{xx}$ allows basic wave components

$$u_q^n = e^{i(kx_q - \tilde{\omega} t_n)}$$

as solution, but it turns out that the frequency in time, $\tilde{\omega}$, is not equal to the exact frequency $\omega = kc$. The goal now is to find exactly what $\tilde{\omega}$ is. We ask two key questions:

- How accurate is $\tilde{\omega}$ compared to ω?
- Does the amplitude of such a wave component preserve its (unit) amplitude, as it should, or does it get amplified or damped in time (because of a complex $\tilde{\omega}$)?

The following analysis will answer these questions. We shall continue using q as an identifier for a certain mesh point in the x direction.

2.63.1. Preliminary results

A key result needed in the investigations is the finite difference approximation of a second-order derivative acting on a complex wave component:

$$[D_t D_t e^{i\omega t}]^n = -\frac{4}{\Delta t^2} \sin^2\left(\frac{\omega \Delta t}{2}\right) e^{i\omega n \Delta t}.$$

By just changing symbols ($\omega \to k$, $t \to x$, $n \to q$) it follows that

$$[D_x D_x e^{ikx}]_q = -\frac{4}{\Delta x^2} \sin^2\left(\frac{k \Delta x}{2}\right) e^{ikq\Delta x}.$$

Numerical wave propagation Inserting a basic wave component $u_q^n = e^{i(kx_q - \tilde{\omega} t_n)}$ in (2.86) results in the need to evaluate two expressions:

$$\begin{aligned}
[D_t D_t e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_t D_t e^{-i\tilde{\omega}t}]^n e^{ikq\Delta x} \\
&= -\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) e^{-i\tilde{\omega}n\Delta t} e^{ikq\Delta x} \qquad (2.87) \\
[D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_x D_x e^{ikx}]_q e^{-i\tilde{\omega}n\Delta t} \\
&= -\frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x} e^{-i\tilde{\omega}n\Delta t}. \qquad (2.88)
\end{aligned}$$

Then the complete scheme,

$$[D_t D_t e^{ikx} e^{-i\tilde{\omega}t} = c^2 D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n$$

leads to the following equation for the unknown numerical frequency $\tilde{\omega}$ (after dividing by $-e^{ikx}e^{-i\tilde{\omega}t}$):

$$\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2 \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right),$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2\sin^2\left(\frac{k\Delta x}{2}\right), \tag{2.89}$$

where

$$C = \frac{c\Delta t}{\Delta x}$$

is the Courant number. Taking the square root of (2.89) yields

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C\sin\left(\frac{k\Delta x}{2}\right), \tag{2.90}$$

Since the exact ω is real it is reasonable to look for a real solution $\tilde{\omega}$ of (2.90). The right-hand side of (2.90) must then be in $[-1,1]$ because the sine function on the left-hand side has values in $[-1,1]$ for real $\tilde{\omega}$. The magnitude of the sine function on the right-hand side attains the value 1 when

$$\frac{k\Delta x}{2} = \frac{\pi}{2} + m\pi, \quad m \in \mathbb{Z}\,.$$

With $m = 0$ we have $k\Delta x = \pi$, which means that the wavelength $\lambda = 2\pi/k$ becomes $2\Delta x$. This is the absolutely shortest wavelength that can be represented on the mesh: the wave jumps up and down between each mesh point. Larger values of $|m|$ are irrelevant since these correspond to k values whose waves are too short to be represented on a mesh with spacing Δx. For the shortest possible wave in the mesh, $\sin(k\Delta x/2) = 1$, and we must require

$$C \leq 1\,. \tag{2.91}$$

Consider a right-hand side in (2.90) of magnitude larger than unity. The solution $\tilde{\omega}$ of (2.90) must then be a complex number $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$ because the sine function is larger than unity for a complex argument. One can show that for any ω_i there will also be a corresponding solution with $-\omega_i$. The component with $\omega_i > 0$ gives an amplification factor $e^{\omega_i t}$ that grows exponentially in time. We cannot allow this and must therefore require $C \leq 1$ as a *stability criterion*.

> **Remark on the stability requirement**
>
> For smoother wave components with longer wave lengths per length Δx, (2.91) can in theory be relaxed. However, small round-off errors are always present in a numerical solution and these vary arbitrarily from mesh point to mesh point and can be viewed as unavoidable noise with wavelength $2\Delta x$. As explained, $C > 1$ will for this very small noise lead to exponential growth of the shortest possible wave component in the mesh. This noise will therefore grow with time and destroy the whole solution.

2.64. Numerical dispersion relation

Equation (2.90) can be solved with respect to $\tilde{\omega}$:

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(C\sin\left(\frac{k\Delta x}{2}\right)\right). \tag{2.92}$$

The relation between the numerical frequency $\tilde{\omega}$ and the other parameters k, c, Δx, and Δt is called a *numerical dispersion relation*. Correspondingly, $\omega = kc$ is the *analytical dispersion relation*. In general, dispersion refers to the phenomenon where the wave velocity depends on the spatial frequency (k, or the wave length $\lambda = 2\pi/k$) of the wave. Since the wave velocity is $\omega/k = c$, we realize that the analytical dispersion relation reflects the fact that there is no dispersion. However, in a numerical scheme we have dispersive waves where the wave velocity depends on k.

The special case $C = 1$ deserves attention since then the right-hand side of (2.92) reduces to

$$\frac{2}{\Delta t}\frac{k\Delta x}{2} = \frac{1}{\Delta t}\frac{\omega\Delta x}{c} = \frac{\omega}{C} = \omega\,.$$

That is, $\tilde{\omega} = \omega$ and the numerical solution is exact at all mesh points regardless of Δx and Δt! This implies that the numerical solution method is also an analytical solution method, at least for computing u at discrete points (the numerical method says nothing about the variation of u *between* the mesh points, and employing the common linear interpolation for extending the discrete solution gives a curve that in general deviates from the exact one).

For a closer examination of the error in the numerical dispersion relation when $C < 1$, we can study $\tilde{\omega} - \omega$, $\tilde{\omega}/\omega$, or the similar error measures in wave velocity: $\tilde{c} - c$ and $\tilde{c}/c$, where $c = \omega/k$ and $\tilde{c} = \tilde{\omega}/k$. It appears that the most convenient expression to work with is $\tilde{c}/c$, since it can be written as a function of just two parameters:

$$\frac{\tilde{c}}{c} = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right),$$

with $p = k\Delta x/2$ as a non-dimensional measure of the spatial frequency. In essence, p tells how many spatial mesh points we have per wave length in space for the wave component with frequency k (recall that the wave length is $2\pi/k$). That is, p reflects how well the spatial variation of the wave component is resolved in the mesh. Wave components with wave length less than $2\Delta x$ ($2\pi/k < 2\Delta x$) are not visible in the mesh, so it does not make sense to have $p > \pi/2$.

We may introduce the function $r(C, p) = \tilde{c}/c$ for further investigation of numerical errors in the wave velocity:

$$r(C,p) = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right), \quad C \in (0,1],\ p \in (0,\pi/2]\,. \tag{2.93}$$

This function is very well suited for plotting since it combines several parameters in the problem into a dependence on two dimensionless numbers, C and p.

Defining

```
def r(C, p):
    return 2/(C*p)*asin(C*sin(p))
```

we can plot $r(C, p)$ as a function of p for various values of C, see Figure Figure 2.6. Note that the shortest waves have the most erroneous velocity, and that short waves move more slowly than they should.

We can also easily make a Taylor series expansion in the discretization parameter p:

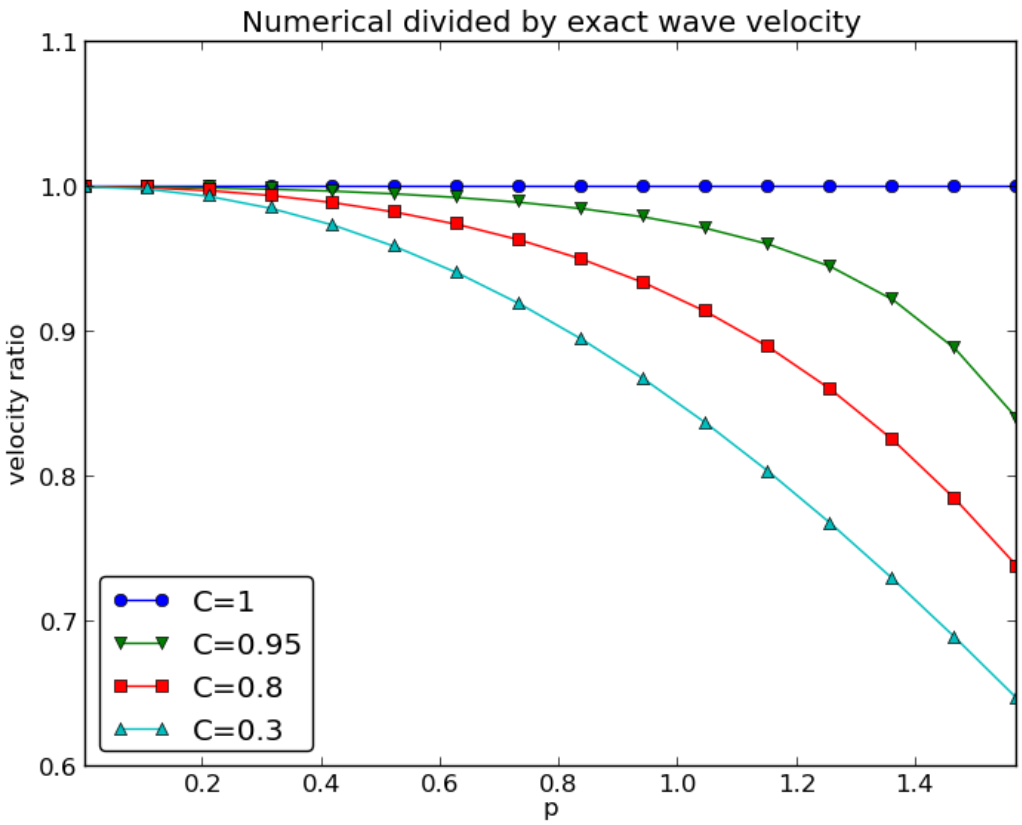


Figure 2.6.: The fractional error in the wave velocity for different Courant numbers.

```
>>> import sympy as sym
>>> C, p = sym.symbols('C p')
>>> # Compute the 7 first terms around p=0 with no O() term
>>> rs = r(C, p).series(p, 0, 7).removeO()
>>> rs
p**6*(5*C**6/112 - C**4/16 + 13*C**2/720 - 1/5040) +
p**4*(3*C**4/40 - C**2/12 + 1/120) +
p**2*(C**2/6 - 1/6) + 1

>>> # Pick out the leading order term, but drop the constant 1
>>> rs_error_leading_order = (rs - 1).extract_leading_order(p)
>>> rs_error_leading_order
p**2*(C**2/6 - 1/6)

>>> # Turn the series expansion into a Python function
>>> rs_pyfunc = lambdify([C, p], rs, modules='numpy')

>>> # Check: rs_pyfunc is exact (=1) for C=1
>>> rs_pyfunc(1, 0.1)
1.0
```

Note that without the `.removeO()` call the series gets an `O(x**7)` term that makes it impossible to convert the series to a Python function (for, e.g., plotting).

From the `rs_error_leading_order` expression above, we see that the leading order term in the error of this series expansion is

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2 (C^2-1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right),$$

pointing to an error $\mathcal{O}(\Delta t^2, \Delta x^2)$, which is compatible with the errors in the difference approximations (D_tD_tu and D_xD_xu).

We can do more with a series expansion, e.g., factor it to see how the factor $C-1$ plays a significant role. To this end, we make a list of the terms, factor each term, and then sum the terms:

```
>>> rs = r(C, p).series(p, 0, 4).removeO().as_ordered_terms()
>>> rs
[1, C**2*p**2/6 - p**2/6,
 3*C**4*p**4/40 - C**2*p**4/12 + p**4/120,
 5*C**6*p**6/112 - C**4*p**6/16 + 13*C**2*p**6/720 - p**6/5040]
>>> rs = [factor(t) for t in rs]
>>> rs
[1, p**2*(C - 1)*(C + 1)/6,
 p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120,
 p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040]
>>> rs = sum(rs)  # Python's sum function sums the list
>>> rs
```

```
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

We see from the last expression that $C = 1$ makes all the terms in `rs` vanish. Since we already know that the numerical solution is exact for $C = 1$, the remaining terms in the Taylor series expansion will also contain factors of $C - 1$ and cancel for $C = 1$.

2.65. Extending the analysis to 2D and 3D

The typical analytical solution of a 2D wave equation

$$u_{tt} = c^2(u_{xx} + u_{yy}),$$

is a wave traveling in the direction of $\boldsymbol{k} = k_x\boldsymbol{i} + k_y\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively ($\boldsymbol{i}$ should not be confused with $i = \sqrt{-1}$ here). Such a wave can be expressed by

$$u(x, y, t) = g(k_x x + k_y y - kct)$$

for some twice differentiable function g, or with $\omega = kc$, $k = |\boldsymbol{k}|$:

$$u(x, y, t) = g(k_x x + k_y y - \omega t)\,.$$

We can, in particular, build a solution by adding complex Fourier components of the form

$$e^{(i(k_x x + k_y y - \omega t))}\,.$$

A discrete 2D wave equation can be written as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]^n_{q,r}\,. \tag{2.94}$$

This equation admits a Fourier component

$$u^n_{q,r} = e^{(i(k_x q\Delta x + k_y r\Delta y - \tilde{\omega} n\Delta t))}, \tag{2.95}$$

as solution. Letting the operators $D_t D_t$, $D_x D_x$, and $D_y D_y$ act on $u^n_{q,r}$ from (2.95) transforms (2.94) to

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) + c^2\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right)\,.$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2\sin^2 p_x + C_y^2\sin^2 p_y,$$

where we have eliminated the factor 4 and introduced the symbols

$$C_x = \frac{c\Delta t}{\Delta x},\quad C_y = \frac{c\Delta t}{\Delta y},\quad p_x = \frac{k_x\Delta x}{2},\quad p_y = \frac{k_y\Delta y}{2}\,.$$

For a real-valued $\tilde{\omega}$ the right-hand side must be less than or equal to unity in absolute value, requiring in general that

$$C_x^2 + C_y^2 \leq 1\,. \tag{2.96}$$

This gives the stability criterion, more commonly expressed directly in an inequality for the time step:

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{2.97}$$

A similar, straightforward analysis for the 3D case leads to

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2}$$

In the case of a variable coefficient $c^2 = c^2(\boldsymbol{x})$, we must use the worst-case value

$$\bar{c} = \sqrt{\max_{\boldsymbol{x}\in\Omega} c^2(\boldsymbol{x})}$$

in the stability criteria. Often, especially in the variable wave velocity case, it is wise to introduce a safety factor $\beta \in (0, 1]$ too:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2}$$

The exact numerical dispersion relations in 2D and 3D becomes, for constant c,

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin^2 p_y\right)^{\frac{1}{2}}\right), \tag{2.98}$$

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin^2 p_y + C_z^2\sin^2 p_z\right)^{\frac{1}{2}}\right). \tag{2.99}$$

We can visualize the numerical dispersion error in 2D much like we did in 1D. To this end, we need to reduce the number of parameters in $\tilde{\omega}$. The direction of the wave is parameterized by the polar angle θ, which means that

$$k_x = k\sin\theta, \quad k_y = k\cos\theta\,.$$

A simplification is to set $\Delta x = \Delta y = h$. Then $C_x = C_y = c\Delta t/h$, which we call C. Also,

$$p_x = \frac{1}{2}kh\cos\theta, \quad p_y = \frac{1}{2}kh\sin\theta\,.$$

The numerical frequency $\tilde{\omega}$ is now a function of three parameters:

- C, reflecting the number of cells a wave is displaced during a time step,
- $p = \frac{1}{2}kh$, reflecting the number of cells per wave length in space,
- θ, expressing the direction of the wave.

We want to visualize the error in the numerical frequency. To avoid having Δt as a free parameter in $\tilde{\omega}$, we work with $\tilde{c}/c = \tilde{\omega}/(kc)$. The coefficient in front of the $\sin^{-1}$ factor is then

$$\frac{2}{kc\Delta t} = \frac{2}{2kc\Delta t h/h} = \frac{1}{Ckh} = \frac{2}{Cp},$$

and

$$\frac{\tilde{c}}{c} = \frac{2}{Cp}\sin^{-1}\left(C\left(\sin^2(p\cos\theta) + \sin^2(p\sin\theta)\right)^{\frac{1}{2}}\right).$$

We want to visualize this quantity as a function of p and θ for some values of $C \leq 1$. It is instructive to make color contour plots of $1 - \tilde{c}/c$ in *polar coordinates* with θ as the angular coordinate and p as the radial coordinate.

The stability criterion (2.96) becomes $C \leq C_{\max} = 1/\sqrt{2}$ in the present 2D case with the C defined above. Let us plot $1 - \tilde{c}/c$ in polar coordinates for $C_{\max}, 0.9C_{\max}, 0.5C_{\max}, 0.2C_{\max}$. The program below does the somewhat tricky work in Matplotlib, and the result appears in Figure Figure 2.7. From the figure we clearly see that the maximum C value gives the best results, and that waves whose propagation direction makes an angle of 45 degrees with an axis are the most accurate.

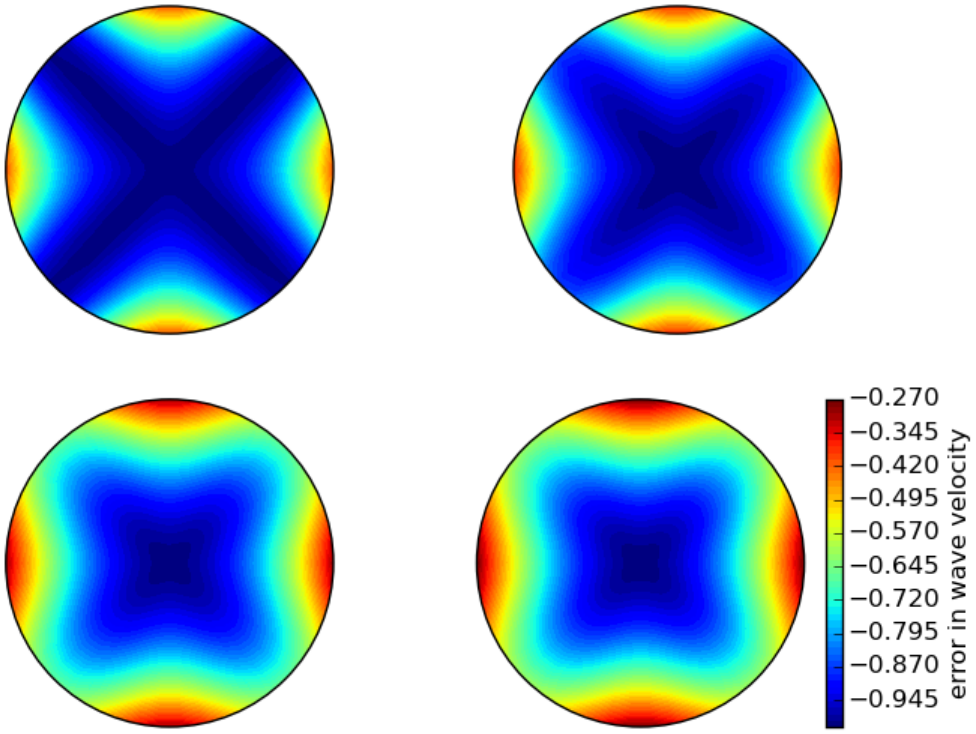


Figure 2.7.: Error in numerical dispersion in 2D.

2.66. Multi-dimensional wave equations

A natural next step is to consider extensions of the methods for various variants of the one-dimensional wave equation to two-dimensional (2D) and three-dimensional (3D) versions of the wave equation.

2.67. Multi-dimensional wave equations

The general wave equation in d space dimensions, with constant wave velocity c, can be written in the compact form

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0,T], \tag{2.100}$$

where

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

in a 2D problem ($d = 2$) and

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2},$$

in three space dimensions ($d = 3$).

Many applications involve variable coefficients, and the general wave equation in d dimensions is in this case written as

$$\varrho\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(q\nabla u) + f \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0,T], \tag{2.101}$$

which in, e.g., 2D becomes

$$\varrho(x,y)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t)\,.$$

To save some writing and space we may use the index notation, where subscript t, x, or y means differentiation with respect to that coordinate. For example,

$$\frac{\partial^2 u}{\partial t^2} = u_{tt},$$
$$\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) = (qu_y)_y$$

.These comments extend straightforwardly to 3D, which means that the 3D versions of the two wave PDEs, with and without variable coefficients, can be stated as

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f, \tag{2.102}$$
$$\varrho u_{tt} = (qu_x)_x + (qu_y)_y + (qu_z)_z + f\,. \tag{2.103}$$

At *each point* of the boundary $\partial\Omega$ (of Ω) we need *one* boundary condition involving the unknown u. The boundary conditions are of three principal types:

1. u is prescribed ($u = 0$ or a known time variation of u at the boundary points, e.g., modeling an incoming wave),
2. $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ is prescribed (zero for reflecting boundaries),
3. an open boundary condition (also called radiation condition) is specified to let waves travel undisturbed out of the domain, see Exercise Section 2.57 for details.

All the listed wave equations with *second-order* derivatives in time need *two* initial conditions:

1. $u = I$,
2. $u_t = V$.

2.68. Mesh

We introduce a mesh in time and in space. The mesh in time consists of time points

$$t_0 = 0 < t_1 < \cdots < t_{N_t},$$

normally, for wave equation problems, with a constant spacing $\Delta t = t_{n+1} - t_n$, $n \in \mathcal{I}_t^-$.

Finite difference methods are easy to implement on simple rectangle- or box-shaped *spatial domains*. More complicated shapes of the spatial domain require substantially more advanced techniques and implementational efforts (and a finite element method is usually a more convenient approach). On a rectangle- or box-shaped domain, mesh points are introduced separately in the various space directions:

$$\begin{aligned} x_0 < x_1 < \cdots < x_{N_x} &\text{ in the } x \text{ direction,} \\ y_0 < y_1 < \cdots < y_{N_y} &\text{ in the } y \text{ direction,} \\ z_0 < z_1 < \cdots < z_{N_z} &\text{ in the } z \text{ direction .} \end{aligned}$$

We can write a general mesh point as (x_i, y_j, z_k, t_n), with $i \in \mathcal{I}_x$, $j \in \mathcal{I}_y$, $k \in \mathcal{I}_z$, and $n \in \mathcal{I}_t$.

It is a very common choice to use constant mesh spacings: $\Delta x = x_{i+1} - x_i$, $i \in \mathcal{I}_x^-$, $\Delta y = y_{j+1} - y_j$, $j \in \mathcal{I}_y^-$, and $\Delta z = z_{k+1} - z_k$, $k \in \mathcal{I}_z^-$. With equal mesh spacings one often introduces $h = \Delta x = \Delta y = \Delta z$.

The unknown u at mesh point (x_i, y_j, z_k, t_n) is denoted by $u^n_{i,j,k}$. In 2D problems we just skip the z coordinate (by assuming no variation in that direction: $\partial/\partial z = 0$) and write $u^n_{i,j}$.

2.69. Discretization

Two- and three-dimensional wave equations are easily discretized by assembling building blocks for discretization of 1D wave equations, because the multi-dimensional versions just contain terms of the same type as those in 1D.

2.69.1. Discretizing the PDEs

Equation (2.102) can be discretized as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u + D_z D_z u) + f]^n_{i,j,k}\,.$$

A 2D version might be instructive to write out in detail:

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]^n_{i,j},$$

which becomes

$$\frac{u^{n+1} * *i, j - 2u^n * *i, j + u^{n-1}_{i,j}}{\Delta t^2} = c^2 \frac{u^n * *i+1, j - 2u^n * *i, j + u^n_{i-1,j}}{\Delta x^2} + c^2 \frac{u^n * *i, j+1 - 2u^n * *i, j + u^n_{i,j-1}}{\Delta y^2} + f^n_{i,j}$$

Assuming, as usual, that all values at time levels n and $n-1$ are known, we can solve for the only unknown $u^{n+1}_{i,j}$. The result can be compactly written as

$$u^{n+1} * *i,j = 2u^n * *i,j + u^{n-1}_{i,j} + c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{2.104}$$

As in the 1D case, we need to develop a special formula for $u^1_{i,j}$ where we combine the general scheme for $u^{n+1}_{i,j}$, when $n = 0$, with the discretization of the initial condition:

$$[D_{2t}u = V]^0_{i,j} \quad \Rightarrow \quad u^{-1} * *i,j = u^1 * *i,j - 2\Delta t V_{i,j}\,.$$

The result becomes, in compact form,

$$u^1 * *i,j = u^0 * *i,j - 2\Delta V_{i,j} + \frac{1}{2}c^2\Delta t^2[D_xD_xu + D_yD_yu]^0_{i,j}\,. \tag{2.105}$$

The PDE (2.103) with variable coefficients is discretized term by term using the corresponding elements from the 1D case:

$$[\varrho D_tD_tu = (D_x\overline{q}^xD_xu + D_y\overline{q}^yD_yu + D_z\overline{q}^zD_zu) + f]^n_{i,j,k}\,.$$

When written out and solved for the unknown $u^{n+1}_{i,j,k}$, one gets the scheme

$$\begin{aligned}
u^{n+1} * *i,j,k = {} & -u^{n-1} * *i,j,k + 2u^n_{i,j,k} + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i+1,j,k})(u^n * *i+1,j,k - u^n * *i,j,k) - \\
& \quad \frac{1}{2}(q_{i-1,j,k} + q_{i,j,k})(u^n * *i,j,k - u^n * *i-1,j,k)) + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta y^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j+1,k})(u^n * *i,j+1,k - u^n * *i,j,k) - \\
& \quad \frac{1}{2}(q_{i,j-1,k} + q_{i,j,k})(u^n * *i,j,k - u^n * *i,j-1,k)) + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta z^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j,k+1})(u^n * *i,j,k+1 - u^n * *i,j,k) - \\
& \quad \frac{1}{2}(q_{i,j,k-1} + q_{i,j,k})(u^n * *i,j,k - u^n * *i,j,k-1)) + \\
& \Delta t^2 f^n_{i,j,k}
\end{aligned}$$

.

Also here we need to develop a special formula for $u^1_{i,j,k}$ by combining the scheme for $n = 0$ with the discrete initial condition, which is just a matter of inserting $u^{-1} * *i,j,k = u^1 * *i,j,k - 2\Delta t V_{i,j,k}$ in the scheme and solving for $u^1_{i,j,k}$.

2.69.2. Handling boundary conditions where u is known

The schemes listed above are valid for the internal points in the mesh. After updating these, we need to visit all the mesh points at the boundaries and set the prescribed u value.

2.69.3. Discretizing the Neumann condition

The condition $\partial u/\partial n = 0$ was implemented in 1D by discretizing it with a $D_{2x}u$ centered difference, followed by eliminating the fictitious u point outside the mesh by using the general scheme at the boundary point. Alternatively, one can introduce ghost cells and update a ghost value for use in the Neumann condition. Exactly the same ideas are reused in multiple dimensions.

Consider the condition $\partial u/\partial n = 0$ at a boundary $y = 0$ of a rectangular domain $[0, L_x] \times [0, L_y]$ in 2D. The normal direction is then in $-y$ direction, so

$$\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y},$$

and we set

$$[-D_{2y}u = 0]^n_{i,0} \quad \Rightarrow \quad \frac{u^n_{i,1} - u^n_{i,-1}}{2\Delta y} = 0\,.$$

From this it follows that $u^n_{i,-1} = u^n_{i,1}$. The discretized PDE at the boundary point $(i, 0)$ reads

$$\frac{u^{n+1} * *i, 0 - 2u^n * *i, 0 + u^{n-1}_{i,0}}{\Delta t^2} = c^2 \frac{u^n * *i + 1, 0 - 2u^n * *i, 0 + u^n_{i-1,0}}{\Delta x^2} + c^2 \frac{u^n * *i, 1 - 2u^n * *i, 0 + u^n_{i,-1}}{\Delta y^2} + f^n_{i,j},$$

We can then just insert $u^n_{i,1}$ for $u^n_{i,-1}$ in this equation and solve for the boundary value $u^{n+1}_{i,0}$, just as was done in 1D.

From these calculations, we see a pattern: the general scheme applies at the boundary $j = 0$ too if we just replace $j - 1$ by $j + 1$. Such a pattern is particularly useful for implementations. The details follow from the explained 1D case in Section Section 2.38.

The alternative approach to eliminating fictitious values outside the mesh is to have $u^n_{i,-1}$ available as a ghost value. The mesh is extended with one extra line (2D) or plane (3D) of ghost cells at a Neumann boundary. In the present example it means that we need a line with ghost cells below the y axis. The ghost values must be updated according to $u^{n+1} * *i, -1 = u^{n+1} * *i, 1$.

2.70. The 2D Wave Equation with Devito

Extending the wave solver to two dimensions illustrates the power of Devito's dimension-agnostic approach. The same symbolic patterns apply, and Devito automatically generates optimized 2D stencils.

2.70.1. The 2D Wave Equation

The two-dimensional wave equation on $[0, L_x] \times [0, L_y]$ is:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = c^2 \nabla^2 u \tag{2.106}$$

where $\nabla^2 u = u_{xx} + u_{yy}$ is the Laplacian.

2.70.2. Devito's Dimension-Agnostic Laplacian

Devito provides the `.laplace` attribute that works in any dimension:

```
from devito import Grid, TimeFunction

# 2D grid
grid = Grid(shape=(Nx + 1, Ny + 1), extent=(Lx, Ly))

# 2D wave field
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

# The Laplacian works the same as in 1D!
laplacian = u.laplace  # Returns u_xx + u_yy automatically
```

This is one of Devito's key strengths: the same code pattern scales from 1D to 2D to 3D without changes.

2.70.3. CFL Stability Condition in 2D

The stability condition in 2D is more restrictive than in 1D:

$$C = c \cdot \Delta t \cdot \sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}} \leq 1$$

For equal grid spacing $\Delta x = \Delta y = h$:

$$\Delta t \leq \frac{h}{c\sqrt{2}}$$

Compared to the 1D condition $\Delta t \leq h/c$, the 2D condition allows smaller time steps by a factor of $1/\sqrt{2} \approx 0.707$.

2.70.4. The 2D Solver

The `src.wave` module provides `solve_wave_2d`:

```
from src.wave import solve_wave_2d
import numpy as np

# Initial condition: 2D standing wave
def I(X, Y):
    return np.sin(np.pi * X) * np.sin(np.pi * Y)

result = solve_wave_2d(
    Lx=1.0, Ly=1.0,    # Domain size
    c=1.0,              # Wave speed
```

```
    Nx=50, Ny=50,        # Grid points
    T=1.0,               # Final time
    C=0.5,               # Courant number
    I=I,                 # Initial displacement
)

# Result is a 2D array
print(result.u.shape)  # (51, 51)
```

2.70.5. 2D Boundary Conditions

Dirichlet conditions must be applied on all four boundaries:

```
from devito import Eq

t_dim = grid.stepping_dim
x_dim, y_dim = grid.dimensions

# Boundary conditions (u = 0 on all boundaries)
bc_x0 = Eq(u[t_dim + 1, 0, y_dim], 0)      # Left
bc_xN = Eq(u[t_dim + 1, Nx, y_dim], 0)     # Right
bc_y0 = Eq(u[t_dim + 1, x_dim, 0], 0)      # Bottom
bc_yN = Eq(u[t_dim + 1, x_dim, Ny], 0)     # Top
```

2.70.6. Standing Waves in 2D

The exact solution for the initial condition $I(x, y) = \sin(\pi x/L_x)\sin(\pi y/L_y)$ with $V = 0$ is:

$$u(x, y, t) = \sin\left(\frac{\pi x}{L_x}\right)\sin\left(\frac{\pi y}{L_y}\right)\cos(\omega t)$$

where the angular frequency is:

$$\omega = c\pi\sqrt{\frac{1}{L_x^2} + \frac{1}{L_y^2}}$$

This can be used for verification:

```
from src.wave import convergence_test_wave_2d

grid_sizes, errors, rate = convergence_test_wave_2d(
    grid_sizes=[10, 20, 40, 80],
    T=0.25,
    C=0.5,
)

print(f"Observed convergence rate: {rate:.2f}")  # Should be ~2.0
```

2.70.7. Visualizing 2D Solutions

For 2D problems, surface plots and contour plots are useful:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

result = solve_wave_2d(Lx=1.0, Ly=1.0, Nx=50, Ny=50, T=0.5, C=0.5)

X, Y = np.meshgrid(result.x, result.y, indexing='ij')

fig = plt.figure(figsize=(12, 5))

# Surface plot
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, result.u, cmap='viridis')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_zlabel('u')
ax1.set_title(f't = {result.t:.3f}')

# Contour plot
ax2 = fig.add_subplot(122)
c = ax2.contourf(X, Y, result.u, levels=20, cmap='RdBu_r')
plt.colorbar(c, ax=ax2)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Contour plot')
ax2.set_aspect('equal')
```

2.70.8. Animation of 2D Waves

```
from matplotlib.animation import FuncAnimation

result = solve_wave_2d(
    Lx=1.0, Ly=1.0, Nx=50, Ny=50, T=2.0, C=0.5,
    save_history=True,
)

fig, ax = plt.subplots()
X, Y = np.meshgrid(result.x, result.y, indexing='ij')

vmax = np.abs(result.u_history).max()
im = ax.contourf(X, Y, result.u_history[0], levels=20,
                 cmap='RdBu_r', vmin=-vmax, vmax=vmax)
```

```
def update(frame):
    ax.clear()
    ax.contourf(X, Y, result.u_history[frame], levels=20,
                cmap='RdBu_r', vmin=-vmax, vmax=vmax)
    ax.set_title(f't = {result.t_history[frame]:.3f}')
    ax.set_aspect('equal')
    return []

anim = FuncAnimation(fig, update, frames=len(result.t_history),
                     interval=50)
```

2.70.9. From 2D to 3D

The pattern extends naturally to three dimensions. In Devito, the main changes are:

1. Add a third dimension to the grid
2. The `.laplace` attribute automatically includes u_{zz}

```
# 3D grid
grid = Grid(shape=(Nx+1, Ny+1, Nz+1), extent=(Lx, Ly, Lz))

# 3D wave field
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)

# The PDE is unchanged!
pde = u.dt2 - c**2 * u.laplace
```

The CFL condition in 3D becomes:

$$\Delta t \leq \frac{h}{c\sqrt{3}}$$

for equal grid spacing in all directions.

2.70.10. Computational Considerations

2D and 3D wave simulations can become computationally expensive. Devito helps through:

- **Automatic parallelization**: Set `OMP_NUM_THREADS` for OpenMP
- **Cache optimization**: Loop tiling is applied automatically
- **GPU support**: Use `platform='nvidiaX'` for CUDA execution

For large-scale simulations, the generated C code is highly optimized and can match hand-tuned implementations.

2.70.11. Summary

Key points for 2D wave equations with Devito:

1. The `.laplace` attribute handles the dimension automatically
2. CFL conditions are more restrictive (factor of $1/\sqrt{d}$ in d dimensions)
3. Boundary conditions must be applied on all boundaries
4. Visualization requires surface/contour plots and animations
5. The same code patterns extend to 3D with minimal changes

Devito's abstraction means we write the physics once and let the framework handle the computational complexity across dimensions.

2.71. Implementation of 2D and 3D wave equations

We shall now describe in detail various Python implementations for solving a standard 2D, linear wave equation with constant wave velocity and $u = 0$ on the boundary. The wave equation is to be solved in the space-time domain $\Omega \times (0, T]$, where $\Omega = (0, L_x) \times (0, L_y)$ is a rectangular spatial domain. More precisely, the complete initial-boundary value problem is defined by

$$u_{tt} = c^2(u_{xx} + u_{yy}) + f(x, y, t), \quad (x, y) \in \Omega,\ t \in (0, T], \tag{2.107}$$
$$u(x, y, 0) = I(x, y), \quad (x, y) \in \Omega, \tag{2.108}$$
$$u_t(x, y, 0) = V(x, y), \quad (x, y) \in \Omega, \tag{2.109}$$
$$u = 0, \quad (x, y) \in \partial\Omega,\ t \in (0, T], \tag{2.110}$$

where $\partial\Omega$ is the boundary of Ω, in this case the four sides of the rectangle $\Omega = [0, L_x] \times [0, L_y]$: $x = 0$, $x = L_x$, $y = 0$, and $y = L_y$.

The PDE is discretized as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]^n_{i,j},$$

which leads to an explicit updating formula to be implemented in a program:

$$\begin{aligned} u^{n+1}_{i,j} = &-u^{n-1}_{i,j} + 2u^n_{i,j} + \\ &C_x^2(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}) + C_y^2(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}) + \Delta t^2 f^n_{i,j}, \end{aligned} \tag{2.111}$$

for all interior mesh points $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$, for $n \in \mathcal{I}^+_t$. The constants C_x and C_y are defined as

$$C_x = c\frac{\Delta t}{\Delta x}, \quad C_y = c\frac{\Delta t}{\Delta y}.$$

At the boundary, we simply set $u^{n+1}_{i,j} = 0$ for $i = 0$, $j = 0, \ldots, N_y$; $i = N_x$, $j = 0, \ldots, N_y$; $j = 0$, $i = 0, \ldots, N_x$; and $j = N_y$, $i = 0, \ldots, N_x$. For the first step, $n = 0$, (2.111) is combined with the discretization of the initial condition $u_t = V$, $[D_{2t}u = V]^0_{i,j}$ to obtain a special formula for $u^1_{i,j}$ at the interior mesh points:

$$u^1_{i,j} = u^0_{i,j} + \Delta t V_{i,j} + \frac{1}{2}C_x^2(u^0_{i+1,j} - 2u^0_{i,j} + u^0_{i-1,j}) + \frac{1}{2}C_y^2(u^0_{i,j+1} - 2u^0_{i,j} + u^0_{i,j-1}) + \frac{1}{2}\Delta t^2 f^n_{i,j}, \tag{2.112}$$

The algorithm is very similar to the one in 1D:

1. Set initial condition $u^0_{i,j} = I(x_i, y_j)$
2. Compute $u^1_{i,j}$ from (2.111)
3. Set $u^1_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$
4. For $n = 1, 2, \ldots, N_t$:
5. Find $u^{n+1}_{i,j}$ from (2.111) for all internal mesh points, $i \in \mathcal{I}^i_x$, $j \in \mathcal{I}^i_y$
6. Set $u^{n+1}_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

2.72. Scalar computations

The `solver` function for a 2D case with constant wave velocity and boundary condition $u = 0$ is analogous to the 1D case with similar parameter values (see `wave1D_u0.py`), apart from a few necessary extensions. The code is found in the program `wave2D_u0.py`.

2.72.1. Domain and mesh

The spatial domain is now $[0, L_x] \times [0, L_y]$, specified by the arguments `Lx` and `Ly`. Similarly, the number of mesh points in the x and y directions, N_x and N_y, become the arguments `Nx` and `Ny`. In multi-dimensional problems it makes less sense to specify a Courant number since the wave velocity is a vector and mesh spacings may differ in the various spatial directions. We therefore give Δt explicitly. The signature of the `solver` function is then

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Key parameters used in the calculations are created as

```
x = linspace(0, Lx, Nx+1)                    # mesh points in x dir
y = linspace(0, Ly, Ny+1)                    # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)                   # mesh points in time
Cx2 = (c*dt/dx)**2;  Cy2 = (c*dt/dy)**2      # help variables
dt2 = dt**2
```

2.72.2. Solution arrays

We store $u^{n+1} * *i, j$, $u^n * *i, j$, and $u^{n-1}_{i,j}$ in three two-dimensional arrays,

```
u   = zeros((Nx+1,Ny+1))   # solution array
u_n = [zeros((Nx+1,Ny+1)), zeros((Nx+1,Ny+1))]  # t-dt, t-2*dt
```

where $u^{n+1}_{i,j}$ corresponds to `u[i,j]`, $u^n_{i,j}$ to `u_n[i,j]`, and $u^{n-1}_{i,j}$ to `u_nm1[i,j]`.

2.72.3. Index sets

It is also convenient to introduce the index sets (cf. Section Section 2.39)

```
Ix = range(0, u.shape[0])
It = range(0, u.shape[1])
It = range(0, t.shape[0])
```

2.72.4. Computing the solution

Inserting the initial condition `I` in `u_n` and making a callback to the user in terms of the `user_action` function is a straightforward generalization of the 1D code from Section Section 2.7:

```
for i in Ix:
    for j in It:
        u_n[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_n, x, xv, y, yv, t, 0)
```

The `user_action` function has additional arguments compared to the 1D case. The arguments `xv` and `yv` will be commented upon in Section Section 2.73.

The key finite difference formula (2.104) for updating the solution at a time level is implemented in a separate function as

```
def advance_scalar(u, u_n, u_nm1, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  It = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
```

```
        for j in It[1:-1]:
            u_xx = u_n[i-1,j] - 2*u_n[i,j] + u_n[i+1,j]
            u_yy = u_n[i,j-1] - 2*u_n[i,j] + u_n[i,j+1]
            u[i,j] = D1*u_n[i,j] - D2*u_nm1[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    j = It[0]
    for i in Ix: u[i,j] = 0
    j = It[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in It: u[i,j] = 0
    i = Ix[-1]
    for j in It: u[i,j] = 0
    return u
```

The `step1` variable has been introduced to allow the formula to be reused for the first step, computing $u^1_{i,j}$:

```
u = advance_scalar(u, u_n, f, x, y, t,
                   n, Cx2, Cy2, dt, V, step1=True)
```

Below, we will make many alternative implementations of the `advance_scalar` function to speed up the code since most of the CPU time in simulations is spent in this function.

> **i** Remark: How to use the solution
>
> The `solver` function in the `wave2D_u0.py` code updates arrays for the next time step by switching references as described in Section Section 2.27. Any use of `u` on the user's side is assumed to take place in the user action function. However, should the code be changed such that `u` is returned and used as solution, have in mind that you must return `u_n` after the time limit, otherwise a `return u` will actually return `u_nm1` (due to the switching of array indices in the loop)!

2.73. Vectorized computations

The scalar code above turns out to be extremely slow for large 2D meshes, and probably useless in 3D beyond debugging of small test cases. Vectorization is therefore a must for multi-dimensional finite difference computations in Python. For example, with a mesh consisting of 30×30 cells, vectorization brings down the CPU time by a factor of 70 (!). Equally important, vectorized code can also easily be parallelized to take (usually) optimal advantage of parallel computer platforms.

In the vectorized case, we must be able to evaluate user-given functions like $I(x, y)$ and $f(x, y, t)$ for the entire mesh in one operation (without loops). These user-given functions are provided as

Python functions `I(x,y)` and `f(x,y,t)`, respectively. Having the one-dimensional coordinate arrays `x` and `y` is not sufficient when calling `I` and `f` in a vectorized way. We must extend `x` and `y` to their vectorized versions `xv` and `yv`:

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]
xv = x.reshape((x.size, 1))
yv = y.reshape((1, y.size))
```

This is a standard required technique when evaluating functions over a 2D mesh, say `sin(xv)*cos(xv)`, which then gives a result with shape `(Nx+1,Ny+1)`. Calling `I(xv, yv)` and `f(xv, yv, t[n])` will now return `I` and `f` values for the entire set of mesh points.

With the `xv` and `yv` arrays for vectorized computing, setting the initial condition is just a matter of

```
u_n[:,:] = I(xv, yv)
```

One could also have written `u_n = I(xv, yv)` and let `u_n` point to a new object, but vectorized operations often make use of direct insertion in the original array through `u_n[:,:]`, because sometimes not all of the array is to be filled by such a function evaluation. This is the case with the computational scheme for $u_{i,j}^{n+1}$:

```
def advance_vectorized(u, u_n, u_nm1, f_a, Cx2, Cy2, dt2,
                       V=None, step1=False):
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    u_xx = u_n[:-2,1:-1] - 2*u_n[1:-1,1:-1] + u_n[2:,1:-1]
    u_yy = u_n[1:-1,:-2] - 2*u_n[1:-1,1:-1] + u_n[1:-1,2:]
    u[1:-1,1:-1] = D1*u_n[1:-1,1:-1] - D2*u_nm1[1:-1,1:-1] + \
                   Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]
    if step1:
        u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
    j = 0
    u[:,j] = 0
    j = u.shape[1]-1
    u[:,j] = 0
    i = 0
    u[i,:] = 0
    i = u.shape[0]-1
    u[i,:] = 0
    return u
```

Array slices in 2D are more complicated to understand than those in 1D, but the logic from 1D applies to each dimension separately. For example, when doing $u^n**i,j - u^n**i-1,j$ for $i \in \mathcal{I}_x^+$, we just keep `j` constant and make a slice in the first index: `u_n[1:,j] - u_n[:-1,j]`, exactly as in 1D. The `1:` slice specifies all the indices $i = 1, 2, \ldots, N_x$ (up to the last valid index), while `:-1` specifies the relevant indices for the second term: $0, 1, \ldots, N_x - 1$ (up to, but not including the last index).

In the above code segment, the situation is slightly more complicated, because each displaced slice in one direction is accompanied by a `1:-1` slice in the other direction. The reason is that we only work with the internal points for the index that is kept constant in a difference.

The boundary conditions along the four sides make use of a slice consisting of all indices along a boundary:

```
u[: ,0] = 0
u[:,Ny] = 0
u[0 ,:] = 0
u[Nx,:] = 0
```

In the vectorized update of `u` (above), the function `f` is first computed as an array over all mesh points:

```
f_a = f(xv, yv, t[n])
```

We could, alternatively, have used the call `f(xv, yv, t[n])[1:-1,1:-1]` in the last term of the update statement, but other implementations in compiled languages benefit from having `f` available in an array rather than calling our Python function `f(x,y,t)` for every point.

Also in the `advance_vectorized` function we have introduced a boolean `step1` to reuse the formula for the first time step in the same way as we did with `advance_scalar`. We refer to the `solver` function in `wave2D_u0.py` for the details on how the overall algorithm is implemented.

The callback function now has the arguments `u, x, xv, y, yv, t, n`. The inclusion of `xv` and `yv` makes it easy to, e.g., compute an exact 2D solution in the callback function and compute errors, through an expression like `u - u_exact(xv, yv, t[n])`.

2.74. Verification

2.74.1. Testing a quadratic solution

The 1D solution from Section Section 2.11 can be generalized to multi-dimensions and provides a test case where the exact solution also fulfills the discrete equations, such that we know (to machine precision) what numbers the solver function should produce. In 2D we use the following generalization of (2.25):

$$u_{\mathrm{e}}(x, y, t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t)\,. \tag{2.113}$$

This solution fulfills the PDE problem if $I(x,y) = u_{\mathrm{e}}(x,y,0)$, $V = \frac{1}{2}u_{\mathrm{e}}(x,y,0)$, and $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$. To show that u_{e} also solves the discrete equations, we start with the general results $[D_tD_t 1]^n = 0$, $[D_tD_t t]^n = 0$, and $[D_tD_t t^2] = 2$, and use these to compute

$$
\begin{aligned}
[D_xD_x u_{\mathrm{e}}]^n_{i,j} &= [y(L_y - y)(1 + \frac{1}{2}t)D_xD_x x(L_x - x)]^n_{i,j} \\
&= y_j(L_y - y_j)(1 + \frac{1}{2}t_n)(-2)\,.
\end{aligned}
$$

A similar calculation must be carried out for the $[D_yD_y u_{\mathrm{e}}]^n_{i,j}$ and $[D_tD_t u_{\mathrm{e}}]^n_{i,j}$ terms. One must also show that the quadratic solution fits the special formula for $u^1_{i,j}$. The details are left as Exercise Section 2.76. The `test_quadratic` function in the wave2D_u0.py program implements this verification as a proper test function for the pytest and nose frameworks.

2.75. Visualization

Eventually, we are ready for a real application with our code! Look at the `wave2D_u0.py` and the `gaussian` function. It starts with a Gaussian function to see how it propagates in a square with $u = 0$ on the boundaries:

```
def gaussian(plot_method=2, version='vectorized', save_plot=True):
    """
    Initial Gaussian bell in the middle of the domain.
    plot_method=1 applies mesh function,
    =2 means surf, =3 means Matplotlib, =4 means mayavi,
    =0 means no plot.
    """
    for name in glob('tmp_*.png'):
        os.remove(name)

    Lx = 10
    Ly = 10
    c = 1.0

    from numpy import exp

    def I(x, y):
        """Gaussian peak at (Lx/2, Ly/2)."""
        return exp(-0.5*(x-Lx/2.0)**2 - 0.5*(y-Ly/2.0)**2)

    def plot_u(u, x, xv, y, yv, t, n):
        """User action function for plotting."""
    ...

    Nx = 40; Ny = 40; T = 20
```

```
    dt, cpu = solver(I, None, None, c, Lx, Ly, Nx, Ny, -1, T,
                     user_action=plot_u, version=version)
```

2.75.1. Matplotlib

We want to animate a 3D surface in Matplotlib, but this is a really slow process and not recommended, so we consider Matplotlib not an option as long as on-screen animation is desired. One can use the recipes for single shots of u, where it does produce high-quality 3D plots.

2.75.2. Gnuplot

Let us look at different ways for visualization using Gnuplot. If you have the C package Gnuplot and the `Gnuplot.py` Python interface module installed, you can get nice 3D surface plots with contours beneath (Figure Figure 2.8). It gives a nice visualization with lifted surface and contours beneath. Figure Figure 2.8 shows four plots of u.

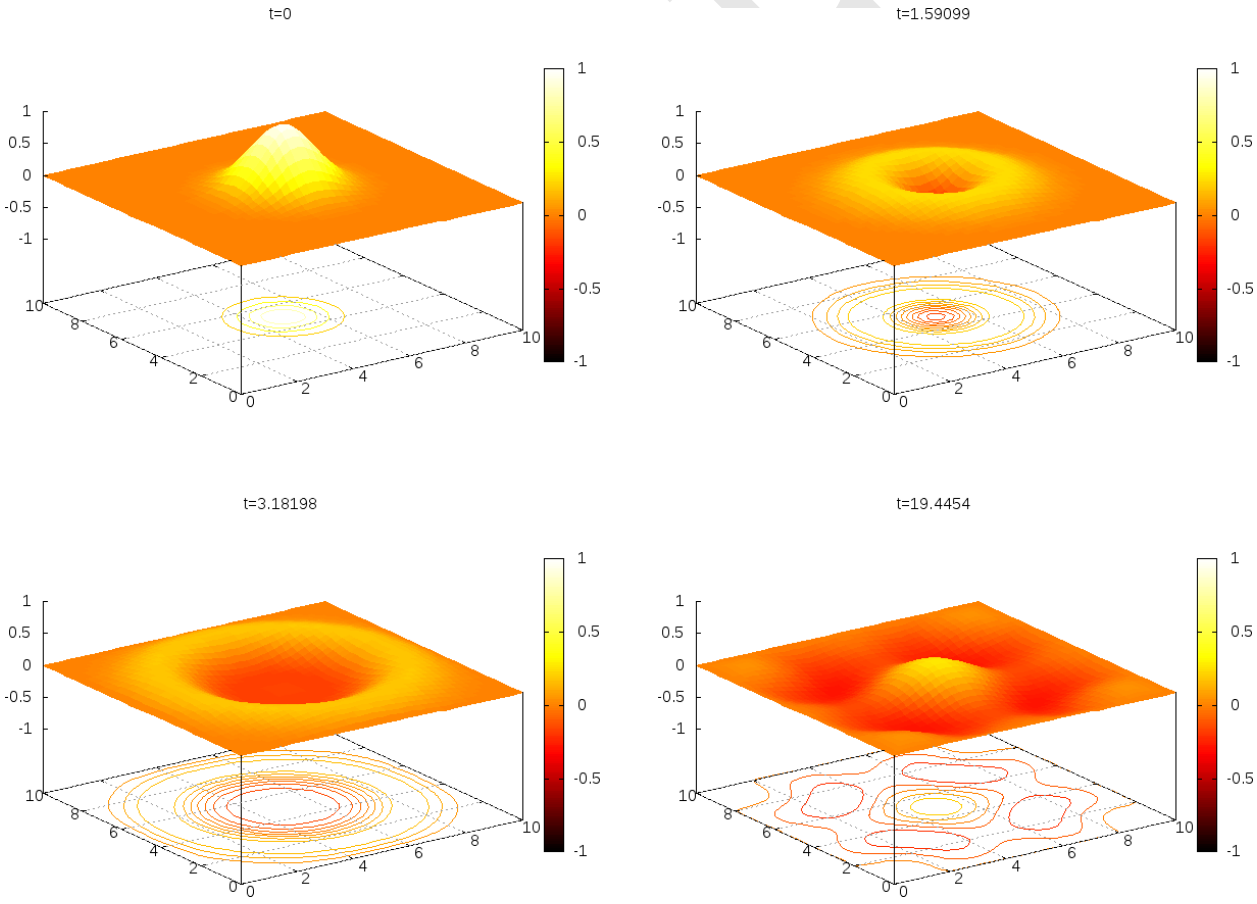


Figure 2.8.: Snapshots of the surface plotted by Gnuplot.

Video files can be made of the PNG frames:

```
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec flv movie.flv
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec linx264 movie.mp4
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec libvpx movie.webm
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec libtheora movie.ogg
```

It is wise to use a high frame rate – a low one will just skip many frames. There may also be considerable quality differences between the different formats.

MOVIE: [https://raw.githubusercontent.com/hplgit/fdm-book/master/doc/pub/book/html/mov-wave/gnuplot/wave2D_u0_gaussian/movie25.mp4]

2.75.3. Mayavi

The best option for doing visualization of 2D and 3D scalar and vector fields in Python programs is Mayavi, which is an interface to the high-quality package VTK in C++. There is good online documentation and also an introduction in Chapter 5 of (Langtangen 2016a).

To obtain Mayavi on Ubuntu platforms you can write

```
pip install mayavi --upgrade
```

For Mac OS X and Windows, we recommend using Anaconda. To obtain Mayavi for Anaconda you can write

```
conda install mayavi
```

Mayavi has a MATLAB-like interface called `mlab`. We can do

```
import mayavi.mlab as plt
from mayavi import mlab
```

and have `plt` (as usual) or `mlab` as a kind of MATLAB visualization access inside our program (just more powerful and with higher visual quality).

The official documentation of the `mlab` module is provided in two places, one for the basic functionality and one for further functionality. Basic figure handling is very similar to the one we know from Matplotlib. Just as for Matplotlib, all plotting commands you do in `mlab` will go into the same figure, until you manually change to a new figure.

Back to our application, the following code for the user action function with plotting in Mayavi is relevant to add.

```
try:
    import mayavi.mlab as mlab
except:
    pass

def solver(...):
    ...

def gaussian(...):
    ...
    if plot_method == 3:
        from mpl_toolkits.mplot3d import axes3d
        import matplotlib.pyplot as plt
        from matplotlib import cm
        plt.ion()
        fig = plt.figure()
        u_surf = None

    def plot_u(u, x, xv, y, yv, t, n):
        """User action function for plotting."""
        if t[n] == 0:
            time.sleep(2)
        if plot_method == 1:
            st.mesh(x, y, u, title='t=%g' % t[n], zlim=[-1,1],
                    caxis=[-1,1])
        elif plot_method == 2:
            st.surfc(xv, yv, u, title='t=%g' % t[n], zlim=[-1, 1],
                  colorbar=True, colormap=st.hot(), caxis=[-1,1],
                  shading='flat')
        elif plot_method == 3:
            print 'Experimental 3D matplotlib...not recommended'
        elif plot_method == 4:
            mlab.clf()
            extent1 = (0, 20, 0, 20,-2, 2)
            s = mlab.surf(x , y, u,
                          colormap='Blues',
                          warp_scale=5,extent=extent1)
            mlab.axes(s, color=(.7, .7, .7), extent=extent1,
                      ranges=(0, 10, 0, 10, -1, 1),
                      xlabel='', ylabel='', zlabel='',
                      x_axis_visibility=False,
                      z_axis_visibility=False)
            mlab.outline(s, color=(0.7, .7, .7), extent=extent1)
            mlab.text(6, -2.5, '', z=-4, width=0.14)
            mlab.colorbar(object=None, title=None,
                          orientation='horizontal',
                          nb_labels=None, nb_colors=None,
```

```
                        label_fmt=None)
            mlab.title('Gaussian t=%g' % t[n])
            mlab.view(142, -72, 50)
            f = mlab.gcf()
            camera = f.scene.camera
            camera.yaw(0)

        if plot_method > 0:
            time.sleep(0) # pause between frames
            if save_plot:
                filename = 'tmp_%04d.png' % n
        if plot_method == 4:
                    mlab.savefig(filename)  # time consuming!
        elif plot_method in (1,2):
                    st.savefig(filename)  # time consuming!
```

This is a point to get started – visualization is as always a very time-consuming and experimental discipline. With the PNG files we can use `ffmpeg` to create videos.

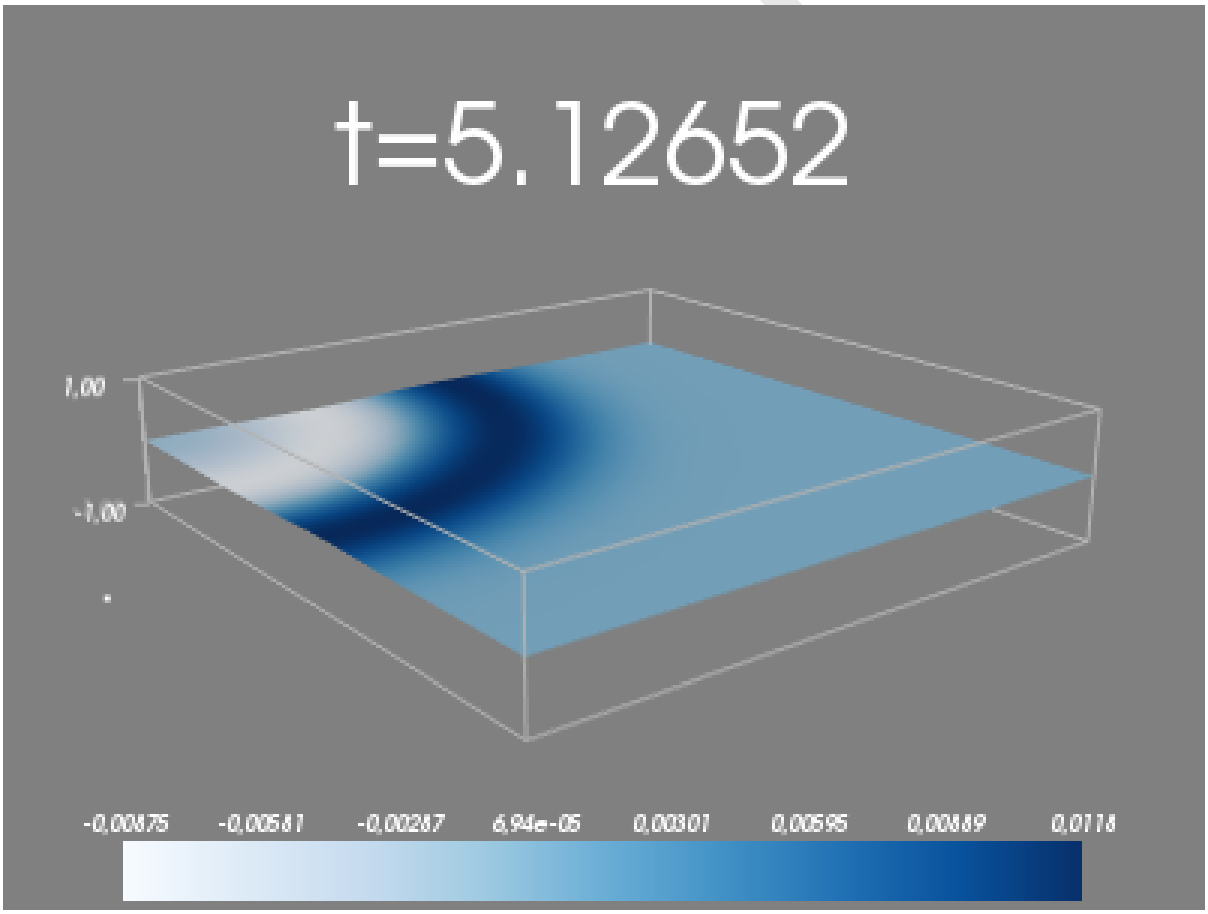


Figure 2.9.: Plot with Mayavi.

MOVIE: [https://github.com/hplgit/fdm-book/blob/master/doc/pub/book/html/mov-wave/mayavi/wave2D_u0_gaussian/movie.mp4]

2.76. Exercise: Check that a solution fulfills the discrete model

Carry out all mathematical details to show that (2.113) is indeed a solution of the discrete model for a 2D wave equation with $u = 0$ on the boundary. One must check the boundary conditions, the initial conditions, the general discrete equation at a time level and the special version of this equation for the first time level.

2.77. Project: Calculus with 2D mesh functions

The goal of this project is to redo Project Section 2.34 with 2D mesh functions ($f_{i,j}$).

Differentiation. The differentiation results in a discrete gradient function, which in the 2D case can be represented by a three-dimensional array `df[d,i,j]` where `d` represents the direction of the derivative, and `i,j` is a mesh point in 2D. Use centered differences for the derivative at inner points and one-sided forward or backward differences at the boundary points. Construct unit tests and write a corresponding test function.

Integration. The integral of a 2D mesh function $f_{i,j}$ is defined as

$$F_{i,j} = \int_{y_0}^{y_j}\int_{x_0}^{x_i} f(x,y)dxdy,$$

where $f(x,y)$ is a function that takes on the values of the discrete mesh function $f_{i,j}$ at the mesh points, but can also be evaluated in between the mesh points. The particular variation between mesh points can be taken as bilinear, but this is not important as we will use a product Trapezoidal rule to approximate the integral over a cell in the mesh and then we only need to evaluate $f(x,y)$ at the mesh points.

Suppose $F_{i,j}$ is computed. The calculation of $F_{i+1,j}$ is then

$$\begin{aligned} F_{i+1,j} &= F_{i,j} + \int_{x_i}^{x_{i+1}}\int_{y_0}^{y_j} f(x,y)dydx \\ &\approx \Delta x\frac{1}{2}\left(\int_{y_0}^{y_j} f(x_i,y)dy + \int_{y_0}^{y_j} f(x_{i+1},y)dy\right) \end{aligned}$$

The integrals in the y direction can be approximated by a Trapezoidal rule. A similar idea can be used to compute $F_{i,j+1}$. Thereafter, $F_{i+1,j+1}$ can be computed by adding the integral over the final corner cell to $F_{i+1,j} + F_{i,j+1} - F_{i,j}$. Carry out the details of these computations and implement a function that can return $F_{i,j}$ for all mesh indices i and j. Use the fact that the Trapezoidal rule is exact for linear functions and write a test function.

2.78. Exercise: Implement Neumann conditions in 2D

Modify the wave2D_u0.py program, which solves the 2D wave equation $u_{tt} = c^2(u_{xx} + u_{yy})$ with constant wave velocity c and $u = 0$ on the boundary, to have Neumann boundary conditions: $\partial u/\partial n = 0$. Include both scalar code (for debugging and reference) and vectorized code (for speed).

To test the code, use $u = 1.2$ as solution ($I(x, y) = 1.2$, $V = f = 0$, and c arbitrary), which should be exactly reproduced with any mesh as long as the stability criterion is satisfied. Another test is to use the plug-shaped pulse in the `pulse` function from Section Section 2.49 and the wave1D_dn_vc.py program. This pulse is exactly propagated in 1D if $c\Delta t/\Delta x = 1$. Check that also the 2D program can propagate this pulse exactly in x direction ($c\Delta t/\Delta x = 1$, Δy arbitrary) and y direction ($c\Delta t/\Delta y = 1$, Δx arbitrary).

2.79. Exercise: Test the efficiency of compiled loops in 3D

Extend the `wave2D_u0.py` code and the Cython, Fortran, and C versions to 3D. Set up an efficiency experiment to determine the relative efficiency of pure scalar Python code, vectorized code, Cython-compiled loops, Fortran-compiled loops, and C-compiled loops. Normalize the CPU time for each mesh by the fastest version.

2.80. Applications of wave equations

This section presents a range of wave equation models for different physical phenomena. Although many wave motion problems in physics can be modeled by the standard linear wave equation, or a similar formulation with a system of first-order equations, there are some exceptions. Perhaps the most important is water waves: these are modeled by the Laplace equation with time-dependent boundary conditions at the water surface (long water waves, however, can be approximated by a standard wave equation, see Section Section 2.87). Quantum mechanical waves constitute another example where the waves are governed by the Schrödinger equation, i.e., not by a standard wave equation. Many wave phenomena also need to take nonlinear effects into account when the wave amplitude is significant. Shock waves in the air is a primary example.

The derivations in the following are very brief. Those with a firm background in continuum mechanics will probably have enough knowledge to fill in the details, while other readers will hopefully get some impression of the physics and approximations involved when establishing wave equation models.

2.81. Waves on a string

Figure Figure 2.10 shows a model we may use to derive the equation for waves on a string. The string is modeled as a set of discrete point masses (at mesh points) with elastic strings in between. The string has a large constant tension T. We let the mass at mesh point x_i be m_i. The displacement of this mass point in the y direction is denoted by $u_i(t)$.

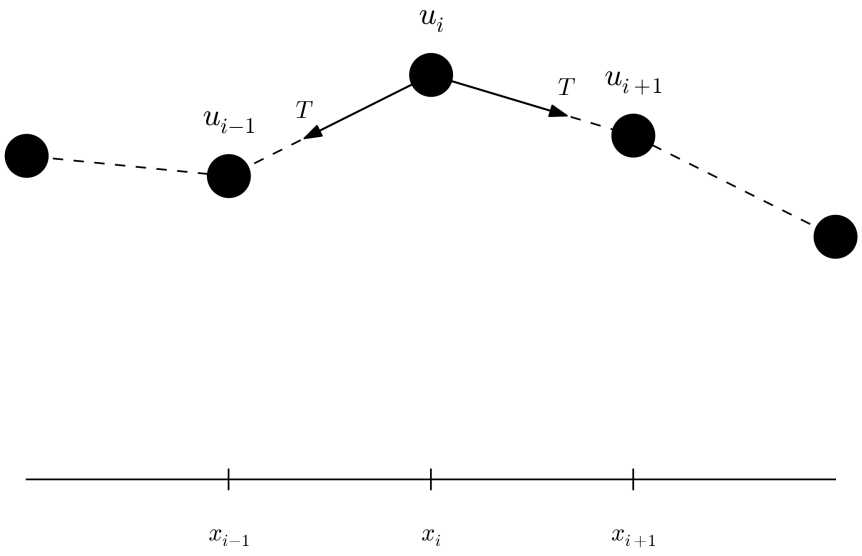


Figure 2.10.: Discrete string model with point masses connected by elastic strings.

The motion of mass m_i is governed by Newton's second law of motion. The position of the mass at time t is $x_i\boldsymbol{i} + u_i(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y direction, respectively. The acceleration is then $u_i''(t)\boldsymbol{j}$. Two forces are acting on the mass as indicated in Figure Figure 2.10. The force $\boldsymbol{T}^-$ acting toward the point x_{i-1} can be decomposed as

$$\boldsymbol{T}^- = -T\sin\phi\boldsymbol{i} - T\cos\phi\boldsymbol{j},$$

where ϕ is the angle between the force and the line $x = x_i$. Let $\Delta u_i = u_i - u_{i-1}$ and let $\Delta s_i = \sqrt{\Delta u_i^2 + (x_i - x_{i-1})^2}$ be the distance from mass m_{i-1} to mass m_i. It is seen that $\cos\phi = \Delta u_i/\Delta s_i$ and $\sin\phi = (x_i - x_{i-1})/\Delta s$ or $\Delta x/\Delta s_i$ if we introduce a constant mesh spacing $\Delta x = x_i - x_{i-1}$. The force can then be written

$$\boldsymbol{T}^- = -T\frac{\Delta x}{\Delta s_i}\boldsymbol{i} - T\frac{\Delta u_i}{\Delta s_i}\boldsymbol{j}\,.$$

The force $\boldsymbol{T}^+$ acting toward x_{i+1} can be calculated in a similar way:

$$\boldsymbol{T}^+ = T\frac{\Delta x}{\Delta s_{i+1}}\boldsymbol{i} + T\frac{\Delta u_{i+1}}{\Delta s_{i+1}}\boldsymbol{j}\,.$$

Newton's second law becomes

$$m_i u_i''(t)\boldsymbol{j} = \boldsymbol{T}^+ + \boldsymbol{T}^-,$$

which gives the component equations

$$T\frac{\Delta x}{\Delta s_i} = T\frac{\Delta x}{\Delta s_{i+1}}, \tag{2.114}$$

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta s_{i+1}} - T\frac{\Delta u_i}{\Delta s_i}\,. \tag{2.115}$$

A basic reasonable assumption for a string is small displacements u_i and small displacement gradients $\Delta u_i/\Delta x$. For small $g = \Delta u_i/\Delta x$ we have that

$$\Delta s_i = \sqrt{\Delta u_i^2 + \Delta x^2} = \Delta x\sqrt{1+g^2} + \Delta x(1 + \frac{1}{2}g^2 + \mathcal{O}(g^4)) \approx \Delta x\,.$$

Equation (2.114) is then simply the identity $T = T$, while (2.115) can be written as

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta x} - T\frac{\Delta u_i}{\Delta x},$$

which upon division by Δx and introducing the density $\varrho_i = m_i/\Delta x$ becomes

$$\varrho_i u_i''(t) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,. \tag{2.116}$$

We can now choose to approximate u_i'' by a finite difference in time and get the discretized wave equation,

$$\varrho_i\frac{1}{\Delta t^2}\left(u_i^{n+1} - 2u_i^n - u_i^{n-1}\right) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,.$$

On the other hand, we may go to the continuum limit $\Delta x \to 0$ and replace $u_i(t)$ by $u(x,t)$, ϱ_i by $\varrho(x)$, and recognize that the right-hand side of (2.116) approaches $\partial^2 u/\partial x^2$ as $\Delta x \to 0$. We end up with the continuous model for waves on a string:

$$\varrho\frac{\partial^2 u}{\partial t^2} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{2.117}$$

Note that the density ϱ may change along the string, while the tension T is a constant. With variable wave velocity $c(x) = \sqrt{T/\varrho(x)}$ we can write the wave equation in the more standard form

$$\frac{\partial^2 u}{\partial t^2} = c^2(x)\frac{\partial^2 u}{\partial x^2}\,. \tag{2.118}$$

Because of the way ϱ enters the equations, the variable wave velocity does *not* appear inside the derivatives as in many other versions of the wave equation. However, most strings of interest have constant ϱ.

The end points of a string are fixed so that the displacement u is zero. The boundary conditions are therefore $u = 0$.

2.81.1. Damping

Air resistance and non-elastic effects in the string will contribute to reduce the amplitudes of the waves so that the motion dies out after some time. This damping effect can be modeled by a term bu_t on the left-hand side of the equation

$$\varrho\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{2.119}$$

The parameter $b \geq 0$ is small for most wave phenomena, but the damping effect may become significant in long time simulations.

2.81.2. External forcing

It is easy to include an external force acting on the string. Say we have a vertical force $\tilde{f}_i\boldsymbol{j}$ acting on mass m_i, modeling the effect of gravity on a string. This force affects the vertical component of Newton's law and gives rise to an extra term $\tilde{f}(x,t)$ on the right-hand side of (2.117). In the model (2.118) we would add a term $f(x,t) = \tilde{f}(x,t)/\varrho(x)$.

2.81.3. Modeling the tension via springs

We assumed, in the derivation above, that the tension in the string, T, was constant. It is easy to check this assumption by modeling the string segments between the masses as standard springs, where the force (tension T) is proportional to the elongation of the spring segment. Let k be the spring constant, and set $T_i = k\Delta\ell$ for the tension in the spring segment between x_{i-1} and x_i, where $\Delta\ell$ is the elongation of this segment from the tension-free state. A basic feature of a string is that it has high tension in the equilibrium position $u = 0$. Let the string segment have an elongation $\Delta\ell_0$ in the equilibrium position. After deformation of the string, the elongation is $\Delta\ell = \Delta\ell_0 + \Delta s_i$: $T_i = k(\Delta\ell_0 + \Delta s_i) \approx k(\Delta\ell_0 + \Delta x)$. This shows that T_i is independent of i. Moreover, the extra approximate elongation Δx is very small compared to $\Delta\ell_0$, so we may well set $T_i = T = k\Delta\ell_0$. This means that the tension is completely dominated by the initial tension determined by the tuning of the string. The additional deformations of the spring during the vibrations do not introduce significant changes in the tension.

2.82. Elastic waves in a rod

Consider an elastic rod subject to a hammer impact at the end. This experiment will give rise to an elastic deformation pulse that travels through the rod. A mathematical model for longitudinal waves along an elastic rod starts with the general equation for deformations and stresses in an elastic medium,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{2.120}$$

where ϱ is the density, $\boldsymbol{u}$ the displacement field, $\boldsymbol{\sigma}$ the stress tensor, and $\boldsymbol{f}$ body forces. The latter has normally no impact on elastic waves.

For stationary deformation of an elastic rod, aligned with the x axis, one has that $\sigma_{xx} = Eu_x$, with all other stress components being zero. The parameter E is known as Young's modulus. Moreover, we set $\boldsymbol{u} = u(x,t)\boldsymbol{i}$ and neglect the radial contraction and expansion (where Poisson's ratio is the important parameter). Assuming that this simple stress and deformation field is a good approximation, (2.120) simplifies to

$$\varrho \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(E\frac{\partial u}{\partial x}\right). \tag{2.121}$$

The associated boundary conditions are u or $\sigma_{xx} = Eu_x$ known, typically $u = 0$ for a fixed end and $\sigma_{xx} = 0$ for a free end.

2.83. Waves on a membrane

Think of a thin, elastic membrane with shape as a circle or rectangle. This membrane can be brought into oscillatory motion and will develop elastic waves. We can model this phenomenon somewhat similar to waves in a rod: waves in a membrane are simply the two-dimensional counterpart. We assume that the material is deformed in the z direction only and write the elastic displacement field on the form $\boldsymbol{u}(x,y,t) = w(x,y,t)\boldsymbol{i}$. The z coordinate is omitted since the membrane is thin and all properties are taken as constant throughout the thickness. Inserting this displacement field in Newton's 2nd law of motion (2.120) results in

$$\varrho \frac{\partial^2 w}{\partial t^2} = \frac{\partial}{\partial x}\left(\mu\frac{\partial w}{\partial x}\right) + \frac{\partial}{\partial y}\left(\mu\frac{\partial w}{\partial y}\right). \tag{2.122}$$

This is nothing but a wave equation in $w(x,y,t)$, which needs the usual initial conditions on w and w_t as well as a boundary condition $w = 0$. When computing the stress in the membrane, one needs to split $\boldsymbol{\sigma}$ into a constant high-stress component due to the fact that all membranes are normally pre-stressed, plus a component proportional to the displacement and governed by the wave motion.

2.84. The acoustic model for seismic waves

Seismic waves are used to infer properties of subsurface geological structures. The physical model is a heterogeneous elastic medium where sound is propagated by small elastic vibrations. The general

mathematical model for deformations in an elastic medium is based on Newton's second law,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{2.123}$$

and a constitutive law relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$, often Hooke's generalized law,

$$\boldsymbol{\sigma} = K\nabla \cdot \boldsymbol{u}\, \boldsymbol{I} + G(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3}\nabla \cdot \boldsymbol{u}\, \boldsymbol{I})\,. \tag{2.124}$$

Here, $\boldsymbol{u}$ is the displacement field, $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{I}$ is the identity tensor, ϱ is the medium's density, $\boldsymbol{f}$ are body forces (such as gravity), K is the medium's bulk modulus and G is the shear modulus. All these quantities may vary in space, while $\boldsymbol{u}$ and $\boldsymbol{\sigma}$ will also show significant variation in time during wave motion.

The acoustic approximation to elastic waves arises from a basic assumption that the second term in Hooke's law, representing the deformations that give rise to shear stresses, can be neglected. This assumption can be interpreted as approximating the geological medium by a fluid. Neglecting also the body forces $\boldsymbol{f}$, (2.123) becomes

$$\varrho \boldsymbol{u}_{tt} = \nabla(K\nabla \cdot \boldsymbol{u}) \tag{2.125}$$

Introducing p as a pressure via

$$p = -K\nabla \cdot \boldsymbol{u}, \tag{2.126}$$

and dividing (2.125) by ϱ, we get

$$\boldsymbol{u}_{tt} = -\frac{1}{\varrho}\nabla p\,.$$

Taking the divergence of this equation, using $\nabla \cdot \boldsymbol{u} = -p/K$ from (2.126), gives the *acoustic approximation to elastic waves*:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right)\,. \tag{2.127}$$

This is a standard, linear wave equation with variable coefficients. It is common to add a source term $s(x, y, z, t)$ to model the generation of sound waves:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) + s\,. \tag{2.128}$$

A common additional approximation of (2.128) is based on using the chain rule on the right-hand side,

$$K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) = \frac{K}{\varrho}\nabla^2 p + K\nabla\left(\frac{1}{\varrho}\right) \cdot \nabla p \approx \frac{K}{\varrho}\nabla^2 p,$$

under the assumption that the relative spatial gradient $\nabla \varrho^{-1} = -\varrho^{-2}\nabla\varrho$ is small. This approximation results in the simplified equation

$$p_{tt} = \frac{K}{\varrho}\nabla^2 p + s\,. \tag{2.129}$$

The acoustic approximations to seismic waves are used for sound waves in the ground, and the Earth's surface is then a boundary where p equals the atmospheric pressure p_0 such that the boundary condition becomes $p = p_0$.

2.84.1. Anisotropy

Quite often in geological materials, the effective wave velocity $c = \sqrt{K/\varrho}$ is different in different spatial directions because geological layers are compacted, and often twisted, in such a way that the properties in the horizontal and vertical direction differ. With z as the vertical coordinate, we can introduce a vertical wave velocity c_z and a horizontal wave velocity c_h, and generalize (2.129) to

$$p_{tt} = c_z^2 p_{zz} + c_h^2(p_{xx} + p_{yy}) + s\,. \tag{2.130}$$

2.85. Sound waves in liquids and gases

Sound waves arise from pressure and density variations in fluids. The starting point of modeling sound waves is the basic equations for a compressible fluid where we omit viscous (frictional) forces, body forces (gravity, for instance), and temperature effects:

$$\varrho_t + \nabla\cdot(\varrho\boldsymbol{u}) = 0, \tag{2.131}$$

$$\varrho\boldsymbol{u}_t + \varrho\boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p, \tag{2.132}$$

$$\varrho = \varrho(p)\,. \tag{2.133}$$

These equations are often referred to as the Euler equations for the motion of a fluid. The parameters involved are the density ϱ, the velocity $\boldsymbol{u}$, and the pressure p. Equation (2.131) reflects mass balance, (2.132) is Newton's second law for a fluid, with frictional and body forces omitted, and (2.133) is a constitutive law relating density to pressure by thermodynamic considerations. A typical model for (2.133) is the so-called isentropic relation, valid for adiabatic processes where there is no heat transfer:

$$\varrho = \varrho_0\left(\frac{p}{p_0}\right)^{1/\gamma}\,. \tag{2.134}$$

Here, p_0 and ϱ_0 are reference values for p and ϱ when the fluid is at rest, and γ is the ratio of specific heat at constant pressure and constant volume ($\gamma = 5/3$ for air).

The key approximation in a mathematical model for sound waves is to assume that these waves are small perturbations to the density, pressure, and velocity. We therefore write

$$\begin{aligned} p &= p_0 + \hat{p},\\ \varrho &= \varrho_0 + \hat{\varrho},\\ \boldsymbol{u} &= \hat{\boldsymbol{u}}, \end{aligned}$$

where we have decomposed the fields in a constant equilibrium value, corresponding to $\boldsymbol{u} = 0$, and a small perturbation marked with a hat symbol. By inserting these decompositions in (2.131) and (2.132), neglecting all product terms of small perturbations and/or their derivatives, and dropping the hat symbols, one gets the following linearized PDE system for the small perturbations in density, pressure, and velocity:

$$\varrho_t + \varrho_0\nabla\cdot\boldsymbol{u} = 0,$$

$$\varrho_0 \boldsymbol{u}_t = -\nabla p\,.$$

Now we can eliminate ϱ_t by differentiating the relation $\varrho(p)$,

$$\varrho_t = \varrho_0 \frac{1}{\gamma}\left(\frac{p}{p_0}\right)^{1/\gamma-1}\frac{1}{p_0}p_t = \frac{\varrho_0}{\gamma p_0}\left(\frac{p}{p_0}\right)^{1/\gamma-1}p_t\,.$$

The product term $p^{1/\gamma-1}p_t$ can be linearized as $p_0^{1/\gamma-1}p_t$, resulting in

$$\varrho_t \approx \frac{\varrho_0}{\gamma p_0}p_t\,.$$

We then get

$$p_t + \gamma p_0 \nabla\cdot\boldsymbol{u} = 0, \tag{2.135}$$

$$\boldsymbol{u}_t = -\frac{1}{\varrho_0}\nabla p\,. \tag{2.136}$$

Taking the divergence of (2.136) and differentiating (2.135) with respect to time gives the possibility to easily eliminate $\nabla\cdot\boldsymbol{u}_t$ and arrive at a standard, linear wave equation for p:

$$p_{tt} = c^2\nabla^2 p,$$

where $c = \sqrt{\gamma p_0/\varrho_0}$ is the speed of sound in the fluid.

2.86. Spherical waves

Spherically symmetric three-dimensional waves propagate in the radial direction r only so that $u = u(r,t)$. The fully three-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(c^2\nabla u) + f$$

then reduces to the spherically symmetric wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial r}\right) + f(r,t), \quad r\in(0,R),\ t>0\,.$$

One can easily show that the function $v(r,t) = ru(r,t)$ fulfills a standard wave equation in Cartesian coordinates if c is constant. To this end, insert $u = v/r$ in

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial r}\right)$$

to obtain

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + c^2\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(c^2\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one Cartesian coordinate. The spherically symmetric wave equation in terms of $v(r,t)$ now becomes

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial r}\left(c^2(r)\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{dc^2}{dr}v + rf(r,t), \quad r \in (0,R),\ t > 0\,.$$

In the case of constant wave velocity c, this equation reduces to the wave equation in a single Cartesian coordinate called r:

$$\frac{\partial^2 v}{\partial t^2} = c^2\frac{\partial^2 v}{\partial r^2} + rf(r,t), \quad r \in (0,R),\ t > 0\,. \tag{2.137}$$

That is, any program for solving the one-dimensional wave equation in a Cartesian coordinate system can be used to solve (2.137), provided the source term is multiplied by the coordinate, and that we divide the Cartesian mesh solution by r to get the spherically symmetric solution. Moreover, if $r = 0$ is included in the domain, spherical symmetry demands that $\partial u/\partial r = 0$ at $r = 0$, which means that

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r = 0\,.$$

For this to hold in the limit $r \to 0$, we must have $v(0,t) = 0$ at least as a necessary condition. In most practical applications, we exclude $r = 0$ from the domain and assume that some boundary condition is assigned at $r = \epsilon$, for some $\epsilon > 0$.

2.87. The linear shallow water equations

The next example considers water waves whose wavelengths are much larger than the depth and whose wave amplitudes are small. This class of waves may be generated by catastrophic geophysical events, such as earthquakes at the sea bottom, landslides moving into water, or underwater slides (or a combination, as earthquakes frequently release avalanches of masses). For example, a subsea earthquake will normally have an extension of many kilometers but lift the water only a few meters. The wave length will have a size dictated by the earthquake area, which is much lager than the water depth, and compared to this wave length, an amplitude of a few meters is very small. The water is essentially a thin film, and mathematically we can average the problem in the vertical direction and approximate the 3D wave phenomenon by 2D PDEs. Instead of a moving water domain in three space dimensions, we get a horizontal 2D domain with an unknown function for the surface elevation and the water depth as a variable coefficient in the PDEs.

Let $\eta(x,y,t)$ be the elevation of the water surface, $H(x,y)$ the water depth corresponding to a flat surface ($\eta = 0$), $u(x,y,t)$ and $v(x,y,t)$ the depth-averaged horizontal velocities of the water. Mass and momentum balance of the water volume give rise to the PDEs involving these quantities:

$$\eta_t = -(Hu)_x - (Hv)_x \tag{2.138}$$

$$u_t = -g\eta_x, \tag{2.139}$$

$$v_t = -g\eta_y, \tag{2.140}$$

where g is the acceleration of gravity. Equation (2.138) corresponds to mass balance while the other two are derived from momentum balance (Newton's second law).

The initial conditions associated with (2.138)-(2.140) are η, u, and v prescribed at $t = 0$. A common condition is to have some water elevation $\eta = I(x, y)$ and assume that the surface is at rest: $u = v = 0$. A subsea earthquake usually means a sufficiently rapid motion of the bottom and the water volume to say that the bottom deformation is mirrored at the water surface as an initial lift $I(x, y)$ and that $u = v = 0$.

Boundary conditions may be η prescribed for incoming, known waves, or zero normal velocity at reflecting boundaries (steep mountains, for instance): $un_x + vn_y = 0$, where (n_x, n_y) is the outward unit normal to the boundary. More sophisticated boundary conditions are needed when waves run up at the shore, and at open boundaries where we want the waves to leave the computational domain undisturbed.

Equations (2.138), (2.139), and (2.140) can be transformed to a standard, linear wave equation. First, multiply (2.139) and (2.140) by H, differentiate (2.139)) with respect to x and (2.140) with respect to y. Second, differentiate (2.138) with respect to t and use that $(Hu)_{xt} = (Hu_t) * *x$ and $(Hv) * *yt = (Hv_t)_y$ when H is independent of t. Third, eliminate $(Hu_t)_x$ and $(Hv_t)_y$ with the aid of the other two differentiated equations. These manipulations result in a standard, linear wave equation for η:

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla \cdot (gH\nabla\eta)\,. \tag{2.141}$$

In the case we have an initial non-flat water surface at rest, the initial conditions become $\eta = I(x, y)$ and $\eta_t = 0$. The latter follows from (2.138) if $u = v = 0$, or simply from the fact that the vertical velocity of the surface is η_t, which is zero for a surface at rest.

The system (2.138)-(2.140) can be extended to handle a time-varying bottom topography, which is relevant for modeling long waves generated by underwater slides. In such cases the water depth function H is also a function of t, due to the moving slide, and one must add a time-derivative term H_t to the left-hand side of (2.138). A moving bottom is best described by introducing $z = H_0$ as the still-water level, $z = B(x, y, t)$ as the time- and space-varying bottom topography, so that $H = H_0 - B(x, y, t)$. In the elimination of u and v one may assume that the dependence of H on t can be neglected in the terms $(Hu) * *xt$ and $(Hv) * *yt$. We then end up with a source term in (2.141), because of the moving (accelerating) bottom:

$$\eta_{tt} = \nabla \cdot (gH\nabla\eta) + B_{tt}\,. \tag{2.142}$$

The reduction of (2.142) to 1D, for long waves in a straight channel, or for approximately plane waves in the ocean, is trivial by assuming no change in y direction ($\partial/\partial y = 0$):

$$\eta_{tt} = (gH\eta_x) * *x + B * *tt\,. \tag{2.143}$$

2.87.1. Wind drag on the surface

Surface waves are influenced by the drag of the wind, and if the wind velocity some meters above the surface is (U, V), the wind drag gives contributions $C_V\sqrt{U^2 + V^2}U$ and $C_V\sqrt{U^2 + V^2}V$ to (2.139) and (2.140), respectively, on the right-hand sides.

2.87.2. Bottom drag

The waves will experience a drag from the bottom, often roughly modeled by a term similar to the wind drag: $C_B\sqrt{u^2+v^2}u$ on the right-hand side of (2.139) and $C_B\sqrt{u^2+v^2}v$ on the right-hand side of (2.140). Note that in this case the PDEs (2.139) and (2.140) become nonlinear and the elimination of u and v to arrive at a 2nd-order wave equation for η is not possible anymore.

2.87.3. Effect of the Earth's rotation

Long geophysical waves will often be affected by the rotation of the Earth because of the Coriolis force. This force gives rise to a term fv on the right-hand side of (2.139) and $-fu$ on the right-hand side of (2.140). Also in this case one cannot eliminate u and v to work with a single equation for η. The Coriolis parameter is $f = 2\Omega\sin\phi$, where Ω is the angular velocity of the earth and ϕ is the latitude.

2.88. Waves in blood vessels

The flow of blood in our bodies is basically fluid flow in a network of pipes. Unlike rigid pipes, the walls in the blood vessels are elastic and will increase their diameter when the pressure rises. The elastic forces will then push the wall back and accelerate the fluid. This interaction between the flow of blood and the deformation of the vessel wall results in waves traveling along our blood vessels.

A model for one-dimensional waves along blood vessels can be derived from averaging the fluid flow over the cross section of the blood vessels. Let x be a coordinate along the blood vessel and assume that all cross sections are circular, though with different radii $R(x,t)$. The main quantities to compute is the cross section area $A(x,t)$, the averaged pressure $P(x,t)$, and the total volume flux $Q(x,t)$. The area of this cross section is

$$A(x,t) = 2\pi\int_0^{R(x,t)} r dr,$$

Let $v_x(x,t)$ be the velocity of blood averaged over the cross section at point x. The volume flux, being the total volume of blood passing a cross section per time unit, becomes

$$Q(x,t) = A(x,t)v_x(x,t)$$

Mass balance and Newton's second law lead to the PDEs

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{2.144}$$

$$\frac{\partial Q}{\partial t} + \frac{\gamma+2}{\gamma+1}\frac{\partial}{\partial x}\left(\frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} = -2\pi(\gamma+2)\frac{\mu}{\varrho}\frac{Q}{A}, \tag{2.145}$$

where γ is a parameter related to the velocity profile, ϱ is the density of blood, and μ is the dynamic viscosity of blood.

We have three unknowns A, Q, and P, and two equations (2.144) and (2.145). A third equation is needed to relate the flow to the deformations of the wall. A common form for this equation is

$$\frac{\partial P}{\partial t} + \frac{1}{C}\frac{\partial Q}{\partial x} = 0, \tag{2.146}$$

where C is the compliance of the wall, given by the constitutive relation

$$C = \frac{\partial A}{\partial P} + \frac{\partial A}{\partial t},$$

which requires a relationship between A and P. One common model is to view the vessel wall, locally, as a thin elastic tube subject to an internal pressure. This gives the relation

$$P = P_0 + \frac{\pi h E}{(1-\nu^2)A_0}(\sqrt{A} - \sqrt{A_0}),$$

where P_0 and A_0 are corresponding reference values when the wall is not deformed, h is the thickness of the wall, and E and ν are Young's modulus and Poisson's ratio of the elastic material in the wall. The derivative becomes

$$C = \frac{\partial A}{\partial P} = \frac{2(1-\nu^2)A_0}{\pi h E}\sqrt{A_0} + 2\left(\frac{(1-\nu^2)A_0}{\pi h E}\right)^2 (P - P_0)\,.$$

Another (nonlinear) deformation model of the wall, which has a better fit with experiments, is

$$P = P_0 \exp\left(\beta(A/A_0 - 1)\right),$$

where β is some parameter to be estimated. This law leads to

$$C = \frac{\partial A}{\partial P} = \frac{A_0}{\beta P}\,.$$

Reduction to the standard wave equation. It is not uncommon to neglect the viscous term on the right-hand side of (2.145) and also the quadratic term with Q^2 on the left-hand side. The reduced equations (2.145) and (2.146) form a first-order linear wave equation system:

$$C\frac{\partial P}{\partial t} = -\frac{\partial Q}{\partial x},$$
$$\frac{\partial Q}{\partial t} = -\frac{A}{\varrho}\frac{\partial P}{\partial x}\,.$$

These can be combined into standard 1D wave PDE by differentiating the first equation with respect to t and the second with respect to x,

$$\frac{\partial}{\partial t}\left(C\frac{\partial P}{\partial t}\right) = \frac{\partial}{\partial x}\left(\frac{A}{\varrho}\frac{\partial P}{\partial x}\right),$$

which can be approximated by

$$\frac{\partial^2 Q}{\partial t^2} = c^2\frac{\partial^2 Q}{\partial x^2}, \quad c = \sqrt{\frac{A}{\varrho C}},$$

where the A and C in the expression for c are taken as constant reference values.

2.89. Electromagnetic waves

Light and radio waves are governed by standard wave equations arising from Maxwell's general equations. When there are no charges and no currents, as in a vacuum, Maxwell's equations take the form

$$\begin{aligned}\nabla \cdot \boldsymbol{E} &= 0,\\ \nabla \cdot \boldsymbol{B} &= 0,\\ \nabla \times \boldsymbol{E} &= -\frac{\partial \boldsymbol{B}}{\partial t},\\ \nabla \times \boldsymbol{B} &= \mu_0\epsilon_0\frac{\partial \boldsymbol{E}}{\partial t},\end{aligned}$$

where $\epsilon_0 = 8.854187817620 \cdot 10^{-12}$ (F/m) is the permittivity of free space, also known as the electric constant, and $\mu_0 = 1.2566370614 \cdot 10^{-6}$ (H/m) is the permeability of free space, also known as the magnetic constant. Taking the curl of the two last equations and using the mathematical identity

$$\nabla \times (\nabla \times \boldsymbol{E}) = \nabla(\nabla \cdot \boldsymbol{E}) + \nabla^2\boldsymbol{E} = -\nabla^2\boldsymbol{E} \text{ when } \nabla \cdot \boldsymbol{E} = 0,$$

gives the wave equation governing the electric and magnetic field:

$$\frac{\partial^2 \boldsymbol{E}}{\partial t^2} = c^2\nabla^2\boldsymbol{E}, \tag{2.147}$$

$$\frac{\partial^2 \boldsymbol{B}}{\partial t^2} = c^2\nabla^2\boldsymbol{B}, \tag{2.148}$$

with $c = 1/\sqrt{\mu_0\epsilon_0}$ as the velocity of light. Each component of $\boldsymbol{E}$ and $\boldsymbol{B}$ fulfills a wave equation and can hence be solved independently.

2.90. Exercise: Simulate waves on a non-homogeneous string

Simulate waves on a string that consists of two materials with different density. The tension in the string is constant, but the density has a jump at the middle of the string. Experiment with different sizes of the jump and produce animations that visualize the effect of the jump on the wave motion.

According to Section Section 2.81,

the density enters the mathematical model as ϱ in $\varrho u_{tt} = Tu_{xx}$, where T is the string tension. Modify, e.g., the `wave1D_u0v.py` code to incorporate the tension and two density values. Make a mesh function `rho` with density values at each spatial mesh point. A value for the tension may be 150 N. Corresponding density values can be computed from the wave velocity estimations in the `guitar` function in the `wave1D_u0v.py` file.

2.91. Exercise: Simulate damped waves on a string

Formulate a mathematical model for damped waves on a string. Use data from Section Section 2.20, and tune the damping parameter so that the string is very close to the rest state after 15 s. Make a movie of the wave motion.

2.92. Exercise: Simulate elastic waves in a rod

A hammer hits the end of an elastic rod. The exercise is to simulate the resulting wave motion using the model (2.121) from Section Section 2.82. Let the rod have length L and let the boundary $x = L$ be stress free so that $\sigma_{xx} = 0$, implying that $\partial u/\partial x = 0$. The left end $x = 0$ is subject to a strong stress pulse (the hammer), modeled as

$$\sigma_{xx}(t) = \begin{cases} S, & 0 < t \leq t_s, \\ 0, & t > t_s \end{cases}$$

The corresponding condition on u becomes $u_x = S/E$ for $t \leq t_s$ and zero afterwards (recall that $\sigma_{xx} = Eu_x$). This is a non-homogeneous Neumann condition, and you will need to approximate this condition and combine it with the scheme (the ideas and manipulations follow closely the handling of a non-zero initial condition $u_t = V$ in wave PDEs or the corresponding second-order ODEs for vibrations).

2.93. Exercise: Simulate spherical waves

Implement a model for spherically symmetric waves using the method described in Section Section 2.86. The boundary condition at $r = 0$ must be $\partial u/\partial r = 0$, while the condition at $r = R$ can either be $u = 0$ or a radiation condition as described in Problem Section 2.57. The $u = 0$ condition is sufficient if R is so large that the amplitude of the spherical wave has become insignificant. Make movie(s) of the case where the source term is located around $r = 0$ and sends out pulses

$$f(r,t) = \begin{cases} Q\exp\left(-\frac{r^2}{2\Delta r^2}\right)\sin\omega t, & \sin\omega t \geq 0 \\ 0, & \sin\omega t < 0 \end{cases}$$

Here, Q and ω are constants to be chosen.

Use the program `wave1D_u0v.py` as a starting point. Let `solver`

compute the v function and then set $u = v/r$. However, $u = v/r$ for $r = 0$ requires special treatment. One possibility is to compute `u[1:] = v[1:]/r[1:]` and then set `u[0]=u[1]`. The latter makes it evident that $\partial u/\partial r = 0$ in a plot.

2.94. Problem: Earthquake-generated tsunami over a subsea hill

A subsea earthquake leads to an immediate lift of the water surface, see Figure Figure 2.11. The lifted water surface splits into two tsunamis, one traveling to the right and one to the left, as depicted in Figure Figure 2.12. Since tsunamis are normally very long waves, compared to the depth, with a small amplitude, compared to the wave length, a standard wave equation is relevant:

$$\eta_{tt} = (gH(x)\eta_x)_x,$$

where η is the elevation of the water surface, g is the acceleration of gravity, and $H(x)$ is the still water depth.

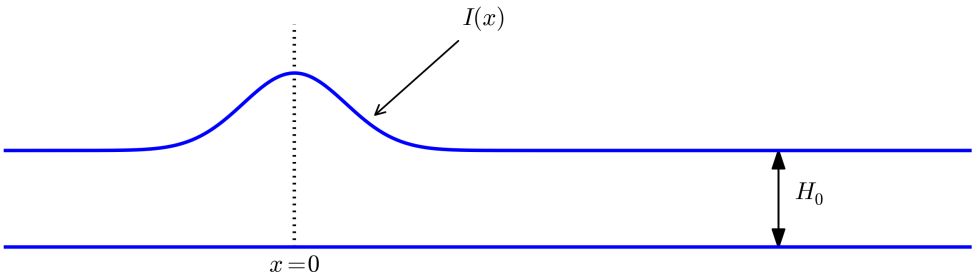


Figure 2.11.: Sketch of initial water surface due to a subsea earthquake.

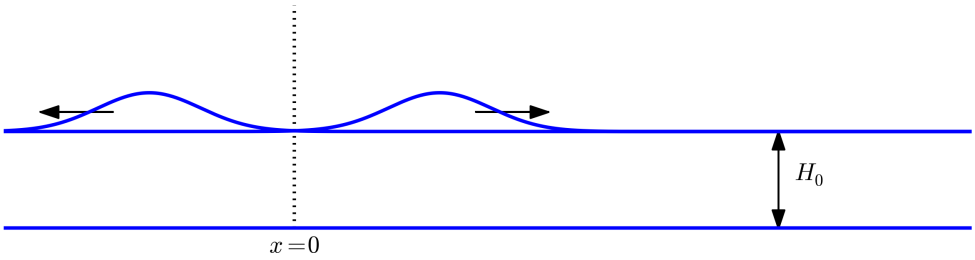


Figure 2.12.: An initial surface elevation is split into two waves.

To simulate the right-going tsunami, we can impose a symmetry boundary at $x = 0$: $\partial\eta/\partial x = 0$. We then simulate the wave motion in $[0, L]$. Unless the ocean ends at $x = L$, the waves should travel undisturbed through the boundary $x = L$. A radiation condition as explained in Problem

Section 2.57 can be used for this purpose. Alternatively, one can just stop the simulations before the wave hits the boundary at $x = L$. In that case it does not matter what kind of boundary condition we use at $x = L$. Imposing $\eta = 0$ and stopping the simulations when $|\eta_i^n| > \epsilon$, $i = N_x - 1$, is a possibility (ϵ is a small parameter).

The shape of the initial surface can be taken as a Gaussian function,

$$I(x; I_0, I_a, I_m, I_s) = I_0 + I_a \exp\left(-\left(\frac{x - I_m}{I_s}\right)^2\right),$$

with $I_m = 0$ reflecting the location of the peak of $I(x)$ and I_s being a measure of the width of the function $I(x)$ (I_s is $\sqrt{2}$ times the standard deviation of the familiar normal distribution curve).

Now we extend the problem with a hill at the sea bottom, see Figure Figure 2.13. The wave speed $c = \sqrt{gH(x)} = \sqrt{g(H_0 - B(x))}$ will then be reduced in the shallow water above the hill.

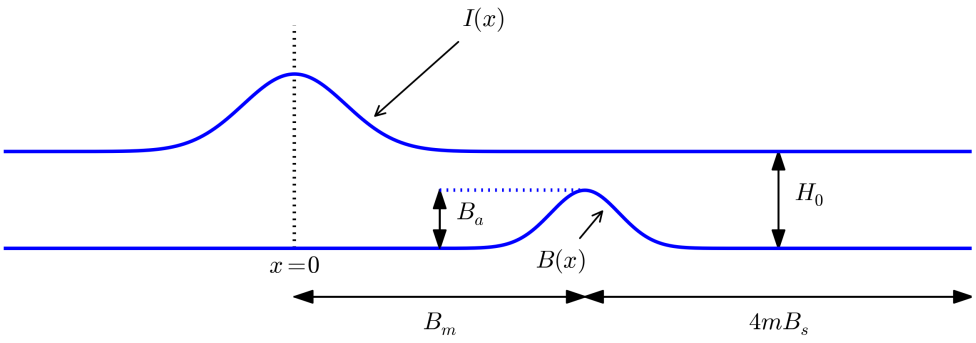


Figure 2.13.: Sketch of an earthquake-generated tsunami passing over a subsea hill.

One possible form of the hill is a Gaussian function,

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \exp\left(-\left(\frac{x - B_m}{B_s}\right)^2\right), \tag{2.149}$$

but many other shapes are also possible, e.g., a "cosine hat" where

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_m}{2B_s}\right), \tag{2.150}$$

when $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

Also an abrupt construction may be tried:

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a, \tag{2.151}$$

for $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

The wave1D_dn_vc.py program can be used as starting point for the implementation. Visualize both the bottom topography and the water surface elevation in the same plot. Allow for a flexible choice of bottom shape: (2.149), (2.150), (2.151), or $B(x) = B_0$ (flat).

The purpose of this problem is to explore the quality of the numerical solution η_i^n for different shapes of the bottom obstruction. The "cosine hat" and the box-shaped hills have abrupt changes in the derivative of $H(x)$ and are more likely to generate numerical noise than the smooth Gaussian shape of the hill. Investigate if this is true.

2.95. Problem: Earthquake-generated tsunami over a 3D hill

This problem extends Problem Section 2.94 to a three-dimensional wave phenomenon, governed by the 2D PDE

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla\cdot(gH\nabla\eta)\,. \tag{2.152}$$

We assume that the earthquake arises from a fault along the line $x = 0$ in the xy-plane so that the initial lift of the surface can be taken as $I(x)$ in Problem Section 2.94. That is, a plane wave is propagating to the right, but will experience bending because of the bottom.

The bottom shape is now a function of x and y. An "elliptic" Gaussian function in two dimensions, with its peak at (B_{mx}, B_{my}), generalizes (2.149):

$$B = B_0 + B_a \exp\left(-\left(\frac{x - B_{mx}}{B_s}\right)^2 - \left(\frac{y - B_{my}}{bB_s}\right)^2\right), \tag{2.153}$$

where b is a scaling parameter: $b = 1$ gives a circular Gaussian function with circular contour lines, while $b \neq 1$ gives an elliptic shape with elliptic contour lines. To indicate the input parameters in the model, we may write

$$B = B(x; B_0, B_a, B_{mx}, B_{my}, B_s, b)\,.$$

The "cosine hat" (2.150) can also be generalized to

$$B = B_0 + B_a \cos\left(\pi\frac{x - B_{mx}}{2B_s}\right)\cos\left(\pi\frac{y - B_{my}}{2B_s}\right), \tag{2.154}$$

when $0 \leq \sqrt{x^2 + y^2} \leq B_s$ and $B = B_0$ outside this circle.

A box-shaped obstacle means that

$$B(x; B_0, B_a, B_m, B_s, b) = B_0 + B_a \tag{2.155}$$

for x and y inside a rectangle

$$B_{mx} - B_s \leq x \leq B_{mx} + B_s, \quad B_{my} - bB_s \leq y \leq B_{my} + bB_s,$$

and $B = B_0$ outside this rectangle. The b parameter controls the rectangular shape of the cross section of the box.

Note that the initial condition and the listed bottom shapes are symmetric around the line $y = B_{my}$. We therefore expect the surface elevation also to be symmetric with respect to this line. This means that we can halve the computational domain by working with $[0, L_x] \times [0, B_{my}]$. Along the

upper boundary, $y = B_{my}$, we must impose the symmetry condition $\partial\eta/\partial n = 0$. Such a symmetry condition ($-\eta_x = 0$) is also needed at the $x = 0$ boundary because the initial condition has a symmetry here. At the lower boundary $y = 0$ we also set a Neumann condition (which becomes $-\eta_y = 0$). The wave motion is to be simulated until the wave hits the reflecting boundaries where $\partial\eta/\partial n = \eta_x = 0$ (one can also set $\eta = 0$ - the particular condition does not matter as long as the simulation is stopped before the wave is influenced by the boundary condition).

Visualize the surface elevation. Investigate how different hill shapes, different sizes of the water gap above the hill, and different resolutions $\Delta x = \Delta y = h$ and Δt influence the numerical quality of the solution.

2.96. Problem: Investigate Mayavi for visualization

Play with Mayavi code for visualizing 2D solutions of the wave equation with variable wave velocity. See if there are effective ways to visualize both the solution and the wave velocity scalar field at the same time.

2.97. Problem: Investigate visualization packages

Create some fancy 3D visualization of the water waves *and* the subsea hill in Problem Section 2.95. Try to make the hill transparent. Possible visualization tools are Mayavi, Paraview, and OpenDX.

2.98. Problem: Implement loops in compiled languages

Extend the program from Problem Section 2.95 such that the loops over mesh points, inside the time loop, are implemented in compiled languages. Consider implementations in Cython, Fortran via `f2py`, C via Cython, C via `f2py`, C/C++ via Instant, and C/C++ via `scipy.weave`. Perform efficiency experiments to investigate the relative performance of the various implementations. It is often advantageous to normalize CPU times by the fastest method on a given mesh.

2.99. Exercise: Simulate seismic waves in 2D

The goal of this exercise is to simulate seismic waves using the PDE model (2.130) in a 2D xz domain with geological layers. Introduce m horizontal layers of thickness h_i, $i = 0, \ldots, m-1$. Inside layer number i we have a vertical wave velocity $c_{z,i}$ and a horizontal wave velocity $c_{h,i}$. Make a program for simulating such 2D waves. Test it on a case with 3 layers where

$$c_{z,0} = c_{z,1} = c_{z,2}, \quad c_{h,0} = c_{h,2}, \quad c_{h,1} \ll c_{h,0}\,.$$

Let s be a localized point source at the middle of the Earth's surface (the upper boundary) and investigate how the resulting wave travels through the medium. The source can be a localized Gaussian peak that oscillates in time for some time interval. Place the boundaries far enough from the expanding wave so that the boundary conditions do not disturb the wave. Then the type of

boundary condition does not matter, except that we physically need to have $p = p_0$, where p_0 is the atmospheric pressure, at the upper boundary.

2.100. Project: Model 3D acoustic waves in a room

The equation for sound waves in air is derived in Section Section 2.85 and reads

$$p_{tt} = c^2\nabla^2 p,$$

where $p(x, y, z, t)$ is the pressure and c is the speed of sound, taken as 340 m/s. However, sound is absorbed in the air due to relaxation of molecules in the gas. A model for simple relaxation, valid for gases consisting only of one type of molecules, is a term $c^2\tau_s\nabla^2 p_t$ in the PDE, where τ_s is the relaxation time. If we generate sound from, e.g., a loudspeaker in the room, this sound source must also be added to the governing equation.

The PDE with the mentioned type of damping and source then becomes

$$p_t t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f,$$

where $f(x, y, z, t)$ is the source term.

The walls can absorb some sound. A possible model is to have a "wall layer" (thicker than the physical wall) outside the room where c is changed such that some of the wave energy is reflected and some is absorbed in the wall. The absorption of energy can be taken care of by adding a damping term bp_t in the equation:

$$p_t t + bp_t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f\,.$$

Typically, $b = 0$ in the room and $b > 0$ in the wall. A discontinuity in b or c will give rise to reflections. It can be wise to use a constant c in the wall to control reflections because of the discontinuity between c in the air and in the wall, while b is gradually increased as we go into the wall to avoid reflections because of rapid changes in b. At the outer boundary of the wall the condition $p = 0$ or $\partial p/\partial n = 0$ can be imposed. The waves should anyway be approximately dampened to $p = 0$ this far out in the wall layer.

There are two strategies for discretizing the $\nabla^2 p_t$ term: using a center difference between times $n + 1$ and $n - 1$ (if the equation is sampled at level n), or use a one-sided difference based on levels n and $n - 1$. The latter has the advantage of not leading to any equation system, while the former is second-order accurate as the scheme for the simple wave equation $p_t t = c^2\nabla^2 p$. To avoid an equation system, go for the one-sided difference such that the overall scheme becomes explicit and only of first order in time.

Develop a 3D solver for the specified PDE and introduce a wall layer. Test the solver with the method of manufactured solutions. Make some demonstrations where the wall reflects and absorbs the waves (reflection because of discontinuity in b and absorption because of growing b). Experiment with the impact of the τ_s parameter.

2.101. Project: Solve a 1D transport equation

We shall study the wave equation

$$u_t + cu_x = 0, \quad x \in (0, L],\ t \in (0, T], \tag{2.156}$$

with initial condition

$$u(x, 0) = I(x), \quad x \in [0, L],$$

and *one* periodic boundary condition

$$u(0, t) = u(L, t)\,.$$

This boundary condition means that what goes out of the domain at $x = L$ comes in at $x = 0$. Roughly speaking, we need only one boundary condition because the spatial derivative is of first order only.

Physical interpretation. The parameter c can be constant or variable, $c = c(x)$. The equation (2.156) arises in *transport* problems where a quantity u, which could be temperature or concentration of some contaminant, is transported with the velocity c of a fluid. In addition to the transport imposed by "travelling with the fluid'', u may also be transported by diffusion (such as heat conduction or Fickian diffusion), but we have in the model $u_t + cu_x$ assumed that diffusion effects are negligible, which they often are.

a)

Show that under the assumption of $a = \text{const}$,

$$u(x, t) = I(x - ct) \tag{2.157}$$

fulfills the PDE as well as the initial and boundary condition (provided $I(0) = I(L)$).

A widely used numerical scheme for (2.156) applies a forward difference in time and a backward difference in space when $c > 0$:

$$[D_t^+ u + cD_x^- u = 0]_i^n\,. \tag{2.158}$$

For $c < 0$ we use a forward difference in space: $[cD_x^+ u]_i^n$.

b)

Set up a computational algorithm and implement it in a function. Assume a is constant and positive.

c)

Test the implementation by using the remarkable property that the numerical solution is exact at the mesh points if $\Delta t = c^{-1}\Delta x$.

d)

Make a movie comparing the numerical and exact solution for the following two choices of initial conditions:

$$I(x) = \left[\sin\left(\pi\frac{x}{L}\right)\right]^{2n} \tag{2.159}$$

where n is an integer, typically $n = 5$, and

$$I(x) = \exp\left(-\frac{(x - L/2)^2}{2\sigma 2}\right). \tag{2.160}$$

Choose $\Delta t = c^{-1}\Delta x, 0.9c^{-1}\Delta x, 0.5c^{-1}\Delta x$.

e)

The performance of the suggested numerical scheme can be investigated by analyzing the numerical dispersion relation. Analytically, we have that the *Fourier component*

$$u(x,t) = e^{i(kx-\omega t)},$$

is a solution of the PDE if $\omega = kc$. This is the *analytical dispersion relation*. A complete solution of the PDE can be built by adding up such Fourier components with different amplitudes, where the initial condition I determines the amplitudes. The solution u is then represented by a Fourier series.

A similar discrete Fourier component at (x_p, t_n) is

$$u_p^q = e^{i(kp\Delta x - \tilde{\omega} n\Delta t)},$$

where in general $\tilde{\omega}$ is a function of k, Δt, and Δx, and differs from the exact $\omega = kc$.

Insert the discrete Fourier component in the numerical scheme and derive an expression for $\tilde{\omega}$, i.e., the discrete dispersion relation. Show in particular that if $\Delta t/(c\Delta x) = 1$, the discrete solution coincides with the exact solution at the mesh points, regardless of the mesh resolution (!). Show that if the stability condition

$$\frac{\Delta t}{c\Delta x} \leq 1,$$

the discrete Fourier component cannot grow (i.e., $\tilde{\omega}$ is real).

f)

Write a test for your implementation where you try to use information from the numerical dispersion relation.

We shall hereafter assume that $c(x) > 0$.

g)

Set up a computational algorithm for the variable coefficient case and implement it in a function. Make a test that the function works for constant a.

h)

It can be shown that for an observer moving with velocity $c(x)$, u is constant. This can be used to derive an exact solution when a varies with x. Show first that

$$u(x,t) = f(C(x) - t), \tag{2.161}$$

where

$$C'(x) = \frac{1}{c(x)},$$

is a solution of (2.156) for any differentiable function f.

Solution

Let $\xi = C(x) - t$. We have that

$$u_t = f'(\xi)(-1),$$

while

$$u_x = f'(\xi)C'(x) = f'(\xi)\frac{1}{c(x)},$$

implying that $au_x = f'(\xi)$. Then we have $u_t + cu_x = -f'(\xi) + f'(\xi) = 0$.

i)

Use the initial condition to show that an exact solution is

$$u(x,t) = I(C^{-1}(C(x) - t)),$$

with C^{-1} being the inverse function of $C = \int c^1 dx$. Since $C(x)$ is an integral $\int_0^x (1/c)dx$, $C(x)$ is monotonically increasing and there exists hence an inverse function C^{-1} with values in $[0, L]$.

Solution

In general we have $u(x,t) = f(C(x) - t)$ and the solution is of this form with $f(\xi) = I(C^{-1}(\xi))$. Moreover, at $t = 0$ we have $I(C^{-1}(C(x))) = I(x)$, which is the required initial condition.

To compute (2.161) we need to integrate $1/c$ to obtain C and then compute the inverse of C.

The inverse function computation can be easily done if we first think discretely. Say we have some function $y = g(x)$ and seek its inverse. Plotting (x_i, y_i), where $y_i = g(x_i)$ for some mesh points x_i, displays g as a function of x. The inverse function is simply x as a function of g, i.e., the curve with points (y_i, x_i). We can therefore quickly compute points at the curve of the inverse function. One way of extending these points to a continuous function is to assume a linear variation (known as linear interpolation) between the points (which actually means to draw straight lines between the points, exactly as done by a plotting program).

The function `scipy.interpolate.interp1d` can take a set of points and return a continuous function that corresponds to linear variation between the points. The computation of the inverse of a function g on $[0, L]$ can then be done by

```
def inverse(g, domain, resolution=101):
    x = linspace(domain[0], domain[L], resolution)
    y = g(x)
    from scipy.interpolate import interp1d
    g_inverse = interp1d(y, x, kind='linear', fill_value='extrapolate')
    return g_inverse
```

To compute $C(x)$ we need to integrate $1/c$, which can be done by a Trapezoidal rule. Suppose we have computed $C(x_i)$ and need to compute $C(x_{i+1})$. Using the Trapezoidal rule with m subintervals over the integration domain $[x_i, x_{i+1}]$ gives

$$C(x_{i+1}) = C(x_i) + \int_{x_i}^{x_{i+1}} \frac{dx}{c} \approx h\left(\frac{1}{2}\frac{1}{c(x_i)} + \frac{1}{2}\frac{1}{c(x_{i+1})} + \sum_{j=1}^{m-1}\frac{1}{c(x_i + jh)}\right), \tag{2.162}$$

where $h = (x_{i+1} - x_i)/m$ is the length of the subintervals used for the integral over $[x_i, x_{i+1}]$. We observe that (2.162) is a *difference equation* which we can solve by repeatedly applying (2.162) for $i = 0, 1, \ldots, N_x - 1$ if a mesh $x_0, x, \ldots, x_{N_x}$ is prescribed. Note that $C(0) = 0$.

j)

Implement a function for computing $C(x_i)$ and one for computing $C^{-1}(x)$ for any x. Use these two functions for computing the exact solution $I(C^{-1}(C(x) - t))$. End up with a function `u_exact_variable_c(x, n, c, I)` that returns the value of $I(C^{-1}(C(x) - t_n))$.

k)

Make movies showing a comparison of the numerical and exact solutions for the two initial conditions (1) and (2.160). Choose $\Delta t = \Delta x / \max_{0,L} c(x)$ and the velocity of the medium as

1. $c(x) = 1 + \epsilon \sin(k\pi x/L)$, $\epsilon < 1$,
2. $c(x) = 1 + I(x)$, where I is given by

(1) or (2.160).

The PDE $u_t + cu_x = 0$ expresses that the initial condition $I(x)$ is transported with velocity $c(x)$.

2.102. Problem: General analytical solution of a 1D damped wave equation

2.103. For solution, see damped_wave_equation.pdf in joakibo on bitbucket.

We consider an initial-boundary value problem for the damped wave equation:

$$
\begin{aligned}
u_{tt} + bu_t &= c^2 u_{xx}, \quad x \in (0, L),\ t \in (0, T] \\
u(0, t) &= 0, \\
u(L, t) &= 0, \\
u(x, 0) &= I(x), \\
u_t(x, 0) &= V(x)\,.
\end{aligned}
$$

Here, $b \geq 0$ and c are given constants. The aim is to derive a general analytical solution of this problem. Familiarity with the method of separation of variables for solving PDEs will be assumed.

a)

Seek a solution on the form $u(x, t) = X(x)T(t)$. Insert this solution in the PDE and show that it leads to two differential equations for X and T:

$$
T'' + bT' + \lambda T = 0, \quad c^2 X'' + \lambda X = 0,
$$

with $X(0) = X(L) = 0$ as boundary conditions, and λ as a constant to be determined.

b)

Show that $X(x)$ is on the form

$$X_n(x) = C_n \sin kx, \quad k = \frac{n\pi}{L}, \quad n = 1, 2, \ldots$$

where C_n is an arbitrary constant.

c)

Under the assumption that $(b/2)^2 < k^2$, show that $T(t)$ is on the form

$$T_n(t) = e^{-\frac{1}{2}bt}(a_n \cos \omega t + b_n \sin \omega t), \quad \omega = \sqrt{k^2 - \frac{1}{4}b^2}, \quad n = 1, 2, \ldots$$

The complete solution is then

$$u(x,t) = \sum_{n=1}^{\infty} \sin kx e^{-\frac{1}{2}bt}(A_n \cos \omega t + B_n \sin \omega t),$$

where the constants A_n and B_n must be computed from the initial conditions.

d)

Derive a formula for A_n from $u(x,0) = I(x)$ and developing $I(x)$ as a sine Fourier series on $[0, L]$.

e)

Derive a formula for B_n from $u_t(x,0) = V(x)$ and developing $V(x)$ as a sine Fourier series on $[0, L]$.

f)

Calculate A_n and B_n from vibrations of a string where $V(x) = 0$ and

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases}$$

g)

Implement a function `u_series(x, t, tol=1E-10)` for the series for $u(x,t)$, where `tol` is a tolerance for truncating the series. Simply sum the terms until $|a_n|$ and $|b_b|$ both are less than `tol`.

h)

What will change in the derivation of the analytical solution if we have $u_x(0,t) = u_x(L,t) = 0$ as boundary conditions? And how will you solve the problem with $u(0,t) = 0$ and $u_x(L,t) = 0$?

2.104. Problem: General analytical solution of a 2D damped wave equation

Carry out Problem Section 2.102 in the 2D case: $u_{tt} + bu_t = c^2(u_{xx} + u_{yy})$, where $(x,y) \in (0, L_x) \times (0, L_y)$. Assume a solution on the form $u(x,y,t) = X(x)Y(y)T(t)$.

2.105. Exercises: Wave Equations with Devito

These exercises explore wave equation solutions using the Devito DSL. They progress from basic usage through verification techniques to more advanced applications.

2.105.1. Exercise 1: Standing Wave Simulation

Use the `solve_wave_1d` function to simulate a standing wave with:

- Domain: $L = 1$, wave speed $c = 1$
- Initial condition: $I(x) = \sin(2\pi x)$ (two half-wavelengths)
- Initial velocity: $V = 0$
- Boundary conditions: $u(0, t) = u(1, t) = 0$

a) Compute and plot the solution at $t = 0, 0.25, 0.5, 0.75, 1.0$. How does the pattern differ from the fundamental mode?

b) Derive the exact solution for this initial condition and compare with the numerical solution. Compute the maximum error at $t = 1$ for $N_x = 50, 100, 200$.

Solution

```python
from src.wave import solve_wave_1d, exact_standing_wave
import numpy as np
import matplotlib.pyplot as plt

# Part (a)
def I(x):
    return np.sin(2 * np.pi * x)

times = [0, 0.25, 0.5, 0.75, 1.0]
fig, axes = plt.subplots(1, 5, figsize=(15, 3))

for ax, T in zip(axes, times):
    result = solve_wave_1d(L=1.0, c=1.0, Nx=100, T=T, C=0.9, I=I)
    ax.plot(result.x, result.u)
    ax.set_title(f't = {T}')
    ax.set_ylim(-1.2, 1.2)

plt.tight_layout()

# Part (b) - The exact solution for m=2 mode
# u(x,t) = sin(2*pi*x) * cos(2*pi*t)
def u_exact(x, t):
    return np.sin(2 * np.pi * x) * np.cos(2 * np.pi * t)

for Nx in [50, 100, 200]:
    result = solve_wave_1d(L=1.0, c=1.0, Nx=Nx, T=1.0, C=0.9, I=I)
    error = np.abs(result.u - u_exact(result.x, 1.0)).max()
    print(f"Nx = {Nx:3d}: max error = {error:.2e}")
```

2.105.2. Exercise 2: Convergence Rate Verification

The theoretical convergence rate for the wave equation solver is $O(\Delta t^2 + \Delta x^2) = O(h^2)$ when $\Delta t \propto \Delta x$.

a) Use `convergence_test_wave_1d` with grid sizes $N_x = 20, 40, 80, 160, 320$ and verify the observed rate is close to 2.

b) Repeat with Courant number $C = 1$. What happens to the errors? Explain why.

Solution

```python
from src.wave import convergence_test_wave_1d
import numpy as np

# Part (a)
grid_sizes, errors, rate = convergence_test_wave_1d(
    grid_sizes=[20, 40, 80, 160, 320],
    T=0.5,
    C=0.9,
)
print(f"C = 0.9: Observed rate = {rate:.3f}")

# Compute individual rates
for i in range(1, len(errors)):
    r = np.log(errors[i-1] / errors[i]) / np.log(2)
    print(f"  Nx {grid_sizes[i-1]} -> {grid_sizes[i]}: rate = {r:.3f}")

# Part (b)
grid_sizes, errors, rate = convergence_test_wave_1d(
    grid_sizes=[20, 40, 80, 160, 320],
    T=0.5,
    C=1.0,
)
print(f"\nC = 1.0: Observed rate = {rate:.3f}")
print(f"Errors: {errors}")

# At C=1, the numerical method is exact for the standing wave!
# Errors should be near machine precision.
```

2.105.3. Exercise 3: Guitar String

Simulate a plucked guitar string with a triangular initial shape:

$$I(x) = \begin{cases} ax/x_0 & x < x_0 \\ a(L-x)/(L-x_0) & x \geq x_0 \end{cases}$$

where $L = 0.75$ m, $x_0 = 0.8L$, and $a = 0.005$ m.

a) For a guitar with fundamental frequency 440 Hz, compute the wave speed c given that $\lambda = 2L$.

b) Simulate one complete period and create an animation. Does the triangular shape remain sharp as time progresses?

c) Run with $C = 1$ and observe the difference. Explain why the result is different.

Solution

```python
from src.wave import solve_wave_1d
import numpy as np

# Parameters
L = 0.75
x0 = 0.8 * L
a = 0.005
freq = 440  # Hz

# Part (a)
wavelength = 2 * L
c = freq * wavelength
print(f"Wave speed c = {c} m/s")

# Period
period = 1 / freq
print(f"Period = {period*1000:.3f} ms")

# Part (b)
def I(x):
    return np.where(x < x0, a * x / x0, a * (L - x) / (L - x0))

result = solve_wave_1d(
    L=L, c=c, Nx=150, T=period,
    C=0.9, I=I, save_history=True
)

# The triangular shape becomes "wavy" due to numerical dispersion
# Different Fourier components travel at slightly different speeds

# Part (c)
result_exact = solve_wave_1d(
    L=L, c=c, Nx=150, T=period,
    C=1.0, I=I, save_history=True
)

# At C=1, D'Alembert's solution is exactly reproduced:
# The triangular pulse splits into two, bounces off walls, and
# recombines after one period to give the original shape.
```

2.105.4. Exercise 4: Source Wavelets

a) Use `ricker_wavelet` to create wavelets with peak frequencies $f_0 = 10, 25, 50$ Hz. Plot them and their frequency spectra.

b) What is the relationship between f_0 and the dominant wavelength λ in a medium with $c = 1500$ m/s?

c) For seismic imaging, we typically want the wavelet to have negligible amplitude at $t = 0$. What constraint does this place on t_0 relative to f_0?

Solution

```python
from src.wave import ricker_wavelet, get_source_spectrum
import numpy as np
import matplotlib.pyplot as plt

# Part (a)
t = np.linspace(0, 0.5, 1001)
dt = t[1] - t[0]

fig, axes = plt.subplots(2, 3, figsize=(12, 6))

for i, f0 in enumerate([10, 25, 50]):
    wavelet = ricker_wavelet(t, f0=f0)
    freq, amp = get_source_spectrum(wavelet, dt)

    axes[0, i].plot(t, wavelet)
    axes[0, i].set_title(f'f0 = {f0} Hz')
    axes[0, i].set_xlabel('Time (s)')

    axes[1, i].plot(freq[:100], amp[:100])
    axes[1, i].axvline(f0, color='r', linestyle='--')
    axes[1, i].set_xlabel('Frequency (Hz)')

# Part (b)
c = 1500  # m/s
for f0 in [10, 25, 50]:
    wavelength = c / f0
    print(f"f0 = {f0} Hz: wavelength = {wavelength} m")

# Part (c)
# The Ricker wavelet is centered at t0, and has amplitude ~0 when
# |t - t0| > 1/f0. For the wavelet to be ~0 at t=0, we need:
# t0 > 1/f0, typically t0 = 1.5/f0 is used as default
```

2.105.5. Exercise 5: 2D Wave Propagation

a) Solve the 2D wave equation with an initial Gaussian pulse centered at $(0.5, 0.5)$:

$$I(x, y) = e^{-100((x-0.5)^2+(y-0.5)^2)}$$

Plot the solution at $t = 0, 0.1, 0.2, 0.3$ using contour plots.

b) How does the wave pattern differ from the 1D case? Explain the amplitude decay you observe.

Solution

```python
from src.wave import solve_wave_2d
import numpy as np
import matplotlib.pyplot as plt

# Part (a)
def I(X, Y):
    return np.exp(-100 * ((X - 0.5)**2 + (Y - 0.5)**2))

fig, axes = plt.subplots(1, 4, figsize=(16, 4))

for ax, T in zip(axes, [0, 0.1, 0.2, 0.3]):
    result = solve_wave_2d(
        Lx=1.0, Ly=1.0, Nx=100, Ny=100,
        T=T, C=0.5, I=I
    )

    X, Y = np.meshgrid(result.x, result.y, indexing='ij')
    c = ax.contourf(X, Y, result.u, levels=20, cmap='RdBu_r')
    ax.set_title(f't = {T}')
    ax.set_aspect('equal')

# Part (b)
# In 2D, the wave spreads as a circular wavefront. The amplitude
# decays as 1/sqrt(r) due to geometric spreading - the energy is
# distributed over an expanding circle rather than staying constant
# as in 1D.
```

2.105.6. Exercise 6: Reflection from Interface

Consider a 1D domain $[0, 2]$ with a velocity interface at $x = 1$: $c(x) = 1$ for $x < 1$ and $c(x) = 2$ for $x \geq 1$.

a) Starting with a Gaussian pulse centered at $x = 0.3$, simulate the wave propagation until $t = 2.0$.

b) Identify the reflected and transmitted waves. Do the amplitudes match the theoretical reflection ($R = 1/3$) and transmission ($T = 4/3$) coefficients?

c) What happens at the boundaries $x = 0$ and $x = 2$? Are there additional reflections?

Solution

```
# This requires implementing variable velocity, which is
# demonstrated in the wave1D_features.qmd chapter.
# A simplified approach using manual stencil computation:

import numpy as np
import matplotlib.pyplot as plt

L = 2.0
Nx = 400
dx = L / Nx
x = np.linspace(0, L, Nx + 1)

# Velocity profile
c = np.where(x < 1.0, 1.0, 2.0)
c_max = 2.0

# Time stepping
C = 0.5
dt = C * dx / c_max
T = 2.0
Nt = int(T / dt)

# Initial condition
sigma = 0.1
x0 = 0.3
u_nm1 = np.exp(-((x - x0) / sigma)**2)
u_n = u_nm1.copy()
u = np.zeros_like(u_n)

# Store snapshots
snapshots = []
times = []

for n in range(Nt):
    # Update interior
    for i in range(1, Nx):
        C_local = c[i] * dt / dx
        u[i] = 2*u_n[i] - u_nm1[i] + C_local**2 * (u_n[i+1] - 2*u_n[i] + u_n[i-1])

    # Dirichlet BCs
    u[0] = 0
    u[Nx] = 0

    # Swap
    u_nm1, u_n, u = u_n, u, u_nm1

    # Store snapshots
    if n % 50 == 0:
        snapshots.append(u_n.copy())
        times.append(n * dt)

# Plot snapshots
fig, axes = plt.subplots(2, 4, figsize=(16, 6))
```

2.105.7. Exercise 7: Manufactured Solution

Use the method of manufactured solutions to verify the solver. Choose $u(x,t) = x(L-x)(1+t/2)$ which satisfies zero Dirichlet boundary conditions.

a) Compute the required source term $f(x,t)$ and initial conditions $I(x)$, $V(x)$.

b) Modify the solver (or use the source term capability) to solve with this $f(x,t)$. Verify the numerical solution matches the exact solution to machine precision.

Solution

```
# Manufactured solution: u = x(L-x)(1 + t/2)
# u_t = x(L-x) * (1/2)
# u_tt = 0
# u_x = (L - 2x)(1 + t/2)
# u_xx = -2(1 + t/2)
#
# PDE: u_tt = c^2 * u_xx + f
# 0 = c^2 * (-2)(1 + t/2) + f
# f = 2*c^2*(1 + t/2)

L = 2.5
c = 1.5

def u_exact(x, t):
    return x * (L - x) * (1 + 0.5 * t)

def I(x):
    return u_exact(x, 0)

def V(x):
    return 0.5 * x * (L - x)

def f(x, t):
    return 2 * c**2 * (1 + 0.5 * t)

# The solution should be exact to machine precision because
# the discretization error is zero for quadratic solutions
```

2.105.8. Exercise 8: Wave Energy Conservation

The total energy of the wave system is:

$$E = \frac{1}{2}\int_0^L \left[u_t^2 + c^2 u_x^2\right] dx$$

a) Implement a function to compute the discrete energy at each time step.

b) Run a simulation with zero Dirichlet BCs and plot the energy versus time. Is energy conserved?

c) What happens to energy conservation if $C > 1$?

Solution

```python
from src.wave import solve_wave_1d
import numpy as np

def compute_energy(u_history, x, dt, c):
    """Compute discrete energy at each time step."""
    dx = x[1] - x[0]
    Nt = u_history.shape[0]
    energy = np.zeros(Nt)

    for n in range(1, Nt-1):
        # u_t approximation (central difference)
        u_t = (u_history[n+1] - u_history[n-1]) / (2 * dt)

        # u_x approximation
        u_x = np.zeros_like(u_history[n])
        u_x[1:-1] = (u_history[n, 2:] - u_history[n, :-2]) / (2 * dx)

        # Energy integral
        energy[n] = 0.5 * dx * np.sum(u_t**2 + c**2 * u_x**2)

    return energy

# Part (b)
result = solve_wave_1d(
    L=1.0, c=1.0, Nx=100, T=5.0, C=0.9,
    save_history=True
)

E = compute_energy(result.u_history, result.x, result.dt, 1.0)

import matplotlib.pyplot as plt
plt.plot(result.t_history[1:-1], E[1:-1])
plt.xlabel('Time')
plt.ylabel('Energy')
plt.title('Energy Conservation')
# Energy should be nearly constant for stable schemes

# Part (c)
# For C > 1, the scheme is unstable and energy grows exponentially
```

2.105.9. Exercise 9: Numerical Dispersion

The numerical scheme introduces dispersion: different frequencies travel at different speeds.

a) Create an initial condition with multiple frequencies:

$$I(x) = \sin(2\pi x) + 0.5\sin(6\pi x)$$

Simulate for several periods and observe how the shape changes.

b) Run the same simulation with $C = 1$. Is dispersion present?

Solution

```python
from src.wave import solve_wave_1d
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(2 * np.pi * x) + 0.5 * np.sin(6 * np.pi * x)

# Part (a) - C < 1: dispersion present
result_a = solve_wave_1d(
    L=1.0, c=1.0, Nx=100, T=10.0, C=0.8,
    I=I, save_history=True
)

# Part (b) - C = 1: no dispersion
result_b = solve_wave_1d(
    L=1.0, c=1.0, Nx=100, T=10.0, C=1.0,
    I=I, save_history=True
)

# Compare at final time
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(result_a.x, I(result_a.x), 'k--', label='Initial')
ax1.plot(result_a.x, result_a.u, 'r-', label=f't = {result_a.t}')
ax1.set_title('C = 0.8 (dispersion present)')
ax1.legend()

ax2.plot(result_b.x, I(result_b.x), 'k--', label='Initial')
ax2.plot(result_b.x, result_b.u, 'r-', label=f't = {result_b.t}')
ax2.set_title('C = 1.0 (dispersion-free)')
ax2.legend()

# At C = 1, the solution returns exactly to the initial shape
# after one period, while at C < 1, the shape is distorted.
```

2.105.10. Exercise 10: Extension to Higher Order

Devito supports higher-order spatial discretization through the `space_order` parameter.

a) Compare the errors at $t = 1$ for `space_order = 2, 4, 6, 8` with $N_x = 50$.

b) For which problems might higher spatial order be beneficial?

Solution

```
# Note: This requires modifying the solver to accept space_order
# as a parameter. The key change is:
#
# u = TimeFunction(name='u', grid=grid, time_order=2, space_order=order)
#
# Higher order gives more accurate spatial derivatives but
# requires wider stencils and more boundary handling.
#
# Higher order is beneficial when:
# 1. The solution is smooth
# 2. Long propagation distances are needed
# 3. Minimizing numerical dispersion is important
# 4. Fewer grid points are desired for a given accuracy
```

3. Diffusion Equations

The famous *diffusion equation*, also known as the *heat equation*, reads

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2},$$

where $u(x,t)$ is the unknown function to be solved for, x is a coordinate in space, and t is time. The coefficient α is the *diffusion coefficient* and determines how fast u changes in time. A quick short form for the diffusion equation is $u_t = \alpha u_{xx}$.

Compared to the wave equation, $u_{tt} = c^2 u_{xx}$, which looks very similar, the diffusion equation features solutions that are very different from those of the wave equation. Also, the diffusion equation makes quite different demands to the numerical methods.

Typical diffusion problems may experience rapid change in the very beginning, but then the evolution of u becomes slower and slower. The solution is usually very smooth, and after some time, one cannot recognize the initial shape of u. This is in sharp contrast to solutions of the wave equation where the initial shape is preserved in homogeneous media – the solution is then basically a moving initial condition. The standard wave equation $u_{tt} = c^2 u_{xx}$ has solutions that propagate with speed c forever, without changing shape, while the diffusion equation converges to a *stationary solution* $\bar{u}(x)$ as $t \to \infty$. In this limit, $u_t = 0$, and $\bar{u}$ is governed by $\bar{u}''(x) = 0$. This stationary limit of the diffusion equation is called the *Laplace* equation and arises in a very wide range of applications throughout the sciences.

It is possible to solve for $u(x,t)$ using an explicit scheme, as we do in Section Section 3.1, but the time step restrictions soon become much less favorable than for an explicit scheme applied to the wave equation. And of more importance, since the solution u of the diffusion equation is very smooth and changes slowly, small time steps are not convenient and not required by accuracy as the diffusion process converges to a stationary state. Therefore, implicit schemes (as described in Section Section 3.7) are popular, but these require solutions of systems of algebraic equations. We shall use ready-made software for this purpose, but also program some simple iterative methods. The exposition is, as usual in this book, very basic and focuses on the basic ideas and how to implement. More comprehensive mathematical treatments and classical analysis of the methods are found in lots of textbooks. A favorite of ours in this respect is the one by LeVeque (LeVeque 2007). The books by Strikwerda (Strikwerda 2007) and by Lapidus and Pinder (Lapidus and Pinder 1982) are also highly recommended as additional material on the topic.

3.1. An explicit method for the 1D diffusion equation

Explicit finite difference methods for the wave equation $u_{tt} = c^2 u_{xx}$ can be used, with small modifications, for solving $u_t = \alpha u_{xx}$ as well. The exposition below assumes that the reader is familiar with the basic ideas of discretization and implementation of wave equations from Chapter

Chapter 2. Readers not familiar with the Forward Euler, Backward Euler, and Crank-Nicolson (or centered or midpoint) discretization methods in time should consult, e.g., Section 1.1 in (Langtangen 2016b).

3.2. The initial-boundary value problem for 1D diffusion

To obtain a unique solution of the diffusion equation, or equivalently, to apply numerical methods, we need initial and boundary conditions. The diffusion equation goes with one initial condition $u(x,0) = I(x)$, where I is a prescribed function. One boundary condition is required at each point on the boundary, which in 1D means that u must be known, u_x must be known, or some combination of them.

We shall start with the simplest boundary condition: $u = 0$. The complete initial-boundary value diffusion problem in one space dimension can then be specified as

$$\begin{aligned} \frac{\partial u}{\partial t} &= \alpha \frac{\partial^2 u}{\partial x^2} + f, && x \in (0,L),\ t \in (0,T] \\ u(x,0) &= I(x), && x \in [0,L] \\ u(0,t) &= 0, && t > 0, \\ u(L,t) &= 0, && t > 0\,. \end{aligned} \tag{3.1}$$

With only a first-order derivative in time, only one *initial condition* is needed, while the second-order derivative in space leads to a demand for two *boundary conditions*. We have added a source term $f = f(x,t)$, which is convenient when testing implementations.

Diffusion equations like (3.1) have a wide range of applications throughout physical, biological, and financial sciences. One of the most common applications is propagation of heat, where $u(x,t)$ represents the temperature of some substance at point x and time t. Other applications are listed in Section Section 3.66.

3.3. Forward Euler scheme

The first step in the discretization procedure is to replace the domain $[0,L] \times [0,T]$ by a set of mesh points. Here we apply equally spaced mesh points

$$x_i = i\Delta x, \quad i = 0, \ldots, N_x,$$

and

$$t_n = n\Delta t, \quad n = 0, \ldots, N_t\,.$$

Moreover, u_i^n denotes the mesh function that approximates $u(x_i, t_n)$ for $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$. Requiring the PDE (3.1) to be fulfilled at a mesh point (x_i, t_n) leads to the equation

$$\frac{\partial}{\partial t} u(x_i, t_n) = \alpha \frac{\partial^2}{\partial x^2} u(x_i, t_n) + f(x_i, t_n), \tag{3.2}$$

The next step is to replace the derivatives by finite difference approximations. The computationally simplest method arises from using a forward difference in time and a central difference in space:

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n \,. \tag{3.3}$$

Written out,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \,. \tag{3.4}$$

We have turned the PDE into algebraic equations, also often called discrete equations. The key property of the equations is that they are algebraic, which makes them easy to solve. As usual, we anticipate that u_i^n is already computed such that u_i^{n+1} is the only unknown in (3.4). Solving with respect to this unknown is easy:

$$u_i^{n+1} = u_i^n + F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t f_i^n, \tag{3.5}$$

where we have introduced the *mesh Fourier number*:

$$F = \alpha \frac{\Delta t}{\Delta x^2} \,.$$

F is the key parameter in the discrete diffusion equation

Note that F is a *dimensionless* number that lumps the key physical parameter in the problem, α, and the discretization parameters Δx and Δt into a single parameter. Properties of the numerical method are critically dependent upon the value of F (see Section Section 3.15 for details).

The computational algorithm then becomes

1. compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. for $n = 0, 1, \ldots, N_t$: 1. apply (3.5) for all the internal spatial points $i = 1, \ldots, N_x - 1$ 1. set the boundary values $u_i^{n+1} = 0$ for $i = 0$ and $i = N_x$

The algorithm is compactly and fully specified in Python:

```
import numpy as np
x = np.linspace(0, L, Nx+1)     # mesh points in space
dx = x[1] - x[0]
t = np.linspace(0, T, Nt+1)     # mesh points in time
dt = t[1] - t[0]
F = a*dt/dx**2
u   = np.zeros(Nx+1)            # unknown u at new time level
u_n = np.zeros(Nx+1)            # u at the previous time level

for i in range(0, Nx+1):
    u_n[i] = I(x[i])

for n in range(0, Nt):
    for i in range(1, Nx):
```

```
        u[i] = u_n[i] + F*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
           dt*f(x[i], t[n])

    u[0] = 0;  u[Nx] = 0

    u_n[:]= u
```

Note that we use `a` for α in the code, motivated by easy visual mapping between the variable name and the mathematical symbol in formulas.

We need to state already now that the shown algorithm does not produce meaningful results unless $F \leq 1/2$. Why is explained in Section Section 3.15.

3.4. Implementation

The file diffu1D_u0.py contains a complete function `solver_FE_simple` for solving the 1D diffusion equation with $u = 0$ on the boundary as specified in the algorithm above:

```
import numpy as np
import scipy.sparse
import scipy.sparse.linalg

def solver_FE_simple(I, a, f, L, dt, F, T):
    """
    Simplest expression of the computational algorithm
    using the Forward Euler method and explicit Python loops.
    For this method F <= 0.5 for stability.
    """
    import time

    t0 = time.perf_counter()  # For measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])
```

```
    for n in range(0, Nt):
        for i in range(1, Nx):
            u[i] = (
                u_n[i] + F * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1]) + dt * f(x[i], t[n])
            )

        u[0] = 0
        u[Nx] = 0

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return u_n, x, t, t1 - t0  # u_n holds latest u

def solver_FE(I, a, f, L, dt, F, T, user_action=None, version="scalar"):
    """
    Vectorized implementation of solver_FE_simple.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)  # solution array
    u_n = np.zeros(Nx + 1)  # solution at t-dt

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        if version == "scalar":
            for i in range(1, Nx):
                u[i] = (
                    u_n[i]
                    + F * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
                    + dt * f(x[i], t[n])
                )
```

```
        elif version == "vectorized":
            u[1:Nx] = (
                u_n[1:Nx]
                + F * (u_n[0 : Nx - 1] - 2 * u_n[1:Nx] + u_n[2 : Nx + 1])
                + dt * f(x[1:Nx], t[n])
            )
        else:
            raise ValueError("version=%s" % version)

        u[0] = 0
        u[Nx] = 0
        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return t1 - t0

def solver_BE_simple(I, a, f, L, dt, F, T, user_action=None):
    """
    Simplest expression of the computational algorithm
    for the Backward Euler method, using explicit Python loops
    and a dense matrix format for the coefficient matrix.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)

    A = np.zeros((Nx + 1, Nx + 1))
    b = np.zeros(Nx + 1)

    for i in range(1, Nx):
        A[i, i - 1] = -F
        A[i, i + 1] = -F
        A[i, i] = 1 + 2 * F
```

```
    A[0, 0] = A[Nx, Nx] = 1

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])
    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        for i in range(1, Nx):
            b[i] = u_n[i] + dt * f(x[i], t[n + 1])
        b[0] = b[Nx] = 0
        u[:] = np.linalg.solve(A, b)

        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return t1 - t0

def solver_BE(I, a, f, L, dt, F, T, user_action=None):
    """
    Vectorized implementation of solver_BE_simple using also
    a sparse (tridiagonal) matrix for efficiency.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)  # solution array at t[n+1]
    u_n = np.zeros(Nx + 1)  # solution at t[n]

    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    diagonal[:] = 1 + 2 * F
```

```
    lower[:] = -F  # 1
    upper[:] = -F  # 1
    diagonal[0] = 1
    upper[0] = 0
    diagonal[Nx] = 1
    lower[-1] = 0

    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )
    print(A.todense())

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        b = u_n + dt * f(x[:], t[n + 1])
        b[0] = b[-1] = 0.0  # boundary conditions
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return t1 - t0

def solver_theta(I, a, f, L, dt, F, T, theta=0.5, u_L=0, u_R=0, user_action=None):
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time (no restriction on F,
    i.e., the time step when theta >= 0.5).
    Vectorized implementation and sparse (tridiagonal)
    coefficient matrix.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time

    Nt = int(round(T / float(dt)))
```

```
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)  # solution array at t[n+1]
    u_n = np.zeros(Nx + 1)  # solution at t[n]

    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    Fl = F * theta
    Fr = F * (1 - theta)
    diagonal[:] = 1 + 2 * Fl
    lower[:] = -Fl  # 1
    upper[:] = -Fl  # 1
    diagonal[0] = 1
    upper[0] = 0
    diagonal[Nx] = 1
    lower[-1] = 0

    diags = [0, -1, 1]
    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        b[1:-1] = (
            u_n[1:-1]
            + Fr * (u_n[:-2] - 2 * u_n[1:-1] + u_n[2:])
            + dt * theta * f(x[1:-1], t[n + 1])
            + dt * (1 - theta) * f(x[1:-1], t[n])
        )
        b[0] = u_L
```

```
        b[-1] = u_R  # boundary conditions
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return t1 - t0

def viz(I, a, L, dt, F, T, umin, umax, scheme="FE", animate=True, framefiles=True):
    def plot_u(u, x, t, n):
        plt.plot(x, u, "r-", axis=[0, L, umin, umax], title="t=%f" % t[n])
        if framefiles:
            plt.savefig("tmp_frame%04d.png" % n)
        if t[n] == 0:
            time.sleep(2)
        elif not framefiles:
            time.sleep(0.2)

    user_action = plot_u if animate else lambda u, x, t, n: None

    cpu = eval("solver_" + scheme)(I, a, L, dt, F, T, user_action=user_action)
    return cpu

def plug(scheme="FE", F=0.5, Nx=50):
    L = 1.0
    a = 1.0
    T = 0.1
    dx = L / Nx
    dt = F / a * dx**2

    def I(x):
        """Plug profile as initial condition."""
        if abs(x - L / 2.0) > 0.1:
            return 0
        else:
            return 1

    cpu = viz(
        I,
        a,
        L,
        dt,
        F,
        T,
```

```
        umin=-0.1,
        umax=1.1,
        scheme=scheme,
        animate=True,
        framefiles=True,
    )
    print("CPU time:", cpu)

def gaussian(scheme="FE", F=0.5, Nx=50, sigma=0.05):
    L = 1.0
    a = 1.0
    T = 0.1
    dx = L / Nx
    dt = F / a * dx**2

    def I(x):
        """Gaussian profile as initial condition."""
        return exp(-0.5 * ((x - L / 2.0) ** 2) / sigma**2)

    u, cpu = viz(
        I,
        a,
        L,
        dt,
        F,
        T,
        umin=-0.1,
        umax=1.1,
        scheme=scheme,
        animate=True,
        framefiles=True,
    )
    print("CPU time:", cpu)

def expsin(scheme="FE", F=0.5, m=3):
    L = 10.0
    a = 1
    T = 1.2

    def exact(x, t):
        return exp(-(m**2) * pi**2 * a / L**2 * t) * sin(m * pi / L * x)

    def I(x):
        return exact(x, 0)

    Nx = 80
    dx = L / Nx
```

```
    dt = F / a * dx**2
    viz(I, a, L, dt, F, T, -1, 1, scheme=scheme, animate=True, framefiles=True)

    def action(u, x, t, n):
        e = abs(u - exact(x, t[n])).max()
        errors.append(e)

    errors = []
    Nx_values = [10, 20, 40, 80, 160]
    for Nx in Nx_values:
        eval("solver_" + scheme)(I, a, L, Nx, F, T, user_action=action)
        dt = F * (L / Nx) ** 2 / a
        print(dt, errors[-1])

def test_solvers():
    def u_exact(x, t):
        return x * (L - x) * 5 * t  # fulfills BC at x=0 and x=L

    def I(x):
        return u_exact(x, 0)

    def f(x, t):
        return 5 * x * (L - x) + 10 * a * t

    a = 3.5
    L = 1.5
    Nx = 4
    F = 0.5
    dx = L / Nx
    dt = F / a * dx**2

    def compare(u, x, t, n):  # user_action function
        """Compare exact and computed solution."""
        u_e = u_exact(x, t[n])
        diff = abs(u_e - u).max()
        tol = 1e-14
        assert diff < tol, "max diff: %g" % diff

    import functools

    s = functools.partial  # object for calling a function w/args
    solvers = [
        s(solver_FE_simple, I=I, a=a, f=f, L=L, dt=dt, F=F, T=0.2),
        s(
            solver_FE,
            I=I,
            a=a,
```

```
            f=f,
            L=L,
            dt=dt,
            F=F,
            T=2,
            user_action=compare,
            version="scalar",
        ),
        s(
            solver_FE,
            I=I,
            a=a,
            f=f,
            L=L,
            dt=dt,
            F=F,
            T=2,
            user_action=compare,
            version="vectorized",
        ),
        s(solver_BE_simple, I=I, a=a, f=f, L=L, dt=dt, F=F, T=2, user_action=compare),
        s(solver_BE, I=I, a=a, f=f, L=L, dt=dt, F=F, T=2, user_action=compare),
        s(
            solver_theta,
            I=I,
            a=a,
            f=f,
            L=L,
            dt=dt,
            F=F,
            T=2,
            theta=0,
            u_L=0,
            u_R=0,
            user_action=compare,
        ),
    ]
    u, x, t, cpu = solvers[0]()
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1e-14
    print(u_e)
    print(u)
    assert diff < tol, "max diff solver_FE_simple: %g" % diff

    for solver in solvers:
        solver()
```

```
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("""Usage %s function arg1 arg2 arg3 ...""" % sys.argv[0])
        sys.exit(0)
    cmd = "%s(%s)" % (sys.argv[1], ", ".join(sys.argv[2:]))
    print(cmd)
    eval(cmd)
```

A faster alternative is available in the function `solver_FE`, which adds the possibility of solving the finite difference scheme by vectorization. The vectorized version replaces the explicit loop

```
for i in range(1, Nx):
    u[i] = u_n[i] + F*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) \
           + dt*f(x[i], t[n])
```

by arithmetics on displaced slices of the `u` array:

```
u[1:Nx] = u_n[1:Nx] + F*(u_n[0:Nx-1] - 2*u_n[1:Nx] + u_n[2:Nx+1]) \
          + dt*f(x[1:Nx], t[n])
u[1:-1] = u_n[1:-1] + F*(u_n[0:-2] - 2*u_n[1:-1] + u_n[2:]) \
          + dt*f(x[1:-1], t[n])
```

For example, the vectorized version runs 70 times faster than the scalar version in a case with 100 time steps and a spatial mesh of 10^5 cells.

The `solver_FE` function also features a callback function such that the user can process the solution at each time level. The callback function looks like `user_action(u, x, t, n)`, where `u` is the array containing the solution at time level `n`, `x` holds all the spatial mesh points, while `t` holds all the temporal mesh points. The `solver_FE` function is very similar to `solver_FE_simple` above:

```
"""
Functions for solving a 1D diffusion equations of simplest types
(constant coefficient, no source term):

      u_t = a*u_xx on (0,L)

with boundary conditions u=0 on x=0,L, for t in (0,T].
Initial condition: u(x,0)=I(x).

The following naming convention of variables are used.

===== ============================================================
Name  Description
===== ============================================================
Nx    The total number of mesh cells; mesh points are numbered
      from 0 to Nx.
```

```
F     The dimensionless number a*dt/dx**2, which implicitly
      specifies the time step.
T     The stop time for the simulation.
I     Initial condition (Python function of x).
a     Variable coefficient (constant).
L     Length of the domain ([0,L]).
x     Mesh points in space.
t     Mesh points in time.
n     Index counter in time.
u     Unknown at current/new time level.
u_n   u at the previous time level.
dx    Constant mesh spacing in x.
dt    Constant mesh spacing in t.
===== ==========================================================

user_action is a function of (u, x, t, n), u[i] is the solution at
spatial mesh point x[i] at time t[n], where the calling code
can add visualization, error computations, data analysis,
store solutions, etc.
"""

import sys
import time

import matplotlib.pyplot as plt
import numpy as np
import scipy.sparse
import scipy.sparse.linalg

def solver_FE_simple(I, a, f, L, dt, F, T):
    """
    Simplest expression of the computational algorithm
    using the Forward Euler method and explicit Python loops.
    For this method F <= 0.5 for stability.
    """
    import time

    t0 = time.perf_counter()  # For measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
```

```
    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    for n in range(0, Nt):
        for i in range(1, Nx):
            u[i] = (
                u_n[i] + F * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1]) + dt * f(x[i], t[n])
            )

        u[0] = 0
        u[Nx] = 0

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return u_n, x, t, t1 - t0  # u_n holds latest u

def solver_FE(I, a, f, L, dt, F, T, user_action=None, version="scalar"):
    """
    Vectorized implementation of solver_FE_simple.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time

    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)  # solution array
    u_n = np.zeros(Nx + 1)  # solution at t-dt

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        if version == "scalar":
```

```
            for i in range(1, Nx):
                u[i] = (
                    u_n[i]
                    + F * (u_n[i - 1] - 2 * u_n[i] + u_n[i + 1])
                    + dt * f(x[i], t[n])
                )

        elif version == "vectorized":
            u[1:Nx] = (
                u_n[1:Nx]
                + F * (u_n[0 : Nx - 1] - 2 * u_n[1:Nx] + u_n[2 : Nx + 1])
                + dt * f(x[1:Nx], t[n])
            )
        else:
            raise ValueError("version=%s" % version)

        u[0] = 0
        u[Nx] = 0
        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return t1 - t0
```

3.5. Verification

3.5.1. Exact solution of discrete equations

Before thinking about running the functions in the previous section, we need to construct a suitable test example for verification. It appears that a manufactured solution that is linear in time and at most quadratic in space fulfills the Forward Euler scheme exactly. With the restriction that $u = 0$ for $x = 0, L$, we can try the solution

$$u(x,t) = 5tx(L-x)\,.$$

Inserted in the PDE, it requires a source term

$$f(x,t) = 10\alpha t + 5x(L-x)\,.$$

With the formulas from Appendix Section 6.3 we can easily check that the manufactured `u` fulfills the scheme:

$$\begin{aligned}[D_t^+ u = \alpha D_x D_x u + f]_i^n &= [5x(L-x)D_t^+ t = 5t\alpha D_x D_x(xL - x^2) + \\ &\qquad 10\alpha t + 5x(L-x)]_i^n \\ &= [5x(L-x) = 5t\alpha(-2) + 10\alpha t + 5x(L-x)]_i^n,\end{aligned}$$

which is a 0=0 expression. The computation of the source term, given any u, is easily automated with `sympy`:

```
import sympy as sym
x, t, a, L = sym.symbols('x t a L')
u = x*(L-x)*5*t

def pde(u):
    return sym.diff(u, t) - a*sym.diff(u, x, x)

f = sym.simplify(pde(u))
```

Now we can choose any expression for `u` and automatically get the suitable source term `f`. However, the manufactured solution `u` will in general not be exactly reproduced by the scheme: only constant and linear functions are differentiated correctly by a forward difference, while only constant, linear, and quadratic functions are differentiated exactly by a $[D_xD_xu]_i^n$ difference.

The numerical code will need to access the `u` and `f` above as Python functions. The exact solution is wanted as a Python function `u_exact(x, t)`, while the source term is wanted as `f(x, t)`. The parameters `a` and `L` in `u` and `f` above are symbols and must be replaced by `float` objects in a Python function. This can be done by redefining `a` and `L` as `float` objects and performing substitutions of symbols by numbers in `u` and `f`. The appropriate code looks like this:

```
a = 0.5
L = 1.5
u_exact = sym.lambdify(
    [x, t], u.subs('L', L).subs('a', a), modules='numpy')
f = sym.lambdify(
    [x, t], f.subs('L', L).subs('a', a), modules='numpy')
I = lambda x: u_exact(x, 0)
```

Here we also make a function `I` for the initial condition.

The idea now is that our manufactured solution should be exactly reproduced by the code (to machine precision). For this purpose we make a test function for comparing the exact and numerical solutions at the end of the time interval:

```
def test_solver_FE():

    dx = L/3  # 3 cells
    F = 0.5
    dt = F*dx**2

    u, x, t, cpu = solver_FE_simple(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2)
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
```

```
    assert diff < tol, 'max diff solver_FE_simple: %g' % diff

    u, x, t, cpu = solver_FE(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2,
        user_action=None, version='scalar')
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    assert diff < tol, 'max diff solver_FE, scalar: %g' % diff

    u, x, t, cpu = solver_FE(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2,
        user_action=None, version='vectorized')
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    assert diff < tol, 'max diff solver_FE, vectorized: %g' % diff
```

i The critical value $F = 0.5$

We emphasize that the value `F=0.5` is critical: the tests above will fail if `F` has a larger value. This is because the Forward Euler scheme is unstable for $F > 1/2$.
The reader may wonder if $F = 1/2$ is safe or if $F < 1/2$ should be required. Experiments show that $F = 1/2$ works fine for $u_t = \alpha u_{xx}$, so there is no accumulation of rounding errors in this case and hence no need to introduce any safety factor to keep F away from the limiting value 0.5.

3.5.2. Checking convergence rates

If our chosen exact solution does not satisfy the discrete equations exactly, we are left with checking the convergence rates, just as we did previously for the wave equation. However, with the Euler scheme here, we have different accuracies in time and space, since we use a second order approximation to the spatial derivative and a first order approximation to the time derivative. Thus, we must expect different convergence rates in time and space. For the numerical error,

$$E = C_t \Delta t^r + C_x \Delta x^p,$$

we should get convergence rates $r = 1$ and $p = 2$ (C_t and C_x are unknown constants). As previously, in Section Section 2.10.1, we simplify matters by introducing a single discretization parameter h:

$$h = \Delta t, \quad \Delta x = K h^{r/p},$$

where K is any constant. This allows us to factor out only *one* discretization parameter h from the formula:

$$E = C_t h + C_x (K h^{r/p})^p = \tilde{C} h^r, \quad \tilde{C} = C_t + C_s K^r \,.$$

The computed rate r should approach 1 with increasing resolution.

It is tempting, for simplicity, to choose $K = 1$, which gives $\Delta x = h^{r/p}$, expected to be $\sqrt{\Delta t}$. However, we have to control the stability requirement: $F \leq \frac{1}{2}$, which means

$$\frac{\alpha \Delta t}{\Delta x^2} \leq \frac{1}{2} \quad \Rightarrow \quad \Delta x \geq \sqrt{2\alpha} h^{1/2},$$

implying that $K = \sqrt{2\alpha}$ is our choice in experiments where we lie on the stability limit $F = 1/2$.

3.6. Numerical experiments

When a test function like the one above runs silently without errors, we have some evidence for a correct implementation of the numerical method. The next step is to do some experiments with more interesting solutions.

We target a scaled diffusion problem where x/L is a new spatial coordinate and $\alpha t/L^2$ is a new time coordinate. The source term f is omitted, and u is scaled by $\max_{x\in[0,L]} |I(x)|$ (see Section 3.2 in (Langtangen and Pedersen 2016) for details). The governing PDE is then

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2},$$

in the spatial domain $[0, L]$, with boundary conditions $u(0) = u(1) = 0$. Two initial conditions will be tested: a discontinuous plug,

$$I(x) = \begin{cases} 0, & |x - L/2| > 0.1 \\ 1, & \text{otherwise} \end{cases}$$

and a smooth Gaussian function,

$$I(x) = e^{-\frac{1}{2\sigma^2}(x-L/2)^2}.$$

The functions `plug` and `gaussian` in `diffu1D_u0.py` run the two cases, respectively:

```
def plug(scheme="FE", F=0.5, Nx=50):
    L = 1.0
    a = 1.0
    T = 0.1
    dx = L / Nx
    dt = F / a * dx**2

    def I(x):
        """Plug profile as initial condition."""
        if abs(x - L / 2.0) > 0.1:
            return 0
        else:
            return 1

    cpu = viz(
        I,
        a,
```

```
        L,
        dt,
        F,
        T,
        umin=-0.1,
        umax=1.1,
        scheme=scheme,
        animate=True,
        framefiles=True,
    )
    print("CPU time:", cpu)

def gaussian(scheme="FE", F=0.5, Nx=50, sigma=0.05):
    L = 1.0
    a = 1.0
    T = 0.1
    dx = L / Nx
    dt = F / a * dx**2

    def I(x):
        """Gaussian profile as initial condition."""
        return exp(-0.5 * ((x - L / 2.0) ** 2) / sigma**2)

    u, cpu = viz(
        I,
        a,
        L,
        dt,
        F,
        T,
        umin=-0.1,
        umax=1.1,
        scheme=scheme,
        animate=True,
        framefiles=True,
    )
    print("CPU time:", cpu)
```

These functions make use of the function `viz` for running the solver and visualizing the solution using a callback function with plotting:

```
def viz(I, a, L, dt, F, T, umin, umax, scheme="FE", animate=True, framefiles=True):
    def plot_u(u, x, t, n):
        plt.plot(x, u, "r-", axis=[0, L, umin, umax], title="t=%f" % t[n])
        if framefiles:
            plt.savefig("tmp_frame%04d.png" % n)
        if t[n] == 0:
```

```
            time.sleep(2)
        elif not framefiles:
            time.sleep(0.2)

    user_action = plot_u if animate else lambda u, x, t, n: None

    cpu = eval("solver_" + scheme)(I, a, L, dt, F, T, user_action=user_action)
    return cpu
```

Notice that this `viz` function stores all the solutions in a list `solutions` in the callback function. Modern computers have hardly any problem with storing a lot of such solutions for moderate values of N_x in 1D problems, but for 2D and 3D problems, this technique cannot be used and solutions must be stored in files.

Our experiments employ a time step $\Delta t = 0.0002$ and simulate for $t \in [0, 0.1]$. First we try the highest value of F: $F = 0.5$. This resolution corresponds to $N_x = 50$. A possible terminal command is

```
Terminal> python -c 'from diffu1D_u0 import gaussian
          gaussian("solver_FE", F=0.5, dt=0.0002)'
```

The $u(x, t)$ curve as a function of x is shown in Figure Figure 3.1 at four time levels.

We see that the curves have saw-tooth waves in the beginning of the simulation. This non-physical noise is smoothed out with time, but solutions of the diffusion equations are known to be smooth, and this numerical solution is definitely not smooth. Lowering F helps: $F \leq 0.25$ gives a smooth solution, see Figure Figure 3.2.

Increasing F slightly beyond the limit 0.5, to $F = 0.51$, gives growing, non-physical instabilities, as seen in Figure Figure 3.3.

Instead of a discontinuous initial condition we now try the smooth Gaussian function for $I(x)$. A simulation for $F = 0.5$ is shown in Figure Figure 3.4. Now the numerical solution is smooth for all times, and this is true for any $F \leq 0.5$.

Experiments with these two choices of $I(x)$ reveal some important observations:

- The Forward Euler scheme leads to growing solutions if $F > \frac{1}{2}$.
- $I(x)$ as a discontinuous plug leads to a saw tooth-like noise for $F = \frac{1}{2}$, which is absent for $F \leq \frac{1}{4}$.
- The smooth Gaussian initial function leads to a smooth solution for all relevant F values ($F \leq \frac{1}{2}$).

3.7. Implicit methods for the 1D diffusion equation

Simulations with the Forward Euler scheme show that the time step restriction, $F \leq \frac{1}{2}$, which means $\Delta t \leq \Delta x^2/(2\alpha)$, may be relevant in the beginning of the diffusion process, when the solution changes quite fast, but as time increases, the process slows down, and a small Δt may be inconvenient. With

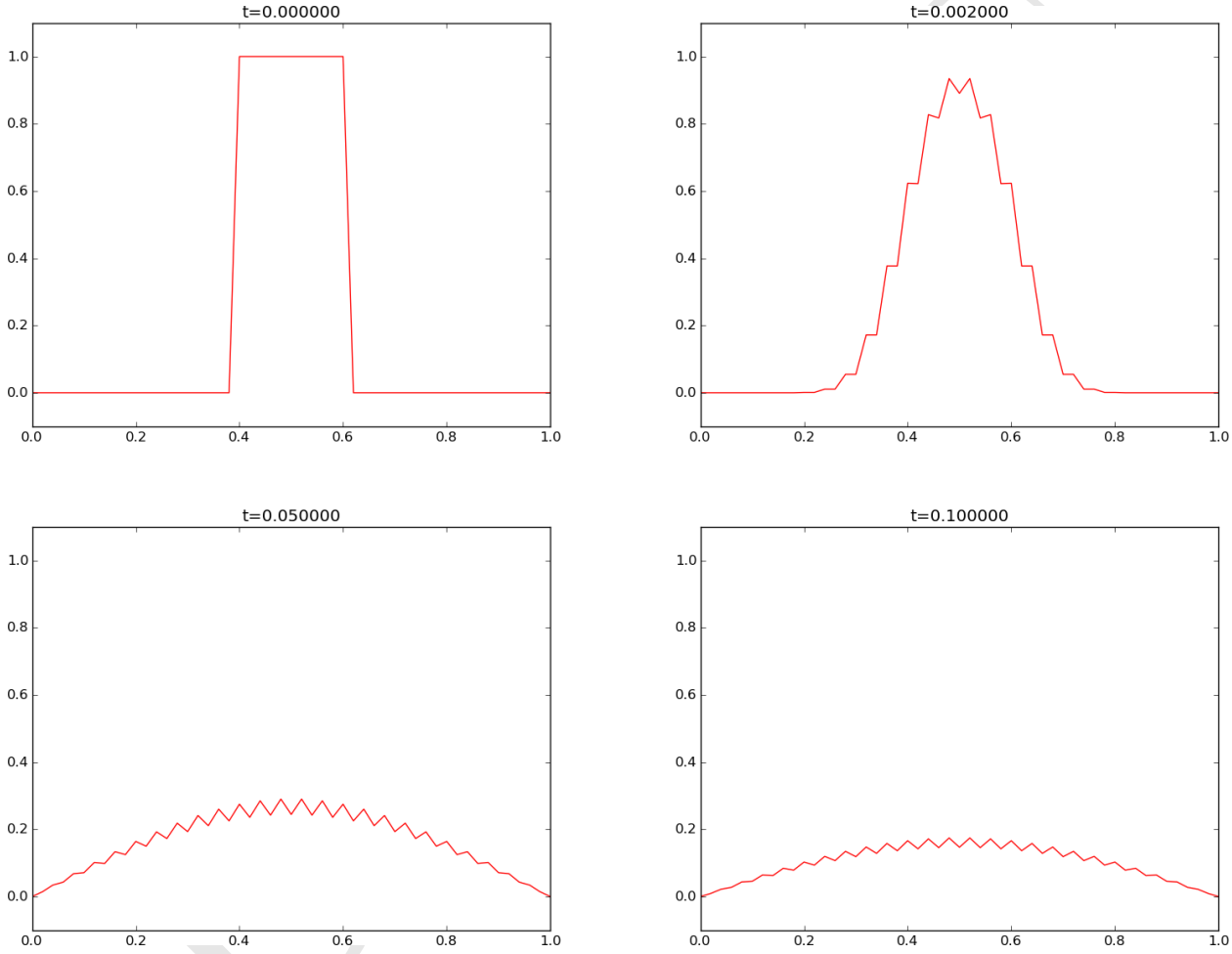


Figure 3.1.: Forward Euler scheme for $F = 0.5$.

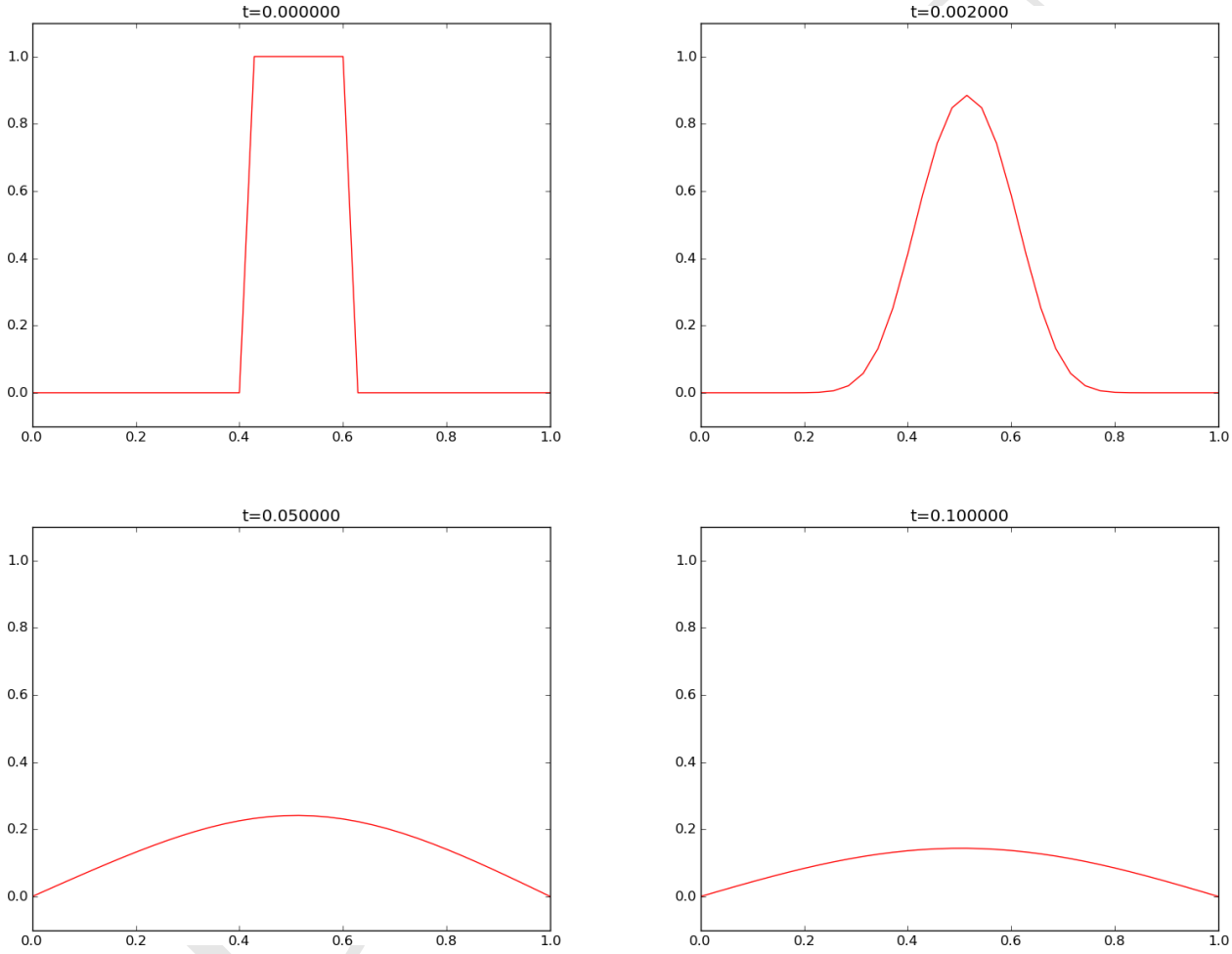


Figure 3.2.: Forward Euler scheme for $F = 0.25$.

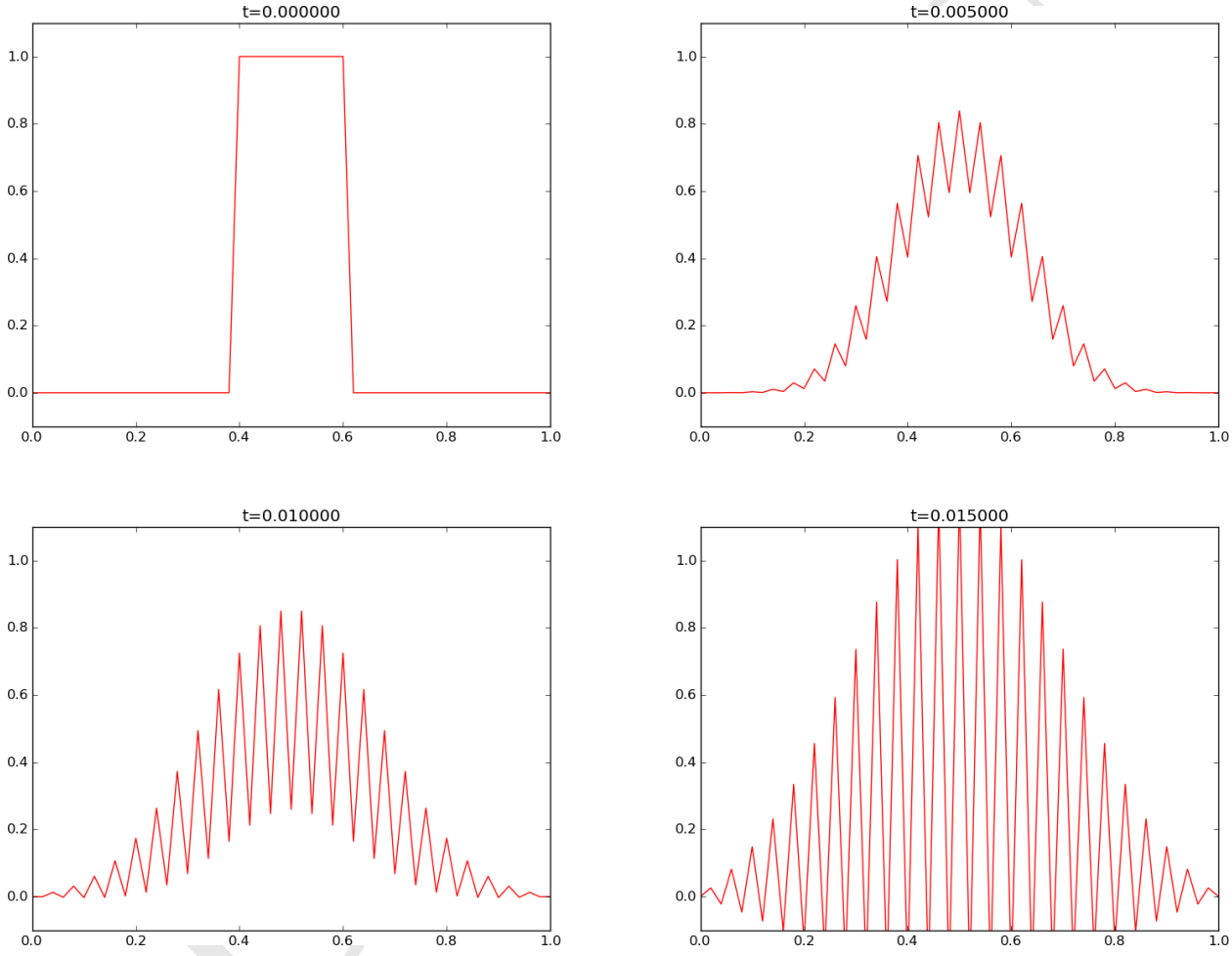


Figure 3.3.: Forward Euler scheme for $F = 0.51$.

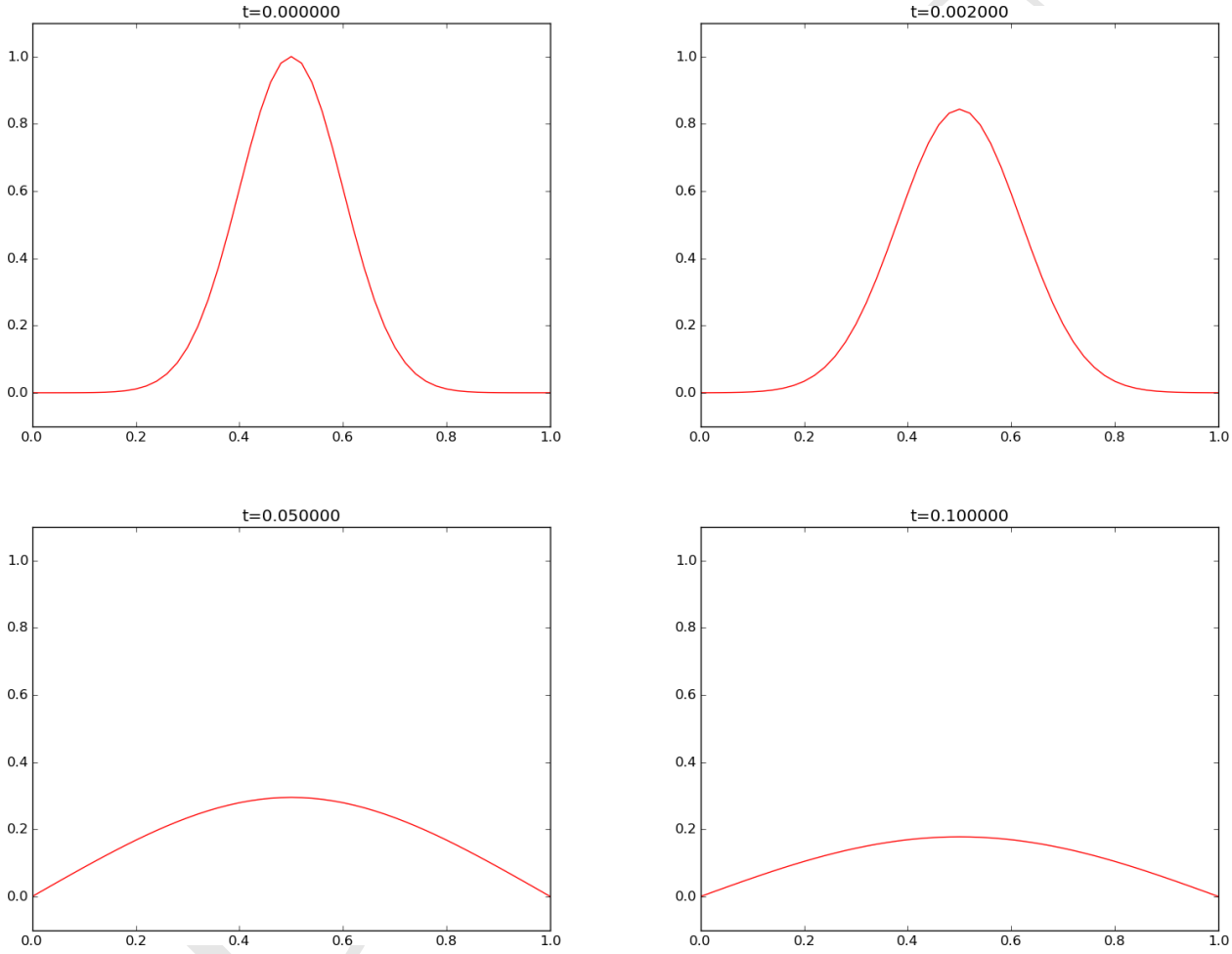


Figure 3.4.: Forward Euler scheme for $F = 0.5$.

implicit schemes, which lead to coupled systems of linear equations to be solved at each time level, any size of Δt is possible (but the accuracy decreases with increasing Δt). The Backward Euler scheme, derived and implemented below, is the simplest implicit scheme for the diffusion equation.

3.8. Backward Euler scheme

In (3.2), we now apply a backward difference in time, but the same central difference in space:

$$[D_t^- u = D_x D_x u + f]_i^n, \tag{3.6}$$

which written out reads

$$\frac{u_i^n - u^{n-1} **i}{\Delta t} = \alpha \frac{u^n **i + 1 - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \,. \tag{3.7}$$

Now we assume u_i^{n-1} is already computed, but that all quantities at the "new" time level n are unknown. This time it is not possible to solve with respect to u_i^n because this value couples to its neighbors in space, u_{i-1}^n and u_{i+1}^n, which are also unknown. Let us examine this fact for the case when $N_x = 3$. Equation (3.7) written for $i = 1, \ldots, Nx - 1 = 1, 2$ becomes

$$\frac{u_1^n - u^{n-1} **1}{\Delta t} = \alpha \frac{u^n **2 - 2u_1^n + u_0^n}{\Delta x^2} + f_1^n \tag{3.8}$$

$$\frac{u_2^n - u^{n-1} **2}{\Delta t} = \alpha \frac{u^n **3 - 2u_2^n + u_1^n}{\Delta x^2} + f_2^n \tag{3.9}$$

The boundary values u_0^n and u_3^n are known as zero. Collecting the unknown new values u_1^n and u_2^n on the left-hand side and multiplying by Δt gives

$$(1 + 2F)\, u^n **1 - F u^n **2 = u_1^{n-1} + \Delta t f_1^n, \tag{3.10}$$

$$-F u_1^n + (1 + 2F)\, u_2^n = u_2^{n-1} + \Delta t f_2^n \,. \tag{3.11}$$

This is a coupled 2×2 system of algebraic equations for the unknowns u_1^n and u_2^n. The equivalent matrix form is

$$\begin{pmatrix} 1 + 2F & -F \\ -F & 1 + 2F \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \end{pmatrix} = \begin{pmatrix} u_1^{n-1} + \Delta t f_1^n \\ u_2^{n-1} + \Delta t f_2^n \end{pmatrix}$$

i Terminology: implicit vs. explicit methods

Discretization methods that lead to a coupled system of equations for the unknown function at a new time level are said to be *implicit methods*. The counterpart, *explicit methods*, refers to discretization methods where there is a simple explicit formula for the values of the unknown function at each of the spatial mesh points at the new time level. From an implementational point of view, implicit methods are more comprehensive to code since they require the solution of coupled equations, i.e., a matrix system, at each time level. With explicit methods we have a closed-form formula for the value of the unknown at each mesh point.

Very often explicit schemes have a restriction on the size of the time step that can be relaxed by using implicit schemes. In fact, implicit schemes are frequently unconditionally stable, so the size of the time step is governed by accuracy and not by stability. This is the great advantage of implicit schemes.

In the general case, (3.7) gives rise to a coupled $(N_x-1)\times(N_x-1)$ system of algebraic equations for all the unknown u_i^n at the interior spatial points $i=1,\ldots,N_x-1$. Collecting the unknowns on the left-hand side, (3.7) can be written

$$-Fu_{i-1}^n + (1+2F)\,u^n**i - Fu^n**i + 1 = u_{i-1}^{n-1}, \tag{3.12}$$

for $i=1,\ldots,N_x-1$. One can either view these equations as a system where the u_i^n values at the internal mesh points, $i=1,\ldots,N_x-1$, are unknown, or we may append the boundary values u_0^n and $u_{N_x}^n$ to the system. In the latter case, all u_i^n for $i=0,\ldots,N_x$ are considered unknown, and we must add the boundary equations to the N_x-1 equations in (3.12):

$$u_0^n = 0, \tag{3.13}$$

$$u_{N_x}^n = 0\,. \tag{3.14}$$

A coupled system of algebraic equations can be written on matrix form, and this is important if we want to call up ready-made software for solving the system. The equations (3.12) and (3.13)–(3.14) correspond to the matrix equation

$$AU = b$$

where $U=(u_0^n,\ldots,u_{N_x}^n)$, and the matrix A has the following structure:

$$A = \begin{pmatrix}
A_{0,0} & A_{0,1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
A_{1,0} & A_{1,1} & A_{1,2} & \ddots & & & & & \vdots \\
0 & A_{2,1} & A_{2,2} & A_{2,3} & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & A_{i,i-1} & A_{i,i} & A_{i,i+1} & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & \ddots & \ddots & A_{N_x-1,N_x} \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & A_{N_x,N_x-1} & A_{N_x,N_x}
\end{pmatrix} \tag{3.15}$$

The nonzero elements are given by

$$A_{i,i-1} = -F \tag{3.16}$$

$$A_{i,i} = 1+2F \tag{3.17}$$

$$A_{i,i+1} = -F \tag{3.18}$$

in the equations for internal points, $i=1,\ldots,N_x-1$. The first and last equation correspond to the boundary condition, where we know the solution, and therefore we must have

$$A_{0,0} = 1, \tag{3.19}$$
$$A_{0,1} = 0, \tag{3.20}$$
$$A_{N_x,N_x-1} = 0, \tag{3.21}$$
$$A_{N_x,N_x} = 1\,. \tag{3.22}$$

The right-hand side b is written as

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{N_x} \end{pmatrix}$$

with

$$b_0 = 0, \tag{3.23}$$
$$b_i = u_i^{n-1}, \quad i = 1, \ldots, N_x - 1, \tag{3.24}$$
$$b_{N_x} = 0\,. \tag{3.25}$$

We observe that the matrix A contains quantities that do not change in time. Therefore, A can be formed once and for all before we enter the recursive formulas for the time evolution. The right-hand side b, however, must be updated at each time step. This leads to the following computational algorithm, here sketched with Python code:

```
x = np.linspace(0, L, Nx+1)   # mesh points in space
dx = x[1] - x[0]
t = np.linspace(0, T, N+1)    # mesh points in time
u   = np.zeros(Nx+1)          # unknown u at new time level
u_n = np.zeros(Nx+1)          # u at the previous time level

A = np.zeros((Nx+1, Nx+1))
b = np.zeros(Nx+1)

for i in range(1, Nx):
    A[i,i-1] = -F
    A[i,i+1] = -F
    A[i,i] = 1 + 2*F
A[0,0] = A[Nx,Nx] = 1

for i in range(0, Nx+1):
    u_n[i] = I(x[i])

import scipy.linalg
```

```
for n in range(0, Nt):
    for i in range(1, Nx):
        b[i] = -u_n[i]
    b[0] = b[Nx] = 0
    u[:] = scipy.linalg.solve(A, b)

    u_n[:] = u
```

Regarding verification, the same considerations apply as for the Forward Euler method (Section Section 3.5.1).

3.9. Sparse matrix implementation

We have seen from (3.15) that the matrix A is tridiagonal. The code segment above used a full, dense matrix representation of A, which stores a lot of values we know are zero beforehand, and worse, the solution algorithm computes with all these zeros. With $N_x + 1$ unknowns, the work by the solution algorithm is $\frac{1}{3}(N_x + 1)^3$ and the storage requirements $(N_x + 1)^2$. By utilizing the fact that A is tridiagonal and employing corresponding software tools that work with the three diagonals, the work and storage demands can be proportional to N_x only. This leads to a dramatic improvement: with $N_x = 200$, which is a realistic resolution, the code runs about 40,000 times faster and reduces the storage to just 1.5%! It is no doubt that we should take advantage of the fact that A is tridiagonal.

The key idea is to apply a data structure for a tridiagonal or sparse matrix. The `scipy.sparse` package has relevant utilities. For example, we can store only the nonzero diagonals of a matrix. The package also has linear system solvers that operate on sparse matrix data structures. The code below illustrates how we can store only the main diagonal and the upper and lower diagonals.

```
main  = np.zeros(Nx+1)
lower = np.zeros(Nx)
upper = np.zeros(Nx)
b     = np.zeros(Nx+1)

main[:] = 1 + 2*F
lower[:] = -F
upper[:] = -F
main[0] = 1
main[Nx] = 1

A = scipy.sparse.diags(
    diagonals=[main, lower, upper],
    offsets=[0, -1, 1], shape=(Nx+1, Nx+1),
    format='csr')
print A.todense()  # Check that A is correct
```

```
for i in range(0,Nx+1):
    u_n[i] = I(x[i])

for n in range(0, Nt):
    b = u_n
    b[0] = b[-1] = 0.0  # boundary conditions
    u[:] = scipy.sparse.linalg.spsolve(A, b)
    u_n[:] = u
```

The `scipy.sparse.linalg.spsolve` function utilizes the sparse storage structure of `A` and performs, in this case, a very efficient Gaussian elimination solve.

The program diffu1D_u0.py contains a function `solver_BE`, which implements the Backward Euler scheme sketched above. As mentioned in Section Section 3.3, the functions `plug` and `gaussian` run the case with $I(x)$ as a discontinuous plug or a smooth Gaussian function. All experiments point to two characteristic features of the Backward Euler scheme: 1) it is always stable, and 2) it always gives a smooth, decaying solution.

3.10. Crank-Nicolson scheme

The idea in the Crank-Nicolson scheme is to apply centered differences in space and time, combined with an average in time. We demand the PDE to be fulfilled at the spatial mesh points, but midway between the points in the time mesh:

$$\frac{\partial}{\partial t}u(x_i, t_{n+\frac{1}{2}}) = \alpha\frac{\partial^2}{\partial x^2}u(x_i, t_{n+\frac{1}{2}}) + f(x_i, t_{n+\frac{1}{2}}),$$

for $i = 1, \ldots, N_x - 1$ and $n = 0, \ldots, N_t - 1$.

With centered differences in space and time, we get

$$[D_t u = \alpha D_x D_x u + f]_i^{n+\frac{1}{2}}.$$

On the right-hand side we get an expression

$$\frac{1}{\Delta x^2}\left(u_{i-1}^{n+\frac{1}{2}} - 2u^{n+\frac{1}{2}} * *i + u^{n+\frac{1}{2}} * *i + 1\right) + f_i^{n+\frac{1}{2}}.$$

This expression is problematic since $u_i^{n+\frac{1}{2}}$ is not one of the unknowns we compute. A possibility is to replace $u_i^{n+\frac{1}{2}}$ by an arithmetic average:

$$u_i^{n+\frac{1}{2}} \approx \frac{1}{2}\left(u^n * *i + u^{n+1} * *i\right).$$

In the compact notation, we can use the arithmetic average notation $\overline{u}^t$:

$$[D_t u = \alpha D_x D_x \overline{u}^t + f]_i^{n+\frac{1}{2}}.$$

We can also use an average for $f_i^{n+\frac{1}{2}}$:

$$[D_t u = \alpha D_x D_x \overline{u}^t + \overline{f}^t]_i^{n+\frac{1}{2}} .$$

After writing out the differences and average, multiplying by Δt, and collecting all unknown terms on the left-hand side, we get

$$u^{n+1}**i - \frac{1}{2}F(u^{n+1}**i - 1 - 2u^{n+1}**i + u^{n+1}**i + 1) = u^n**i + \frac{1}{2}F(u^n**i - 1 - 2u^n**i + u^n**i + 1) + \frac{1}{2}f_i^{n+1} + \frac{1}{2}f_i^n . \tag{3.26}$$

Also here, as in the Backward Euler scheme, the new unknowns $u^{n+1}**i - 1$, $u^{n+1}**i$, and u_{i+1}^{n+1} are coupled in a linear system $AU = b$, where A has the same structure as in (3.15), but with slightly different entries:

$$A_{i,i-1} = -\frac{1}{2}F \tag{3.27}$$
$$A_{i,i} = 1 + F \tag{3.28}$$
$$A_{i,i+1} = -\frac{1}{2}F \tag{3.29}$$

in the equations for internal points, $i = 1, \ldots, N_x - 1$. The equations for the boundary points correspond to

$$A_{0,0} = 1, \tag{3.30}$$
$$A_{0,1} = 0, \tag{3.31}$$
$$A_{N_x,N_x-1} = 0, \tag{3.32}$$
$$A_{N_x,N_x} = 1 . \tag{3.33}$$

The right-hand side b has entries

$$b_0 = 0, \tag{3.34}$$
$$b_i = u_i^{n-1} + \frac{1}{2}(f_i^n + f_i^{n+1}), \quad i = 1, \ldots, N_x - 1, \tag{3.35}$$
$$b_{N_x} = 0 . \tag{3.36}$$

When verifying some implementation of the Crank-Nicolson scheme by convergence rate testing, one should note that the scheme is second order accurate in both space and time. The numerical error then reads

$$E = C_t \Delta t^r + C_x \Delta x^r ,$$

where $r = 2$ (C_t and C_x are unknown constants, as before). When introducing a single discretization parameter, we may now simply choose

$$h = \Delta x = \Delta t,$$

which gives

$$E = C_t h^r + C_x h^r = (C_t + C_x)h^r,$$

where r should approach 2 as resolution is increased in the convergence rate computations.

3.11. The unifying θ rule

For the equation

$$\frac{\partial u}{\partial t} = G(u),$$

where $G(u)$ is some spatial differential operator, the θ-rule looks like

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \theta G(u_i^{n+1}) + (1-\theta)G(u_i^n)\,.$$

The important feature of this time discretization scheme is that we can implement one formula and then generate a family of well-known and widely used schemes:

- $\theta = 0$ gives the Forward Euler scheme in time
- $\theta = 1$ gives the Backward Euler scheme in time
- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme in time

In the compact difference notation, we write the θ rule as

$$[\overline{D}_t u = \alpha D_x D_x u]^{n+\theta}\,.$$

We have that $t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n$.

Applied to the 1D diffusion problem, the θ-rule gives

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha\left(\theta\frac{u_{i+1}^{n+1} - 2u^{n+1}**i + u^{n+1}**i - 1}{\Delta x^2} + (1-\theta)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\right) + \theta f_i^{n+1} + (1-\theta)f_i^n$$

.This scheme also leads to a matrix system with entries

$$A_{i,i-1} = -F\theta, \quad A_{i,i} = 1 + 2F\theta \quad , A_{i,i+1} = -F\theta,$$

while right-hand side entry b_i is

$$b_i = u_i^n + F(1-\theta)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \Delta t\theta f_i^{n+1} + \Delta t(1-\theta)f_i^n\,.$$

The corresponding entries for the boundary points are as in the Backward Euler and Crank-Nicolson schemes listed earlier.

Note that convergence rate testing with implementations of the theta rule must adjust the error expression according to which of the underlying schemes is actually being run. That is, if $\theta = 0$ (i.e., Forward Euler) or $\theta = 1$ (i.e., Backward Euler), there should be first order convergence, whereas with $\theta = 0.5$ (i.e., Crank-Nicolson), one should get second order convergence (as outlined in previous sections).

$$[D_t u = \alpha D_x D_x \overline{u}^{t,\theta}]_i^n$$

3.12. Experiments

We can repeat the experiments from Section Section 3.6 to see if the Backward Euler or Crank-Nicolson schemes have problems with sawtooth-like noise when starting with a discontinuous initial condition. We can also verify that we can have $F > \frac{1}{2}$, which allows larger time steps than in the Forward Euler method.

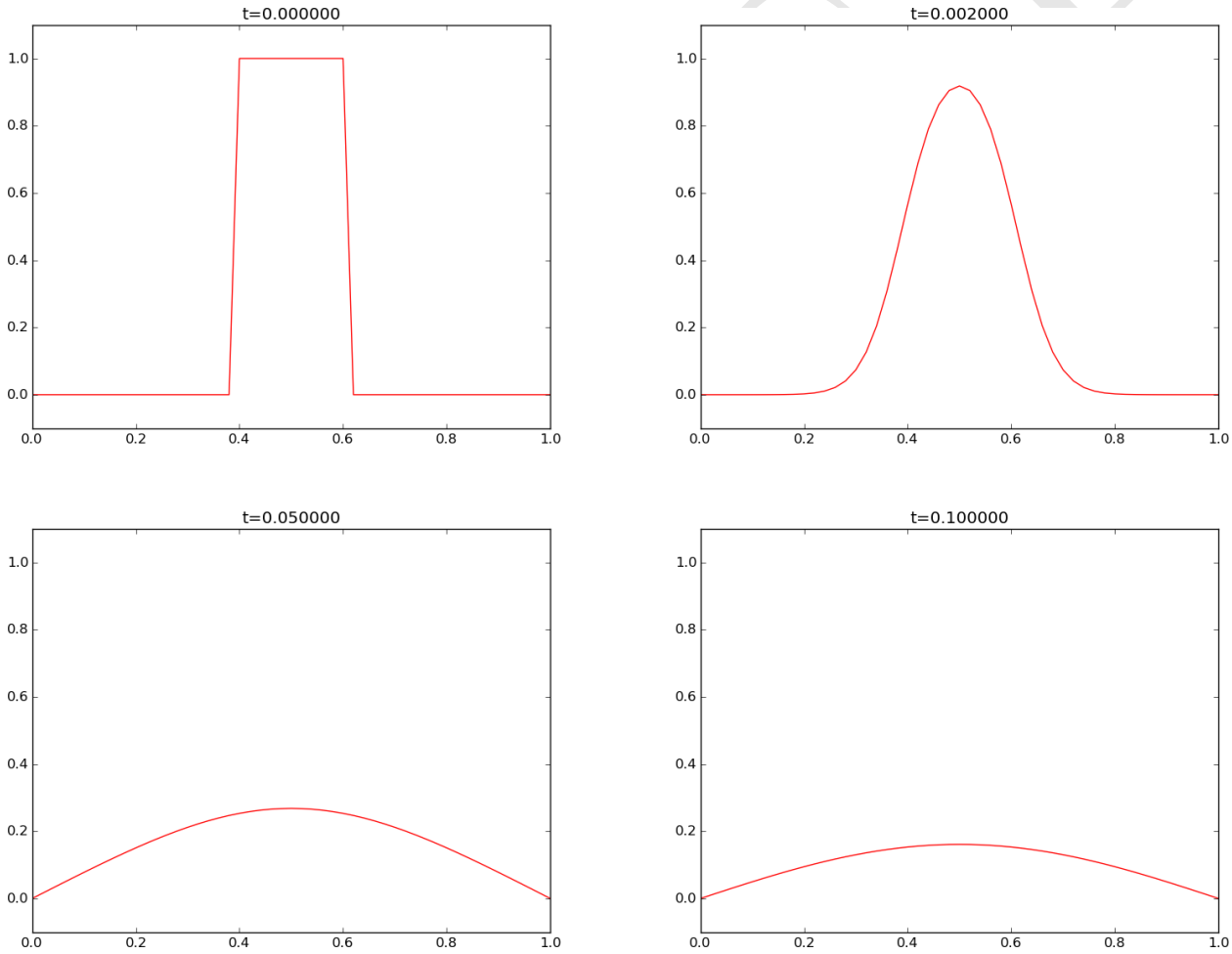


Figure 3.5.: Backward Euler scheme for $F = 0.5$.

The Backward Euler scheme always produces smooth solutions for any F. Figure Figure 3.5 shows one example. Note that the mathematical discontinuity at $t = 0$ leads to a linear variation on a mesh, but the approximation to a jump becomes better as N_x increases. In our simulation, we

specify Δt and F, and set N_x to $L/\sqrt{\alpha \Delta t/F}$. Since $N_x \sim \sqrt{F}$, the discontinuity looks sharper in the Crank-Nicolson simulations with larger F.

The Crank-Nicolson method produces smooth solutions for small F, $F \leq \frac{1}{2}$, but small noise gets more and more evident as F increases. Figures Figure 3.6 and Figure 3.7 demonstrate the effect for $F = 3$ and $F = 10$, respectively. Section Section 3.15 explains why such noise occur.

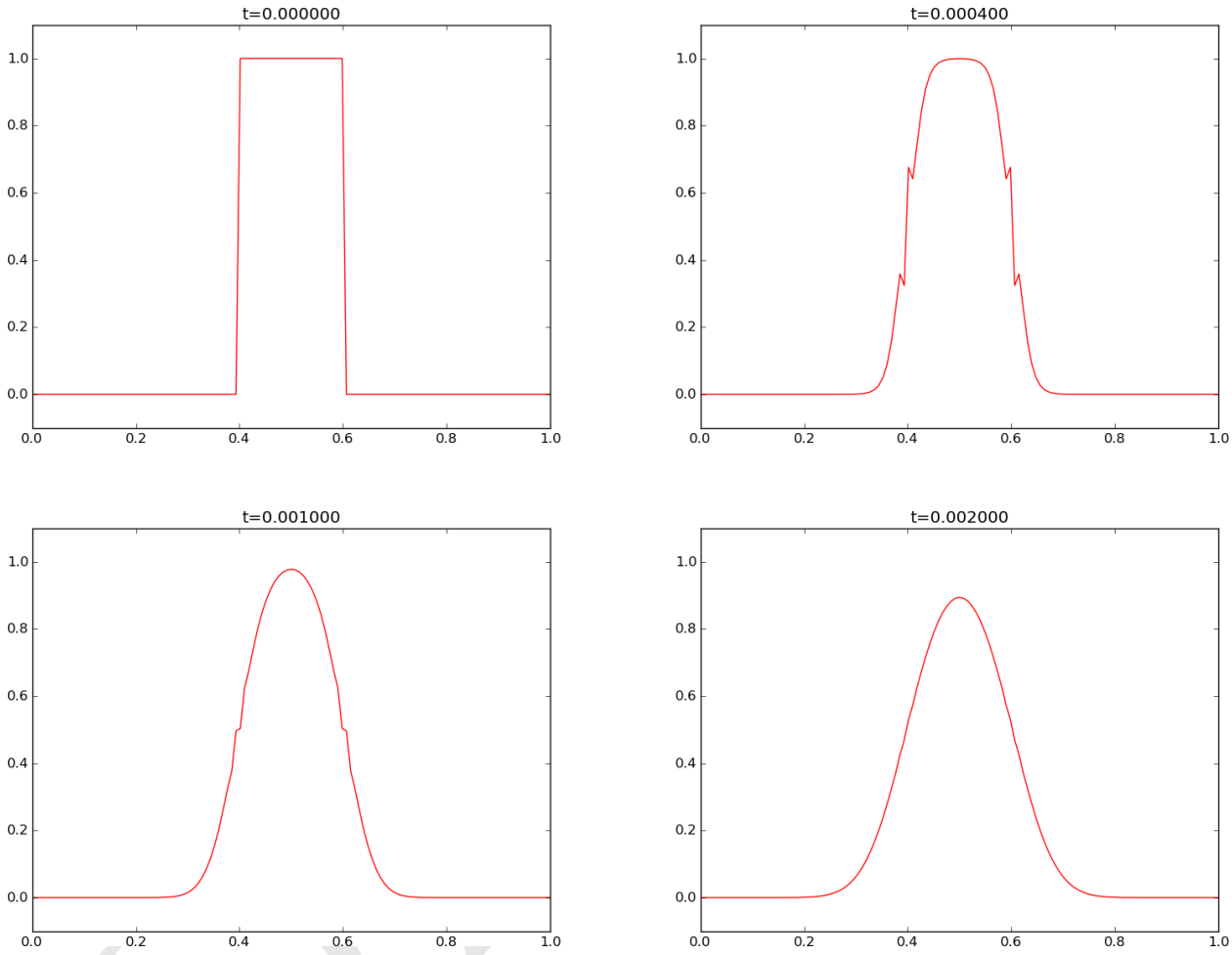


Figure 3.6.: Crank-Nicolson scheme for $F = 3$.

3.13. The Laplace and Poisson equation

The Laplace equation, $\nabla^2 u = 0$, and the Poisson equation, $-\nabla^2 u = f$, occur in numerous applications throughout science and engineering. In 1D these equations read $u''(x) = 0$ and $-u''(x) = f(x)$, respectively. We can solve 1D variants of the Laplace equations with the listed software, because we can interpret $u_{xx} = 0$ as the limiting solution of $u_t = \alpha u_{xx}$ when u reaches a steady state limit where $u_t \to 0$. Similarly, Poisson's equation $-u_{xx} = f$ arises from solving $u_t = u_{xx} + f$ and letting $t \to \infty$ so $u_t \to 0$.

Technically in a program, we can simulate $t \to \infty$ by just taking one large time step: $\Delta t \to \infty$. In

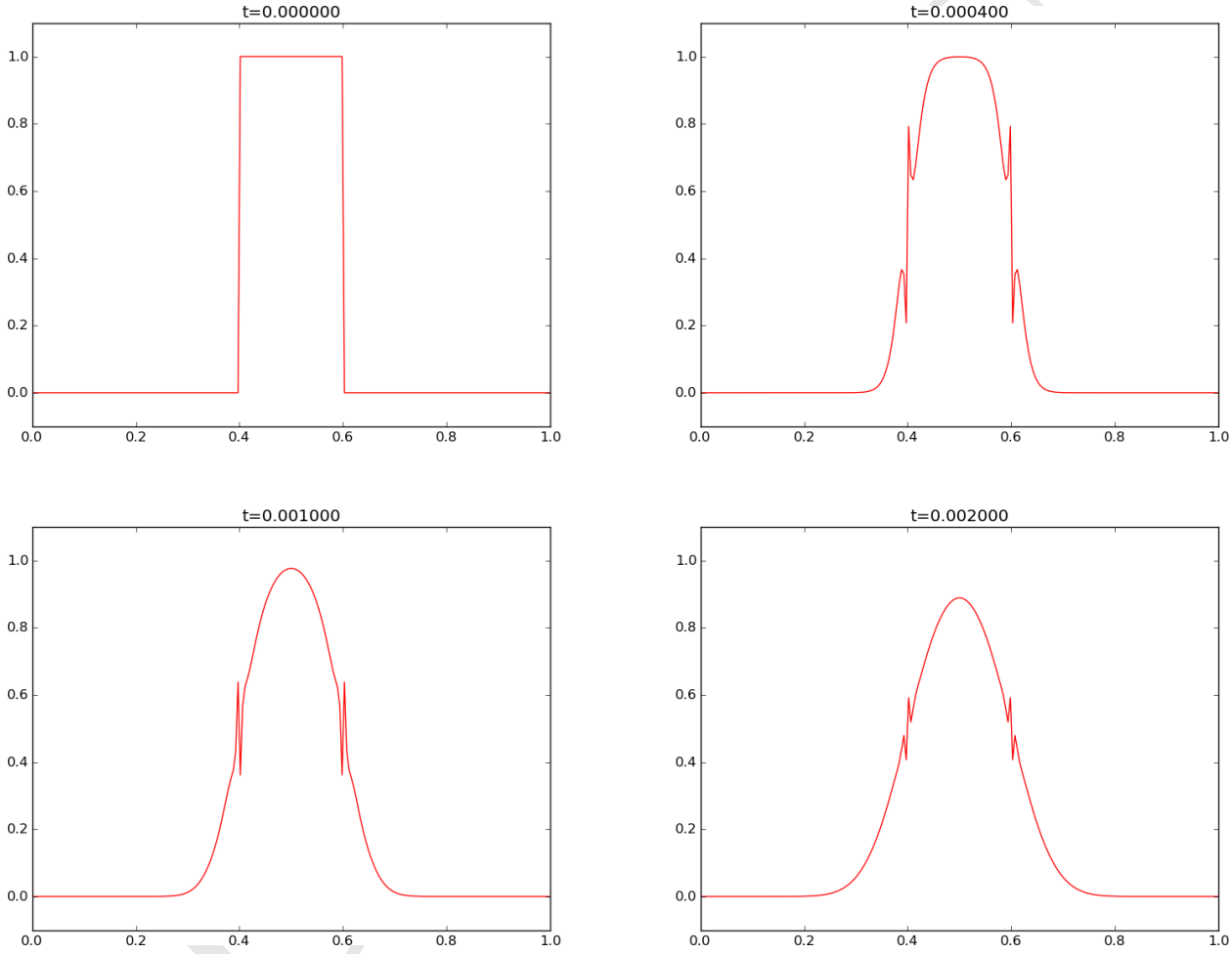


Figure 3.7.: Crank-Nicolson scheme for $F = 10$.

the limit, the Backward Euler scheme gives

$$-\frac{u_{i+1}^{n+1} - 2u^{n+1} * *i + u^{n+1} * *i - 1}{\Delta x^2} = f_i^{n+1},$$

which is nothing but the discretization $[-D_x D_x u = f]_i^{n+1} = 0$ of $-u_{xx} = f$.

The result above means that the Backward Euler scheme can solve the limit equation directly and hence produce a solution of the 1D Laplace equation. With the Forward Euler scheme we must do the time stepping since $\Delta t > \Delta x^2/\alpha$ is illegal and leads to instability. We may interpret this time stepping as solving the equation system from $-u_{xx} = f$ by iterating on a pseudo time variable.

3.14. Solving the Diffusion Equation with Devito

Having established the finite difference discretization of the diffusion equation, we now implement the Forward Euler scheme using Devito. The symbolic approach allows us to express the PDE directly and let Devito generate optimized code.

3.14.1. From Discretization to Devito

Recall the Forward Euler scheme for the diffusion equation:

$$u_i^{n+1} = u_i^n + F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

where the Fourier number $F = \alpha\Delta t/\Delta x^2$ must satisfy $F \leq 0.5$ for stability.

In Devito, we express this as the PDE $u_t = \alpha u_{xx}$ and let the framework derive the update formula automatically.

3.14.2. The Devito Implementation

```
from devito import Grid, TimeFunction, Eq, solve, Operator, Constant
import numpy as np

# Domain and discretization
L = 1.0           # Domain length
Nx = 100          # Grid points
a = 1.0           # Diffusion coefficient
F = 0.5           # Fourier number

dx = L / Nx
dt = F * dx**2 / a  # Time step from stability condition

# Create Devito grid
grid = Grid(shape=(Nx + 1,), extent=(L,))
```

```
# Time-varying temperature field
# time_order=1 for first-order time derivative
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)
```

3.14.3. Key Differences from the Wave Equation

Compare this to the wave equation setup:

Parameter	Wave Equation	Diffusion Equation
`time_order`	2 (for u_{tt})	1 (for u_t)
Time derivative	`.dt2`	`.dt`
Time levels	3 (u^{n-1}, u^n, u^{n+1})	2 (u^n, u^{n+1})
Stability number	Courant: $C = c\Delta t/\Delta x \leq 1$	Fourier: $F = \alpha\Delta t/\Delta x^2 \leq 0.5$

3.14.4. Symbolic PDE Definition

With `time_order=1`, Devito provides the `.dt` derivative:

```
# Diffusion coefficient as a Devito constant
a_const = Constant(name='a_const')

# PDE: u_t = a * u_xx  =>  u_t - a * u_xx = 0
pde = u.dt - a_const * u.dx2

# Solve for u at the forward time level
stencil = Eq(u.forward, solve(pde, u.forward))
```

When we print the stencil, we see:

```
print(stencil)
# Eq(u(t + dt, x), dt*a_const*u(t, x).dx2 + u(t, x))
```

This is exactly the Forward Euler update: $u^{n+1} = u^n + \Delta t \cdot \alpha \cdot u^n_{xx}$.

3.14.5. Boundary Conditions

For homogeneous Dirichlet conditions $u(0,t) = u(L,t) = 0$:

```
t_dim = grid.stepping_dim
bc_left = Eq(u[t_dim + 1, 0], 0)
bc_right = Eq(u[t_dim + 1, Nx], 0)
```

3.14.6. Complete Solver

The `src.diffu` module provides `solve_diffusion_1d`:

```
from src.diffu import solve_diffusion_1d
import numpy as np

# Initial condition: sinusoidal temperature profile
def I(x):
    return np.sin(np.pi * x)

result = solve_diffusion_1d(
    L=1.0,           # Domain length
    a=1.0,           # Diffusion coefficient
    Nx=100,          # Grid points
    T=0.1,           # Final time
    F=0.5,           # Fourier number (at stability limit)
    I=I,             # Initial condition
)

print(f"Final time: {result.t:.4f}")
print(f"Max temperature: {result.u.max():.6f}")
```

3.14.7. Verification with Exact Solution

For the initial condition $I(x) = \sin(\pi x/L)$, the exact solution is:

$$u(x,t) = e^{-\alpha(\pi/L)^2 t} \sin(\pi x/L)$$

This exponentially decaying sinusoid can verify our implementation:

```
from src.diffu import exact_diffusion_sine

# Compare numerical and exact solutions
u_exact = exact_diffusion_sine(result.x, result.t, L=1.0, a=1.0)
error = np.max(np.abs(result.u - u_exact))
print(f"Maximum error: {error:.2e}")
```

3.14.8. Convergence Testing

We verify second-order spatial accuracy:

```
from src.diffu import convergence_test_diffusion_1d

grid_sizes, errors, rate = convergence_test_diffusion_1d(
    grid_sizes=[10, 20, 40, 80],
    T=0.1,
    F=0.5,
)

print(f"Observed convergence rate: {rate:.2f}")  # Should approach 2.0
```

With F fixed, refining the grid means $\Delta x \to \Delta x/2$ and $\Delta t \to \Delta t/4$ (since $F = \alpha \Delta t/\Delta x^2$). The spatial error $O(\Delta x^2)$ dominates, giving second-order convergence.

3.14.9. Visualizing the Solution Evolution

```
import matplotlib.pyplot as plt

result = solve_diffusion_1d(
    L=1.0, a=1.0, Nx=100, T=0.5, F=0.5,
    save_history=True,
)

# Plot at several times
times_to_plot = [0, 0.1, 0.2, 0.3, 0.5]
plt.figure(figsize=(10, 6))

for t in times_to_plot:
    idx = int(t / result.dt)
    if idx < len(result.t_history):
        plt.plot(result.x, result.u_history[idx],
                 label=f't = {result.t_history[idx]:.2f}')

plt.xlabel('x')
plt.ylabel('u(x, t)')
plt.title('Diffusion of a Sinusoidal Profile')
plt.legend()
plt.grid(True)
```

The solution shows the characteristic behavior of the heat equation: the sinusoidal profile decays exponentially in time while maintaining its shape.

3.14.10. The Fourier Number and Physical Interpretation

The Fourier number $F = \alpha \Delta t/\Delta x^2$ has a physical interpretation. It represents the ratio of the diffusion time scale to the computational time step:

- **Large** F: Heat diffuses quickly relative to the time step
- **Small** F: Slow diffusion, finer time resolution

The stability limit $F \leq 0.5$ means we cannot take time steps larger than half the time for heat to diffuse across one grid cell.

3.14.11. Handling Different Initial Conditions

The diffusion equation smooths out discontinuities over time. Let's compare a smooth Gaussian and a discontinuous "plug":

```
from src.diffu import gaussian_initial_condition, plug_initial_condition

# Gaussian: smooth initial condition
result_gaussian = solve_diffusion_1d(
    L=1.0, Nx=100, T=0.1, F=0.5,
    I=lambda x: gaussian_initial_condition(x, L=1.0, sigma=0.05),
)

# Plug: discontinuous initial condition
result_plug = solve_diffusion_1d(
    L=1.0, Nx=100, T=0.1, F=0.5,
    I=lambda x: plug_initial_condition(x, L=1.0, width=0.1),
)
```

For smooth initial conditions, the Forward Euler scheme with $F = 0.5$ works well. For discontinuous initial conditions, a smaller Fourier number ($F \leq 0.25$) may be needed to avoid oscillations.

3.14.12. Summary

Key points for the diffusion equation with Devito:

1. Use `time_order=1` for the first-order time derivative
2. The `.dt` attribute provides the time derivative
3. The Fourier number $F = \alpha \Delta t / \Delta x^2$ must satisfy $F \leq 0.5$
4. The exact sinusoidal solution provides excellent verification
5. Smooth initial conditions work well at $F = 0.5$; discontinuous conditions may need smaller F

The Forward Euler scheme is simple and explicit, but the time step restriction can be severe for accuracy. In the next section, we discuss implicit methods that remove this restriction.

3.15. Analysis of schemes for the diffusion equation

The numerical experiments in Sections Section 3.6 and Section 3.12 reveal that there are some numerical problems with the Forward Euler and Crank-Nicolson schemes: sawtooth-like noise is sometimes present in solutions that are, from a mathematical point of view, expected to be smooth. This section presents a mathematical analysis that explains the observed behavior and arrives at criteria for obtaining numerical solutions that reproduce the qualitative properties of the exact solutions. In short, we shall explain what is observed in Figures Figure 3.13.7.

3.16. Properties of the solution

A particular characteristic of diffusive processes, governed by an equation like

$$u_t = \alpha u_{xx}, \tag{3.37}$$

is that the initial shape $u(x,0) = I(x)$ spreads out in space with time, along with a decaying amplitude. Three different examples will illustrate the spreading of u in space and the decay in time.

3.16.1. Similarity solution

The diffusion equation (3.37) admits solutions that depend on $\eta = (x-c)/\sqrt{4\alpha t}$ for a given value of c. One particular solution is

$$u(x,t) = a\,\mathrm{erf}(\eta) + b, \tag{3.38}$$

where

$$\mathrm{erf}(\eta) = \frac{2}{\sqrt{\pi}}\int_0^\eta e^{-\zeta^2}d\zeta, \tag{3.39}$$

is the *error function*, and a and b are arbitrary constants. The error function lies in $(-1,1)$, is odd around $\eta = 0$, and goes relatively quickly to ± 1:

$$\begin{aligned}
\lim_{\eta\to-\infty}\mathrm{erf}(\eta) &= -1,\\
\lim_{\eta\to\infty}\mathrm{erf}(\eta) &= 1,\\
\mathrm{erf}(\eta) &= -\mathrm{erf}(-\eta),\\
\mathrm{erf}(0) &= 0,\\
\mathrm{erf}(2) &= 0.99532227,\\
\mathrm{erf}(3) &= 0.99997791
\end{aligned}$$

.

As $t \to 0$, the error function approaches a step function centered at $x = c$. For a diffusion problem posed on the unit interval $[0,1]$, we may choose the step at $x = 1/2$ (meaning $c = 1/2$), $a = -1/2$, $b = 1/2$. Then

$$u(x,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right)\right) = \frac{1}{2}\mathrm{erfc}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right), \tag{3.40}$$

where we have introduced the *complementary error function* $\text{erfc}(\eta) = 1 - \text{erf}(\eta)$. The solution (3.40) implies the boundary conditions

$$u(0,t) = \frac{1}{2}\left(1 - \text{erf}\left(\frac{-1/2}{\sqrt{4\alpha t}}\right)\right), \tag{3.41}$$

$$u(1,t) = \frac{1}{2}\left(1 - \text{erf}\left(\frac{1/2}{\sqrt{4\alpha t}}\right)\right). \tag{3.42}$$

For small enough t, $u(0,t) \approx 1$ and $u(1,t) \approx 0$, but as $t \to \infty$, $u(x,t) \to 1/2$ on $[0,1]$.

3.16.2. Solution for a Gaussian pulse

The standard diffusion equation $u_t = \alpha u_{xx}$ admits a Gaussian function as solution:

$$u(x,t) = \frac{1}{\sqrt{4\pi\alpha t}} \exp\left(-\frac{(x-c)^2}{4\alpha t}\right). \tag{3.43}$$

At $t = 0$ this is a Dirac delta function, so for computational purposes one must start to view the solution at some time $t = t_\epsilon > 0$. Replacing t by $t_\epsilon + t$ in (3.43) makes it easy to operate with a (new) t that starts at $t = 0$ with an initial condition with a finite width. The important feature of (3.43) is that the standard deviation σ of a sharp initial Gaussian pulse increases in time according to $\sigma = \sqrt{2\alpha t}$, making the pulse diffuse and flatten out.

3.16.3. Solution for a sine component

Also, (3.37) admits a solution of the form

$$u(x,t) = Qe^{-\alpha t}\sin(kx). \tag{3.44}$$

The parameters Q and k can be freely chosen, while inserting (3.44) in (3.37) gives the constraint

$$a = -\alpha k^2.$$

A very important feature is that the initial shape $I(x) = Q\sin(kx)$ undergoes a damping $\exp(-\alpha k^2 t)$, meaning that rapid oscillations in space, corresponding to large k, are very much faster dampened than slow oscillations in space, corresponding to small k. This feature leads to a smoothing of the initial condition with time. (In fact, one can use a few steps of the diffusion equation as a method for removing noise in signal processing.) To judge how good a numerical method is, we may look at its ability to smoothen or dampen the solution in the same way as the PDE does.

The following example illustrates the damping properties of (3.44). We consider the specific problem

$$\begin{aligned} u_t &= u_{xx}, \quad x \in (0,1),\ t \in (0,T], \\ u(0,t) &= u(1,t) = 0, \quad t \in (0,T], \\ u(x,0) &= \sin(\pi x) + 0.1\sin(100\pi x) \end{aligned}$$

.The initial condition has been chosen such that adding two solutions like (3.44) constructs an analytical solution to the problem:

$$u(x,t) = e^{-\pi^2 t}\sin(\pi x) + 0.1e^{-\pi^2 10^4 t}\sin(100\pi x)\,. \tag{3.45}$$

Figure Figure 3.8 illustrates the rapid damping of rapid oscillations $\sin(100\pi x)$ and the very much slower damping of the slowly varying $\sin(\pi x)$ term. After about $t = 0.5 \cdot 10^{-4}$ the rapid oscillations do not have a visible amplitude, while we have to wait until $t \sim 0.5$ before the amplitude of the long wave $\sin(\pi x)$ becomes very small.

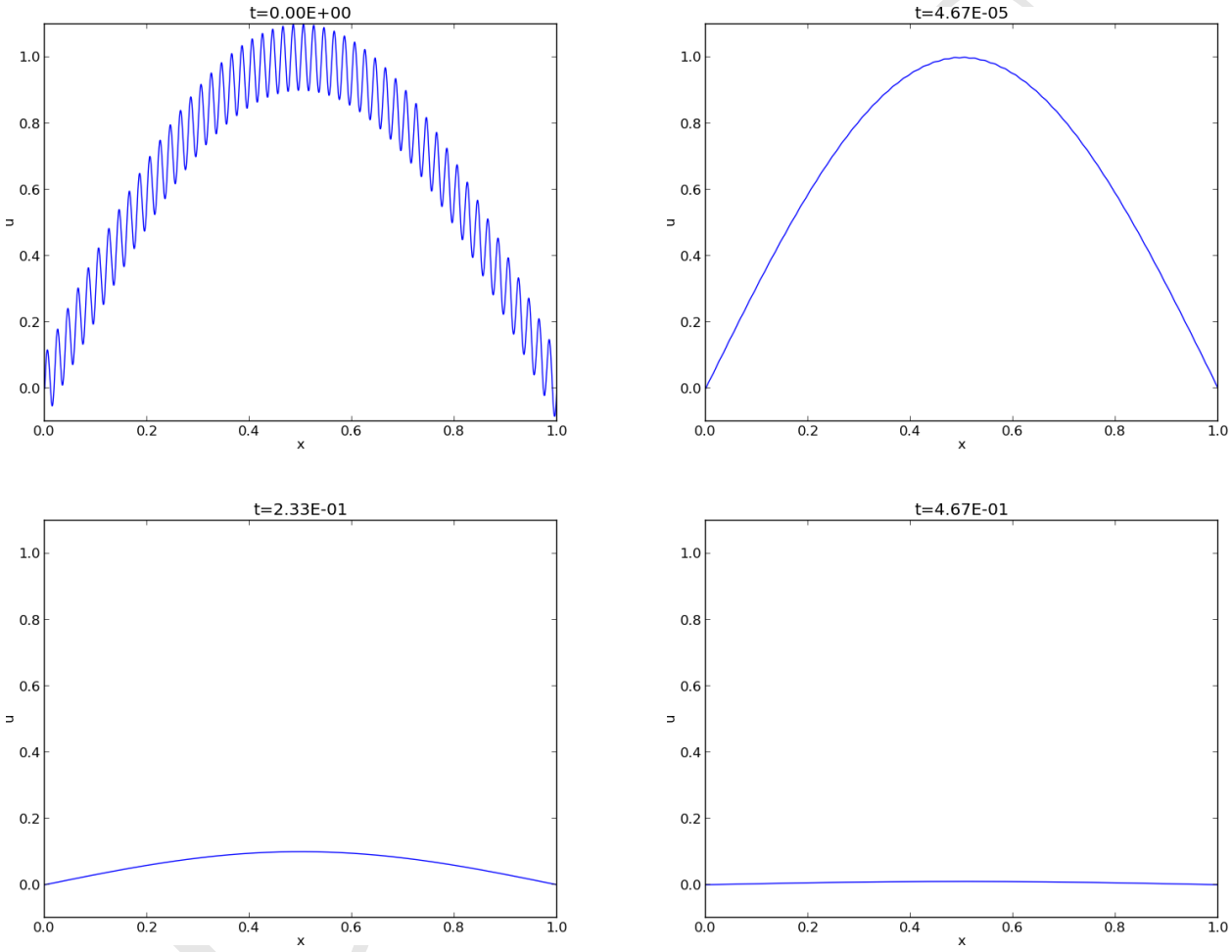


Figure 3.8.: Evolution of the solution of a diffusion problem: initial condition (upper left), 1/100 reduction of the small waves (upper right), 1/10 reduction of the long wave (lower left), and 1/100 reduction of the long wave (lower right).

3.17. Analysis of discrete equations

A counterpart to (3.44) is the complex representation of the same function:

$$u(x,t) = Qe^{-at}e^{ikx},$$

where $i = \sqrt{-1}$ is the imaginary unit. We can add such functions, often referred to as wave components, to make a Fourier representation of a general solution of the diffusion equation:

$$u(x,t) \approx \sum_{k\in K} b_k e^{-\alpha k^2 t} e^{ikx}, \tag{3.46}$$

where K is a set of an infinite number of k values needed to construct the solution. In practice, however, the series is truncated and K is a finite set of k values needed to build a good approximate solution. Note that (3.45) is a special case of (3.46) where $K = \{\pi, 100\pi\}$, $b_\pi = 1$, and $b_{100\pi} = 0.1$.

The amplitudes b_k of the individual Fourier waves must be determined from the initial condition. At $t = 0$ we have $u \approx \sum_k b_k \exp(ikx)$ and find K and b_k such that

$$I(x) \approx \sum_{k\in K} b_k e^{ikx}.$$

(The relevant formulas for b_k come from Fourier analysis, or equivalently, a least-squares method for approximating $I(x)$ in a function space with basis $\exp(ikx)$.)

Much insight about the behavior of numerical methods can be obtained by investigating how a wave component $\exp(-\alpha k^2 t)\exp(ikx)$ is treated by the numerical scheme. It appears that such wave components are also solutions of the schemes, but the damping factor $\exp(-\alpha k^2 t)$ varies among the schemes. To ease the forthcoming algebra, we write the damping factor as A^n. The exact amplification factor corresponding to A is $A_\mathrm{e} = \exp(-\alpha k^2 \Delta t)$.

3.18. Analysis of the finite difference schemes

We have seen that a general solution of the diffusion equation can be built as a linear combination of basic components

$$e^{-\alpha k^2 t} e^{ikx}.$$

A fundamental question is whether such components are also solutions of the finite difference schemes. This is indeed the case, but the amplitude $\exp(-\alpha k^2 t)$ might be modified (which also happens when solving the ODE counterpart $u' = -\alpha u$). We therefore look for numerical solutions of the form

$$u_q^n = A^n e^{ikq\Delta x} = A^n e^{ikx}, \tag{3.47}$$

where the amplification factor A must be determined by inserting the component into an actual scheme. Note that A^n means A raised to the power of n, n being the index in the time mesh, while the superscript n in u_q^n just denotes u at time t_n.

3.18.1. Stability

The exact amplification factor is $A_\mathrm{e} = \exp(-\alpha^2 k^2 \Delta t)$. We should therefore require $|A| < 1$ to have a decaying numerical solution as well. If $-1 \leq A < 0$, A^n will change sign from time level to time level, and we get stable, non-physical oscillations in the numerical solutions that are not present in the exact solution.

3.18.2. Accuracy

To determine how accurately a finite difference scheme treats one wave component (3.47), we see that the basic deviation from the exact solution is reflected in how well A^n approximates A_{e}^n, or how well A approximates A_{e}. We can plot A_{e} and the various expressions for A, and we can make Taylor expansions of A/A_{e} to see the error more analytically.

3.18.3. Truncation error

As an alternative to examining the accuracy of the damping of a wave component, we can perform a general truncation error analysis as explained in Chapter 7. Such results are more general, but less detailed than what we get from the wave component analysis. The truncation error can almost always be computed and represents the error in the numerical model when the exact solution is substituted into the equations. In particular, the truncation error analysis tells the order of the scheme, which is of fundamental importance when verifying codes based on empirical estimation of convergence rates.

3.19. Analysis of the Forward Euler scheme

The Forward Euler finite difference scheme for $u_t = \alpha u_{xx}$ can be written as

$$[D_t^+ u = \alpha D_x D_x u]_q^n .$$

Inserting a wave component (3.47) in the scheme demands calculating the terms

$$e^{ikq\Delta x}[D_t^+ A]^n = e^{ikq\Delta x} A^n \frac{A-1}{\Delta t},$$

and

$$A^n D_x D_x [e^{ikx}]_q = A^n \left(-e^{ikq\Delta x} \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right)\right) .$$

Inserting these terms in the discrete equation and dividing by $A^n e^{ikq\Delta x}$ leads to

$$\frac{A-1}{\Delta t} = -\alpha \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right),$$

and consequently

$$A = 1 - 4F \sin^2 p$$

where

$$F = \frac{\alpha \Delta t}{\Delta x^2}$$

is the *numerical Fourier number*, and $p = k\Delta x/2$. The complete numerical solution is then

$$u_q^n = \left(1 - 4F \sin^2 p\right)^n e^{ikq\Delta x} .$$

Stability We easily see that $A \leq 1$. However, the A can be less than -1, which will lead to growth of a numerical wave component. The criterion $A \geq -1$ implies

$$4F \sin^2(p/2) \leq 2 .$$

The worst case is when $\sin^2(p/2) = 1$, so a sufficient criterion for stability is

$$F \leq \frac{1}{2},$$

or expressed as a condition on Δt:

$$\Delta t \leq \frac{\Delta x^2}{2\alpha}\,.$$

Note that halving the spatial mesh size, $\Delta x \to \frac{1}{2}\Delta x$, requires Δt to be reduced by a factor of $1/4$. The method hence becomes very expensive for fine spatial meshes.

3.19.1. Accuracy

Since A is expressed in terms of F and the parameter we now call $p = k\Delta x/2$, we should also express A_e by F and p. The exponent in A_e is $-\alpha k^2 \Delta t$, which equals $-Fk^2\Delta x^2 = -F4p^2$. Consequently,

$$A_\mathrm{e} = \exp\left(-\alpha k^2 \Delta t\right) = \exp\left(-4Fp^2\right).$$

All our A expressions as well as A_e are now functions of the two dimensionless parameters F and p.

Computing the Taylor series expansion of A/A_e in terms of F can easily be done with aid of `sympy`:

```
def A_exact(F, p):
    return exp(-4*F*p**2)

def A_FE(F, p):
    return 1 - 4*F*sin(p)**2

from sympy import *
F, p = symbols('F p')
A_err_FE = A_FE(F, p)/A_exact(F, p)
print A_err_FE.series(F, 0, 6)
```

The result is

$$\frac{A}{A_\mathrm{e}} = 1 - 4F\sin^2 p + 2Fp^2 - 16F^2p^2\sin^2 p + 8F^2p^4 + \cdots$$

Recalling that $F = \alpha\Delta t/\Delta x^2$, $p = k\Delta x/2$, and that $\sin^2 p \leq 1$, we realize that the dominating terms in A/A_e are at most

$$1 - 4\alpha\frac{\Delta t}{\Delta x^2} + \alpha\Delta t - 4\alpha^2\Delta t^2 + \alpha^2\Delta t^2\Delta x^2 + \cdots\,.$$

Truncation error We follow the theory explained in Chapter 7. The recipe is to set up the scheme in operator notation and use formulas from Section 7.7 to derive an expression for the residual. The details are documented in Section 7.27. We end up with a truncation error

$$R_i^n = \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)\,.$$

Although this is not the true error $u_\mathrm{e}(x_i, t_n) - u_i^n$, it indicates that the true error is of the form

$$E = C_t\Delta t + C_x\Delta x^2$$

for two unknown constants C_t and C_x.

3.20. Analysis of the Backward Euler scheme

Discretizing $u_t = \alpha u_{xx}$ by a Backward Euler scheme,

$$[D_t^- u = \alpha D_x D_x u]_q^n,$$

and inserting a wave component (3.47), leads to calculations similar to those arising from the Forward Euler scheme, but since

$$e^{ikq\Delta x}[D_t^- A]^n = A^n e^{ikq\Delta x}\frac{1 - A^{-1}}{\Delta t},$$

we get

$$\frac{1 - A^{-1}}{\Delta t} = -\alpha\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right),$$

and then

$$A = \left(1 + 4F\sin^2 p\right)^{-1}. \tag{3.48}$$

The complete numerical solution can be written

$$u_q^n = \left(1 + 4F\sin^2 p\right)^{-n} e^{ikq\Delta x}.$$

Stability We see from (3.48) that $0 < A < 1$, which means that all numerical wave components are stable and non-oscillatory for any $\Delta t > 0$.

3.20.1. Truncation error

The derivation of the truncation error for the Backward Euler scheme is almost identical to that for the Forward Euler scheme. We end up with

$$R_i^n = \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2).$$

3.21. Analysis of the Crank-Nicolson scheme

The Crank-Nicolson scheme can be written as

$$[D_t u = \alpha D_x D_x \overline{u}^x]_q^{n+\frac{1}{2}},$$

or

$$[D_t u]_q^{n+\frac{1}{2}} = \frac{1}{2}\alpha\left([D_x D_x u]_q^n + [D_x D_x u]_q^{n+1}\right).$$

Inserting (3.47) in the time derivative approximation leads to

$$[D_t A^n e^{ikq\Delta x}]^{n+\frac{1}{2}} = A^{n+\frac{1}{2}} e^{ikq\Delta x}\frac{A^{\frac{1}{2}} - A^{-\frac{1}{2}}}{\Delta t} = A^n e^{ikq\Delta x}\frac{A - 1}{\Delta t}.$$

Inserting (3.47) in the other terms and dividing by $A^n e^{ikq\Delta x}$ gives the relation

$$\frac{A - 1}{\Delta t} = -\frac{1}{2}\alpha\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)(1 + A),$$

and after some more algebra,

$$A = \frac{1 - 2F\sin^2 p}{1 + 2F\sin^2 p}\,.$$

The exact numerical solution is hence

$$u_q^n = \left(\frac{1 - 2F\sin^2 p}{1 + 2F\sin^2 p}\right)^n e^{ikq\Delta x}\,.$$

Stability The criteria $A > -1$ and $A < 1$ are fulfilled for any $\Delta t > 0$. Therefore, the solution cannot grow, but it will oscillate if $1 - 2F\sin^p < 0$. To avoid such non-physical oscillations, we must demand $F \leq \frac{1}{2}$.

3.21.1. Truncation error

The truncation error is derived in Section 7.27:

$$R_i^{n+\frac{1}{2}} = \mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta t^2)\,.$$

Analysis of the Leapfrog scheme {#sec-diffu-pde1-analysis-leapfrog}

An attractive feature of the Forward Euler scheme is the explicit time stepping and no need for solving linear systems. However, the accuracy in time is only $\mathcal{O}(\Delta t)$. We can get an explicit *second-order* scheme in time by using the Leapfrog method:

$$[D_{2t}u = \alpha D_x D x u + f]_q^n\,.$$

Written out,

$$u_q^{n+1} = u_q^{n-1} + \frac{2\alpha\Delta t}{\Delta x^2}(u_{q+1}^n - 2u_q^n + u_{q-1}^n) + f(x_q, t_n)\,.$$

We need some formula for the first step, u_q^1, but for that we can use a Forward Euler step.

Unfortunately, the Leapfrog scheme is always unstable for the diffusion equation. To see this, we insert a wave component $A^n e^{ikx}$ and get

$$\frac{A - A^{-1}}{\Delta t} = -\alpha\frac{4}{\Delta x^2}\sin^2 p,$$

or

$$A^2 + 4F\sin^2 p\, A - 1 = 0,$$

which has roots

$$A = -2F\sin^2 p \pm \sqrt{4F^2\sin^4 p + 1}\,.$$

Both roots have $|A| > 1$ so the amplitude always grows, which is not in accordance with the physics of the problem. However, for a PDE with a first-order derivative in space, instead of a second-order one, the Leapfrog scheme performs very well. Details are provided in Section Section 4.4.1.

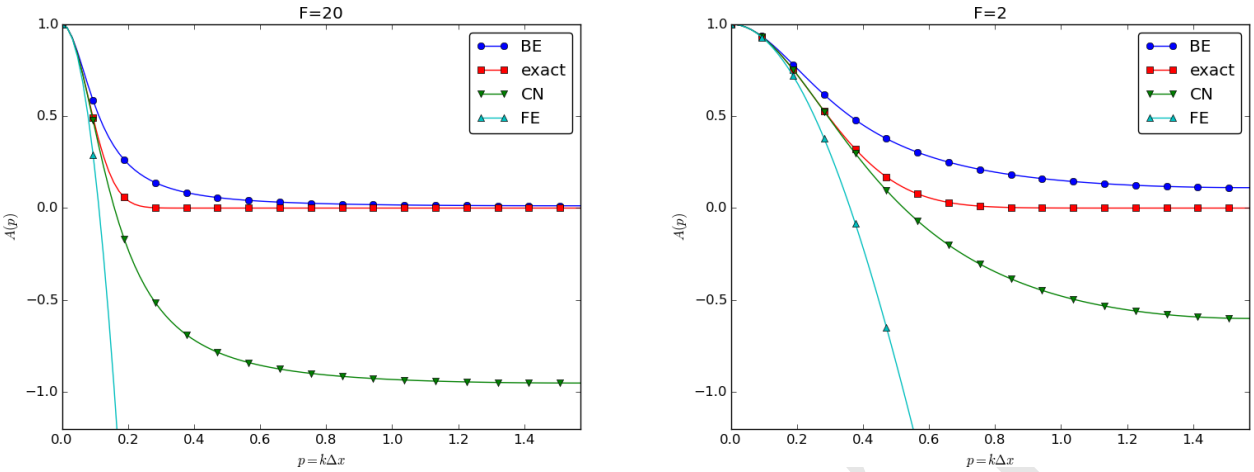


Figure 3.9.: Amplification factors for large time steps.

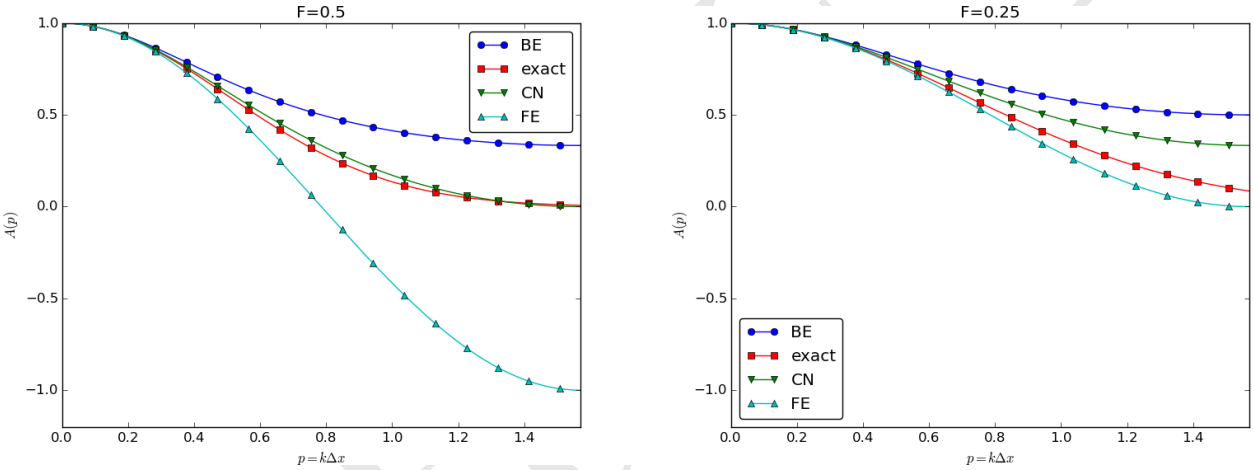


Figure 3.10.: Amplification factors for time steps around the Forward Euler stability limit.

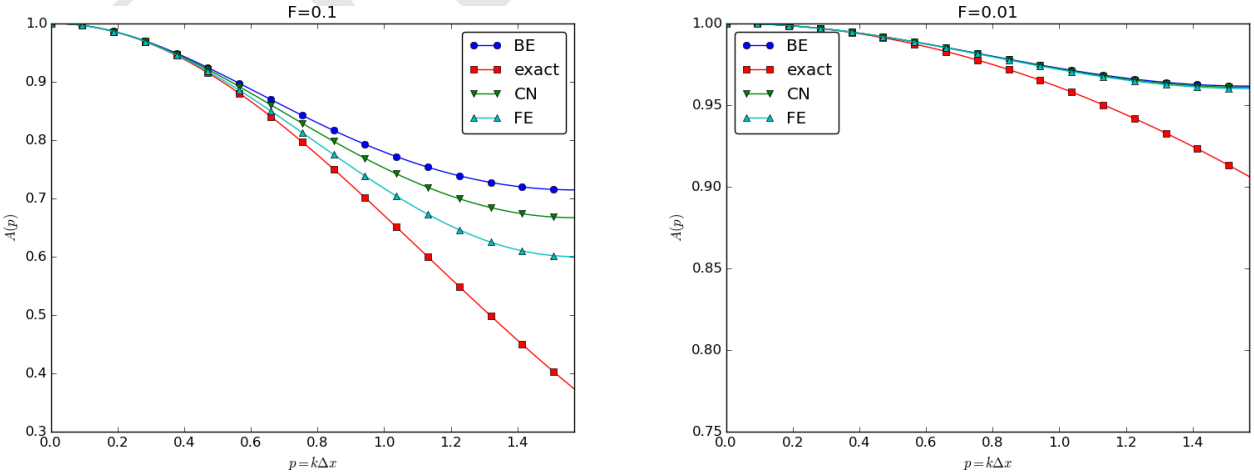


Figure 3.11.: Amplification factors for small time steps.

3.22. Summary of accuracy of amplification factors

We can plot the various amplification factors against $p = k\Delta x/2$ for different choices of the F parameter. Figures Figure 3.9, Figure 3.10, and Figure 3.11 show how long and small waves are damped by the various schemes compared to the exact damping. As long as all schemes are stable, the amplification factor is positive, except for Crank-Nicolson when $F > 0.5$.

The effect of negative amplification factors is that A^n changes sign from one time level to the next, thereby giving rise to oscillations in time in an animation of the solution. We see from Figure Figure 3.9 that for $F = 20$, waves with $p \geq \pi/4$ undergo a damping close to -1, which means that the amplitude does not decay and that the wave component jumps up and down (flips amplitude) in time. For $F = 2$ we have a damping of a factor of 0.5 from one time level to the next, which is very much smaller than the exact damping. Short waves will therefore fail to be effectively dampened. These waves will manifest themselves as high frequency oscillatory noise in the solution.

A value $p = \pi/4$ corresponds to four mesh points per wave length of e^{ikx}, while $p = \pi/2$ implies only two points per wave length, which is the smallest number of points we can have to represent the wave on the mesh.

To demonstrate the oscillatory behavior of the Crank-Nicolson scheme, we choose an initial condition that leads to short waves with significant amplitude. A discontinuous $I(x)$ will in particular serve this purpose: Figures Figure 3.6 and Figure 3.7 correspond to $F = 3$ and $F = 10$, respectively, and we see how short waves pollute the overall solution.

3.23. Analysis of the 2D diffusion equation

Diffusion in several dimensions is treated later, but it is appropriate to include the analysis here. We first consider the 2D diffusion equation

$$u_t = \alpha(u_{xx} + u_{yy}),$$

which has Fourier component solutions of the form

$$u(x, y, t) = Ae^{-\alpha k^2 t}e^{i(k_x x + k_y y)},$$

and the schemes have discrete versions of this Fourier component:

$$u^n_{q,r} = A\xi^n e^{i(k_x q\Delta x + k_y r\Delta y)}.$$

The Forward Euler scheme For the Forward Euler discretization,

$$[D^+_t u = \alpha(D_x D_x u + D_y D_y u)]^n_{q,r},$$

we get

$$\frac{\xi - 1}{\Delta t} = -\alpha\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) - \alpha\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right).$$

Introducing

$$p_x = \frac{k_x\Delta x}{2}, \quad p_y = \frac{k_y\Delta y}{2},$$

we can write the equation for ξ more compactly as

$$\frac{\xi - 1}{\Delta t} = -\alpha \frac{4}{\Delta x^2} \sin^2 p_x - \alpha \frac{4}{\Delta y^2} \sin^2 p_y,$$

and solve for ξ:

$$\xi = 1 - 4F_x \sin^2 p_x - 4F_y \sin^2 p_y \,. \tag{3.49}$$

The complete numerical solution for a wave component is

$$u^n_{q,r} = A(1 - 4F_x \sin^2 p_x - 4F_y \sin^2 p_y)^n e^{i(k_x q \Delta x + k_y r \Delta y)} \,. \tag{3.50}$$

For stability we demand $-1 \leq \xi \leq 1$, and $-1 \leq \xi$ is the critical limit, since clearly $\xi \leq 1$, and the worst case happens when the sines are at their maximum. The stability criterion becomes

$$F_x + F_y \leq \frac{1}{2} \,. \tag{3.51}$$

For the special, yet common, case $\Delta x = \Delta y = h$, the stability criterion can be written as

$$\Delta t \leq \frac{h^2}{2d\alpha},$$

where d is the number of space dimensions: $d = 1, 2, 3$.

3.23.1. The Backward Euler scheme

The Backward Euler method,

$$[D_t^- u = \alpha(D_x D_x u + D_y D_y u)]^n_{q,r},$$

results in

$$1 - \xi^{-1} = -4F_x \sin^2 p_x - 4F_y \sin^2 p_y,$$

and

$$\xi = (1 + 4F_x \sin^2 p_x + 4F_y \sin^2 p_y)^{-1},$$

which is always in $(0, 1]$. The solution for a wave component becomes

$$u^n_{q,r} = A(1 + 4F_x \sin^2 p_x + 4F_y \sin^2 p_y)^{-n} e^{i(k_x q \Delta x + k_y r \Delta y)} \,. \tag{3.52}$$

3.23.2. The Crank-Nicolson scheme

With a Crank-Nicolson discretization,

$$[D_t u]^{n+\frac{1}{2}}_{q,r} = \frac{1}{2}[\alpha(D_x D_x u + D_y D_y u)]^{n+1}_{q,r} + \frac{1}{2}[\alpha(D_x D_x u + D_y D_y u)]^n_{q,r},$$

we have, after some algebra,

$$\xi = \frac{1 - 2(F_x \sin^2 p_x + F_x \sin^2 p_y)}{1 + 2(F_x \sin^2 p_x + F_x \sin^2 p_y)} \,.$$

The fraction on the right-hand side is always less than 1, so stability in the sense of non-growing wave components is guaranteed for all physical and numerical parameters. However, the fraction can become negative and result in non-physical oscillations. This phenomenon happens when

$$F_x \sin^2 p_x + F_x \sin^2 p_y > \frac{1}{2}\,.$$

A criterion against non-physical oscillations is therefore

$$F_x + F_y \leq \frac{1}{2},$$

which is the same limit as the stability criterion for the Forward Euler scheme.

The exact discrete solution is

$$u^n_{q,r} = A\left(\frac{1 - 2(F_x \sin^2 p_x + F_x \sin^2 p_y)}{1 + 2(F_x \sin^2 p_x + F_x \sin^2 p_y)}\right)^n e^{i(k_x q\Delta x + k_y r\Delta y)}\,. \tag{3.53}$$

3.24. Explanation of numerical artifacts

The behavior of the solution generated by Forward Euler discretization in time (and centered differences in space) is summarized at the end of Section Section 3.6. Can we, from the analysis above, explain the behavior?

We may start by looking at Figure Figure 3.3 where $F = 0.51$. The figure shows that the solution is unstable and grows in time. The stability limit for such growth is $F = 0.5$ and since the F in this simulation is slightly larger, growth is unavoidable.

Figure Figure 3.1 has unexpected features: we would expect the solution of the diffusion equation to be smooth, but the graphs in Figure Figure 3.1 contain non-smooth noise. Turning to Figure Figure 3.4, which has a quite similar initial condition, we see that the curves are indeed smooth. The problem with the results in Figure Figure 3.1 is that the initial condition is discontinuous. To represent it, we need a significant amplitude on the shortest waves in the mesh. However, for $F = 0.5$, the shortest wave ($p = \pi/2$) gives the amplitude in the numerical solution as $(1 - 4F)^n$, which oscillates between negative and positive values at subsequent time levels for $F > \frac{1}{4}$. Since the shortest waves have visible amplitudes in the solution profile, the oscillations becomes visible. The smooth initial condition in Figure Figure 3.4, on the other hand, leads to very small amplitudes of the shortest waves. That these waves then oscillate in a non-physical way for $F = 0.5$ is not a visible effect. The oscillations in time in the amplitude $(1 - 4F)^n$ disappear for $F \leq \frac{1}{4}$, and that is why also the discontinuous initial condition always leads to smooth solutions in Figure Figure 3.2, where $F = \frac{1}{4}$.

Turning the attention to the Backward Euler scheme and the experiments in Figure Figure 3.5, we see that even the discontinuous initial condition gives smooth solutions for $F = 0.5$ (and in fact all other F values). From the exact expression of the numerical amplitude, $(1 + 4F \sin^2 p)^{-1}$, we realize that this factor can never flip between positive and negative values, and no instabilities can occur. The conclusion is that the Backward Euler scheme always produces smooth solutions. Also, the Backward Euler scheme guarantees that the solution cannot grow in time (unless we add a source term to the PDE, but that is meant to represent a physically relevant growth).

Finally, we have some small, strange artifacts when simulating the development of the initial plug profile with the Crank-Nicolson scheme, see Figure Figure 3.7, where $F = 3$. The Crank-Nicolson scheme cannot give growing amplitudes, but it may give oscillating amplitudes in time. The critical factor is $1 - 2F\sin^2 p$, which for the shortest waves ($p = \pi/2$) indicates a stability limit $F = 0.5$. With the discontinuous initial condition, we have enough amplitude on the shortest waves so their wrong behavior is visible, and this is what we see as small instabilities in Figure Figure 3.7. The only remedy is to lower the F value.

3.25. Exercise: Explore symmetry in a 1D problem

This exercise simulates the exact solution (3.43). Suppose for simplicity that $c = 0$.

a)

Formulate an initial-boundary value problem that has (3.43) as solution in the domain $[-L, L]$. Use the exact solution (3.43) as Dirichlet condition at the boundaries. Simulate the diffusion of the Gaussian peak. Observe that the solution is symmetric around $x = 0$.

b)

Show from (3.43) that $u_x(c, t) = 0$. Since the solution is symmetric around $x = c = 0$, we can solve the numerical problem in half of the domain, using a *symmetry boundary condition* $u_x = 0$ at $x = 0$. Set up the initial-boundary value problem in this case. Simulate the diffusion problem in $[0, L]$ and compare with the solution in a).

Solution

$$
\begin{aligned}
u_t &= \alpha u_x x, \\
u_x(0,t) &= 0, \\
u(L,t) &= \frac{1}{\sqrt{4\pi\alpha t}} \exp\left(-\frac{x^2}{4\alpha t}\right).
\end{aligned}
$$

3.26. Exercise: Investigate approximation errors from a $u_x = 0$ boundary condition

We consider the problem solved in Exercise Section 3.25 part b). The boundary condition $u_x(0, t) = 0$ can be implemented in two ways: 1) by a standard symmetric finite difference $[D_{2x}u]_i^n = 0$, or 2) by a one-sided difference $[D^+u = 0]_i^n = 0$. Investigate the effect of these two conditions on the convergence rate in space.

If you use a Forward Euler scheme, choose a discretization parameter

$h = \Delta t = \Delta x^2$ and assume the error goes like $E \sim h^r$. The error in the scheme is $\mathcal{O}(\Delta t, \Delta x^2)$ so one should expect that the estimated r approaches 1. The question is if a one-sided difference

approximation to $u_x(0,t) = 0$ destroys this convergence rate.

3.27. Exercise: Experiment with open boundary conditions in 1D

We address diffusion of a Gaussian function as in Exercise Section 3.25, in the domain $[0, L]$, but now we shall explore different types of boundary conditions on $x = L$. In real-life problems we do not know the exact solution on $x = L$ and must use something simpler.

a)

Imagine that we want to solve the problem numerically on $[0, L]$, with a symmetry boundary condition $u_x = 0$ at $x = 0$, but we do not know the exact solution and cannot of that reason assign a correct Dirichlet condition at $x = L$. One idea is to simply set $u(L, t) = 0$ since this will be an accurate approximation before the diffused pulse reaches $x = L$ and even thereafter it might be a satisfactory condition if the exact u has a small value. Let u_e be the exact solution and let u be the solution of $u_t = \alpha u_{xx}$ with an initial Gaussian pulse and the boundary conditions $u_x(0,t) = u(L,t) = 0$. Derive a diffusion problem for the error $e = u_\mathrm{e} - u$. Solve this problem numerically using an exact Dirichlet condition at $x = L$. Animate the evolution of the error and make a curve plot of the error measure

$$E(t) = \sqrt{\frac{\int_0^L e^2 dx}{\int_0^L u dx}}.$$

Is this a suitable error measure for the present problem?

b)

Instead of using $u(L, t) = 0$ as approximate boundary condition for letting the diffused Gaussian pulse move out of our finite domain, one may try $u_x(L, t) = 0$ since the solution for large t is quite flat. Argue that this condition gives a completely wrong asymptotic solution as $t \to 0$. To do this, integrate the diffusion equation from 0 to L, integrate u_{xx} by parts (or use Gauss' divergence theorem in 1D) to arrive at the important property

$$\frac{d}{dt}\int_0^L u(x,t)dx = 0,$$

implying that $\int_0^L u dx$ must be constant in time, and therefore

$$\int_0^L u(x,t)dx = \int_0^L I(x)dx\,.$$

The integral of the initial pulse is 1.

c)

Another idea for an artificial boundary condition at $x = L$ is to use a cooling law

$$-\alpha u_x = q(u - u_S), \tag{3.54}$$

where q is an unknown heat transfer coefficient and u_S is the surrounding temperature in the medium outside of $[0, L]$. (Note that arguing that u_S is approximately $u(L, t)$ gives the $u_x = 0$

condition from the previous subexercise that is qualitatively wrong for large t.) Develop a diffusion problem for the error in the solution using (3.54) as boundary condition. Assume one can take $u_S = 0$ "outside the domain" since $u_e \to 0$ as $x \to \infty$. Find a function $q = q(t)$ such that the exact solution obeys the condition (3.54). Test some constant values of q and animate how the corresponding error function behaves. Also compute $E(t)$ curves as defined above.

3.28. Exercise: Simulate a diffused Gaussian peak in 2D/3D

a)

Generalize (3.43) to multi dimensions by assuming that one-dimensional solutions can be multiplied to solve $u_t = \alpha\nabla^2 u$. Set $c = 0$ such that the peak of the Gaussian is at the origin.

b)

One can from the exact solution show that $u_x = 0$ on $x = 0$, $u_y = 0$ on $y = 0$, and $u_z = 0$ on $z = 0$. The approximately correct condition $u = 0$ can be set on the remaining boundaries (say $x = L$, $y = L$, $z = L$), cf. Exercise Section 3.27. Simulate a 2D case and make an animation of the diffused Gaussian peak.

c)

The formulation in b) makes use of symmetry of the solution such that we can solve the problem in the first quadrant (2D) or octant (3D) only. To check that the symmetry assumption is correct, formulate the problem without symmetry in a domain $[-L, L] \times [L, L]$ in 2D. Use $u = 0$ as approximately correct boundary condition. Simulate the same case as in b), but in a four times as large domain. Make an animation and compare it with the one in b).

3.29. Exercise: Examine stability of a diffusion model with a source term

Consider a diffusion equation with a linear u term:

$$u_t = \alpha u_{xx} + \beta u\,.$$

a)

Derive in detail the Forward Euler, Backward Euler, and Crank-Nicolson schemes for this type of diffusion model. Thereafter, formulate a θ-rule to summarize the three schemes.

b)

Assume a solution like (3.44) and find the relation between a, k, α, and β.

Insert (3.44) in the PDE problem.

c)

Calculate the stability of the Forward Euler scheme. Design numerical experiments to confirm the results.

Hint

Insert the discrete counterpart to (3.44) in the numerical scheme. Run experiments at the stability limit and slightly above.

d)

Repeat c) for the Backward Euler scheme.

e)

Repeat c) for the Crank-Nicolson scheme.

f)

How does the extra term bu impact the accuracy of the three schemes?

For analysis of the accuracy,

compare the numerical and exact amplification factors, in graphs and/or by Taylor series expansion.

3.30. Diffusion with variable coefficient

Diffusion in heterogeneous media normally implies a non-constant diffusion coefficient $\alpha = \alpha(x)$. A 1D diffusion model with such a variable diffusion coefficient reads

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(x)\frac{\partial u}{\partial x}\right) + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{3.55}$$

$$u(x,0) = I(x), \qquad x \in [0,L], \tag{3.56}$$

$$u(0,t) = U_0, \qquad t > 0, \tag{3.57}$$

$$u(L,t) = U_L, \qquad t > 0. \tag{3.58}$$

A short form of the diffusion equation with variable coefficients is $u_t = (\alpha u_x)_x + f$.

3.31. Discretization

We can discretize the diffusion equation $u_t = (\alpha u_x)_x + f$ by a θ-rule in time and centered differences in space:

$$[D_t u]_i^{n+\frac{1}{2}} = \theta[D_x(\overline{\alpha}^x D_x u) + f]_i^{n+1} + (1-\theta)[D_x(\overline{\alpha}^x D_x u) + f]_i^n\,.$$

Written out, this becomes

$$\frac{u_i^{n+1}-u_i^n}{\Delta t} = \theta\frac{1}{\Delta x^2}(\alpha_{i+\frac{1}{2}}(u^{n+1}**i+1-u^{n+1}**i)-\alpha_{i-\frac{1}{2}}(u^{n+1}**i-u^{n+1}**i-1))+ (1-\theta)\frac{1}{\Delta x^2}(\alpha_{i+\frac{1}{2}}(u^{n}**i+1-u^{n}**i)-\alpha_{i-\frac{1}{2}}(u^{n}**i-u^{n}**i-1))+ \theta f_i^{n+1}+(1-\theta)f_i^n,$$

where, e.g., an arithmetic mean can to be used for $\alpha_{i+\frac{1}{2}}$:

$$\alpha_{i+\frac{1}{2}} = \frac{1}{2}(\alpha_i+\alpha_{i+1})\,.$$

Implementation {#sec-diffu-varcoeff-impl}

Suitable code for solving the discrete equations is very similar to what we created for a constant α. Since the Fourier number has no meaning for varying α, we introduce a related parameter $D = \Delta t/\Delta x^2$.

```
def solver_theta(I, a, L, Nx, D, T, theta=0.5, u_L=1, u_R=0,
                 user_action=None):
    x = linspace(0, L, Nx+1)   # mesh points in space
    dx = x[1] - x[0]
    dt = D*dx**2
    Nt = int(round(T/float(dt)))
    t = linspace(0, T, Nt+1)   # mesh points in time

    u   = zeros(Nx+1)   # solution array at t[n+1]
    u_n = zeros(Nx+1)   # solution at t[n]

    Dl = 0.5*D*theta
    Dr = 0.5*D*(1-theta)

    diagonal = zeros(Nx+1)
    lower    = zeros(Nx)
    upper    = zeros(Nx)
    b        = zeros(Nx+1)

    diagonal[1:-1] = 1 + Dl*(a[2:] + 2*a[1:-1] + a[:-2])
    lower[:-1] = -Dl*(a[1:-1] + a[:-2])
    upper[1:]  = -Dl*(a[2:] + a[1:-1])
    diagonal[0] = 1
    upper[0] = 0
    diagonal[Nx] = 1
    lower[-1] = 0

    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
```

```
        shape=(Nx+1, Nx+1),
        format='csr')

    for i in range(0,Nx+1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        b[1:-1] = u_n[1:-1] + Dr*(
            (a[2:] + a[1:-1])*(u_n[2:] - u_n[1:-1]) -
            (a[1:-1] + a[0:-2])*(u_n[1:-1] - u_n[:-2]))
        b[0]  = u_L(t[n+1])
        b[-1] = u_R(t[n+1])
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, n+1)

        u_n, u = u, u_n
```

The code is found in the file diffu1D_vc.py.

3.32. Stationary solution

As $t \to \infty$, the solution of the variable-coefficient diffusion problem will approach a stationary limit where $\partial u/\partial t = 0$. The governing equation is then

$$\frac{d}{dx}\left(\alpha \frac{du}{dx}\right) = 0, \tag{3.59}$$

with boundary conditions $u(0) = U_0$ and $u(L) = U_L$. It is possible to obtain an exact solution of (3.59) for any α. Integrating twice and applying the boundary conditions to determine the integration constants gives

$$u(x) = U_0 + (U_L - U_0)\frac{\int_0^x (\alpha(\xi))^{-1} d\xi}{\int_0^L (\alpha(\xi))^{-1} d\xi}. \tag{3.60}$$

3.33. Piecewise constant medium

Consider a medium built of M layers. The layer boundaries are denoted $b_0, \ldots, b_M$, where $b_0 = 0$ and $b_M = L$. If the layers potentially have different material properties, but these properties are

constant within each layer, we can express α as a *piecewise constant function* according to

$$\alpha(x) = \begin{cases} \alpha_0, & b_0 \leq x < b_1, \\ \vdots & \\ \alpha_i, & b_i \leq x < b_{i+1}, \\ \vdots & \\ \alpha_{M-1}, & b_{M-1} \leq x \leq b_M. \end{cases} \tag{3.61}$$

The exact solution (3.60) in case of such a piecewise constant α function is easy to derive. Assume that x is in the m-th layer: $x \in [b_m, b_{m+1}]$. In the integral $\int_0^x (a(\xi))^{-1} d\xi$ we must integrate through the first $m-1$ layers and then add the contribution from the remaining part $x - b_m$ into the m-th layer:

$$u(x) = U_0 + (U_L - U_0)\frac{\sum_{j=0}^{m-1}(b_{j+1} - b_j)/\alpha(b_j) + (x - b_m)/\alpha(b_m)}{\sum_{j=0}^{M-1}(b_{j+1} - b_j)/\alpha(b_j)} \tag{3.62}$$

Remark. It may sound strange to have a discontinuous α in a differential equation where one is to differentiate, but a discontinuous α is compensated by a discontinuous u_x such that αu_x is continuous and therefore can be differentiated as $(\alpha u_x)_x$.

3.34. Implementation of diffusion in a piecewise constant medium

Programming with piecewise function definitions quickly becomes cumbersome as the most naive approach is to test for which interval x lies, and then start evaluating a formula like (3.62). In Python, vectorized expressions may help to speed up the computations. The convenience classes `PiecewiseConstant` and `IntegratedPiecewiseConstant` in the `Heaviside` module were made to simplify programming with functions like (3.61) and expressions like (3.62). These utilities not only represent piecewise constant functions, but also *smoothed* versions of them where the discontinuities can be smoothed out in a controlled fashion.

The `PiecewiseConstant` class is created by sending in the domain as a 2-tuple or 2-list and a `data` object describing the boundaries $b_0, \ldots, b_M$ and the corresponding function values $\alpha_0, \ldots, \alpha_{M-1}$. More precisely, `data` is a nested list, where `data[i][0]` holds b_i and `data[i][1]` holds the corresponding value α_i, for $i = 0, \ldots, M-1$. Given b_i and α_i in arrays `b` and `a`, it is easy to fill out the nested list `data`.

In our application, we want to represent α and $1/\alpha$ as piecewise constant functions, in addition to the $u(x)$ function which involves the integrals of $1/\alpha$. A class creating the functions we need and a method for evaluating u, can take the form

```
class SerialLayers:
    """
    b: coordinates of boundaries of layers, b[0] is left boundary
    and b[-1] is right boundary of the domain [0,L].
    a: values of the functions in each layer (len(a) = len(b)-1).
    U_0: u(x) value at left boundary x=0=b[0].
    U_L: u(x) value at right boundary x=L=b[0].
```

```
    """

    def __init__(self, a, b, U_0, U_L, eps=0):
        self.a, self.b = np.asarray(a), np.asarray(b)
        self.eps = eps  # smoothing parameter for smoothed a
        self.U_0, self.U_L = U_0, U_L

        a_data = [[bi, ai] for bi, ai in zip(self.b, self.a)]
        domain = [b[0], b[-1]]
        self.a_func = PiecewiseConstant(domain, a_data, eps)

        inv_a_data = [[bi, 1./ai] for bi, ai in zip(self.b, self.a)]
        self.inv_a_func = \
             PiecewiseConstant(domain, inv_a_data, eps)
        self.integral_of_inv_a_func = \
             IntegratedPiecewiseConstant(domain, inv_a_data, eps)
        self.inv_a_0L = self.integral_of_inv_a_func(b[-1])

    def __call__(self, x):
        solution = self.U_0 + (self.U_L-self.U_0)*\
                   self.integral_of_inv_a_func(x)/self.inv_a_0L
        return solution
```

A visualization method is also convenient to have. Below we plot $u(x)$ along with $\alpha(x)$ (which works well as long as $\max \alpha(x)$ is of the same size as $\max u = \max(U_0, U_L)$).

```
class SerialLayers:
    ...

    def plot(self):
        x, y_a = self.a_func.plot()
        x = np.asarray(x); y_a = np.asarray(y_a)
        y_u = self.u_exact(x)
        import matplotlib.pyplot as plt
        plt.figure()
        plt.plot(x, y_u, 'b')
        plt.hold('on')  # Matlab style
        plt.plot(x, y_a, 'r')
        ymin = -0.1
        ymax = 1.2*max(y_u.max(), y_a.max())
        plt.axis([x[0], x[-1], ymin, ymax])
        plt.legend(['solution $u$', 'coefficient $a$'], loc='upper left')
        if self.eps > 0:
            plt.title('Smoothing eps: %s' % self.eps)
        plt.savefig('tmp.pdf')
        plt.savefig('tmp.png')
        plt.show()
```

Figure Figure 3.12 shows the case where

```
b = [0, 0.25, 0.5, 1]   # material boundaries
a = [0.2, 0.4, 4]       # material values
U_0 = 0.5;  U_L = 5     # boundary conditions
```

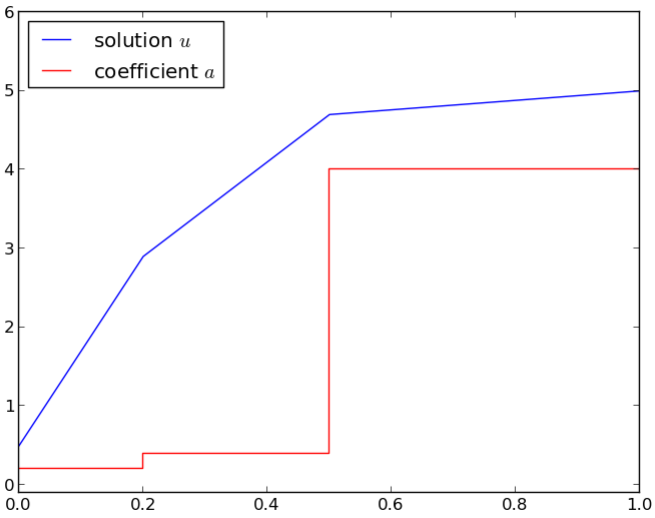


Figure 3.12.: Solution of the stationary diffusion equation corresponding to a piecewise constant diffusion coefficient.

By adding the `eps` parameter to the constructor of the `SerialLayers` class, we can experiment with smoothed versions of α and see the (small) impact on u. Figure Figure 3.13 shows the result.

3.35. Axi-symmetric diffusion

Suppose we have a diffusion process taking place in a straight tube with radius R. We assume axi-symmetry such that u is just a function of r and t, with r being the radial distance from the center axis of the tube to a point. With such axi-symmetry it is advantageous to introduce *cylindrical coordinates* r, θ, and z, where z is in the direction of the tube and (r, θ) are polar coordinates in a cross section. Axi-symmetry means that all quantities are independent of θ. From the relations $x = \cos\theta$, $y = \sin\theta$, and $z = z$, between Cartesian and cylindrical coordinates, one can (with some effort) derive the diffusion equation in cylindrical coordinates, which with axi-symmetry takes the form

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha(r,z)\frac{\partial u}{\partial r}\right) + \frac{\partial}{\partial z}\left(\alpha(r,z)\frac{\partial u}{\partial z}\right) + f(r,z,t)\,.$$

Let us assume that u does not change along the tube axis so it suffices to compute variations in a cross section. Then $\partial u/\partial z = 0$ and we have a 1D diffusion equation in the radial coordinate r and time t. In particular, we shall address the initial-boundary value problem

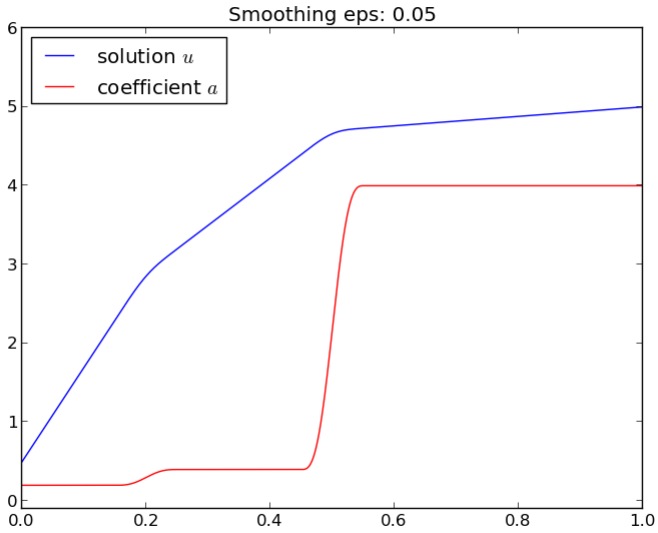


Figure 3.13.: Solution of the stationary diffusion equation corresponding to a *smoothed* piecewise constant diffusion coefficient.

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha(r)\frac{\partial u}{\partial r}\right) + f(t), \quad r \in (0,R),\ t \in (0,T], \tag{3.63}$$

$$\frac{\partial u}{\partial r}(0,t) = 0, \quad t \in (0,T], \tag{3.64}$$

$$u(R,t) = 0, \quad t \in (0,T], \tag{3.65}$$

$$u(r,0) = I(r), \quad r \in [0,R]. \tag{3.66}$$

The condition (3.64) is a necessary symmetry condition at $r = 0$, while (3.65) could be any Dirichlet or Neumann condition (or Robin condition in case of cooling or heating).

The finite difference approximation will need the discretized version of the PDE for $r = 0$ (just as we use the PDE at the boundary when implementing Neumann conditions). However, discretizing the PDE at $r = 0$ poses a problem because of the $1/r$ factor. We therefore need to work out the PDE for discretization at $r = 0$ with care. Let us, for the case of constant α, expand the spatial derivative term to

$$\alpha\frac{\partial^2 u}{\partial r^2} + \alpha\frac{1}{r}\frac{\partial u}{\partial r}.$$

The last term faces a difficulty at $r = 0$, since it becomes a $0/0$ expression caused by the symmetry condition at $r = 0$. However, L'Hospital's rule can be used:

$$\lim_{r\to 0}\frac{1}{r}\frac{\partial u}{\partial r} = \frac{\partial^2 u}{\partial r^2}.$$

The PDE at $r = 0$ therefore becomes

$$\frac{\partial u}{\partial t} = 2\alpha\frac{\partial^2 u}{\partial r^2} - f(t). \tag{3.67}$$

For a variable coefficient $\alpha(r)$ the expanded spatial derivative term reads

$$\alpha(r)\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}(\alpha(r) + r\alpha'(r))\frac{\partial u}{\partial r}\,.$$

We are interested in this expression for $r = 0$. A necessary condition for u to be axi-symmetric is that all input data, including α, must also be axi-symmetric, implying that $\alpha'(0) = 0$ (the second term vanishes anyway because of $r = 0$). The limit of interest is

$$\lim_{r\to 0}\frac{1}{r}\alpha(r)\frac{\partial u}{\partial r} = \alpha(0)\frac{\partial^2 u}{\partial r^2}\,.$$

The PDE at $r = 0$ now looks like

$$\frac{\partial u}{\partial t} = 2\alpha(0)\frac{\partial^2 u}{\partial r^2} - f(t), \tag{3.68}$$

so there is no essential difference between the constant coefficient and variable coefficient cases.

The second-order derivative in (3.67) and (3.68) is discretized in the usual way.

$$2\alpha\frac{\partial^2}{\partial r^2}u(r_0, t_n) \approx [2\alpha D_r D_r u]_0^n = 2\alpha\frac{u_1^n - 2u^n * *0 + u^n * *{-1}}{\Delta r^2}\,.$$

The fictitious value u_{-1}^n can be eliminated using the discrete symmetry condition

$$[D_{2r}u = 0]_0^n \quad \Rightarrow \quad u_{-1}^n = u_1^n,$$

which then gives the modified approximation to the term with the second-order derivative of u in r at $r = 0$:

$$4\alpha\frac{u_1^n - u_0^n}{\Delta r^2}\,.$$

The discretization of the term with the second-order derivative in r at any internal mesh point is straightforward:

$$\begin{aligned}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha\frac{\partial u}{\partial r}\right)\right]_i^n &\approx [r^{-1}D_r(r\alpha D_r u)]_i^n \\ &= \frac{1}{r_i}\frac{1}{\Delta r^2}\left(r_{i+\frac{1}{2}}\alpha_{i+\frac{1}{2}}(u_{i+1}^n - u_i^n) - r_{i-\frac{1}{2}}\alpha_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right)\,.\end{aligned}$$

To complete the discretization, we need a scheme in time, but that can be done as before and does not interfere with the discretization in space.

3.36. Spherically-symmetric diffusion

3.36.1. Discretization in spherical coordinates

Let us now pose the problem from Section Section 3.35 in spherical coordinates, where u only depends on the radial coordinate r and time t. That is, we have spherical symmetry. For simplicity we restrict the diffusion coefficient α to be a constant. The PDE reads

$$\frac{\partial u}{\partial t} = \frac{\alpha}{r^\gamma}\frac{\partial}{\partial r}\left(r^\gamma\frac{\partial u}{\partial r}\right) + f(t),$$

for $r \in (0, R)$ and $t \in (0, T]$. The parameter γ is 2 for spherically-symmetric problems and 1 for axi-symmetric problems. The boundary and initial conditions have the same mathematical form as in (3.63)-(3.66).

Since the PDE in spherical coordinates has the same form as the PDE in Section Section 3.35, just with the γ parameter being different, we can use the same discretization approach. At the origin $r = 0$ we get problems with the term

$$\frac{\gamma}{r}\frac{\partial u}{\partial t},$$

but L'Hospital's rule shows that this term equals $\gamma\partial^2 u/\partial r^2$, and the PDE at $r = 0$ becomes

$$\frac{\partial u}{\partial t} = (\gamma + 1)\alpha\frac{\partial^2 u}{\partial r^2} - f(t)\,.$$

The associated discrete form is then

$$[D_t u = \frac{1}{2}(\gamma + 1)\alpha D_r D_r \overline{u}^t + \overline{f}^t]_i^{n+\frac{1}{2}},$$

for a Crank-Nicolson scheme.

3.36.2. Discretization in Cartesian coordinates

The spherically-symmetric spatial derivative can be transformed to the Cartesian counterpart by introducing

$$v(r, t) = ru(r, t)\,.$$

Inserting $u = v/r$ in

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(\alpha(r)r^2\frac{\partial u}{\partial r}\right),$$

yields

$$r\left(\frac{d\alpha}{dr}\frac{\partial v}{\partial r} + \alpha\frac{\partial^2 v}{\partial r^2}\right) - \frac{d\alpha}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(\alpha\frac{\partial v}{\partial r}\right)\,.$$

The PDE for v takes the form

$$\frac{\partial v}{\partial t} = \frac{\partial}{\partial r}\left(\alpha\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{d\alpha}{dr}v + rf(r, t), \quad r \in (0, R),\ t \in (0, T]\,.$$

For α constant we immediately realize that we can reuse a solver in Cartesian coordinates to compute v. With variable α, a "reaction" term v/r needs to be added to the Cartesian solver. The boundary condition $\partial u/\partial r = 0$ at $r = 0$, implied by symmetry, forces $v(0, t) = 0$, because

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r = 0\,.$$

3.37. Diffusion in 2D

We now address diffusion in two space dimensions:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2}\right) + f(x,y), \tag{3.69}$$

in a domain

$$(x,y) \in (0,L_x)\times(0,L_y),\ t\in(0,T],$$

with $u=0$ on the boundary and $u(x,y,0)=I(x,y)$ as initial condition.

3.38. Discretization

For generality, it is natural to use a θ-rule for the time discretization. Standard, second-order accurate finite differences are used for the spatial derivatives. We sample the PDE at a space-time point $(i,j,n+\frac{1}{2})$ and apply the difference approximations:

$$\begin{aligned}[D_t u]^{n+\frac{1}{2}} = \theta[\alpha(D_xD_xu + D_yD_yu) + f]^{n+1} + \\ (1-\theta)[\alpha(D_xD_xu + D_yD_yu) + f]^{n}\,.\end{aligned} \tag{3.70}$$

Written out,

$$\begin{aligned}&\frac{u^{n+1}**i,j - u^n**i,j}{\Delta t} = \\ &\quad\theta(\alpha(\frac{u^{n+1}**i-1,j - 2u^{n+1}**i,j + u^{n+1}_{i+1,j}}{\Delta x^2} + \frac{u^{n+1}**i,j-1 - 2u^{n+1}**i,j + u^{n+1}_{i,j+1}}{\Delta y^2}) + f^{n+1}_{i,j}) + \\ &\quad(1-\theta)(\alpha(\frac{u^{n}**i-1,j - 2u^{n}**i,j + u^{n}_{i+1,j}}{\Delta x^2} + \frac{u^{n}**i,j-1 - 2u^{n}**i,j + u^{n}_{i,j+1}}{\Delta y^2}) + f^{n}_{i,j})\end{aligned} \tag{3.71}$$

We collect the unknowns on the left-hand side

$$\begin{aligned}&u^{n+1}_{i,j} - \theta\left(F_x(u^{n+1}**i-1,j - 2u^{n+1}**i,j + u^{n+1}_{i+1,j}) + F_y(u^{n+1}**i,j-1 - 2u^{n+1}**i,j + u^{n+1}_{i,j+1})\right) = \\ &\quad(1-\theta)\left(F_x(u^{n}**i-1,j - 2u^{n}**i,j + u^{n}_{i+1,j}) + F_y(u^{n}**i,j-1 - 2u^{n}**i,j + u^{n}_{i,j+1})\right) + \\ &\quad\theta\Delta t f^{n+1}**i,j + (1-\theta)\Delta t f^{n}**i,j + u^n_{i,j},\end{aligned} \tag{3.72}$$

where

$$F_x = \frac{\alpha\Delta t}{\Delta x^2},\quad F_y = \frac{\alpha\Delta t}{\Delta y^2},$$

are the Fourier numbers in x and y direction, respectively.

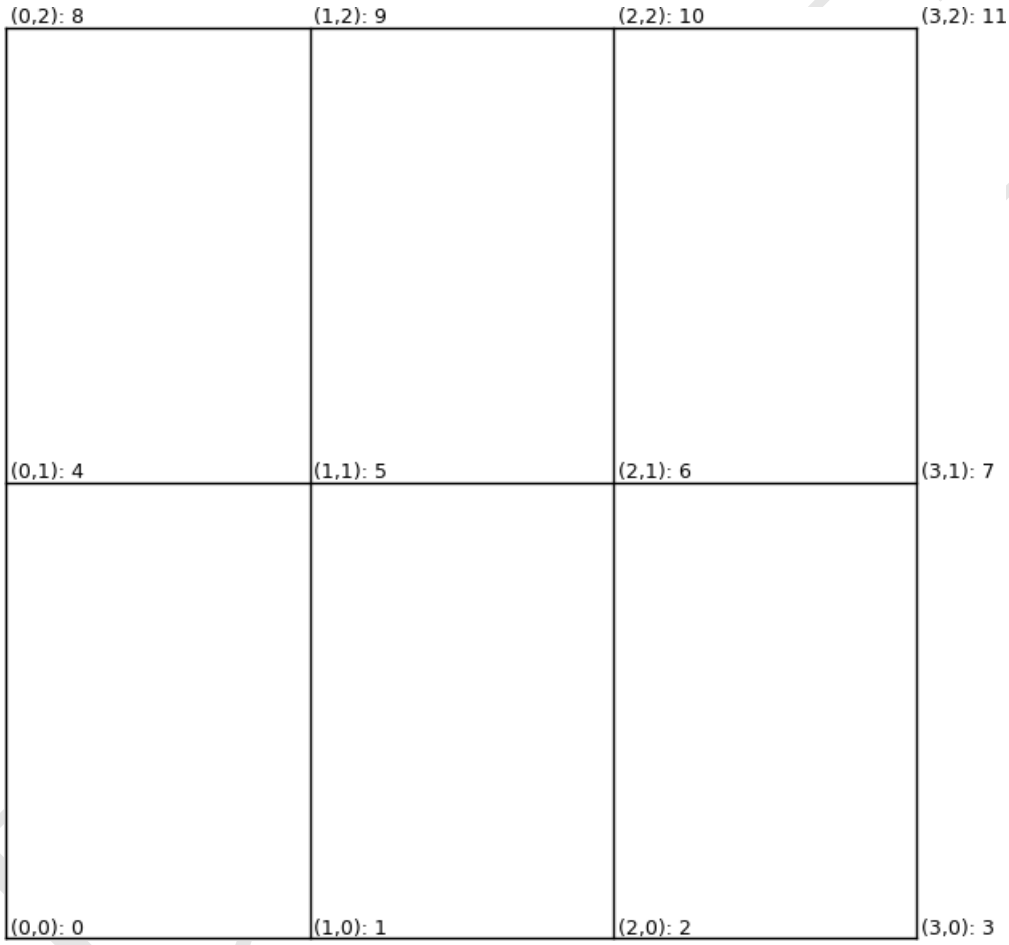


Figure 3.14.: 3x2 2D mesh.

3.39. Numbering of mesh points versus equations and unknowns

The equations (3.72) are coupled at the new time level $n+1$. That is, we must solve a system of (linear) algebraic equations, which we will write as $Ac = b$, where A is the coefficient matrix, c is the vector of unknowns, and b is the right-hand side.

Let us examine the equations in $Ac = b$ on a mesh with $N_x = 3$ and $N_y = 2$ cells in the respective spatial directions. The spatial mesh is depicted in Figure Figure 3.14. The equations at the boundary just implement the boundary condition $u = 0$:

$$u^{n+1}_{0,0} = u^{n+1}_{1,0} = u^{n+1}_{2,0} = u^{n+1}_{3,0} = u^{n+1}_{0,1} = u^{n+1}_{3,1} = u^{n+1}_{0,2} = u^{n+1}_{1,2} = u^{n+1}_{2,2} = u^{n+1}_{3,2} = 0\,.$$

We are left with two interior points, with $i = 1$, $j = 1$ and $i = 2$, $j = 1$. The corresponding equations are

$$\begin{aligned} u^{n+1}_{i,j} - \theta\left(F_x(u^{n+1} * *i-1, j-2u^{n+1} * *i, j+u^{n+1}_{i+1,j}) + F_y(u^{n+1} * *i, j-1-2u^{n+1} * *i, j+u^{n+1}_{i,j+1})\right) = \\ (1-\theta)\left(F_x(u^{n} * *i-1, j-2u^{n} * *i, j+u^{n}_{i+1,j}) + F_y(u^{n} * *i, j-1-2u^{n} * *i, j+u^{n}_{i,j+1})\right) + \\ \theta\Delta t f^{n+1} * *i, j + (1-\theta)\Delta t f^{n} * *i, j + u^n_{i,j}, \end{aligned}$$

There are in total 12 unknowns $u^{n+1}_{i,j}$ for $i = 0, 1, 2, 3$ and $j = 0, 1, 2$. To solve the equations, we need to form a matrix system $Ac = b$. In that system, the solution vector c can only have one index. Thus, we need a numbering of the unknowns with one index, not two as used in the mesh. We introduce a mapping $m(i, j)$ from a mesh point with indices (i, j) to the corresponding unknown p in the equation system:

$$p = m(i, j) = j(N_x + 1) + i\,.$$

When i and j run through their values, we see the following mapping to p:

$$\begin{aligned} &(0,0) \to 0, \quad (0,1) \to 1, \quad (0,2) \to 2, \quad (0,3) \to 3, \\ &(1,0) \to 4, \quad (1,1) \to 5, \quad (1,2) \to 6, \quad (1,3) \to 7, \\ &(2,0) \to 8, \quad (2,1) \to 9, \quad (2,2) \to 10, \quad (2,3) \to 11\,. \end{aligned}$$

That is, we number the points along the x axis, starting with $y = 0$, and then progress one "horizontal" mesh line at a time. In Figure Figure 3.14 you can see that the (i, j) and the corresponding single index (p) are listed for each mesh point.

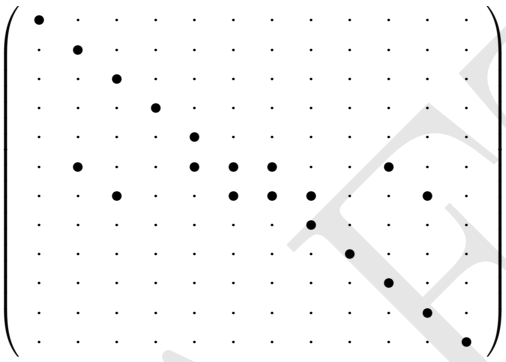
We could equally well have numbered the equations in other ways, e.g., let the j index be the fastest varying index: $p = m(i, j) = i(N_y + 1) + j$.

Let us form the coefficient matrix A, or more precisely, insert a matrix element (according Python's convention with zero as base index) for each of the nonzero elements in A (the indices run through

the values of p, i.e., $p = 0, \ldots, 11$):

$$\begin{pmatrix}
(0,0) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & (1,1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & (2,2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & (3,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & (4,4) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & (5,1) & 0 & 0 & (5,4) & (5,5) & (5,6) & 0 & 0 & (5,9) & 0 & 0 \\
0 & 0 & (6,2) & 0 & 0 & (6,5) & (6,6) & (6,7) & 0 & 0 & (6,10) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & (7,7) & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (8,8) & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (9,9) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (10,10) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (11,11)
\end{pmatrix}$$

Here is a more compact visualization of the coefficient matrix where we insert dots for zeros and bullets for non-zero elements:



It is clearly seen that most of the elements are zero. This is a general feature of coefficient matrices arising from discretizing PDEs by finite difference methods. We say that the matrix is *sparse*.

Let $A_{p,q}$ be the value of element (p, q) in the coefficient matrix A, where p and q now correspond to the numbering of the unknowns in the equation system. We have $A_{p,q} = 1$ for $p = q = 0, 1, 2, 3, 4, 7, 8, 9, 10, 11$, corresponding to all the known boundary values. Let p be $m(i, j)$, i.e., the single index corresponding to mesh point (i, j). Then we have

$$A_{m(i,j),m(i,j)} = A_{p,p} = 1 + \theta(F_x + F_y), \tag{3.73}$$

$$A_{p,m(i-1,j)} = A_{p,p-1} = -\theta F_x, \tag{3.74}$$

$$A_{p,m(i+1,j)} = A_{p,p+1} = -\theta F_x, \tag{3.75}$$

$$A_{p,m(i,j-1)} = A_{p,p-(N_x+1)} = -\theta F_y, \tag{3.76}$$

$$A_{p,m(i,j+1)} = A_{p,p+(N_x+1)} = -\theta F_y, \tag{3.77}$$

for the equations associated with the two interior mesh points. At these interior points, the single index p takes on the specific values $p = 5, 6$, corresponding to the values $(1, 1)$ and $(1, 2)$ of the pair (i, j).

The above values for $A_{p,q}$ can be inserted in the matrix:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\theta F_y & 0 & 0 & -\theta F_x & 1+2\theta F_x & -\theta F_x & 0 & 0 & -\theta F_y & 0 & 0 \\
0 & 0 & -\theta F_y & 0 & 0 & -\theta F_x & 1+2\theta F_x & -\theta F_x & 0 & 0 & -\theta F_y & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

The corresponding right-hand side vector in the equation system has the entries b_p, where p numbers the equations. We have

$$b_0 = b_1 = b_2 = b_3 = b_4 = b_7 = b_8 = b_9 = b_{10} = b_{11} = 0,$$

for the boundary values. For the equations associated with the interior points, we get for $p = 5, 6$, corresponding to $i = 1, 2$ and $j = 1$:

$$b_p = u^n_{i,j} + (1-\theta)\left(F_x(u^n * *i-1, j-2u^n * *i, j+u^n_{i+1,j}) + F_y(u^n * *i, j-1-2u^n * *i, j+u^n_{i,j+1})\right) + \theta\Delta t f^{n+1} * *i, j + (1-\theta)\Delta t f^n * *i, j\,.$$

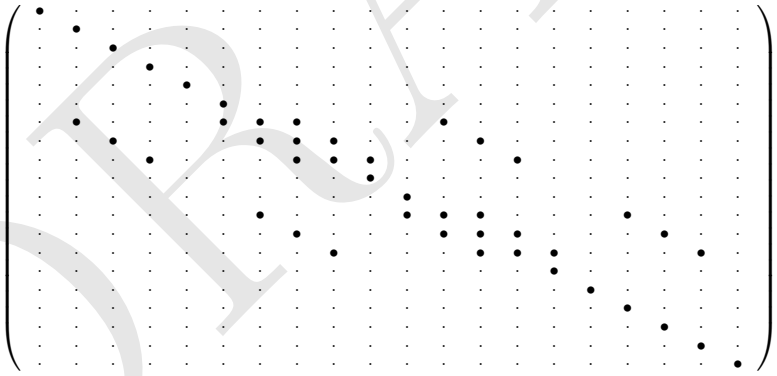
Recall that $p = m(i, j) = j(N_x + 1) + j$ in this expression.

We can, as an alternative, leave the boundary mesh points out of the matrix system. For a mesh with $N_x = 3$ and $N_y = 2$ there are only two internal mesh points whose unknowns will enter the matrix system. We must now number the unknowns at the interior points:

$$p = (j-1)(N_x - 1) + i,$$

for $i = 1, \ldots, N_x - 1$, $j = 1, \ldots, N_y - 1$.

We can continue with illustrating a bit larger mesh, $N_x = 4$ and $N_y = 3$, see Figure Figure 3.15. The corresponding coefficient matrix with dots for zeros and bullets for non-zeroes looks as follows (values at boundary points are included in the equation system):



The coefficient matrix is banded

Besides being sparse, we observe that the coefficient matrix is *banded*: it has five distinct bands. We have the diagonal $A_{i,i}$, the subdiagonal $A_{i-1,j}$, the superdiagonal $A_{i,i+1}$, a lower diagonal $A_{i,i-(Nx+1)}$, and an upper diagonal $A_{i,i+(Nx+1)}$. The other matrix entries are known to be zero. With $N_x + 1 = N_y + 1 = N$, only a fraction $5N^{-2}$ of the matrix entries are nonzero, so the matrix is clearly very sparse for relevant N values. The more we can compute with the nonzeros only, the faster the solution methods will potentially be.

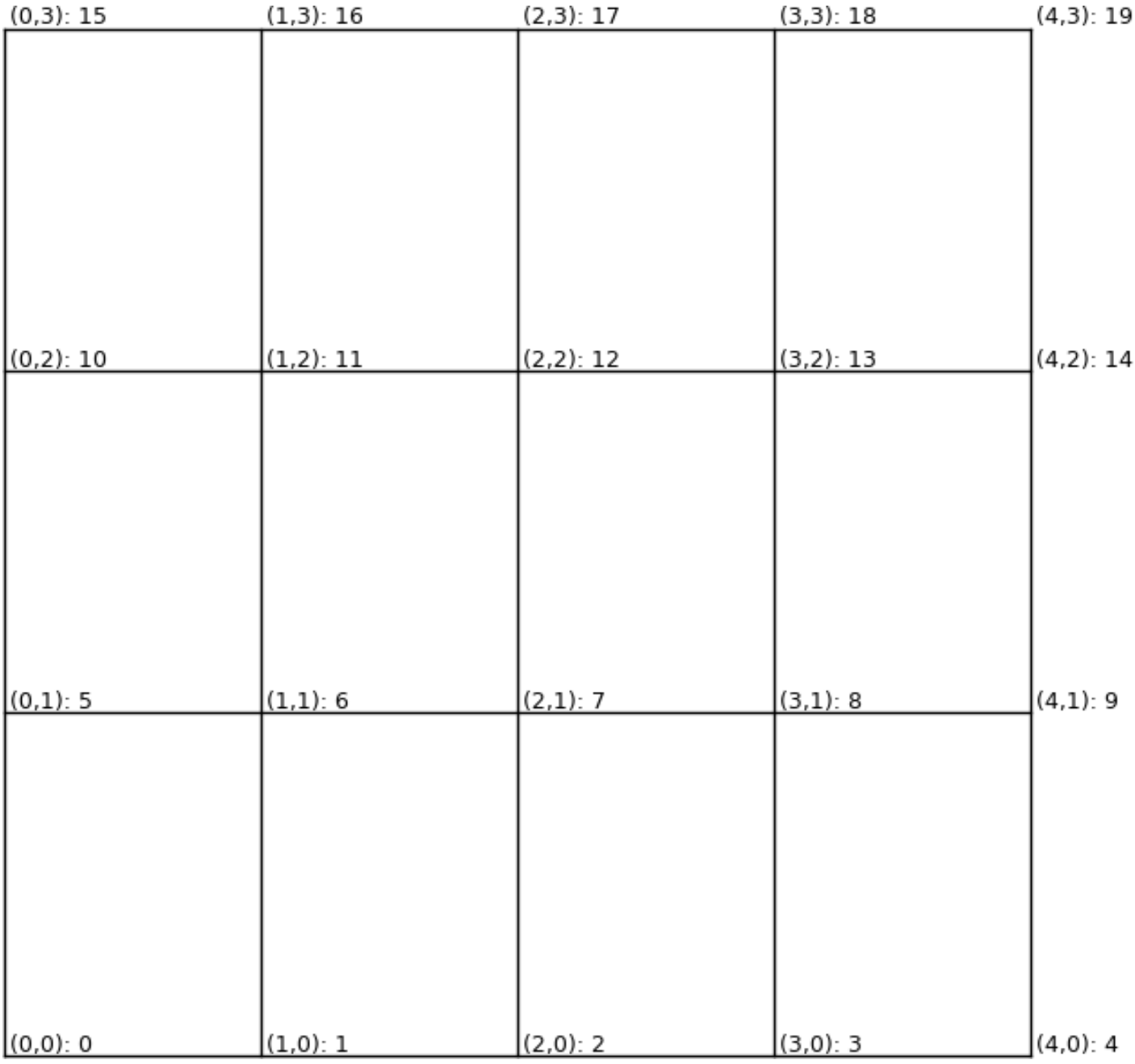


Figure 3.15.: 4x3 2D mesh.

3.40. Algorithm for setting up the coefficient matrix

We looked at a specific mesh in the previous section, formulated the equations, and saw what the corresponding coefficient matrix and right-hand side are. Now our aim is to set up a general algorithm, for any choice of N_x and N_y, that produces the coefficient matrix and the right-hand side vector. We start with a zero matrix and vector, run through each mesh point, and fill in the values depending on whether the mesh point is an interior point or on the boundary.

- for $i = 0, \ldots, N_x$
- for $j = 0, \ldots, N_y$
  - $p = j(N_x + 1) + i$
  - if point (i, j) is on the boundary:
    - $A_{p,p} = 1$, $b_p = 0$
  - else:
    - fill $A_{p,m(i-1,j)}$, $A_{p,m(i+1,j)}$, $A_{p,m(i,j)}$, $A_{p,m(i,j-1)}$, $A_{p,m(i,j+1)}$, and b_p

To ease the test on whether (i, j) is on the boundary or not, we can split the loops a bit, starting with the boundary line $j = 0$, then treat the interior lines $1 \leq j < N_y$, and finally treat the boundary line $j = N_y$:

- for $i = 0, \ldots, N_x$
- boundary $j = 0$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
- for $j = 0, \ldots, N_y$
- boundary $i = 0$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
- for $i = 1, \ldots, N_x - 1$
  - interior point $p = j(N_x + 1) + i$
  - fill $A_{p,m(i-1,j)}$, $A_{p,m(i+1,j)}$, $A_{p,m(i,j)}$, $A_{p,m(i,j-1)}$, $A_{p,m(i,j+1)}$, and b_p
- boundary $i = N_x$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
- for $i = 0, \ldots, N_x$
- boundary $j = N_y$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$

The right-hand side is set up as follows.

- for $i = 0, \ldots, N_x$
- boundary $j = 0$: $p = j(N_x + 1) + i$, $b_p = 0$
- for $j = 0, \ldots, N_y$
- boundary $i = 0$: $p = j(N_x + 1) + i$, $b_p = 0$
- for $i = 1, \ldots, N_x - 1$
  - interior point $p = j(N_x + 1) + i$
  - fill b_p
- boundary $i = N_x$: $p = j(N_x + 1) + i$, $b_p = 0$
- for $i = 0, \ldots, N_x$
- boundary $j = N_y$: $p = j(N_x + 1) + i$, $b_p = 0$

3.41. Implementation with a dense coefficient matrix

The goal now is to map the algorithms in the previous section to Python code. One should, for computational efficiency reasons, take advantage of the fact that the coefficient matrix is sparse and/or banded, i.e., take advantage of all the zeros. However, we first demonstrate how to fill an $N \times N$ dense square matrix, where N is the number of unknowns, here $N = (N_x + 1)(N_y + 1)$. The dense matrix is much easier to understand than the sparse matrix case.

```
import numpy as np

def solver_dense(
    I, a, f, Lx, Ly, Nx, Ny, dt, T, theta=0.5, user_action=None):
    """
    Solve u_t = a*(u_xx + u_yy) + f, u(x,y,0)=I(x,y), with u=0
    on the boundary, on [0,Lx]x[0,Ly]x[0,T], with time step dt,
    using the theta-scheme.
    """
    x = np.linspace(0, Lx, Nx+1)       # mesh points in x dir
    y = np.linspace(0, Ly, Ny+1)       # mesh points in y dir
    dx = x[1] - x[0]
    dy = y[1] - y[0]

    dt = float(dt)                     # avoid integer division
    Nt = int(round(T/float(dt)))
    t = np.linspace(0, Nt*dt, Nt+1)    # mesh points in time

    Fx = a*dt/dx**2
    Fy = a*dt/dy**2
```

The $u^{n+1} **i, j$ and $u^n **i, j$ mesh functions are represented by their spatial values at the mesh points:

```
u   = np.zeros((Nx+1, Ny+1))      # unknown u at new time level
u_n = np.zeros((Nx+1, Ny+1))      # u at the previous time level
```

It is a good habit (for extensions) to introduce index sets for all mesh points:

```
Ix = range(0, Nx+1)
It = range(0, Ny+1)
It = range(0, Nt+1)
```

The initial condition is easy to fill in:

```
for i in Ix:
    for j in It:
        u_n[i,j] = I(x[i], y[j])
```

The memory for the coefficient matrix and right-hand side vector is allocated by

```
N = (Nx+1)*(Ny+1)  # no of unknowns
A = np.zeros((N, N))
b = np.zeros(N)
```

The filling of `A` goes like this:

```
m = lambda i, j: j*(Nx+1) + i

j = 0
for i in Ix:
    p = m(i,j);  A[p, p] = 1

for j in It[1:-1]:
    i = 0;  p = m(i,j);  A[p, p] = 1   # Boundary
    for i in Ix[1:-1]:                 # Interior points
        p = m(i,j)
        A[p, m(i,j-1)] = - theta*Fy
        A[p, m(i-1,j)] = - theta*Fx
        A[p, p]        = 1 + 2*theta*(Fx+Fy)
        A[p, m(i+1,j)] = - theta*Fx
        A[p, m(i,j+1)] = - theta*Fy
    i = Nx;  p = m(i,j);  A[p, p] = 1  # Boundary
j = Ny
for i in Ix:
    p = m(i,j);  A[p, p] = 1
```

Since `A` is independent of time, it can be filled once and for all before the time loop. The right-hand side vector must be filled at each time level inside the time loop:

```
import scipy.linalg

for n in It[0:-1]:
    j = 0
    for i in Ix:
        p = m(i,j);  b[p] = 0          # Boundary
    for j in It[1:-1]:
        i = 0;  p = m(i,j);  b[p] = 0  # Boundary
        for i in Ix[1:-1]:             # Interior points
            p = m(i,j)
            b[p] = u_n[i,j] + \
              (1-theta)*(
              Fx*(u_n[i+1,j] - 2*u_n[i,j] + u_n[i-1,j]) +\
              Fy*(u_n[i,j+1] - 2*u_n[i,j] + u_n[i,j-1]))\
                + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
              (1-theta)*dt*f(i*dx,j*dy,n*dt)
```

```
        i = Nx;  p = m(i,j);  b[p] = 0  # Boundary
    j = Ny
    for i in Ix:
        p = m(i,j);  b[p] = 0           # Boundary

    c = scipy.linalg.solve(A, b)

    for i in Ix:
        for j in It:
            u[i,j] = c[m(i,j)]

    u_n, u = u, u_n
```

We use `solve` from `scipy.linalg` and not from `numpy.linalg`. The difference is stated below.

`scipy.linalg` versus `numpy.linalg`

Quote from the SciPy documentation:
`scipy.linalg` contains all the functions in `numpy.linalg` plus some other more advanced ones not contained in `numpy.linalg`.
Another advantage of using `scipy.linalg` over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.
Therefore, unless you don't want to add SciPy as a dependency to your NumPy program, use `scipy.linalg` instead of `numpy.linalg`.

The code shown above is available in the `solver_dense` function in the file `diffu2D_u0.py`, differing only in the boundary conditions, which in the code can be an arbitrary function along each side of the domain.

We do not bother to look at vectorized versions of filling `A` since a dense matrix is just used of pedagogical reasons for the very first implementation. Vectorization will be treated when `A` has a sparse matrix representation, as in Section Section 3.44.

How to debug the computation of A and b

A good starting point for debugging the filling of A and b is to choose a very coarse mesh, say $N_x = N_y = 2$, where there is just one internal mesh point, compute the equations by hand, and print out `A` and `b` for comparison in the code. If wrong elements in `A` or `b` occur, print out each assignment to elements in `A` and `b` inside the loops and compare with what you expect.

To let the user store, analyze, or visualize the solution at each time level, we include a callback function, named `user_action`, to be called before the time loop and in each pass in that loop. The function has the signature

```
user_action(u, x, xv, y, yv, t, n)
```

where `u` is a two-dimensional array holding the solution at time level `n` and time `t[n]`. The x and y coordinates of the mesh points are given by the arrays `x` and `y`, respectively. The arrays `xv` and `yv` are vectorized representations of the mesh points such that vectorized function evaluations can be invoked. The `xv` and `yv` arrays are defined by

```
xv = x[:,np.newaxis]
yv = y[np.newaxis,:]
```

One can then evaluate, e.g., $f(x, y, t)$ at all internal mesh points at time level `n` by first evaluating f at all points,

```
f_a = f(xv, yv, t[n])
```

and then use slices to extract a view of the values at the internal mesh points: `f_a[1:-1,1:-1]`. The next section features an example on writing a `user_action` callback function.

3.42. Verification: exact numerical solution

A good test example to start with is one that preserves the solution $u = 0$, i.e., $f = 0$ and $I(x, y) = 0$. This trivial solution can uncover some bugs.

The first real test example is based on having an exact solution of the discrete equations. This solution is linear in time and quadratic in space:

$$u(x, y, t) = 5tx(L_x - x)y(y - L_y)\,.$$

Inserting this manufactured solution in the PDE shows that the source term f must be

$$f(x, y, t) = 5x(L_x - x)y(y - L_y) + 10\alpha t(x(L_x - x) + y(y - L_y))\,.$$

We can use the `user_action` function to compare the numerical solution with the exact solution at each time level. A suitable helper function for checking the solution goes like this:

```
def quadratic(theta, Nx, Ny):

    def u_exact(x, y, t):
        return 5*t*x*(Lx-x)*y*(Ly-y)
    def I(x, y):
        return u_exact(x, y, 0)
    def f(x, y, t):
        return 5*x*(Lx-x)*y*(Ly-y) + 10*a*t*(y*(Ly-y)+x*(Lx-x))

    Lx = 0.75
    Ly = 1.5
    a = 3.5
    dt = 0.5
    T = 2
```

```
    def assert_no_error(u, x, xv, y, yv, t, n):
        """Assert zero error at all mesh points."""
        u_e = u_exact(xv, yv, t[n])
        diff = abs(u - u_e).max()
        tol = 1E-12
        msg = 'diff=%g, step %d, time=%g' % (diff, n, t[n])
        print msg
        assert diff < tol, msg

    solver_dense(
        I, a, f, Lx, Ly, Nx, Ny,
        dt, T, theta, user_action=assert_no_error)
```

A true test function for checking the quadratic solution for several different meshes and θ values can take the form

```
def test_quadratic():
    for theta in [1, 0.5, 0]:
        for Nx in range(2, 6, 2):
            for Ny in range(2, 6, 2):
                print 'testing for %dx%d mesh' % (Nx, Ny)
                quadratic(theta, Nx, Ny)
```

3.43. Verification: convergence rates

For 2D verification with convergence rate computations, the expressions and computations just build naturally on what we saw for 1D diffusion. Truncation error analysis and other forms of error analysis point to a numerical error formula like

$$E = C_t\Delta t^p + C_x\Delta x^2 + C_y\Delta y^2,$$

where p, C_t, C_x, and C_y are constants. Often, the analysis of a Crank-Nicolson method can show that $p = 2$, while the Forward and Backward Euler schemes have $p = 1$.

When checking the error formula empirically, we need to reduce it to a form $E = Ch^r$ with a single discretization parameter h and some rate r to be estimated. For the Backward Euler method, where $p = 1$, we can introduce a single discretization parameter according to

$$h = \Delta x^2 = \Delta y^2, \quad h = K^{-1}\Delta t,$$

where K is a constant. The error formula then becomes

$$E = C_tKh + C_xh + C_yh = \tilde{C}h, \quad \tilde{C} = C_tK + C_x + C_y\,.$$

The simplest choice is obviously $K = 1$. With the Forward Euler method, however, stability requires $\Delta t = hK \leq h/(4\alpha)$, so $K \leq 1/(4\alpha)$.

For the Crank-Nicolson method, $p = 2$, and we can simply choose

$$h = \Delta x = \Delta y = \Delta t,$$

since there is no restriction on Δt in terms of Δx and Δy.

A frequently used error measure is the ℓ^2 norm of the error mesh point values. Section Section 2.10.1 and the formula (2.21) shows the error measure for a 1D time-dependent problem. The extension to the current 2D problem reads

$$E = \left(\Delta t \Delta x \Delta y \sum_{n=0}^{N_t} \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} (u_{\mathrm{e}}(x_i, y_j, t_n) - u^n_{i,j})^2 \right)^{\frac{1}{2}}.$$

One attractive manufactured solution is

$$u_{\mathrm{e}} = e^{-pt} \sin(k_x x) \sin(k_y y), \quad k_x = \frac{\pi}{L_x}, k_y = \frac{\pi}{L_y},$$

where p can be arbitrary. The required source term is

$$f = (\alpha(k_x^2 + k_y^2) - p)u_{\mathrm{e}}\,.$$

The function `convergence_rates` in `diffu2D_u0.py` implements a convergence rate test. Two potential difficulties are important to be aware of:

1. The error formula is assumed to be correct when $h \to 0$, so for coarse meshes the estimated rate r may be somewhat away from the expected value. Fine meshes may lead to prohibitively long execution times.
2. Choosing $p = \alpha(k_x^2 + k_y^2)$ in the manufactured solution above seems attractive ($f = 0$), but leads to a slower approach to the asymptotic range where the error formula is valid (i.e., r fluctuates and needs finer meshes to stabilize).

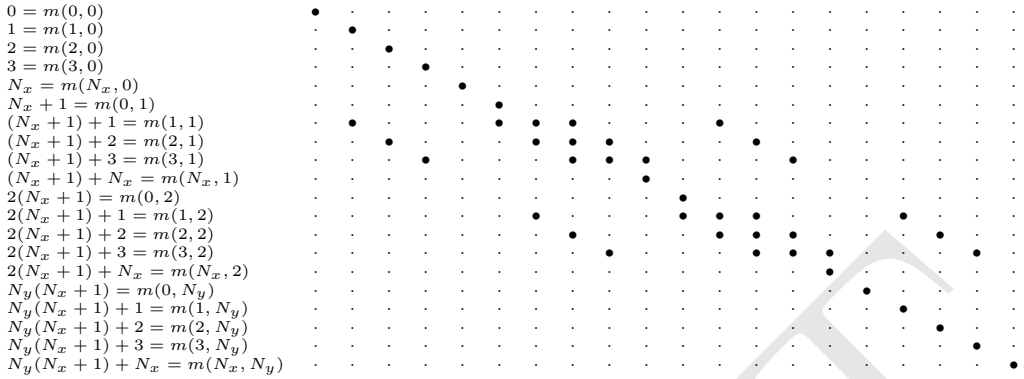
3.44. Implementation with a sparse coefficient matrix

We used a sparse matrix implementation in Section Section 3.9 for a 1D problem with a tridiagonal matrix. The present matrix, arising from a 2D problem, has five diagonals, but we can use the same sparse matrix data structure `scipy.sparse.diags`.

3.44.1. Understanding the diagonals

Let us look closer at the diagonals in the example with a 4×3 mesh as depicted in Figure Figure 3.15 and its associated matrix visualized by dots for zeros and bullets for nonzeros. From the example

mesh, we may generalize to an $N_x \times N_y$ mesh.



The main diagonal has $N = (N_x+1)(N_y+1)$ elements, while the sub- and super-diagonals have $N-1$ elements. By looking at the matrix above, we realize that the lower diagonal starts in row N_x+1 and goes to row N, so its length is $N-(N_x+1)$. Similarly, the upper diagonal starts at row 0 and lasts to row $N-(N_x+1)$, so it has the same length. Based on this information, we declare the diagonals by

```
main   = np.zeros(N)             # diagonal
lower  = np.zeros(N-1)           # subdiagonal
upper  = np.zeros(N-1)           # superdiagonal
lower2 = np.zeros(N-(Nx+1))      # lower diagonal
upper2 = np.zeros(N-(Nx+1))      # upper diagonal
b      = np.zeros(N)             # right-hand side
```

3.44.2. Filling the diagonals

We run through all mesh points and fill in elements on the various diagonals. The line of mesh points corresponding to $j = 0$ are all on the boundary, and only the main diagonal gets a contribution:

```
m = lambda i, j: j*(Nx+1) + i
j = 0; main[m(0,j):m(Nx+1,j)] = 1  # j=0 boundary line
```

Then we run through all interior $j = \text{const}$ lines of mesh points. The first and the last point on each line, $i = 0$ and $i = N_x$, correspond to boundary points:

```
for j in It[1:-1]:               # Interior mesh lines j=1,...,Ny-1
    i = 0;   main[m(i,j)] = 1
    i = Nx;  main[m(i,j)] = 1  # Boundary
```

For the interior mesh points $i = 1, \ldots, N_x - 1$ on a mesh line $y = \text{const}$ we can start with the main diagonal. The entries to be filled go from $i = 1$ to $i = N_x - 1$ so the relevant slice in the `main` vector is `m(1,j):m(Nx,j)`:

```
main[m(1,j):m(Nx,j)] = 1 + 2*theta*(Fx+Fy)
```

The `upper` array for the superdiagonal has its index 0 corresponding to row 0 in the matrix, and the array entries to be set go from $m(1,j)$ to $m(N_x-1,j)$:

```
upper[m(1,j):m(Nx,j)] = - theta*Fx
```

The subdiagonal (`lower` array), however, has its index 0 corresponding to row 1, so there is an offset of 1 in indices compared to the matrix. The first nonzero occurs (interior point) at a mesh line $j=$ const corresponding to matrix row $m(1,j)$, and the corresponding array index in `lower` is then $m(1,j)$. To fill the entries from $m(1,j)$ to $m(N_x-1,j)$ we set the following slice in `lower`:

```
lower_offset = 1
lower[m(1,j)-lower_offset:m(Nx,j)-lower_offset] = - theta*Fx
```

For the upper diagonal, its index 0 corresponds to matrix row 0, so there is no offset and we can set the entries correspondingly to `upper`:

```
upper2[m(1,j):m(Nx,j)] = - theta*Fy
```

The `lower2` diagonal, however, has its first index 0 corresponding to row N_x+1, so here we need to subtract the offset N_x+1:

```
lower2_offset = Nx+1
lower2[m(1,j)-lower2_offset:m(Nx,j)-lower2_offset] = - theta*Fy
```

We can now summarize the above code lines for setting the entries in the sparse matrix representation of the coefficient matrix:

```
lower_offset = 1
lower2_offset = Nx+1
m = lambda i, j: j*(Nx+1) + i

j = 0; main[m(0,j):m(Nx+1,j)] = 1  # j=0 boundary line
for j in It[1:-1]:              # Interior mesh lines j=1,...,Ny-1
    i = 0;   main[m(i,j)] = 1  # Boundary
    i = Nx;  main[m(i,j)] = 1  # Boundary
    lower2[m(1,j)-lower2_offset:m(Nx,j)-lower2_offset] = - theta*Fy
    lower[m(1,j)-lower_offset:m(Nx,j)-lower_offset] = - theta*Fx
    main[m(1,j):m(Nx,j)] = 1 + 2*theta*(Fx+Fy)
    upper[m(1,j):m(Nx,j)] = - theta*Fx
    upper2[m(1,j):m(Nx,j)] = - theta*Fy
j = Ny; main[m(0,j):m(Nx+1,j)] = 1  # Boundary line
```

The next task is to create the sparse matrix from these diagonals:

```
import scipy.sparse

A = scipy.sparse.diags(
    diagonals=[main, lower, upper, lower2, upper2],
    offsets=[0, -lower_offset, lower_offset,
             -lower2_offset, lower2_offset],
    shape=(N, N), format='csr')
```

3.44.3. Filling the right-hand side; scalar version

Setting the entries in the right-hand side is easier, since there are no offsets in the array to take into account. The right-hand side is in fact similar to the one previously shown, when we used a dense matrix representation (the right-hand side vector is, of course, independent of what type of representation we use for the coefficient matrix). The complete time loop goes as follows.

```
import scipy.sparse.linalg

for n in It[0:-1]:
    j = 0
    for i in Ix:
        p = m(i,j);  b[p] = 0                       # Boundary
    for j in It[1:-1]:
        i = 0;  p = m(i,j);  b[p] = 0               # Boundary
        for i in Ix[1:-1]:
            p = m(i,j)                              # Interior
            b[p] = u_n[i,j] + \
              (1-theta)*(
              Fx*(u_n[i+1,j] - 2*u_n[i,j] + u_n[i-1,j]) +\
              Fy*(u_n[i,j+1] - 2*u_n[i,j] + u_n[i,j-1]))\
                + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
              (1-theta)*dt*f(i*dx,j*dy,n*dt)
        i = Nx;  p = m(i,j);  b[p] = 0              # Boundary
    j = Ny
    for i in Ix:
        p = m(i,j);  b[p] = 0                       # Boundary

    c = scipy.sparse.linalg.spsolve(A, b)

    for i in Ix:
        for j in It:
            u[i,j] = c[m(i,j)]

    u_n, u = u, u_n
```

3.44.4. Filling the right-hand side; vectorized version

Since we use a sparse matrix and try to speed up the computations, we should examine the loops and see if some can be easily removed by vectorization. In the filling of A we have already used vectorized expressions at each $j =$ const line of mesh points. We can very easily do the same in the code above and remove the need for loops over the `i` index:

```
for n in It[0:-1]:

    f_a_np1 = f(xv, yv, t[n+1])
    f_a_n   = f(xv, yv, t[n])

    j = 0; b[m(0,j):m(Nx+1,j)] = 0     # Boundary
    for j in It[1:-1]:
        i = 0;   p = m(i,j);  b[p] = 0 # Boundary
        i = Nx;  p = m(i,j);  b[p] = 0 # Boundary
        imin = Ix[1]
        imax = Ix[-1]  # for slice, max i index is Ix[-1]-1
        b[m(imin,j):m(imax,j)] = u_n[imin:imax,j] + \
              (1-theta)*(Fx*(
          u_n[imin+1:imax+1,j] -
        2*u_n[imin:imax,j] + \
          u_n[imin-1:imax-1,j]) +
                         Fy*(
          u_n[imin:imax,j+1] -
        2*u_n[imin:imax,j] +
          u_n[imin:imax,j-1])) + \
            theta*dt*f_a_np1[imin:imax,j] + \
          (1-theta)*dt*f_a_n[imin:imax,j]
    j = Ny;  b[m(0,j):m(Nx+1,j)] = 0 # Boundary

    c = scipy.sparse.linalg.spsolve(A, b)

    u[:,:] = c.reshape(Ny+1,Nx+1).T

    u_n, u = u, u_n
```

The most tricky part of this code snippet is the loading of values from the one-dimensional array `c` into the two-dimensional array `u`. With our numbering of unknowns from left to right along "horizontal" mesh lines, the correct reordering of the one-dimensional array `c` as a two-dimensional array requires first a reshaping to an `(Ny+1,Nx+1)` two-dimensional array and then taking the transpose. The result is an `(Nx+1,Ny+1)` array compatible with `u` both in size and appearance of the function values.

The `spsolve` function in `scipy.sparse.linalg` is an efficient version of Gaussian elimination suited for matrices described by diagonals. The algorithm is known as *sparse Gaussian elimination*, and `spsolve` calls up a well-tested C code called SuperLU.

The complete code utilizing `spsolve` is found in the `solver_sparse` function in the file diffu2D_u0.py.

3.44.5. Verification

We can easily extend the function `quadratic` from Section Section 3.42 to include a test of the `solver_sparse` function as well.

```
def quadratic(theta, Nx, Ny):
    ...
    t, cpu = solver_sparse(
        I, a, f, Lx, Ly, Nx, Ny,
        dt, T, theta, user_action=assert_no_error)
```

3.45. The Jacobi iterative method

So far we have created a matrix and right-hand side of a linear system $Ac = b$ and solved the system for c by calling an exact algorithm based on Gaussian elimination. A much simpler implementation, which requires no memory for the coefficient matrix A, arises if we solve the system by *iterative* methods. These methods are only approximate, and the core algorithm is repeated many times until the solution is considered to be converged.

3.45.1. Numerical scheme and linear system

To illustrate the idea of the Jacobi method, we simplify the numerical scheme to the Backward Euler case, $\theta = 1$, so there are fewer terms to write:

$$u^{n+1}_{i,j} - \left(F_x(u^{n+1}_{i-1,j} - 2u^{n+1}_{i,j} + u^{n+1}_{i+1,j}) + F_y(u^{n+1}_{i,j-1} - 2u^{n+1}_{i,j} + u^{n+1}_{i,j+1})\right) = u^n_{i,j} + \Delta t f^{n+1}_{i,j} \tag{3.78}$$

The idea of the *Jacobi* iterative method is to introduce an iteration, here with index r, where we in each iteration treat $u^{n+1}_{i,j}$ as unknown, but use values from the previous iteration for the other unknowns $u^{n+1}_{i\pm1,j\pm1}$.

3.45.2. Iterations

Let $u^{n+1,r}_{i,j}$ be the approximation to $u^{n+1}_{i,j}$ in iteration r, for all relevant i and j indices. We first solve with respect to $u^{n+1}_{i,j}$ to get the equation to solve:

$$u^{n+1}_{i,j} = (1 + 2F_x + 2F_y)^{-1}\left(F_x(u^{n+1}_{i-1,j} + u^{n+1}_{i+1,j}) + F_y(u^{n+1}_{i,j-1} + u^{n+1}_{i,j+1})\right) + u^n_{i,j} + \Delta t f^{n+1}_{i,j} \tag{3.79}$$

The iteration is introduced by using iteration index r, for computed values, on the right-hand side and $r+1$ (unknown in this iteration) on the left-hand side:

$$u_{i,j}^{n+1,r+1} = (1+2F_x+2F_y)^{-1}\left(F_x(u_{i-1,j}^{n+1,r}+u_{i+1,j}^{n+1,r})+F_y(u_{i,j-1}^{n+1,r}+u_{i,j+1}^{n+1,r})\right) + u_{i,j}^{n}+\Delta t f_{i,j}^{n+1} \tag{3.80}$$

3.45.3. Initial guess

We start the iteration with the computed values at the previous time level:

$$u^{n+1,0}**i,j = u^n**i,j, \quad i=0,\ldots,N_x,\ j=0,\ldots,N_y\,. \tag{3.81}$$

3.45.4. Relaxation

A common technique in iterative methods is to introduce a *relaxation*, which means that the new approximation is a weighted mean of the approximation as suggested by the algorithm and the previous approximation. Naming the quantity on the left-hand side of (3.80) as $u_{i,j}^{n+1,*}$, a new approximation based on relaxation reads

$$u^{n+1,r+1} = \omega u^{n+1,*}**i,j + (1-\omega)u^{n+1,r}**i,j\,. \tag{3.82}$$

Under-relaxation means $\omega < 1$, while over-relaxation has $\omega > 1$.

3.45.5. Stopping criteria

The iteration can be stopped when the change from one iteration to the next is sufficiently small ($\leq \epsilon$), using either an infinity norm,

$$\max_{i,j}\left|u^{n+1,r+1}**i,j - u^{n+1,r}**i,j\right| \leq \epsilon,$$

or an L^2 norm,

$$\left(\Delta x\Delta y\sum_{i,j}(u^{n+1,r+1}**i,j - u^{n+1,r}**i,j)^2\right)^{\frac{1}{2}} \leq \epsilon\,.$$

Another widely used criterion measures how well the equations are solved by looking at the residual (essentially $b-Ac^{r+1}$ if c^{r+1} is the approximation to the solution in iteration $r+1$). The residual, defined in terms of the finite difference stencil, is

$$\begin{aligned} R_{i,j} = u_{i,j}^{n+1,r+1} - (&F_x(u_{i-1,j}^{n+1,r+1} - 2u_{i,j}^{n+1,r+1} + u_{i+1,j}^{n+1,r+1}) + \\ &F_y(u_{i,j-1}^{n+1,r+1} - 2u_{i,j}^{n+1,r+1} + u_{i,j+1}^{n+1,r+1})) - \\ &u_{i,j}^{n} - \Delta t f_{i,j}^{n+1} \end{aligned} \tag{3.83}$$

One can then iterate until the norm of the mesh function $R_{i,j}$ is less than some tolerance:

$$\left(\Delta x\Delta y\sum_{i,j}R_{i,j}^2\right)^{\frac{1}{2}} \leq \epsilon\,.$$

Code-friendly notation To make the mathematics as close as possible to what we will write in a computer program, we may introduce some new notation: $u_{i,j}$ is a short notation for $u^{n+1,r+1}**i,j$, u^-**i,j is a short notation for $u^{n+1,r}**i,j$, and $u^{(s)}**i,j$ denotes $u^{n+1-s}**i,j$. That is, $u**i,j$ is the unknown, $u^-_{i,j}$ is its most recently computed approximation, and s counts time levels backwards in time. The Jacobi method (3.80)) takes the following form with the new notation:

$$\begin{aligned} u^*_{i,j} = (1+2F_x+2F_y)^{-1}((F_x(u^-_{i-1,j}+u^-_{i+1,j})+F_y(u^-_{i,j-1}+u^-_{i,j+1}))+\\ u^{(1)}_{i,j}+\Delta t f^{n+1}_{i,j}) \end{aligned} \tag{3.84}$$

3.45.6. Generalization of the scheme

We can also quite easily introduce the θ rule for discretization in time and write up the Jacobi iteration in that case as well:

$$\begin{aligned} u^*_{i,j} = (1+2\theta(F_x+F_y))^{-1}(\theta(F_x(u^-_{i-1,j}+u^-_{i+1,j})+F_y(u^-_{i,j-1}+u^-_{i,j+1}))+\\ u^{(1)}_{i,j}+\theta\Delta t f^{n+1}_{i,j}+(1-\theta)\Delta t f^n_{i,j}+\\ (1-\theta)(F_x(u^{(1)}_{i-1,j}-2u^{(1)}_{i,j}+u^{(1)}_{i+1,j})+F_y(u^{(1)}_{i,j-1}-2u^{(1)}_{i,j}+u^{(1)}_{i,j+1}))\,. \end{aligned} \tag{3.85}$$

The final update of u applies relaxation:

$$u_{i,j} = \omega u^**i,j+(1-\omega)u^-**i,j\,.$$

Implementation of the Jacobi method {#sec-diffu-2D-Jacobi-impl}

The Jacobi method needs no coefficient matrix and right-hand side vector, but it needs an array for u in the previous iteration. We call this array `u_`, using the notation at the end of the previous section (at the same time level). The unknown itself is called `u`, while `u_n` is the computed solution one time level back in time. With a θ rule in time, the time loop can be coded like this:

```
for n in It[0:-1]:
    u_[:,:] = u_n  # Start value
    converged = False
    r = 0
    while not converged:
        if version == 'scalar':
            j = 0
            for i in Ix:
                u[i,j] = U_0y(t[n+1])           # Boundary
            for j in It[1:-1]:
                i = 0;   u[i,j] = U_0x(t[n+1])  # Boundary
                i = Nx;  u[i,j] = U_Lx(t[n+1])  # Boundary
```

```
                for i in Ix[1:-1]:
                    u_new = 1.0/(1.0 + 2*theta*(Fx + Fy))*(theta*(
                        Fx*(u_[i+1,j] + u_[i-1,j]) +
                        Fy*(u_[i,j+1] + u_[i,j-1])) + \
                    u_n[i,j] + \
                    (1-theta)*(Fx*(
                    u_n[i+1,j] - 2*u_n[i,j] + u_n[i-1,j]) +
                      Fy*(
                    u_n[i,j+1] - 2*u_n[i,j] + u_n[i,j-1]))\
                      + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
                    (1-theta)*dt*f(i*dx,j*dy,n*dt))
                    u[i,j] = omega*u_new + (1-omega)*u_[i,j]
            j = Ny
            for i in Ix:
                u[i,j] = U_Ly(t[n+1])      # Boundary

        elif version == 'vectorized':
            j = 0;  u[:,j] = U_0y(t[n+1])  # Boundary
            i = 0;  u[i,:] = U_0x(t[n+1])  # Boundary
            i = Nx; u[i,:] = U_Lx(t[n+1])  # Boundary
            j = Ny; u[:,j] = U_Ly(t[n+1])  # Boundary
            f_a_np1 = f(xv, yv, t[n+1])
            f_a_n   = f(xv, yv, t[n])
            u_new = 1.0/(1.0 + 2*theta*(Fx + Fy))*(theta*(Fx*(
              u_[2:,1:-1] + u_[:-2,1:-1]) +
                Fy*(
              u_[1:-1,2:] + u_[1:-1,:-2])) +\
            u_n[1:-1,1:-1] + \
              (1-theta)*(Fx*(
              u_n[2:,1:-1] - 2*u_n[1:-1,1:-1] + u_n[:-2,1:-1]) +\
                Fy*(
              u_n[1:-1,2:] - 2*u_n[1:-1,1:-1] + u_n[1:-1,:-2]))\
              + theta*dt*f_a_np1[1:-1,1:-1] + \
              (1-theta)*dt*f_a_n[1:-1,1:-1])
            u[1:-1,1:-1] = omega*u_new + (1-omega)*u_[1:-1,1:-1]
        r += 1
        converged = np.abs(u-u_).max() < tol or r >= max_iter
        u_[:,:] = u

    u_n, u = u, u_n
```

The vectorized version should be quite straightforward to understand once one has an understanding of how a standard 2D finite stencil is vectorized.

The first natural verification is to use the test problem in the function `quadratic` from Section Section 3.42. This problem is known to have no approximation error, but any iterative method will produce an approximate solution with unknown error. For a tolerance 10^{-k} in the iterative method,

we can, e.g., use a slightly larger tolerance $10^{-(k-1)}$ for the difference between the exact and the computed solution.

```
def quadratic(theta, Nx, Ny):
    ...
    def assert_small_error(u, x, xv, y, yv, t, n):
        """Assert small error for iterative methods."""
        u_e = u_exact(xv, yv, t[n])
        diff = abs(u - u_e).max()
        tol = 1E-4
        msg = 'diff=%g, step %d, time=%g' % (diff, n, t[n])
        assert diff < tol, msg

    for version in 'scalar', 'vectorized':
        for theta in 1, 0.5:
            print 'testing Jacobi, %s version, theta=%g' % \
                  (version, theta)
            t, cpu = solver_Jacobi(
                I=I, a=a, f=f, Lx=Lx, Ly=Ly, Nx=Nx, Ny=Ny,
                dt=dt, T=T, theta=theta,
                U_0x=0, U_0y=0, U_Lx=0, U_Ly=0,
                user_action=assert_small_error,
                version=version, iteration='Jacobi',
                omega=1.0, max_iter=100, tol=1E-5)
```

Even for a very coarse 4×4 mesh, the Jacobi method requires 26 iterations to reach a tolerance of 10^{-5}, which is quite many iterations, given that there are only 25 unknowns.

3.46. Test problem: diffusion of a sine hill

It can be shown that

$$u_{\mathrm{e}} = Ae^{-\alpha\pi^2(L_x^{-2}+L_y^{-2})t}\sin\left(\frac{\pi}{L_x}x\right)\sin\left(\frac{\pi}{L_y}y\right), \tag{3.86}$$

is a solution of the 2D homogeneous diffusion equation $u_t = \alpha(u_{xx}+u_{yy})$ in a rectangle $[0, L_x]\times[0, L_y]$, for any value of the amplitude A. This solution vanishes at the boundaries, and the initial condition is the product of two sines. We may choose $A = 1$ for simplicity.

It is difficult to know if our solver based on the Jacobi method works properly since we are faced with two sources of errors: one from the discretization, E_Δ, and one from the iterative Jacobi method, E_i. The total error in the computed u can be represented as

$$E_u = E_\Delta + E_i\,.$$

One error measure is to look at the maximum value, which is obtained for the midpoint $x = L_x/2$ and $y = L_x/2$. This midpoint is represented in the discrete `u` if N_x and N_y are even numbers. We can then compute E_u as $E_u = |\max u_{\mathrm{e}} - \max u|$, when we know an exact solution u_{e} of the problem.

What about E_Δ? If we use the maximum value as a measure of the error, we have in fact analytical insight into the approximation error in this particular problem. According to Section Section 3.23, the exact solution (3.86) of the PDE problem is also an exact solution of the discrete equations, except that the damping factor in time is different. More precisely, (3.52) and (3.53) are solutions of the discrete problem for $\theta = 1$ (Backward Euler) and $\theta = \frac{1}{2}$ (Crank-Nicolson), respectively. The factors raised to the power n is the numerical amplitude, and the errors in these factors become

$$E_\Delta = e^{-\alpha k^2 t} - \left(\frac{1 - 2(F_x \sin^2 p_x + F_x \sin^2 p_y)}{1 + 2(F_x \sin^2 p_x + F_x \sin^2 p_y)}\right)^n, \quad \theta = \frac{1}{2},$$
$$E_\Delta = e^{-\alpha k^2 t} - (1 + 4F_x \sin^2 p_x + 4F_y \sin^2 p_y)^{-n}, \quad \theta = 1\,.$$

We are now in a position to compute E_i numerically. That is, we can compute the error due to iterative solution of the linear system and see if it corresponds to the convergence tolerance used in the method. Note that the convergence is based on measuring the difference in two consecutive approximations, which is not exactly the error due to the iteration, but it is a kind of measure, and it should have about the same size as E_i.

The function `demo_classic_iterative` in `diffu2D_u0.py` implements the idea above (also for the methods in Section Section 3.48). The value of E_i is in particular printed at each time level. By changing the tolerance in the convergence criterion of the Jacobi method, we can see that E_i is of the same order of magnitude as the prescribed tolerance in the Jacobi method. For example: $E_\Delta \sim 10^{-2}$ with $N_x = N_y = 10$ and $\theta = \frac{1}{2}$, as long as $\max u$ has some significant size ($\max u > 0.02$). An appropriate value of the tolerance is then 10^{-3}, such that the error in the Jacobi method does not become bigger than the discretization error. In that case, E_i is around $5 \cdot 10^{-3}$. The corresponding number of Jacobi iterations (with $\omega = 1$) varies from 31 to 12 during the time simulation (for $\max u > 0.02$). Changing the tolerance to 10^{-5} causes many more iterations (61 to 42) without giving any contribution to the overall accuracy, because the total error is dominated by E_Δ.

Also, with an $N_x = N_y = 20$, the spatial accuracy increases and many more iterations are needed (143 to 45), but the dominating error is from the time discretization. However, with such a finer spatial mesh, a higher tolerance in the convergence criterion 10^{-4} is needed to keep $E_i \sim 10^{-3}$. More experiments show the disadvantage of the very simple Jacobi iteration method: the number of iterations increases with the number of unknowns, keeping the tolerance fixed, but the tolerance should also be lowered to avoid the iteration error to dominate the total error. A small adjustment of the Jacobi method, as described in Section Section 3.48, provides a better method.

3.47. The relaxed Jacobi method and its relation to the Forward Euler method

We shall now show that solving the Poisson equation $-\alpha\nabla^2 u = f$ by the Jacobi iterative method is in fact equivalent to using a Forward Euler scheme on $u_t = \alpha\nabla^2 u + f$ and letting $t \to \infty$.

A Forward Euler discretization of the 2D diffusion equation,

$$[D_t^+ u = \alpha(D_x D_x u + D_y D_y u) + f]_{i,j}^n,$$

can be written out as

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\Delta t}{\alpha h^2}\left(u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n - 4u_{i,j}^n + h^2 f_{i,j}\right),$$

where $h = \Delta x = \Delta y$ has been introduced for simplicity. The scheme can be reordered as

$$u_{i,j}^{n+1} = (1-\omega)\, u_{i,j}^n + \frac{1}{4}\omega\left(u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n - 4u_{i,j}^n + h^2 f_{i,j}\right),$$

with

$$\omega = 4\frac{\Delta t}{\alpha h^2},$$

but this latter form is nothing but the relaxed Jacobi method applied to

$$[D_x D_x u + D_y D_y u = -f]_{i,j}^n\,.$$

From the equivalence above we know a couple of things about the Jacobi method for solving $-\nabla^2 u = f$:

1. The method is unstable if $\omega > 1$ (since the Forward Euler method is then unstable).
2. The convergence is really slow as the iteration index increases (coming from the fact that the Forward Euler scheme requires many small time steps to reach the stationary solution).

These observations are quite disappointing: if we already have a time-dependent diffusion problem and want to take larger time steps by an implicit time discretization method, we will with the Jacobi method end up with something close to a slow Forward Euler simulation of the original problem at each time level. Nevertheless, the are two reasons for why the Jacobi method remains a fundamental building block for solving linear systems arising from PDEs: 1) a couple of iterations remove large parts of the error and this is effectively used in the very efficient class of multigrid methods; and 2) the idea of the Jacobi method can be developed into more efficient methods, especially the SOR method, which is treated next.

3.48. The Gauss-Seidel and SOR methods

If we update the mesh points according to the Jacobi method (3.79) for a Backward Euler discretization with a loop over $i = 1, \ldots, N_x - 1$ and $j = 1, \ldots, N_y - 1$, we realize that when $u^{n+1,r+1} **i, j$ is computed, $u^{n+1,r+1} **i-1, j$ and $u_{i,j-1}^{n+1,r+1}$ are already computed, so these new values can be used rather than $u^{n+1,r} **i-1, j$ and $u^{n+1,r} **i, j-1$ (respectively) in the formula for $u_{i,j}^{n+1,r+1}$. This idea gives rise to the *Gauss-Seidel* iteration method, which mathematically is just a small adjustment of (3.79):

$$\begin{aligned} u_{i,j}^{n+1,r+1} &= (1 + 2F_x + 2F_y)^{-1}((\\ &\quad F_x(u_{i-1,j}^{n+1,r+1} + u_{i+1,j}^{n+1,r}) + F_y(u_{i,j-1}^{n+1,r+1} + u_{i,j+1}^{n+1,r})) + u_{i,j}^n + \Delta t f_{i,j}^{n+1})\,. \end{aligned} \tag{3.87}$$

Observe that the way we access the mesh points in the formula (3.87) is important: points with $i-1$ must be computed before points with i, and points with $j-1$ must be computed before points with j. Any sequence of mesh points can be used in the Gauss-Seidel method, but the particular math

formula must distinguish between already visited points in the current iteration and the points not yet visited.

The idea of relaxation (3.82) can equally well be applied to the Gauss-Seidel method. Actually, the Gauss-Seidel method with an arbitrary $0 < \omega \leq 2$ has its own name: the *Successive Over-Relaxation* method, abbreviated as SOR.

The SOR method for a θ rule discretization, with the shortened u and u^- notation, can be written

$$\begin{aligned} u^*_{i,j} = (1 + 2\theta(F_x + F_y))^{-1}(\theta(F_x(u_{i-1,j} + u^-_{i+1,j}) + F_y(u_{i,j-1} + u^-_{i,j+1}))+ \\ u^{(1)}_{i,j} + \theta\Delta t f^{n+1}_{i,j} + (1-\theta)\Delta t f^n_{i,j} + \\ (1-\theta)(F_x(u^{(1)}_{i-1,j} - 2u^{(1)}_{i,j} + u^{(1)}_{i+1,j}) + F_y(u^{(1)}_{i,j-1} - 2u^{(1)}_{i,j} + u^{(1)}_{i,j+1}))), \end{aligned}$$

$$u_{i,j} = \omega u^*_{i,j} + (1-\omega)u^-_{i,j} \tag{3.88}$$

The sequence of mesh points in (3.88) is $i = 1, \ldots, N_x - 1$, $j = 1, \ldots, N_y - 1$ (but whether i runs faster or slower than j does not matter).

3.49. Scalar implementation of the SOR method

Since the Jacobi and Gauss-Seidel methods with relaxation are so similar, we can easily make a common code for the two:

```
for n in It[0:-1]:
    u_[:,:] = u_n  # Start value
    converged = False
    r = 0
    while not converged:
        if version == 'scalar':
            if iteration == 'Jacobi':
                u__ = u_
            elif iteration == 'SOR':
                u__ = u
            j = 0
            for i in Ix:
                u[i,j] = U_0y(t[n+1])  # Boundary
            for j in It[1:-1]:
                i = 0;   u[i,j] = U_0x(t[n+1])  # Boundary
                i = Nx;  u[i,j] = U_Lx(t[n+1])  # Boundary
                for i in Ix[1:-1]:
                    u_new = 1.0/(1.0 + 2*theta*(Fx + Fy))*(theta*(
                        Fx*(u_[i+1,j] + u__[i-1,j]) +
                        Fy*(u_[i,j+1] + u__[i,j-1])) + \
                    u_n[i,j] + (1-theta)*(
                      Fx*(
                    u_n[i+1,j] - 2*u_n[i,j] + u_n[i-1,j]) +
```

```
                      Fy*(
                    u_n[i,j+1] - 2*u_n[i,j] + u_n[i,j-1]))\
                      + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
                    (1-theta)*dt*f(i*dx,j*dy,n*dt))
                    u[i,j] = omega*u_new + (1-omega)*u_[i,j]
                j = Ny
                for i in Ix:
                    u[i,j] = U_Ly(t[n+1])  # boundary
        r += 1
        converged = np.abs(u-u_).max() < tol or r >= max_iter
        u_[:,:] = u

    u_n, u = u, u_n  # Get ready for next iteration
```

The idea here is to introduce `u__` to be used for already computed values (`u`) in the Gauss-Seidel/SOR version of the implementation, or just values from the previous iteration (`u_`) in case of the Jacobi method.

3.50. Vectorized implementation of the SOR method

Vectorizing the Gauss-Seidel iteration step turns out to be non-trivial. The problem is that vectorized operations typically imply operations on arrays where the sequence in which we visit the elements does not matter. In particular, this principle makes vectorized code trivial to parallelize. However, in the Gauss-Seidel algorithm, the sequence in which we visit the elements in the arrays does matter, and it is well known that the basic method as explained above cannot be parallelized. Therefore, also vectorization will require new thinking.

The strategy for vectorizing (and parallelizing) the Gauss-Seidel method is to use a special numbering of the mesh points called red-black numbering: every other point is red or black as in a checkerboard pattern. This numbering requires N_x and N_y to be even numbers. Here is an example of a 6×6 mesh:

```
r b r b r b r
b r b r b r b
r b r b r b r
b r b r b r b
r b r b r b r
b r b r b r b
r b r b r b r
```

The idea now is to first update all the red points. Each formula for updating a red point involves only the black neighbors. Thereafter, we update all the black points, and at each black point, only the recently computed red points are involved.

The scalar implementation of the red-black numbered Gauss-Seidel method is really compact, since we can update values directly in `u` (this guarantees that we use the most recently computed values). Here is the relevant code for the Backward Euler scheme in time and without a source term:

```
for sweep in 'red', 'black':
    for j in range(1, Ny, 1):
        if sweep == 'red':
            start = 1 if j % 2 == 1 else 2
        elif sweep == 'black':
            start = 2 if j % 2 == 1 else 1
        for i in range(start, Nx, 2):
        u[i,j] = 1.0/(1.0 + 2*(Fx + Fy))*(
                     Fx*(u[i+1,j] + u[i-1,j]) +
                     Fy*(u[i,j+1] + u[i,j-1]) + u_n[i,j])
```

The vectorized version must be based on slices. Looking at a typical red-black pattern, e.g.,

```
r b r b r b r
b r b r b r b
r b r b r b r
b r b r b r b
r b r b r b r
b r b r b r b
r b r b r b r
```

we want to update the internal points (marking boundary points with `x`):

```
x x x x x x x
x r b r b r x
x b r b r b x
x r b r b r x
x b r b r b x
x r b r b r x
x x x x x x x
```

It is impossible to make one slice that picks out all the internal red points. Instead, we need two slices. The first involves points marked with `R`:

```
x x x x x x x
x R b R b R x
x b r b r b x
x R b R b R x
x b r b r b x
x R b R b R x
x x x x x x x
```

This slice is specified as `1::2` for `i` and `1::2` for `j`, or with `slice` objects:

```
i = slice(1, None, 2);  j = slice(1, None, 2)
```

The second slice involves the red points with `R`:

```
x x x x x x x
x r b r b r x
x b R b R b x
x r b r b r x
x b R b R b x
x r b r b r x
x x x x x x x
```

The slices are

```
i = slice(2, None, 2);  j = slice(2, None, 2)
```

For the black points, the first slice involves the `B` points:

```
x x x x x x x
x r B r B r x
x b r b r b x
x r B r B r x
x b r b r b x
x r B r B r x
x x x x x x x
```

with slice objects

```
i = slice(2, None, 2);  j = slice(1, None, 2)
```

The second set of black points is shown here:

```
x x x x x x x
x r b r b r x
x B r B r B x
x r b r b r x
x B r B r B x
x r b r b r x
x x x x x x x
```

with slice objects

```
i = slice(1, None, 2);  j = slice(2, None, 2)
```

That is, we need four sets of slices. The simplest way of implementing the algorithm is to make a function with variables for the slices representing i, $i-1$, $i+1$, j, $j-1$, and $j+1$, here called `ic` ("i center"), `im1` ("i minus 1", `ip1` ("i plus 1"), `jc`, `jm1`, and `jp1`, respectively.

```
def update(u_, u_n, ic, im1, ip1, jc, jm1, jp1):
    return \
       1.0/(1.0 + 2*theta*(Fx + Fy))*(theta*(
           Fx*(u_[ip1,jc] + u_[im1,jc]) +
           Fy*(u_[ic,jp1] + u_[ic,jm1])) +\
       u_n[ic,jc] + (1-theta)*(
         Fx*(u_n[ip1,jc] - 2*u_n[ic,jc] + u_n[im1,jc]) +\
         Fy*(u_n[ic,jp1] - 2*u_n[ic,jc] + u_n[ic,jm1]))+\
         theta*dt*f_a_np1[ic,jc] + \
         (1-theta)*dt*f_a_n[ic,jc])
```

The formula returned from `update` is to be compared with (3.88).

The relaxed Jacobi iteration can be implemented by

```
ic  = jc  = slice(1,-1)
im1 = jm1 = slice(0,-2)
ip1 = jp1 = slice(2,None)
u_new[ic,jc] = update(
    u_, u_n, ic, im1, ip1, jc, jm1, jp1)
u[ic,jc] = omega*u_new[ic,jc] + (1-omega)*u_[ic,jc]
```

The Gauss-Seidel (or SOR) updates need four different steps. The `ic` and `jc` slices are specified above. For each of these, we must specify the corresponding `im1`, `ip1`, `jm1`, and `jp1` slices. The code below contains the details.

```
ic  = slice(1,-1,2)
im1 = slice(0,-2,2)
ip1 = slice(2,None,2)
jc  = slice(1,-1,2)
jm1 = slice(0,-2,2)
jp1 = slice(2,None,2)
u_new[ic,jc] = update(
    u_new, u_n, ic, im1, ip1, jc, jm1, jp1)

ic  = slice(2,-1,2)
im1 = slice(1,-2,2)
ip1 = slice(3,None,2)
jc  = slice(2,-1,2)
jm1 = slice(1,-2,2)
jp1 = slice(3,None,2)
u_new[ic,jc] = update(
    u_new, u_n, ic, im1, ip1, jc, jm1, jp1)

ic  = slice(2,-1,2)
im1 = slice(1,-2,2)
ip1 = slice(3,None,2)
```

```
jc  = slice(1,-1,2)
jm1 = slice(0,-2,2)
jp1 = slice(2,None,2)
u_new[ic,jc] = update(
    u_new, u_n, ic, im1, ip1, jc, jm1, jp1)

ic  = slice(1,-1,2)
im1 = slice(0,-2,2)
ip1 = slice(2,None,2)
jc  = slice(2,-1,2)
jm1 = slice(1,-2,2)
jp1 = slice(3,None,2)
u_new[ic,jc] = update(
    u_new, u_n, ic, im1, ip1, jc, jm1, jp1)

c = slice(1,-1)
u[c,c] = omega*u_new[c,c] + (1-omega)*u_[c,c]
```

The function `solver_classic_iterative` in diffu2D_u0.py contains a unified implementation of the relaxed Jacobi and SOR methods in scalar and vectorized versions using the techniques explained above.

3.51. Direct versus iterative methods

3.51.1. Direct methods

There are two classes of methods for solving linear systems: direct methods and iterative methods. Direct methods are based on variants of the Gaussian elimination procedure and will produce an exact solution (in exact arithmetics) in an a priori known number of steps. Iterative methods, on the other hand, produce an approximate solution, and the amount of work for reaching a given accuracy is usually not known.

The most common direct method today is to use the *LU factorization* procedure to factor the coefficient matrix A as the product of a lower-triangular matrix L (with unit diagonal terms) and an upper-triangular matrix U: $A = LU$. As soon as we have L and U, a system of equations $LUc = b$ is easy to solve because of the triangular nature of L and U. We first solve $Ly = b$ for y (forward substitution), and thereafter we find c from solving $Uc = y$ (backward substitution). When A is a dense $N \times N$ matrix, the LU factorization costs $\frac{1}{3}N^3$ arithmetic operations, while the forward and backward substitution steps each require of the order N^2 arithmetic operations. That is, factorization dominates the costs, while the substitution steps are cheap.

Symmetric, positive definite coefficient matrices often arise when discretizing PDEs. In this case, the LU factorization becomes $A = LL^T$, and the associated algorithm is known as *Cholesky factorization*. Most linear algebra software offers highly optimized implementations of LU and Cholesky factorization as well as forward and backward substitution (`scipy.linalg` is the relevant Python package).

Finite difference discretizations lead to sparse coefficient matrices. An extreme case arose in Section Section 3.8 where A was tridiagonal. For a tridiagonal matrix, the amount of arithmetic operations in the LU and Cholesky factorization algorithms is just of the order N, not N^3. Tridiagonal matrices are special cases of *banded matrices*, where the matrices contain just a set of diagonal bands. Finite difference methods on regularly numbered rectangular and box-shaped meshes give rise to such banded matrices, with 5 bands in 2D and 7 in 3D for diffusion problems. Gaussian elimination only needs to work within the bands, leading to much more efficient algorithms.

If $A_{i,j} = 0$ for $j > i + p$ and $j < i - p$, p is the *half-bandwidth* of the matrix. We have in our 2D problem $p = N_x + 2$, while in 3D, $p = (N_x + 1)(N_y + 1) + 2$. The cost of Gaussian elimination is then $\mathcal{O}(Np^2)$, so with $p \ll N$, we see that banded matrices are much more efficient to compute with. By reordering the unknowns in clever ways, one can reduce the work of Gaussian elimination further. Fortunately, the Python programmer has access to such algorithms through the `scipy.sparse.linalg` package.

Although a direct method is an exact algorithm, rounding errors may in practice accumulate and pollute the solution. The effect grows with the size of the linear system, so both for accuracy and efficiency, iterative methods are better suited than direct methods for solving really large linear systems.

3.51.2. Iterative methods

The Jacobi and SOR iterative methods belong to a class of iterative methods where the idea is to solve $Au = b$ by splitting A into two parts, $A = M - N$, such that solving systems $Mu = c$ is easy and efficient. With the splitting, we get a system

$$Mu = Nu + b,$$

which suggests an iterative method

$$Mu^{r+1} = Nu^r + b, \quad r = 0, 1, 2, \ldots,$$

where u^{r+1} is a new approximation to u in the $r + 1$-th iteration. To initiate the iteration, we need a start vector u^0.

The Jacobi and SOR methods are based on splitting A into a lower tridiagonal part L, the diagonal D, and an upper tridiagonal part U, such that $A = L + D + U$. The Jacobi method corresponds to $M = D$ and $N = -L - U$. The Gauss-Seidel method employs $M = L + D$ and $N = -U$, while the SOR method corresponds to

$$M = \frac{1}{\omega}D + L, \quad N = \frac{1-\omega}{\omega}D - U\,.$$

The relaxed Jacobi method has similar expressions:

$$M = \frac{1}{\omega}D, \quad N = \frac{1-\omega}{\omega}D - L - U\,.$$

With the matrix forms of the Jacobi and SOR methods as written above, we could in an implementation alternatively fill the matrix A with entries and call general implementations of the Jacobi or SOR methods that work on a system $Au = b$. However, this is almost never done since forming the matrix A requires quite some code and storing A in the computer's memory is unnecessary. It is

much easier to just apply the Jacobi and SOR ideas to the finite difference stencils directly in an implementation, as we have shown in detail.

Nevertheless, the matrix formulation of the Jacobi and SOR methods have been important for analyzing their convergence behavior. One can show that the error $u^r - u$ fulfills $u^r - u = G^r(u^0 - u)$, where $G = M^{-1}N$ and G^k is a matrix exponential. For the method to converge, $\lim_{r\to\infty} ||G^r|| = 0$ is a necessary and sufficient condition. This implies that the *spectral radius* of G must be less than one. Since G is directly related to the finite difference scheme for the underlying PDE problem, one can in principle compute the spectral radius. For a given PDE problem, however, this is not a practical strategy, since it is very difficult to develop useful formulas. Analysis of model problems, usually related to the Poisson equation, reveals some trends of interest: the convergence rate of the Jacobi method goes like h^2, while that of SOR with an optimal ω goes like h, where h is the spatial spacing: $h = \Delta x = \Delta y$. That is, the efficiency of the Jacobi method quickly deteriorates with the increasing mesh resolution, and SOR is much to be preferred (even if the optimal ω remains an open question). We refer to Chapter 4 of (Saad 2003) for more information on the convergence theory. One important result is that if A is symmetric and positive definite, then SOR will converge for any $0 < \omega < 2$.

The optimal ω parameter can be theoretically established for a Poisson problem as

$$\omega_o = \frac{2}{1 + \sqrt{1 - \varrho^2}}, \quad \varrho = \frac{\cos(\pi/N_x) + (\Delta x/\Delta y)^2 \cos(\pi/N_y)}{1 + (\Delta x/\Delta y)^2}\,.$$

This formula can be used as a guide also in other problems.

The Jacobi and the SOR methods have their great advantage of being trivial to implement, so they are obviously popular of this reason. However, the slow convergence of these methods limits the popularity to fairly small linear systems (i.e., coarse meshes). As soon as the matrix size grows, one is better off with more sophisticated iterative methods like the preconditioned Conjugate gradient method, which we now turn to.

Finally, we mention that there is a variant of the SOR method, called the *Symmetric Successive Over-relaxation* method, known as SSOR, where one runs a standard SOR sweep through the mesh points and then a new sweep while visiting the points in reverse order.

3.52. The Conjugate gradient method

There is no simple intuitive derivation of the Conjugate gradient method, so we refer to the many excellent expositions in the literature for the idea of the method and how the algorithm is derived. In particular, we recommend the books (Barrett et al. 1994; Axelsson 1996; Saad 2003; Grief and Ascher 2011). A brief overview is provided in the Wikipedia article. Here, we just state the pros and cons of the method from a user's perspective and how we utilize it in code.

The original Conjugate gradient method is limited to linear systems $Au = b$, where A is a symmetric and positive definite matrix. There are, however, extensions of the method to non-symmetric matrices.

A major advantage of all conjugate gradient methods is that the matrix A is only used in matrix-vector products, so we do not need form and store A if we can provide code for computing a matrix-vector product Au. Another important feature is that the algorithm is very easy to vectorize

and parallelize. The primary downside of the method is that it converges slowly unless one has an effective *preconditioner* for the system. That is, instead of solving $Au = b$, we try to solve $M^{-1}Au = M^{-1}b$ in the hope that the method works better for this *preconditioned* system. The matrix M is the *preconditioner* or preconditioning matrix. Now we need to perform matrix-vector products $y = M^{-1}Au$, which is done in two steps: first the matrix-vector product $v = Au$ is carried out and then the system $My = v$ must be solved. Therefore, M must be cheap to compute and systems $My = v$ must be cheap to solve.

A perfect preconditioner is $M = A$, but in each iteration in the Conjugate gradient method one then has so solve a system with A as coefficient matrix! A key idea is to let M be some kind of *cheap approximation* to A. The simplest preconditioner is to set $M = D$, where D is the diagonal of A. This choice means running one Jacobi iteration as preconditioner. Exercise Section 3.70 shows that the Jacobi and SOR methods can also be viewed as preconditioners.

Constructing good preconditioners is a scientific field on its own. Here we shall treat the topic just very briefly. For a user having access to the `scipy.sparse.linalg` library, there are Conjugate gradient methods and preconditioners readily available:

- For positive definite, symmetric systems: `cg` (the Conjugate gradient method)
- For symmetric systems: `minres` (Minimum residual method)
- For non-symmetric systems:
- `gmres` (GMRES: Generalized minimum residual method)
- `bicg` (BiConjugate gradient method)
- `bicgstab` (Stabilized BiConjugate gradient method)
- `cgs` (Conjugate gradient squared method)
- `qmr` (Quasi-minimal residual iteration)
- Preconditioner: `spilu` (Sparse, incomplete LU factorization)

The ILU preconditioner is an attractive all-round type of preconditioner that is suitable for most problems on serial computers. A more efficient preconditioner is the multigrid method, and algebraic multigrid is also an all-round choice as preconditioner. The Python package PyAMG offers efficient implementations of the algebraic multigrid method, to be used both as a preconditioner and as a stand-alone iterative method.

The matrix arising from implicit time discretization methods applied to the diffusion equation is symmetric and positive definite. Thus, we can use the Conjugate gradient method (`cg`), typically in combination with an ILU preconditioner. The code is very similar to the one we created when solving the linear system by sparse Gaussian elimination, the main difference is that we now allow for calling up the Conjugate gradient function as an alternative solver.

```
def solver_sparse(
    I, a, f, Lx, Ly, Nx, Ny, dt, T, theta=0.5,
    U_0x=0, U_0y=0, U_Lx=0, U_Ly=0, user_action=None,
    method='direct', CG_prec='ILU', CG_tol=1E-5):
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time. Sparse matrix with
    dedicated Gaussian elimination algorithm (method='direct')
    or ILU preconditioned Conjugate Gradients (method='CG' with
```

```
    tolerance CG_tol and preconditioner CG_prec ('ILU' or None)).
    """

    ...

    A = scipy.sparse.diags(
        diagonals=[main, lower, upper, lower2, upper2],
        offsets=[0, -lower_offset, lower_offset,
                 -lower2_offset, lower2_offset],
        shape=(N, N), format='csc')

    if method == 'CG':
        if CG_prec == 'ILU':
            A_ilu = scipy.sparse.linalg.spilu(A)  # SuperLU defaults
            M = scipy.sparse.linalg.LinearOperator(
                shape=(N, N), matvec=A_ilu.solve)
        else:
            M = None
        CG_iter = []  # No of CG iterations at time level n

    for n in It[0:-1]:

        if method == 'direct':
            c = scipy.sparse.linalg.spsolve(A, b)
        elif method == 'CG':
            x0 = u_n.T.reshape(N)  # Start vector is u_n
            CG_iter.append(0)

            def CG_callback(c_k):
                """Trick to count the no of iterations in CG."""
                CG_iter[-1] += 1

            c, info = scipy.sparse.linalg.cg(
                A, b, x0=x0, tol=CG_tol, maxiter=N, M=M,
                callback=CG_callback)

        u_n, u = u, u_n
```

The number of iterations in the Conjugate gradient method is of interest, but is unfortunately not available from the `cg` function. Therefore, we perform a trick: in each iteration a user function `CG_callback` is called where we accumulate the number of iterations in a list `CG_iter`.

3.53. What is the recommended method for solving linear systems?

There is no clear answer to this question. If you have enough memory and computing time available, direct methods such as `spsolve` are to be preferred since they are easy to use and finds almost an

exact solution. However, in larger 2D and in 3D problems, direct methods usually run too slowly or require too much memory, so one is forced to use iterative methods. The fastest and most reliable methods are in the Conjugate Gradient family, but these require suitable preconditioners. ILU is an all-round preconditioner, but it is not suited for parallel computing. The Jacobi and SOR iterative methods are easy to implement, and popular for that reason, but run slowly. Jacobi iteration is not an option in real problems, but SOR may be.

3.54. Random walk

Models leading to diffusion equations, see Section Section 3.66, are usually based on reasoning with *averaged* physical quantities such as concentration, temperature, and velocity. The underlying physical processes involve complicated microscopic movement of atoms and molecules, but an average of a large number of molecules is performed in a small volume before the modeling starts, and the averaged quantity inside this volume is assigned as a point value at the centroid of the volume. This means that concentration, temperature, and velocity at a space-time point represent averages around the point in a small time interval and small spatial volume.

Random walk is a principally different kind of modeling procedure compared to the reasoning behind partial differential equations. The idea in random walk is to have a large number of "particles" that undergo random movements. Averaging can then be used afterwards to compute macroscopic quantities like concentration. The"particles" and their random movement represent a very simplified microscopic behavior of molecules, much simpler and computationally much more efficient than direct molecular simulation, yet the random walk model has been very powerful to describe a wide range of phenomena, including heat conduction, quantum mechanics, polymer chains, population genetics, neuroscience, hazard games, and pricing of financial instruments.

It can be shown that random walk, when averaged, produces models that are mathematically equivalent to diffusion equations. This is the primary reason why we treat random walk in this chapter: two very different algorithms (finite difference stencils and random walk) solve the same type of problems. The simplicity of the random walk algorithm makes it particularly attractive for solving diffusion equations on massively parallel computers. The exposition here is as simple as possible, and good thorough derivation of the models is provided by Hjorth-Jensen (Hjorth-Jensen 2016).

3.55. Random walk in 1D

Imagine that we have some particles that perform random moves, either to the right or to the left. We may flip a coin to decide the movement of each particle, say head implies movement to the right and tail means movement to the left. Each move is one unit length. Physicists use the term *random walk* for this type of movement. The movement is also known as drunkard's walk. You may try this yourself: flip the coin and make one step to the left or right, and repeat the process.

We introduce the symbol N for the number of steps in a random walk. Figure Figure 3.16 shows four different random walks with $N = 200$.

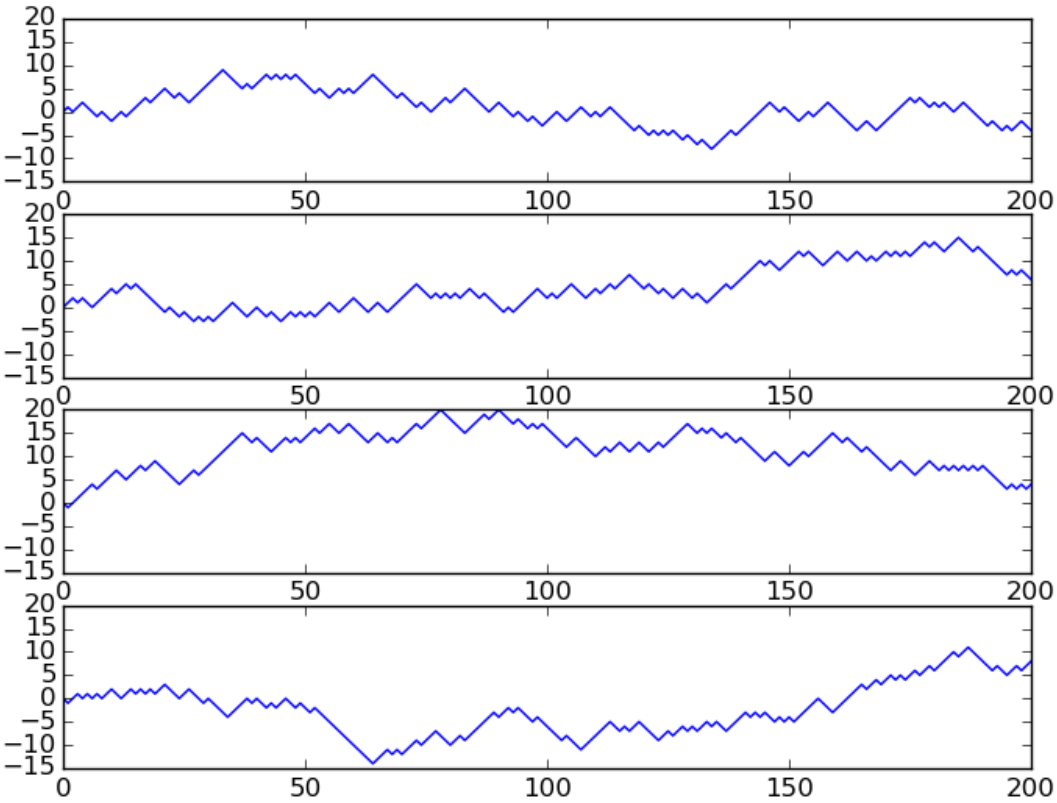


Figure 3.16.: Ensemble of 4 random walks, each with 200 steps.

3.56. Statistical considerations

Let S_k be the stochastic variable representing a step to the left or to the right in step number k. We have that $S_k = -1$ with probability p and $S_k = 1$ with probability $q = 1 - p$. The variable S_k is known as a Bernoulli variable. The expectation of S_k is

$$\mathrm{E}[S_k] = p \cdot (-1) + q \cdot 1 = 1 - 2p,$$

and the variance is

$$\mathrm{Var}[S_k] = \mathrm{E}[S_k^2] - \mathrm{E}[S_k]^2 = 1 - (1-2p)^2 = 4p(1-p)\,.$$

The position after k steps is another stochastic variable

$$\bar{X}_k = \sum_{i=0}^{k-1} S_i\,.$$

The expected position is

$$\mathrm{E}[\bar{X}_k] = \mathrm{E}[\sum_{i=0}^{k-1} S_i] = \sum_{i=0}^{k-1} \mathrm{E}[S_i] = k(1-2p)\,.$$

All the S_k variables are independent. The variance therefore becomes

$$\mathrm{Var}[\bar{X}_k] = \mathrm{Var}[\sum_{i=0}^{k-1} S_i] = \sum_{i=0}^{k-1} \mathrm{Var}[S_i] = k4p(1-p)\,.$$

We see that $\mathrm{Var}[\bar{X}_k]$ is proportional with the number of steps k. For the very important case $p = q = \frac{1}{2}$, $\mathrm{E}[\bar{X}_k] = 0$ and $\mathrm{Var}[\bar{X}_k] = k$.

How can we estimate $\mathrm{E}[\bar{X}_k] = 0$ and $\mathrm{Var}[\bar{X}_k] = N$? We must have many random walks of the type in Figure Figure 3.16. For a given k, say $k = 100$, we find all the values of $\bar{X}_k$, name them $\bar{x}_{0,k}$, $\bar{x}_{1,k}$, $\bar{x}_{2,k}$, and so on. The empirical estimate of $\mathrm{E}[\bar{X}_k]$ is the average,

$$\mathrm{E}[\bar{X}_k] \approx \frac{1}{W} \sum_{j=0}^{W-1} \bar{x}_{j,k},$$

while an empirical estimate of $\mathrm{Var}[\bar{X}_k]$ is

$$\mathrm{Var}[\bar{X}_k] \approx \frac{1}{W} \sum_{j=0}^{W-1} (\bar{x}_{j,k})^2 - \left(\frac{1}{W} \sum_{j=0}^{W-1} \bar{x}_{j,k} \right)^2 .$$

That is, we take the statistics for a given K across the ensemble of random walks ("vertically" in Figure Figure 3.16). The key quantities to record are $\sum_i \bar{x}_{i,k}$ and $\sum_i \bar{x}_{i,k}^2$.

3.57. Playing around with some code

3.57.1. Scalar code

Python has a `random` module for drawing random numbers, and this module has a function `uniform(a, b)` for drawing a uniformly distributed random number in the interval $[a, b)$. If an event happens with probability p, we can simulate this on the computer by drawing a random number r in $[0, 1)$, because then $r \leq p$ with probability p and $r > p$ with probability $1 - p$:

```
import random
r = random.uniform(0, 1)
if r <= p:
else:
```

A random walk with N steps, starting at x_0, where we move to the left with probability p and to the right with probability $1 - p$ can now be implemented by

```
import random, numpy as np

def random_walk1D(x0, N, p):
    """1D random walk with 1 particle."""
    position = np.zeros(N)
    position[0] = x0
    current_pos = x0
    for k in range(N-1):
        r = random.uniform(0, 1)
        if r <= p:
            current_pos -= 1
        else:
            current_pos += 1
        position[k+1] = current_pos
    return position
```

3.57.2. Vectorized code

Since N is supposed to be large and we want to repeat the process for many particles, we should speed up the code as much as possible. Vectorization is the obvious technique here: we draw all the random numbers at once with aid of `numpy`, and then we formulate vector operations to get rid of the loop over the steps (`k`). The `numpy.random` module has vectorized versions of the functions in Python's built-in `random` module. For example, `numpy.random.uniform(a, b, N)` returns `N` random numbers uniformly distributed between `a` (included) and `b` (not included).

We can then make an array of all the steps in a random walk: if the random number is less than or equal to p, the step is -1, otherwise the step is 1:

```
r = np.random.uniform(0, 1, size=N)
steps = np.where(r <= p, -1, 1)
```

The value of `position[k]` is the sum of all steps up to step `k`. Such sums are often needed in vectorized algorithms and therefore available by the `numpy.cumsum` function:

```
>>> import numpy as np
>>> np.cumsum(np.array([1,3,4,6]))
array([ 1,  4,  8, 14])
```

The resulting array in this demo has elements 1, $1+3=4$, $1+3+4=8$, and $1+3+4+6=14$.

We can now vectorize the `random_walk1D` function:

```
def random_walk1D_vec(x0, N, p):
    """Vectorized version of random_walk1D."""
    position = np.zeros(N + 1)
    position[0] = x0
    r = np.random.uniform(0, 1, size=N)
    steps = np.where(r <= p, -1, 1)
    position[1:] = x0 + np.cumsum(steps)
    return position
```

This code runs about 10 times faster than the scalar version. With a parallel `numpy` library, the code can also automatically take advantage of hardware for parallel computing because each of the four array operations can be trivially parallelized.

3.57.3. Fixing the random sequence

During software development with random numbers it is advantageous to always generate the same sequence of random numbers, as this may help debugging processes. To fix the sequence, we set a *seed* of the random number generator to some chosen integer, e.g.,

```
np.random.seed(10)
```

Calls to `random_walk1D_vec` give positions of the particle as depicted in Figure Figure 3.17. The particle starts at the origin and moves with $p = \frac{1}{2}$. Since the seed is the same, the plot to the left is just a magnification of the first 1,000 steps in the plot to the right.

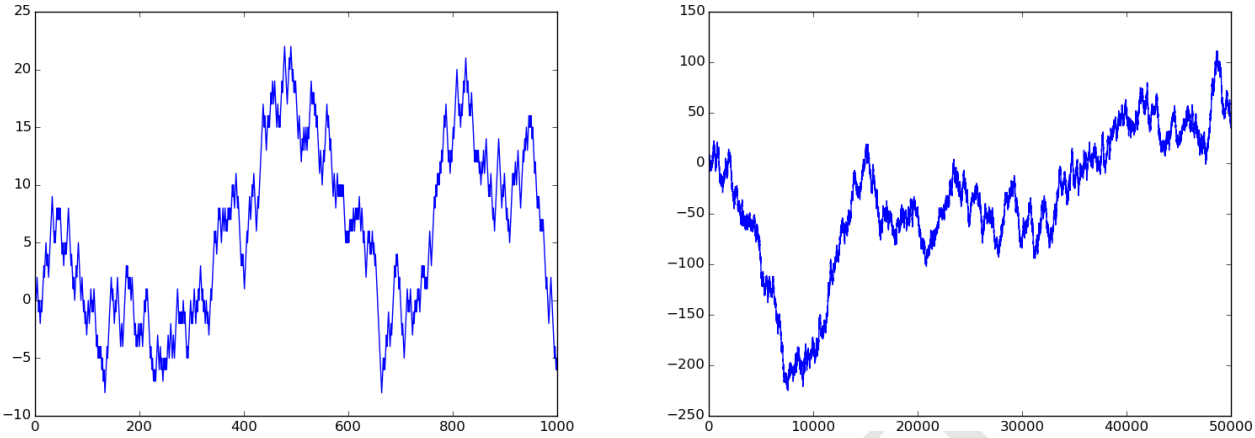


Figure 3.17.: 1,000 (left) and 50,000 (right) steps of a random walk.

3.57.4. Verification

When we have a scalar and a vectorized code, it is always a good idea to develop a unit test for checking that they produce the same result. A problem in the present context is that the two versions apply two different random number generators. For a test to be meaningful, we need to fix the seed and use the same generator. This means that the scalar version must either use `np.random` or have this as an option. An option is the most flexible choice:

```
import random

def random_walk1D(x0, N, p, random=random):
    ...
    r = random.uniform(0, 1)
```

Using `random=np.random`, the `r` variable gets computed by `np.random.uniform`, and the sequence of random numbers will be the same as in the vectorized version that employs the same generator (given that the seed is also the same). A proper test function may be to check that the positions in the walk are the same in the scalar and vectorized implementations:

```
def test_random_walk1D():
    x0 = 2
    N = 4
    p = 0.6
    np.random.seed(10)
    scalar_computed = random_walk1D(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walk1D_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()
```

Note that we employ `==` for arrays with real numbers, which is normally an inadequate test due to rounding errors, but in the present case, all arithmetics consists of adding or subtracting one, so these operations are expected to have no rounding errors. Comparing two `numpy` arrays with

`==` results in a boolean array, so we need to call the `all()` method to ensure that all elements are `True`, i.e., that all elements in the two arrays match each other pairwise.

3.58. Equivalence with diffusion

The original random walk algorithm can be said to work with dimensionless coordinates $\bar{x}_i = -N+i$, $i = 0, 1, \ldots, 2N+1$ ($i \in [-N, N]$), and $\bar{t}_n = n$, $n = 0, 1, \ldots, N$. A mesh with spacings Δx and Δt with dimensions can be introduced by

$$x_i = X_0 + \bar{x}_i \Delta x, \quad t_n = \bar{t}_n \Delta t\,.$$

If we implement the algorithm with dimensionless coordinates, we can just use this rescaling to obtain the movement in a coordinate system without unit spacings.

Let P_i^{n+1} be the probability of finding the particle at mesh point $\bar{x}_i$ at time $\bar{t}_{n+1}$. We can reach mesh point $(i, n+1)$ in two ways: either coming in from the left from $(i-1, n)$ or from the right $(i+1, n)$. Each has probability $\frac{1}{2}$ (if we assume $p = q = \frac{1}{2}$). The fundamental equation for P_i^{n+1} is

$$P^{n+1} * *i = \frac{1}{2} P^n * *i - 1 + \frac{1}{2} P^n_{i+1}\,. \tag{3.89}$$

(This equation is easiest to understand if one looks at the random walk as a Markov process and applies the transition probabilities, but this is beyond scope of the present text.)

Subtracting P_i^n from (3.89) results in

$$P_i^{n+1} - P^n * *i = \frac{1}{2}(P^n * *i - 1 - 2P^n * *i + \frac{1}{2} P^n * *i + 1)\,.$$

Readers who have seen the Forward Euler discretization of a 1D diffusion equation recognize this scheme as very close to such a discretization. We have

$$\frac{\partial}{\partial t} P(x_i, t_n) = \frac{P_i^{n+1} - P_i^n}{\Delta t} + \mathcal{O}(\Delta t),$$

or in dimensionless coordinates

$$\frac{\partial}{\partial \bar{t}} P(\bar{x}_i, \bar{t}_n) \approx P_i^{n+1} - P_i^n\,.$$

Similarly, we have

$$\frac{\partial^2}{\partial x^2} P(x_i, t_n) = \frac{P^n_{i-1} - 2P^n * *i + \frac{1}{2} P^n * *i + 1}{\Delta x^2} + \mathcal{O}(\Delta x^2),$$
$$\frac{\partial^2}{\partial x^2} P(\bar{x}_i, \bar{t}_n) \approx P^n_{i-1} - 2P^n * *i + \frac{1}{2} P^n * *i + 1\,.$$

Equation (3.89) is therefore equivalent with the dimensionless diffusion equation

$$\frac{\partial P}{\partial \bar{t}} = \frac{1}{2} \frac{\partial^2 P}{\partial \bar{x}^2}, \tag{3.90}$$

or the diffusion equation

$$\frac{\partial P}{\partial t} = D\frac{\partial^2 P}{\partial x^2}, \tag{3.91}$$

with diffusion coefficient

$$D = \frac{\Delta x^2}{2\Delta t}\,.$$

This derivation shows the tight link between random walk and diffusion. If we keep track of where the particle is, and repeat the process many times, or run the algorithms for lots of particles, the histogram of the positions will approximate the solution of the diffusion equation for the local probability P_i^n.

Suppose all the random walks start at the origin. Then the initial condition for the probability distribution is the Dirac delta function $\delta(x)$. The solution of (3.90) can be shown to be

$$\bar{P}(\bar{x},\bar{t}) = \frac{1}{\sqrt{4\pi\alpha t}}e^{-\frac{x^2}{4\alpha t}}, \tag{3.92}$$

where $\alpha = \frac{1}{2}$.

3.59. Implementation of multiple walks

Our next task is to implement an ensemble of walks (for statistics, see Section Section 3.56) and also provide data from the walks such that we can compute the probabilities of the positions as introduced in the previous section. An appropriate representation of probabilities P_i^n are histograms (with i along the x axis) for a few selected values of n.

To estimate the expectation and variance of the random walks, Section Section 3.56 points to recording $\sum_j x_{j,k}$ and $\sum_j x_{j,k}^2$, where $x_{j,k}$ is the position at time/step level k in random walk number j. The histogram of positions needs the individual values $x_{i,k}$ for all i values and some selected k values.

We introduce `position[k]` to hold $\sum_j x_{j,k}$, `position2[k]` to hold $\sum_j (x_{j,k})^2$, and `pos_hist[i,k]` to hold $x_{i,k}$. A selection of k values can be specified by saying how many, `num_times`, and let them be equally spaced through time:

```
pos_hist_times = [(N//num_times)*i for i in range(num_times)]
```

This is one of the few situations where we want integer division (`//`) or real division rounded to an integer.

3.59.1. Scalar version

Our scalar implementation of running `num_walks` random walks may go like this:

```
import random

import matplotlib.pyplot as plt
import numpy as np

random.seed(10)
np.random.seed(10)

def random_walk1D(x0, N, p, random=random):
    """1D random walk with 1 particle and N moves."""

    position = np.zeros(N + 1)
    position[0] = x0
    current_pos = x0
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= p:
            current_pos -= 1
        else:
            current_pos += 1
        position[k + 1] = current_pos
    return position

def random_walk1D_vec(x0, N, p):
    """Vectorized version of random_walk1D."""
    position = np.zeros(N + 1)
    position[0] = x0
    r = np.random.uniform(0, 1, size=N)
    steps = np.where(r <= p, -1, 1)
    position[1:] = x0 + np.cumsum(steps)
    return position

def test_random_walk1D():
    x0 = 2
    N = 4
    p = 0.6
    np.random.seed(10)
    scalar_computed = random_walk1D(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walk1D_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def demo_random_walk1D(N=50000):
    np.random.seed(10)
    pos = random_walk1D_vec(x0=0, N=N, p=0.5)
    plt.figure()
    plt.plot(pos)
```

```
    plt.savefig("tmp1.pdf")
    plt.savefig("tmp1.png")
    plt.figure()
    plt.plot(pos * pos)
    plt.savefig("tmp2.pdf")
    plt.savefig("tmp2.png")
    plt.show()

def demo_fig_random_walk1D(N=200):
    """Make ensamble of positions (to illustrate E[] operator)."""
    np.random.seed(10)
    num_plots = 4
    for n in range(num_plots):
        plt.subplot(num_plots, 1, n + 1)
        pos = random_walk1D_vec(x0=0, N=N, p=0.5)
        plt.plot(pos)
        plt.axis([0, N, -15, 20])
    plt.savefig("tmp.pdf")
    plt.savefig("tmp.png")
    plt.show()

def demo_random_walk1D_timing():
    import time

    x0 = 0
    N = 10000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walk1D(x0, N, p, random=np.random)
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos = random_walk1D_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def random_walks1D(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    position = np.zeros(N + 1)  # Accumulated positions
    position[0] = x0 * num_walks
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    position2[0] = x0**2 * num_walks
```

```
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
        current_pos = x0
        for k in range(N):
            if k in pos_hist_times:
                pos_hist[n, num_times_counter] = current_pos
                num_times_counter += 1
            r = random.uniform(0, 1)
            if r <= p:
                current_pos -= 1
            else:
                current_pos += 1
            position[k + 1] += current_pos
            position2[k + 1] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)
```

3.59.2. Vectorized version

We have already vectorized a single random walk. The additional challenge here is to vectorize the computation of the data for the histogram, `pos_hist`, but given the selected steps in `pos_hist_times`, we can find the corresponding positions by indexing with the list `pos_hist_times`: `position[post_hist_times]`, which are to be inserted in `pos_hist[n,:]`.

```
def random_walks1D_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walk = np.zeros(N + 1)  # Positions of current walk
    walk[0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        r = np.random.uniform(0, 1, size=N)
        steps = np.where(r <= p, -1, 1)
        walk[1:] = x0 + np.cumsum(steps)  # Positions of this walk
        position += walk
        position2 += walk**2
        pos_hist[n, :] = walk[pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)
```

3.59.3. Improved vectorized version

Looking at the vectorized version above, we still have one potentially long Python loop over `n`. Normally, `num_walks` will be much larger than `N`. The vectorization of the loop over `N` certainly speeds up the program, but if we think of vectorization as also a way to parallelize the code, all the independent walks (the `n` loop) can be executed in parallel. Therefore, we should include this loop as well in the vectorized expressions, at the expense of using more memory.

We introduce the array `walks` to hold the $N+1$ steps of all the walks: each row represents the steps in one walk.

```
walks = np.zeros((num_walks, N+1))  # Positions of each walk
walks[:,0] = x0
```

Since all the steps are independent, we can just make one long vector of enough random numbers (`N*num_walks`), translate these numbers to ± 1, then we reshape the array such that the steps of each walk are stored in the rows.

```
r = np.random.uniform(0, 1, size=N*num_walks)
steps = np.where(r <= p, -1, 1).reshape(num_walks, N)
```

The next step is to sum up the steps in each walk. We need the `np.cumsum` function for this, with the argument `axis=1` for indicating a sum across the columns:

```
>>> a = np.arrange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.cumsum(a, axis=1)
array([[ 0,  1,  3],
       [ 3,  7, 12]])
```

Now `walks` can be computed by

```
walks[:,1:] = x0 + np.cumsum(steps, axis=1)
```

The `position` vector is the sum of all the walks. That is, we want to sum all the rows, obtained by

```
position  = np.sum(walks, axis=0)
```

A corresponding expression computes the squares of the positions. Finally, we need to compute `pos_hist`, but that is a matter of grabbing some of the walks (according to `pos_hist_times`):

```
pos_hist[:,:] = walks[:,pos_hist_times]
```

The complete vectorized algorithm without any loop can now be summarized:

```
def random_walks1D_vec2(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1))  # Positions of each walk
    walks[:, 0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks)
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N)
    walks[:, 1:] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :] = walks[:, pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)
```

What is the gain of the vectorized implementations? One important gain is that each vectorized operation can be automatically parallelized if one applies a parallel `numpy` library like Numba. On a single CPU, however, the speed up of the vectorized operations is also significant. With $N = 1,000$ and 50,000 repeated walks, the two vectorized versions run about 25 and 18 times faster than the scalar version, with `random_walks1D_vec1` being fastest.

3.59.4. Remark on vectorized code and parallelization

Our first attempt on vectorization removed the loop over the N steps in a single walk. However, the number of walks is usually much larger than N, because of the need for accurate statistics. Therefore, we should rather remove the loop over all walks. It turns out, from our efficiency experiments, that the function `random_walks1D_vec2` (with no loops) is slower than `random_walks1D_vec1`. This is a bit surprising and may be explained by less efficiency in the statements involving very large arrays, containing all steps for all walks at once.

From a parallelization and improved vectorization point of view, it would be more natural to switch the sequence of the loops in the serial code such that the shortest loop is the outer loop:

```
def random_walks1D2(x0, N, p, num_walks=1, num_times=1, ...):
    ...
    current_pos = x0 + np.zeros(num_walks)
    num_times_counter = -1

    for k in range(N):
        if k in pos_hist_times:
```

```
        num_times_counter += 1
        store_hist = True
    else:
        store_hist = False

        for n in range(num_walks):
            r = random.uniform(0, 1)
        if r <= p:
                current_pos[n] -= 1
            else:
                current_pos[n] += 1
            position [k+1] += current_pos[n]
            position2[k+1] += current_pos[n]**2
            if store_hist:
                pos_hist[n,num_times_counter] = current_pos[n]
    return position, position2, pos_hist, np.array(pos_hist_times)
```

The vectorized version of this code, where we just vectorize the loop over `n`, becomes

```
def random_walks1D2_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D2."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    current_pos = np.zeros(num_walks)
    current_pos[0] = x0
    num_times_counter = -1

    for k in range(N):
        if k in pos_hist_times:
            num_times_counter += 1
            store_hist = True  # Store histogram data for this k
        else:
            store_hist = False

        r = np.random.uniform(0, 1, size=num_walks)
        steps = np.where(r <= p, -1, 1)
        current_pos += steps
        position[k + 1] = np.sum(current_pos)
        position2[k + 1] = np.sum(current_pos**2)
        if store_hist:
            pos_hist[:, num_times_counter] = current_pos
    return position, position2, pos_hist, np.array(pos_hist_times)
```

This function runs significantly faster than the `random_walks1D_vec1` function above, typically

1.7 times faster. The code is also more appropriate in a parallel computing context since each vectorized statement can work with data of size `num_walks` over the compute units, repeated `N` times (compared with data of size `N`, repeated `num_walks` times, in `random_walks1D_vec1`).

The scalar code with switched loops, `random_walks1D2` runs a bit slower than the original code in `random_walks1D`, so with the longest loop as the inner loop, the vectorized function `random_walks1D2_vec1` is almost 60 times faster than the scalar counterpart, while the code `random_walks1D_vec2` without loops is only around 18 times faster. Taking into account the very large arrays required by the latter function, we end up with `random_walks1D2_vec1` as the preferred implementation.

3.59.5. Test function

During program development, it is highly recommended to carry out computations by hand for, e.g., `N=4` and `num_walks=3`. Normally, this is done by executing the program with these parameters and checking with pen and paper that the computations make sense. The next step is to use this test for correctness in a formal test function.

First, we need to check that the simulation of multiple random walks reproduces the results of `random_walk1D`, `random_walk1D_vec1`, and `random_walk1D_vec2` for the first walk, if the seed is the same. Second, we run three random walks (`N=4`) with the scalar and the two vectorized versions and check that the returned arrays are identical.

For this type of test to be successful, we must be sure that exactly the same set of random numbers are used in the three versions, a fact that requires the same random number generator and the same seed, of course, but also the same sequence of computations. This is not obviously the case with the three `random_walk1D*` functions we have presented. The critical issue in `random_walk1D_vec1` is that the first random numbers are used for the first walk, the second set of random numbers is used for the second walk and so on, to be compatible with how the random numbers are used in the function `random_walk1D`. For the function `random_walk1D_vec2` the situation is a bit more complicated since we generate all the random numbers at once. However, the critical step now is the reshaping of the array returned from `np.where`: we must reshape as `(num_walks, N)` to ensure that the first `N` random numbers are used for the first walk, the next `N` numbers are used for the second walk, and so on.

We arrive at the test function below.

```
def test_random_walks1D():
    x0 = 0
    N = 4
    p = 0.5

    num_walks = 1
    np.random.seed(10)
    computed = random_walks1D(x0, N, p, num_walks, random=np.random)
    np.random.seed(10)
    expected = random_walk1D(x0, N, p, random=np.random)
    assert (computed[0] == expected).all()
```

```
    np.random.seed(10)
    computed = random_walks1D_vec1(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()
    np.random.seed(10)
    computed = random_walks1D_vec2(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()

    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walks1D(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized1_computed = random_walks1D_vec1(x0, N, p, num_walks, num_times)
    np.random.seed(10)
    vectorized2_computed = random_walks1D_vec2(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized1_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg
    for s, v, r in zip(serial_computed, vectorized2_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg
```

Such test functions are indispensable for further development of the code as we can at any time test whether the basic computations remain correct or not. This is particularly important in stochastic simulations since without test functions and fixed seeds, we always experience variations from run to run, and it can be very difficult to spot bugs through averaged statistical quantities.

3.60. Demonstration of multiple walks

Assuming now that the code works, we can just scale up the number of steps in each walk and the number of walks. The latter influences the accuracy of the statistical estimates. Figure Figure 3.18 shows the impact of the number of walks on the expectation, which should approach zero. Figure Figure 3.19 displays the corresponding estimate of the variance of the position, which should grow linearly with the number of steps. It does, seemingly very accurately, but notice that the scale on the y axis is so much larger than for the expectation, so irregularities due to the stochastic nature of the process become so much less visible in the variance plots. The probability of finding a particle at a certain position at time (or step) 800 is shown in Figure Figure 3.20. The dashed red line is the theoretical distribution (3.92) arising from solving the diffusion equation (3.90) instead. As always, we realize that one needs significantly more statistical samples to estimate a histogram accurately than the expectation or variance.

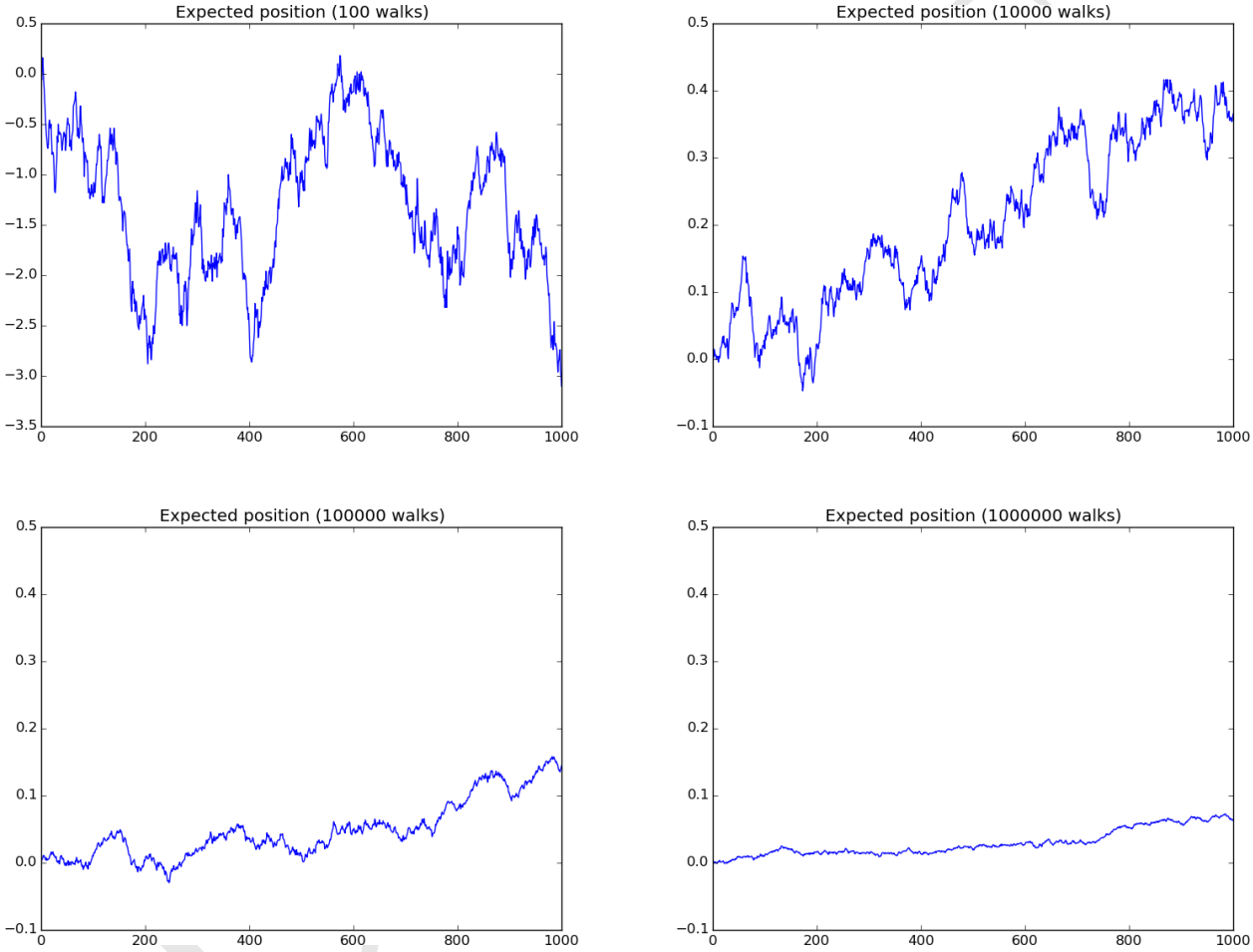


Figure 3.18.: Estimated expected value for 1000 steps, using 100 walks (upper left), 10,000 (upper right), 100,000 (lower left), and 1,000,000 (lower right).

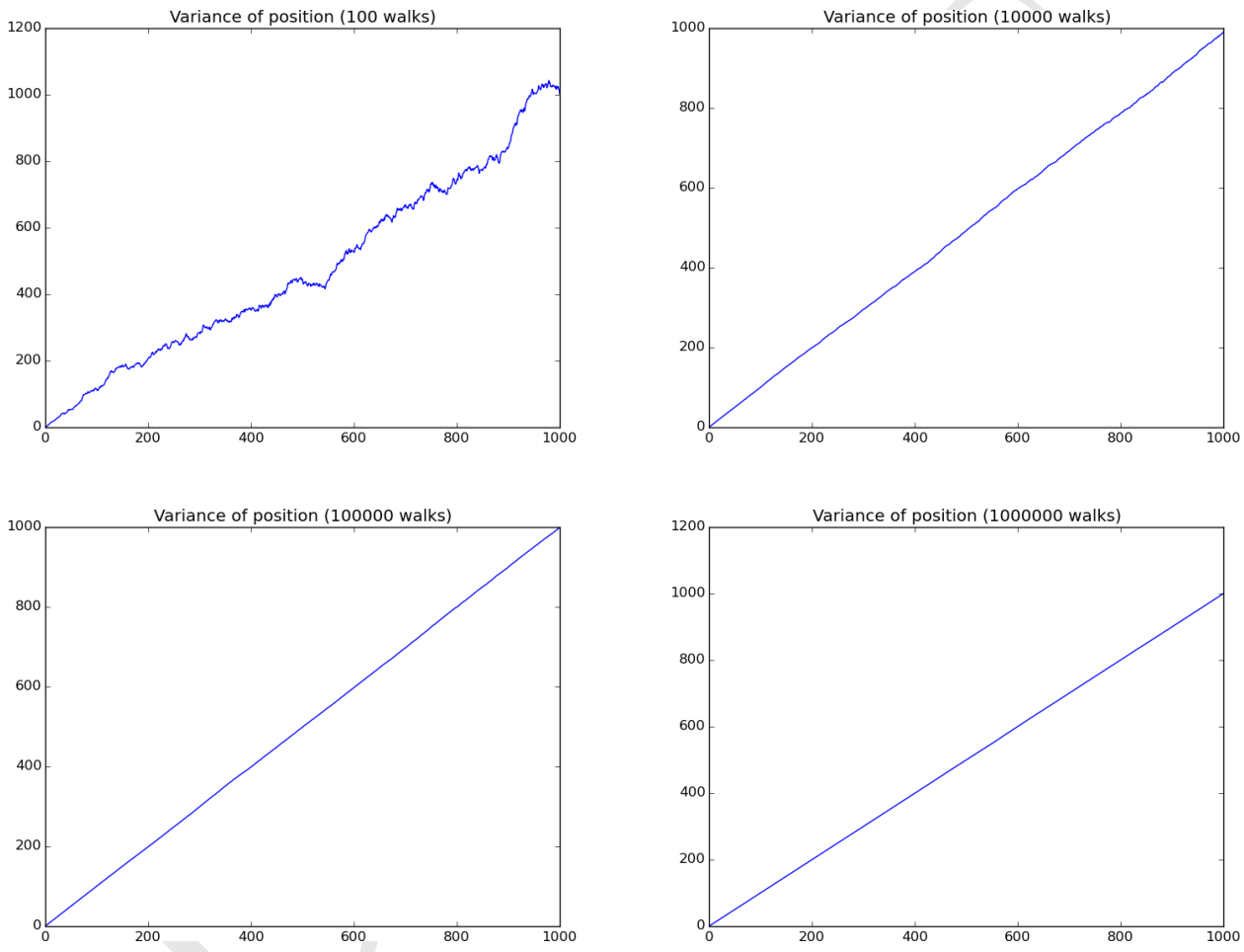


Figure 3.19.: Estimated variance over 1000 steps, using 100 walks (upper left), 10,000 (upper right), 100,000 (lower left), and 1,000,000 (lower right).

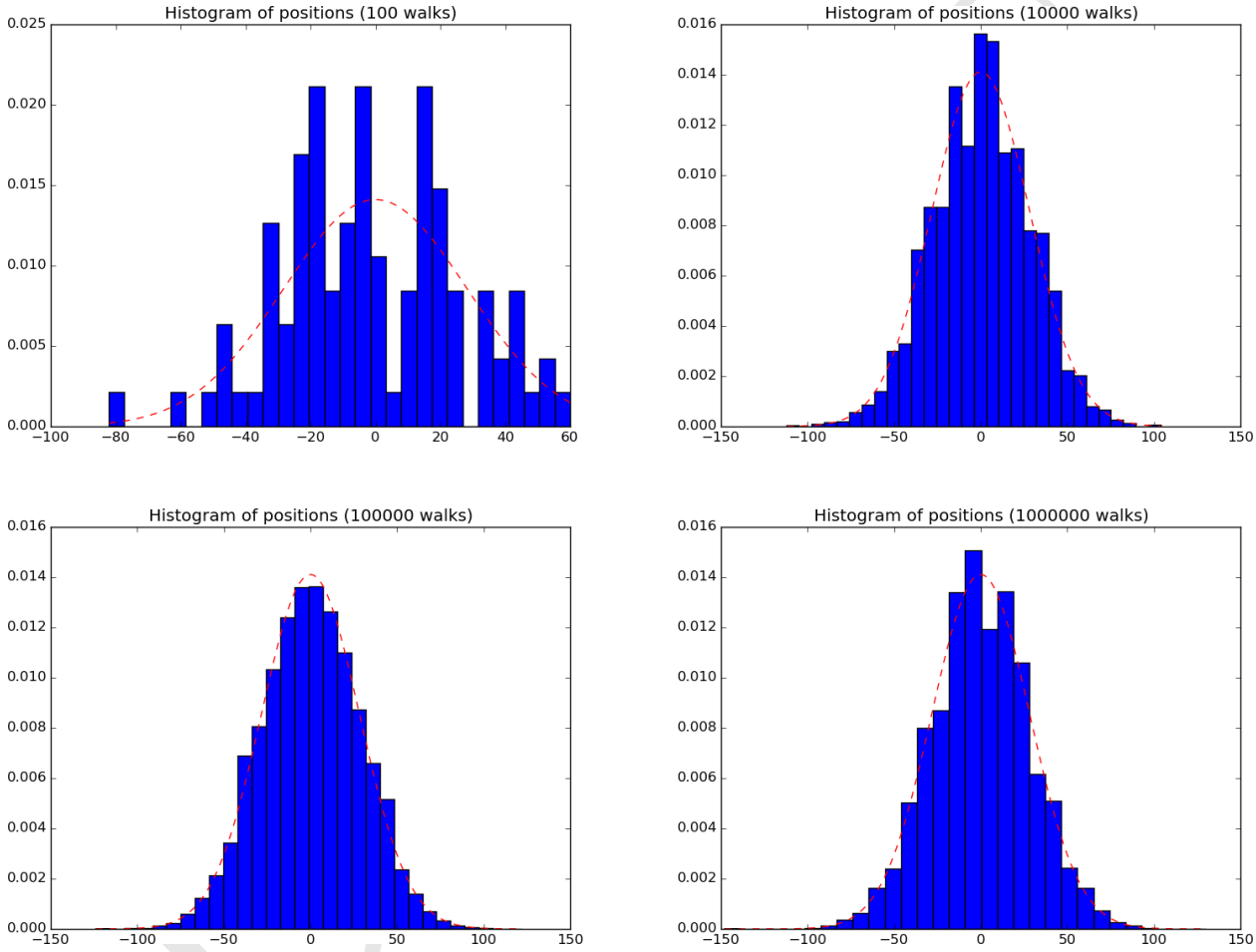


Figure 3.20.: Estimated probability distribution at step 800, using 100 walks (upper left), 10,000 (upper right), 100,000 (lower left), and 1,000,000 (lower right).

3.61. Empty figure cache

3.62. Random walk as a stochastic equation

The (dimensionless) position in a random walk, $\bar{X}_k$, can be expressed as a stochastic difference equation:

$$\bar{X}_k = \bar{X}_{k-1} + s, \quad x_0 = 0, \tag{3.93}$$

where s is a Bernoulli variable, taking on the two values $s = -1$ and $s = 1$ with equal probability:

$$\mathrm{P}(s = 1) = \frac{1}{2}, \quad \mathrm{P}(s = -1) = \frac{1}{2}\,.$$

The s variable in a step is independent of the s variable in other steps.

The difference equation expresses essentially the sum of independent Bernoulli variables. Because of the central limit theorem, X_k, will then be normally distributed with expectation $k\mathrm{E}[s]$ and $k\mathrm{Var}[s]$. The expectation and variance of a Bernoulli variable with values $r = 0$ and $r = 1$ are p and $p(1-p)$, respectively. The variable $s = 2r - 1$ then has expectation $2\mathrm{E}[r] - 1 = 2p - 1 = 0$ and variance $2^2\mathrm{Var}[r] = 4p(1-p) = 1$. The position X_k is normally distributed with zero expectation and variance k, as we found in Section Section 3.56.

The central limit theorem tells that as long as k is not small, the distribution of X_k remains the same if we replace the Bernoulli variable s by any other stochastic variable with the same expectation and variance. In particular, we may let s be a standardized Gaussian variable (zero mean, unit variance).

Dividing (3.93) by Δt gives

$$\frac{\bar{X}_k - \bar{X}_{k-1}}{\Delta t} = \frac{1}{\Delta t} s\,.$$

In the limit $\Delta t \to 0$, $s/\Delta t$ approaches a white noise stochastic process. With $\bar{X}(t)$ as the continuous process in the limit $\Delta t \to 0$ ($X_k \to X(t_k)$), we formally get the stochastic differential equation

$$d\bar{X} = dW,$$

where $W(t)$ is a Wiener process. Then X is also a Wiener process. It follows from the stochastic ODE $dX = dW$ that the probability distribution of X is given by the Fokker-Planck equation (3.90). In other words, the key results for random walk we found earlier can alternatively be derived via a stochastic ordinary differential equation and its related Fokker-Planck equation.

3.63. Random walk in 2D

The most obvious generalization of 1D random walk to two spatial dimensions is to allow movements to the north, east, south, and west, with equal probability $\frac{1}{4}$.

```
def random_walk2D(x0, N, p, random=random):
    """2D random walk with 1 particle and N moves: N, E, W, S."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
```

```
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= 0.25:
            current_pos += np.array([0, 1])  # Move north
        elif 0.25 < r <= 0.5:
            current_pos += np.array([1, 0])  # Move east
        elif 0.5 < r <= 0.75:
            current_pos += np.array([0, -1])  # Move south
        else:
            current_pos += np.array([-1, 0])  # Move west
        position[k + 1, :] = current_pos
    return position
```

The left plot in Figure Figure 3.21 provides an example on 200 steps with this kind of walk. We may refer to this walk as a walk on a *rectangular mesh* as we move from any spatial mesh point (i, j) to one of its four neighbors in the rectangular directions: $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, or $(i, j-1)$.

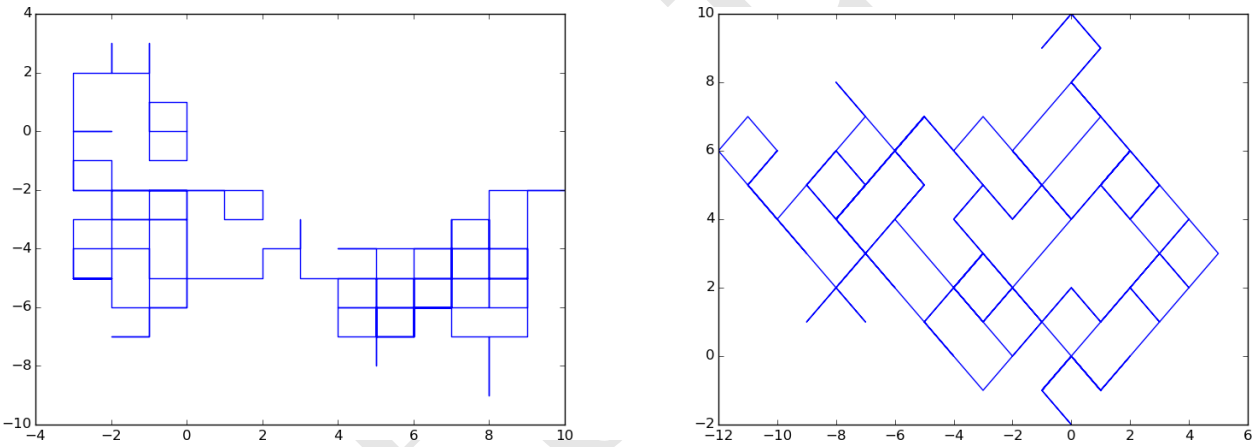


Figure 3.21.: Random walks in 2D with 200 steps: rectangular mesh (left) and diagonal mesh (right).

3.64. Random walk in any number of space dimensions

From a programming point of view, especially when implementing a random walk in any number of dimensions, it is more natural to consider a walk in the diagonal directions NW, NE, SW, and SE. On a two-dimensional spatial mesh it means that we go from (i, j) to either $(i+1, j+1)$, $(i-1, j+1)$, $(i+1, j-1)$, or $(i-1, j-1)$. We can with such a *diagonal mesh* (see right plot in Figure Figure 3.21) draw a Bernoulli variable for the step in each spatial direction and trivially write code that works in any number of spatial directions:

```
import random

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

```
random.seed(10)
np.random.seed(10)

def random_walk1D(x0, N, p, random=random):
    """1D random walk with 1 particle and N moves."""

    position = np.zeros(N + 1)
    position[0] = x0
    current_pos = x0
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= p:
            current_pos -= 1
        else:
            current_pos += 1
        position[k + 1] = current_pos
    return position

def random_walk1D_vec(x0, N, p):
    """Vectorized version of random_walk1D."""
    position = np.zeros(N + 1)
    position[0] = x0
    r = np.random.uniform(0, 1, size=N)
    steps = np.where(r <= p, -1, 1)
    position[1:] = x0 + np.cumsum(steps)
    return position

def test_random_walk1D():
    x0 = 2
    N = 4
    p = 0.6
    np.random.seed(10)
    scalar_computed = random_walk1D(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walk1D_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def demo_random_walk1D(N=50000):
    np.random.seed(10)
    pos = random_walk1D_vec(x0=0, N=N, p=0.5)
    plt.figure()
    plt.plot(pos)
    plt.savefig("tmp1.pdf")
    plt.savefig("tmp1.png")
    plt.figure()
    plt.plot(pos * pos)
    plt.savefig("tmp2.pdf")
```

```
    plt.savefig("tmp2.png")
    plt.show()

def demo_fig_random_walk1D(N=200):
    """Make ensamble of positions (to illustrate E[] operator)."""
    np.random.seed(10)
    num_plots = 4
    for n in range(num_plots):
        plt.subplot(num_plots, 1, n + 1)
        pos = random_walk1D_vec(x0=0, N=N, p=0.5)
        plt.plot(pos)
        plt.axis([0, N, -15, 20])
    plt.savefig("tmp.pdf")
    plt.savefig("tmp.png")
    plt.show()

def demo_random_walk1D_timing():
    import time

    x0 = 0
    N = 10000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walk1D(x0, N, p, random=np.random)
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos = random_walk1D_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def random_walks1D(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    position = np.zeros(N + 1)  # Accumulated positions
    position[0] = x0 * num_walks
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    position2[0] = x0**2 * num_walks
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
```

```
        current_pos = x0
        for k in range(N):
            if k in pos_hist_times:
                pos_hist[n, num_times_counter] = current_pos
                num_times_counter += 1
            r = random.uniform(0, 1)
            if r <= p:
                current_pos -= 1
            else:
                current_pos += 1
            position[k + 1] += current_pos
            position2[k + 1] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walk = np.zeros(N + 1)  # Positions of current walk
    walk[0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        r = np.random.uniform(0, 1, size=N)
        steps = np.where(r <= p, -1, 1)
        walk[1:] = x0 + np.cumsum(steps)  # Positions of this walk
        position += walk
        position2 += walk**2
        pos_hist[n, :] = walk[pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec2(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1))  # Positions of each walk
    walks[:, 0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks)
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N)
    walks[:, 1:] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :] = walks[:, pos_hist_times]
```

```
    return position, position2, pos_hist, np.array(pos_hist_times)

def test_random_walks1D():
    x0 = 0
    N = 4
    p = 0.5

    num_walks = 1
    np.random.seed(10)
    computed = random_walks1D(x0, N, p, num_walks, random=np.random)
    np.random.seed(10)
    expected = random_walk1D(x0, N, p, random=np.random)
    assert (computed[0] == expected).all()

    np.random.seed(10)
    computed = random_walks1D_vec1(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()
    np.random.seed(10)
    computed = random_walks1D_vec2(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()

    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walks1D(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized1_computed = random_walks1D_vec1(x0, N, p, num_walks, num_times)
    np.random.seed(10)
    vectorized2_computed = random_walks1D_vec2(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized1_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg
    for s, v, r in zip(serial_computed, vectorized2_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg

def demo_random_walks1D(N=1000, num_walks=10000, EX_minmax=None):
    import time

    t0 = time.perf_counter()
    pos, pos2, hist, hist_times = random_walks1D_vec1(
        x0=0,
```

```
        N=N,
        p=0.5,
        num_walks=num_walks,
        num_times=10,
    )
    t1 = time.perf_counter()
    print("histogram times:", hist_times)
    print("random walk: %.1fs" % (t1 - t0))
    E_X = pos / float(num_walks)
    Var_X = pos2 / float(num_walks) - E_X**2
    if N <= 50:
        print(pos)

    plt.figure()
    plt.plot(E_X)
    if EX_minmax is not None:
        plt.axis([0, N, EX_minmax[0], EX_minmax[1]])
    plt.title("Expected position (%d walks)" % num_walks)
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.figure()
    plt.plot(Var_X)
    plt.title("Variance of position (%d walks)" % num_walks)
    plt.savefig("tmp2.png")
    plt.savefig("tmp2.pdf")

    plt.figure()
    a = 0.5
    exact = (
        lambda x, t: 1.0 / np.sqrt(4 * np.pi * t * a) * np.exp(-(x**2) / (4.0 * t * a))
    )
    hist_time_index = -2
    n, bins, patches = plt.hist(hist[:, hist_time_index], bins=30, normed=True)
    x = np.linspace(bins[0], bins[-1], 301)
    t = hist_times[hist_time_index]
    plt.plot(x, exact(x, t), "r--")
    plt.title("Histogram of positions (%d walks)" % num_walks)
    plt.savefig("tmp3.png")
    plt.savefig("tmp3.pdf")
    plt.show()

def demo_fig_random_walks1D():
    """Make figures with statistics and dependence on no of walks."""
    import os
    import shutil

    N = 1000
```

```
    num_walks = [100, 10000, 100000, 1000000]
    for n in num_walks:
        np.random.seed(10)  # Use same seq. for all experiments
        if n == 100:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=None)
        else:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=[-0.1, 0.5])
        d = "tmp_%d" % n
        if os.path.isdir(d):
            shutil.rmtree(d)
        os.mkdir(d)
        for p in 1, 2, 3:
            os.rename("tmp%d.png" % p, os.path.join(d, "tmp%d.png" % p))
            os.rename("tmp%d.pdf" % p, os.path.join(d, "tmp%d.pdf" % p))
    plots = ["EX", "VarX", "HistX"]
    for j, plot in enumerate(plots):
        for ext in "png", "pdf":
            files = [
                os.path.join("tmp_%d" % n, "tmp%d.%s" % (j + 1, ext)) for n in num_walks
            ]
            ncols = 3 if len(num_walks) == 3 else 2
            output = "rw1D_%s_%s.%s" % (plot, "_".join([str(n) for n in num_walks]), ext)
            cmd = "montage %s -tile %dx1 -geometry +0+0 %s" % (
                " ".join(files),
                ncols,
                output,
            )
            print(cmd)
            os.system(cmd)

def demo_random_walks1D_timing():
    import time

    x0 = 0
    N = 1000
    num_walks = 50000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D(
        x0, N, p, num_walks, num_times=4, random=np.random
    )
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
```

```
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec1(
        x0, N, p, num_walks, num_times=4
    )
    t2 = time.perf_counter()
    cpu_vec1 = t2 - t1
    print("CPU vectorized1: %.1f" % cpu_vec1)
    print("CPU scalar/vectorized1: %.1f" % (cpu_scalar / cpu_vec1))
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec2(
        x0, N, p, num_walks, num_times=4
    )
    t3 = time.perf_counter()
    cpu_vec2 = t3 - t2
    print("CPU vectorized2: %.1f" % cpu_vec2)
    print("CPU scalar/vectorized2: %.1f" % (cpu_scalar / cpu_vec2))

def random_walk2D(x0, N, p, random=random):
    """2D random walk with 1 particle and N moves: N, E, W, S."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= 0.25:
            current_pos += np.array([0, 1])  # Move north
        elif 0.25 < r <= 0.5:
            current_pos += np.array([1, 0])  # Move east
        elif 0.5 < r <= 0.75:
            current_pos += np.array([0, -1])  # Move south
        else:
            current_pos += np.array([-1, 0])  # Move west
        position[k + 1, :] = current_pos
    return position

def demo_random_walk2D():
    x0 = (0, 0)
    N = 200
    p = 0.5
    np.random.seed(10)
    pos = random_walk2D(x0, N, p, random=np.random)
    plt.plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def random_walkdD(x0, N, p, random=random):
```

```
    """Any-D (diagonal) random walk with 1 particle and N moves."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        for i in range(d):
            r = random.uniform(0, 1)
            if r <= p:
                current_pos[i] -= 1
            else:
                current_pos[i] += 1
        position[k + 1, :] = current_pos
    return position

def random_walkD_vec(x0, N, p):
    """Vectorized version of random_walkD."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0] = np.array(x0, dtype=float)
    r = np.random.uniform(0, 1, size=N * d)
    steps = np.where(r <= p, -1, 1).reshape(N, d)
    position[1:, :] = x0 + np.cumsum(steps, axis=0)
    return position

def demo_random_walkD():
    x0 = (0, 0)
    N = 200
    p = 0.5
    np.random.seed(10)
    pos = random_walkD(x0, N, p, random=np.random)
    plt.plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def demo_random_walkD_timing():
    import time

    x0 = (0, 0)
    N = 4000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walkD(x0, N, p, random=np.random)
    t1 = time.perf_counter()
```

```
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos = random_walkdD_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def demo_fig_random_walkdD():
    x0 = (0, 0)
    N = 5000
    p = 0.5
    n = 2  # nxn subplots
    f, axarr = plt.subplots(n, n, sharex=True, sharey=True)
    for i in range(n):
        for j in range(n):
            seed = 3 * i + 8 * j
            np.random.seed(seed)
            pos = random_walkdD(x0, N, p, random=np.random)
            axarr[i, j].plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def test_ramdom_walkdD():
    x0 = (0, 0)
    N = 7
    p = 0.5
    np.random.seed(10)
    scalar_computed = random_walkdD(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walkdD_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def random_walksdD(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    d = len(x0)
    position = np.zeros((N + 1, d))  # Accumulated positions
    position2 = np.zeros((N + 1, d))  # Accumulated positions**2
    pos_hist = np.zeros((num_walks, num_times, d))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
        current_pos = np.array(x0, dtype=float)
        for k in range(N):
```

```
            if k in pos_hist_times:
                pos_hist[n, num_times_counter, :] = current_pos
                num_times_counter += 1
            for i in range(d):
                r = random.uniform(0, 1)
                if r <= p:
                    current_pos[i] -= 1
                else:
                    current_pos[i] += 1
            position[k + 1, :] += current_pos
            position2[k + 1, :] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walksdD_vec(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    d = len(x0)
    position = np.zeros((N + 1, d))  # Accumulated positions
    position2 = np.zeros((N + 1, d))  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1, d))  # Positions of each walk
    walks[:, 0, :] = x0
    pos_hist = np.zeros((num_walks, num_times, d))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks * d)
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N, d)
    walks[:, 1:, :] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :, :] = walks[:, pos_hist_times, :]
    return position, position2, pos_hist, np.array(pos_hist_times)

def test_random_walksdD():
    x0 = (0, 0)
    N = 4
    p = 0.5

    num_walks = 1
    np.random.seed(10)
    computed = random_walksdD(x0, N, p, num_walks, random=np.random)
    np.random.seed(10)
    expected = random_walkdD(x0, N, p, random=np.random)
    assert (computed[0] == expected).all()

    np.random.seed(10)
    computed = random_walksdD_vec(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walkdD_vec(x0, N, p)
```

```
    assert (computed[0] == expected).all()

    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walksdD(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walksdD_vec(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized_computed, return_values):
        msg = "%s: %s\n%s (serial)\nvs\n%s\n%s (vectorized)" % (r, s.shape, s, v.shape, v)
        assert (s == v).all(), msg

def demo_random_walksdD():
    x0 = (0, 0)
    N = 1000
    num_walks = 1000
    p = 0.5
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walksdD(
        x0, N, p, num_walks, num_times=4, random=np.random
    )
    print(pos_hist_times)
    plt.figure()
    plt.plot(pos[:, 0], pos[:, 1])
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walksdD_vec(
        x0, N, p, num_walks, num_times=4
    )
    plt.figure()
    plt.plot(pos[:, 0], pos[:, 1])

    plt.show()

def demo_random_walksdD_timing():
    import time

    x0 = (0, 0, 0)
    N = 1000
    num_walks = 10000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walksdD(
        x0, N, p, num_walks, num_times=4, random=np.random
    )
```

```
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walksdD_vec(
        x0, N, p, num_walks, num_times=4
    )
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def random_walks1D2(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    position = np.zeros(N + 1)  # Accumulated positions
    position[0] = x0 * num_walks
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    position2[0] = x0**2 * num_walks
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    current_pos = x0 + np.zeros(num_walks)
    num_times_counter = -1

    for k in range(N):
        if k in pos_hist_times:
            num_times_counter += 1
            store_hist = True  # Store histogram data for this k
        else:
            store_hist = False

        for n in range(num_walks):
            r = random.uniform(0, 1)
            if r <= p:
                current_pos[n] -= 1
            else:
                current_pos[n] += 1
            position[k + 1] += current_pos[n]
            position2[k + 1] += current_pos[n] ** 2
            if store_hist:
                pos_hist[n, num_times_counter] = current_pos[n]
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D2_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D2."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
```

```
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    current_pos = np.zeros(num_walks)
    current_pos[0] = x0
    num_times_counter = -1

    for k in range(N):
        if k in pos_hist_times:
            num_times_counter += 1
            store_hist = True  # Store histogram data for this k
        else:
            store_hist = False

        r = np.random.uniform(0, 1, size=num_walks)
        steps = np.where(r <= p, -1, 1)
        current_pos += steps
        position[k + 1] = np.sum(current_pos)
        position2[k + 1] = np.sum(current_pos**2)
        if store_hist:
            pos_hist[:, num_times_counter] = current_pos
    return position, position2, pos_hist, np.array(pos_hist_times)

def test_random_walks1D2():
    x0 = 0
    N = 4
    p = 0.5
    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walks1D2(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walks1D2_vec1(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg

def demo_random_walks1D2_timing():
    """Timing of random 1D walks with reversed loops."""
    import time

    x0 = 0
    N = 1000
    num_walks = 50000
    p = 0.5
```

```
    t0 = time.perf_counter()
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D2(
        x0, N, p, num_walks, num_times=4, random=np.random
    )
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D2_vec1(
        x0, N, p, num_walks, num_times=4
    )
    t2 = time.perf_counter()
    cpu_vec1 = t2 - t1
    print("CPU vectorized1: %.1f" % cpu_vec1)
    print("CPU scalar/vectorized1: %.1f" % (cpu_scalar / cpu_vec1))
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec2(
        x0, N, p, num_walks, num_times=4
    )
    t3 = time.perf_counter()
    cpu_vec2 = t3 - t2
    print("CPU vectorized2: %.1f" % cpu_vec2)
    print("CPU scalar/vectorized2: %.1f" % (cpu_scalar / cpu_vec2))

if __name__ == "__main__":
    demo_random_walks1D2_timing()
    print("----")
    demo_random_walks1D_timing()
```

A vectorized version is desired. We follow the ideas from Section Section 3.57.1, but each step is now a vector in d spatial dimensions. We therefore need to draw *And* random numbers in `r`, compute steps in the various directions through `np.where(r <=p, -1, 1)` (each step being -1 or 1), and then we can reshape this array to an $N \times d$ array of step *vectors*. Doing an `np.cumsum` summation along axis 0 will add the vectors, as this demo shows:

```
>>> a = np.arrange(6).reshape(3,2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> np.cumsum(a, axis=0)
array([[ 0,  1],
       [ 2,  4],
       [ 6,  9]])
```

With such summation of step vectors, we get all the positions to be filled in the `position` array:

```
import random

import matplotlib.pyplot as plt
import numpy as np

random.seed(10)
np.random.seed(10)

def random_walk1D(x0, N, p, random=random):
    """1D random walk with 1 particle and N moves."""

    position = np.zeros(N + 1)
    position[0] = x0
    current_pos = x0
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= p:
            current_pos -= 1
        else:
            current_pos += 1
        position[k + 1] = current_pos
    return position

def random_walk1D_vec(x0, N, p):
    """Vectorized version of random_walk1D."""
    position = np.zeros(N + 1)
    position[0] = x0
    r = np.random.uniform(0, 1, size=N)
    steps = np.where(r <= p, -1, 1)
    position[1:] = x0 + np.cumsum(steps)
    return position

def test_random_walk1D():
    x0 = 2
    N = 4
    p = 0.6
    np.random.seed(10)
    scalar_computed = random_walk1D(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walk1D_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def demo_random_walk1D(N=50000):
    np.random.seed(10)
    pos = random_walk1D_vec(x0=0, N=N, p=0.5)
    plt.figure()
    plt.plot(pos)
```

```
    plt.savefig("tmp1.pdf")
    plt.savefig("tmp1.png")
    plt.figure()
    plt.plot(pos * pos)
    plt.savefig("tmp2.pdf")
    plt.savefig("tmp2.png")
    plt.show()

def demo_fig_random_walk1D(N=200):
    """Make ensamble of positions (to illustrate E[] operator)."""
    np.random.seed(10)
    num_plots = 4
    for n in range(num_plots):
        plt.subplot(num_plots, 1, n + 1)
        pos = random_walk1D_vec(x0=0, N=N, p=0.5)
        plt.plot(pos)
        plt.axis([0, N, -15, 20])
    plt.savefig("tmp.pdf")
    plt.savefig("tmp.png")
    plt.show()

def demo_random_walk1D_timing():
    import time

    x0 = 0
    N = 10000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walk1D(x0, N, p, random=np.random)
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos = random_walk1D_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def random_walks1D(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    position = np.zeros(N + 1)  # Accumulated positions
    position[0] = x0 * num_walks
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    position2[0] = x0**2 * num_walks
```

```
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
        current_pos = x0
        for k in range(N):
            if k in pos_hist_times:
                pos_hist[n, num_times_counter] = current_pos
                num_times_counter += 1
            r = random.uniform(0, 1)
            if r <= p:
                current_pos -= 1
            else:
                current_pos += 1
            position[k + 1] += current_pos
            position2[k + 1] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walk = np.zeros(N + 1)  # Positions of current walk
    walk[0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        r = np.random.uniform(0, 1, size=N)
        steps = np.where(r <= p, -1, 1)
        walk[1:] = x0 + np.cumsum(steps)  # Positions of this walk
        position += walk
        position2 += walk**2
        pos_hist[n, :] = walk[pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec2(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1))  # Positions of each walk
    walks[:, 0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks)
```

```
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N)
    walks[:, 1:] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :] = walks[:, pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)

def test_random_walks1D():
    x0 = 0
    N = 4
    p = 0.5

    num_walks = 1
    np.random.seed(10)
    computed = random_walks1D(x0, N, p, num_walks, random=np.random)
    np.random.seed(10)
    expected = random_walk1D(x0, N, p, random=np.random)
    assert (computed[0] == expected).all()

    np.random.seed(10)
    computed = random_walks1D_vec1(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()
    np.random.seed(10)
    computed = random_walks1D_vec2(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()

    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walks1D(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized1_computed = random_walks1D_vec1(x0, N, p, num_walks, num_times)
    np.random.seed(10)
    vectorized2_computed = random_walks1D_vec2(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized1_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg
    for s, v, r in zip(serial_computed, vectorized2_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg

def demo_random_walks1D(N=1000, num_walks=10000, EX_minmax=None):
```

```
    import time

    t0 = time.perf_counter()
    pos, pos2, hist, hist_times = random_walks1D_vec1(
        x0=0,
        N=N,
        p=0.5,
        num_walks=num_walks,
        num_times=10,
    )
    t1 = time.perf_counter()
    print("histogram times:", hist_times)
    print("random walk: %.1fs" % (t1 - t0))
    E_X = pos / float(num_walks)
    Var_X = pos2 / float(num_walks) - E_X**2
    if N <= 50:
        print(pos)

    plt.figure()
    plt.plot(E_X)
    if EX_minmax is not None:
        plt.axis([0, N, EX_minmax[0], EX_minmax[1]])
    plt.title("Expected position (%d walks)" % num_walks)
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.figure()
    plt.plot(Var_X)
    plt.title("Variance of position (%d walks)" % num_walks)
    plt.savefig("tmp2.png")
    plt.savefig("tmp2.pdf")

    plt.figure()
    a = 0.5
    exact = (
        lambda x, t: 1.0 / np.sqrt(4 * np.pi * t * a) * np.exp(-(x**2) / (4.0 * t * a))
    )
    hist_time_index = -2
    n, bins, patches = plt.hist(hist[:, hist_time_index], bins=30, normed=True)
    x = np.linspace(bins[0], bins[-1], 301)
    t = hist_times[hist_time_index]
    plt.plot(x, exact(x, t), "r--")
    plt.title("Histogram of positions (%d walks)" % num_walks)
    plt.savefig("tmp3.png")
    plt.savefig("tmp3.pdf")
    plt.show()

def demo_fig_random_walks1D():
```

```
    """Make figures with statistics and dependence on no of walks."""
    import os
    import shutil

    N = 1000
    num_walks = [100, 10000, 100000, 1000000]
    for n in num_walks:
        np.random.seed(10)  # Use same seq. for all experiments
        if n == 100:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=None)
        else:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=[-0.1, 0.5])
        d = "tmp_%d" % n
        if os.path.isdir(d):
            shutil.rmtree(d)
        os.mkdir(d)
        for p in 1, 2, 3:
            os.rename("tmp%d.png" % p, os.path.join(d, "tmp%d.png" % p))
            os.rename("tmp%d.pdf" % p, os.path.join(d, "tmp%d.pdf" % p))
    plots = ["EX", "VarX", "HistX"]
    for j, plot in enumerate(plots):
        for ext in "png", "pdf":
            files = [
                os.path.join("tmp_%d" % n, "tmp%d.%s" % (j + 1, ext)) for n in num_walks
            ]
            ncols = 3 if len(num_walks) == 3 else 2
            output = "rw1D_%s_%s.%s" % (plot, "_".join([str(n) for n in num_walks]), ext)
            cmd = "montage %s -tile %dx1 -geometry +0+0 %s" % (
                " ".join(files),
                ncols,
                output,
            )
            print(cmd)
            os.system(cmd)

def demo_random_walks1D_timing():
    import time

    x0 = 0
    N = 1000
    num_walks = 50000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D(
        x0, N, p, num_walks, num_times=4, random=np.random
```

```
    )
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec1(
        x0, N, p, num_walks, num_times=4
    )
    t2 = time.perf_counter()
    cpu_vec1 = t2 - t1
    print("CPU vectorized1: %.1f" % cpu_vec1)
    print("CPU scalar/vectorized1: %.1f" % (cpu_scalar / cpu_vec1))
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec2(
        x0, N, p, num_walks, num_times=4
    )
    t3 = time.perf_counter()
    cpu_vec2 = t3 - t2
    print("CPU vectorized2: %.1f" % cpu_vec2)
    print("CPU scalar/vectorized2: %.1f" % (cpu_scalar / cpu_vec2))

def random_walk2D(x0, N, p, random=random):
    """2D random walk with 1 particle and N moves: N, E, W, S."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= 0.25:
            current_pos += np.array([0, 1])  # Move north
        elif 0.25 < r <= 0.5:
            current_pos += np.array([1, 0])  # Move east
        elif 0.5 < r <= 0.75:
            current_pos += np.array([0, -1])  # Move south
        else:
            current_pos += np.array([-1, 0])  # Move west
        position[k + 1, :] = current_pos
    return position

def demo_random_walk2D():
    x0 = (0, 0)
    N = 200
    p = 0.5
    np.random.seed(10)
    pos = random_walk2D(x0, N, p, random=np.random)
    plt.plot(pos[:, 0], pos[:, 1])
```

```
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def random_walkdD(x0, N, p, random=random):
    """Any-D (diagonal) random walk with 1 particle and N moves."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        for i in range(d):
            r = random.uniform(0, 1)
            if r <= p:
                current_pos[i] -= 1
            else:
                current_pos[i] += 1
        position[k + 1, :] = current_pos
    return position

def random_walkdD_vec(x0, N, p):
    """Vectorized version of random_walkdD."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0] = np.array(x0, dtype=float)
    r = np.random.uniform(0, 1, size=N * d)
    steps = np.where(r <= p, -1, 1).reshape(N, d)
    position[1:, :] = x0 + np.cumsum(steps, axis=0)
    return position
```

3.65. Multiple random walks in any number of space dimensions

As we did in 1D, we extend one single walk to a number of walks (`num_walks` in the code).

3.65.1. Scalar code

As always, we start with implementing the scalar case:

```
import random

import matplotlib.pyplot as plt
import numpy as np

random.seed(10)
```

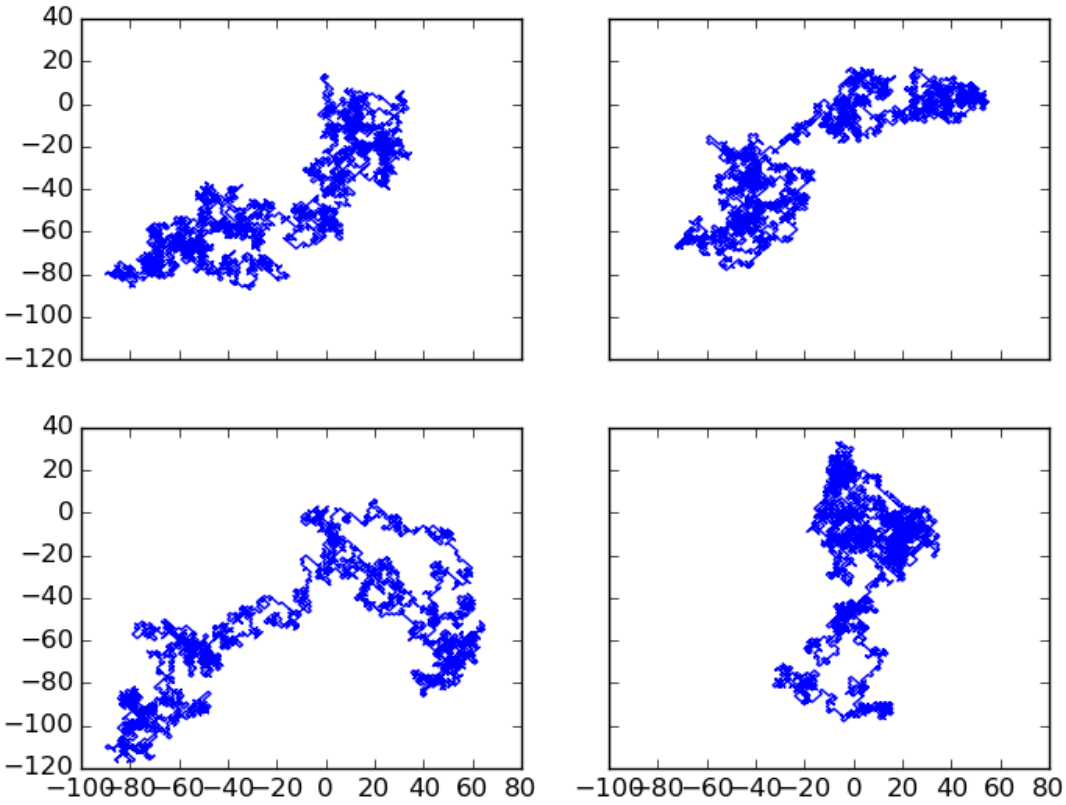


Figure 3.22.: Four random walks with 5000 steps in 2D.

```
np.random.seed(10)

def random_walk1D(x0, N, p, random=random):
    """1D random walk with 1 particle and N moves."""

    position = np.zeros(N + 1)
    position[0] = x0
    current_pos = x0
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= p:
            current_pos -= 1
        else:
            current_pos += 1
        position[k + 1] = current_pos
    return position

def random_walk1D_vec(x0, N, p):
    """Vectorized version of random_walk1D."""
    position = np.zeros(N + 1)
    position[0] = x0
    r = np.random.uniform(0, 1, size=N)
    steps = np.where(r <= p, -1, 1)
    position[1:] = x0 + np.cumsum(steps)
    return position

def test_random_walk1D():
    x0 = 2
    N = 4
    p = 0.6
    np.random.seed(10)
    scalar_computed = random_walk1D(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walk1D_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def demo_random_walk1D(N=50000):
    np.random.seed(10)
    pos = random_walk1D_vec(x0=0, N=N, p=0.5)
    plt.figure()
    plt.plot(pos)
    plt.savefig("tmp1.pdf")
    plt.savefig("tmp1.png")
    plt.figure()
    plt.plot(pos * pos)
    plt.savefig("tmp2.pdf")
    plt.savefig("tmp2.png")
```

```
    plt.show()

def demo_fig_random_walk1D(N=200):
    """Make ensamble of positions (to illustrate E[] operator)."""
    np.random.seed(10)
    num_plots = 4
    for n in range(num_plots):
        plt.subplot(num_plots, 1, n + 1)
        pos = random_walk1D_vec(x0=0, N=N, p=0.5)
        plt.plot(pos)
        plt.axis([0, N, -15, 20])
    plt.savefig("tmp.pdf")
    plt.savefig("tmp.png")
    plt.show()

def demo_random_walk1D_timing():
    import time

    x0 = 0
    N = 10000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walk1D(x0, N, p, random=np.random)
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos = random_walk1D_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def random_walks1D(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    position = np.zeros(N + 1)  # Accumulated positions
    position[0] = x0 * num_walks
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    position2[0] = x0**2 * num_walks
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
        current_pos = x0
```

```
        for k in range(N):
            if k in pos_hist_times:
                pos_hist[n, num_times_counter] = current_pos
                num_times_counter += 1
            r = random.uniform(0, 1)
            if r <= p:
                current_pos -= 1
            else:
                current_pos += 1
            position[k + 1] += current_pos
            position2[k + 1] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec1(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walk = np.zeros(N + 1)  # Positions of current walk
    walk[0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        r = np.random.uniform(0, 1, size=N)
        steps = np.where(r <= p, -1, 1)
        walk[1:] = x0 + np.cumsum(steps)  # Positions of this walk
        position += walk
        position2 += walk**2
        pos_hist[n, :] = walk[pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)

def random_walks1D_vec2(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    position = np.zeros(N + 1)  # Accumulated positions
    position2 = np.zeros(N + 1)  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1))  # Positions of each walk
    walks[:, 0] = x0
    pos_hist = np.zeros((num_walks, num_times))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks)
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N)
    walks[:, 1:] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :] = walks[:, pos_hist_times]
    return position, position2, pos_hist, np.array(pos_hist_times)
```

```
def test_random_walks1D():
    x0 = 0
    N = 4
    p = 0.5

    num_walks = 1
    np.random.seed(10)
    computed = random_walks1D(x0, N, p, num_walks, random=np.random)
    np.random.seed(10)
    expected = random_walk1D(x0, N, p, random=np.random)
    assert (computed[0] == expected).all()

    np.random.seed(10)
    computed = random_walks1D_vec1(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()
    np.random.seed(10)
    computed = random_walks1D_vec2(x0, N, p, num_walks)
    np.random.seed(10)
    expected = random_walk1D_vec(x0, N, p)
    assert (computed[0] == expected).all()

    num_walks = 3
    num_times = N
    np.random.seed(10)
    serial_computed = random_walks1D(x0, N, p, num_walks, num_times, random=np.random)
    np.random.seed(10)
    vectorized1_computed = random_walks1D_vec1(x0, N, p, num_walks, num_times)
    np.random.seed(10)
    vectorized2_computed = random_walks1D_vec2(x0, N, p, num_walks, num_times)
    return_values = ["pos", "pos2", "pos_hist", "pos_hist_times"]
    for s, v, r in zip(serial_computed, vectorized1_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg
    for s, v, r in zip(serial_computed, vectorized2_computed, return_values):
        msg = "%s: %s (serial) vs %s (vectorized)" % (r, s, v)
        assert (s == v).all(), msg

def demo_random_walks1D(N=1000, num_walks=10000, EX_minmax=None):
    import time

    t0 = time.perf_counter()
    pos, pos2, hist, hist_times = random_walks1D_vec1(
        x0=0,
        N=N,
        p=0.5,
```

```
        num_walks=num_walks,
        num_times=10,
    )
    t1 = time.perf_counter()
    print("histogram times:", hist_times)
    print("random walk: %.1fs" % (t1 - t0))
    E_X = pos / float(num_walks)
    Var_X = pos2 / float(num_walks) - E_X**2
    if N <= 50:
        print(pos)

    plt.figure()
    plt.plot(E_X)
    if EX_minmax is not None:
        plt.axis([0, N, EX_minmax[0], EX_minmax[1]])
    plt.title("Expected position (%d walks)" % num_walks)
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.figure()
    plt.plot(Var_X)
    plt.title("Variance of position (%d walks)" % num_walks)
    plt.savefig("tmp2.png")
    plt.savefig("tmp2.pdf")

    plt.figure()
    a = 0.5
    exact = (
        lambda x, t: 1.0 / np.sqrt(4 * np.pi * t * a) * np.exp(-(x**2) / (4.0 * t * a))
    )
    hist_time_index = -2
    n, bins, patches = plt.hist(hist[:, hist_time_index], bins=30, normed=True)
    x = np.linspace(bins[0], bins[-1], 301)
    t = hist_times[hist_time_index]
    plt.plot(x, exact(x, t), "r--")
    plt.title("Histogram of positions (%d walks)" % num_walks)
    plt.savefig("tmp3.png")
    plt.savefig("tmp3.pdf")
    plt.show()

def demo_fig_random_walks1D():
    """Make figures with statistics and dependence on no of walks."""
    import os
    import shutil

    N = 1000
    num_walks = [100, 10000, 100000, 1000000]
    for n in num_walks:
```

```
        np.random.seed(10)  # Use same seq. for all experiments
        if n == 100:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=None)
        else:
            demo_random_walks1D(N=N, num_walks=n, EX_minmax=[-0.1, 0.5])
        d = "tmp_%d" % n
        if os.path.isdir(d):
            shutil.rmtree(d)
        os.mkdir(d)
        for p in 1, 2, 3:
            os.rename("tmp%d.png" % p, os.path.join(d, "tmp%d.png" % p))
            os.rename("tmp%d.pdf" % p, os.path.join(d, "tmp%d.pdf" % p))
    plots = ["EX", "VarX", "HistX"]
    for j, plot in enumerate(plots):
        for ext in "png", "pdf":
            files = [
                os.path.join("tmp_%d" % n, "tmp%d.%s" % (j + 1, ext)) for n in num_walks
            ]
            ncols = 3 if len(num_walks) == 3 else 2
            output = "rw1D_%s_%s.%s" % (plot, "_".join([str(n) for n in num_walks]), ext)
            cmd = "montage %s -tile %dx1 -geometry +0+0 %s" % (
                " ".join(files),
                ncols,
                output,
            )
            print(cmd)
            os.system(cmd)

def demo_random_walks1D_timing():
    import time

    x0 = 0
    N = 1000
    num_walks = 50000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D(
        x0, N, p, num_walks, num_times=4, random=np.random
    )
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec1(
        x0, N, p, num_walks, num_times=4
```

```
    )
    t2 = time.perf_counter()
    cpu_vec1 = t2 - t1
    print("CPU vectorized1: %.1f" % cpu_vec1)
    print("CPU scalar/vectorized1: %.1f" % (cpu_scalar / cpu_vec1))
    np.random.seed(10)
    pos, pos2, pos_hist, pos_hist_times = random_walks1D_vec2(
        x0, N, p, num_walks, num_times=4
    )
    t3 = time.perf_counter()
    cpu_vec2 = t3 - t2
    print("CPU vectorized2: %.1f" % cpu_vec2)
    print("CPU scalar/vectorized2: %.1f" % (cpu_scalar / cpu_vec2))

def random_walk2D(x0, N, p, random=random):
    """2D random walk with 1 particle and N moves: N, E, W, S."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        r = random.uniform(0, 1)
        if r <= 0.25:
            current_pos += np.array([0, 1])  # Move north
        elif 0.25 < r <= 0.5:
            current_pos += np.array([1, 0])  # Move east
        elif 0.5 < r <= 0.75:
            current_pos += np.array([0, -1])  # Move south
        else:
            current_pos += np.array([-1, 0])  # Move west
        position[k + 1, :] = current_pos
    return position

def demo_random_walk2D():
    x0 = (0, 0)
    N = 200
    p = 0.5
    np.random.seed(10)
    pos = random_walk2D(x0, N, p, random=np.random)
    plt.plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def random_walkdD(x0, N, p, random=random):
    """Any-D (diagonal) random walk with 1 particle and N moves."""
    d = len(x0)
```

```
    position = np.zeros((N + 1, d))
    position[0, :] = x0
    current_pos = np.array(x0, dtype=float)
    for k in range(N):
        for i in range(d):
            r = random.uniform(0, 1)
            if r <= p:
                current_pos[i] -= 1
            else:
                current_pos[i] += 1
        position[k + 1, :] = current_pos
    return position

def random_walkD_vec(x0, N, p):
    """Vectorized version of random_walkD."""
    d = len(x0)
    position = np.zeros((N + 1, d))
    position[0] = np.array(x0, dtype=float)
    r = np.random.uniform(0, 1, size=N * d)
    steps = np.where(r <= p, -1, 1).reshape(N, d)
    position[1:, :] = x0 + np.cumsum(steps, axis=0)
    return position

def demo_random_walkD():
    x0 = (0, 0)
    N = 200
    p = 0.5
    np.random.seed(10)
    pos = random_walkD(x0, N, p, random=np.random)
    plt.plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def demo_random_walkD_timing():
    import time

    x0 = (0, 0)
    N = 4000000
    p = 0.5

    t0 = time.perf_counter()
    np.random.seed(10)
    pos = random_walkD(x0, N, p, random=np.random)
    t1 = time.perf_counter()
    cpu_scalar = t1 - t0
    print("CPU scalar: %.1f" % cpu_scalar)
```

```
    np.random.seed(10)
    pos = random_walkdD_vec(x0, N, p)
    t2 = time.perf_counter()
    cpu_vec = t2 - t1
    print("CPU vectorized: %.1f" % cpu_vec)
    print("CPU scalar/vectorized: %.1f" % (cpu_scalar / cpu_vec))

def demo_fig_random_walkdD():
    x0 = (0, 0)
    N = 5000
    p = 0.5
    n = 2  # nxn subplots
    f, axarr = plt.subplots(n, n, sharex=True, sharey=True)
    for i in range(n):
        for j in range(n):
            seed = 3 * i + 8 * j
            np.random.seed(seed)
            pos = random_walkdD(x0, N, p, random=np.random)
            axarr[i, j].plot(pos[:, 0], pos[:, 1])
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()

def test_ramdom_walkdD():
    x0 = (0, 0)
    N = 7
    p = 0.5
    np.random.seed(10)
    scalar_computed = random_walkdD(x0, N, p, random=np.random)
    np.random.seed(10)
    vectorized_computed = random_walkdD_vec(x0, N, p)
    assert (scalar_computed == vectorized_computed).all()

def random_walksdD(x0, N, p, num_walks=1, num_times=1, random=random):
    """Simulate num_walks random walks from x0 with N steps."""
    d = len(x0)
    position = np.zeros((N + 1, d))  # Accumulated positions
    position2 = np.zeros((N + 1, d))  # Accumulated positions**2
    pos_hist = np.zeros((num_walks, num_times, d))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    for n in range(num_walks):
        num_times_counter = 0
        current_pos = np.array(x0, dtype=float)
        for k in range(N):
            if k in pos_hist_times:
                pos_hist[n, num_times_counter, :] = current_pos
```

```
                num_times_counter += 1
            for i in range(d):
                r = random.uniform(0, 1)
                if r <= p:
                    current_pos[i] -= 1
                else:
                    current_pos[i] += 1
            position[k + 1, :] += current_pos
            position2[k + 1, :] += current_pos**2
    return position, position2, pos_hist, np.array(pos_hist_times)
```

3.65.2. Vectorized code

Significant speed-ups can be obtained by vectorization. We get rid of the loops in the previous function and arrive at the following vectorized code.

```
def random_walksdD_vec(x0, N, p, num_walks=1, num_times=1):
    """Vectorized version of random_walks1D; no loops."""
    d = len(x0)
    position = np.zeros((N + 1, d))  # Accumulated positions
    position2 = np.zeros((N + 1, d))  # Accumulated positions**2
    walks = np.zeros((num_walks, N + 1, d))  # Positions of each walk
    walks[:, 0, :] = x0
    pos_hist = np.zeros((num_walks, num_times, d))
    pos_hist_times = [(N // num_times) * i for i in range(num_times)]

    r = np.random.uniform(0, 1, size=N * num_walks * d)
    steps = np.where(r <= p, -1, 1).reshape(num_walks, N, d)
    walks[:, 1:, :] = x0 + np.cumsum(steps, axis=1)
    position = np.sum(walks, axis=0)
    position2 = np.sum(walks**2, axis=0)
    pos_hist[:, :, :] = walks[:, pos_hist_times, :]
    return position, position2, pos_hist, np.array(pos_hist_times)
```

3.66. Applications

3.66.1. Diffusion of a substance

The first process to be considered is a substance that gets transported through a fluid at rest by pure diffusion. We consider an arbitrary volume V of this fluid, containing the substance with concentration function $c(\boldsymbol{x}, t)$. Physically, we can think of a very small volume with centroid $\boldsymbol{x}$ at time t and assign the ratio of the volume of the substance and the total volume to $c(\boldsymbol{x}, t)$. This means that the mass of the substance in a small volume ΔV is approximately $\varrho c \Delta V$, where ϱ is the density of the substance. Consequently, the total mass of the substance inside the volume V is the sum of all $\varrho c \Delta V$, which becomes the volume integral $\int_V \varrho c dV$.

Let us reason how the mass of the substance changes and thereby derive a PDE governing the concentration c. Suppose the substance flows out of V with a flux $\boldsymbol{q}$. If ΔS is a small part of the boundary ∂V of V, the volume of the substance flowing out through dS in a small time interval Δt is $\varrho \boldsymbol{q} \cdot \boldsymbol{n} \Delta t \Delta S$, where $\boldsymbol{n}$ is an outward unit normal to the boundary ∂V, see Figure Figure 3.23. We realize that only the normal component of $\boldsymbol{q}$ is able to transport mass in and out of V. The total outflow of the mass of the substance in a small time interval Δt becomes the surface integral

$$\int_{\partial V} \varrho \boldsymbol{q} \cdot \boldsymbol{n} \Delta t \, dS \, .$$

Assuming conservation of mass, this outflow of mass must be balanced by a loss of mass inside the volume. The increase of mass inside the volume, during a small time interval Δt, is

$$\int_V \varrho(c(\boldsymbol{x}, t + \Delta t) - c(\boldsymbol{x}, t)) dV,$$

assuming ϱ is constant, which is reasonable. The outflow of mass balances the loss of mass in V, which is the increase with a minus sign. Setting the two contributions equal to each other ensures balance of mass inside V. Dividing by Δt gives

$$\int_V \varrho \frac{c(\boldsymbol{x}, t + \Delta t) - c(\boldsymbol{x}, t)}{\Delta t} dV = - \int_{\partial V} \varrho \boldsymbol{q} \cdot \boldsymbol{n} \, dS \, .$$

Note the minus sign on the right-hand side: the left-hand side expresses loss of mass, while the integral on the right-hand side is the gain of mass.

Now, letting $\Delta t \to 0$, we have

$$\frac{c(\boldsymbol{x}, t + \Delta t) - c(\boldsymbol{x}, t)}{\Delta t} \to \frac{\partial c}{\partial t},$$

so

$$\int_V \varrho \frac{\partial c}{\partial t} dV + \int_{\partial V} \varrho \boldsymbol{q} \cdot \boldsymbol{n} \, dS = 0 \, . \tag{3.94}$$

To arrive at a PDE, we express the surface integral as a volume integral using Gauss' divergence theorem:

$$\int_V (\varrho \frac{\partial c}{\partial t} + \nabla \cdot (\varrho \boldsymbol{q})) dV = 0 \, .$$

Since ϱ is constant, we can divide by this quantity. If the integral is to vanish for an arbitrary volume V, the integrand must vanish too, and we get the mass conservation PDE for the substance:

$$\frac{\partial c}{\partial t} + \nabla \cdot \boldsymbol{q} = 0 \, . \tag{3.95}$$

A fundamental problem is that this is a scalar PDE for four unknowns: c and the three components of $\boldsymbol{q}$. We therefore need additional equations. Here, Fick's law comes at rescue: it models how the flux $\boldsymbol{q}$ of the substance is related to the concentration c. Diffusion is recognized by mass flowing from regions with high concentration to regions of low concentration. This principle suggests that $\boldsymbol{q}$ is proportional to the negative gradient of c:

$$\boldsymbol{q} = -\alpha \nabla c, \tag{3.96}$$

where α is an empirically determined constant. The relation (3.96) is known as Fick's law. Inserting (3.96) in (3.95) gives a scalar PDE for the concentration c:

$$\frac{\partial c}{\partial t} = \alpha \nabla^2 c \, . \tag{3.97}$$

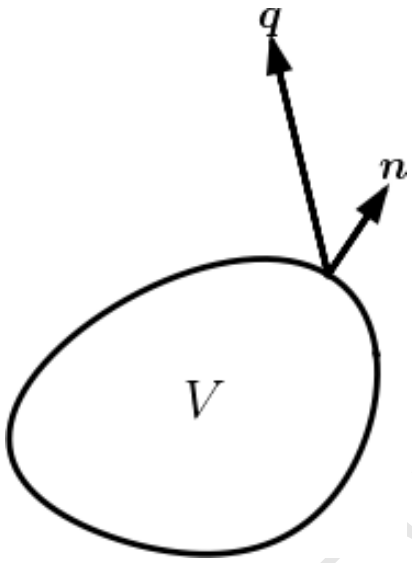


Figure 3.23.: An arbitrary volume of a fluid.

3.66.2. Heat conduction

Heat conduction is a well-known diffusion process. The governing PDE is in this case based on the first law of thermodynamics: the increase in energy of a system is equal to the work done on the system, plus the supplied heat. Here, we shall consider media at rest and neglect work done on the system. The principle then reduces to a balance between increase in internal energy and supplied heat flow by conduction.

Let $e(x,t)$ be the *internal energy* per unit mass. The increase of the internal energy in a small volume ΔV in a small time interval Δt is then

$$\varrho(e(\boldsymbol{x},t+\Delta t)-e(\boldsymbol{x},t))\Delta V,$$

where ϱ is the density of the material subject to heat conduction. In an arbitrary volume V, as depicted in Figure Figure 3.23, the corresponding increase in internal energy becomes the volume integral

$$\int_V \varrho(e(\boldsymbol{x},t+\Delta t)-e(\boldsymbol{x},t))dV\,.$$

This increase in internal energy is balanced by heat supplied by conduction. Let $\boldsymbol{q}$ be the heat flow per time unit. Through the surface ∂V of V the following amount of heat flows out of V during a time interval Δt:

$$\int_{\partial V} \boldsymbol{q}\cdot\boldsymbol{n}\Delta t\,dS\,.$$

The simplified version of the first law of thermodynamics then states that

$$\int_V \varrho(e(\boldsymbol{x},t+\Delta t)-e(\boldsymbol{x},t))dV = -\int_{\partial V} \boldsymbol{q}\cdot\boldsymbol{n}\Delta t\,dS\,.$$

The minus sign on the right-hand side ensures that the integral there models net *inflow* of heat (since $\boldsymbol{n}$ is an outward unit normal, $\boldsymbol{q}\cdot\boldsymbol{n}$ models *outflow*). Dividing by Δt and notifying that

$$\lim_{\Delta t\to 0}\frac{e(\boldsymbol{x},t+\Delta t)-e(\boldsymbol{x},t)}{\Delta t}=\frac{\partial e}{\partial t},$$

we get (in the limit $\Delta t\to 0$)

$$\int_V \varrho\frac{\partial e}{\partial t}dV+\int_{\partial V}\boldsymbol{q}\cdot\boldsymbol{n}\Delta t\,dS=0\,.$$

This is the integral equation for heat conduction, but we aim at a PDE. The next step is therefore to transform the surface integral to a volume integral via Gauss' divergence theorem. The result is

$$\int_V\left(\varrho\frac{\partial e}{\partial t}+\nabla\cdot\boldsymbol{q}\right)dV=0\,.$$

If this equality is to hold for all volumes V, the integrand must vanish, and we have the PDE

$$\varrho\frac{\partial e}{\partial t}=-\nabla\cdot\boldsymbol{q}\,. \tag{3.98}$$

Sometimes the supplied heat can come from the medium itself. This is the case, for instance, when radioactive rock generates heat. Let us add this effect. If $f(\boldsymbol{x},t)$ is the supplied heat per unit volume per unit time, the heat supplied in a small volume is $f\Delta t\Delta V$, and inside an arbitrary volume V the supplied generated heat becomes

$$\int_V f\Delta t dV\,.$$

Adding this to the integral statement of the (simplified) first law of thermodynamics, and continuing the derivation, leads to the PDE

$$\varrho\frac{\partial e}{\partial t}=-\nabla\cdot\boldsymbol{q}+f\,. \tag{3.99}$$

There are four unknown scalar fields: e and $\boldsymbol{q}$. Moreover, the temperature T, which is our primary quantity to compute, does not enter the model yet. We need an additional equation, called the *equation of state*, relating e, $V=1/\varrho=$, and T: $e=e(V,T)$. By the chain rule we have

$$\frac{\partial e}{\partial t}=\left.\frac{\partial e}{\partial T}\right|_V\frac{\partial T}{\partial t}+\left.\frac{\partial e}{\partial V}\right|_T\frac{\partial V}{\partial t}\,.$$

The first coefficient $\partial e/\partial T$ is called *specific heat capacity at constant volume*, denoted by c_v:

$$c_v=\left.\frac{\partial e}{\partial T}\right|_V\,.$$

The specific heat capacity will in general vary with T, but taking it as a constant is a good approximation in many applications.

The term $\partial e/\partial V$ models effects due to compressibility and volume expansion. These effects are often small and can be neglected. We shall do so here. Using $\partial e/\partial t=c_v\partial T/\partial t$ in the PDE gives

$$\varrho c_v\frac{\partial T}{\partial t}=-\nabla\cdot\boldsymbol{q}+f\,.$$

We still have four unknown scalar fields (T and $\boldsymbol{q}$). To close the system, we need a relation between the heat flux $\boldsymbol{q}$ and the temperature T called *Fourier's law*:

$$\boldsymbol{q} = -k\nabla T,$$

which simply states that heat flows from hot to cold areas, along the path of greatest variation. In a solid medium, k depends on the material of the medium, and in multi-material media one must regard k as spatially dependent. In a fluid, it is common to assume that k is constant. The value of k reflects how easy heat is conducted through the medium, and k is named the *coefficient of heat conduction*.

We now have one scalar PDE for the unknown temperature field $T(\boldsymbol{x}, t)$:

$$\varrho c_v \frac{\partial T}{\partial t} = \nabla \cdot (k\nabla T) + f\,. \tag{3.100}$$

3.66.3. Porous media flow

The requirement of mass balance for flow of a single, incompressible fluid through a deformable (elastic) porous medium leads to the equation

$$S\frac{\partial p}{\partial t} + \nabla \cdot (\boldsymbol{q} - \alpha\frac{\partial \boldsymbol{u}}{\partial t}) = 0,$$

where p is the fluid pressure, $\boldsymbol{q}$ is the fluid velocity, $\boldsymbol{u}$ is the displacement (deformation) of the medium, S is the storage coefficient of the medium (related to the compressibility of the fluid and the material in the medium), and α is another coefficient. In many circumstances, the last term with $\boldsymbol{u}$ can be neglected, an assumption that decouples the equation above from a model for the deformation of the medium. The famous *Darcy's law* relates $\boldsymbol{q}$ to p:

$$\boldsymbol{q} = -\frac{K}{\mu}(\nabla p - \varrho \boldsymbol{g}),$$

where K is the permeability of the medium, μ is the dynamic viscosity of the fluid, ϱ is the density of the fluid, and $\boldsymbol{g}$ is the acceleration of gravity, here taken as $\boldsymbol{g} = -g\boldsymbol{k}$. Combining the two equations results in the diffusion model

$$S\frac{\partial p}{\partial t} = \mu^{-1}\nabla(K\nabla p) + \frac{\varrho g}{\mu}\frac{\partial K}{\partial z}\,. \tag{3.101}$$

Boundary conditions consist of specifying p or $\boldsymbol{q}\cdot\boldsymbol{n}$ (i.e., normal velocity) at each point of the boundary.

3.66.4. Potential fluid flow

Let $\boldsymbol{v}$ be the velocity of a fluid. The condition $\nabla \times \boldsymbol{v} = 0$ is relevant for many flows, especially in geophysics when viscous effects are negligible. From vector calculus it is known that $\nabla \times \boldsymbol{v} = 0$ implies that v can be derived from a scalar potential field ϕ: $\boldsymbol{v} = \nabla\phi$. If the fluid is incompressible, $\nabla \cdot \boldsymbol{v} = 0$, it follows that $\nabla \cdot \nabla\phi = 0$, or

$$\nabla^2\phi = 0\,.$$

This Laplace equation is sufficient for determining ϕ and thereby describe the fluid motion. This type of flow is known as potential flow. One very important application where potential flow is a good model is water waves. As boundary condition we must prescribe $\boldsymbol{v} \cdot \boldsymbol{n} = \partial\phi/\partial n$. This gives rise to what is known as a pure Neumann problem and will cause numerical difficulties because ϕ and ϕ plus any constant are two solutions of the problem. The simplest remedy is to fix the value of ϕ at a point.

3.66.5. Streamlines for 2D fluid flow

The streamlines in a two-dimensional stationary fluid flow are lines tangential to the flow. The stream function ψ is often introduced in two-dimensional flow such that its contour lines, $\psi = \text{const}$, gives the streamlines. The relation between ψ and the velocity field $\boldsymbol{v} = (u, v)$ is

$$u = \frac{\partial\psi}{\partial y}, \quad v = -\frac{\partial\psi}{\partial x}\,.$$

It follows that $\nabla\boldsymbol{v} = \psi_{yx} - \psi_{xy} = 0$, so the stream function can only be used for incompressible flows. Since

$$\nabla \times \boldsymbol{v} = \left(\frac{\partial v}{\partial y} - \frac{\partial u}{\partial x}\right)\boldsymbol{k} \equiv \omega\boldsymbol{k},$$

we can derive the relation

$$\nabla^2\psi = -\omega,$$

which is a governing equation for the stream function $\psi(x, y)$ if the vorticity ω is known.

3.66.6. The potential of an electric field

Under the assumption of time independence, Maxwell's equations for the electric field $\boldsymbol{E}$ become

$$\nabla \cdot \boldsymbol{E} = \frac{\rho}{\epsilon_0},$$
$$\nabla \times \boldsymbol{E} = 0,$$

where ρ is the electric charge density and ϵ_0 is the electric permittivity of free space (i.e., vacuum). Since $\nabla \times \boldsymbol{E} = 0$, $\boldsymbol{E}$ can be derived from a potential φ, $\boldsymbol{E} = -\nabla\varphi$. The electric field potential is therefore governed by the Poisson equation

$$\nabla^2\varphi = -\frac{\rho}{\epsilon_0}\,.$$

If the medium is heterogeneous, ρ will depend on the spatial location $\boldsymbol{r}$. Also, ϵ_0 must be exchanged with an electric permittivity function $\epsilon(\boldsymbol{r})$.

Each point of the boundary must be accompanied by, either a Dirichlet condition $\varphi(\boldsymbol{r}) = \varphi_D(\boldsymbol{r})$, or a Neumann condition $\frac{\partial\varphi(\boldsymbol{r})}{\partial n} = \varphi_N(\boldsymbol{r})$.

[sl: is this what you were thinking of?]

3.66.7. Development of flow between two flat plates

Diffusion equations may also arise as simplified versions of other mathematical models, especially in fluid flow. Consider a fluid flowing between two flat, parallel plates. The velocity is uni-directional, say along the z axis, and depends only on the distance x from the plates; $\boldsymbol{u} = u(x,t)\boldsymbol{k}$. The flow is governed by the Navier-Stokes equations,

$$\varrho\frac{\partial \boldsymbol{u}}{\partial t} + \varrho\boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + \mu\nabla^2\boldsymbol{u} + \varrho\boldsymbol{f},$$
$$\nabla\cdot\boldsymbol{u} = 0,$$

where p is the pressure field, unknown along with the velocity $\boldsymbol{u}$, ϱ is the fluid density, μ the dynamic viscosity, and $\boldsymbol{f}$ is some external body force. The geometric restrictions of flow between two flat plates puts restrictions on the velocity, $\boldsymbol{u} = u(x,t)\boldsymbol{i}$, and the z component of the Navier-Stokes equations collapses to a diffusion equation:

$$\varrho\frac{\partial u}{\partial t} = -\frac{\partial p}{\partial z} + \mu\frac{\partial^2 u}{\partial z^2} + \varrho f_z,$$

if f_z is the component of $\boldsymbol{f}$ in the z direction.

The boundary conditions are derived from the fact that the fluid sticks to the plates, which means $\boldsymbol{u} = 0$ at the plates. Say the location of the plates are $z = 0$ and $z = L$. We then have

$$u(0,t) = u(L,t) = 0\,.$$

One can easily show that $\partial p/\partial z$ must be a constant or just a function of time t. We set $\partial p/\partial z = -\beta(t)$. The body force could be a component of gravity, if desired, set as $f_z = \gamma g$. Switching from z to x as independent variable gives a very standard one-dimensional diffusion equation:

$$\varrho\frac{\partial u}{\partial t} = \mu\frac{\partial^2 u}{\partial x^2} + \beta(t) + \varrho\gamma g, \quad x\in[0,L],\ t\in(0,T]\,.$$

The boundary conditions are

$$u(0,t) = u(L,t) = 0,$$

while some initial condition

$$u(x,0) = I(x)$$

must also be prescribed.

The flow is driven by either the pressure gradient β or gravity, or a combination of both. One may also consider one moving plate that drives the fluid. If the plate at $x = L$ moves with velocity $U_L(t)$, we have the adjusted boundary condition

$$u(L,t) = U_L(t)\,.$$

Flow in a straight tube {#sec-diffu-app-pipeflow}

Now we consider viscous fluid flow in a straight tube with radius R and rigid walls. The governing equations are the Navier-Stokes equations, but as in Section Section 3.66.7, it is natural to assume that the velocity is directed along the tube, and that it is axi-symmetric. These assumptions reduced

the velocity field to $\boldsymbol{u} = u(r,x,t)\boldsymbol{i}$, if the x axis is directed along the tube. From the equation of continuity, $\nabla\cdot\boldsymbol{u}=0$, we see that u must be independent of x. Inserting $\boldsymbol{u}=u(r,t)\boldsymbol{i}$ in the Navier-Stokes equations, expressed in axi-symmetric cylindrical coordinates, results in

$$\varrho\frac{\partial u}{\partial t} = \mu\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \beta(t) + \varrho\gamma g, \quad r\in[0,R],\ t\in(0,T]\,. \tag{3.102}$$

Here, $\beta(t) = -\partial p/\partial x$ is the pressure gradient along the tube. The associated boundary condition is $u(R,t)=0$.

3.66.8. Tribology: thin film fluid flow

Thin fluid films are extremely important inside machinery to reduce friction between gliding surfaces. The mathematical model for the fluid motion takes the form of a diffusion problem and is quickly derived here. We consider two solid surfaces whose distance is described by a gap function $h(x,y)$. The space between these surfaces is filled with a fluid with dynamic viscosity μ. The fluid may move partially because of pressure gradients and partially because the surfaces move. Let $U\boldsymbol{i}+V\boldsymbol{j}$ be the relative velocity of the two surfaces and p the pressure in the fluid. The mathematical model builds on two principles: 1) conservation of mass, 2) assumption of locally quasi-static flow between flat plates.

The conservation of mass equation reads $\nabla\cdot\boldsymbol{u}$, where $\boldsymbol{u}$ is the local fluid velocity. For thin films the detailed variation between the surfaces is not of interest, so $\nabla\cdot\boldsymbol{u}=0$ is integrated (average) in the direction perpendicular to the surfaces. This gives rise to the alternative mass conservation equation

$$\nabla\cdot\boldsymbol{q}=0, \quad \boldsymbol{q} = \int_0^{h(x,y)} \boldsymbol{u}dz,$$

where z is the coordinate perpendicular to the surfaces, and $\boldsymbol{q}$ is then the volume flux in the fluid gap.

Locally, we may assume that we have steady flow between two flat surfaces, with a pressure gradient and where the lower surface is at rest and the upper moves with velocity $U\boldsymbol{i}+V\boldsymbol{j}$. The corresponding mathematical problem is actually the limit problem in Section Section 3.66.7 as $t\to\infty$. The limit problem can be solved analytically, and the local volume flux becomes

$$\boldsymbol{q}(x,y,z) = \int_0^h \boldsymbol{u}(x,y,z)dz = -\frac{h^3}{12\mu}\nabla p + \frac{1}{2}Uh\boldsymbol{i} + \frac{1}{2}Vh\boldsymbol{j}\,.$$

The idea is to use this expression locally also when the surfaces are not flat, but slowly varying, and if U, V, or p varies in time, provided the time variation is sufficiently slow. This is a common quasi-static approximation, much used in mathematical modeling.

Inserting the expression for $\boldsymbol{q}$ via p, U, and V in the equation $\nabla\boldsymbol{q}=0$ gives a diffusion PDE for p:

$$\nabla\cdot\left(\frac{h^3}{12\mu}\nabla p\right) = \frac{1}{2}\frac{\partial}{\partial x}(hU) + \frac{1}{2}\frac{\partial}{\partial x}(hV)\,.$$

The boundary conditions must involve p or $\boldsymbol{q}$ at the boundary.

3.66.9. Propagation of electrical signals in the brain

One can make a model of how electrical signals are propagated along the neuronal fibers that receive synaptic inputs in the brain. The signal propagation is one-dimensional and can, in the simplest cases, be governed by the Cable equation:

$$c_m \frac{\partial V}{\partial t} = \frac{1}{r_l}\frac{\partial^2 V}{\partial x^2} - \frac{1}{r_m} V label$$

where $V(x,t)$ is the voltage to be determined, c_m is capacitance of the neuronal fiber, while r_l and r_m are measures of the resistance. The boundary conditions are often taken as $V = 0$ at a short circuit or open end, $\partial V/\partial x = 0$ at a sealed end, or $\partial V/\partial x \propto V$ where there is an injection of current.

3.67. 2D Diffusion with Devito

Extending the diffusion solver to two dimensions illustrates Devito's dimension-agnostic approach. The same symbolic patterns apply, and the `.laplace` attribute automatically generates the correct 2D stencil.

3.67.1. The 2D Diffusion Equation

The two-dimensional diffusion equation on $[0, L_x] \times [0, L_y]$ is:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \alpha \nabla^2 u \tag{3.103}$$

where $\nabla^2 u = u_{xx} + u_{yy}$ is the Laplacian.

3.67.2. Devito's Dimension-Agnostic Laplacian

The `.laplace` attribute works identically in 1D, 2D, and 3D:

```
from devito import Grid, TimeFunction

# 2D grid
grid = Grid(shape=(Nx + 1, Ny + 1), extent=(Lx, Ly))

# 2D temperature field
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# The Laplacian automatically includes both u_xx and u_yy
laplacian = u.laplace  # Returns u_xx + u_yy
```

3.67.3. Stability Condition in 2D

The Forward Euler stability condition in 2D is more restrictive:

$$F = \alpha \cdot \Delta t \cdot \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right) \leq \frac{1}{2}$$

For equal grid spacing $\Delta x = \Delta y = h$:

$$\Delta t \leq \frac{h^2}{4\alpha}$$

This means $F \leq 0.25$ with equal spacing, compared to $F \leq 0.5$ in 1D.

3.67.4. The 2D Solver

The `src.diffu` module provides `solve_diffusion_2d`:

```
from src.diffu import solve_diffusion_2d
import numpy as np

# Initial condition: 2D sinusoidal mode
def I(X, Y):
    return np.sin(np.pi * X) * np.sin(np.pi * Y)

result = solve_diffusion_2d(
    Lx=1.0, Ly=1.0,    # Domain size
    a=1.0,              # Diffusion coefficient
    Nx=50, Ny=50,       # Grid points
    T=0.1,              # Final time
    F=0.25,             # Fourier number (2D stability limit)
    I=I,                # Initial temperature
)

# Result is a 2D array
print(result.u.shape)  # (51, 51)
```

3.67.5. 2D Boundary Conditions

Dirichlet conditions must be applied on all four boundaries:

```
from devito import Eq

t_dim = grid.stepping_dim
x_dim, y_dim = grid.dimensions

# Boundary conditions (u = 0 on all boundaries)
bc_x0 = Eq(u[t_dim + 1, 0, y_dim], 0)      # Left
```

```
bc_xN = Eq(u[t_dim + 1, Nx, y_dim], 0)      # Right
bc_y0 = Eq(u[t_dim + 1, x_dim, 0], 0)       # Bottom
bc_yN = Eq(u[t_dim + 1, x_dim, Ny], 0)      # Top
```

3.67.6. Exact Solution for Verification

The exact solution for the initial condition $I(x, y) = \sin(\pi x/L_x)\sin(\pi y/L_y)$ is:

$$u(x, y, t) = e^{-\alpha\kappa t} \sin\left(\frac{\pi x}{L_x}\right) \sin\left(\frac{\pi y}{L_y}\right)$$

where the decay rate is:

$$\kappa = \pi^2 \left(\frac{1}{L_x^2} + \frac{1}{L_y^2}\right)$$

This can be used for verification:

```
from src.diffu import convergence_test_diffusion_2d

grid_sizes, errors, rate = convergence_test_diffusion_2d(
    grid_sizes=[10, 20, 40, 80],
    T=0.05,
    F=0.25,
)

print(f"Observed convergence rate: {rate:.2f}")  # Should be ~2.0
```

3.67.7. Visualizing 2D Solutions

For 2D problems, contour plots and surface plots are useful:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

result = solve_diffusion_2d(Lx=1.0, Ly=1.0, Nx=50, Ny=50, T=0.1, F=0.25)

X, Y = np.meshgrid(result.x, result.y, indexing='ij')

fig = plt.figure(figsize=(12, 5))

# Surface plot
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, result.u, cmap='hot')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
```

```
ax1.set_zlabel('Temperature')
ax1.set_title(f't = {result.t:.3f}')

# Contour plot
ax2 = fig.add_subplot(122)
c = ax2.contourf(X, Y, result.u, levels=20, cmap='hot')
plt.colorbar(c, ax=ax2)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Temperature distribution')
ax2.set_aspect('equal')
```

3.67.8. Heat Diffusion from a Point Source

A classic problem is the diffusion of heat from a localized hot spot:

```
from src.diffu import gaussian_2d_initial_condition

# Gaussian "hot spot" in the center
result = solve_diffusion_2d(
    Lx=1.0, Ly=1.0, Nx=50, Ny=50, T=0.2, F=0.25,
    I=lambda X, Y: gaussian_2d_initial_condition(X, Y, 1.0, 1.0, sigma=0.1),
    save_history=True,
)
```

The Gaussian spreads out and decays over time, eventually approaching zero as heat is lost through the boundaries.

3.67.9. Animation of 2D Diffusion

```
from matplotlib.animation import FuncAnimation

result = solve_diffusion_2d(
    Lx=1.0, Ly=1.0, Nx=50, Ny=50, T=0.5, F=0.25,
    save_history=True,
)

fig, ax = plt.subplots()
X, Y = np.meshgrid(result.x, result.y, indexing='ij')

vmax = result.u_history[0].max()
im = ax.contourf(X, Y, result.u_history[0], levels=20,
                 cmap='hot', vmin=0, vmax=vmax)
```

```
def update(frame):
    ax.clear()
    ax.contourf(X, Y, result.u_history[frame], levels=20,
                cmap='hot', vmin=0, vmax=vmax)
    ax.set_title(f't = {result.t_history[frame]:.3f}')
    ax.set_aspect('equal')
    return []

anim = FuncAnimation(fig, update, frames=len(result.t_history),
                     interval=50)
```

3.67.10. From 2D to 3D

The pattern extends naturally to three dimensions:

```
# 3D grid
grid = Grid(shape=(Nx+1, Ny+1, Nz+1), extent=(Lx, Ly, Lz))

# 3D temperature field
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# The PDE is unchanged - .laplace now includes u_zz
pde = u.dt - a * u.laplace
```

The stability condition in 3D becomes:

$$F \leq \frac{1}{6} \approx 0.167$$

for equal grid spacing in all directions.

3.67.11. Computational Efficiency

2D and 3D diffusion simulations can become computationally expensive as the number of grid points grows. Devito helps through:

- **Automatic parallelization**: Set `OMP_NUM_THREADS` for OpenMP
- **Cache optimization**: Loop tiling is applied automatically
- **GPU support**: Use `platform='nvidiaX'` for CUDA execution

The explicit Forward Euler scheme is embarrassingly parallel since each new value depends only on neighbors at the previous time level.

3.67.12. Comparison: Diffusion vs Wave Equation

Property	Diffusion	Wave
Time derivative	First order	Second order
Stability (2D)	$F \leq 0.25$	$C \leq 1/\sqrt{2}$
Solution character	Smoothing, decaying	Propagating, oscillating
Physical process	Heat conduction	Vibrations, acoustics

3.67.13. Summary

Key points for 2D diffusion with Devito:

1. The `.laplace` attribute handles dimension automatically
2. Stability conditions are more restrictive in higher dimensions
3. Equal spacing gives $F \leq 0.25$ in 2D, $F \leq 1/6$ in 3D
4. The same code patterns extend from 1D to 2D to 3D
5. Visualization requires contour/surface plots and animations

Devito's abstraction means we write the physics symbolically and let the framework handle the computational complexity across dimensions.

3.68. Exercise: Stabilizing the Crank-Nicolson method by Rannacher time stepping

It is well known that the Crank-Nicolson method may give rise to non-physical oscillations in the solution of diffusion equations if the initial data exhibit jumps (see Section Section 3.21). Rannacher (Rannacher 1984) suggested a stabilizing technique consisting of using the Backward Euler scheme for the first two time steps with step length $\frac{1}{2}\Delta t$. One can generalize this idea to taking $2m$ time steps of size $\frac{1}{2}\Delta t$ with the Backward Euler method and then continuing with the Crank-Nicolson method, which is of second-order in time. The idea is that the high frequencies of the initial solution are quickly damped out, and the Backward Euler scheme treats these high frequencies correctly. Thereafter, the high frequency content of the solution is gone and the Crank-Nicolson method will do well.

Test this idea for $m = 1, 2, 3$ on a diffusion problem with a discontinuous initial condition. Measure the convergence rate using the solution (3.40) with the boundary conditions (3.41)-(3.42) for t values such that the conditions are in the vicinity of ± 1. For example, $t < 5a1.6 \cdot 10^{-2}$ makes the solution diffusion from a step to almost a straight line. The program `diffu_erf_sol.py` shows how to compute the analytical solution.

3.69. Project: Energy estimates for diffusion problems

This project concerns so-called *energy estimates* for diffusion problems that can be used for qualitative analytical insight and for verification of implementations.

a)

We start with a 1D homogeneous diffusion equation with zero Dirichlet conditions:

$$u_t = \alpha u_x x, x \in \Omega = (0, L), \ t \in (0, T], \tag{3.104}$$
$$u(0, t) = u(L, t) = 0, \qquad t \in (0, T], \tag{3.105}$$
$$u(x, 0) = I(x), \qquad x \in [0, L]\,. \tag{3.106}$$

The energy estimate for this problem reads

$$||u|| * *L^2 \leq ||I|| * *L^2, \tag{3.107}$$

where the $||\cdot||_{L^2}$ norm is defined by

$$||g||_{L^2} = \sqrt{\int_0^L g^2 dx}\,. \tag{3.108}$$

The quantify $||u|| * *L^2$ or $\frac{1}{2}||u|| * *L^2$ is known as the *energy* of the solution, although it is not the physical energy of the system. A mathematical tradition has introduced the notion *energy* in this context.

The estimate (3.107) says that the "size of u" never exceeds that of the initial condition, or more precisely, it says that the area under the u curve decreases with time.

To show (3.107), multiply the PDE by u and integrate from 0 to L. Use that uu_t can be expressed as the time derivative of u^2 and that $u_x xu$ can integrated by parts to form an integrand u_x^2. Show that the time derivative of $||u||_{L^2}^2$ must be less than or equal to zero. Integrate this expression and derive (3.107).

b)

Now we address a slightly different problem,

$$u_t = \alpha u_x x + f(x, t), x \in \Omega = (0, L), \ t \in (0, T], \tag{3.109}$$
$$u(0, t) = u(L, t) = 0, \qquad t \in (0, T], \tag{3.110}$$
$$u(x, 0) = 0, \qquad x \in [0, L]\,. \tag{3.111}$$

The associated energy estimate is

$$||u|| * *L^2 \leq ||f|| * *L^2\,. \tag{3.112}$$

(This result is more difficult to derive.)

Now consider the compound problem with an initial condition $I(x)$ and a right-hand side $f(x, t)$:

$$u_t = \alpha u_x x + f(x, t), x \in \Omega = (0, L), \ t \in (0, T], \tag{3.113}$$
$$u(0, t) = u(L, t) = 0, \qquad t \in (0, T], \tag{3.114}$$
$$u(x, 0) = I(x), \qquad x \in [0, L]\,. \tag{3.115}$$

Show that if w_1 fulfills the first problem (with I and $f = 0$) and w_2 fulfills the second problem (with f and $I = 0$), then $u = w_1 + w_2$ is the solution of the compound problem above. Using the triangle inequality for norms,

$$||a + b|| \leq ||a|| + ||b||,$$

show that the energy estimate for the compound problem becomes

$$||u|| * *L^2 \leq ||I|| * *L^2 + ||f||_{L^2}\,. \tag{3.116}$$

c)

One application of (3.116) is to prove uniqueness of the solution. Suppose u_1 and u_2 both fulfill the compound problem. Show that $u = u_1 - u_2$ then fulfills the compound problem with $f = 0$ and $I = 0$. Use (3.116) to deduce that the energy must be zero for all times and therefore that $u_1 = u_2$, which proves that the solution is unique.

d)

Generalize (3.116) to a 2D/3D diffusion equation $u_t = \nabla \cdot (\alpha \nabla u)$ for $x \in \Omega$.

Use integration by parts in multi dimensions:

$$\int_\Omega u\nabla \cdot (\alpha\nabla u)\,\mathrm{d}x = -\int_\Omega \alpha\nabla u \cdot \nabla u\,\mathrm{d}x - \int_{\partial\Omega} u\alpha\frac{\partial u}{\partial n},$$

where $\frac{\partial u}{\partial n} = \boldsymbol{n} \cdot \nabla u$, $\boldsymbol{n}$ being the outward unit normal to the boundary $\partial\Omega$ of the domain Ω.

e)

Now we also consider the multi-dimensional PDE $u_t = \nabla \cdot (\alpha \nabla u)$. Integrate both sides over Ω and use Gauss' divergence theorem, $\int_\Omega \nabla \cdot \boldsymbol{q}\,\mathrm{d}x = \int_{\partial\Omega} \boldsymbol{q} \cdot \boldsymbol{n}\,\mathrm{d}s$ for a vector field $\boldsymbol{q}$. Show that if we have homogeneous Neumann conditions on the boundary, $\partial u/\partial n = 0$, area under the u surface remains constant in time and

$$\int_\Omega u\,\mathrm{d}x = \int_\Omega I\,\mathrm{d}x\,. \tag{3.117}$$

f)

Establish a code in 1D, 2D, or 3D that can solve a diffusion equation with a source term f, initial condition I, and zero Dirichlet or Neumann conditions on the whole boundary.

We can use (3.116) and (3.117) as a partial verification of the code. Choose some functions f and I and check that (3.116) is obeyed at any time when zero Dirichlet conditions are used. Iterate over the same I functions and check that (3.117) is fulfilled when using zero Neumann conditions.

g)

Make a list of some possible bugs in the code, such as indexing errors in arrays, failure to set the correct boundary conditions, evaluation of a term at a wrong time level, and similar. For each of the bugs, see if the verification tests from the previous subexercise pass or fail. This investigation shows how strong the energy estimates and the estimate (3.117) are for pointing out errors in the implementation.

3.70. Exercise: Splitting methods and preconditioning

In Section Section 3.51.1, we outlined a class of iterative methods for $Au = b$ based on splitting A into $A = M - N$ and introducing the iteration

$$Mu^k = Nu^k + b\,.$$

The very simplest splitting is $M = I$, where I is the identity matrix. Show that this choice corresponds to the iteration

$$u^k = u^{k-1} + r^{k-1}, \quad r^{k-1} = b - Au^{k-1}, \tag{3.118}$$

where r^{k-1} is the residual in the linear system in iteration $k-1$. The formula (3.118) is known as Richardson's iteration. Show that if we apply the simple iteration method (3.118) to the *preconditioned* system $M^{-1}Au = M^{-1}b$, we arrive at the Jacobi method by choosing $M = D$ (the diagonal of A) as preconditioner and the SOR method by choosing $M = \omega^{-1}D + L$ (L being the lower triangular part of A). This equivalence shows that we can apply one iteration of the Jacobi or SOR method as preconditioner.

Solution

Inserting $M = I$ and $N = I - A$ in the iterative method leads to

$$u^k = (I - A)u^{k-1} + b = u^{k-1} + (b - Au^{k-1}),$$

which is (3.118). Replacing A by $M^{-1}A$ and b by $M^{-1}b$ in this equation gives

$$u^k = u^{k-1} + M^{-1}r^{k-1}, \quad r^{k-1} = b - Au^{k-1},$$

which we after multiplication by M and reordering can write as

$$Mu^k = (M - A)u^{k-1} + b = Nu^{k-1} + b,$$

which is the standard form for the Jacobi and SOR methods. Choosing $M = D$ gives Jacobi and $M = \omega^{-1}D + L$ gives SOR. We have shown that we may view M as a preconditioner of a simplest possible iteration method.

3.71. Problem: Oscillating surface temperature of the earth

Consider a day-and-night or seasonal variation in temperature at the surface of the earth. How deep down in the ground will the surface oscillations reach? For simplicity, we model only the vertical variation along a coordinate x, where $x = 0$ at the surface, and x increases as we go down in the ground. The temperature is governed by the heat equation

$$\varrho c_v \frac{\partial T}{\partial t} = \nabla \cdot (k\nabla T),$$

in some spatial domain $x \in [0, L]$, where L is chosen large enough such that we can assume that T is approximately constant, independent of the surface oscillations, for $x > L$. The parameters ϱ,

c_v, and k are the density, the specific heat capacity at constant volume, and the heat conduction coefficient, respectively.

a)

Derive the mathematical model for computing $T(x,t)$. Assume the surface oscillations to be sinusoidal around some mean temperature T_m. Let $T = T_m$ initially. At $x = L$, assume $T \approx T_m$.

Solution

The surface temperature is set as

$$T(0,t) = T_m + A\sin(\omega t)\,.$$

With only one "active" spatial coordinate we get the initial-boundary value problem

$$\begin{aligned}
\varrho c_v \frac{\partial T}{\partial t} &= \frac{\partial}{\partial x}\left(k(x)\frac{\partial T}{\partial x}\right), x \in (0,L),\ t \in (0,T],\\
T(x,0) &= T_m, && x \in [0,L],\\
T(0,t) &= T_m + A\sin(\omega t), && t \in (0,T],\\
T(L,t) &= T_m, && t \in (0,T].
\end{aligned}$$

b)

Scale the model in a) assuming k is constant. Use a time scale $t_c = \omega^{-1}$ and a length scale $x_c = \sqrt{2\alpha/\omega}$, where $\alpha = k/(\varrho c_v)$. The primary unknown can be scaled as $\frac{T-T_m}{2A}$.

Show that the scaled PDE is

$$\frac{\partial u}{\partial \bar{t}} = \frac{1}{2}\frac{\partial^2 u}{\partial \bar{x}^2},$$

with initial condition $u(\bar{x},0) = 0$, left boundary condition $u(0,\bar{t}) = \sin(\bar{t})$, and right boundary condition $u(\bar{L},\bar{t}) = 0$. The bar indicates a dimensionless quantity.

Show that $u(\bar{x},\bar{t}) = e^{-\bar{x}}\sin(\bar{x}-\bar{t})$ is a solution that fulfills the PDE and the boundary condition at $\bar{x} = 0$ (this is the solution we will experience as $\bar{t} \to \infty$ and $L \to \infty$). Conclude that an appropriate domain for x is $[0,4]$ if a damping $e^{-4} \approx 0.18$ is appropriate for implementing $\bar{u} \approx$ const; increasing to $[0,6]$ damps $\bar{u}$ to 0.0025.

Solution

Chapter 3.2.4 in the book (Langtangen and Pedersen 2016) describes the scaling of this problem in detail. Inserting dimensionless variables $\bar{t} = \omega t$, $\bar{x} = \sqrt{\omega/(2\alpha)}x$, and

$$u = \frac{T - T_m}{2A},$$

leads to

$$
\begin{aligned}
\frac{\partial u}{\partial \bar{t}} &= \frac{1}{2}\frac{\partial^2 u}{\partial x^2}, \quad \bar{x} \in (0, \bar{L}),\ \bar{t} \in (0, \bar{T}], \\
u(\bar{x}, 0) &= 0, && \bar{x} \in [0, 1], \\
u(0, \bar{t}) &= \sin(\bar{t}), && \bar{t} \in (0, \bar{T}], \\
u(\bar{L}, \bar{t}) &= 0, && \bar{t} \in (0, \bar{T}].
\end{aligned}
$$

The domain lengths $\bar{L}$ and $\bar{T}$ follows from straightforward scaling of L and T.
Inserting $u(\bar{x}, \bar{t}) = e^{-\bar{x}} \sin(\bar{t} - \bar{x})$ in the PDE shows that this is a solution. It also obeys the boundary condition $\bar{u}(0, \bar{t}) = sin(\bar{t})$. As $\bar{t} \to \infty$, the initial condition has no longer impact on the solution and is "forgotten" and of no interest. The boundary condition at $\bar{x} = \bar{L}$ is never compatible with the given solution unless $\bar{u}$ is damped to zero, which happens mathematically as $\bar{L} \to \infty$. For a numerical solution, however, we may use a small finite value such as $\bar{L} = 4$.

c)

Compute the scaled temperature and make animations comparing two solutions with $\bar{L} = 4$ and $\bar{L} = 8$, respectively (keep Δx the same).

Solution

We can use the `viz` function in `diff1D_vc.py` to do the number crunching. Appropriate calls and visualization go here:

```
import os
import sys

sys.path.insert(0, os.path.join(os.pardir, "src-diffu"))
from diffu1D_vc import viz

sol = []  # store solutions
for Nx, L in [[20, 4], [40, 8]]:
    dt = 0.1
    dx = float(L) / Nx
    D = dt / dx**2
    from math import pi, sin

    T = 2 * pi * 6
    from numpy import zeros

    a = zeros(Nx + 1) + 0.5
    cpu, u_ = viz(
        I=lambda x: 0,
        a=a,
        L=L,
        Nx=Nx,
        D=D,
        T=T,
        umin=-1.1,
        umax=1.1,
        theta=0.5,
        u_L=lambda t: sin(t),
        u_R=0,
        animate=False,
        store_u=True,
    )
    sol.append(u_)
    print("computed solution for Nx=%d in [0,%g]" % (Nx, L))

print(sol[0].shape)
print(sol[1].shape)
import matplotlib.pyplot as plt

counter = 0
for u0, u1 in zip(sol[0][2:], sol[1][2:], strict=False):
    x0 = sol[0][0]
    x1 = sol[1][0]
    plt.clf()
    plt.plot(x0, u0, "r-", label="short")
    plt.plot(x1, u1, "b-", label="long")
    plt.legend()
    plt.axis([x1[0], x1[-1], -1.1, 1.1])
    plt.savefig("tmp_%04d.png" % counter)
    counter += 1
```

MOVIE: [https://github.com/hplgit/fdm-book/raw/master/doc/pub/book/html/mov-diffu/surface_osc/movie.mp4]

3.72. Problem: Oscillating and pulsating flow in tubes

We consider flow in a straight tube with radius R and straight walls. The flow is driven by a pressure gradient $\beta(t)$. The effect of gravity can be neglected. The mathematical problem reads

$$\varrho\frac{\partial u}{\partial t} = \mu\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \beta(t), \quad r\in[0,R],\ t\in(0,T], \tag{3.119}$$

$$u(r,0) = I(r), \qquad r\in[0,R], \tag{3.120}$$

$$u(R,t) = 0, \qquad t\in(0,T], \tag{3.121}$$

$$\frac{\partial u}{\partial r}(0,t) = 0, \qquad t\in(0,T]. \tag{3.122}$$

We consider two models for $\beta(t)$. One plain, sinusoidal oscillation:

$$\beta = A\sin(\omega t),$$

and one with periodic pulses,

$$\beta = A\sin^{16}(\omega t),$$

Note that both models can be written as $\beta = A\sin^m(\omega t)$, with $m = 1$ and $m = 16$, respectively.

a)

Scale the mathematical model, using the viscous time scale $\varrho R^2/\mu$.

Solution

We can introduce

$$\bar{r} = \frac{r}{R}, \quad \bar{t} = \frac{t}{\varrho R^2/\mu}, \quad u = \frac{u}{u_c}\,.$$

Inserted in the PDE, we get

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \frac{1}{\bar{r}}\frac{\partial}{\partial \bar{r}}\left(\bar{r}\frac{\partial \bar{u}}{\partial \bar{r}}\right) + \frac{R^2A}{u_c\mu}\sin^m(\alpha\bar{t})$$

where α is a dimensionless number

$$\alpha = \frac{\omega\varrho R^2}{\mu} = \frac{\varrho R^2/\mu}{1/\omega},$$

reflecting the ratio of the viscous diffusion time scale and the time scale of the oscillating pressure gradient. We may choose u_c such that the coefficient in the pressure gradient term equals unity:

$$u_c = \frac{R^2A}{\mu}\,.$$

The governing PDE, dropping the bars, then reads

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \sin^m(\alpha\bar{t}), \quad r \in (0,1),\ t \in (0,T]\,.$$

b)

Implement the scaled model from a), using the unifying θ scheme in time and centered differences in space.

Solution

We need to take into account extensions below: a coefficient in front of the viscous term, and an extra source term.
A preliminary and unfinished code:

```
import time

import scipy.sparse
import scipy.sparse.linalg
import sympy as sym
from numpy import linspace, log, ones, sqrt, sum, zeros


def solver_theta(I, a, R, Nr, D, T, theta=0.5, u_L=None, u_R=0, user_action=None, f=0):
    """
    Solve the diffusion equation for axi-symmetric case:
        u_t = 1/r * (r*a(r)*u_r)_r + f(r,t)
    on (0,R) with boundary conditions u(0,t)_r = 0 and u(R,t) = 0.

    Method: (implicit) theta-rule in time.
    D = dt/dr**2 and implicitly specifies the time step.
    u_L = None implies du/dr = 0, i.e. a symmetry condition.
    """
    t0 = time.perf_counter()

    r = linspace(0, R, Nr + 1)  # mesh points in space
    dr = r[1] - r[0]
    dt = D * dr**2
    Nt = int(round(T / float(dt)))
    t = linspace(0, T, Nt + 1)  # mesh points in time

    if isinstance(u_L, (float, int)):
        u_L_ = float(u_L)
        u_L = lambda t: u_L
```

c)

Verify the implementation in b) using a manufactured solution that is quadratic in r and linear in t. Make a corresponding test function.

You need to include an extra source term

in the equation to allow for such tests. Let the spatial variation be $1 - r^2$ such that the boundary condition is fulfilled.

d)

Make animations for $m = 1, 16$ and $\alpha = 1, 0.1$. Choose T such that the motion has reached a steady state (non-visible changes from period to period in u).

e)

For $\alpha \gg 1$, the scaling in a) is not good, because the characteristic time for changes (due to the pressure) is much smaller than the viscous diffusion time scale (α becomes large). We should in this case base the short time scale on $1/\omega$. Scale the model again, and make an animation for $m = 1, 16$ and $\alpha = 10$.

Solution

Now the governing PDE becomes

$$\frac{\partial u}{\partial t} = \alpha^{-1}\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \sin^m t, \quad r \in (0,1),\ t \in (0,T]\,.$$

In this case,

$$u_c = \frac{A}{\varrho\omega}\,.$$

We see that for $\alpha \gg 1$, we can neglect the viscous term, and we basically have a balance between the acceleration and the driving pressure gradient:

$$\frac{\partial u}{\partial t} = \sin^m t\,.$$

3.73. Problem: Scaling a welding problem

Welding equipment makes a very localized heat source that moves in time. We shall investigate the heating due to welding and choose, for maximum simplicity, a one-dimensional heat equation with a fixed temperature at the ends, and we neglect melting. We shall scale the problem, and besides solving such a problem numerically, the aim is to investigate the appropriateness of alternative scalings.

The governing PDE problem reads

$$\begin{aligned} \varrho c\frac{\partial u}{\partial t} &= k\frac{\partial^2 u}{\partial x^2} + f, & x \in (0,L),\ t \in (0,T),\\ u(x,0) &= U_s, & x \in [0,L],\\ u(0,t) = u(L,t) &= 0, & t \in (0,T]. \end{aligned}$$

Here, u is the temperature, ϱ the density of the material, c a heat capacity, k the heat conduction coefficient, f is the heat source from the welding equipment, and U_s is the initial constant (room) temperature in the material.

A possible model for the heat source is a moving Gaussian function:

$$f = A\exp\left(-\frac{1}{2}\left(\frac{x-vt}{\sigma}\right)^2\right),$$

where A is the strength, σ is a parameter governing how peak-shaped (or localized in space) the heat source is, and v is the velocity (in positive x direction) of the source.

a)

Let x_c, t_c, u_c, and f_c be scales, i.e., characteristic sizes, of x, t, u, and f, respectively. The natural choice of x_c and f_c is L and A, since these make the scaled x and f in the interval $[0,1]$. If each of the three terms in the PDE are equally important, we can find t_c and u_c by demanding that the coefficients in the scaled PDE are all equal to unity. Perform this scaling. Use scaled quantities in the arguments for the exponential function in f too and show that

$$\bar{f} = e^{-\frac{1}{2}\beta^2(\bar{x}-\gamma\bar{t})^2},$$

where β and γ are dimensionless numbers. Give an interpretation of β and γ.

Solution

We introduce

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{t}{t_c}, \quad \bar{u} = \frac{u-U_s}{u_c}, \quad \bar{f} = \frac{f}{A}.$$

Inserted in the PDE and dividing by $\varrho c u_c/t_c$ such that the coefficient in front of $\partial\bar{u}/\partial\bar{t}$ becomes unity, and thereby all terms become dimensionless, we get

$$\frac{\partial\bar{u}}{\partial\bar{t}} = \frac{kt_c}{\varrho cL^2}\frac{\partial^2\bar{u}}{\partial\bar{x}^2} - \frac{At_c}{\varrho cu_c}\bar{f}.$$

Demanding that all three terms are equally important, it follows that

$$\frac{kt_c}{\varrho cL^2} = 1, \quad \frac{At_c}{\varrho cu_c} = 1.$$

These constraints imply the *diffusion time scale*

$$t_c = \frac{\varrho cL^2}{k},$$

and a scale for u_c,

$$u_c = \frac{AL^2}{k}.$$

The scaled PDE reads

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \frac{\partial^2 \bar{u}}{\partial \bar{x}^2} - \bar{f}\,.$$

Scaling f results in

$$\begin{aligned}\bar{f} &= \exp\left(-\frac{1}{2}\left(\frac{x-vt}{\sigma}\right)^2\right)\\ &= \exp\left(-\frac{1}{2}\frac{L^2}{\sigma^2}\left(\bar{x}-\frac{vt_c}{L}t\right)^2\right)\\ &= \exp\left(-\frac{1}{2}\beta^2\left(\bar{x}-\gamma\bar{t}\right)^2\right),\end{aligned}$$

where β and γ are dimensionless numbers:

$$\beta = \frac{L}{\sigma}, \quad \gamma = \frac{vt_c}{L} = \frac{v\varrho cL}{k}\,.$$

The σ parameter measures the width of the Gaussian peak, so β is the ratio of the domain and the width of the heat source (large β implies a very peak-formed heat source). The γ parameter arises from $t_c/(L/v)$, which is the ratio of the diffusion time scale and the time it takes for the heat source to travel through the domain. Equivalently, we can multiply by t_c/t_c to get $\gamma = v/(t_c L)$ as the ratio between the velocity of the heat source and the diffusion velocity.

b)

Argue that for large γ we should base the time scale on the movement of the heat source. Show that this gives rise to the scaled PDE

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \gamma^{-1}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2} - \bar{f},$$

and

$$\bar{f} = \exp\left(-\frac{1}{2}\beta^2(\bar{x}-\bar{t})^2\right).$$

Discuss when the scalings in a) and b) are appropriate.

Solution

We perform the scaling as in a), but this time we determine t_c such that the heat source moves with unit velocity. This means that

$$\frac{vt_c}{L} = 1 \quad \Rightarrow \quad t_c = \frac{L}{v}\,.$$

Scaling of the PDE gives, as before,

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \frac{kt_c}{\varrho cL^2}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2} - \frac{At_c}{\varrho cu_c}\bar{f}\,.$$

Inserting the expression for t_c, we have

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \frac{kL}{\varrho c L^2 v}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2} - \frac{AL}{v\varrho c u_c}\bar{f}\,.$$

We recognize the first coefficient as γ^{-1}, while u_c can be determined from demanding the second coefficient to be unity:

$$u_c = \frac{AL}{v\varrho c}\,.$$

The scaled PDE is therefore

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \gamma^{-1}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2} - \bar{f}\,.$$

If the heat source moves very fast, there is little time for the diffusion to transport the heat away from the source, and the heat conduction term becomes insignificant. This is reflected in the coefficient γ^{-1}, which is small when γ, the ratio of the heat source velocity and the diffusion velocity, is large.
The scaling in a) is therefore appropriate if diffusion is a significant process, i.e., the welding equipment moves at a slow speed so heat can efficiently spread out by diffusion. For large γ, the scaling in b) is appropriate, and $t = 1$ corresponds to having the heat source traveled through the domain (with the scaling in a), the heat source will leave the domain in short time).

c)

One aim with scaling is to get a solution that lies in the interval $[-1, 1]$. This is not always the case when u_c is based on a scale involving a source term, as we do in a) and b). However, from the scaled PDE we realize that if we replace $\bar{f}$ with $\delta\bar{f}$, where δ is a dimensionless factor, this corresponds to replacing u_c by u_c/δ. So, if we observe that $\bar{u} \sim 1/\delta$ in simulations, we can just replace $\bar{f}$ by $\delta\bar{f}$ in the scaled PDE.

Use this trick and implement the two scaled models. Reuse software for the diffusion equation (e.g., the `solver` function in `diffu1D_vc.py`). Make a function `run(gamma, beta=10, delta=40, scaling=1, animate=False)` that runs the model with the given γ, β, and δ parameters as well as an indicator `scaling` that is 1 for the scaling in a) and 2 for the scaling in b). The last argument can be used to turn screen animations on or off.

Experiments show that with $\gamma = 1$ and $\beta = 10$, $\delta = 20$ is appropriate. Then $\max|\bar{u}|$ will be larger than 4 for $\gamma = 40$, but that is acceptable.

Equip the `run` function with visualization, both animation of $\bar{u}$ and $\bar{f}$, and plots with $\bar{u}$ and $\bar{f}$ for $t = 0.2$ and $t = 0.5$.

Since the amplitudes of $\bar{u}$ and $\bar{f}$ differs by a factor δ,

it is attractive to plot $\bar{f}/\delta$ together with $\bar{u}$.

Solution

Here is a possible `run` function:

```
import os
import sys

sys.path.insert(0, os.path.join(os.pardir, "src-diffu"))
import numpy as np
from diffu1D_vc import solver


def run(gamma, beta=10, delta=40, scaling=1, animate=False):
    """Run the scaled model for welding."""
    if scaling == 1:
        v = gamma
        a = 1
    elif scaling == 2:
        v = 1
        a = 1.0 / gamma

    b = 0.5 * beta**2
    L = 1.0
    ymin = 0
    global ymax
    ymax = 1.2

    I = lambda x: 0
    f = lambda x, t: delta * np.exp(-b * (x - v * t) ** 2)

    import time

    import matplotlib.pyplot as plt

    plot_arrays = []

    def process_u(u, x, t, n):
        global ymax
        if animate:
            plt.clf()
            plt.plot(x, u, "r-", x, f(x, t[n]) / delta, "b-")
            plt.axis([0, L, ymin, ymax])
            plt.title(f"t={t[n]:f}")
            plt.xlabel("x")
            plt.ylabel(f"u and f/{delta:g}")
            plt.draw()
            plt.pause(0.001)
        if t[n] == 0:
            time.sleep(1)
            plot_arrays.append(x)
        dt = t[1] - t[0]
        tol = dt / 10.0
        if abs(t[n] - 0.2) < tol or abs(t[n] - 0.5) < tol:
            plot_arrays.append((u.copy(), f(x, t[n]) / delta))
            if u.max() > ymax:
                ymax = u.max()
```

Note that we have dropped the bar notation in the plots. It is common to drop the bars as soon as the scaled problem is established.

d)

Use the software in c) to investigate $\gamma = 0.2, 1, 5, 40$ for the two scalings. Discuss the results.

Solution

For these investigations, we compare the two scalings for each of the different γ values. An appropriate function for automating the tasks is

```
def investigate():
    """Do scientific experiments with the run function above."""
    import glob

    # Clean up old files
    for filename in glob.glob("tmp1_gamma*") + glob.glob("welding_gamma*"):
        os.remove(filename)

    gamma_values = 1, 40, 5, 0.2, 0.025
    for gamma in gamma_values:
        for scaling in 1, 2:
            run(gamma=gamma, beta=10, delta=20, scaling=scaling)

    # Combine images
    for gamma in gamma_values:
        for ext in "pdf", "png":
            cmd = (
                "montage "
                "tmp1_gamma{gamma:g}_s1.{ext} "
                "tmp1_gamma{gamma:g}_s2.{ext} "
                "-tile 2x1 -geometry +0+0 "
                "welding_gamma{gamma:g}.{ext}".format(**vars())
            )
            os.system(cmd)
            # pdflatex doesn't like 0.2 in filenames...
            if "." in str(gamma):
                os.rename(
                    "welding_gamma{gamma:g}.{ext}".format(**vars()),
                    ("welding_gamma{gamma:g}".format(**vars())).replace(".", "_")
                    + "."
                    + ext,
                )
```

We run here a Backward Euler scheme with $N_x = 100$ and quite long time steps.
Running the `investigate` function, we get the following plots:

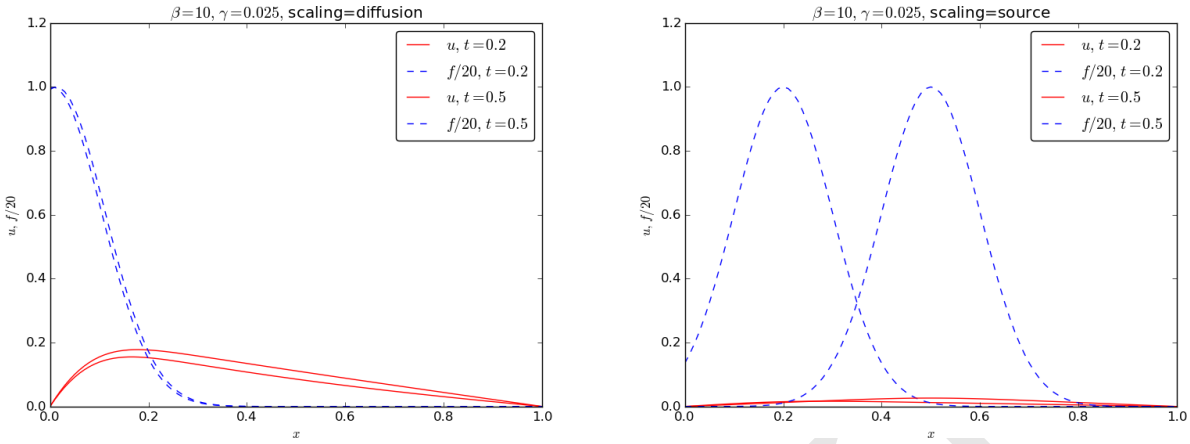


Figure 3.24.: FIGURE: [fig-diffu/welding_gamma0_2, width=800 frac=1]

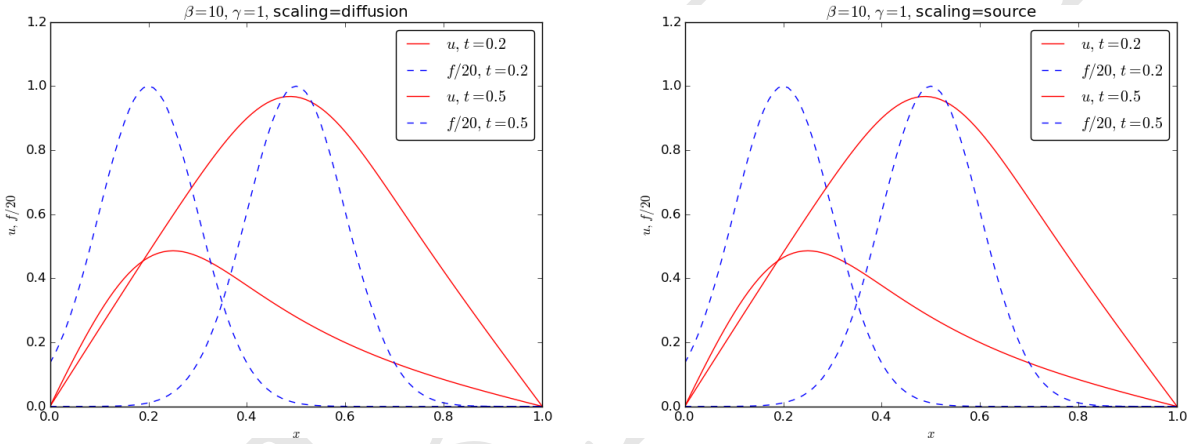


Figure 3.25.: FIGURE: [fig-diffu/welding_gamma5, width=800 frac=1]

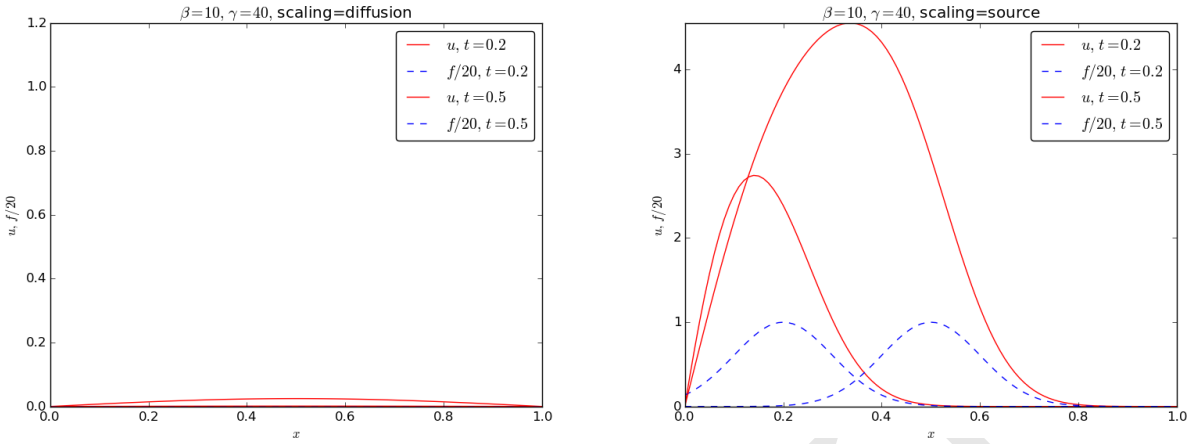


Figure 3.26.: For $\gamma \ll 1$ as in $\gamma = 0.025$, the heat source moves very slowly on the diffusion time scale and has hardly entered the medium, while the scaling in b) is not inappropriate, but a larger δ is needed to bring $\bar{u}$ around unity. We see that for $\gamma = 0.2$, each of the scalings work, but with the diffusion time scale, the heat source has not moved much into the domain. For $\gamma = 1$, the mathematical problems are identical and hence the plots too. For $\gamma = 5$, the time scale based on the source is clearly the best choice, and for $\gamma = 40$, only this scale is appropriate.

A conclusion is that the scaling in b) works well for a range of γ values, also in the case $\gamma \ll 1$.

3.74. Exercise: Implement a Forward Euler scheme for axi-symmetric diffusion

Based on the discussion in Section Section 3.35, derive in detail the discrete equations for a Forward Euler in time, centered in space, finite difference method for axi-symmetric diffusion. The diffusion coefficient may be a function of the radial coordinate. At the outer boundary $r = R$, we may have either a Dirichlet or Robin condition. Implement this scheme. Construct appropriate test problems.

Solution

We start with the equation at $r = 0$. According to Section Section 3.35, we get

$$\frac{u_0^{n+1} - u_0^n}{\Delta t} = 4\alpha(0)\frac{u_1^n - u_0^n}{\Delta r^2} - f_0^n\,.$$

For $i > 0$, we have

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{r_i\Delta r^2}(\frac{1}{2}(r_i + r_{i+1})\frac{1}{2}(\alpha_i + \alpha_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(r_{i-1} + r_i)\frac{1}{2}(\alpha_{i-1} + \alpha_i)(u_i^n - u_{i-1}^n)) + f_i^n$$

Solving with respect to u_i^{n+1} and introducing $D = \Delta t/\Delta r^2$ results in

$$
\begin{aligned}
u_0^{n+1} &= u_0^n + 4D\alpha(0)(u_1^n - u_0^n) + f_0^n, \\
u_i^{n+1} &= u_i^n + D\frac{1}{r_i}(\frac{1}{2}(r_i + r_{i+1})\frac{1}{2}(\alpha_i + \alpha_{i+1})(u_{i+1}^n - u_i^n) - \\
&\quad \frac{1}{2}(r_{i-1} + r_i)\frac{1}{2}(\alpha_{i-1} + \alpha_i)(u_i^n - u_{i-1}^n)) + \Delta t f_i^n, \\
&\quad i = 1, \ldots, N_r - 1,
\end{aligned}
$$

and u_i^{n+1} at the end point $i = N_r$ is assumed known in case of a Dirichlet condition. A Robin condition

$$-\alpha\frac{\partial u}{\partial n} = h_T(u - U_s),$$

can be discretized at $i = N_r$ by

$$-\alpha_i\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta r} = h_T(u_i^n - U_s)\,.$$

Solving with respect to the value at the fictitious point $i + 1$ gives

$$u_{i+1}^n = u_{i-1}^n - 2\Delta r\frac{h_T}{\alpha_i}(u_i^n - U_s)\,.$$

This value is then inserted for u_{i+1}^n in the discrete PDE at $i = N_r$.

3.75. Exercises: Diffusion with Devito

These exercises explore the diffusion equation using Devito's symbolic finite difference framework.

3.75.1. Exercise 1: Verify the Fourier Stability Limit

The Forward Euler scheme for the diffusion equation requires $F \leq 0.5$ for stability.

a) Use `solve_diffusion_1d` with $F = 0.5$ and verify that the solution decays smoothly.
b) Try $F = 0.51$ and observe what happens.
c) Plot the solution at several time steps for both cases.

Solution

```
from src.diffu import solve_diffusion_1d
import numpy as np
import matplotlib.pyplot as plt

# Stable case: F = 0.5
result_stable = solve_diffusion_1d(
    L=1.0, a=1.0, Nx=50, T=0.1, F=0.5,
    save_history=True,
)

# Unstable case: F = 0.51
# Note: The solver will raise a ValueError for F > 0.5
# To demonstrate instability, we would need to bypass the check
# or use the legacy NumPy implementation

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
for i in [0, 5, 10, 20]:
    if i < len(result_stable.t_history):
        plt.plot(result_stable.x, result_stable.u_history[i],
                 label=f't = {result_stable.t_history[i]:.3f}')
plt.xlabel('x')
plt.ylabel('u')
plt.title('Stable: F = 0.5')
plt.legend()

# The F > 0.5 case shows exponential growth with oscillations
plt.subplot(1, 2, 2)
plt.text(0.5, 0.5, 'F > 0.5 causes instability:\n'
         'Solution grows exponentially\nwith oscillations',
         ha='center', va='center', fontsize=12)
plt.title('Unstable: F > 0.5')
plt.tight_layout()
```

3.75.2. Exercise 2: Convergence Rate Verification

Verify that the Forward Euler scheme achieves second-order spatial convergence when the Fourier number F is held fixed.

a) Use grid sizes $N_x = 10, 20, 40, 80, 160$.
b) Compute the L^2 error against the exact sinusoidal solution.
c) Plot the error vs. grid spacing on a log-log scale.
d) Compute the observed convergence rate.

Solution

```
from src.diffu import solve_diffusion_1d, exact_diffusion_sine
import numpy as np
import matplotlib.pyplot as plt

grid_sizes = [10, 20, 40, 80, 160]
errors = []
L = 1.0
a = 1.0
T = 0.1
F = 0.5

for Nx in grid_sizes:
    result = solve_diffusion_1d(L=L, a=a, Nx=Nx, T=T, F=F)
    u_exact = exact_diffusion_sine(result.x, result.t, L, a)
    error = np.sqrt(np.mean((result.u - u_exact)**2))
    errors.append(error)
    print(f"Nx = {Nx:3d}, error = {error:.4e}")

# Compute convergence rate
errors = np.array(errors)
dx = L / np.array(grid_sizes)
log_dx = np.log(dx)
log_err = np.log(errors)
rate = np.polyfit(log_dx, log_err, 1)[0]

print(f"\nObserved convergence rate: {rate:.2f}")
print(f"Expected rate: 2.0")

# Plot
plt.figure(figsize=(8, 6))
plt.loglog(dx, errors, 'bo-', label=f'Observed (rate={rate:.2f})')
plt.loglog(dx, errors[0]*(dx/dx[0])**2, 'r--', label='O(dx^2)')
plt.xlabel('Grid spacing dx')
plt.ylabel('L2 error')
plt.legend()
plt.title('Convergence of Forward Euler for Diffusion')
plt.grid(True)
```

3.75.3. Exercise 3: Gaussian Initial Condition

Study the diffusion of a Gaussian temperature profile.

a) Set up a Gaussian initial condition centered at $x = L/2$ with width $\sigma = 0.05$.
b) Simulate for $T = 0.5$ and visualize the spreading.

c) Show that the total "heat content" (integral of u) is conserved over time (with homogeneous Neumann BCs) or decreases (with Dirichlet BCs).

Solution

```python
from src.diffu import solve_diffusion_1d, gaussian_initial_condition
import numpy as np
import matplotlib.pyplot as plt

result = solve_diffusion_1d(
    L=1.0, a=1.0, Nx=100, T=0.5, F=0.5,
    I=lambda x: gaussian_initial_condition(x, L=1.0, sigma=0.05),
    save_history=True,
)

# Plot evolution
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
times = [0, 0.05, 0.1, 0.2, 0.5]
for t in times:
    idx = int(t / result.dt)
    if idx < len(result.t_history):
        plt.plot(result.x, result.u_history[idx],
                 label=f't = {result.t_history[idx]:.2f}')
plt.xlabel('x')
plt.ylabel('u')
plt.title('Gaussian Diffusion')
plt.legend()

# Heat content over time (with Dirichlet BCs, heat is lost at boundaries)
plt.subplot(1, 2, 2)
dx = result.x[1] - result.x[0]
heat_content = [np.trapz(result.u_history[i], result.x)
                for i in range(len(result.t_history))]
plt.plot(result.t_history, heat_content)
plt.xlabel('Time')
plt.ylabel('Total heat content')
plt.title('Heat Loss Through Boundaries')
plt.tight_layout()
```

With Dirichlet BCs ($u = 0$ at boundaries), heat flows out and the total decreases. With Neumann BCs (insulated boundaries), total heat would be conserved.

3.75.4. Exercise 4: Discontinuous Initial Condition

The diffusion equation smooths out discontinuities over time.

a) Use a "plug" initial condition (1 for $|x - L/2| < 0.1$, 0 otherwise).
b) Compare the solution for $F = 0.5$ and $F = 0.25$.
c) Observe the oscillations (Gibbs phenomenon) for $F = 0.5$.

Solution

```
from src.diffu import solve_diffusion_1d, plug_initial_condition
import numpy as np
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for ax, F in zip(axes, [0.5, 0.25]):
    result = solve_diffusion_1d(
        L=1.0, a=1.0, Nx=100, T=0.1, F=F,
        I=lambda x: plug_initial_condition(x, L=1.0, width=0.1),
        save_history=True,
    )

    times = [0, 0.01, 0.02, 0.05, 0.1]
    for t in times:
        idx = int(t / result.dt)
        if idx < len(result.t_history):
            ax.plot(result.x, result.u_history[idx],
                    label=f't = {result.t_history[idx]:.3f}')

    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(f'Plug Diffusion (F = {F})')
    ax.legend()

plt.tight_layout()
```

At $F = 0.5$, oscillations appear near the discontinuity (numerical Gibbs phenomenon). At $F = 0.25$, the solution is smoother but the simulation takes more time steps.

3.75.5. Exercise 5: 2D Heat Diffusion

Simulate heat diffusion in a 2D square domain.

a) Set up a Gaussian "hot spot" centered at $(0.5, 0.5)$.
b) Apply $u = 0$ on all boundaries (heat sink).
c) Visualize the temperature distribution at several times.

d) Compute the decay rate of the maximum temperature.

Solution

```
from src.diffu import solve_diffusion_2d, gaussian_2d_initial_condition
import numpy as np
import matplotlib.pyplot as plt

result = solve_diffusion_2d(
    Lx=1.0, Ly=1.0, a=1.0, Nx=50, Ny=50, T=0.2, F=0.25,
    I=lambda X, Y: gaussian_2d_initial_condition(X, Y, 1.0, 1.0, sigma=0.1),
    save_history=True,
)

# Plot at several times
fig, axes = plt.subplots(2, 3, figsize=(12, 8))
X, Y = np.meshgrid(result.x, result.y, indexing='ij')

times = [0, 0.04, 0.08, 0.12, 0.16, 0.2]
for ax, t in zip(axes.flat, times):
    idx = int(t / result.dt)
    if idx >= len(result.t_history):
        idx = -1
    c = ax.contourf(X, Y, result.u_history[idx], levels=20, cmap='hot')
    ax.set_title(f't = {result.t_history[idx]:.3f}')
    ax.set_aspect('equal')

plt.tight_layout()

# Maximum temperature decay
max_temps = [result.u_history[i].max() for i in range(len(result.t_history))]
plt.figure()
plt.semilogy(result.t_history, max_temps)
plt.xlabel('Time')
plt.ylabel('Maximum temperature')
plt.title('Exponential Decay of Peak Temperature')
plt.grid(True)
```

3.75.6. Exercise 6: Variable Diffusion Coefficient

In heterogeneous materials, the diffusion coefficient varies in space.

a) Modify the solver to accept a spatially varying $\alpha(x)$.
b) Set up a two-layer problem: $\alpha = 1$ for $x < L/2$, $\alpha = 0.1$ for $x > L/2$.
c) Observe how heat diffuses differently in the two regions.

Hint: In Devito, use a `Function` instead of a `Constant` for the diffusion coefficient.

Solution

```
from devito import Grid, TimeFunction, Function, Eq, solve, Operator
import numpy as np
import matplotlib.pyplot as plt

# Setup
L = 2.0
Nx = 200
T = 0.5
grid = Grid(shape=(Nx + 1,), extent=(L,))

# Variable diffusion coefficient
a = Function(name='a', grid=grid)
x_coords = np.linspace(0, L, Nx + 1)
a.data[:] = np.where(x_coords < L/2, 1.0, 0.1)

# Temperature field
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

# Initial condition: Gaussian in left region
sigma = 0.1
x0 = 0.5
u.data[0, :] = np.exp(-((x_coords - x0) / sigma)**2)

# PDE: u_t = a(x) * u_xx
# Note: Using variable coefficient
pde = u.dt - a * u.dx2
stencil = Eq(u.forward, solve(pde, u.forward))

# Stability: use max(a) for dt calculation
dx = L / Nx
F = 0.5
dt = F * dx**2 / a.data.max()
Nt = int(T / dt)

# Boundary conditions
bc_left = Eq(u[grid.stepping_dim + 1, 0], 0)
bc_right = Eq(u[grid.stepping_dim + 1, Nx], 0)

op = Operator([stencil, bc_left, bc_right])

# Time stepping with history
history = [u.data[0, :].copy()]
times = [0]

for n in range(Nt):
    op.apply(time_m=0, time_M=0, dt=dt)
    u.data[0, :] = u.data[1, :]
    if n % 100 == 0:
        history.append(u.data[0, :].copy())
        times.append((n + 1) * dt)

# Plot
plt.figure(figsize=(10, 6))
```

Heat diffuses quickly in the left region ($\alpha = 1$) but slowly in the right region ($\alpha = 0.1$). The solution shows a discontinuity in the temperature gradient at the interface.

3.75.7. Exercise 7: Manufactured Solution

Verify the implementation using the Method of Manufactured Solutions.

a) Choose a solution $u(x, t) = x(L - x) \cdot t$.
b) Compute the source term $f(x, t)$ needed to make this satisfy $u_t = \alpha u_{xx} + f$.
c) Verify that the numerical solution matches the manufactured solution to machine precision.

Solution

```
import sympy as sp

# Define symbolic variables
x_sym, t_sym, a_sym, L_sym = sp.symbols('x t a L')

# Manufactured solution
u_mms = x_sym * (L_sym - x_sym) * t_sym

# Compute required source term
u_t = sp.diff(u_mms, t_sym)
u_xx = sp.diff(u_mms, x_sym, 2)
f_sym = u_t - a_sym * u_xx

print(f"Manufactured solution: u = {u_mms}")
print(f"Source term: f = {sp.simplify(f_sym)}")

# f = x*(L-x) - a*(-2)*t = x*(L-x) + 2*a*t

# Numerical verification
from devito import Grid, TimeFunction, Eq, solve, Operator, Constant
import numpy as np

L = 1.5
Nx = 20
a = 0.5
T = 0.2

dx = L / Nx
F = 0.5
dt = F * dx**2 / a
Nt = int(T / dt)

grid = Grid(shape=(Nx + 1,), extent=(L,))
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

x_coords = np.linspace(0, L, Nx + 1)

# Source term as a function
def f_source(x, t):
    return x * (L - x) + 2 * a * t

# Exact solution
def u_exact(x, t):
    return x * (L - x) * t

# Initial condition (t=0 gives u=0)
u.data[0, :] = u_exact(x_coords, 0)

# Include source term in the PDE (simplified for Forward Euler)
# Manual time stepping with source
for n in range(Nt):
    t_n = n * dt
    u_new = (u.data[0, 1:-1] +
```

The Forward Euler scheme is exact for solutions linear in time and quadratic in space, so the error should be near machine precision.

3.75.8. Exercise 8: Energy Decay

The "energy" of the diffusion equation, defined as:

$$E(t) = \frac{1}{2}\int_0^L u^2\,dx$$

always decreases for the diffusion equation (with homogeneous BCs).

a) Compute $E(t)$ numerically at each time step.
b) Verify that $E(t)$ is monotonically decreasing.
c) Compare the decay rate to the theoretical prediction for the fundamental mode: $E(t) \propto e^{-2\alpha(\pi/L)^2 t}$.

Solution

```
from src.diffu import solve_diffusion_1d
import numpy as np
import matplotlib.pyplot as plt

result = solve_diffusion_1d(
    L=1.0, a=1.0, Nx=100, T=1.0, F=0.5,
    I=lambda x: np.sin(np.pi * x),  # Fundamental mode
    save_history=True,
)

# Compute energy at each time step
dx = result.x[1] - result.x[0]
energies = []
for u_n in result.u_history:
    E = 0.5 * np.trapz(u_n**2, result.x)
    energies.append(E)

energies = np.array(energies)

# Theoretical decay: E(t) = E(0) * exp(-2*a*(pi/L)^2 * t)
L = 1.0
a = 1.0
decay_rate = 2 * a * (np.pi / L)**2
E_theory = energies[0] * np.exp(-decay_rate * result.t_history)

# Plot
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.semilogy(result.t_history, energies, 'b-', label='Numerical')
plt.semilogy(result.t_history, E_theory, 'r--', label='Theory')
plt.xlabel('Time')
plt.ylabel('Energy E(t)')
plt.legend()
plt.title('Energy Decay')

plt.subplot(1, 2, 2)
# Verify monotonic decrease
dE = np.diff(energies)
plt.plot(result.t_history[1:], dE)
plt.axhline(0, color='k', linestyle='--')
plt.xlabel('Time')
plt.ylabel('dE/dt')
plt.title('Energy Change (should be < 0)')
plt.tight_layout()

# Compute observed decay rate
log_E = np.log(energies[energies > 0])
t_fit = result.t_history[:len(log_E)]
rate_obs = -np.polyfit(t_fit, log_E, 1)[0]
print(f"Observed decay rate: {rate_obs:.4f}")
print(f"Theoretical rate: {decay_rate:.4f}")
```

3.75.9. Exercise 9: 2D Convergence Test

Verify second-order convergence for the 2D diffusion solver.

a) Use the exact 2D sinusoidal solution.
b) Run with $N_x = N_y = 10, 20, 40, 80$.
c) Compute the observed convergence rate.

Solution

```
from src.diffu import convergence_test_diffusion_2d
import numpy as np
import matplotlib.pyplot as plt

grid_sizes, errors, rate = convergence_test_diffusion_2d(
    grid_sizes=[10, 20, 40, 80],
    T=0.05,
    F=0.25,
)

print(f"Observed convergence rate: {rate:.2f}")

# Plot
plt.figure(figsize=(8, 6))
dx = 1.0 / np.array(grid_sizes)
plt.loglog(dx, errors, 'bo-', label=f'Observed (rate={rate:.2f})')
plt.loglog(dx, errors[0]*(dx/dx[0])**2, 'r--', label='O(dx^2)')
plt.xlabel('Grid spacing')
plt.ylabel('L2 error')
plt.legend()
plt.title('2D Diffusion Convergence')
plt.grid(True)
```

The 2D solver should also achieve second-order spatial convergence when the Fourier number is held fixed.

3.75.10. Exercise 10: Comparison with Legacy Code

Compare the Devito solver with the legacy NumPy implementation.

a) Run both solvers with the same parameters.
b) Verify they produce the same results.
c) Compare execution times.

Solution

```python
from src.diffu import solve_diffusion_1d
from src.diffu.diffu1D_u0 import solver_FE_simple
import numpy as np
import time

# Parameters
L = 1.0
a = 1.0
Nx = 200
F = 0.5
T = 0.1

dx = L / Nx
dt = F * dx**2 / a

# Devito solver
t0 = time.perf_counter()
result_devito = solve_diffusion_1d(
    L=L, a=a, Nx=Nx, T=T, F=F,
    I=lambda x: np.sin(np.pi * x),
)
t_devito = time.perf_counter() - t0

# Legacy NumPy solver
t0 = time.perf_counter()
u_legacy, x_legacy, t_legacy, cpu_legacy = solver_FE_simple(
    I=lambda x: np.sin(np.pi * x),
    a=a,
    f=lambda x, t: 0,
    L=L,
    dt=dt,
    F=F,
    T=T,
)
t_numpy = time.perf_counter() - t0

# Compare results
diff = np.max(np.abs(result_devito.u - u_legacy))
print(f"Maximum difference: {diff:.2e}")
print(f"Devito time: {t_devito:.4f} s")
print(f"NumPy time: {t_numpy:.4f} s")

# Note: For small problems, NumPy may be faster due to compilation
# overhead. For large problems, Devito's optimized C code wins.
```

For large grids, Devito's automatically generated and optimized C code typically outperforms pure Python/NumPy implementations. The advantage grows with problem size.

4. Advection-Dominated Equations

Wave (Chapter Chapter 2) and diffusion (Chapter Chapter 3) equations are solved reliably by finite difference methods. As soon as we add a first-order derivative in space, representing *advective* transport (also known as *convective* transport), the numerics gets more complicated and intuitively attractive methods no longer work well. We shall show how and why such methods fail and provide remedies. The present chapter builds on basic knowledge about finite difference methods for diffusion and wave equations, including the analysis by Fourier components, truncation error analysis (Appendix Chapter 7), and compact difference notation.

i Remark on terminology

It is common to refer to movement of a fluid as convection, while advection is the transport of some material dissolved or suspended in the fluid. We shall mostly choose the word advection here, but both terms are in heavy use, and for mass transport of a substance the PDE has an advection term, while the similar term for the heat equation is a convection term.

Much more comprehensive discussion of dispersion analysis for advection problems can be found in the book by Duran (Duran 2010). This is a an excellent resource for further studies on the topic of advection PDEs, with emphasis on generalizations to real geophysical problems. The book by Fletcher (Fletcher 2013) also has a good overview of methods for advection and convection problems.

4.1. 1D linear advection equations with constant velocity

We consider the pure advection model

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0, \quad x \in (0, L),\ t \in (0, T], \tag{4.1}$$

$$u(x, 0) = I(x), \quad x \in (0, L), \tag{4.2}$$

$$u(0, t) = U_0, \quad t \in (0, T]. \tag{4.3}$$

In (4.3), v is a given parameter, typically reflecting the transport velocity of a quantity u with a flow. There is only one boundary condition (4.2) since the spatial derivative is only first order in the PDE (4.3). The information at $x = 0$ and the initial condition get transported in the positive x direction if $v > 0$ through the domain.

It is easiest to find the solution of (4.3) if we remove the boundary condition and consider a process on the infinite domain $(-\infty, \infty)$. The solution is simply

$$u(x, t) = I(x - vt)\,. \tag{4.4}$$

This is also the solution we expect locally in a finite domain before boundary conditions have reflected or modified the wave.

A particular feature of the solution (4.4) is that

$$u(x_i, t_{n+1}) = u(x_{i-1}, t_n), \tag{4.5}$$

if $x_i = i\Delta x$ and $t_n = n\Delta t$ are points in a uniform mesh. We see this relation from

$$\begin{aligned} u(i\Delta x, (n+1)\Delta t) &= I(i\Delta x - v(n+1)\Delta t) \\ &= I((i-1)\Delta x - vn\Delta t - v\Delta t + \Delta x) \\ &= I((i-1)\Delta x - vn\Delta t) \\ &= u((i-1)\Delta x, n\Delta t), \end{aligned}$$

provided $v = \Delta x/\Delta t$. So, whenever we see a scheme that collapses to

$$u^{n+1} * *i = u * *i - 1^n, \tag{4.6}$$

for the PDE in question, we have in fact a scheme that reproduces the analytical solution, and many of the schemes to be presented possess this nice property!

Finally, we add that a discussion of appropriate boundary conditions for the advection PDE in multiple dimensions is a challenging topic beyond the scope of this text.

4.2. Simplest scheme: forward in time, centered in space

4.2.1. Method

A first attempt to solve a PDE like (4.3) will normally be to look for a time-discretization scheme that is explicit so we avoid solving systems of linear equations. In space, we anticipate that centered differences are most accurate and therefore best. These two arguments lead us to a Forward Euler scheme in time and centered differences in space:

$$[D_t^+ u + vD_{2x}u = 0]_i^n$$

Written out, we see that this expression implies that

$$u^{n+1} = u^n - \frac{1}{2}C(u_{i+1}^n - u_{i-1}^n),$$

with C as the Courant number

$$C = \frac{v\Delta t}{\Delta x}\,.$$

Implementation

A solver function for our scheme goes as follows.

```
import numpy as np

def solver_FECS(I, U0, v, L, dt, C, T, user_action=None):
    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = v * dt / C
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
    C = v * dt / dx

    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        for i in range(1, Nx):
            u[i] = u_n[i] - 0.5 * C * (u_n[i + 1] - u_n[i - 1])

        u[0] = U0

        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_n, u = u, u_n

def solver(I, U0, v, L, dt, C, T, user_action=None, scheme="FE", periodic_bc=True):
    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = v * dt / C
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
    C = v * dt / dx
    print("dt=%g, dx=%g, Nx=%d, C=%g" % (dt, dx, Nx, C))

    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)
    u_nm1 = np.zeros(Nx + 1)
    integral = np.zeros(Nt + 1)
```

```
    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    u[0] = U0

    integral[0] = dx * (0.5 * u_n[0] + 0.5 * u_n[Nx] + np.sum(u_n[1:-1]))

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        if scheme == "FE":
            if periodic_bc:
                i = 0
                u[i] = u_n[i] - 0.5 * C * (u_n[i + 1] - u_n[Nx])
                u[Nx] = u[0]
            for i in range(1, Nx):
                u[i] = u_n[i] - 0.5 * C * (u_n[i + 1] - u_n[i - 1])
        elif scheme == "LF":
            if n == 0:
                if periodic_bc:
                    i = 0
                    u_n[i] = u_n[Nx]
                for i in range(1, Nx + 1):
                    u[i] = u_n[i] - C * (u_n[i] - u_n[i - 1])
            else:
                if periodic_bc:
                    i = 0
                    u[i] = u_nm1[i] - C * (u_n[i + 1] - u_n[Nx - 1])
                for i in range(1, Nx):
                    u[i] = u_nm1[i] - C * (u_n[i + 1] - u_n[i - 1])
                if periodic_bc:
                    u[Nx] = u[0]
        elif scheme == "UP":
            if periodic_bc:
                u_n[0] = u_n[Nx]
            for i in range(1, Nx + 1):
                u[i] = u_n[i] - C * (u_n[i] - u_n[i - 1])
        elif scheme == "LW":
            if periodic_bc:
                i = 0
                u[i] = (
                    u_n[i]
                    - 0.5 * C * (u_n[i + 1] - u_n[Nx - 1])
                    + 0.5 * C * (u_n[i + 1] - 2 * u_n[i] + u_n[Nx - 1])
                )
            for i in range(1, Nx):
```

```
                u[i] = (
                    u_n[i]
                    - 0.5 * C * (u_n[i + 1] - u_n[i - 1])
                    + 0.5 * C * (u_n[i + 1] - 2 * u_n[i] + u_n[i - 1])
                )
            if periodic_bc:
                u[Nx] = u[0]
        else:
            raise ValueError('scheme="%s" not implemented' % scheme)

        if not periodic_bc:
            u[0] = U0

        integral[n + 1] = dx * (0.5 * u[0] + 0.5 * u[Nx] + np.sum(u[1:-1]))

        if user_action is not None:
            user_action(u, x, t, n + 1)

        u_nm1, u_n, u = u_n, u, u_nm1
        print("I:", integral[n + 1])
    return integral

def run_FECS(case):
    """Special function for the FECS case."""
    if case == "gaussian":

        def I(x):
            return np.exp(-0.5 * ((x - L / 10) / sigma) ** 2)
    elif case == "cosinehat":

        def I(x):
            return np.cos(np.pi * 5 / L * (x - L / 10)) if x < L / 5 else 0

    L = 1.0
    sigma = 0.02
    legends = []

    def plot(u, x, t, n):
        """Animate and plot every m steps in the same figure."""
        plt.figure(1)
        if n == 0:
            lines = plot(x, u)
        else:
            lines[0].set_ydata(u)
            plt.draw()
        plt.figure(2)
        m = 40
```

```
        if n % m != 0:
            return
        print(
            "t=%g, n=%d, u in [%g, %g] w/%d points" % (t[n], n, u.min(), u.max(), x.size)
        )
        if np.abs(u).max() > 3:  # Instability?
            return
        plt.plot(x, u)
        legends.append("t=%g" % t[n])

    plt.ion()
    U0 = 0
    dt = 0.001
    C = 1
    T = 1
    solver(I=I, U0=U0, v=1.0, L=L, dt=dt, C=C, T=T, user_action=plot)
    plt.legend(legends, loc="lower left")
    plt.savefig("tmp.png")
    plt.savefig("tmp.pdf")
    plt.axis([0, L, -0.75, 1.1])
    plt.show()

def run(scheme="UP", case="gaussian", C=1, dt=0.01):
    """General admin routine for explicit and implicit solvers."""

    if case == "gaussian":

        def I(x):
            return np.exp(-0.5 * ((x - L / 10) / sigma) ** 2)
    elif case == "cosinehat":

        def I(x):
            return np.cos(np.pi * 5 / L * (x - L / 10)) if 0 < x < L / 5 else 0

    L = 1.0
    sigma = 0.02
    global lines  # needs to be saved between calls to plot

    def plot(u, x, t, n):
        """Plot t=0 and t=0.6 in the same figure."""
        plt.figure(1)
        global lines
        if n == 0:
            lines = plt.plot(x, u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.xlabel("x")
            plt.ylabel("u")
```

```
            plt.axes().set_aspect(0.15)
            plt.savefig("tmp_%04d.png" % n)
            plt.savefig("tmp_%04d.pdf" % n)
        else:
            lines[0].set_ydata(u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.title("C=%g, dt=%g, dx=%g" % (C, t[1] - t[0], x[1] - x[0]))
            plt.legend(["t=%.3f" % t[n]])
            plt.xlabel("x")
            plt.ylabel("u")
            plt.draw()
            plt.savefig("tmp_%04d.png" % n)
        plt.figure(2)
        eps = 1e-14
        if abs(t[n] - 0.6) > eps and abs(t[n] - 0) > eps:
            return
        print(
            "t=%g, n=%d, u in [%g, %g] w/%d points" % (t[n], n, u.min(), u.max(), x.size)
        )
        if np.abs(u).max() > 3:  # Instability?
            return
        plt.plot(x, u)
        plt.draw()
        if n > 0:
            y = [I(x_ - v * t[n]) for x_ in x]
            plt.plot(x, y, "k--")
            if abs(t[n] - 0.6) < eps:
                filename = ("tmp_%s_dt%s_C%s" % (scheme, t[1] - t[0], C)).replace(".", "")
                np.savez(filename, x=x, u=u, u_e=y)

plt.ion()
U0 = 0
T = 0.7
v = 1
codecs = dict(flv="flv", mp4="libx264", webm="libvpx", ogg="libtheora")
import glob
import os

for name in glob.glob("tmp_*.png"):
    os.remove(name)
for ext in codecs:
    name = "movie.%s" % ext
    if os.path.isfile(name):
        os.remove(name)

if scheme == "CN":
    integral = solver_theta(I, v, L, dt, C, T, user_action=plot, FE=False)
```

```
    elif scheme == "BE":
        integral = solver_theta(I, v, L, dt, C, T, theta=1, user_action=plot)
    else:
        integral = solver(
            I=I, U0=U0, v=v, L=L, dt=dt, C=C, T=T, scheme=scheme, user_action=plot
        )
    plt.figure(2)
    plt.axis([0, L, -0.5, 1.1])
    plt.xlabel("$x$")
    plt.ylabel("$u$")
    plt.axes().set_aspect(0.5)  # no effect
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()
    for codec in codecs:
        cmd = "ffmpeg -i tmp_%%04d.png -r 25 -vcodec %s movie.%s" % (codecs[codec], codec)
        os.system(cmd)
    print("Integral of u:", integral.max(), integral.min())

def solver_theta(I, v, L, dt, C, T, theta=0.5, user_action=None, FE=False):
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time (no restriction on F,
    i.e., the time step when theta >= 0.5).
    Vectorized implementation and sparse (tridiagonal)
    coefficient matrix.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time
    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = v * dt / C
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
    C = v * dt / dx
    print("dt=%g, dx=%g, Nx=%d, C=%g" % (dt, dx, Nx, C))

    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)
    u_nm1 = np.zeros(Nx + 1)
    integral = np.zeros(Nt + 1)

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])
```

```
    integral[0] = dx * (0.5 * u_n[0] + 0.5 * u_n[Nx] + np.sum(u_n[1:-1]))

    if user_action is not None:
        user_action(u_n, x, t, 0)

    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    diagonal[:] = 1
    lower[:] = -0.5 * theta * C
    upper[:] = 0.5 * theta * C
    if FE:
        diagonal[:] += 4.0 / 6
        lower[:] += 1.0 / 6
        upper[:] += 1.0 / 6
    upper[0] = 0
    lower[-1] = 0

    diags = [0, -1, 1]
    import scipy.sparse
    import scipy.sparse.linalg

    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )

    for n in range(0, Nt):
        b[1:-1] = u_n[1:-1] + 0.5 * (1 - theta) * C * (u_n[:-2] - u_n[2:])
        if FE:
            b[1:-1] += 1.0 / 6 * u_n[:-2] + 1.0 / 6 * u_n[:-2] + 4.0 / 6 * u_n[1:-1]
        b[0] = u_n[Nx]
        b[-1] = u_n[0]  # boundary conditions
        b[0] = 0
        b[-1] = 0  # boundary conditions
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, n + 1)

        integral[n + 1] = dx * (0.5 * u[0] + 0.5 * u[Nx] + np.sum(u[1:-1]))

        u_n, u = u, u_n
```

```
    t1 = time.perf_counter()
    return integral

if __name__ == "__main__":
    run(scheme="LW", case="gaussian", C=1, dt=0.01)
```

4.2.2. Test cases

The typical solution u has the shape of I and is transported at velocity v to the right (if $v > 0$). Let us consider two different initial conditions, one smooth (Gaussian pulse) and one non-smooth (half-truncated cosine pulse):

$$u(x,0) = Ae^{-\frac{1}{2}\left(\frac{x-L/10}{\sigma}\right)^2},$$
$$u(x,0) = A\cos\left(\frac{5\pi}{L}\left(x - \frac{L}{10}\right)\right), \quad x < \frac{L}{5} \text{ else } 0\,. \tag{4.7}$$

The parameter A is the maximum value of the initial condition.

Before doing numerical simulations, we scale the PDE problem and introduce $\bar{x} = x/L$ and $\bar{t} = vt/L$, which gives

$$\frac{\partial \bar{u}}{\partial \bar{t}} + \frac{\partial \bar{u}}{\partial \bar{x}} = 0\,.$$

The unknown u is scaled by the maximum value of the initial condition: $\bar{u} = u/\max|I(x)|$ such that $|\bar{u}(\bar{x},0)| \in [0,1]$. The scaled problem is solved by setting $v = 1$, $L = 1$, and $A = 1$. From now on we drop the bars.

To run our test cases and plot the solution, we make the function

```
def run_FECS(case):
    """Special function for the FECS case."""
    if case == "gaussian":

        def I(x):
            return np.exp(-0.5 * ((x - L / 10) / sigma) ** 2)
    elif case == "cosinehat":

        def I(x):
            return np.cos(np.pi * 5 / L * (x - L / 10)) if x < L / 5 else 0

    L = 1.0
    sigma = 0.02
    legends = []

    def plot(u, x, t, n):
        """Animate and plot every m steps in the same figure."""
        plt.figure(1)
```

```
        if n == 0:
            lines = plot(x, u)
        else:
            lines[0].set_ydata(u)
            plt.draw()
        plt.figure(2)
        m = 40
        if n % m != 0:
            return
        print(
            "t=%g, n=%d, u in [%g, %g] w/%d points" % (t[n], n, u.min(), u.max(), x.size)
        )
        if np.abs(u).max() > 3:  # Instability?
            return
        plt.plot(x, u)
        legends.append("t=%g" % t[n])

    plt.ion()
    U0 = 0
    dt = 0.001
    C = 1
    T = 1
    solver(I=I, U0=U0, v=1.0, L=L, dt=dt, C=C, T=T, user_action=plot)
    plt.legend(legends, loc="lower left")
    plt.savefig("tmp.png")
    plt.savefig("tmp.pdf")
    plt.axis([0, L, -0.75, 1.1])
    plt.show()

def run(scheme="UP", case="gaussian", C=1, dt=0.01):
    """General admin routine for explicit and implicit solvers."""

    if case == "gaussian":

        def I(x):
            return np.exp(-0.5 * ((x - L / 10) / sigma) ** 2)
    elif case == "cosinehat":

        def I(x):
            return np.cos(np.pi * 5 / L * (x - L / 10)) if 0 < x < L / 5 else 0

    L = 1.0
    sigma = 0.02
    global lines  # needs to be saved between calls to plot

    def plot(u, x, t, n):
        """Plot t=0 and t=0.6 in the same figure."""
```

```
        plt.figure(1)
        global lines
        if n == 0:
            lines = plt.plot(x, u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.xlabel("x")
            plt.ylabel("u")
            plt.axes().set_aspect(0.15)
            plt.savefig("tmp_%04d.png" % n)
            plt.savefig("tmp_%04d.pdf" % n)
        else:
            lines[0].set_ydata(u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.title("C=%g, dt=%g, dx=%g" % (C, t[1] - t[0], x[1] - x[0]))
            plt.legend(["t=%.3f" % t[n]])
            plt.xlabel("x")
            plt.ylabel("u")
            plt.draw()
            plt.savefig("tmp_%04d.png" % n)
        plt.figure(2)
        eps = 1e-14
        if abs(t[n] - 0.6) > eps and abs(t[n] - 0) > eps:
            return
        print(
            "t=%g, n=%d, u in [%g, %g] w/%d points" % (t[n], n, u.min(), u.max(), x.size)
        )
        if np.abs(u).max() > 3:  # Instability?
            return
        plt.plot(x, u)
        plt.draw()
        if n > 0:
            y = [I(x_ - v * t[n]) for x_ in x]
            plt.plot(x, y, "k--")
            if abs(t[n] - 0.6) < eps:
                filename = ("tmp_%s_dt%s_C%s" % (scheme, t[1] - t[0], C)).replace(".", "")
                np.savez(filename, x=x, u=u, u_e=y)

plt.ion()
U0 = 0
T = 0.7
v = 1
codecs = dict(flv="flv", mp4="libx264", webm="libvpx", ogg="libtheora")
import glob
import os

for name in glob.glob("tmp_*.png"):
    os.remove(name)
```

```
    for ext in codecs:
        name = "movie.%s" % ext
        if os.path.isfile(name):
            os.remove(name)

    if scheme == "CN":
        integral = solver_theta(I, v, L, dt, C, T, user_action=plot, FE=False)
    elif scheme == "BE":
        integral = solver_theta(I, v, L, dt, C, T, theta=1, user_action=plot)
    else:
        integral = solver(
            I=I, U0=U0, v=v, L=L, dt=dt, C=C, T=T, scheme=scheme, user_action=plot
        )
    plt.figure(2)
    plt.axis([0, L, -0.5, 1.1])
    plt.xlabel("$x$")
    plt.ylabel("$u$")
    plt.axes().set_aspect(0.5)  # no effect
    plt.savefig("tmp1.png")
    plt.savefig("tmp1.pdf")
    plt.show()
    for codec in codecs:
        cmd = "ffmpeg -i tmp_%%04d.png -r 25 -vcodec %s movie.%s" % (codecs[codec], codec)
        os.system(cmd)
    print("Integral of u:", integral.max(), integral.min())

def solver_theta(I, v, L, dt, C, T, theta=0.5, user_action=None, FE=False):
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time (no restriction on F,
    i.e., the time step when theta >= 0.5).
    Vectorized implementation and sparse (tridiagonal)
    coefficient matrix.
    """
    import time

    t0 = time.perf_counter()  # for measuring the CPU time
    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points in time
    dx = v * dt / C
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
    C = v * dt / dx
    print("dt=%g, dx=%g, Nx=%d, C=%g" % (dt, dx, Nx, C))
```

```
    u = np.zeros(Nx + 1)
    u_n = np.zeros(Nx + 1)
    u_nm1 = np.zeros(Nx + 1)
    integral = np.zeros(Nt + 1)

    for i in range(0, Nx + 1):
        u_n[i] = I(x[i])

    integral[0] = dx * (0.5 * u_n[0] + 0.5 * u_n[Nx] + np.sum(u_n[1:-1]))

    if user_action is not None:
        user_action(u_n, x, t, 0)

    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    diagonal[:] = 1
    lower[:] = -0.5 * theta * C
    upper[:] = 0.5 * theta * C
    if FE:
        diagonal[:] += 4.0 / 6
        lower[:] += 1.0 / 6
        upper[:] += 1.0 / 6
    upper[0] = 0
    lower[-1] = 0

    diags = [0, -1, 1]
    import scipy.sparse
    import scipy.sparse.linalg

    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )

    for n in range(0, Nt):
        b[1:-1] = u_n[1:-1] + 0.5 * (1 - theta) * C * (u_n[:-2] - u_n[2:])
        if FE:
            b[1:-1] += 1.0 / 6 * u_n[:-2] + 1.0 / 6 * u_n[:-2] + 4.0 / 6 * u_n[1:-1]
        b[0] = u_n[Nx]
        b[-1] = u_n[0]  # boundary conditions
        b[0] = 0
        b[-1] = 0  # boundary conditions
```

```
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, n + 1)

        integral[n + 1] = dx * (0.5 * u[0] + 0.5 * u[Nx] + np.sum(u[1:-1]))

        u_n, u = u, u_n

    t1 = time.perf_counter()
    return integral

if __name__ == "__main__":
    run(scheme="LW", case="gaussian", C=1, dt=0.01)
```

4.2.3. Bug?

Running either of the test cases, the plot becomes a mess, and the printout of u values in the `plot` function reveals that u grows very quickly. We may reduce Δt and make it very small, yet the solution just grows. Such behavior points to a bug in the code. However, choosing a coarse mesh and performing one time step by hand calculations produces the same numbers as the code, so the implementation seems to be correct. The hypothesis is therefore that the solution is unstable.

4.3. Analysis of the scheme

It is easy to show that a typical Fourier component

$$u(x,t) = B\sin(k(x-ct))$$

is a solution of our PDE for any spatial wave length $\lambda = 2\pi/k$ and any amplitude B. (Since the PDE to be investigated by this method is homogeneous and linear, B will always cancel out, so we tend to skip this amplitude, but keep it here in the beginning for completeness.)

A general solution may be viewed as a collection of long and short waves with different amplitudes. Algebraically, the work simplifies if we introduce the complex Fourier component

$$u(x,t) = A_{\mathrm{e}}e^{ikx},$$

with

$$A_{\mathrm{e}} = Be^{-ikv\Delta t} = Be^{-iCk\Delta x}\,.$$

Note that $|A_{\mathrm{e}}| \leq 1$.

It turns out that many schemes also allow a Fourier wave component as solution, and we can use the numerically computed values of A_{e} (denoted A) to learn about the quality of the scheme. Hence, to analyze the difference scheme we have just implemented, we look at how it treats the Fourier component

$$u_q^n = A^n e^{ikq\Delta x}\,.$$

Inserting the numerical component in the scheme,

$$[D_t^+ A e^{ikq\Delta x} + vD_{2x} A e^{ikq\Delta x} = 0]_q^n,$$

and making use of (6.5) results in

$$[e^{ikq\Delta x}(\frac{A-1}{\Delta t} + v\frac{1}{\Delta x} i\sin(k\Delta x)) = 0]_q^n,$$

which implies

$$A = 1 - iC\sin(k\Delta x)\,.$$

The numerical solution features the formula A^n. To find out whether A^n means growth in time, we rewrite A in polar form: $A = A_r e^{i\phi}$, for real numbers A_r and ϕ, since we then have $A^n = A_r^n e^{i\phi n}$. The magnitude of A^n is A_r^n. In our case, $A_r = (1 + C^2\sin^2(kx))^{1/2} > 1$, so A_r^n will increase in time, whereas the exact solution will not. Regardless of Δt, we get unstable numerical solutions.

4.4. Leapfrog in time, centered differences in space

4.4.1. Method

Another explicit scheme is to do a "leapfrog" jump over $2\Delta t$ in time and combine it with central differences in space:

$$[D_{2t} u + vD_{2x} u = 0]_i^n,$$

which results in the updating formula

$$u_i^{n+1} = u^{n-1} * *i - C(u * *i + 1^n - u_{i-1}^n)\,.$$

A special scheme is needed to compute u^1, but we leave that problem for now. Anyway, this special scheme can be found in advec1D.py.

4.4.2. Implementation

We now need to work with three time levels and must modify our solver a bit:

```
Nt = int(round(T/float(dt)))
t = np.linspace(0, Nt*dt, Nt+1)   # Mesh points in time
...
u   = np.zeros(Nx+1)
u_1 = np.zeros(Nx+1)
u_2 = np.zeros(Nx+1)
...
for n in range(0, Nt):
    if scheme == 'FE':
        for i in range(1, Nx):
            u[i] = u_1[i] - 0.5*C*(u_1[i+1] - u_1[i-1])
    elif scheme == 'LF':
```

```
        if n == 0:
            for i in range(1, Nx):
                ...
        else:
            for i in range(1, Nx+1):
                u[i] = u_2[i] - C*(u_1[i] - u_1[i-1])

    u_2, u_1, u = u_1, u, u_2
```

4.4.3. Running a test case

Let us try a coarse mesh such that the smooth Gaussian initial condition is represented by 1 at mesh node 1 and 0 at all other nodes. This triangular initial condition should then be advected to the right. Choosing scaled variables as $\Delta t = 0.1$, $T = 1$, and $C = 1$ gives the plot in Figure Figure 4.1, which is in fact identical to the exact solution (!).

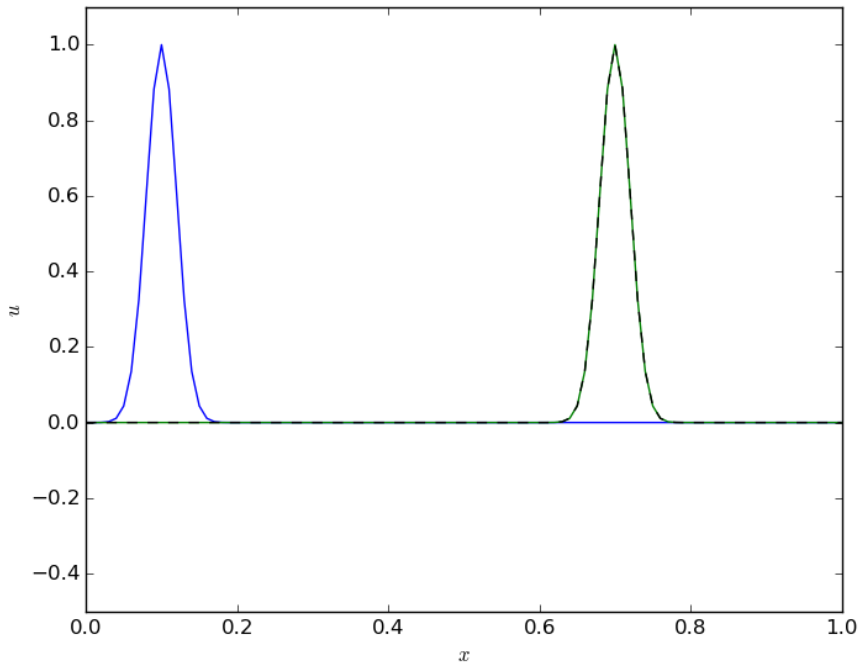


Figure 4.1.: Exact solution obtained by Leapfrog scheme with $\Delta t = 0.1$ and $C = 1$.

4.4.4. Running more test cases

We can run two types of initial conditions for $C = 0.8$: one very smooth with a Gaussian function (Figure Figure 4.2) and one with a discontinuity in the first derivative (Figure Figure 4.3). Unless we have a very fine mesh, as in the left plots in the figures, we get small ripples behind the main wave, and this main wave has the amplitude reduced.

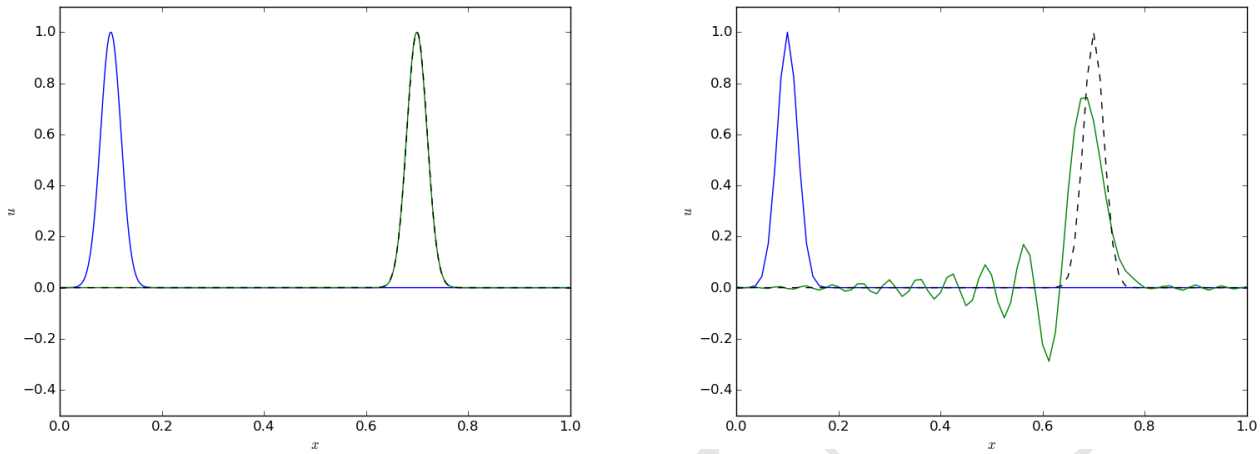


Figure 4.2.: Advection of a Gaussian function with a leapfrog scheme and $C = 0.8$, $\Delta t = 0.001$ (left) and $\Delta t = 0.01$ (right).

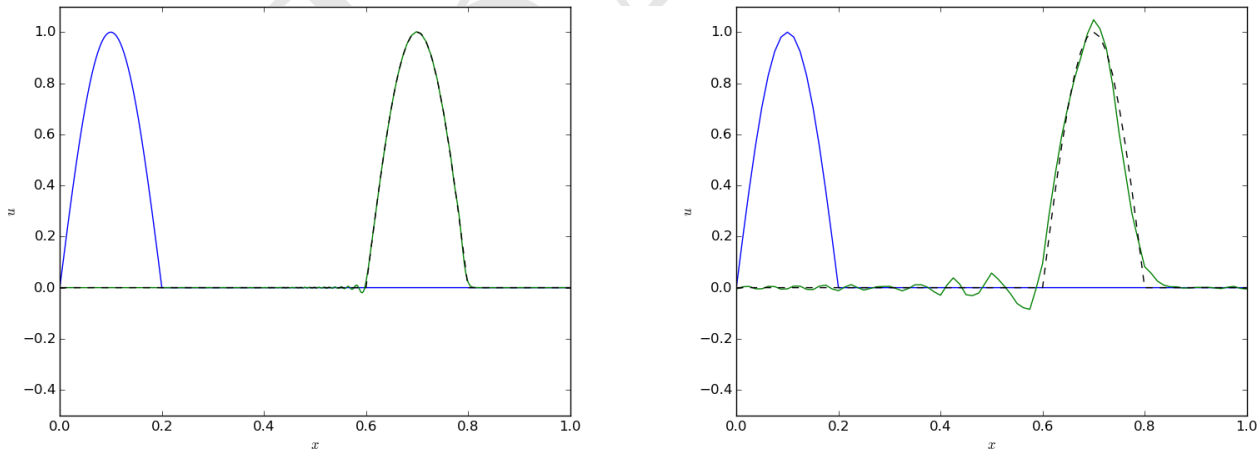


Figure 4.3.: Advection of half a cosine function with a leapfrog scheme and $C = 0.8$, $\Delta t = 0.001$ (left) and $\Delta t = 0.01$ (right).

4.4.5. Analysis

We can perform a Fourier analysis again. Inserting the numerical Fourier component in the Leapfrog scheme, we get

$$A^2 - i2C\sin(k\Delta x)A - 1 = 0,$$

and

$$A = -iC\sin(k\Delta x) \pm \sqrt{1 - C^2\sin^2(k\Delta x)}\,.$$

Rewriting to polar form, $A = A_r e^{i\phi}$, we see that $A_r = 1$, so the numerical component is neither increasing nor decreasing in time, which is exactly what we want. However, for $C > 1$, the square root can become complex valued, so stability is obtained only as long as $C \leq 1$.

Stability

For all the working schemes to be presented in this chapter, we get the stability condition $C \leq 1$:

$$\Delta t \leq \frac{\Delta x}{v}\,.$$

This is called the CFL condition and applies almost always to successful schemes for advection problems. Of course, one can use Crank-Nicolson or Backward Euler schemes for increased and even unconditional stability (no Δt restrictions), but these have other less desired damping problems.

We introduce $p = k\Delta x$. The amplification factor now reads

$$A = -iC\sin p \pm \sqrt{1 - C^2\sin^2 p},$$

and is to be compared to the exact amplification factor

$$A_{\mathrm{e}} = e^{-ikv\Delta t} = e^{-ikC\Delta x} = e^{-iCp}\,.$$

Section Section 4.10 compares numerical amplification factors of many schemes with the exact expression.

4.5. Upwind differences in space

Since the PDE reflects transport of information along with a flow in positive x direction, when $v > 0$, it could be natural to go (what is called) upstream and not downstream in the spatial derivative to collect information about the change of the function. That is, we approximate

$$\frac{\partial u}{\partial x}(x_i, t_n) \approx [D_x^- u]_i^n = \frac{u_i^n - u_{i-1}^n}{\Delta x}\,.$$

This is called an *upwind difference* (the corresponding difference in the time direction would be called a backward difference, and we could use that name in space too, but *upwind* is the common name for a difference against the flow in advection problems). This spatial approximation does magic compared to the scheme we had with Forward Euler in time and centered difference in space. With an upwind difference,

$$[D_t^+ u + vD_x^- u = 0]_i^n, \tag{4.8}$$

written out as

$$u_i^{n+1} = u_i^n - C(u^n * *i - u^n * *i - 1),$$

gives a generally popular and robust scheme that is stable if $C \leq 1$. As with the Leapfrog scheme, it becomes exact if $C = 1$, exactly as shown in Figure Figure 4.1. This is easy to see since $C = 1$ gives the property (4.6). However, any $C < 1$ gives a significant reduction in the amplitude of the solution, which is a purely numerical effect, see Figures Figure 4.4 and Figure 4.5. Experiments show, however, that reducing Δt or Δx, while keeping C reduces the error.

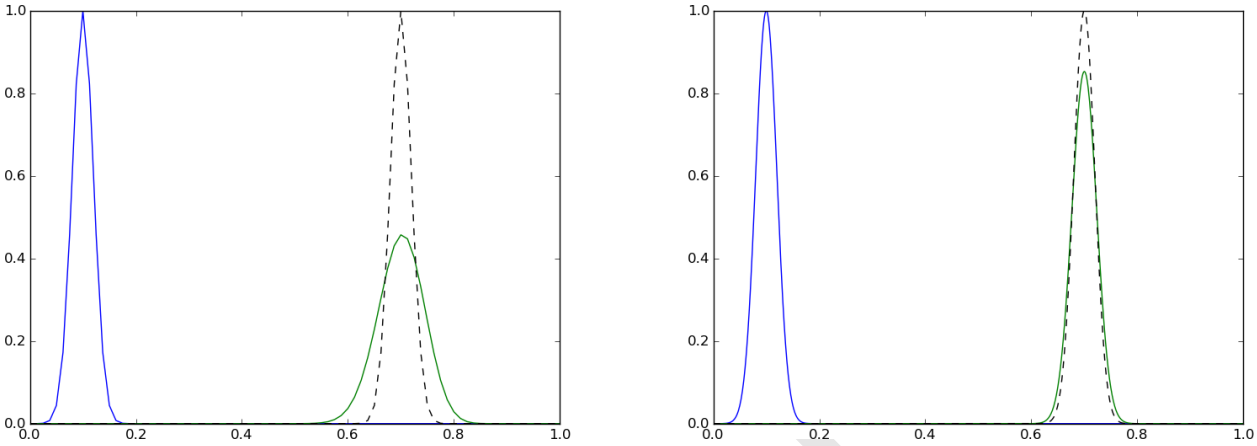


Figure 4.4.: Advection of a Gaussian function with a forward in time, upwind in space scheme and $C = 0.8$, $\Delta t = 0.01$ (left) and $\Delta t = 0.001$ (right).

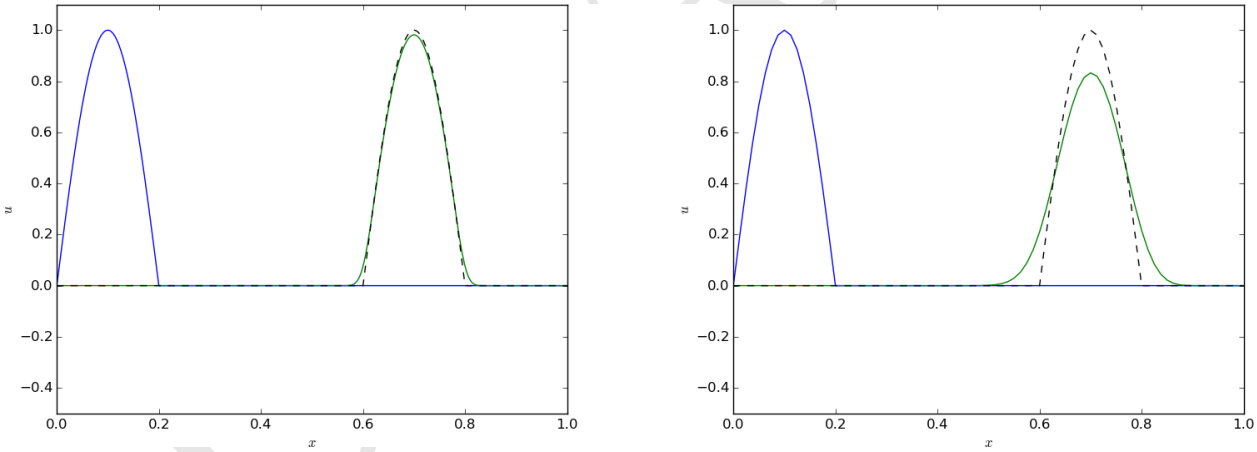


Figure 4.5.: Advection of half a cosine function with a forward in time, upwind in space scheme and $C = 0.8$, $\Delta t = 0.001$ (left) and $\Delta t = 0.01$ (right).

The amplification factor can be computed using the formula (6.4),

$$\frac{A-1}{\Delta t} + \frac{v}{\Delta x}(1 - e^{-ik\Delta x}) = 0,$$

which means

$$A = 1 - C(1 - \cos(p) - i\sin(p)).$$

For $C < 1$ there is, unfortunately, non-physical damping of discrete Fourier components, giving rise to reduced amplitude of u_i^n as in Figures Figure 4.4 and Figure 4.5. The damping seen in these figures is quite severe. Stability requires $C \leq 1$.

Interpretation of upwind difference as artificial diffusion

One can interpret the upwind difference as extra, artificial diffusion in the equation. Solving

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2},$$

by a forward difference in time and centered differences in space,

$$D^+ * *tu + vD * *2xu = \nu D_x D_x u]_i^n,$$

actually gives the upwind scheme (4.8) if $\nu = v\Delta x/2$. That is, solving the PDE $u_t + vu_x = 0$ by centered differences in space and forward difference in time is unsuccessful, but by adding some artificial diffusion νu_{xx}, the method becomes stable:

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = \left(\alpha + \frac{v\Delta x}{2}\right)\frac{\partial^2 u}{\partial x^2}.$$

4.6. Periodic boundary conditions

So far, we have given the value on the left boundary, u_0^n, and used the scheme to propagate the solution signal through the domain. Often, we want to follow such signals for long time series, and periodic boundary conditions are then relevant since they enable a signal that leaves the right boundary to immediately enter the left boundary and propagate through the domain again.

The periodic boundary condition is

$$u(0,t) = u(L,t), \quad u_0^n = u_{N_x}^n.$$

It means that we in the first equation, involving u_0^n, insert $u_{N_x}^n$, and that we in the last equation, involving $u_{N_x}^{n+1}$ insert u_0^{n+1}. Normally, we can do this in the simple way that `u_1[0]` is updated as `u_1[Nx]` at the beginning of a new time level.

In some schemes we may need $u_{N_x+1}^n$ and u_{-1}^n. Periodicity then means that these values are equal to u_1^n and $u_{N_x-1}^n$, respectively. For the upwind scheme, it is sufficient to set `u_1[0]=u_1[Nx]` at a new time level before computing `u[1]`. This ensures that `u[1]` becomes right and at the next time level `u[0]` at the current time level is correctly updated. For the Leapfrog scheme we must update `u[0]` and `u[Nx]` using the scheme:

```
if periodic_bc:
    i = 0
    u[i] = u_2[i] - C*(u_1[i+1] - u_1[Nx-1])
for i in range(1, Nx):
    u[i] = u_2[i] - C*(u_1[i+1] - u_1[i-1])
if periodic_bc:
    u[Nx] = u[0]
```

4.7. Implementation

4.7.1. Test condition

Analytically, we can show that the integral in space under the $u(x,t)$ curve is constant:

$$\int_0^L \left(\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x}\right) dx = 0$$
$$\frac{\partial}{\partial t}\int_0^L u dx = -\int_0^L v\frac{\partial u}{\partial x} dx$$
$$\frac{\partial u}{\partial t}\int_0^L u dx = [vu]_0^L = 0$$

as long as $u(0) = u(L) = 0$. We can therefore use the property

$$\int_0^L u(x,t)dx = \text{const}$$

as a partial verification during the simulation. Now, any numerical method with $C \neq 1$ will deviate from the constant, expected value, so the integral is a measure of the error in the scheme. The integral can be computed by the Trapezoidal integration rule

```
dx*(0.5*u[0] + 0.5*u[Nx] + np.sum(u[1:-1]))
```

if `u` is an array holding the solution.

4.7.2. The code

An appropriate `solver` function for multiple schemes may go as shown below.

```
def solver(I, U0, v, L, dt, C, T, user_action=None,
           scheme='FE', periodic_bc=True):

    Nt = int(round(T/float(dt)))
    t = np.linspace(0, Nt*dt, Nt+1)   # Mesh points in time
    dx = v*dt/C
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)       # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]
    C = v*dt/dx
    print 'dt=%g, dx=%g, Nx=%d, C=%g' % (dt, dx, Nx, C)

    u   = np.zeros(Nx+1)
    u_n = np.zeros(Nx+1)
    u_nm1 = np.zeros(Nx+1)
```

```
    integral = np.zeros(Nt+1)

    for i in range(0, Nx+1):
        u_n[i] = I(x[i])

    u[0] = U0

    integral[0] = dx*(0.5*u_n[0] + 0.5*u_n[Nx] + np.sum(u_n[1:-1]))

    if user_action is not None:
        user_action(u_n, x, t, 0)

    for n in range(0, Nt):
        if scheme == 'FE':
            if periodic_bc:
                i = 0
                u[i] = u_n[i] - 0.5*C*(u_n[i+1] - u_n[Nx])
                u[Nx] = u[0]
            for i in range(1, Nx):
                u[i] = u_n[i] - 0.5*C*(u_n[i+1] - u_n[i-1])
        elif scheme == 'LF':
            if n == 0:
                if periodic_bc:
                    i = 0
                    u_n[i] = u_n[Nx]
                for i in range(1, Nx+1):
                    u[i] = u_n[i] - C*(u_n[i] - u_n[i-1])
            else:
                if periodic_bc:
                    i = 0
                    u[i] = u_nm1[i] - C*(u_n[i+1] - u_n[Nx-1])
                for i in range(1, Nx):
                    u[i] = u_nm1[i] - C*(u_n[i+1] - u_n[i-1])
                if periodic_bc:
                    u[Nx] = u[0]
        elif scheme == 'UP':
            if periodic_bc:
                u_n[0] = u_n[Nx]
            for i in range(1, Nx+1):
                u[i] = u_n[i] - C*(u_n[i] - u_n[i-1])
        else:
            raise ValueError('scheme="%s" not implemented' % scheme)

        if not periodic_bc:
            u[0] = U0

        integral[n+1] = dx*(0.5*u[0] + 0.5*u[Nx] + np.sum(u[1:-1]))
```

```
        if user_action is not None:
            user_action(u, x, t, n+1)

        u_nm1, u_n, u = u_n, u, u_nm1
    return integral
```

4.7.3. Solving a specific problem

We need to call up the `solver` function in some kind of administering problem solving function that can solve specific problems and make appropriate visualization. The function below makes both static plots, screen animation, and hard copy videos in various formats.

```
def run(scheme='UP', case='gaussian', C=1, dt=0.01):
    """General admin routine for explicit and implicit solvers."""

    if case == 'gaussian':
        def I(x):
            return np.exp(-0.5*((x-L/10)/sigma)**2)
    elif case == 'cosinehat':
        def I(x):
            return np.cos(np.pi*5/L*(x - L/10)) if x < L/5 else 0

    L = 1.0
    sigma = 0.02
    global lines  # needs to be saved between calls to plot

    def plot(u, x, t, n):
        """Plot t=0 and t=0.6 in the same figure."""
        plt.figure(1)
        global lines
        if n == 0:
            lines = plt.plot(x, u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.xlabel('x'); plt.ylabel('u')
            plt.axes().set_aspect(0.15)
            plt.savefig('tmp_%04d.png' % n)
            plt.savefig('tmp_%04d.pdf' % n)
        else:
            lines[0].set_ydata(u)
            plt.axis([x[0], x[-1], -0.5, 1.5])
            plt.title('C=%g, dt=%g, dx=%g' %
                      (C, t[1]-t[0], x[1]-x[0]))
            plt.legend(['t=%.3f' % t[n]])
            plt.xlabel('x'); plt.ylabel('u')
            plt.draw()
            plt.savefig('tmp_%04d.png' % n)
```

```
        plt.figure(2)
        eps = 1E-14
        if abs(t[n] - 0.6) > eps and abs(t[n] - 0) > eps:
            return
        print 't=%g, n=%d, u in [%g, %g] w/%d points' % \
              (t[n], n, u.min(), u.max(), x.size)
        if np.abs(u).max() > 3:  # Instability?
            return
        plt.plot(x, u)
        plt.hold('on')
        plt.draw()
        if n > 0:
            y = [I(x_-v*t[n]) for x_ in x]
            plt.plot(x, y, 'k--')
            if abs(t[n] - 0.6) < eps:
                filename = ('tmp_%s_dt%s_C%s' % \
                            (scheme, t[1]-t[0], C)).replace('.', '')
                np.savez(filename, x=x, u=u, u_e=y)

    plt.ion()
    U0 = 0
    T = 0.7
    v = 1
    codecs = dict(flv='flv', mp4='libx264', webm='libvpx',
                  ogg='libtheora')
    import glob, os
    for name in glob.glob('tmp_*.png'):
        os.remove(name)
    for ext in codecs:
        name = 'movie.%s' % ext
        if os.path.isfile(name):
            os.remove(name)

    integral = solver(
        I=I, U0=U0, v=v, L=L, dt=dt, C=C, T=T,
        scheme=scheme, user_action=plot)
    plt.figure(2)
    plt.axis([0, L, -0.5, 1.1])
    plt.xlabel('$x$');  plt.ylabel('$u$')
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')
    plt.show()
    for codec in codecs:
        cmd = 'ffmpeg -i tmp_%%04d.png -r 25 -vcodec %s movie.%s' % \
              (codecs[codec], codec)
        os.system(cmd)
    print 'Integral of u:', integral.max(), integral.min()
```

The complete code is found in the file advec1D.py.

4.8. A Crank-Nicolson discretization in time and centered differences in space

Another obvious candidate for time discretization is the Crank-Nicolson method combined with centered differences in space:

$$[D_t u]^n_i + v\frac{1}{2}([D_{2x}u]^{n+1} * *i + [D * *2xu]^n_i) = 0\,.$$

It can be nice to include the Backward Euler scheme too, via the θ-rule,

$$[D_t u]^n_i + v\theta[D_{2x}u]^{n+1} * *i + v(1-\theta)[D * *2xu]^n_i = 0\,.$$

When θ is different from zero, this gives rise to an *implicit* scheme,

$$u^{n+1} * *i + \frac{\theta}{2}C(u^{n+1} * *i + 1 - u^{n+1}_{i-1}) = u^n_i - \frac{1-\theta}{2}C(u^n * *i + 1 - u^n * *i - 1)$$

for $i = 1, \ldots, N_x - 1$. At the boundaries we set $u = 0$ and simulate just to the point of time when the signal hits the boundary (and gets reflected).

$$u^{n+1} * *0 = u^{n+1} * *N_x = 0\,.$$

The elements on the diagonal in the matrix become:

$$A_{i,i} = 1, \quad i = 0, \ldots, N_x\,.$$

On the subdiagonal and superdiagonal we have

$$A_{i-1,i} = -\frac{\theta}{2}C, \quad A_{i+1,i} = \frac{\theta}{2}C, \quad i = 1, \ldots, N_x - 1,$$

with $A_{0,1} = 0$ and $A_{N_x-1,N_x} = 0$ due to the known boundary conditions. And finally, the right-hand side becomes

$$\begin{aligned} b_0 &= u^n_{N_x} \\ b_i &= u^n_i - \frac{1-\theta}{2}C(u^n * *i + 1 - u^n * *i - 1), \quad i = 1, \ldots, N_x - 1 \\ b_{N_x} &= u^n_0 \end{aligned}$$

The dispersion relation follows from inserting $u^n_q = A^n e^{ikx}$ and using the formula (6.5) for the spatial differences:

$$A = \frac{1 - (1-\theta)iC\sin p}{1 + \theta iC\sin p}\,.$$

Figure Figure 4.6 depicts a numerical solution for $C = 0.8$ and the Crank-Nicolson with severe oscillations behind the main wave. These oscillations are damped as the mesh is refined. Switching

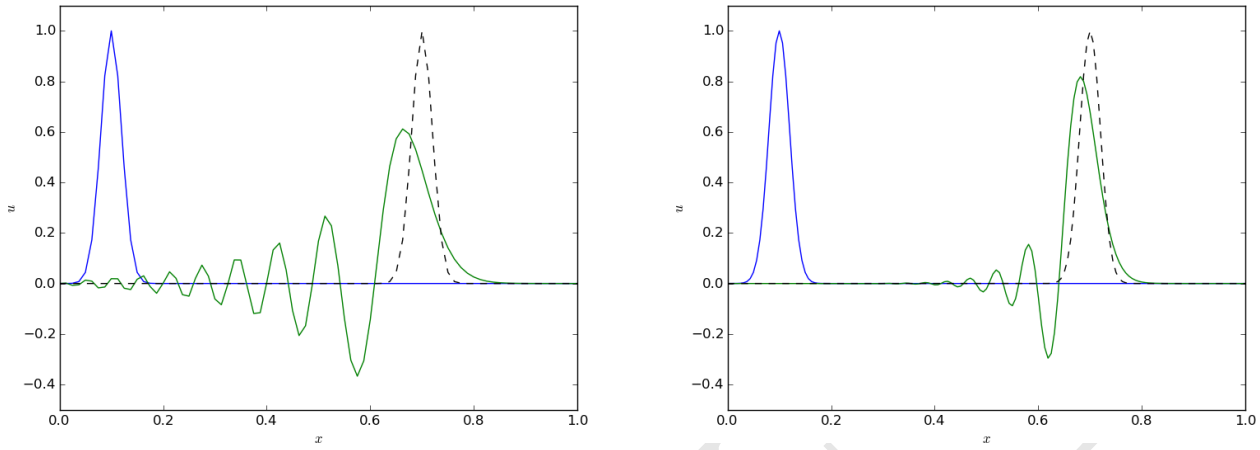


Figure 4.6.: Crank-Nicolson in time, centered in space, Gaussian profile, $C = 0.8$, $\Delta t = 0.01$ (left) and $\Delta t = 0.005$ (right).

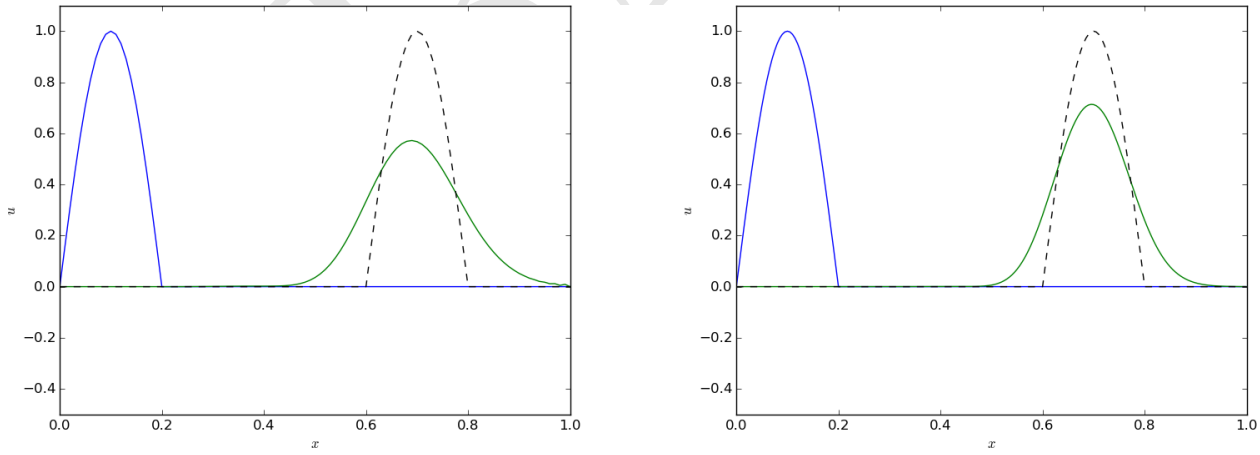


Figure 4.7.: Backward-Euler in time, centered in space, half a cosine profile, $C = 0.8$, $\Delta t = 0.01$ (left) and $\Delta t = 0.005$ (right).

to the Backward Euler scheme removes the oscillations, but the amplitude is significantly reduced. One could expect that the discontinuous derivative in the initial condition of the half a cosine wave would make even stronger demands on producing a smooth profile, but Figure Figure 4.7 shows that also here, Backward-Euler is capable of producing a smooth profile. All in all, there are no major differences between the Gaussian initial condition and the half a cosine condition for any of the schemes.

4.9. The Lax-Wendroff method

The Lax-Wendroff method is based on three ideas:

1. Express the new unknown u_i^{n+1} in terms of known quantities at $t = t_n$ by means of a Taylor polynomial of second degree.
2. Replace time-derivatives at $t = t_n$ by spatial derivatives, using the PDE.
3. Discretize the spatial derivatives by second-order differences so we achieve a scheme of accuracy $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2)$.

Let us follow the recipe. First we have the three-term Taylor polynomial,

$$u_i^{n+1} = u_i^n + \Delta t \left(\frac{\partial u}{\partial t}\right)_i^n + \frac{1}{2}\Delta t^2 \left(\frac{\partial^2 u}{\partial t^2}\right)_i^n .$$

From the PDE we have that temporal derivatives can be substituted by spatial derivatives:

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x},$$

and furthermore,

$$\frac{\partial^2 u}{\partial t^2} = v^2\frac{\partial^2 u}{\partial x^2}.$$

Inserted in the Taylor polynomial formula, we get

$$u_i^{n+1} = u_i^n - v\Delta t \left(\frac{\partial u}{\partial x}\right)_i^n + \frac{1}{2}\Delta t^2 v^2 \left(\frac{\partial^2 u}{\partial x^2}\right)_i^n .$$

To obtain second-order accuracy in space we now use central differences:

$$u_i^{n+1} = u_i^n - v\Delta t[D_{2x}u]_i^n + \frac{1}{2}\Delta t^2 v^2 [D_x D_x u]_i^n,$$

or written out,

$$u_i^{n+1} = u_i^n - \frac{1}{2}C(u^n **i + 1 - u^n **i - 1) + \frac{1}{2}C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n).$$

This is the explicit Lax-Wendroff scheme.

Lax-Wendroff works because of artificial viscosity

From the formulas above, we notice that the Lax-Wendroff method is nothing but a Forward Euler, central difference in space scheme, which we have shown to be useless because of chronic instability, plus an artificial diffusion term of strength $\frac{1}{2}\Delta t v^2$. It means that we can take an unstable scheme and add some diffusion to stabilize it. This is a common trick to deal with advection problems. Sometimes, the real physical diffusion is not sufficiently large to make schemes stable, so then we also add artificial diffusion.

From an analysis similar to the ones carried out above, we get an amplification factor for the Lax-Wendroff method that equals

$$A = 1 - iC\sin p - 2C^2\sin^2(p/2)\,.$$

This means that $|A| = 1$ and also that we have an exact solution if $C = 1$!

4.10. Analysis of dispersion relations

We have developed expressions for $A(C, p)$ in the exact solution $u_q^n = A^n e^{ikq\Delta x}$ of the discrete equations. Note that the Fourier component that solves the original PDE problem has no damping and moves with constant velocity v. There are two basic errors in the numerical Fourier component: there may be damping and the wave velocity may depend on C and $p = k\Delta x$.

The shortest wavelength that can be represented is $\lambda = 2\Delta x$. The corresponding k is $k = 2\pi/\lambda = \pi/\Delta x$, so $p = k\Delta x \in (0, \pi]$.

Given a complex A as a function of C and p, how can we visualize it? The two key ingredients in A is the magnitude, reflecting damping or growth of the wave, and the angle, closely related to the velocity of the wave. The Fourier component

$$D^n e^{ik(x-ct)}$$

has damping D and wave velocity c. Let us express our A in polar form, $A = A_r e^{-i\phi}$, and insert this expression in our discrete component $u_q^n = A^n e^{ikq\Delta x} = A^n e^{ikx}$:

$$u_q^n = A_r^n e^{-i\phi n} e^{ikx} = A_r^n e^{i(kx - n\phi)} = A_r^n e^{i(k(x-ct))},$$

for

$$c = \frac{\phi}{k\Delta t}\,.$$

Now,

$$k\Delta t = \frac{Ck\Delta x}{v} = \frac{Cp}{v},$$

so

$$c = \frac{\phi v}{Cp}\,.$$

An appropriate dimensionless quantity to plot is the scaled wave velocity c/v:

$$\frac{c}{v} = \frac{\phi}{Cp}\,.$$

Figures Figure 4.8–Figure 4.13 contain dispersion curves, velocity and damping, for various values of C. The horizontal axis shows the dimensionless frequency p of the wave, while the figures to the left illustrate the error in wave velocity c/v (should ideally be 1 for all p), and the figures to the right display the absolute value (magnitude) of the damping factor A_r. The curves are labeled according to the table below.

Label	Method
FE	Forward Euler in time, centered difference in space
LF	Leapfrog in time, centered difference in space
UP	Forward Euler in time, upwind difference in space
CN	Crank-Nicolson in time, centered difference in space
LW	Lax-Wendroff's method
BE	Backward Euler in time, centered difference in space

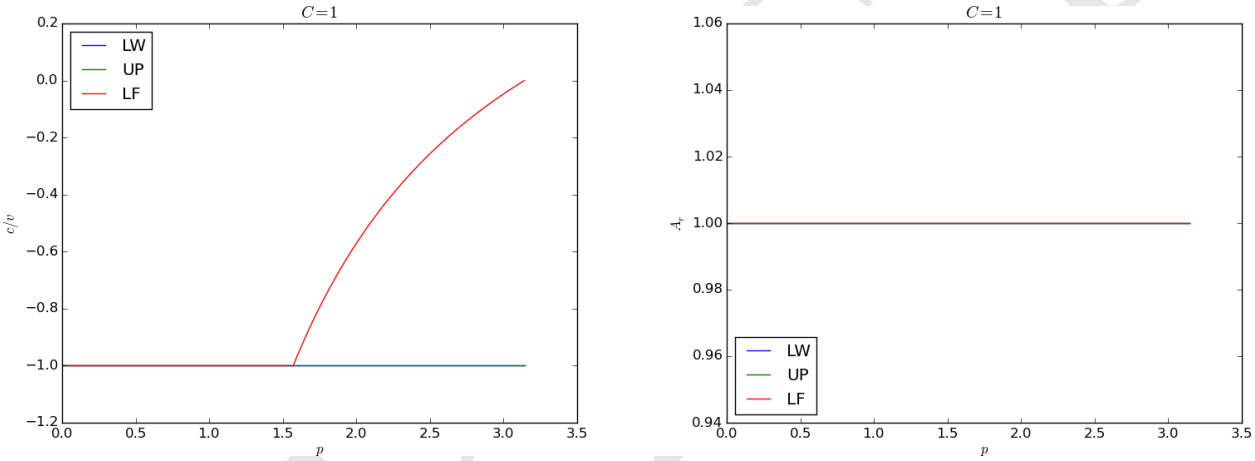


Figure 4.8.: Dispersion relations for $C = 1$.

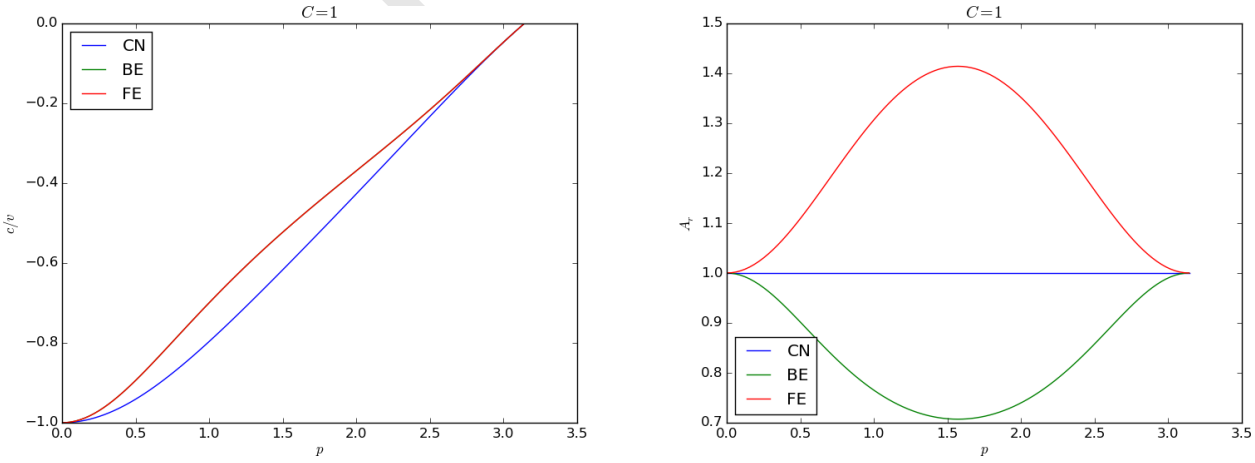


Figure 4.9.: Dispersion relations for $C = 1$.

The total damping after some time $T = n\Delta t$ is reflected by $A_r(C, p)^n$. Since normally $A_r < 1$, the damping goes like $A_r^{1/\Delta t}$ and approaches zero as $\Delta t \to 0$. The only way to reduce damping is to

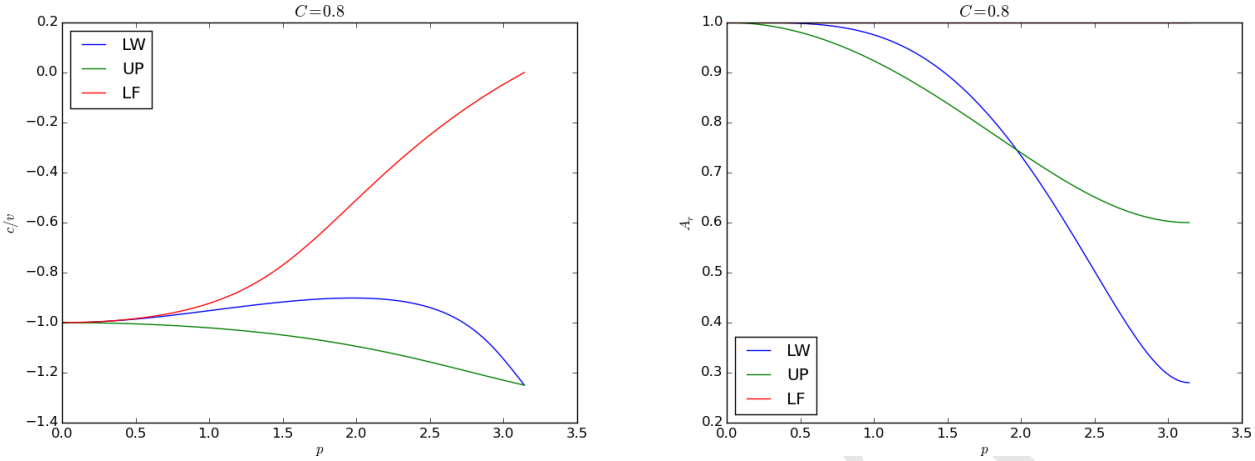


Figure 4.10.: Dispersion relations for $C = 0.8$.

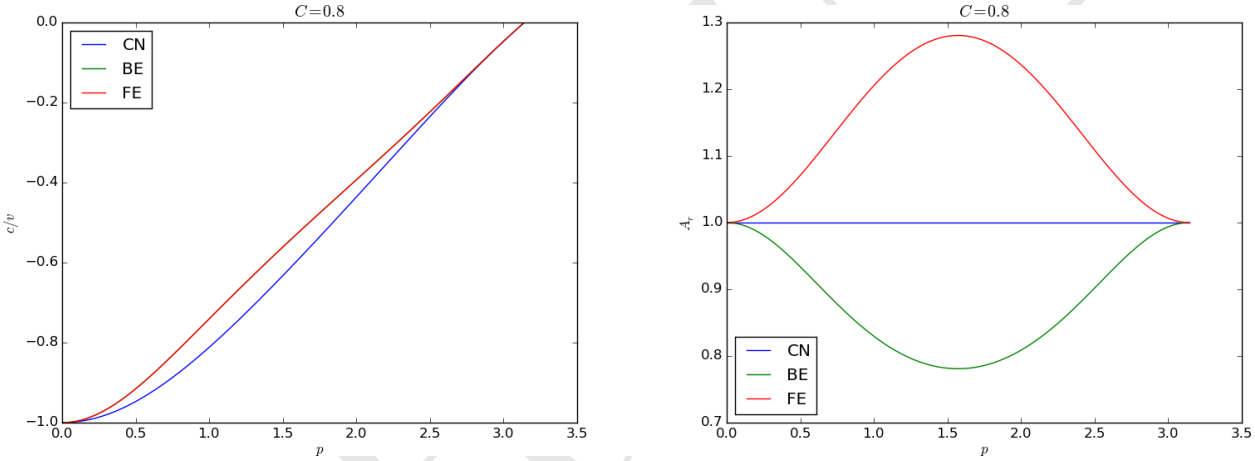


Figure 4.11.: Dispersion relations for $C = 0.8$.

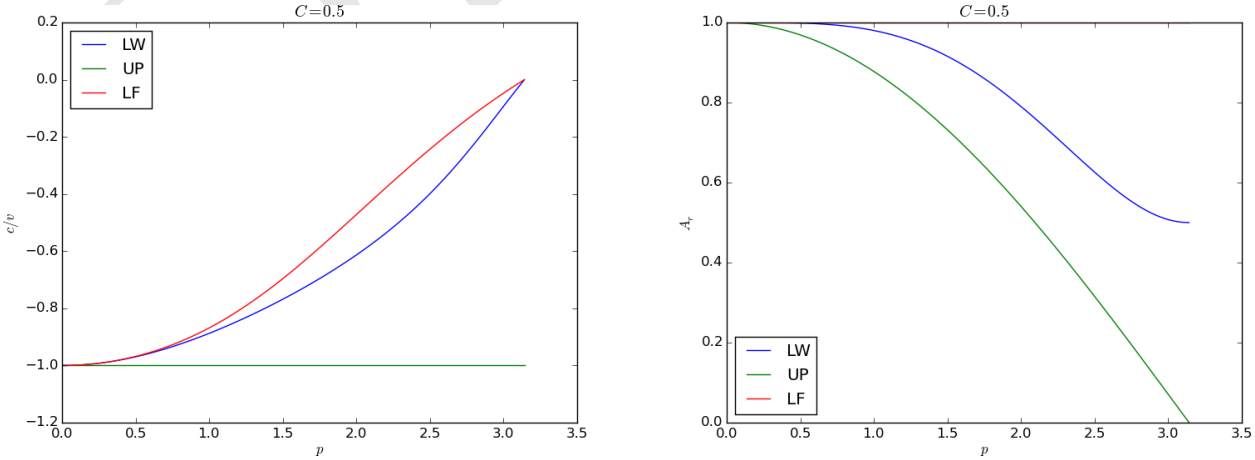


Figure 4.12.: Dispersion relations for $C = 0.5$.

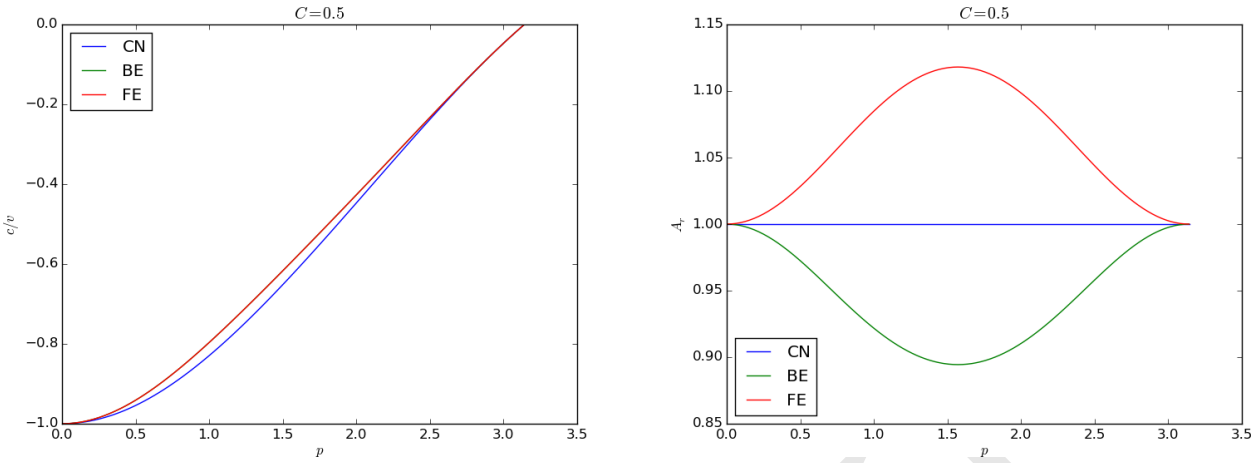


Figure 4.13.: Dispersion relations for $C = 0.5$.

increase C and/or the mesh resolution.

We can learn a lot from the dispersion relation plots. For example, looking at the plots for $C = 1$, the schemes LW, UP, and LF has no amplitude reduction, but LF has wrong phase velocity for the shortest wave in the mesh. This wave does not (normally) have enough amplitude to be seen, so for all practical purposes, there is no damping or wrong velocity of the individual waves, so the total shape of the wave is also correct. For the CN scheme, see Figure Figure 4.6, each individual wave has its amplitude, but they move with different velocities, so after a while, we see some of these waves lagging behind. For the BE scheme, see Figure Figure 4.7, all the shorter waves are so heavily dampened that we cannot see them after a while. We see only the longest waves, which have slightly wrong velocity, but visible amplitudes are sufficiently equal to produce what looks like a smooth profile.

Another feature was that the Leapfrog method produced oscillations, while the upwind scheme did not. Since the Leapfrog method does not dampen the shorter waves, which have wrong wave velocities of order 10 percent, we can see these waves as noise. The upwind scheme, however, dampens these waves. The same effect is also present in the Lax-Wendroff scheme, but the damping of the intermediate waves is hardly present, so there is visible noise in the total signal.

We realize that, compared to pure truncation error analysis, dispersion analysis sheds more light on the behavior of the computational schemes. Truncation analysis just says that Lax-Wendroff is better than upwind, because of the increased order in time, but most people would say upwind is the better one when looking at the plots.

4.11. Stationary 1D advection-diffusion

Now we pay attention to a physical process where advection (or convection) is in balance with diffusion:

$$v\frac{du}{dx} = \alpha\frac{d^2u}{dx^2}\,. \tag{4.9}$$

For simplicity, we assume v and α to be constant, but the extension to the variable-coefficient case is trivial. This equation can be viewed as the stationary limit of the corresponding time-dependent

problem

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = \alpha\frac{\partial^2 u}{\partial x^2}\,. \tag{4.10}$$

Equations of the form (4.9) or (4.10) arise from transport phenomena, either mass or heat transport. One can also view the equations as a simple model problem for the Navier-Stokes equations. With the chosen boundary conditions, the differential equation problem models the phenomenon of a *boundary layer*, where the solution changes rapidly very close to the boundary. This is a characteristic of many fluid flow problems, which makes strong demands to numerical methods. The fundamental numerical difficulty is related to non-physical oscillations of the solution (instability) if the first-derivative spatial term dominates over the second-derivative term.

4.12. A simple model problem

We consider (4.9) on $[0, L]$ equipped with the boundary conditions $u(0) = U_0$, $u(L) = U_L$. By scaling we can reduce the number of parameters in the problem. We scale x by $\bar{x} = x/L$, and u by

$$\bar{u} = \frac{u - U_0}{U_L - U_0}\,.$$

Inserted in the governing equation we get

$$\frac{v(U_L - U_0)}{L}\frac{d\bar{u}}{d\bar{x}} = \frac{\alpha(U_L - U_0)}{L^2}\frac{d^2\bar{u}}{d\bar{x}^2}, \quad \bar{u}(0) = 0,\ \bar{u}(1) = 1\,.$$

Dropping the bars is common. We can then simplify to

$$\frac{du}{dx} = \epsilon\frac{d^2u}{dx^2}, \quad u(0) = 0,\ u(1) = 1\,. \tag{4.11}$$

There are two competing effects in this equation: the advection term transports signals to the right, while the diffusion term transports signals to the left and the right. The value $u(0) = 0$ is transported through the domain if ϵ is small, and $u \approx 0$ except in the vicinity of $x = 1$, where $u(1) = 1$ and the diffusion transports some information about $u(1) = 1$ to the left. For large ϵ, diffusion dominates and the u takes on the "average" value, i.e., u gets a linear variation from 0 to 1 throughout the domain.

It turns out that we can find an exact solution to the differential equation problem and also to many of its discretizations. This is one reason why this model problem has been so successful in designing and investigating numerical methods for mixed convection/advection and diffusion. The exact solution reads

$$u_{\mathrm{e}}(x) = \frac{e^{x/\epsilon} - 1}{e^{1/\epsilon} - 1}\,.$$

The forthcoming plots illustrate this function for various values of ϵ.

4.13. A centered finite difference scheme

The most obvious idea to solve (4.11) is to apply centered differences:

$$[D_{2x}u = \epsilon D_x D_x u]_i$$

for $i = 1, \ldots, N_x - 1$, with $u_0 = 0$ and $u_{N_x} = 1$. Note that this is a coupled system of algebraic equations involving $u_0, \ldots, u_{N_x}$.

Written out, the scheme becomes a tridiagonal system

$$A_{i-1,i}u_{i-1} + A_{i,i}u_i + A_{i+1.i}u_{i+1} = 0,$$

for $i = 1, \ldots, N_x - 1$

$$\begin{aligned} A_{0,0} &= 1, \\ A_{i-1,i} &= -\frac{1}{\Delta x} - \epsilon\frac{1}{\Delta x^2}, \\ A_{i,i} &= 2\epsilon\frac{1}{\Delta x^2}, \\ A_{i,i+1} &= \frac{1}{\Delta x} - \epsilon\frac{1}{\Delta x^2}, \\ A_{N_x,N_x} &= 1\,. \end{aligned}$$

The right-hand side of the linear system is zero except $b_{N_x} = 1$.

Figure Figure 4.14 shows reasonably accurate results with $N_x=20 $ and $N_x = 40$ cells in x direction and a value of $\epsilon = 0.1$. Decreasing ϵ to 0.01 leads to oscillatory solutions as depicted in Figure Figure 4.15. This is, unfortunately, a typical phenomenon in this type of problem: non-physical oscillations arise for small ϵ unless the resolution N_x is big enough. Exercise Section 4.18 develops a precise criterion: u is oscillation-free if

$$\Delta x \leq \frac{2}{\epsilon}\,.$$

If we take the present model as a simplified model for a *viscous boundary layer* in real, industrial fluid flow applications, $\epsilon \sim 10^{-6}$ and millions of cells are required to resolve the boundary layer. Fortunately, this is not strictly necessary as we have methods in the next section to overcome the problem!

> **Solver**
>
> A suitable solver for doing the experiments is presented below.

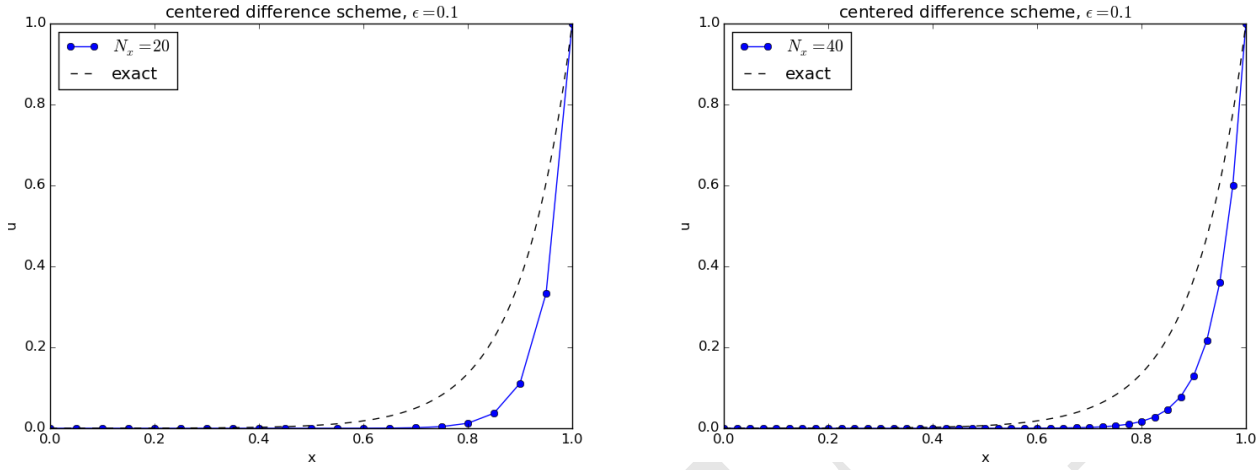


Figure 4.14.: Comparison of exact and numerical solution for $\epsilon = 0.1$ and $N_x = 20, 40$ with centered differences.

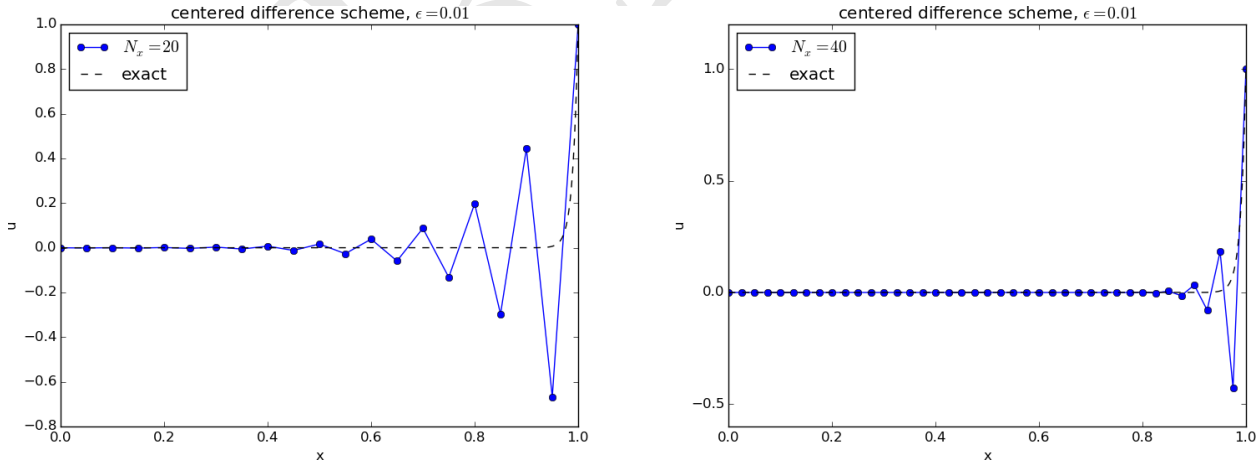


Figure 4.15.: Comparison of exact and numerical solution for $\epsilon = 0.01$ and $N_x = 20, 40$ with centered differences.

```
import numpy as np

def solver(eps, Nx, method="centered"):
    """
    Solver for the two point boundary value problem u'=eps*u'',
    u(0)=0, u(1)=1.
    """
    x = np.linspace(0, 1, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    u = np.zeros(Nx + 1)

    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    if method == "centered":
        diagonal[:] = 2 * eps / dx**2
        lower[:] = -1 / dx - eps / dx**2
        upper[:] = 1 / dx - eps / dx**2
    elif method == "upwind":
        diagonal[:] = 1 / dx + 2 * eps / dx**2
        lower[:] = 1 / dx - eps / dx**2
        upper[:] = -eps / dx**2

    upper[0] = 0
    lower[-1] = 0
    diagonal[0] = diagonal[-1] = 1
    b[-1] = 1.0

    diags = [0, -1, 1]
    import scipy.sparse
    import scipy.sparse.linalg

    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )
    u[:] = scipy.sparse.linalg.spsolve(A, b)
    return u, x
```

4.14. Remedy: upwind finite difference scheme

The scheme can be stabilized by letting the advective transport term, which is the dominating term, collect its information in the flow direction, i.e., upstream or upwind of the point in question. So, instead of using a centered difference

$$\frac{du}{dx} * *i \approx \frac{u * *i + 1 - u_{i-1}}{2\Delta x},$$

we use the one-sided *upwind* difference

$$\frac{du}{dx} * *i \approx \frac{u * *i - u_{i-1}}{\Delta x},$$

in case $v > 0$. For $v < 0$ we set

$$\frac{du}{dx} * *i \approx \frac{u * *i + 1 - u_i}{\Delta x},$$

On compact operator notation form, our upwind scheme can be expressed as

$$[D_x^- u = \epsilon D_x D_x u]_i$$

provided $v > 0$ (and $\epsilon > 0$).

We write out the equations and implement them as shown in the program in Section Section 4.13. The results appear in Figures Figure 4.16 and Figure 4.17: no more oscillations!

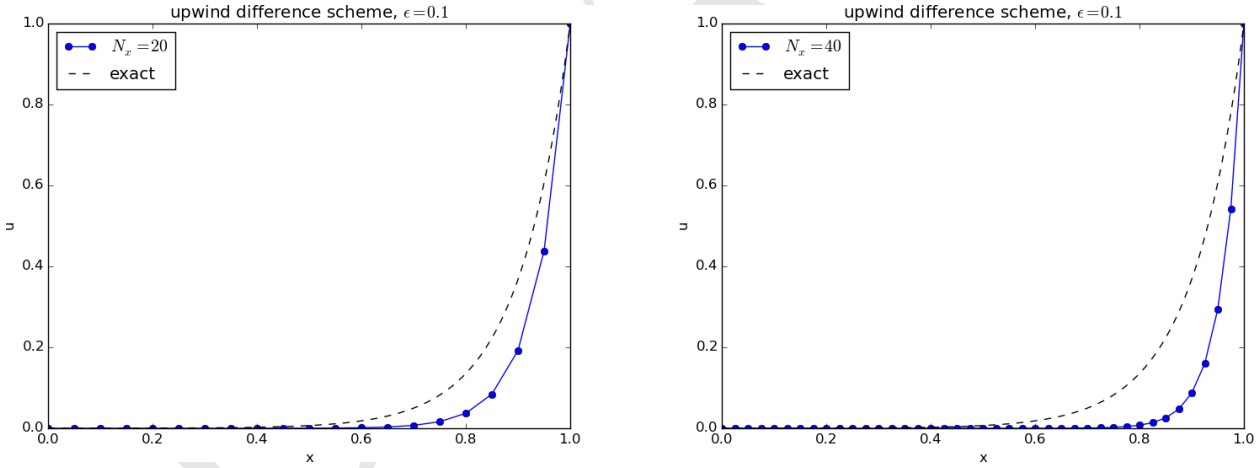


Figure 4.16.: Comparison of exact and numerical solution for $\epsilon = 0.1$ and $N_x = 20, 40$ with upwind difference.

We see that the upwind scheme is always stable, but it gives a thicker boundary layer when the centered scheme is also stable. Why the upwind scheme is always stable is easy to understand as soon as we undertake the mathematical analysis in Exercise Section 4.18. Moreover, the thicker layer (seemingly larger diffusion) can be understood by doing Exercise Section 4.19.

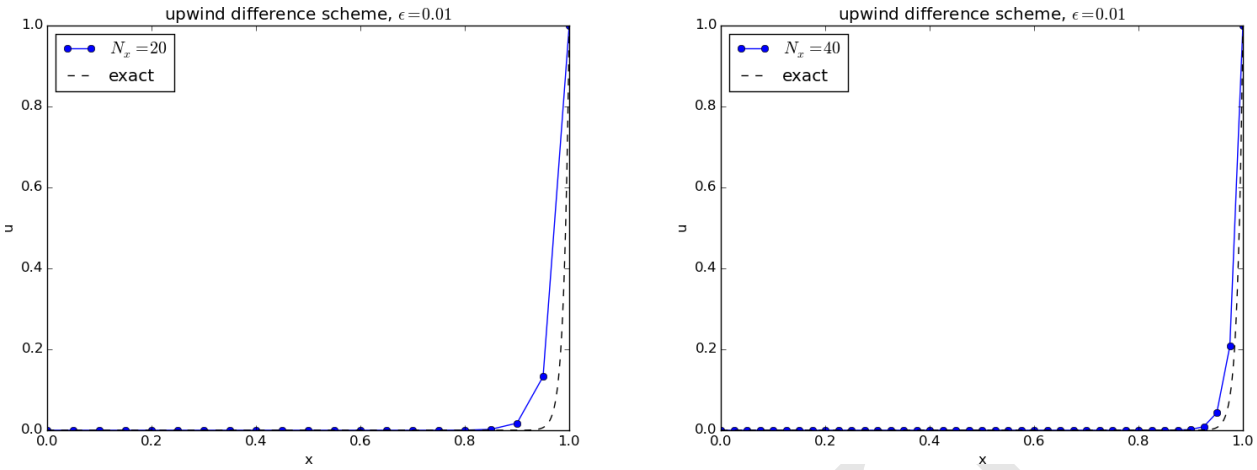


Figure 4.17.: Comparison of exact and numerical solution for $\epsilon = 0.01$ and $N_x = 20, 40$ with upwind difference.

Exact solution for this model problem

It turns out that one can introduce a linear combination of the centered and upwind differences for the first-derivative term in this model problem. One can then adjust the weight in the linear combination so that the numerical solution becomes identical to the analytical solution of the differential equation problem at any mesh point.

Now it is time to combine time-dependency, convection (advection) and diffusion into one equation:

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = \alpha\frac{\partial^2 u}{\partial x^2}\,. \tag{4.12}$$

4.14.1. Analytical insight

The diffusion is now dominated by convection, a wave, and diffusion, a loss of amplitude. One possible analytical solution is a traveling Gaussian function

$$u(x,t) = B\exp\left(-\left(\frac{x-vt}{4at}\right)\right).$$

This function moves with velocity $v > 0$ to the right ($v < 0$ to the left) due to convection, but at the same time we have a damping $e^{-16a^2t^2}$ from diffusion.

4.15. Forward in time, centered in space scheme

The Forward Euler for the diffusion equation is a successful scheme, but it has a very strict stability condition. The similar Forward in time, centered in space strategy always gives unstable solutions for the advection PDE. What happens when we have both diffusion and advection present at once?

$$[D_t u + vD_{2x}u = \alpha D_x D_x u + f]_i^n\,.$$

We expect that diffusion will stabilize the scheme, but that advection will destabilize it.

Another problem is non-physical oscillations, but not growing amplitudes, due to centered differences in the advection term. There will hence be two types of instabilities to consider. Our analysis showed that pure advection with centered differences in space needs some artificial diffusion to become stable (and then it produces upwind differences for the advection term). Adding more physical diffusion should further help the numerics to stabilize the non-physical oscillations.

The scheme is quickly implemented, but suffers from the need for small space and time steps, according to this reasoning. A better approach is to get rid of the non-physical oscillations in space by simply applying an upwind difference on the advection term.

4.16. Forward in time, upwind in space scheme

A good approximation for the pure advection equation is to use upwind discretization of the advection term. We also know that centered differences are good for the diffusion term, so let us combine these two discretizations:

$$[D_t u + vD_x^- u = \alpha D_x D_x u + f]_i^n,$$

for $v > 0$. Use vD^+u if $v < 0$. In this case the physical diffusion and the extra numerical diffusion $v\Delta x/2$ will stabilize the solution, but give an overall too large reduction in amplitude compared with the exact solution.

We may also interpret the upwind difference as artificial numerical diffusion and centered differences in space everywhere, so the scheme can be expressed as

$$[D_t u + vD_{2x}^- u = \alpha \frac{v\Delta x}{2} D_x D_x u + f]_i^n \,.$$

4.17. Applications of advection equations

There are two major areas where advection and convection applications arise: transport of a substance and heat transport *in a fluid*. To derive the models, we may look at the similar derivations of diffusion models in Section Section 3.66, but change the assumption from a solid to fluid medium. This gives rise to the extra advection or convection term $\boldsymbol{v}\cdot\nabla u$. We briefly show how this is done.

Normally, transport in a fluid is dominated by the fluid flow and not diffusion, so we can neglect diffusion compared to advection or convection. The end result is anyway an equation of the form

$$\frac{\partial u}{\partial t} + \boldsymbol{v}\cdot\nabla u = 0\,.$$

Transport of a substance {#sec-advec-app-mass}

The diffusion of a substance in Section Section 3.66.1 takes place in a solid medium, but in a fluid we can have two transport mechanisms: one by diffusion and one by advection. The latter arises from the fact that the substance particles are moved with the fluid velocity $\boldsymbol{v}$ such that the effective flux now consists of two and not only one component as in (3.96):

$$\boldsymbol{q} = -\alpha\nabla c + \boldsymbol{v}_s$$

Inserted in the equation $\nabla \cdot \boldsymbol{q} = 0$ we get the extra advection term $\nabla \cdot (\boldsymbol{v}$. Very often we deal with incompressible flows, $\nabla \cdot \boldsymbol{v} = 0$ such that the advective term becomes $\boldsymbol{v} \cdot \nabla c$. The mass transport equation for a substance then reads

$$\frac{\partial c}{\partial t} + \boldsymbol{v} \cdot \nabla c = \alpha \nabla^2 c\,.$$

Transport of heat in fluids {#sec-advec-app-heat}

The derivation of the heat equation in Section Section 3.66.2 is limited to heat transport in solid bodies. If we turn the attention to heat transport in fluids, we get a material derivative of the internal energy in (3.98),

$$\frac{De}{dt} = -\nabla \cdot \boldsymbol{q},$$

and more terms if work by stresses is also included, where

$$\frac{De}{dt} = \frac{\partial e}{\partial t} + \boldsymbol{v} \cdot \nabla e,$$

$\boldsymbol{v}$ being the velocity of the fluid. The convective term $\boldsymbol{v} \cdot \nabla e$ must therefore be added to the governing equation, resulting typically in

$$\varrho c \left(\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T \right) = \nabla \cdot (k \nabla T) + f, \tag{4.13}$$

where f is some external heating inside the medium.

4.18. Exercise: Analyze 1D stationary convection-diffusion problem

Explain the observations in the numerical experiments from Sections Section 4.13 and Section 4.14 by finding exact numerical solutions.

The difference equations allow solutions on the form A^i, where

A is an unknown constant and i is a mesh point counter. There are two solutions for A, so the general solution is a linear combination of the two, where the constants in the linear combination are determined from the boundary conditions.

4.19. Exercise: Interpret upwind difference as artificial diffusion

Consider an upwind, one-sided difference approximation to a term du/dx in a differential equation. Show that this formula can be expressed as a centered difference plus an artificial diffusion term of strength proportional to Δx. This means that introducing an upwind difference also means introducing extra diffusion of order $\mathcal{O}(\Delta x)$.

4.20. Advection Schemes with Devito

Having understood the mathematical properties and challenges of advection schemes in the previous sections, we now implement these methods using Devito's symbolic framework. Devito allows us to write the discrete equations in a form close to the mathematical notation while generating optimized code automatically.

4.20.1. The Advection Equation

The 1D linear advection equation is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{4.14}$$

where c is the advection velocity (assumed constant and positive). The exact solution is:

$$u(x,t) = I(x - ct)$$

which represents the initial condition $I(x)$ traveling to the right at velocity c without change in shape.

4.20.2. Devito Implementation Patterns

Unlike diffusion and wave equations, the advection equation requires careful treatment of the spatial derivative. Centered differences lead to instability (as we saw with the FTCS scheme), so we need alternative approaches:

Scheme	Spatial Discretization	Order	Key Property
Upwind	Backward difference	1st	Stable, diffusive
Lax-Wendroff	Centered + diffusion	2nd	Less diffusion, some dispersion
Lax-Friedrichs	Averaged neighbors	1st	Very diffusive but robust

All schemes require the CFL condition: $C = c\Delta t/\Delta x \leq 1$.

Property	Diffusion	Wave	Advection

4.20.3. Comparison with Wave and Diffusion Equations

The advection equation differs fundamentally from the diffusion and wave equations we've solved previously:

Property	Diffusion	Wave	Advection
`time_order`	1	2	1
Spatial deriv.	2nd (`.dx2`)	2nd (`.laplace`)	1st (`.dx`)
Stability	$F \leq 0.5$	$C \leq 1$	$C \leq 1$
Centered space	Stable	Stable	**Unstable**
Information	Spreads both ways	Spreads both ways	One direction

The key difference is that advection has directional information flow, which requires using *upwind* differences rather than centered differences.

4.20.4. Upwind Scheme Implementation

The upwind scheme uses a backward difference for the spatial derivative when $c > 0$:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0$$

which gives the update formula:

$$u_i^{n+1} = u_i^n - C(u_i^n - u_{i-1}^n) \tag{4.15}$$

In Devito, we express this using shifted indexing:

```
from devito import Grid, TimeFunction, Eq, Operator, Constant
import numpy as np

def solve_advection_upwind(L, c, Nx, T, C, I):
    """Upwind scheme for 1D advection."""
    # Grid setup
    dx = L / Nx
    dt = C * dx / c

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    x_dim, = grid.dimensions

    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=1)
```

```
    # Set initial condition
    x_coords = np.linspace(0, L, Nx + 1)
    u.data[0, :] = I(x_coords)

    # Courant number as constant
    courant = Constant(name='C', value=C)

    # Upwind stencil: u^{n+1} = u - C*(u - u[x-dx])
    u_minus = u.subs(x_dim, x_dim - x_dim.spacing)
    stencil = u - courant * (u - u_minus)
    update = Eq(u.forward, stencil)

    op = Operator([update])
    # ... time stepping loop
```

The key line is:

```
u_minus = u.subs(x_dim, x_dim - x_dim.spacing)
```

This creates a reference to u_{i-1}^n by substituting `x_dim - x_dim.spacing` for `x_dim` in the `TimeFunction` `u`.

4.20.5. Lax-Wendroff Scheme Implementation

The Lax-Wendroff scheme achieves second-order accuracy by including both a centered advection term and a diffusion-like correction:

$$u_i^{n+1} = u_i^n - \frac{C}{2}(u_{i+1}^n - u_{i-1}^n) + \frac{C^2}{2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

This can be written using Devito's derivative operators:

```
def solve_advection_lax_wendroff(L, c, Nx, T, C, I):
    """Lax-Wendroff scheme for 1D advection."""
    dx = L / Nx
    dt = C * dx / c

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

    x_coords = np.linspace(0, L, Nx + 1)
    u.data[0, :] = I(x_coords)

    courant = Constant(name='C', value=C)

    # Lax-Wendroff: u - (C/2)*dx*u.dx + (C²/2)*dx²*u.dx2
```

```
    # u.dx  = centered first derivative
    # u.dx2 = centered second derivative
    stencil = u - 0.5*courant*dx*u.dx + 0.5*courant**2*dx**2*u.dx2
    update = Eq(u.forward, stencil)

    op = Operator([update])
    # ... time stepping loop
```

Here we use Devito's built-in derivative operators:

- `u.dx` computes the centered first derivative $(u_{i+1} - u_{i-1})/(2\Delta x)$
- `u.dx2` computes the centered second derivative $(u_{i+1} - 2u_i + u_{i-1})/\Delta x^2$

4.20.6. Lax-Friedrichs Scheme Implementation

The Lax-Friedrichs scheme is simpler but more diffusive:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{C}{2}(u_{i+1}^n - u_{i-1}^n)$$

```
def solve_advection_lax_friedrichs(L, c, Nx, T, C, I):
    """Lax-Friedrichs scheme for 1D advection."""
    dx = L / Nx
    dt = C * dx / c

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    x_dim, = grid.dimensions

    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=1)

    x_coords = np.linspace(0, L, Nx + 1)
    u.data[0, :] = I(x_coords)

    courant = Constant(name='C', value=C)

    # Neighbor values
    u_plus = u.subs(x_dim, x_dim + x_dim.spacing)
    u_minus = u.subs(x_dim, x_dim - x_dim.spacing)

    # Lax-Friedrichs stencil
    stencil = 0.5*(u_plus + u_minus) - 0.5*courant*(u_plus - u_minus)
    update = Eq(u.forward, stencil)

    op = Operator([update])
    # ... time stepping loop
```

4.20.7. Periodic Boundary Conditions

For advection problems, periodic boundary conditions are often useful to study wave propagation without boundary effects:

```
t_dim = grid.stepping_dim

# Periodic BC: u[0] wraps to u[Nx], u[Nx] wraps to u[0]
bc_left = Eq(u[t_dim + 1, 0], u[t_dim, Nx])
bc_right = Eq(u[t_dim + 1, Nx], u[t_dim + 1, 0])

op = Operator([update, bc_left, bc_right])
```

4.20.8. Using the Solvers

The complete solver implementation in `src/advec/advec1D_devito.py` provides convenient interfaces:

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs,
    exact_advection_periodic
)
import numpy as np

# Define initial condition
def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

# Solve with upwind scheme
result = solve_advection_upwind(
    L=1.0, c=1.0, Nx=100, T=0.5, C=0.8, I=I,
    periodic_bc=True
)

# Compare with exact solution
u_exact = exact_advection_periodic(result.x, result.t, c=1.0, L=1.0, I=I)
error = np.max(np.abs(result.u - u_exact))
print(f"Max error: {error:.6f}")
```

4.20.9. Scheme Comparison

The three schemes exhibit different numerical behaviors:

```
import matplotlib.pyplot as plt
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs,
    exact_advection_periodic
)
import numpy as np

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

L, c, Nx, T, C = 1.0, 1.0, 50, 0.5, 0.8

# Solve with all three schemes
r_upwind = solve_advection_upwind(L, c, Nx, T, C, I, periodic_bc=True)
r_lw = solve_advection_lax_wendroff(L, c, Nx, T, C, I, periodic_bc=True)
r_lf = solve_advection_lax_friedrichs(L, c, Nx, T, C, I, periodic_bc=True)

# Exact solution
u_exact = exact_advection_periodic(r_upwind.x, r_upwind.t, c, L, I)

plt.figure(figsize=(10, 6))
plt.plot(r_upwind.x, u_exact, 'k-', lw=2, label='Exact')
plt.plot(r_upwind.x, r_upwind.u, 'b--', label='Upwind')
plt.plot(r_lw.x, r_lw.u, 'r-.', label='Lax-Wendroff')
plt.plot(r_lf.x, r_lf.u, 'g:', label='Lax-Friedrichs')
plt.legend()
plt.xlabel('x')
plt.ylabel('u')
plt.title(f'Advection: Nx={Nx}, C={C}, T={T}')
plt.savefig('advec_scheme_comparison.pdf')
```

The Lax-Wendroff scheme typically preserves the wave amplitude better but may show small oscillations. The upwind and Lax-Friedrichs schemes are more diffusive, causing the wave to spread and reduce in amplitude.

4.20.10. Convergence Testing

We can verify the convergence rates of the schemes:

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    convergence_test_advection
)
```

```
# Test upwind (expect 1st order)
sizes, errors, rate = convergence_test_advection(
    solve_advection_upwind,
    grid_sizes=[25, 50, 100, 200],
    T=0.25, C=0.8
)
print(f"Upwind convergence rate: {rate:.2f}")  # ~1.0

# Test Lax-Wendroff (expect 2nd order)
sizes, errors, rate = convergence_test_advection(
    solve_advection_lax_wendroff,
    grid_sizes=[25, 50, 100, 200],
    T=0.25, C=0.8
)
print(f"Lax-Wendroff convergence rate: {rate:.2f}")  # ~2.0
```

4.20.11. Key Takeaways

1. **Upwind differencing** is essential for stable advection schemes—centered differences in space are unconditionally unstable.
2. **The Courant number** $C = c\Delta t/\Delta x$ controls stability; all schemes require $C \leq 1$.
3. **Trade-offs exist** between accuracy and numerical diffusion:
   - Upwind: Stable, 1st order, diffusive
   - Lax-Wendroff: 2nd order, less diffusion, may have small oscillations
   - Lax-Friedrichs: Very stable, very diffusive
4. **Devito's shifted indexing** via `u.subs(x_dim, x_dim - x_dim.spacing)` allows expressing upwind differences naturally.
5. **Periodic BCs** are implemented by explicitly setting boundary equations that copy values from the opposite end of the domain.

4.21. Exercises: Advection with Devito

4.21.1. Exercise 1: Verify CFL Stability Condition

The upwind scheme requires $C \leq 1$ for stability.

a) Run the upwind solver with $C = 0.5$, $C = 0.9$, and $C = 1.0$ for $T = 1.0$ with a Gaussian initial condition. Verify that all solutions remain bounded.

b) Try $C = 1.01$ and observe what happens. How quickly does the instability grow?

c) For $C = 1.0$ exactly, the upwind scheme should reproduce the exact solution (up to machine precision). Verify this numerically.

Solution

```
from src.advec import solve_advection_upwind, exact_advection_periodic
import numpy as np

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

# Part a: Stable Courant numbers
for C in [0.5, 0.9, 1.0]:
    result = solve_advection_upwind(
        L=1.0, c=1.0, Nx=100, T=1.0, C=C, I=I
    )
    print(f"C={C}: u in [{result.u.min():.4f}, {result.u.max():.4f}]")

# Part b: Slightly unstable
# This will raise ValueError since C > 1 violates stability
try:
    result = solve_advection_upwind(
        L=1.0, c=1.0, Nx=100, T=1.0, C=1.01, I=I
    )
except ValueError as e:
    print(f"Error: {e}")

# Part c: Exact at C=1
result = solve_advection_upwind(
    L=1.0, c=1.0, Nx=100, T=0.5, C=1.0, I=I, periodic_bc=True
)
u_exact = exact_advection_periodic(result.x, result.t, 1.0, 1.0, I)
error = np.max(np.abs(result.u - u_exact))
print(f"Error at C=1: {error:.2e}")  # Should be ~machine precision
```

4.21.2. Exercise 2: Compare Numerical Diffusion

The upwind scheme introduces numerical diffusion that causes the wave amplitude to decrease over time.

a) Run all three schemes (upwind, Lax-Wendroff, Lax-Friedrichs) with $C = 0.8$ for $T = 2.0$ and track the maximum value of u over time.

b) Plot the amplitude decay for each scheme. Which scheme preserves the amplitude best?

c) For the Gaussian initial condition, measure the "width" of the pulse (e.g., the distance between points where $u = 0.5 \max(u)$) at $T = 0$ and $T = 2$. How much has each scheme spread the pulse?

Solution

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs
)
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

# Run all schemes with history
L, c, Nx, T, C = 1.0, 1.0, 100, 2.0, 0.8

r_up = solve_advection_upwind(L, c, Nx, T, C, I, save_history=True)
r_lw = solve_advection_lax_wendroff(L, c, Nx, T, C, I, save_history=True)
r_lf = solve_advection_lax_friedrichs(L, c, Nx, T, C, I, save_history=True)

# Part b: Track amplitude decay
max_up = [np.max(u) for u in r_up.u_history]
max_lw = [np.max(u) for u in r_lw.u_history]
max_lf = [np.max(u) for u in r_lf.u_history]

plt.figure()
plt.plot(r_up.t_history, max_up, 'b-', label='Upwind')
plt.plot(r_lw.t_history, max_lw, 'r--', label='Lax-Wendroff')
plt.plot(r_lf.t_history, max_lf, 'g-.', label='Lax-Friedrichs')
plt.axhline(1.0, color='k', linestyle=':', label='Exact')
plt.xlabel('Time')
plt.ylabel('Max amplitude')
plt.legend()
plt.title('Amplitude decay comparison')
plt.savefig('amplitude_decay.pdf')

# Part c: Measure pulse width at half-maximum
def half_width(u, x):
    u_max = np.max(u)
    half_max = 0.5 * u_max
    above = np.where(u >= half_max)[0]
    if len(above) > 0:
        return x[above[-1]] - x[above[0]]
    return 0

print("Initial width:", half_width(I(r_up.x), r_up.x))
print("Upwind width:", half_width(r_up.u, r_up.x))
print("Lax-Wendroff width:", half_width(r_lw.u, r_lw.x))
print("Lax-Friedrichs width:", half_width(r_lf.u, r_lf.x))
```

4.21.3. Exercise 3: Convergence Rate Verification

Verify the theoretical convergence rates: - Upwind: 1st order - Lax-Wendroff: 2nd order - Lax-Friedrichs: 1st order

a) Use the `convergence_test_advection` function with grid sizes [25, 50, 100, 200, 400] and verify the rates.

b) Create a log-log plot of error vs grid size for all three schemes.

c) What happens to the convergence rate if you use a discontinuous initial condition (step function) instead of the smooth Gaussian?

Solution

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs,
    convergence_test_advection
)
import numpy as np
import matplotlib.pyplot as plt

# Part a: Verify rates
grid_sizes = [25, 50, 100, 200, 400]

sizes_up, err_up, rate_up = convergence_test_advection(
    solve_advection_upwind, grid_sizes, T=0.25, C=0.8
)
print(f"Upwind rate: {rate_up:.2f}")

sizes_lw, err_lw, rate_lw = convergence_test_advection(
    solve_advection_lax_wendroff, grid_sizes, T=0.25, C=0.8
)
print(f"Lax-Wendroff rate: {rate_lw:.2f}")

sizes_lf, err_lf, rate_lf = convergence_test_advection(
    solve_advection_lax_friedrichs, grid_sizes, T=0.25, C=0.8
)
print(f"Lax-Friedrichs rate: {rate_lf:.2f}")

# Part b: Log-log plot
plt.figure()
plt.loglog(sizes_up, err_up, 'b-o', label=f'Upwind (rate={rate_up:.2f})')
plt.loglog(sizes_lw, err_lw, 'r-s', label=f'Lax-Wendroff (rate={rate_lw:.2f})')
plt.loglog(sizes_lf, err_lf, 'g-^', label=f'Lax-Friedrichs (rate={rate_lf:.2f})')

# Reference slopes
h = np.array(sizes_up)
plt.loglog(h, err_up[0]*(h[0]/h), 'k--', alpha=0.5, label='O(h)')
plt.loglog(h, err_lw[0]*(h[0]/h)**2, 'k:', alpha=0.5, label='O(h²)')

plt.xlabel('Grid points')
plt.ylabel('L2 Error')
plt.legend()
plt.title('Convergence comparison')
plt.gca().invert_xaxis()
plt.savefig('convergence_advec.pdf')
```

4.21.4. Exercise 4: Step Function Advection

A step (Heaviside) function is a challenging test case for advection schemes because of the discontinuity.

a) Advect a step function from $x = 0.25$ using all three schemes with $C = 0.8$ and $\Delta x = 0.01$. Compare the results at $T = 0.5$.

b) The Lax-Wendroff scheme may show oscillations near the discontinuity (Gibbs phenomenon). Observe and document this behavior.

c) How does the upwind scheme handle the step? Does it preserve the sharp transition?

Solution

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs,
    step_initial_condition,
    exact_advection_periodic
)
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.where(x < 0.25, 1.0, 0.0)

L, c, Nx, T, C = 1.0, 1.0, 100, 0.5, 0.8

r_up = solve_advection_upwind(L, c, Nx, T, C, I, periodic_bc=True)
r_lw = solve_advection_lax_wendroff(L, c, Nx, T, C, I, periodic_bc=True)
r_lf = solve_advection_lax_friedrichs(L, c, Nx, T, C, I, periodic_bc=True)

u_exact = exact_advection_periodic(r_up.x, r_up.t, c, L, I)

plt.figure(figsize=(10, 6))
plt.plot(r_up.x, u_exact, 'k-', lw=2, label='Exact')
plt.plot(r_up.x, r_up.u, 'b--', label='Upwind')
plt.plot(r_lw.x, r_lw.u, 'r-.', label='Lax-Wendroff')
plt.plot(r_lf.x, r_lf.u, 'g:', label='Lax-Friedrichs')
plt.legend()
plt.xlabel('x')
plt.ylabel('u')
plt.title('Step function advection')
plt.ylim(-0.2, 1.3)
plt.savefig('step_advection.pdf')

# Note Lax-Wendroff oscillations near discontinuity
```

4.21.5. Exercise 5: Long-Time Integration

With periodic boundary conditions, a wave should return to its starting position after traveling one domain length.

a) Advect a Gaussian pulse for $T = 1.0$ (one complete cycle with $c = 1$, $L = 1$) and compare the final solution to the initial condition.

b) Run for $T = 10.0$ (10 cycles) and measure how much the amplitude has decayed for each scheme.

c) For each scheme, estimate after how many cycles the peak amplitude drops to 50% of its initial value.

Solution

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs
)
import numpy as np

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

L, c, Nx, C = 1.0, 1.0, 100, 0.8

# Part a: One cycle
for T in [1.0, 10.0]:
    r_up = solve_advection_upwind(L, c, Nx, T, C, I, periodic_bc=True)
    r_lw = solve_advection_lax_wendroff(L, c, Nx, T, C, I, periodic_bc=True)
    r_lf = solve_advection_lax_friedrichs(L, c, Nx, T, C, I, periodic_bc=True)

    print(f"\nT = {T} ({int(T)} cycles):")
    print(f"  Upwind: max = {r_up.u.max():.4f}")
    print(f"  Lax-Wendroff: max = {r_lw.u.max():.4f}")
    print(f"  Lax-Friedrichs: max = {r_lf.u.max():.4f}")

# Part c: Find half-life
def find_halflife(solver_func, L, c, Nx, C, I, max_cycles=100):
    for n in range(1, max_cycles + 1):
        T = float(n)
        result = solver_func(L, c, Nx, T, C, I, periodic_bc=True)
        if result.u.max() < 0.5:
            return n
    return max_cycles

print("\nCycles to 50% amplitude:")
print(f"  Upwind: {find_halflife(solve_advection_upwind, L, c, Nx, C, I)}")
print(f"  Lax-Wendroff: {find_halflife(solve_advection_lax_wendroff, L, c, Nx, C, I)}")
print(f"  Lax-Friedrichs: {find_halflife(solve_advection_lax_friedrichs, L, c, Nx, C, I)}")
```

4.21.6. Exercise 6: Effect of Courant Number

The Courant number C affects both stability and accuracy.

a) For the upwind scheme, run with $C = 0.2$, 0.5, 0.8, and 1.0 for $T = 1.0$. Plot the final solutions on the same figure.

b) Which value of C gives the best accuracy? Why?

c) Measure the L2 error for each C value and create a plot of error vs. C.

Solution

```
from src.advec import solve_advection_upwind, exact_advection_periodic
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

L, c, Nx, T = 1.0, 1.0, 100, 1.0
C_values = [0.2, 0.5, 0.8, 1.0]

plt.figure(figsize=(10, 6))

errors = []
for C in C_values:
    result = solve_advection_upwind(L, c, Nx, T, C, I, periodic_bc=True)
    plt.plot(result.x, result.u, label=f'C={C}')

    u_exact = exact_advection_periodic(result.x, result.t, c, L, I)
    dx = L / Nx
    error = np.sqrt(dx * np.sum((result.u - u_exact)**2))
    errors.append(error)

# Add exact solution
u_exact = exact_advection_periodic(result.x, T, c, L, I)
plt.plot(result.x, u_exact, 'k--', lw=2, label='Exact')

plt.legend()
plt.xlabel('x')
plt.ylabel('u')
plt.title('Effect of Courant number on upwind scheme')
plt.savefig('courant_effect.pdf')

# Error vs C
plt.figure()
plt.plot(C_values, errors, 'bo-')
plt.xlabel('Courant number C')
plt.ylabel('L2 Error')
plt.title('Error vs Courant number (Upwind)')
plt.savefig('error_vs_courant.pdf')

# C=1 gives exact solution for upwind
print("Errors:", dict(zip(C_values, errors)))
```

4.21.7. Exercise 7: Variable Velocity Field

Modify the upwind solver to handle a spatially varying velocity $c(x)$.

a) Implement an upwind scheme for:

$$\frac{\partial u}{\partial t} + c(x)\frac{\partial u}{\partial x} = 0$$

where the local Courant number varies: $C_i = c(x_i)\Delta t/\Delta x$.

b) Test with $c(x) = 1 + 0.5\sin(2\pi x)$ and observe how the wave stretches and compresses as it moves through regions of different velocity.

Solution

```
from devito import Grid, TimeFunction, Function, Eq, Operator, Constant
import numpy as np
import matplotlib.pyplot as plt

def solve_advection_variable_c(L, c_func, Nx, T, dt, I):
    """Upwind scheme with spatially varying velocity."""
    grid = Grid(shape=(Nx + 1,), extent=(L,))
    x_dim, = grid.dimensions
    t_dim = grid.stepping_dim

    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=1)
    c = Function(name='c', grid=grid)

    x_coords = np.linspace(0, L, Nx + 1)
    u.data[0, :] = I(x_coords)
    c.data[:] = c_func(x_coords)

    dx = L / Nx
    dt_const = Constant(name='dt', value=dt)
    dx_const = Constant(name='dx', value=dx)

    # Local Courant number: C_i = c_i * dt / dx
    # Upwind: u^{n+1} = u - (c*dt/dx)*(u - u[x-dx])
    u_minus = u.subs(x_dim, x_dim - x_dim.spacing)
    stencil = u - (c * dt_const / dx_const) * (u - u_minus)
    update = Eq(u.forward, stencil)

    # Periodic BCs
    bc_left = Eq(u[t_dim + 1, 0], u[t_dim, Nx])
    bc_right = Eq(u[t_dim + 1, Nx], u[t_dim + 1, 0])

    op = Operator([update, bc_left, bc_right])

    Nt = int(round(T / dt))
    for n in range(Nt):
        op.apply(time_m=n, time_M=n, dt=dt)

    return u.data[Nt % 2, :].copy(), x_coords

# Test with variable velocity
def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

def c_var(x):
    return 1.0 + 0.5*np.sin(2*np.pi*x)

L, Nx = 1.0, 200
dx = L / Nx
c_max = 1.5  # max of c(x)
dt = 0.5 * dx / c_max  # ensure CFL < 1 everywhere
u_final, x = solve_advection_variable_c(L, c_var, Nx, T=1.0, dt=dt, I=I)
```

4.21.8. Exercise 8: Advection-Diffusion Equation

Combine advection and diffusion:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

a) Implement a solver using upwind for advection and centered differences for diffusion.

b) Compare the behavior for $\nu = 0$ (pure advection), $\nu = 0.01$ (advection-dominated), and $\nu = 0.1$ (diffusion-dominated).

c) What is the stability condition when both advection and diffusion are present?

Solution

```
from devito import Grid, TimeFunction, Eq, Operator, Constant
import numpy as np
import matplotlib.pyplot as plt

def solve_advec_diff(L, c, nu, Nx, T, C, I):
    """Advection-diffusion with upwind advection + centered diffusion."""
    dx = L / Nx

    # Stability requires both CFL and diffusion conditions
    dt_adv = C * dx / c if c > 0 else np.inf
    dt_diff = 0.4 * dx**2 / nu if nu > 0 else np.inf
    dt = min(dt_adv, dt_diff)

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    x_dim, = grid.dimensions
    t_dim = grid.stepping_dim

    u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)

    x_coords = np.linspace(0, L, Nx + 1)
    u.data[0, :] = I(x_coords)

    C_const = Constant(name='C', value=c * dt / dx)
    F_const = Constant(name='F', value=nu * dt / dx**2)

    # Upwind advection + centered diffusion
    u_minus = u.subs(x_dim, x_dim - x_dim.spacing)
    advection = C_const * (u - u_minus)
    diffusion = F_const * dx**2 * u.dx2

    stencil = u - advection + diffusion
    update = Eq(u.forward, stencil)

    # Periodic BCs
    bc_left = Eq(u[t_dim + 1, 0], u[t_dim, Nx])
    bc_right = Eq(u[t_dim + 1, Nx], u[t_dim + 1, 0])

    op = Operator([update, bc_left, bc_right])

    Nt = int(round(T / dt))
    for n in range(Nt):
        op.apply(time_m=n, time_M=n, dt=dt)

    return u.data[Nt % 2, :].copy(), x_coords

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)

L, c, Nx, T, C = 1.0, 1.0, 100, 0.5, 0.8

plt.figure(figsize=(10, 6))
```



```
for nu, style in [(0.0, 'b-'), (0.01, 'r--'), (0.1, 'g-.')]:
```

4.21.9. Exercise 9: Cosine Hat Initial Condition

The "cosine hat" is a smoother alternative to the step function:

$$I(x) = \begin{cases} \cos\left(\frac{5\pi}{L}(x - L/10)\right) & \text{if } x < L/5 \\ 0 & \text{otherwise} \end{cases}$$

a) Implement this initial condition and advect it using all three schemes.

b) Compare the behavior at the sharp cutoff ($x = L/5$) between schemes.

c) Does the Lax-Wendroff scheme show oscillations for this smoother discontinuity?

Solution

```
from src.advec import (
    solve_advection_upwind,
    solve_advection_lax_wendroff,
    solve_advection_lax_friedrichs
)
import numpy as np
import matplotlib.pyplot as plt

def cosine_hat(x, L=1.0):
    """Cosine hat initial condition."""
    result = np.zeros_like(x)
    mask = x < L/5
    result[mask] = np.cos(5*np.pi/L * (x[mask] - L/10))
    return result

def I(x):
    return cosine_hat(x, L=1.0)

L, c, Nx, T, C = 1.0, 1.0, 100, 0.5, 0.8

r_up = solve_advection_upwind(L, c, Nx, T, C, I, periodic_bc=True)
r_lw = solve_advection_lax_wendroff(L, c, Nx, T, C, I, periodic_bc=True)
r_lf = solve_advection_lax_friedrichs(L, c, Nx, T, C, I, periodic_bc=True)

plt.figure(figsize=(10, 6))
plt.plot(r_up.x, I(r_up.x - c*T), 'k-', lw=2, label='Exact')
plt.plot(r_up.x, r_up.u, 'b--', label='Upwind')
plt.plot(r_lw.x, r_lw.u, 'r-.', label='Lax-Wendroff')
plt.plot(r_lf.x, r_lf.u, 'g:', label='Lax-Friedrichs')
plt.legend()
plt.xlabel('x')
plt.ylabel('u')
plt.title('Cosine hat advection')
plt.savefig('cosinehat.pdf')
```

4.21.10. Exercise 10: Implement Leapfrog Scheme

The leapfrog scheme uses a two-level time difference:

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t} + c\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0$$

This is a three-time-level scheme requiring special initialization for u^1.

a) Implement the leapfrog scheme using Devito with `time_order=2`.

b) Use the upwind scheme to compute u^1 from u^0, then switch to leapfrog.

c) Compare the leapfrog scheme's dispersion properties with Lax-Wendroff. Does leapfrog preserve amplitude better?

Solution

```
from devito import Grid, TimeFunction, Eq, Operator, Constant
import numpy as np
import matplotlib.pyplot as plt

def solve_advection_leapfrog(L, c, Nx, T, C, I):
    """Leapfrog scheme with upwind initialization."""
    dx = L / Nx
    dt = C * dx / c

    grid = Grid(shape=(Nx + 1,), extent=(L,))
    x_dim, = grid.dimensions
    t_dim = grid.stepping_dim

    # time_order=2 gives access to u, u.forward, u.backward
    u = TimeFunction(name='u', grid=grid, time_order=2, space_order=1)

    x_coords = np.linspace(0, L, Nx + 1)

    # Set u^0
    u.data[0, :] = I(x_coords)

    # First step: use upwind to get u^1
    courant = Constant(name='C', value=C)
    u_minus = u.subs(x_dim, x_dim - x_dim.spacing)
    upwind_stencil = u - courant * (u - u_minus)

    # For leapfrog: u^{n+1} = u^{n-1} - C*(u^n_{i+1} - u^n_{i-1})
    u_plus_x = u.subs(x_dim, x_dim + x_dim.spacing)
    u_minus_x = u.subs(x_dim, x_dim - x_dim.spacing)
    leapfrog_stencil = u.backward - courant * (u_plus_x - u_minus_x)

    # Periodic BCs
    bc_left = Eq(u[t_dim + 1, 0], u[t_dim, Nx])
    bc_right = Eq(u[t_dim + 1, Nx], u[t_dim + 1, 0])

    # First step with upwind
    update_first = Eq(u.forward, upwind_stencil)
    op_first = Operator([update_first, bc_left, bc_right])
    op_first.apply(time_m=0, time_M=0, dt=dt)

    # Leapfrog for remaining steps
    update_lf = Eq(u.forward, leapfrog_stencil)
    op_lf = Operator([update_lf, bc_left, bc_right])

    Nt = int(round(T / dt))
    for n in range(1, Nt):
        op_lf.apply(time_m=n, time_M=n, dt=dt)

    return u.data[Nt % 3, :].copy(), x_coords

def I(x):
    return np.exp(-0.5*((x - 0.25)/0.05)**2)
```

5. Nonlinear Problems

5.1. Linear versus nonlinear equations

5.1.1. Algebraic equations

A linear, scalar, algebraic equation in x has the form

$$ax + b = 0,$$

for arbitrary real constants a and b. The unknown is a number x. All other algebraic equations, e.g., $x^2 + ax + b = 0$, are nonlinear. The typical feature in a nonlinear algebraic equation is that the unknown appears in products with itself, like x^2 or $e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \cdots$.

We know how to solve a linear algebraic equation, $x = -b/a$, but there are no general methods for finding the exact solutions of nonlinear algebraic equations, except for very special cases (quadratic equations constitute a primary example). A nonlinear algebraic equation may have no solution, one solution, or many solutions. The tools for solving nonlinear algebraic equations are *iterative methods*, where we construct a series of linear equations, which we know how to solve, and hope that the solutions of the linear equations converge to a solution of the nonlinear equation we want to solve. Typical methods for nonlinear algebraic equation equations are Newton's method, the Bisection method, and the Secant method.

5.1.2. Differential equations

The unknown in a differential equation is a function and not a number. In a linear differential equation, all terms involving the unknown function are linear in the unknown function or its derivatives. Linear here means that the unknown function, or a derivative of it, is multiplied by a number or a known function. All other differential equations are non-linear.

The easiest way to see if an equation is nonlinear, is to spot nonlinear terms where the unknown function or its derivatives are multiplied by each other. For example, in

$$u'(t) = -a(t)u(t) + b(t),$$

the terms involving the unknown function u are linear: u' contains the derivative of the unknown function multiplied by unity, and au contains the unknown function multiplied by a known function. However,

$$u'(t) = u(t)(1 - u(t)),$$

is nonlinear because of the term $-u^2$ where the unknown function is multiplied by itself. Also

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0,$$

is nonlinear because of the term uu_x where the unknown function appears in a product with its derivative. (Note here that we use different notations for derivatives: u' or du/dt for a function $u(t)$ of one variable, $\frac{\partial u}{\partial t}$ or u_t for a function of more than one variable.)

Another example of a nonlinear equation is

$$u'' + \sin(u) = 0,$$

because $\sin(u)$ contains products of u, which becomes clear if we expand the function in a Taylor series:

$$\sin(u) = u - \frac{1}{3}u^3 + \ldots$$

i Mathematical proof of linearity

To really prove mathematically that some differential equation in an unknown u is linear, show for each term $T(u)$ that with $u = au_1 + bu_2$ for constants a and b,

$$T(au_1 + bu_2) = aT(u_1) + bT(u_2)\,.$$

For example, the term $T(u) = (\sin^2 t)u'(t)$ is linear because

$$\begin{aligned} T(au_1 + bu_2) &= (\sin^2 t)(au_1(t) + bu_2(t)) \\ &= a(\sin^2 t)u_1(t) + b(\sin^2 t)u_2(t) \\ &= aT(u_1) + bT(u_2)\,. \end{aligned}$$

However, $T(u) = \sin u$ is nonlinear because

$$T(au_1 + bu_2) = \sin(au_1 + bu_2) \neq a\sin u_1 + b\sin u_2\,.$$

5.2. A simple model problem

A series of forthcoming examples will explain how to tackle nonlinear differential equations with various techniques. We start with the (scaled) logistic equation as model problem:

$$u'(t) = u(t)(1 - u(t))\,. \tag{5.1}$$

This is a nonlinear ordinary differential equation (ODE) which will be solved by different strategies in the following. Depending on the chosen time discretization of (5.1), the mathematical problem to be solved at every time level will either be a linear algebraic equation or a nonlinear algebraic equation. In the former case, the time discretization method transforms the nonlinear ODE into linear subproblems at each time level, and the solution is straightforward to find since linear algebraic equations are easy to solve. However, when the time discretization leads to nonlinear algebraic equations, we cannot (except in very rare cases) solve these without turning to approximate, iterative solution methods.

The next subsections introduce various methods for solving nonlinear differential equations, using (5.1) as model. We shall go through the following set of cases:

- explicit time discretization methods (with no need to solve nonlinear algebraic equations)
- implicit Backward Euler time discretization, leading to nonlinear algebraic equations solved by
- an exact analytical technique
- Picard iteration based on manual linearization
- a single Picard step
- Newton's method
- implicit Crank-Nicolson time discretization and linearization via a geometric mean formula

Thereafter, we compare the performance of the various approaches. Despite the simplicity of (5.1), the conclusions reveal typical features of the various methods in much more complicated nonlinear PDE problems.

5.3. Linearization by explicit time discretization

Time discretization methods are divided into explicit and implicit methods. Explicit methods lead to a closed-form formula for finding new values of the unknowns, while implicit methods give a linear or nonlinear system of equations that couples (all) the unknowns at a new time level. Here we shall demonstrate that explicit methods constitute an efficient way to deal with nonlinear differential equations.

The Forward Euler method is an explicit method. When applied to (5.1), sampled at $t = t_n$, it results in

$$\frac{u^{n+1} - u^n}{\Delta t} = u^n(1 - u^n),$$

which is a *linear* algebraic equation for the unknown value u^{n+1} that we can easily solve:

$$u^{n+1} = u^n + \Delta t\, u^n(1 - u^n)\,.$$

In this case, the nonlinearity in the original equation poses no difficulty in the discrete algebraic equation. Any other explicit scheme in time will also give only linear algebraic equations to solve. For example, a typical 2nd-order Runge-Kutta method for (5.1) leads to the following formulas:

$$u^* = u^n + \Delta t u^n(1 - u^n),$$
$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(u^n(1 - u^n) + u^*(1 - u^*)\right)\,.$$

The first step is linear in the unknown u^*. Then u^* is known in the next step, which is linear in the unknown u^{n+1} .

5.4. Exact solution of nonlinear algebraic equations

Switching to a Backward Euler scheme for (5.1),

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n(1 - u^n), \tag{5.2}$$

results in a nonlinear algebraic equation for the unknown value u^n. The equation is of quadratic type:

$$\Delta t(u^n)^2 + (1-\Delta t)u^n - u^{n-1} = 0,$$

and may be solved exactly by the well-known formula for such equations. Before we do so, however, we will introduce a shorter, and often cleaner, notation for nonlinear algebraic equations at a given time level. The notation is inspired by the natural notation (i.e., variable names) used in a program, especially in more advanced partial differential equation problems. The unknown in the algebraic equation is denoted by u, while $u^{(1)}$ is the value of the unknown at the previous time level (in general, $u^{(\ell)}$ is the value of the unknown ℓ levels back in time). The notation will be frequently used in later sections. What is meant by u should be evident from the context: u may either be 1) the exact solution of the ODE/PDE problem, 2) the numerical approximation to the exact solution, or 3) the unknown solution at a certain time level.

The quadratic equation for the unknown u^n in (5.2) can, with the new notation, be written

$$F(u) = \Delta t u^2 + (1-\Delta t)u - u^{(1)} = 0\,. \tag{5.3}$$

The solution is readily found to be

$$u = \frac{1}{2\Delta t}\left(-1 + \Delta t \pm \sqrt{(1-\Delta t)^2 - 4\Delta t u^{(1)}}\right)\,. \tag{5.4}$$

Now we encounter a fundamental challenge with nonlinear algebraic equations: the equation may have more than one solution. How do we pick the right solution? This is in general a hard problem. In the present simple case, however, we can analyze the roots mathematically and provide an answer. The idea is to expand the roots in a series in Δt and truncate after the linear term since the Backward Euler scheme will introduce an error proportional to Δt anyway. Using `sympy`, we find the following Taylor series expansions of the roots:

```
>>> import sympy as sym
>>> dt, u_1, u = sym.symbols('dt u_1 u')
>>> r1, r2 = sym.solve(dt*u**2 + (1-dt)*u - u_1, u)  # find roots
>>> r1
(dt - sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> r2
(dt + sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> print r1.series(dt, 0, 2)    # 2 terms in dt, around dt=0
-1/dt + 1 - u_1 + dt*(u_1**2 - u_1) + O(dt**2)
>>> print r2.series(dt, 0, 2)
u_1 + dt*(-u_1**2 + u_1) + O(dt**2)
```

We see that the `r1` root, corresponding to a minus sign in front of the square root in (5.4), behaves as $1/\Delta t$ and will therefore blow up as $\Delta t \to 0$! Since we know that u takes on finite values, actually it is less than or equal to 1, only the `r2` root is of relevance in this case: as $\Delta t \to 0$, $u \to u^{(1)}$, which is the expected result.

For those who are not well experienced with approximating mathematical formulas by series expansion, an alternative method of investigation is simply to compute the limits of the two roots as $\Delta t \to 0$ and see if a limit appears unreasonable:

```
>>> print r1.limit(dt, 0)
-oo
>>> print r2.limit(dt, 0)
u_1
```

5.5. Linearization

When the time integration of an ODE results in a nonlinear algebraic equation, we must normally find its solution by defining a sequence of linear equations and hope that the solutions of these linear equations converge to the desired solution of the nonlinear algebraic equation. Usually, this means solving the linear equation repeatedly in an iterative fashion. Alternatively, the nonlinear equation can sometimes be approximated by one linear equation, and consequently there is no need for iteration.

Constructing a linear equation from a nonlinear one requires *linearization* of each nonlinear term. This can be done manually as in Picard iteration, or fully algorithmically as in Newton's method. Examples will best illustrate how to linearize nonlinear problems.

5.6. Picard iteration

Let us write (5.3) in a more compact form

$$F(u) = au^2 + bu + c = 0,$$

with $a = \Delta t$, $b = 1 - \Delta t$, and $c = -u^{(1)}$. Let u^- be an available approximation of the unknown u. Then we can linearize the term u^2 simply by writing $u^- u$. The resulting equation, $\hat{F}(u) = 0$, is now linear and hence easy to solve:

$$F(u) \approx \hat{F}(u) = au^- u + bu + c = 0\,.$$

Since the equation $\hat{F} = 0$ is only approximate, the solution u does not equal the exact solution u_e of the exact equation $F(u_\mathrm{e}) = 0$, but we can hope that u is closer to u_e than u^- is, and hence it makes sense to repeat the procedure, i.e., set $u^- = u$ and solve $\hat{F}(u) = 0$ again. There is no guarantee that u is closer to u_e than u^-, but this approach has proven to be effective in a wide range of applications.

The idea of turning a nonlinear equation into a linear one by using an approximation u^- of u in nonlinear terms is a widely used approach that goes under many names: *fixed-point iteration*, the method of *successive substitutions*, *nonlinear Richardson iteration*, and *Picard iteration*. We will stick to the latter name.

Picard iteration for solving the nonlinear equation arising from the Backward Euler discretization of the logistic equation can be written as

$$u = -\frac{c}{au^- + b}, \quad u^- \leftarrow u\,.$$

The $\leftarrow$ symbols means assignment (we set u^- equal to the value of u). The iteration is started with the value of the unknown at the previous time level: $u^- = u^{(1)}$.

Some prefer an explicit iteration counter as superscript in the mathematical notation. Let u^k be the computed approximation to the solution in iteration k. In iteration $k+1$ we want to solve

$$au^k u^{k+1} + bu^{k+1} + c = 0 \quad \Rightarrow \quad u^{k+1} = -\frac{c}{au^k + b}, \quad k = 0, 1, \ldots$$

Since we need to perform the iteration at every time level, the time level counter is often also included:

$$au^{n,k}u^{n,k+1} + bu^{n,k+1} - u^{n-1} = 0 \quad \Rightarrow \quad u^{n,k+1} = \frac{u^n}{au^{n,k} + b}, \quad k = 0, 1, \ldots,$$

with the start value $u^{n,0} = u^{n-1}$ and the final converged value $u^n = u^{n,k}$ for sufficiently large k.

However, we will normally apply a mathematical notation in our final formulas that is as close as possible to what we aim to write in a computer code and then it becomes natural to use u and u^- instead of u^{k+1} and u^k or $u^{n,k+1}$ and $u^{n,k}$.

5.6.1. Stopping criteria

The iteration method can typically be terminated when the change in the solution is smaller than a tolerance ϵ_u:

$$|u - u^-| \leq \epsilon_u,$$

or when the residual in the equation is sufficiently small ($< \epsilon_r$),

$$|F(u)| = |au^2 + bu + c| < \epsilon_r\,.$$

A single Picard iteration Instead of iterating until a stopping criterion is fulfilled, one may iterate a specific number of times. Just one Picard iteration is popular as this corresponds to the intuitive idea of approximating a nonlinear term like $(u^n)^2$ by $u^{n-1}u^n$. This follows from the linearization u^-u^n and the initial choice of $u^- = u^{n-1}$ at time level t_n. In other words, a single Picard iteration corresponds to using the solution at the previous time level to linearize nonlinear terms. The resulting discretization becomes (using proper values for a, b, and c)

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n(1 - u^{n-1}), \tag{5.5}$$

which is a linear algebraic equation in the unknown u^n, making it easy to solve for u^n without any need for an alternative notation.

We shall later refer to the strategy of taking one Picard step, or equivalently, linearizing terms with use of the solution at the previous time step, as the *Picard1* method. It is a widely used approach in science and technology, but with some limitations if Δt is not sufficiently small (as will be illustrated later).

Equation (5.5) does not

correspond to a "pure" finite difference method where the equation is sampled at a point and derivatives replaced by differences (because the u^{n-1} term on the right-hand side must then be u^n). The best interpretation of the scheme (5.5) is a Backward Euler difference combined with a single (perhaps insufficient) Picard iteration at each time level, with the value at the previous time level as start for the Picard iteration.

5.7. Linearization by a geometric mean

We consider now a Crank-Nicolson discretization of (5.1). This means that the time derivative is approximated by a centered difference,

$$[D_t u = u(1-u)]^{n+\frac{1}{2}},$$

written out as

$$\frac{u^{n+1}-u^n}{\Delta t} = u^{n+\frac{1}{2}} - (u^{n+\frac{1}{2}})^2 . \tag{5.6}$$

The term $u^{n+\frac{1}{2}}$ is normally approximated by an arithmetic mean,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}),$$

such that the scheme involves the unknown function only at the time levels where we actually intend to compute it. The same arithmetic mean applied to the nonlinear term gives

$$(u^{n+\frac{1}{2}})^2 \approx \frac{1}{4}(u^n + u^{n+1})^2,$$

which is nonlinear in the unknown u^{n+1}. However, using a *geometric mean* for $(u^{n+\frac{1}{2}})^2$ is a way of linearizing the nonlinear term in (5.6):

$$(u^{n+\frac{1}{2}})^2 \approx u^n u^{n+1} .$$

Using an arithmetic mean on the linear $u^{n+\frac{1}{2}}$ term in (5.6) and a geometric mean for the second term, results in a linearized equation for the unknown u^{n+1}:

$$\frac{u^{n+1}-u^n}{\Delta t} = \frac{1}{2}(u^n + u^{n+1}) + u^n u^{n+1},$$

which can readily be solved:

$$u^{n+1} = \frac{1+\frac{1}{2}\Delta t}{1+\Delta t u^n - \frac{1}{2}\Delta t} u^n .$$

This scheme can be coded directly, and since there is no nonlinear algebraic equation to iterate over, we skip the simplified notation with u for u^{n+1} and $u^{(1)}$ for u^n. The technique with using a geometric average is an example of transforming a nonlinear algebraic equation to a linear one, without any need for iterations.

The geometric mean approximation is often very effective for linearizing quadratic nonlinearities. Both the arithmetic and geometric mean approximations have truncation errors of order Δt^2 and are therefore compatible with the truncation error $\mathcal{O}(\Delta t^2)$ of the centered difference approximation for u' in the Crank-Nicolson method.

Applying the operator notation for the means and finite differences, the linearized Crank-Nicolson scheme for the logistic equation can be compactly expressed as

$$[D_t u = \overline{u}^t + \overline{u^2}^{t,g}]^{n+\frac{1}{2}} .$$

Remark

If we use an arithmetic instead of a geometric mean for the nonlinear term in (5.6), we end up with a nonlinear term $(u^{n+1})^2$. This term can be linearized as $u^- u^{n+1}$ in a Picard iteration approach and in particular as $u^n u^{n+1}$ in a Picard1 iteration approach. The latter gives a scheme almost identical to the one arising from a geometric mean (the difference in u^{n+1} being $\frac{1}{4}\Delta t u^n(u^{n+1} - u^n) \approx \frac{1}{4}\Delta t^2 u' u$, i.e., a difference of size Δt^2).

5.8. Newton's method

The Backward Euler scheme (5.2) for the logistic equation leads to a nonlinear algebraic equation (5.3). Now we write any nonlinear algebraic equation in the general and compact form

$$F(u) = 0 .$$

Newton's method linearizes this equation by approximating $F(u)$ by its Taylor series expansion around a computed value u^- and keeping only the linear part:

$$\begin{aligned} F(u) &= F(u^-) + F'(u^-)(u - u^-) + \frac{1}{2}F''(u^-)(u - u^-)^2 + \cdots \\ &\approx F(u^-) + F'(u^-)(u - u^-) = \hat{F}(u) . \end{aligned}$$

The linear equation $\hat{F}(u) = 0$ has the solution

$$u = u^- - \frac{F(u^-)}{F'(u^-)} .$$

Expressed with an iteration index in the unknown, Newton's method takes on the more familiar mathematical form

$$u^{k+1} = u^k - \frac{F(u^k)}{F'(u^k)}, \quad k = 0, 1, \ldots$$

It can be shown that the error in iteration $k+1$ of Newton's method is proportional to the square of the error in iteration k, a result referred to as *quadratic convergence*. This means that for small errors the method converges very fast, and in particular much faster than Picard iteration and other iteration methods. (The proof of this result is found in most textbooks on numerical analysis.) However, the quadratic convergence appears only if u^k is sufficiently close to the solution.

Further away from the solution the method can easily converge very slowly or diverge. The reader is encouraged to do Exercise Section 5.38 to get a better understanding for the behavior of the method.

Application of Newton's method to the logistic equation discretized by the Backward Euler method is straightforward as we have

$$F(u) = au^2 + bu + c, \quad a = \Delta t,\ b = 1 - \Delta t,\ c = -u^{(1)},$$

and then

$$F'(u) = 2au + b\,.$$

The iteration method becomes

$$u = u^- + \frac{a(u^-)^2 + bu^- + c}{2au^- + b}, \quad u^- \leftarrow u\,. \tag{5.7}$$

At each time level, we start the iteration by setting $u^- = u^{(1)}$. Stopping criteria as listed for the Picard iteration can be used also for Newton's method.

An alternative mathematical form, where we write out a, b, and c, and use a time level counter n and an iteration counter k, takes the form

$$u^{n,k+1} = u^{n,k} + \frac{\Delta t(u^{n,k})^2 + (1 - \Delta t)u^{n,k} - u^{n-1}}{2\Delta t u^{n,k} + 1 - \Delta t}, \quad u^{n,0} = u^{n-1}, \tag{5.8}$$

for $k = 0, 1, \ldots$. A program implementation is much closer to (5.7) than to (5.8), but the latter is better aligned with the established mathematical notation used in the literature.

5.9. Relaxation

One iteration in Newton's method or Picard iteration consists of solving a linear problem $\hat{F}(u) = 0$. Sometimes convergence problems arise because the new solution u of $\hat{F}(u) = 0$ is "too far away" from the previously computed solution u^-. A remedy is to introduce a relaxation, meaning that we first solve $\hat{F}(u^*) = 0$ for a suggested value u^* and then we take u as a weighted mean of what we had, u^-, and what our linearized equation $\hat{F} = 0$ suggests, u^*:

$$u = \omega u^* + (1 - \omega)u^-\,.$$

The parameter ω is known as a *relaxation parameter*, and a choice $\omega < 1$ may prevent divergent iterations.

Relaxation in Newton's method can be directly incorporated in the basic iteration formula:

$$u = u^- - \omega\frac{F(u^-)}{F'(u^-)}\,. \tag{5.9}$$

5.10. Implementation and experiments

The program logistic.py contains implementations of all the methods described above. Below is an extract of the file showing how the Picard and Newton methods are implemented for a Backward Euler discretization of the logistic equation.

def BE_logistic(u0, dt, Nt, choice="Picard", eps_r=1e-3, omega=1, max_iter=1000): if choice == "Picard1": choice = "Picard" max_iter = 1

```
u = np.zeros(Nt + 1)
iterations = []
u[0] = u0
for n in range(1, Nt + 1):
    a = dt
    b = 1 - dt
    c = -u[n - 1]
    if choice in ("r1", "r2"):
        r1, r2 = quadratic_roots(a, b, c)
        u[n] = r1 if choice == "r1" else r2
        iterations.append(0)

    elif choice == "Picard":

        def F(u):
            return a * u**2 + b * u + c

        u_ = u[n - 1]
        k = 0
        while abs(F(u_)) > eps_r and k < max_iter:
            u_ = omega * (-c / (a * u_ + b)) + (1 - omega) * u_
            k += 1
        u[n] = u_
        iterations.append(k)

    elif choice == "Newton":

        def F(u):
            return a * u**2 + b * u + c

        def dF(u):
            return 2 * a * u + b

        u_ = u[n - 1]
        k = 0
        while abs(F(u_)) > eps_r and k < max_iter:
            u_ = u_ - F(u_) / dF(u_)
            k += 1
        u[n] = u_
```

```
        iterations.append(k)
return u, iterations
```

```python
def BE_logistic(u0, dt, Nt, choice='Picard',
                eps_r=1E-3, omega=1, max_iter=1000):
    if choice == 'Picard1':
        choice = 'Picard'
        max_iter = 1

    u = np.zeros(Nt+1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt+1):
        a = dt
        b = 1 - dt
        c = -u[n-1]

        if choice == 'Picard':

            def F(u):
                return a*u**2 + b*u + c

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = omega*(-c/(a*u_ + b)) + (1-omega)*u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == 'Newton':

            def F(u):
                return a*u**2 + b*u + c

            def dF(u):
                return 2*a*u + b

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_)/dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)
    return u, iterations
```

The Crank-Nicolson method utilizing a linearization based on the geometric mean gives a simpler algorithm:

```
def CN_logistic(u0, dt, Nt):
    u = np.zeros(Nt + 1)
    u[0] = u0
    for n in range(0, Nt):
        u[n + 1] = (1 + 0.5 * dt) / (1 + dt * u[n] - 0.5 * dt) * u[n]
    return u
```

We may run experiments with the model problem (5.1) and the different strategies for dealing with nonlinearities as described above. For a quite coarse time resolution, $\Delta t = 0.9$, use of a tolerance $\epsilon_r = 0.1$ in the stopping criterion introduces an iteration error, especially in the Picard iterations, that is visibly much larger than the time discretization error due to a large Δt. This is illustrated by comparing the upper two plots in Figure Figure 5.1. The one to the right has a stricter tolerance $\epsilon = 10^{-3}$, which causes all the curves corresponding to Picard and Newton iteration to be on top of each other (and no changes can be visually observed by reducing ϵ_r further). The reason why Newton's method does much better than Picard iteration in the upper left plot is that Newton's method with one step comes far below the ϵ_r tolerance, while the Picard iteration needs on average 7 iterations to bring the residual down to $\epsilon_r = 10^{-1}$, which gives insufficient accuracy in the solution of the nonlinear equation. It is obvious that the Picard1 method gives significant errors in addition to the time discretization unless the time step is as small as in the lower right plot.

The *BE exact* curve corresponds to using the exact solution of the quadratic equation at each time level, so this curve is only affected by the Backward Euler time discretization. The *CN gm* curve corresponds to the theoretically more accurate Crank-Nicolson discretization, combined with a geometric mean for linearization. This curve appears more accurate, especially if we take the plot in the lower right with a small Δt and an appropriately small ϵ_r value as the exact curve.

When it comes to the need for iterations, Figure Figure 5.2 displays the number of iterations required at each time level for Newton's method and Picard iteration. The smaller Δt is, the better starting value we have for the iteration, and the faster the convergence is. With $\Delta t = 0.9$ Picard iteration requires on average 32 iterations per time step, but this number is dramatically reduced as Δt is reduced.

However, introducing relaxation and a parameter $\omega = 0.8$ immediately reduces the average of 32 to 7, indicating that for the large $\Delta t = 0.9$, Picard iteration takes too long steps. An approximately optimal value for ω in this case is 0.5, which results in an average of only 2 iterations! An even more dramatic impact of ω appears when $\Delta t = 1$: Picard iteration does not convergence in 1000 iterations, but $\omega = 0.5$ again brings the average number of iterations down to 2.

Remark. The simple Crank-Nicolson method with a geometric mean for the quadratic nonlinearity gives visually more accurate solutions than the Backward Euler discretization. Even with a tolerance of $\epsilon_r = 10^{-3}$, all the methods for treating the nonlinearities in the Backward Euler discretization give graphs that cannot be distinguished. So for accuracy in this problem, the time discretization is much more crucial than ϵ_r. Ideally, one should estimate the error in the time discretization, as the solution progresses, and set ϵ_r accordingly.

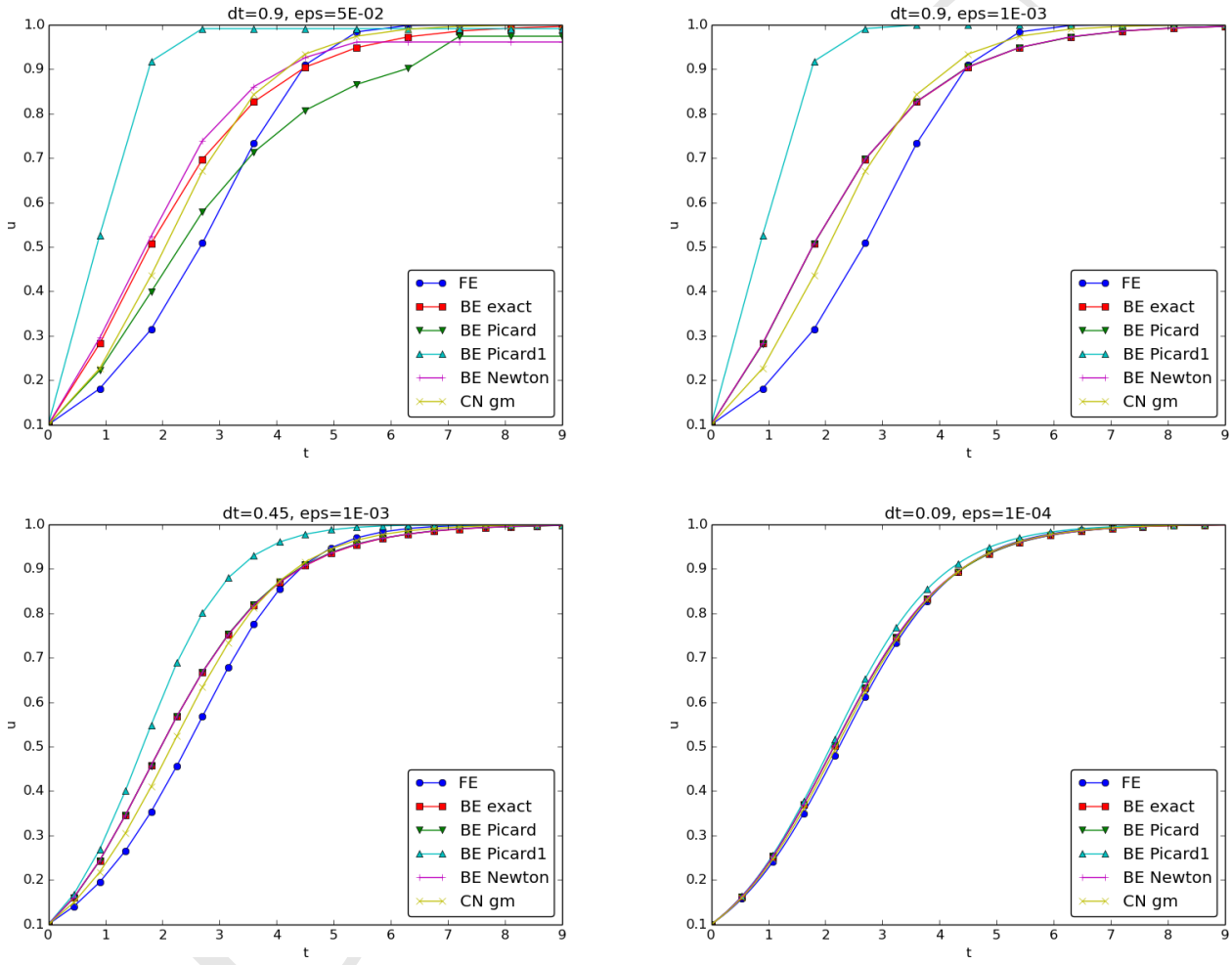


Figure 5.1.: Impact of solution strategy and time step length on the solution.

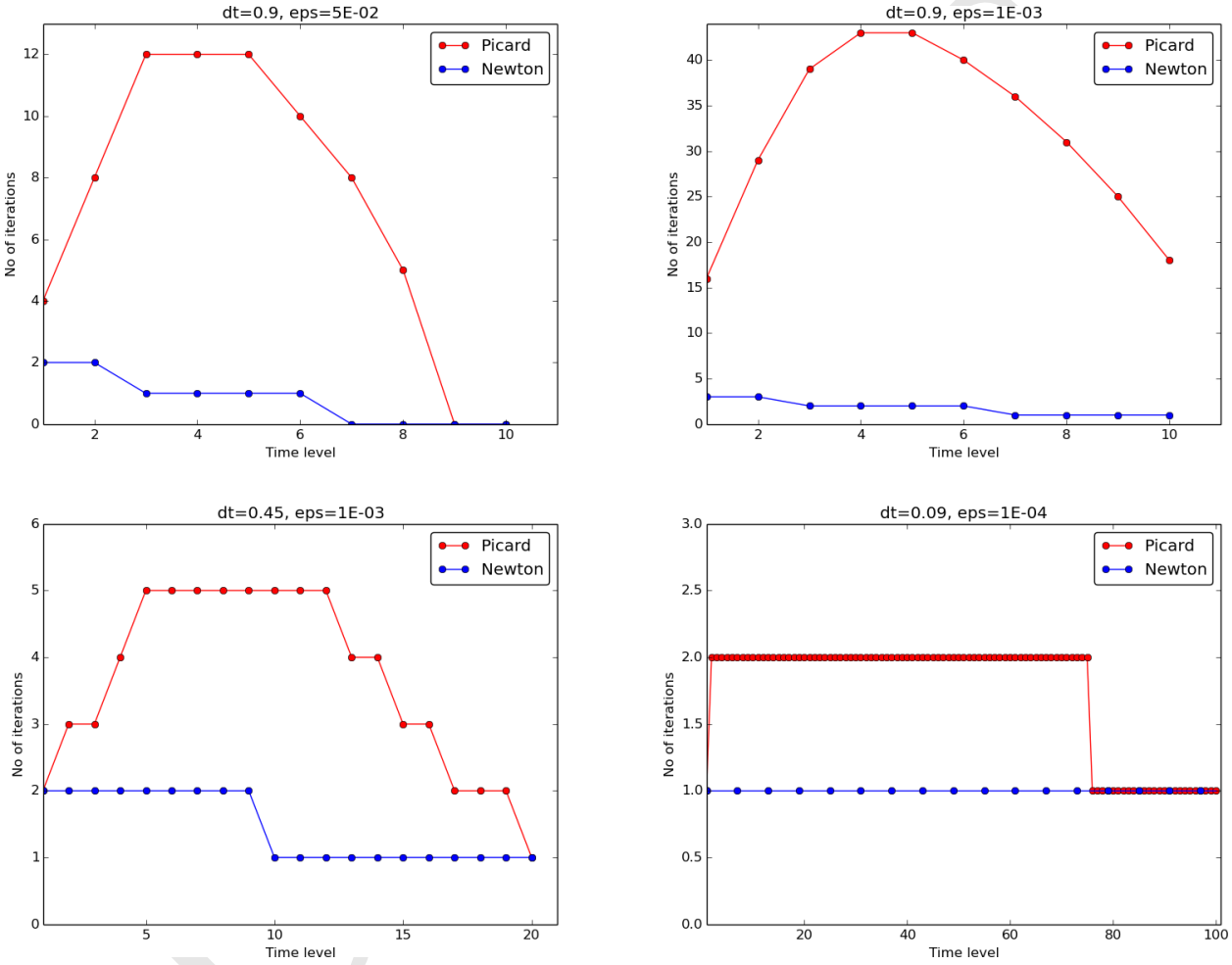


Figure 5.2.: Comparison of the number of iterations at various time levels for Picard and Newton iteration.

5.11. Generalization to a general nonlinear ODE

Let us see how the various methods in the previous sections can be applied to the more generic model

$$u' = f(u, t), \tag{5.10}$$

where f is a nonlinear function of u.

5.11.1. Explicit time discretization

Explicit ODE methods like the Forward Euler scheme, Runge-Kutta methods and Adams-Bashforth methods all evaluate f at time levels where u is already computed, so nonlinearities in f do not pose any difficulties.

5.11.2. Backward Euler discretization

Approximating u' by a backward difference leads to a Backward Euler scheme, which can be written as

$$F(u^n) = u^n - \Delta t\, f(u^n, t_n) - u^{n-1} = 0,$$

or alternatively

$$F(u) = u - \Delta t\, f(u, t_n) - u^{(1)} = 0\,.$$

A simple Picard iteration, not knowing anything about the nonlinear structure of f, must approximate $f(u, t_n)$ by $f(u^-, t_n)$:

$$\hat{F}(u) = u - \Delta t\, f(u^-, t_n) - u^{(1)}\,.$$

The iteration starts with $u^- = u^{(1)}$ and proceeds with repeating

$$u^* = \Delta t\, f(u^-, t_n) + u^{(1)}, \quad u = \omega u^* + (1-\omega)u^-, \quad u^- \leftarrow u,$$

until a stopping criterion is fulfilled.

Explicit vs implicit treatment of nonlinear terms

Evaluating f for a known u^- is referred to as *explicit* treatment of f, while if $f(u, t)$ has some structure, say $f(u, t) = u^3$, parts of f can involve the unknown u, as in the manual linearization $(u^-)^2 u$, and then the treatment of f is "more implicit" and "less explicit". This terminology is inspired by time discretization of $u' = f(u, t)$, where evaluating f for known u values gives explicit schemes, while treating f or parts of f implicitly, makes f contribute to the unknown terms in the equation at the new time level.

Explicit treatment of f usually means stricter conditions on Δt to achieve stability of time discretization schemes. The same applies to iteration techniques for nonlinear algebraic equations: the "less" we linearize f (i.e., the more we keep of u in the original formula), the faster the convergence may be.

We may say that $f(u, t) = u^3$ is treated explicitly if we evaluate f as $(u^-)^3$, partially implicit if we linearize as $(u^-)^2 u$ and fully implicit if we represent f by u^3. (Of course, the fully implicit representation will require further linearization, but with $f(u, t) = u^2$ a fully implicit treatment is possible if the resulting quadratic equation is solved with a formula.)

For the ODE $u' = -u^3$ with $f(u,t) = -u^3$ and coarse time resolution $\Delta t = 0.4$, Picard iteration with $(u^-)^2 u$ requires 8 iterations with $\epsilon_r = 10^{-3}$ for the first time step, while $(u^-)^3$ leads to 22 iterations. After about 10 time steps both approaches are down to about 2 iterations per time step, but this example shows a potential of treating f more implicitly.
A trick to treat f implicitly in Picard iteration is to evaluate it as $f(u^-,t)u/u^-$. For a polynomial f, $f(u,t) = u^m$, this corresponds to $(u^-)^m u/u^- = (u^-)^{m-1}u$. Sometimes this more implicit treatment has no effect, as with $f(u,t) = \exp(-u)$ and $f(u,t) = \ln(1+u)$, but with $f(u,t) = \sin(2(u+1))$, the $f(u^-,t)u/u^-$ trick leads to 7, 9, and 11 iterations during the first three steps, while $f(u^-,t)$ demands 17, 21, and 20 iterations. (Experiments can be done with the code ODE_Picard_tricks.py.)

Newton's method applied to a Backward Euler discretization of $u' = f(u,t)$ requires computation of the derivative

$$F'(u) = 1 - \Delta t \frac{\partial f}{\partial u}(u, t_n)\,.$$

Starting with the solution at the previous time level, $u^- = u^{(1)}$, we can just use the standard formula

$$u = u^- - \omega \frac{F(u^-)}{F'(u^-)} = u^- - \omega \frac{u^- - \Delta t\, f(u^-, t_n) - u^{(1)}}{1 - \Delta t \frac{\partial}{\partial u} f(u^-, t_n)}\,. \tag{5.11}$$

5.11.3. Crank-Nicolson discretization

The standard Crank-Nicolson scheme with arithmetic mean approximation of f takes the form

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(f(u^{n+1}, t_{n+1}) + f(u^n, t_n))\,.$$

We can write the scheme as a nonlinear algebraic equation

$$F(u) = u - u^{(1)} - \Delta t \frac{1}{2} f(u, t_{n+1}) - \Delta t \frac{1}{2} f(u^{(1)}, t_n) = 0\,. \tag{5.12}$$

A Picard iteration scheme must in general employ the linearization

$$\hat{F}(u) = u - u^{(1)} - \Delta t \frac{1}{2} f(u^-, t_{n+1}) - \Delta t \frac{1}{2} f(u^{(1)}, t_n),$$

while Newton's method can apply the general formula (5.11) with $F(u)$ given in (5.12) and

$$F'(u) = 1 - \frac{1}{2}\Delta t \frac{\partial f}{\partial u}(u, t_{n+1})\,.$$

5.12. Systems of ODEs

We may write a system of ODEs

$$\begin{aligned}
\frac{d}{dt}u_0(t) &= f_0(u_0(t), u_1(t), \ldots, u_N(t), t),\\
\frac{d}{dt}u_1(t) &= f_1(u_0(t), u_1(t), \ldots, u_N(t), t),\\
&\vdots\\
\frac{d}{dt}u_m(t) &= f_m(u_0(t), u_1(t), \ldots, u_N(t), t),
\end{aligned}$$

as

$$u' = f(u,t), \quad u(0) = U_0,$$

if we interpret u as a vector $u = (u_0(t), u_1(t), \ldots, u_N(t))$ and f as a vector function with components $(f_0(u,t), f_1(u,t), \ldots, f_N(u,t))$.

Most solution methods for scalar ODEs, including the Forward and Backward Euler schemes and the Crank-Nicolson method, generalize in a straightforward way to systems of ODEs simply by using vector arithmetics instead of scalar arithmetics, which corresponds to applying the scalar scheme to each component of the system. For example, here is a backward difference scheme applied to each component,

$$\begin{aligned}
\frac{u_0^n - u_0^{n-1}}{\Delta t} &= f_0(u^n, t_n),\\
\frac{u_1^n - u_1^{n-1}}{\Delta t} &= f_1(u^n, t_n),\\
&\vdots\\
\frac{u_N^n - u_N^{n-1}}{\Delta t} &= f_N(u^n, t_n),
\end{aligned}$$

which can be written more compactly in vector form as

$$\frac{u^n - u^{n-1}}{\Delta t} = f(u^n, t_n)\,.$$

This is a *system of algebraic equations*,

$$u^n - \Delta t\, f(u^n, t_n) - u^{n-1} = 0,$$

or written out

$$\begin{aligned}
u_0^n - \Delta t\, f_0(u^n, t_n) - u_0^{n-1} &= 0,\\
&\vdots\\
u_N^n - \Delta t\, f_N(u^n, t_n) - u_N^{n-1} &= 0\,.
\end{aligned}$$

5.12.1. Example

We shall address the 2×2 ODE system for oscillations of a pendulum subject to gravity and air drag. The system can be written as

$$\dot{\omega} = -\sin\theta - \beta\omega|\omega|, \tag{5.13}$$
$$\dot{\theta} = \omega, \tag{5.14}$$

where β is a dimensionless parameter (this is the scaled, dimensionless version of the original, physical model). The unknown components of the system are the angle $\theta(t)$ and the angular velocity $\omega(t)$. We introduce $u_0 = \omega$ and $u_1 = \theta$, which leads to

$$u_0' = f_0(u,t) = -\sin u_1 - \beta u_0|u_0|,$$
$$u_1' = f_1(u,t) = u_0\,.$$

A Crank-Nicolson scheme reads

$$\begin{aligned}\frac{u_0^{n+1}-u_0^n}{\Delta t} &= -\sin u_1^{n+\frac{1}{2}} - \beta u_0^{n+\frac{1}{2}}|u_0^{n+\frac{1}{2}}| \\ &\approx -\sin\left(\frac{1}{2}(u_1^{n+1}+u_1 n)\right) - \beta\frac{1}{4}(u_0^{n+1}+u_0^n)|u_0^{n+1}+u_0^n|, \end{aligned}\tag{5.15}$$
$$\frac{u_1^{n+1}-u_1^n}{\Delta t} = u_0^{n+\frac{1}{2}} \approx \frac{1}{2}(u_0^{n+1}+u_0^n)\,. \tag{5.16}$$

This is a *coupled system* of two nonlinear algebraic equations in two unknowns u_0^{n+1} and u_1^{n+1}.

Using the notation u_0 and u_1 for the unknowns u_0^{n+1} and u_1^{n+1} in this system, writing $u_0^{(1)}$ and $u_1^{(1)}$ for the previous values u_0^n and u_1^n, multiplying by Δt and moving the terms to the left-hand sides, gives

$$u_0 - u_0^{(1)} + \Delta t\sin\left(\frac{1}{2}(u_1+u_1^{(1)})\right) + \frac{1}{4}\Delta t\beta(u_0+u_0^{(1)})|u_0+u_0^{(1)}| = 0, \tag{5.17}$$
$$u_1 - u_1^{(1)} - \frac{1}{2}\Delta t(u_0+u_0^{(1)}) = 0\,. \tag{5.18}$$

Obviously, we have a need for solving systems of nonlinear algebraic equations, which is the topic of the next section.

5.13. Systems of nonlinear algebraic equations

Implicit time discretization methods for a system of ODEs, or a PDE, lead to *systems* of nonlinear algebraic equations, written compactly as

$$F(u) = 0,$$

where u is a vector of unknowns $u = (u_0, \ldots, u_N)$, and F is a vector function: $F = (F_0, \ldots, F_N)$. The system at the end of Section Section 5.12 fits this notation with $N = 1$, $F_0(u)$ given by the left-hand side of (5.17), while $F_1(u)$ is the left-hand side of (5.18).

Sometimes the equation system has a special structure because of the underlying problem, e.g.,

$$A(u)u = b(u),$$

with $A(u)$ as an $(N+1)\times(N+1)$ matrix function of u and b as a vector function: $b = (b_0, \ldots, b_N)$.

We shall next explain how Picard iteration and Newton's method can be applied to systems like $F(u) = 0$ and $A(u)u = b(u)$. The exposition has a focus on ideas and practical computations. More theoretical considerations, including quite general results on convergence properties of these methods, can be found in Kelley (Kelley 1995).

5.14. Picard iteration

We cannot apply Picard iteration to nonlinear equations unless there is some special structure. For the commonly arising case $A(u)u = b(u)$ we can linearize the product $A(u)u$ to $A(u^-)u$ and $b(u)$ as $b(u^-)$. That is, we use the most previously computed approximation in A and b to arrive at a *linear system* for u:

$$A(u^-)u = b(u^-)\,.$$

A relaxed iteration takes the form

$$A(u^-)u^* = b(u^-), \quad u = \omega u^* + (1-\omega)u^-\,.$$

In other words, we solve a system of nonlinear algebraic equations as a sequence of linear systems.

Algorithm for relaxed Picard iteration

Given $A(u)u = b(u)$ and an initial guess u^-, iterate until convergence:

1. solve $A(u^-)u^* = b(u^-)$ with respect to u^*
2. $u = \omega u^* + (1-\omega)u^-$
3. $u^- \leftarrow u$

"Until convergence" means that the iteration is stopped when the change in the unknown, $||u - u^-||$, or the residual $||A(u)u - b||$, is sufficiently small, see Section Section 5.16 for more details.

5.15. Newton's method

The natural starting point for Newton's method is the general nonlinear vector equation $F(u) = 0$. As for a scalar equation, the idea is to approximate F around a known value u^- by a linear function $\hat{F}$, calculated from the first two terms of a Taylor expansion of F. In the multi-variate case these two terms become

$$F(u^-) + J(u^-)\cdot(u - u^-),$$

where J is the *Jacobian* of F, defined by

$$J_{i,j} = \frac{\partial F_i}{\partial u_j}\,.$$

So, the original nonlinear system is approximated by

$$\hat{F}(u) = F(u^-) + J(u^-)\cdot(u - u^-) = 0,$$

which is linear in u and can be solved in a two-step procedure: first solve $J\delta u = -F(u^-)$ with respect to the vector δu and then update $u = u^- + \delta u$. A relaxation parameter can easily be incorporated:

$$u = \omega(u^- + \delta u) + (1-\omega)u^- = u^- + \omega\delta u\,.$$

Algorithm for Newton's method

Given $F(u) = 0$ and an initial guess u^-, iterate until convergence:

1. solve $J\delta u = -F(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

For the special system with structure $A(u)u = b(u)$,

$$F_i = \sum_k A_{i,k}(u)u_k - b_i(u),$$

one gets

$$J_{i,j} = \sum_k \frac{\partial A_{i,k}}{\partial u_j}u_k + A_{i,j} - \frac{\partial b_i}{\partial u_j}\,.$$

We realize that the Jacobian needed in Newton's method consists of $A(u^-)$ as in the Picard iteration plus two additional terms arising from the differentiation. Using the notation $A'(u)$ for $\partial A/\partial u$ (a quantity with three indices: $\partial A_{i,k}/\partial u_j$), and $b'(u)$ for $\partial b/\partial u$ (a quantity with two indices: $\partial b_i/\partial u_j$), we can write the linear system to be solved as

$$(A + A'u + b')\delta u = -Au + b,$$

or

$$(A(u^-) + A'(u^-)u^- + b'(u^-))\delta u = -A(u^-)u^- + b(u^-)\,.$$

Rearranging the terms demonstrates the difference from the system solved in each Picard iteration:

$$\underbrace{A(u^-)(u^- + \delta u) - b(u^-)}_{\text{Picard system}} + \gamma(A'(u^-)u^- + b'(u^-))\delta u = 0\,.$$

Here we have inserted a parameter γ such that $\gamma = 0$ gives the Picard system and $\gamma = 1$ gives the Newton system. Such a parameter can be handy in software to easily switch between the methods.

Combined algorithm for Picard and Newton iteration

Given $A(u)$, $b(u)$, and an initial guess u^-, iterate until convergence:

1. solve $(A + \gamma(A'(u^-)u^- + b'(u^-)))\delta u = -A(u^-)u^- + b(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

$\gamma = 1$ gives a Newton method while $\gamma = 0$ corresponds to Picard iteration.

5.16. Stopping criteria

Let $||\cdot||$ be the standard Euclidean vector norm. Four termination criteria are much in use:

- Absolute change in solution: $||u - u^-|| \leq \epsilon_u$
- Relative change in solution: $||u - u^-|| \leq \epsilon_u||u_0||$, where u_0 denotes the start value of u^- in the iteration
- Absolute residual: $||F(u)|| \leq \epsilon_r$
- Relative residual: $||F(u)|| \leq \epsilon_r||F(u_0)||$

To prevent divergent iterations to run forever, one terminates the iterations when the current number of iterations k exceeds a maximum value $k_{\max}$.

The relative criteria are most used since they are not sensitive to the characteristic size of u. Nevertheless, the relative criteria can be misleading when the initial start value for the iteration is very close to the solution, since an unnecessary reduction in the error measure is enforced. In such cases the absolute criteria work better. It is common to combine the absolute and relative measures of the size of the residual, as in

$$||F(u)|| \leq \epsilon_{rr}||F(u_0)|| + \epsilon_{ra},$$

where ϵ_{rr} is the tolerance in the relative criterion and ϵ_{ra} is the tolerance in the absolute criterion. With a very good initial guess for the iteration (typically the solution of a differential equation at the previous time level), the term $||F(u_0)||$ is small and ϵ_{ra} is the dominating tolerance. Otherwise, $\epsilon_{rr}||F(u_0)||$ and the relative criterion dominates.

With the change in solution as criterion we can formulate a combined absolute and relative measure of the change in the solution:

$$||\delta u|| \leq \epsilon_{ur}||u_0|| + \epsilon_{ua},$$

The ultimate termination criterion, combining the residual and the change in solution with a test on the maximum number of iterations, can be expressed as

$$||F(u)|| \leq \epsilon_{rr}||F(u_0)|| + \epsilon_{ra} \quad \text{or} \quad ||\delta u|| \leq \epsilon_{ur}||u_0|| + \epsilon_{ua} \quad \text{or} \quad k > k_{\max}\,.$$

Example: A nonlinear ODE model from epidemiology {#sec-nonlin-systems-alg-SI}

A very simple model for the spreading of a disease, such as a flu, takes the form of a 2×2 ODE system

$$S' = -\beta SI, \tag{5.19}$$
$$I' = \beta SI - \nu I, \tag{5.20}$$

where $S(t)$ is the number of people who can get ill (susceptibles) and $I(t)$ is the number of people who are ill (infected). The constants $\beta > 0$ and $\nu > 0$ must be given along with initial conditions $S(0)$ and $I(0)$.

5.16.1. Implicit time discretization

A Crank-Nicolson scheme leads to a 2×2 system of nonlinear algebraic equations in the unknowns S^{n+1} and I^{n+1}:

$$\frac{S^{n+1} - S^n}{\Delta t} = -\beta [SI]^{n+\frac{1}{2}} \approx -\frac{\beta}{2}(S^n I^n + S^{n+1} I^{n+1}), \tag{5.21}$$
$$\frac{I^{n+1} - I^n}{\Delta t} = \beta [SI]^{n+\frac{1}{2}} - \nu I^{n+\frac{1}{2}} \approx \frac{\beta}{2}(S^n I^n + S^{n+1} I^{n+1}) - \frac{\nu}{2}(I^n + I^{n+1})\,. \tag{5.22}$$

Introducing S for S^{n+1}, $S^{(1)}$ for S^n, I for I^{n+1} and $I^{(1)}$ for I^n, we can rewrite the system as

$$F_S(S, I) = S - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)} I^{(1)} + SI) = 0, \tag{5.23}$$
$$F_I(S, I) = I - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)} I^{(1)} + SI) + \frac{1}{2}\Delta t\nu(I^{(1)} + I) = 0\,. \tag{5.24}$$

5.16.2. A Picard iteration

We assume that we have approximations S^- and I^- to S and I, respectively. A way of linearizing the only nonlinear term SI is to write $I^- S$ in the $F_S = 0$ equation and $S^- I$ in the $F_I = 0$ equation, which also *decouples* the equations. Solving the resulting linear equations with respect to the unknowns S and I gives

$$S = \frac{S^{(1)} - \frac{1}{2}\Delta t\beta S^{(1)} I^{(1)}}{1 + \frac{1}{2}\Delta t\beta I^-},$$
$$I = \frac{I^{(1)} + \frac{1}{2}\Delta t\beta S^{(1)} I^{(1)} - \frac{1}{2}\Delta t\nu I^{(1)}}{1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu}\,.$$

Before a new iteration, we must update $S^- \leftarrow S$ and $I^- \leftarrow I$.

5.16.3. Newton's method

The nonlinear system (5.23)-(5.24) can be written as $F(u) = 0$ with $F = (F_S, F_I)$ and $u = (S, I)$. The Jacobian becomes

$$J = \begin{pmatrix} \frac{\partial}{\partial S}F_S & \frac{\partial}{\partial I}F_S \\ \frac{\partial}{\partial S}F_I & \frac{\partial}{\partial I}F_I \end{pmatrix} = \begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I & \frac{1}{2}\Delta t\beta S \\ +\frac{1}{2}\Delta t\beta I & 1 - \frac{1}{2}\Delta t\beta S + \frac{1}{2}\Delta t\nu \end{pmatrix}.$$

The Newton system $J(u^-)\delta u = -F(u^-)$ to be solved in each iteration is then

$$\begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I^- & \frac{1}{2}\Delta t\beta S^- \\ -\frac{1}{2}\Delta t\beta I^- & 1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu \end{pmatrix} \begin{pmatrix} \delta S \\ \delta I \end{pmatrix} = \begin{pmatrix} S^- - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) \\ I^- - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) + \frac{1}{2}\Delta t\nu(I^{(1)} + I^-) \end{pmatrix}$$

Remark. For this particular system of ODEs, explicit time integration methods work very well. Even a Forward Euler scheme is fine, but (as also experienced more generally) the 4-th order Runge-Kutta method is an excellent balance between high accuracy, high efficiency, and simplicity.

5.17. Nonlinear diffusion model

The attention is now turned to nonlinear partial differential equations (PDEs) and application of the techniques explained above for ODEs. The model problem is a nonlinear diffusion equation for $u(\boldsymbol{x}, t)$:

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(u)\nabla u) + f(u), \qquad \boldsymbol{x} \in \Omega,\ t \in (0, T], \tag{5.25}$$

$$-\alpha(u)\frac{\partial u}{\partial n} = g, \qquad \boldsymbol{x} \in \partial\Omega_N,\ t \in (0, T], \tag{5.26}$$

$$u = u_0, \qquad \boldsymbol{x} \in \partial\Omega_D,\ t \in (0, T]\,. \tag{5.27}$$

In the present section, our aim is to discretize this problem in time and then present techniques for linearizing the time-discrete PDE problem "at the PDE level" such that we transform the nonlinear stationary PDE problem at each time level into a sequence of linear PDE problems, which can be solved using any method for linear PDEs. This strategy avoids the solution of systems of nonlinear algebraic equations. In Section Section 5.22 we shall take the opposite (and more common) approach: discretize the nonlinear problem in time and space first, and then solve the resulting nonlinear algebraic equations at each time level by the methods of Section Section 5.13. Very often, the two approaches are mathematically identical, so there is no preference from a computational efficiency point of view. The details of the ideas sketched above will hopefully become clear through the forthcoming examples.

5.18. Explicit time integration

The nonlinearities in the PDE are trivial to deal with if we choose an explicit time integration method for the nonlinear diffusion equation, such as the Forward Euler method:

$$[D_t^+ u = \nabla \cdot (\alpha(u)\nabla u) + f(u)]^n,$$

or written out,

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla \cdot (\alpha(u^n)\nabla u^n) + f(u^n),$$

which is a linear equation in the unknown u^{n+1} with solution

$$u^{n+1} = u^n + \Delta t \nabla \cdot (\alpha(u^n)\nabla u^n) + \Delta t f(u^n)\,.$$

The disadvantage with this discretization is the strict stability criterion $\Delta t \leq h^2/(6\max\alpha)$ for the case $f = 0$ and a standard 2nd-order finite difference discretization in 3D space with mesh cell sizes $h = \Delta x = \Delta y = \Delta z$.

5.19. Backward Euler scheme and Picard iteration

A Backward Euler scheme for the nonlinear diffusion equation reads

$$[D_t^- u = \nabla \cdot (\alpha(u)\nabla u) + f(u)]^n\,.$$

Written out,

$$\frac{u^n - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^n)\nabla u^n) + f(u^n)\,. \tag{5.28}$$

This is a nonlinear PDE for the unknown function $u^n(\boldsymbol{x})$. Such a PDE can be viewed as a time-independent PDE where $u^{n-1}(\boldsymbol{x})$ is a known function.

We introduce a Picard iteration with k as iteration counter. A typical linearization of the $\nabla \cdot (\alpha(u^n)\nabla u^n)$ term in iteration $k+1$ is to use the previously computed $u^{n,k}$ approximation in the diffusion coefficient: $\alpha(u^{n,k})$. The nonlinear source term is treated similarly: $f(u^{n,k})$. The unknown function $u^{n,k+1}$ then fulfills the linear PDE

$$\frac{u^{n,k+1} - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^{n,k})\nabla u^{n,k+1}) + f(u^{n,k})\,. \tag{5.29}$$

The initial guess for the Picard iteration at this time level can be taken as the solution at the previous time level: $u^{n,0} = u^{n-1}$.

We can alternatively apply the implementation-friendly notation where u corresponds to the unknown we want to solve for, i.e., $u^{n,k+1}$ above, and u^- is the most recently computed value, $u^{n,k}$ above. Moreover, $u^{(1)}$ denotes the unknown function at the previous time level, u^{n-1} above. The PDE to be solved in a Picard iteration then looks like

$$\frac{u - u^{(1)}}{\Delta t} = \nabla \cdot (\alpha(u^-)\nabla u) + f(u^-)\,. \tag{5.30}$$

At the beginning of the iteration we start with the value from the previous time level: $u^- = u^{(1)}$, and after each iteration, u^- is updated to u.

Remark on notation

The previous derivations of the numerical scheme for time discretizations of PDEs have, strictly speaking, a somewhat sloppy notation, but it is much used and convenient to read. A more precise notation must distinguish clearly between the exact solution of the PDE problem, here denoted $u_{\mathrm{e}}(\boldsymbol{x}, t)$, and the exact solution of the spatial problem, arising after time discretization at each time level, where (5.28) is an example. The latter is here represented as $u^n(\boldsymbol{x})$ and is an approximation to $u_{\mathrm{e}}(\boldsymbol{x}, t_n)$. Then we have another approximation $u^{n,k}(\boldsymbol{x})$ to $u^n(\boldsymbol{x})$ when solving the nonlinear PDE problem for u^n by iteration methods, as in (5.29).
In our notation, u is a synonym for $u^{n,k+1}$ and $u^{(1)}$ is a synonym for u^{n-1}, inspired by what are natural variable names in a code. We will usually state the PDE problem in terms of u and quickly redefine the symbol u to mean the numerical approximation, while u_{e} is not explicitly introduced unless we need to talk about the exact solution and the approximate solution at the same time.

5.20. Backward Euler scheme and Newton's method

At time level n, we have to solve the stationary PDE (5.28). In the previous section, we saw how this can be done with Picard iterations. Another alternative is to apply the idea of Newton's method in a clever way. Normally, Newton's method is defined for systems of *algebraic equations*, but the idea of the method can be applied at the PDE level too.

5.20.1. Linearization via Taylor expansions

Let $u^{n,k}$ be an approximation to the unknown u^n. We seek a better approximation on the form

$$u^n = u^{n,k} + \delta u\,. \tag{5.31}$$

The idea is to insert (5.31) in (5.28), Taylor expand the nonlinearities and keep only the terms that are linear in δu (which makes (5.31) an approximation for u^n). Then we can solve a linear PDE for the correction δu and use (5.31) to find a new approximation

$$u^{n,k+1} = u^{n,k} + \delta u$$

to u^n. Repeating this procedure gives a sequence $u^{n,k+1}$, $k = 0, 1, \ldots$ that hopefully converges to the goal u^n.

Let us carry out all the mathematical details for the nonlinear diffusion PDE discretized by the Backward Euler method. Inserting (5.31) in (5.28) gives

$$\frac{u^{n,k} + \delta u - u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^{n,k} + \delta u)\nabla(u^{n,k} + \delta u)) + f(u^{n,k} + \delta u)\,. \tag{5.32}$$

We can Taylor expand $\alpha(u^{n,k} + \delta u)$ and $f(u^{n,k} + \delta u)$:

$$\alpha(u^{n,k}+\delta u) = \alpha(u^{n,k}) + \frac{d\alpha}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx \alpha(u^{n,k}) + \alpha'(u^{n,k})\delta u,$$
$$f(u^{n,k}+\delta u) = f(u^{n,k}) + \frac{df}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx f(u^{n,k}) + f'(u^{n,k})\delta u\,.$$

Inserting the linear approximations of α and f in (5.32) results in

$$\begin{aligned}\frac{u^{n,k}+\delta u - u^{n-1}}{\Delta t} &= \nabla\cdot(\alpha(u^{n,k})\nabla u^{n,k}) + f(u^{n,k}) + \\ &\nabla\cdot(\alpha(u^{n,k})\nabla\delta u) + \nabla\cdot(\alpha'(u^{n,k})\delta u\nabla u^{n,k}) + \\ &\nabla\cdot(\alpha'(u^{n,k})\delta u\nabla\delta u) + f'(u^{n,k})\delta u\,.\end{aligned} \tag{5.33}$$

The term $\alpha'(u^{n,k})\delta u\nabla\delta u$ is of order δu^2 and therefore omitted since we expect the correction δu to be small ($\delta u \gg \delta u^2$). Reorganizing the equation gives a PDE for δu that we can write in short form as

$$\delta F(\delta u; u^{n,k}) = -F(u^{n,k}),$$

where

$$\begin{aligned}F(u^{n,k}) &= \frac{u^{n,k}-u^{n-1}}{\Delta t} - \nabla\cdot(\alpha(u^{n,k})\nabla u^{n,k}) + f(u^{n,k}),\\ \delta F(\delta u; u^{n,k}) &= -\frac{1}{\Delta t}\delta u + \nabla\cdot(\alpha(u^{n,k})\nabla\delta u) + \\ &\nabla\cdot(\alpha'(u^{n,k})\delta u\nabla u^{n,k}) + f'(u^{n,k})\delta u\,.\end{aligned} \tag{5.34}$$

Note that δF is a linear function of δu, and F contains only terms that are known, such that the PDE for δu is indeed linear.

Observations

The notational form $\delta F = -F$ resembles the Newton system $J\delta u = -F$ for systems of algebraic equations, with δF as $J\delta u$. The unknown vector in a linear system of algebraic equations enters the system as a linear operator in terms of a matrix-vector product ($J\delta u$), while at the PDE level we have a linear differential operator instead (δF).

5.20.2. Similarity with Picard iteration

We can rewrite the PDE for δu in a slightly different way too if we define $u^{n,k}+\delta u$ as $u^{n,k+1}$.

$$\begin{aligned}\frac{u^{n,k+1}-u^{n-1}}{\Delta t} &= \nabla\cdot(\alpha(u^{n,k})\nabla u^{n,k+1}) + f(u^{n,k}) \\ &+ \nabla\cdot(\alpha'(u^{n,k})\delta u\nabla u^{n,k}) + f'(u^{n,k})\delta u\,.\end{aligned} \tag{5.35}$$

Note that the first line is the same PDE as arises in the Picard iteration, while the remaining terms arise from the differentiations that are an inherent ingredient in Newton's method.

5.20.3. Implementation

For coding we want to introduce u for u^n, u^- for $u^{n,k}$ and $u^{(1)}$ for u^{n-1}. The formulas for F and δF are then more clearly written as

$$\begin{aligned} F(u^-) &= \frac{u^- - u^{(1)}}{\Delta t} - \nabla\cdot(\alpha(u^-)\nabla u^-) + f(u^-), \\ \delta F(\delta u; u^-) &= -\frac{1}{\Delta t}\delta u + \nabla\cdot(\alpha(u^-)\nabla\delta u) + \\ &\quad \nabla\cdot(\alpha'(u^-)\delta u\nabla u^-) + f'(u^-)\delta u\,. \end{aligned} \tag{5.36}$$

The form that orders the PDE as the Picard iteration terms plus the Newton method's derivative terms becomes

$$\begin{aligned} \frac{u - u^{(1)}}{\Delta t} &= \nabla\cdot(\alpha(u^-)\nabla u) + f(u^-) + \\ &\quad \gamma(\nabla\cdot(\alpha'(u^-)(u-u^-)\nabla u^-) + f'(u^-)(u-u^-))\,. \end{aligned} \tag{5.37}$$

The Picard and full Newton versions correspond to $\gamma = 0$ and $\gamma = 1$, respectively.

5.20.4. Derivation with alternative notation

Some may prefer to derive the linearized PDE for δu using the more compact notation. We start with inserting $u^n = u^- + \delta u$ to get

$$\frac{u^- + \delta u - u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^- + \delta u)\nabla(u^- + \delta u)) + f(u^- + \delta u)\,.$$

Taylor expanding,

$$\begin{aligned} \alpha(u^- + \delta u) &\approx \alpha(u^-) + \alpha'(u^-)\delta u, \\ f(u^- + \delta u) &\approx f(u^-) + f'(u^-)\delta u, \end{aligned}$$

and inserting these expressions gives a less cluttered PDE for δu:

$$\begin{aligned} \frac{u^- + \delta u - u^{n-1}}{\Delta t} &= \nabla\cdot(\alpha(u^-)\nabla u^-) + f(u^-) + \\ &\quad \nabla\cdot(\alpha(u^-)\nabla\delta u) + \nabla\cdot(\alpha'(u^-)\delta u\nabla u^-) + \\ &\quad \nabla\cdot(\alpha'(u^-)\delta u\nabla\delta u) + f'(u^-)\delta u\,. \end{aligned}$$

5.21. Crank-Nicolson discretization

A Crank-Nicolson discretization of the nonlinear diffusion equation applies a centered difference at $t_{n+\frac{1}{2}}$:

$$[D_t u = \nabla\cdot(\alpha(u)\nabla u) + f(u)]^{n+\frac{1}{2}}\,.$$

The standard technique is to apply an arithmetic average for quantities defined between two mesh points, e.g.,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1})\,.$$

However, with nonlinear terms we have many choices of formulating an arithmetic mean:

$$[f(u)]^{n+\frac{1}{2}} \approx f(\frac{1}{2}(u^n + u^{n+1})) = [f(\overline{u}^t)]^{n+\frac{1}{2}}, \tag{5.38}$$

$$[f(u)]^{n+\frac{1}{2}} \approx \frac{1}{2}(f(u^n) + f(u^{n+1})) = [\overline{f(u)}^t]^{n+\frac{1}{2}}, \tag{5.39}$$

$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \alpha(\frac{1}{2}(u^n + u^{n+1}))\nabla(\frac{1}{2}(u^n + u^{n+1})) = [\alpha(\overline{u}^t)\nabla\overline{u}^t]^{n+\frac{1}{2}}, \tag{5.40}$$

$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n) + \alpha(u^{n+1}))\nabla(\frac{1}{2}(u^n + u^{n+1})) = [\overline{\alpha(u)}^t\nabla\overline{u}^t]^{n+\frac{1}{2}}, \tag{5.41}$$

$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n)\nabla u^n + \alpha(u^{n+1})\nabla u^{n+1}) = [\overline{\alpha(u)\nabla u}^t]^{n+\frac{1}{2}}\,. \tag{5.42}$$

A big question is whether there are significant differences in accuracy between taking the products of arithmetic means or taking the arithmetic mean of products. Exercise Section 5.41 investigates this question, and the answer is that the approximation is $\mathcal{O}(\Delta t^2)$ in both cases.

5.22. Discretization in space and Newton's method

Section Section 5.17 presented methods for linearizing time-discrete PDEs directly prior to discretization in space. We can alternatively carry out the discretization in space of the time-discrete nonlinear PDE problem and get a system of nonlinear algebraic equations, which can be solved by Picard iteration or Newton's method as presented in Section Section 5.13. This latter approach will now be described in detail.

We shall work with the 1D problem

$$-(\alpha(u)u')' + au = f(u), \quad x\in(0,L), \quad \alpha(u(0))u'(0) = C,\ u(L) = D\,. \tag{5.43}$$

The problem (5.43) arises from the stationary limit of a diffusion equation,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(u)\frac{\partial u}{\partial x}\right) - au + f(u), \tag{5.44}$$

as $t\to\infty$ and $\partial u/\partial t\to 0$. Alternatively, the problem (5.43) arises at each time level from implicit time discretization of (5.44). For example, a Backward Euler scheme for (5.44) leads to

$$\frac{u^n - u^{n-1}}{\Delta t} = \frac{d}{dx}\left(\alpha(u^n)\frac{du^n}{dx}\right) - au^n + f(u^n)\,. \tag{5.45}$$

Introducing $u(x)$ for $u^n(x)$, $u^{(1)}$ for u^{n-1}, and defining $f(u)$ in (5.43) to be $f(u)$ in (5.45) plus $u^{n-1}/\Delta t$, gives (5.43) with $a = 1/\Delta t$.

5.23. Finite difference discretization

The nonlinearity in the differential equation (5.43) poses no more difficulty than a variable coefficient, as in the term $(\alpha(x)u')'$. We can therefore use a standard finite difference approach when discretizing the Laplace term with a variable coefficient:

$$[-D_x\alpha D_x u + au = f]_i\,.$$

Writing this out for a uniform mesh with points $x_i = i\Delta x$, $i = 0, \ldots, N_x$, leads to

$$-\frac{1}{\Delta x^2}\left(\alpha_{i+\frac{1}{2}}(u_{i+1} - u_i) - \alpha_{i-\frac{1}{2}}(u_i - u_{i-1})\right) + au_i = f(u_i)\,. \tag{5.46}$$

This equation is valid at all the mesh points $i = 0, 1, \ldots, N_x - 1$. At $i = N_x$ we have the Dirichlet condition $u_i = 0$. The only difference from the case with $(\alpha(x)u')'$ and $f(x)$ is that now α and f are functions of u and not only of x: $(\alpha(u(x))u')'$ and $f(u(x))$.

The quantity $\alpha_{i+\frac{1}{2}}$, evaluated between two mesh points, needs a comment. Since α depends on u and u is only known at the mesh points, we need to express $\alpha_{i+\frac{1}{2}}$ in terms of u_i and u_{i+1}. For this purpose we use an arithmetic mean, although a harmonic mean is also common in this context if α features large jumps. There are two choices of arithmetic means:

$$\begin{aligned}
\alpha_{i+\frac{1}{2}} &\approx \alpha(\frac{1}{2}(u_i + u_{i+1}) = [\alpha(\overline{u}^x)]^{i+\frac{1}{2}},\\
\alpha_{i+\frac{1}{2}} &\approx \frac{1}{2}(\alpha(u_i) + \alpha(u_{i+1})) = [\overline{\alpha(u)}^x]^{i+\frac{1}{2}}
\end{aligned} \tag{5.47}$$

Equation (5.46) with the latter approximation then looks like

$$\begin{aligned}
-\frac{1}{2\Delta x^2}&((\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - (\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1}))\\
&+ au_i = f(u_i),
\end{aligned} \tag{5.48}$$

or written more compactly,

$$[-D_x\overline{\alpha}^x D_x u + au = f]_i\,.$$

At mesh point $i = 0$ we have the boundary condition $\alpha(u)u' = C$, which is discretized by

$$[\alpha(u)D_{2x}u = C]_0,$$

meaning

$$\alpha(u_0)\frac{u_1 - u_{-1}}{2\Delta x} = C\,. \tag{5.49}$$

The fictitious value u_{-1} can be eliminated with the aid of (5.48) for $i = 0$. Formally, (5.48) should be solved with respect to u_{i-1} and that value (for $i = 0$) should be inserted in (5.49), but it is algebraically much easier to do it the other way around. Alternatively, one can use a ghost cell $[-\Delta x, 0]$ and update the u_{-1} value in the ghost cell according to (5.49) after every Picard or Newton iteration. Such an approach means that we use a known u_{-1} value in (5.48) from the previous iteration.

5.24. Solution of algebraic equations

5.24.1. The structure of the equation system

The nonlinear algebraic equations (5.48) are of the form $A(u)u = b(u)$ with

$$\begin{aligned} A_{i,i} &= \frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i)\alpha(u_{i+1})) + a, \\ A_{i,i-1} &= -\frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + \alpha(u_i)), \\ A_{i,i+1} &= -\frac{1}{2\Delta x^2}(\alpha(u_i) + \alpha(u_{i+1})), \\ b_i &= f(u_i)\,. \end{aligned}$$

The matrix $A(u)$ is tridiagonal: $A_{i,j} = 0$ for $j > i+1$ and $j < i-1$.

The above expressions are valid for internal mesh points $1 \leq i \leq N_x - 1$. For $i = 0$ we need to express $u_{i-1} = u_{-1}$ in terms of u_1 using (5.49):

$$u_{-1} = u_1 - \frac{2\Delta x}{\alpha(u_0)}C\,. \tag{5.50}$$

This value must be inserted in $A_{0,0}$. The expression for $A_{i,i+1}$ applies for $i = 0$, and $A_{i,i-1}$ does not enter the system when $i = 0$.

Regarding the last equation, its form depends on whether we include the Dirichlet condition $u(L) = D$, meaning $u_{N_x} = D$, in the nonlinear algebraic equation system or not. Suppose we choose $(u_0, u_1, \ldots, u_{N_x-1})$ as unknowns, later referred to as *systems without Dirichlet conditions*. The last equation corresponds to $i = N_x - 1$. It involves the boundary value u_{N_x}, which is substituted by D. If the unknown vector includes the boundary value, $(u_0, u_1, \ldots, u_{N_x})$, later referred to as *system including Dirichlet conditions*, the equation for $i = N_x - 1$ just involves the unknown u_{N_x}, and the final equation becomes $u_{N_x} = D$, corresponding to $A_{i,i} = 1$ and $b_i = D$ for $i = N_x$.

5.24.2. Picard iteration

The obvious Picard iteration scheme is to use previously computed values of u_i in $A(u)$ and $b(u)$, as described more in detail in Section Section 5.13. With the notation u^- for the most recently computed value of u, we have the system $F(u) \approx \hat{F}(u) = A(u^-)u - b(u^-)$, with $F = (F_0, F_1, \ldots, F_m)$, $u = (u_0, u_1, \ldots, u_m)$. The index m is N_x if the system includes the Dirichlet condition as a separate equation and $N_x - 1$ otherwise. The matrix $A(u^-)$ is tridiagonal, so the solution procedure is to fill a tridiagonal matrix data structure and the right-hand side vector with the right numbers and call a Gaussian elimination routine for tridiagonal linear systems.

5.24.3. Mesh with two cells

It helps on the understanding of the details to write out all the mathematics in a specific case with a small mesh, say just two cells ($N_x = 2$). We use u_i^- for the i-th component in u^-.

The starting point is the basic expressions for the nonlinear equations at mesh point $i = 0$ and $i = 1$:

$$A_{0,-1}u_{-1} + A_{0,0}u_0 + A_{0,1}u_1 = b_0, \tag{5.51}$$

$$A_{1,0}u_0 + A_{1,1}u_1 + A_{1,2}u_2 = b_1\,. \tag{5.52}$$

Equation (5.51) written out reads

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u_{-1}) + \alpha(u_0))u_{-1} + \\ &(\alpha(u_{-1}) + 2\alpha(u_0) + \alpha(u_1))u_0 - \\ &(\alpha(u_0) + \alpha(u_1)))u_1 + au_0 = f(u_0)\,.\end{aligned}$$

We must then replace u_{-1} by (5.50). With Picard iteration we get

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u^- * *{-1}) + 2\alpha(u^- * *0) + \alpha(u_1^-))u_1 + \\ &(\alpha(u^- * *{-1}) + 2\alpha(u^- * *0) + \alpha(u_1^-))u_0 + au_0 \\ &= f(u_0^-) - \frac{1}{\alpha(u^- * *0)\Delta x}(\alpha(u^- * *{-1}) + \alpha(u_0^-))C,\end{aligned}$$

where

$$u_{-1}^- = u_1^- - \frac{2\Delta x}{\alpha(u_0^-)}C\,.$$

Equation (5.52) contains the unknown u_2 for which we have a Dirichlet condition. In case we omit the condition as a separate equation, (5.52) with Picard iteration becomes

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u^- * *0) + \alpha(u^- * *1))u_0 + \\ &(\alpha(u^- * *0) + 2\alpha(u^- * *1) + \alpha(u_2^-))u_1 - \\ &(\alpha(u^- * *1) + \alpha(u^- * *2)))u_2 + au_1 = f(u_1^-)\,.\end{aligned}$$

We must now move the u_2 term to the right-hand side and replace all occurrences of u_2 by D:

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u^- * *0) + \alpha(u^- * *1))u_0 + \\ &(\alpha(u^- * *0) + 2\alpha(u^- * *1) + \alpha(D)))u_1 + au_1 \\ &= f(u^- * *1) + \frac{1}{2\Delta x^2}(\alpha(u^- * *1) + \alpha(D))D\,.\end{aligned}$$

The two equations can be written as a 2×2 system:

$$\begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \end{pmatrix},$$

where

$$B_{0,0} = \frac{1}{2\Delta x^2}(\alpha(u^- * *{-1}) + 2\alpha(u^- * *0) + \alpha(u_1^-)) + a, \tag{5.53}$$

$$B_{0,1} = -\frac{1}{2\Delta x^2}(\alpha(u^- * *{-1}) + 2\alpha(u^- * *0) + \alpha(u_1^-)), \tag{5.54}$$

$$B_{1,0} = -\frac{1}{2\Delta x^2}(\alpha(u^- * *0) + \alpha(u^- * *1)), \tag{5.55}$$

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u^- * *0) + 2\alpha(u^- * *1) + \alpha(D)) + a, \tag{5.56}$$

$$d_0 = f(u_0^-) - \frac{1}{\alpha(u^- * *0)\Delta x}(\alpha(u^- * *{-1}) + \alpha(u_0^-))C, \tag{5.57}$$

$$d_1 = f(u^- * *1) + \frac{1}{2\Delta x^2}(\alpha(u^- * *1) + \alpha(D))D\,. \tag{5.58}$$

The system with the Dirichlet condition becomes

$$\begin{pmatrix} B_{0,0} & B_{0,1} & 0 \\ B_{1,0} & B_{1,1} & B_{1,2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ D \end{pmatrix},$$

with

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u^- * *0) + 2\alpha(u^- * *1) + \alpha(u_2)) + a, \tag{5.59}$$

$$B_{1,2} = -\frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(u_2))), \tag{5.60}$$

$$d_1 = f(u_1^-)\,. \tag{5.61}$$

Other entries are as in the 2×2 system.

5.24.4. Newton's method

The Jacobian must be derived in order to use Newton's method. Here it means that we need to differentiate $F(u) = A(u)u - b(u)$ with respect to the unknown parameters $u_0, u_1, \ldots, u_m$ ($m = N_x$ or $m = N_x - 1$, depending on whether the Dirichlet condition is included in the nonlinear system $F(u) = 0$ or not). Nonlinear equation number i has the structure

$$F_i = A_{i,i-1}(u_{i-1}, u_i)u_{i-1} + A_{i,i}(u_{i-1}, u_i, u_{i+1})u_i + A_{i,i+1}(u_i, u_{i+1})u_{i+1} - b_i(u_i)\,.$$

Computing the Jacobian requires careful differentiation. For example,

$$\begin{aligned}\frac{\partial}{\partial u_i}(A_{i,i}(u_{i-1},u_i,u_{i+1})u_i) &= \frac{\partial A_{i,i}}{\partial u_i}u_i + A_{i,i}\frac{\partial u_i}{\partial u_i}\\ &= \frac{\partial}{\partial u_i}(\frac{1}{2\Delta x^2}(\alpha(u_{i-1})+2\alpha(u_i)+\alpha(u_{i+1}))+a)u_i+\\ &\quad \frac{1}{2\Delta x^2}(\alpha(u_{i-1})+2\alpha(u_i)+\alpha(u_{i+1}))+a\\ &= \frac{1}{2\Delta x^2}(2\alpha'(u_i)u_i+\alpha(u_{i-1})+2\alpha(u_i)+\alpha(u_{i+1}))+a\,.\end{aligned}$$

The complete Jacobian becomes

$$\begin{aligned}J_{i,i} = \frac{\partial F_i}{\partial u_i} &= \frac{\partial A_{i,i-1}}{\partial u_i}u_{i-1} + \frac{\partial A_{i,i}}{\partial u_i}u_i + A_{i,i} + \frac{\partial A_{i,i+1}}{\partial u_i}u_{i+1} - \frac{\partial b_i}{\partial u_i}\\ &= \frac{1}{2\Delta x^2}(-\alpha'(u_i)u_{i-1}+2\alpha'(u_i)u_i+\alpha(u_{i-1})+2\alpha(u_i)+\alpha(u_{i+1}))+\\ &\quad a-\frac{1}{2\Delta x^2}\alpha'(u_i)u_{i+1}-b'(u_i),\\ J_{i,i-1} = \frac{\partial F_i}{\partial u_{i-1}} &= \frac{\partial A_{i,i-1}}{\partial u_{i-1}}u_{i-1} + A_{i-1,i} + \frac{\partial A_{i,i}}{\partial u_{i-1}}u_i - \frac{\partial b_i}{\partial u_{i-1}}\\ &= \frac{1}{2\Delta x^2}(-\alpha'(u_{i-1})u_{i-1}-(\alpha(u_{i-1})+\alpha(u_i))+\alpha'(u_{i-1})u_i),\\ J_{i,i+1} = \frac{\partial A_{i,i+1}}{\partial u_{i-1}}u_{i+1} &+ A_{i+1,i} + \frac{\partial A_{i,i}}{\partial u_{i+1}}u_i - \frac{\partial b_i}{\partial u_{i+1}}\\ &= \frac{1}{2\Delta x^2}(-\alpha'(u_{i+1})u_{i+1}-(\alpha(u_i)+\alpha(u_{i+1}))+\alpha'(u_{i+1})u_i)\end{aligned}$$

.The explicit expression for nonlinear equation number i, $F_i(u_0,u_1,\ldots)$, arises from moving the $f(u_i)$ term in (5.48) to the left-hand side:

$$\begin{aligned}F_i = -\frac{1}{2\Delta x^2}&((\alpha(u_i)+\alpha(u_{i+1}))(u_{i+1}-u_i)-(\alpha(u_{i-1})+\alpha(u_i))(u_i-u_{i-1}))\\ &+au_i-f(u_i)=0\,.\end{aligned}\tag{5.62}$$

At the boundary point $i=0$, u_{-1} must be replaced using the formula (5.50). When the Dirichlet condition at $i=N_x$ is not a part of the equation system, the last equation $F_m=0$ for $m=N_x-1$ involves the quantity u_{N_x-1} which must be replaced by D. If u_{N_x} is treated as an unknown in the system, the last equation $F_m=0$ has $m=N_x$ and reads

$$F_{N_x}(u_0,\ldots,u_{N_x}) = u_{N_x}-D=0\,.$$

Similar replacement of u_{-1} and u_{N_x} must be done in the Jacobian for the first and last row. When u_{N_x} is included as an unknown, the last row in the Jacobian must help implement the condition $\delta u_{N_x}=0$, since we assume that u contains the right Dirichlet value at the beginning of the iteration ($u_{N_x}=D$), and then the Newton update should be zero for $i=0$, i.e., $\delta u_{N_x}=0$. This also forces the right-hand side to be $b_i=0$, $i=N_x$.

We have seen, and can see from the present example, that the linear system in Newton's method contains all the terms present in the system that arises in the Picard iteration method. The extra terms in Newton's method can be multiplied by a factor such that it is easy to program one linear system and set this factor to 0 or 1 to generate the Picard or Newton system.

5.25. Solving Nonlinear PDEs with Devito

Having established the finite difference discretization of nonlinear PDEs, we now implement several solvers using Devito. The symbolic approach allows us to express nonlinear equations and handle the time-lagged coefficients naturally.

5.25.1. Nonlinear Diffusion: The Explicit Scheme

The nonlinear diffusion equation

$$u_t = \nabla \cdot (D(u)\nabla u)$$

with solution-dependent diffusivity $D(u)$ requires special treatment. In 1D, the equation becomes:

$$u_t = \frac{\partial}{\partial x}\left(D(u)\frac{\partial u}{\partial x}\right)$$

For explicit time stepping, we evaluate D at the previous time level:

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}\left[D_{i+1/2}^n(u_{i+1}^n - u_i^n) - D_{i-1/2}^n(u_i^n - u_{i-1}^n)\right]$$

where $D_{i+1/2}^n = \frac{1}{2}(D(u_i^n) + D(u_{i+1}^n))$.

5.25.2. The Devito Implementation

```
from devito import Grid, TimeFunction, Eq, Operator, Constant
import numpy as np

# Domain and discretization
L = 1.0           # Domain length
Nx = 100          # Grid points
T = 0.1           # Final time
F = 0.4           # Target Fourier number

dx = L / Nx
D_max = 1.0       # Maximum diffusion coefficient
dt = F * dx**2 / D_max  # Time step from stability

# Create Devito grid
grid = Grid(shape=(Nx + 1,), extent=(L,))

# Time-varying field with space_order=2 for halo access
u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)
```

5.25.3. Handling the Nonlinear Diffusion Coefficient

For nonlinear diffusion, the diffusivity depends on the solution. Common forms include:

Type	$D(u)$	Application
Constant	D_0	Linear heat conduction
Linear	$D_0(1 + \alpha u)$	Temperature-dependent conductivity
Porous medium	$D_0 m u^{m-1}$	Flow in porous media

The `src.nonlin` module provides several diffusion coefficient functions:

```
from src.nonlin import (
    constant_diffusion,
    linear_diffusion,
    porous_medium_diffusion,
)

# Constant D(u) = 1.0
D_const = lambda u: constant_diffusion(u, D0=1.0)

# Linear D(u) = 1 + 0.5*u
D_linear = lambda u: linear_diffusion(u, D0=1.0, alpha=0.5)

# Porous medium D(u) = 2*u (m=2)
D_porous = lambda u: porous_medium_diffusion(u, m=2.0, D0=1.0)
```

5.25.4. Complete Nonlinear Diffusion Solver

The `src.nonlin` module provides `solve_nonlinear_diffusion_explicit`:

```
from src.nonlin import solve_nonlinear_diffusion_explicit
import numpy as np

# Initial condition: smooth bump
def I(x):
    return np.sin(np.pi * x)

result = solve_nonlinear_diffusion_explicit(
    L=1.0,           # Domain length
    Nx=100,          # Grid points
    T=0.1,           # Final time
    F=0.4,           # Fourier number
    I=I,             # Initial condition
    D_func=lambda u: linear_diffusion(u, D0=1.0, alpha=0.5),
)
```

```
print(f"Final time: {result.t:.4f}")
print(f"Max solution: {result.u.max():.6f}")
```

5.25.5. Reaction-Diffusion with Operator Splitting

The reaction-diffusion equation

$$u_t = au_{xx} + R(u)$$

combines diffusion with a nonlinear reaction term. Operator splitting separates these effects:

Lie Splitting (first-order): 1. Solve $u_t = au_{xx}$ for time Δt 2. Solve $u_t = R(u)$ for time Δt

Strang Splitting (second-order): 1. Solve $u_t = R(u)$ for time $\Delta t/2$ 2. Solve $u_t = au_{xx}$ for time Δt 3. Solve $u_t = R(u)$ for time $\Delta t/2$

5.25.6. Reaction Terms

The module provides common reaction terms:

```
from src.nonlin import (
    logistic_reaction,
    fisher_reaction,
    allen_cahn_reaction,
)

# Logistic growth: R(u) = r*u*(1 - u/K)
R_logistic = lambda u: logistic_reaction(u, r=1.0, K=1.0)

# Fisher-KPP: R(u) = r*u*(1 - u)
R_fisher = lambda u: fisher_reaction(u, r=1.0)

# Allen-Cahn: R(u) = u - u^3
R_allen_cahn = lambda u: allen_cahn_reaction(u, epsilon=1.0)
```

5.25.7. Reaction-Diffusion Solver

```
from src.nonlin import solve_reaction_diffusion_splitting

# Initial condition with small perturbation
def I(x):
    return 0.5 * np.sin(np.pi * x)

# Strang splitting (second-order)
result = solve_reaction_diffusion_splitting(
```

```
    L=1.0,
    a=0.1,           # Diffusion coefficient
    Nx=100,
    T=0.5,
    F=0.4,
    I=I,
    R_func=lambda u: fisher_reaction(u, r=1.0),
    splitting="strang",
)
```

The Strang splitting achieves second-order accuracy in time, while Lie splitting is only first-order. For problems with fast reactions or long simulation times, the higher accuracy of Strang splitting is beneficial.

5.25.8. Burgers' Equation

The viscous Burgers' equation

$$u_t + uu_x = \nu u_{xx}$$

is a prototype for nonlinear advection with viscous dissipation. The nonlinear term uu_x can cause shock formation for small ν.

We use the conservative form $(u^2/2)_x$ with centered differences:

```
from src.nonlin import solve_burgers_equation

result = solve_burgers_equation(
    L=2.0,           # Domain length
    nu=0.01,         # Viscosity
    Nx=100,          # Grid points
    T=0.5,           # Final time
    C=0.5,           # Target CFL number
)
```

5.25.9. Stability for Burgers' Equation

The time step must satisfy both the CFL condition for advection:

$$C = \frac{|u|_{\max}\Delta t}{\Delta x} \leq 1$$

and the diffusion stability condition:

$$F = \frac{\nu \Delta t}{\Delta x^2} \leq 0.5$$

The solver automatically chooses Δt to satisfy both conditions with a safety factor.

5.25.10. The Effect of Viscosity

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for ax, nu in zip(axes, [0.1, 0.01]):
    result = solve_burgers_equation(
        L=2.0, nu=nu, Nx=100, T=0.5, C=0.3,
        I=lambda x: np.sin(np.pi * x),
        save_history=True,
    )

    for i in range(0, len(result.t_history), len(result.t_history)//5):
        ax.plot(result.x, result.u_history[i],
                label=f't = {result.t_history[i]:.2f}')

    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(f'Burgers, nu = {nu}')
    ax.legend()
```

Higher viscosity ($\nu = 0.1$) smooths the solution, while lower viscosity ($\nu = 0.01$) allows steeper gradients to develop.

5.25.11. Picard Iteration for Implicit Schemes

For stiff nonlinear problems, implicit time stepping may be necessary. Picard iteration solves the nonlinear system by repeated linearization:

1. Guess $u^{n+1,(0)} = u^n$
2. For $k = 0, 1, 2, \ldots$:
    - Evaluate $D^{(k)} = D(u^{n+1,(k)})$
    - Solve the linear system for $u^{n+1,(k+1)}$
    - Check convergence: $\|u^{n+1,(k+1)} - u^{n+1,(k)}\| < \epsilon$

```
from src.nonlin import solve_nonlinear_diffusion_picard

result = solve_nonlinear_diffusion_picard(
    L=1.0,
    Nx=50,
    T=0.05,
    dt=0.001,        # Can use larger dt than explicit
)
```

The implicit scheme removes the time step restriction but requires solving a linear system at each iteration.

5.25.12. Summary

Key points for nonlinear PDEs with Devito:

1. **Nonlinear diffusion**: Use explicit scheme with lagged coefficient evaluation and Fourier number $F \leq 0.5$
2. **Operator splitting**: Separates diffusion and reaction for reaction-diffusion equations; Strang is second-order
3. **Burgers' equation**: Requires both CFL and diffusion stability conditions; viscosity controls smoothness
4. **Picard iteration**: Enables implicit schemes for stiff problems at the cost of solving linear systems

The `src.nonlin` module provides: - `solve_nonlinear_diffusion_explicit` - `solve_reaction_diffusion_split` - `solve_burgers_equation` - `solve_nonlinear_diffusion_picard` - Diffusion coefficients: `constant_diffusion`, `linear_diffusion`, `porous_medium_diffusion` - Reaction terms: `logistic_reaction`, `fisher_reaction`, `allen_cahn_reaction`

The fundamental ideas in the derivation of F_i and $J_{i,j}$ in the 1D model problem are easily generalized to multi-dimensional problems. Nevertheless, the expressions involved are slightly different, with derivatives in x replaced by ∇, so we present some examples below in detail.

5.26. Finite difference discretization

A typical diffusion equation

$$u_t = \nabla \cdot (\alpha(u)\nabla u) + f(u),$$

can be discretized by (e.g.) a Backward Euler scheme, which in 2D can be written

$$[D_t^- u = D_x \overline{\alpha(u)}^x D_x u + D_y \overline{\alpha(u)}^y D_y u + f(u)]^n_{i,j}\,.$$

We do not dive into the details of handling boundary conditions now. Dirichlet and Neumann conditions are handled as in corresponding linear, variable-coefficient diffusion problems.

Writing the scheme out, putting the unknown values on the left-hand side and known values on the right-hand side, and introducing $\Delta x = \Delta y = h$ to save some writing, one gets

$$\begin{aligned} u^n_{i,j} - \frac{\Delta t}{h^2}(&\frac{1}{2}(\alpha(u^n_{i,j}) + \alpha(u^n_{i+1,j}))(u^n_{i+1,j} - u^n_{i,j}) \\ &- \frac{1}{2}(\alpha(u^n_{i-1,j}) + \alpha(u^n_{i,j}))(u^n_{i,j} - u^n_{i-1,j}) \\ &+ \frac{1}{2}(\alpha(u^n_{i,j}) + \alpha(u^n_{i,j+1}))(u^n_{i,j+1} - u^n_{i,j}) \\ &- \frac{1}{2}(\alpha(u^n_{i,j-1}) + \alpha(u^n_{i,j}))(u^n_{i,j} - u^n_{i-1,j-1})) - \Delta t f(u^n_{i,j}) = u^{n-1}_{i,j} \end{aligned}$$

This defines a nonlinear algebraic system on the form $A(u)u = b(u)$.

5.26.1. Picard iteration

The most recently computed values u^- of u^n can be used in α and f for a Picard iteration, or equivalently, we solve $A(u^-)u = b(u^-)$. The result is a linear system of the same type as arising from $u_t = \nabla\cdot(\alpha(\boldsymbol{x})\nabla u) + f(\boldsymbol{x},t)$.

The Picard iteration scheme can also be expressed in operator notation:

$$[D_t^- u = D_x\overline{\alpha(u^-)}^x D_x u + D_y\overline{\alpha(u^-)}^y D_y u + f(u^-)]^n_{i,j}\,.$$

Newton's method

As always, Newton's method is technically more involved than Picard iteration. We first define the nonlinear algebraic equations to be solved, drop the superscript n (use u for u^n), and introduce $u^{(1)}$ for u^{n-1}:

$$\begin{aligned}F_{i,j} = u_{i,j} - \frac{\Delta t}{h^2}(&\\ &\frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i+1,j}))(u_{i+1,j} - u_{i,j}) -\\ &\frac{1}{2}(\alpha(u_{i-1,j}) + \alpha(u_{i,j}))(u_{i,j} - u_{i-1,j}) +\\ &\frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i,j+1}))(u_{i,j+1} - u_{i,j}) -\\ &\frac{1}{2}(\alpha(u_{i,j-1}) + \alpha(u_{i,j}))(u_{i,j} - u_{i-1,j-1})) - \Delta t\, f(u_{i,j}) - u^{(1)}_{i,j} = 0\,.\end{aligned}$$

It is convenient to work with two indices i and j in 2D finite difference discretizations, but it complicates the derivation of the Jacobian, which then gets four indices. (Make sure you really understand the 1D version of this problem as treated in Section Section 5.23.) The left-hand expression of an equation $F_{i,j} = 0$ is to be differentiated with respect to each of the unknowns $u_{r,s}$ (recall that this is short notation for $u^n_{r,s}$), $r \in \mathcal{I}_x$, $s \in \mathcal{I}_y$:

$$J_{i,j,r,s} = \frac{\partial F_{i,j}}{\partial u_{r,s}}\,.$$

The Newton system to be solved in each iteration can be written as

$$\sum_{r\in\mathcal{I}_x}\sum_{s\in\mathcal{I}_y} J_{i,j,r,s}\delta u_{r,s} = -F_{i,j}, \quad i\in\mathcal{I}_x,\ j\in\mathcal{I}_y\,.$$

Given i and j, only a few r and s indices give nonzero contribution to the Jacobian since $F_{i,j}$ contains $u_{i\pm1,j}$, $u_{i,j\pm1}$, and $u_{i,j}$. This means that $J_{i,j,r,s}$ has nonzero contributions only if $r = i\pm1$, $s = j\pm1$, as well as $r = i$ and $s = j$. The corresponding terms in $J_{i,j,r,s}$ are $J_{i,j,i-1,j}$, $J_{i,j,i+1,j}$, $J_{i,j,i,j-1}$, $J_{i,j,i,j+1}$ and $J_{i,j,i,j}$. Therefore, the left-hand side of the Newton system, $\sum_r\sum_s J_{i,j,r,s}\delta u_{r,s}$ collapses to

$$\begin{aligned}J_{i,j,r,s}\delta u_{r,s} = J_{i,j,i,j}\delta u_{i,j} + J_{i,j,i-1,j}\delta u_{i-1,j} + J_{i,j,i+1,j}\delta u_{i+1,j} + J_{i,j,i,j-1}\delta u_{i,j-1}&\\ + J_{i,j,i,j+1}\delta u_{i,j+1}&\end{aligned}$$

The specific derivatives become

$$\begin{aligned}
J_{i,j,i-1,j} &= \frac{\partial F_{i,j}}{\partial u_{i-1,j}} \\
&= \frac{\Delta t}{h^2}(\alpha'(u_{i-1,j})(u_{i,j} - u_{i-1,j}) - \alpha(u_{i-1,j})(-1)), \\
J_{i,j,i+1,j} &= \frac{\partial F_{i,j}}{\partial u_{i+1,j}} \\
&= \frac{\Delta t}{h^2}(-\alpha'(u_{i+1,j})(u_{i+1,j} - u_{i,j}) - \alpha(u_{i-1,j})), \\
J_{i,j,i,j-1} &= \frac{\partial F_{i,j}}{\partial u_{i,j-1}} \\
&= \frac{\Delta t}{h^2}(\alpha'(u_{i,j-1})(u_{i,j} - u_{i,j-1}) - \alpha(u_{i,j-1})(-1)), \\
J_{i,j,i,j+1} &= \frac{\partial F_{i,j}}{\partial u_{i,j+1}} \\
&= \frac{\Delta t}{h^2}(-\alpha'(u_{i,j+1})(u_{i,j+1} - u_{i,j}) - \alpha(u_{i,j-1}))\,.
\end{aligned}$$

The $J_{i,j,i,j}$ entry has a few more terms and is left as an exercise. Inserting the most recent approximation u^- for u in the J and F formulas and then forming $J\delta u = -F$ gives the linear system to be solved in each Newton iteration. Boundary conditions will affect the formulas when any of the indices coincide with a boundary value of an index.

5.27. Continuation methods

Picard iteration or Newton's method may diverge when solving PDEs with severe nonlinearities. Relaxation with $\omega < 1$ may help, but in highly nonlinear problems it can be necessary to introduce a *continuation parameter* Λ in the problem: $\Lambda = 0$ gives a version of the problem that is easy to solve, while $\Lambda = 1$ is the target problem. The idea is then to increase Λ in steps, $\Lambda_0 = 0, \Lambda_1 < \cdots < \Lambda_n = 1$, and use the solution from the problem with Λ_{i-1} as initial guess for the iterations in the problem corresponding to Λ_i.

The continuation method is easiest to understand through an example. Suppose we intend to solve

$$-\nabla \cdot (||\nabla u||^q \nabla u) = f,$$

which is an equation modeling the flow of a non-Newtonian fluid through a channel or pipe. For $q = 0$ we have the Poisson equation (corresponding to a Newtonian fluid) and the problem is linear. A typical value for pseudo-plastic fluids may be $q_n = -0.8$. We can introduce the continuation parameter $\Lambda \in [0, 1]$ such that $q = q_n\Lambda$. Let $\{\Lambda_\ell\}_{\ell=0}^n$ be the sequence of Λ values in $[0, 1]$, with corresponding q values $\{q_\ell\}_{\ell=0}^n$. We can then solve a sequence of problems

$$-\nabla \cdot \left(||\nabla u^\ell||_\ell^q \nabla u^\ell\right) = f, \quad \ell = 0, \ldots, n,$$

where the initial guess for iterating on u^ℓ is the previously computed solution $u^{\ell-1}$. If a particular Λ_ℓ leads to convergence problems, one may try a smaller increase in Λ: $\Lambda_* = \frac{1}{2}(\Lambda_{\ell-1} + \Lambda_\ell)$, and repeat halving the step in Λ until convergence is reestablished.

5.28. Operator splitting methods

Operator splitting is a natural and old idea. When a PDE or system of PDEs contains different terms expressing different physics, it is natural to use different numerical methods for different physical processes. This can optimize and simplify the overall solution process. The idea was especially popularized in the context of the Navier-Stokes equations and reaction-diffusion PDEs. Common names for the technique are *operator splitting*, *fractional step* methods, and *split-step* methods. We shall stick to the former name. In the context of nonlinear differential equations, operator splitting can be used to isolate nonlinear terms and simplify the solution methods.

A related technique, often known as dimensional splitting or alternating direction implicit (ADI) methods, is to split the spatial dimensions and solve a 2D or 3D problem as two or three consecutive 1D problems, but this type of splitting is not to be further considered here.

5.29. Ordinary operator splitting for ODEs

Consider first an ODE where the right-hand side is split into two terms:

$$u' = f_0(u) + f_1(u)\,.$$

In case f_0 and f_1 are linear functions of u, $f_0 = au$ and $f_1 = bu$, we have $u(t) = Ie^{(a+b)t}$, if $u(0) = I$. When going one time step of length Δt from t_n to t_{n+1}, we have

$$u(t_{n+1}) = u(t_n)e^{(a+b)\Delta t}\,.$$

This expression can be also be written as

$$u(t_{n+1}) = u(t_n)e^{a\Delta t}e^{b\Delta t},$$

or

$$u^* = u(t_n)e^{a\Delta t}, \tag{5.63}$$

$$u(t_{n+1}) = u^* e^{b\Delta t} \tag{5.64}$$

The first step (5.63) means solving $u' = f_0$ over a time interval Δt with $u(t_n)$ as start value. The second step (5.64) means solving $u' = f_1$ over a time interval Δt with the value at the end of the first step as start value. That is, we progress the solution in two steps and solve two ODEs $u' = f_0$ and $u' = f_1$. The order of the equations is not important. From the derivation above we see that solving $u' = f_1$ prior to $u' = f_0$ can equally well be done.

The technique is exact if the ODEs are linear. For nonlinear ODEs it is only an approximate method with error Δt. The technique can be extended to an arbitrary number of steps; i.e., we may split the PDE system into any number of subsystems. Examples will illuminate this principle.

5.30. Strange splitting for ODEs

The accuracy of the splitting method in Section Section 5.29 can be improved from $\mathcal{O}(\Delta t)$ to $\mathcal{O}(\Delta t^2)$ using so-called *Strange splitting*, where we take half a step with the f_0 operator, a full step with the f_1 operator, and finally half another step with the f_0 operator. During a time interval Δt the algorithm can be written as follows.

$$\begin{aligned}
\frac{du^*}{dt} &= f_0(u^*), \quad u^*(t_n) = u(t_n), \quad t \in [t_n, t_n + \frac{1}{2}\Delta t], \\
\frac{du^{***}}{dt} &= f_1(u^{***}), \quad u^{***}(t_n) = u^*(t_{n+\frac{1}{2}}), \quad t \in [t_n, t_n + \Delta t], \\
\frac{du^{**}}{dt} &= f_0(u^{**}), \quad u^{**}(t_{n+\frac{1}{2}}) = u^{***}(t_{n+1}), \quad t \in [t_n + \frac{1}{2}\Delta t, t_n + \Delta t]\,.
\end{aligned}$$

The global solution is set as $u(t_{n+1}) = u^{**}(t_{n+1})$.

There is no use in combining higher-order methods with ordinary splitting since the error due to splitting is $\mathcal{O}(\Delta t)$, but for Strange splitting it makes sense to use schemes of order $\mathcal{O}(\Delta t^2)$.

With the notation introduced for Strange splitting, we may express ordinary first-order splitting as

$$\begin{aligned}
\frac{du^*}{dt} &= f_0(u^*), \quad u^*(t_n) = u(t_n), \quad t \in [t_n, t_n + \Delta t], \\
\frac{du^{**}}{dt} &= f_1(u^{**}), \quad u^{**}(t_n) = u^*(t_{n+1}), \quad t \in [t_n, t_n + \Delta t],
\end{aligned}$$

with global solution set as $u(t_{n+1}) = u^{**}(t_{n+1})$.

5.31. Example: Logistic growth

Let us split the (scaled) logistic equation

$$u' = u(1-u), \quad u(0) = 0.1,$$

with solution $u = (9e^{-t} + 1)^{-1}$, into

$$u' = u - u^2 = f_0(u) + f_1(u), \quad f_0(u) = u, \quad f_1(u) = -u^2\,.$$

We solve $u' = f_0(u)$ and $u' = f_1(u)$ by a Forward Euler step. In addition, we add a method where we solve $u' = f_0(u)$ analytically, since the equation is actually $u' = u$ with solution e^t. The software that accompanies the following methods is the file `split_logistic.py`.

5.31.1. Splitting techniques

Ordinary splitting takes a Forward Euler step for each of the ODEs according to

$$\frac{u^{*,n+1}-u^{*,n}}{\Delta t} = f_0(u^{*,n}), \quad u^{*,n}=u(t_n), \quad t\in[t_n,t_n+\Delta t], \tag{5.65}$$

$$\frac{u^{**,n+1}-u^{**,n}}{\Delta t} = f_1(u^{**,n}), \quad u^{**,n}=u^{*,n+1}, \quad t\in[t_n,t_n+\Delta t], \tag{5.66}$$

with $u(t_{n+1}) = u^{**,n+1}$.

Strange splitting takes the form

$$\frac{u^{*,n+\frac{1}{2}}-u^{*,n}}{\frac{1}{2}\Delta t} = f_0(u^{*,n}), \quad u^{*,n}=u(t_n), \quad t\in[t_n,t_n+\frac{1}{2}\Delta t], \tag{5.67}$$

$$\frac{u^{***,n+1}-u^{***,n}}{\Delta t} = f_1(u^{***,n}), \quad u^{***,n}=u^{*,n+\frac{1}{2}}, \quad t\in[t_n,t_n+\Delta t], \tag{5.68}$$

$$\frac{u^{**,n+1}-u^{**,n+\frac{1}{2}}}{\frac{1}{2}\Delta t} = f_0(u^{**,n+\frac{1}{2}}), \quad u^{**,n+\frac{1}{2}}=u^{***,n+1}, \quad t\in[t_n+\frac{1}{2}\Delta t,t_n+\Delta t]\,. \tag{5.69}$$

5.31.2. Verbose implementation

The following function computes four solutions arising from the Forward Euler method, ordinary splitting, Strange splitting, as well as Strange splitting with exact treatment of $u' = f_0(u)$:

```
import numpy as np


def solver(dt, T, f, f_0, f_1):
    """
    Solve u'=f by the Forward Euler method and by ordinary and
    Strange splitting: f(u) = f_0(u) + f_1(u).
    """
    Nt = int(round(T / float(dt)))
    t = np.linspace(0, Nt * dt, Nt + 1)
    u_FE = np.zeros(len(t))
    u_split1 = np.zeros(len(t))  # 1st-order splitting
    u_split2 = np.zeros(len(t))  # 2nd-order splitting
    u_split3 = np.zeros(len(t))  # 2nd-order splitting w/exact f_0

    u_FE[0] = 0.1
    u_split1[0] = 0.1
    u_split2[0] = 0.1
    u_split3[0] = 0.1
```

```
    for n in range(len(t) - 1):
        u_FE[n + 1] = u_FE[n] + dt * f(u_FE[n])

        u_s_n = u_split1[n]
        u_s = u_s_n + dt * f_0(u_s_n)
        u_ss_n = u_s
        u_ss = u_ss_n + dt * f_1(u_ss_n)
        u_split1[n + 1] = u_ss

        u_s_n = u_split2[n]
        u_s = u_s_n + dt / 2.0 * f_0(u_s_n)
        u_sss_n = u_s
        u_sss = u_sss_n + dt * f_1(u_sss_n)
        u_ss_n = u_sss
        u_ss = u_ss_n + dt / 2.0 * f_0(u_ss_n)
        u_split2[n + 1] = u_ss

        u_s_n = u_split3[n]
        u_s = u_s_n * np.exp(dt / 2.0)  # exact
        u_sss_n = u_s
        u_sss = u_sss_n + dt * f_1(u_sss_n)
        u_ss_n = u_sss
        u_ss = u_ss_n * np.exp(dt / 2.0)  # exact
        u_split3[n + 1] = u_ss

    return u_FE, u_split1, u_split2, u_split3, t
```

5.31.3. Compact implementation

We have used quite many lines for the steps in the splitting methods. Many will prefer to condense the code a bit, as done here:

5.31.4. Results

Figure Figure 5.3 shows that the impact of splitting is significant. Interestingly, however, the Forward Euler method applied to the entire problem directly is much more accurate than any of the splitting schemes. We also see that Strange splitting is definitely more accurate than ordinary splitting and that it helps a bit to use an exact solution of $u' = f_0(u)$. With a large time step ($\Delta t = 0.2$, left plot in Figure Figure 5.3), the asymptotic values are off by 20-30%. A more reasonable time step ($\Delta t = 0.05$, right plot in Figure Figure 5.3) gives better results, but still the asymptotic values are up to 10% wrong.

As technique for solving nonlinear ODEs, we realize that the present case study is not particularly promising, as the Forward Euler method both linearizes the original problem and provides a solution

that is much more accurate than any of the splitting techniques. In complicated multi-physics settings, on the other hand, splitting may be the only feasible way to go, and sometimes you really need to apply different numerics to different parts of a PDE problem. But in very simple problems, like the logistic ODE, splitting is just an inferior technique. Still, the logistic ODE is ideal for introducing all the mathematical details and for investigating the behavior.

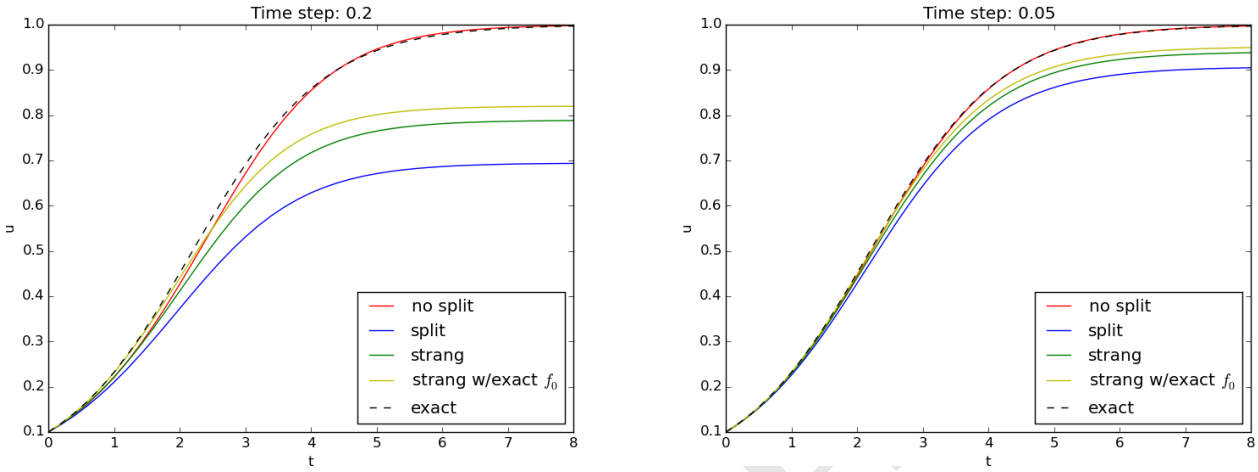


Figure 5.3.: Effect of ordinary and Strange splitting for the logistic equation.

5.32. Reaction-diffusion equation

Consider a diffusion equation coupled to chemical reactions modeled by a nonlinear term $f(u)$:

$$\frac{\partial u}{\partial t} = \alpha\nabla^2 u + f(u)\,.$$

This is a physical process composed of two individual processes: u is the concentration of a substance that is locally generated by a chemical reaction $f(u)$, while u is spreading in space because of diffusion. There are obviously two time scales: one for the chemical reaction and one for diffusion. Typically, fast chemical reactions require much finer time stepping than slower diffusion processes. It could therefore be advantageous to split the two physical effects in separate models and use different numerical methods for the two.

A natural spitting in the present case is

$$\begin{aligned}\frac{\partial u^*}{\partial t} &= \alpha\nabla^2 u^*,\\ \frac{\partial u^{**}}{\partial t} &= f(u^{**})\,.\end{aligned} \tag{5.70}$$

Looking at these familiar problems, we may apply a θ rule (implicit) scheme for (5.70) over one time step and avoid dealing with nonlinearities by applying an explicit scheme for (5.70) over the same time step.

Suppose we have some solution u at time level t_n. For flexibility, we define a θ method for the diffusion part (5.70) by

$$[D_t u^* = \alpha(D_x D_x u^* + D_y D_y u^*)]^{n+\theta}\,.$$

We use u^n as initial condition for u^*.

The reaction part, which is defined at each mesh point (without coupling values in different mesh points), can employ any scheme for an ODE. Here we use an Adams-Bashforth method of order 2. Recall that the overall accuracy of the splitting method is maximum $\mathcal{O}(\Delta t^2)$ for Strange splitting, otherwise it is just $\mathcal{O}(\Delta t)$. Higher-order methods for ODEs will therefore be a waste of work. The 2nd-order Adams-Bashforth method reads

$$u^{**,n+1} * *i,j = u^{**,n} * *i,j + \frac{1}{2}\Delta t \left(3f(u^{**,n}_{i,j}, t_n) - f(u^{**,n-1} * *i,j,t * *n-1)\right) .$$

We can use a Forward Euler step to start the method, i.e, compute $u^{**,1}_{i,j}$.

The algorithm goes like this:

1. Solve the diffusion problem for one time step as usual.
2. Solve the reaction ODEs at each mesh point in $[t_n, t_n + \Delta t]$, using the diffusion solution in 1. as initial condition. The solution of the ODEs constitutes the solution of the original problem at the end of each time step.

We may use a much smaller time step when solving the reaction part, adapted to the dynamics of the problem $u' = f(u)$. This gives great flexibility in splitting methods.

5.33. Example: Reaction-Diffusion with linear reaction term

The methods above may be explored in detail through a specific computational example in which we compute the convergence rates associated with four different solution approaches for the reaction-diffusion equation with a linear reaction term, i.e. $f(u) = -bu$. The methods comprise solving without splitting (just straight Forward Euler), ordinary splitting, first order Strange splitting, and second order Strange splitting. In all four methods, a standard centered difference approximation is used for the spatial second derivative. The methods share the error model $E = Ch^r$, while differing in the step h (being either Δx^2 or Δx) and the convergence rate r (being either 1 or 2).

All code commented below is found in the file `split_diffu_react.py`. When executed, a function `convergence_rates` is called, from which all convergence rate computations are handled:

```
def convergence_rates(scheme="diffusion"):
    """Computes empirical conv. rates for the different
    splitting schemes"""

    F = 0.5
    T = 1.2
    a = 3.5
    b = 1
    L = 1.5
    k = np.pi / L

    def exact(x, t):
        """exact sol. to: du/dt = a*d^2u/dx^2 - b*u"""
```

```
        return np.exp(-(a * k**2 + b) * t) * np.sin(k * x)

    def f(u, t):
        return -b * u

    def I(x):
        return exact(x, 0)

    global error  # error computed in the user action function
    error = 0

    def action(u, x, t, n):
        global error
        if n == 1:  # New simulation, - reset error
            error = 0
        else:
            error = max(error, np.abs(u - exact(x, t[n])).max())

    E = []
    h = []
    Nx_values = [10, 20, 40, 80, 160]
    for Nx in Nx_values:
        dx = L / Nx
        dt = F / a * dx**2
        Nt = int(round(T / float(dt)))
        t = np.linspace(0, Nt * dt, Nt + 1)  # Mesh points, global time

        if scheme == "diffusion":
            print("Running FE on whole eqn...")
            diffusion_FE(I, a, f, L, dt, F, t, T, step_no=0, user_action=action)
        elif scheme == "ordinary_splitting":
            print("Running ordinary splitting...")
            ordinary_splitting(
                I=I,
                a=a,
                b=b,
                f=f,
                L=L,
                dt=dt,
                dt_Rfactor=1,
                F=F,
                t=t,
                T=T,
                user_action=action,
            )
        elif scheme == "Strange_splitting_1stOrder":
            print("Running Strange splitting with 1st order schemes...")
```

```
            Strange_splitting_1stOrder(
                I=I,
                a=a,
                b=b,
                f=f,
                L=L,
                dt=dt,
                dt_Rfactor=1,
                F=F,
                t=t,
                T=T,
                user_action=action,
            )
        elif scheme == "Strange_splitting_2andOrder":
            print("Running Strange splitting with 2nd order schemes...")
            Strange_splitting_2andOrder(
                I=I,
                a=a,
                b=b,
                f=f,
                L=L,
                dt=dt,
                dt_Rfactor=1,
                F=F,
                t=t,
                T=T,
                user_action=action,
            )
        else:
            print("Unknown scheme requested!")
            sys.exit(0)

        h.append(dt)
        E.append(error)

    print("E:", E)
    print("h:", h)

    r = [
        np.log(E[i] / E[i - 1]) / np.log(h[i] / h[i - 1])
        for i in range(1, len(Nx_values))
    ]
    print("Computed rates:", r)


if __name__ == "__main__":
    schemes = [
```

```
        "diffusion",
        "ordinary_splitting",
        "Strange_splitting_1stOrder",
        "Strange_splitting_2andOrder",
    ]

    for scheme in schemes:
        convergence_rates(scheme=scheme)
```

Now, with respect to the error ($E = Ch^r$), the Forward Euler scheme, the ordinary splitting scheme and first order Strange splitting scheme are all first order ($r = 1$), with a step $h = \Delta x^2 = K^{-1}\Delta t$, where K is some constant. This implies that the *ratio* $\frac{\Delta t}{\Delta x^2}$ must be held constant during convergence rate calculations. Furthermore, the Fourier number $F = \frac{\alpha \Delta t}{\Delta x^2}$ is upwards limited to $F = 0.5$, being the stability limit with explicit schemes. Thus, in these cases, we use the fixed value of F and a given (but changing) spatial resolution Δx to compute the corresponding value of Δt according to the expression for F. This assures that $\frac{\Delta t}{\Delta x^2}$ is kept constant. The loop in `convergence_rates` runs over a chosen set of grid points (`Nx_values`) which gives a doubling of spatial resolution with each iteration (Δx is halved).

For the second order Strange splitting scheme, we have $r = 2$ and a step $h = \Delta x = K^{-1}\Delta t$, where K again is some constant. In this case, it is thus the ratio $\frac{\Delta t}{\Delta x}$ that must be held constant during the convergence rate calculations. From the expression for F, it is clear then that F must change with each halving of Δx. In fact, if F is doubled each time Δx is halved, the ratio $\frac{\Delta t}{\Delta x}$ will be constant (this follows, e.g., from the expression for F). This is utilized in our code.

A solver `diffusion_theta` is used in each of the four solution approaches:

```
def diffusion_theta(
    I, a, f, L, dt, F, t, T, step_no, theta=0.5, u_L=0, u_R=0, user_action=None
):
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time (no restriction on F,
    i.e., the time step when theta >= 0.5).
    Vectorized implementation and sparse (tridiagonal)
    coefficient matrix.
    """

    Nt = int(round(T / float(dt)))
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u = np.zeros(Nx + 1)  # solution array at t[n+1]
    u_1 = np.zeros(Nx + 1)  # solution at t[n]
```

```
    diagonal = np.zeros(Nx + 1)
    lower = np.zeros(Nx)
    upper = np.zeros(Nx)
    b = np.zeros(Nx + 1)

    Fl = F * theta
    Fr = F * (1 - theta)
    diagonal[:] = 1 + 2 * Fl
    lower[:] = -Fl  # 1
    upper[:] = -Fl  # 1
    diagonal[0] = 1
    upper[0] = 0
    diagonal[Nx] = 1
    lower[-1] = 0

    diags = [0, -1, 1]
    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1],
        shape=(Nx + 1, Nx + 1),
        format="csr",
    )

    if f is None or f == 0:
        f = lambda x, t: np.zeros(x.size) if isinstance(x, np.ndarray) else 0

    if isinstance(I, np.ndarray):  # I is an array
        u_1 = np.copy(I)
    else:  # I is a function
        for i in range(0, Nx + 1):
            u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, step_no + 0)

    for n in range(0, Nt):
        b[1:-1] = (
            u_1[1:-1]
            + Fr * (u_1[:-2] - 2 * u_1[1:-1] + u_1[2:])
            + dt * theta * f(u_1[1:-1], t[step_no + n + 1])
            + dt * (1 - theta) * f(u_1[1:-1], t[step_no + n])
        )
        b[0] = u_L
        b[-1] = u_R  # boundary conditions
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
```

```
            user_action(u, x, t, step_no + (n + 1))

        u_1, u = u, u_1

    return u_1
```

For the no splitting approach with Forward Euler in time, this solver handles both the diffusion and the reaction term. When splitting, `diffusion_theta` takes care of the diffusion term only, while the reaction term is handled either by a Forward Euler scheme in `reaction_FE`, or by a second order Adams-Bashforth scheme from Odespy. The `reaction_FE` function covers one complete time step `dt` during ordinary splitting, while Strange splitting (both first and second order) applies it with `dt/2` twice during each time step `dt`. Since the reaction term typically represents a much faster process than the diffusion term, a further refinement of the time step is made possible in `reaction_FE`. It was implemented as

```
def reaction_FE(I, f, L, Nx, dt, dt_Rfactor, t, step_no, user_action=None):
    """Reaction solver, Forward Euler method.
    Note that t covers the whole global time interval.
    dt is the step of the diffustion part, i.e. there
    is a local time interval [0, dt] the reaction_FE
    deals with each time it is called. step_no keeps
    track of the (global) time step number (required
    for lookup in t).
    """


    u = np.copy(I)
    dt_local = dt / float(dt_Rfactor)
    Nt_local = int(round(dt / float(dt_local)))
    x = np.linspace(0, L, Nx + 1)

    for n in range(Nt_local):
        time = t[step_no] + n * dt_local
        u[1:Nx] = u[1:Nx] + dt_local * f(u[1:Nx], time)


    return u
```

With the ordinary splitting approach, each time step `dt` is covered twice. First computing the impact of the reaction term, then the contribution from the diffusion term:

```
def ordinary_splitting(I, a, b, f, L, dt, dt_Rfactor, F, t, T, user_action=None):
    """1st order scheme, i.e. Forward Euler is enough for both
    the diffusion and the reaction part. The time step dt is
    given for the diffusion step, while the time step for the
    reaction part is found as dt/dt_Rfactor, where dt_Rfactor >= 1.
```

```
    """
    Nt = int(round(T / float(dt)))
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)  # Mesh points in space
    u = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u[i] = I(x[i])

    for n in range(0, Nt):

        u_s = diffusion_FE(
            I=u, a=a, f=0, L=L, dt=dt, F=F, t=t, T=dt, step_no=n, user_action=None
        )
        u = reaction_FE(
            I=u_s,
            f=f,
            L=L,
            Nx=Nx,
            dt=dt,
            dt_Rfactor=dt_Rfactor,
            t=t,
            step_no=n,
            user_action=None,
        )

        if user_action is not None:
            user_action(u, x, t, n + 1)
```

For the two Strange splitting approaches, each time step `dt` is handled by first computing the reaction step for (the first) `dt/2`, followed by a diffusion step `dt`, before the reaction step is treated once again for (the remaining) `dt/2`. Since first order Strange splitting is no better than first order accurate, both the reaction and diffusion steps are computed explicitly. The solver was implemented as

```
def Strange_splitting_1stOrder(I, a, b, f, L, dt, dt_Rfactor, F, t, T, user_action=None):
    """Strange splitting while still using FE for the diffusion
    step and for the reaction step. Gives 1st order scheme.
    Introduce an extra time mesh t2 for the diffusion part,
    since it steps dt/2.
    """
    Nt = int(round(T / float(dt)))
    t2 = np.linspace(0, Nt * dt, (Nt + 1) + Nt)  # Mesh points in diff
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)
```

```
    u = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u[i] = I(x[i])

    for n in range(0, Nt):
        u_s = diffusion_FE(
            I=u,
            a=a,
            f=0,
            L=L,
            dt=dt / 2.0,
            F=F / 2.0,
            t=t2,
            T=dt / 2.0,
            step_no=2 * n,
            user_action=None,
        )

        u_sss = reaction_FE(
            I=u_s,
            f=f,
            L=L,
            Nx=Nx,
            dt=dt,
            dt_Rfactor=dt_Rfactor,
            t=t,
            step_no=n,
            user_action=None,
        )

        u = diffusion_FE(
            I=u_sss,
            a=a,
            f=0,
            L=L,
            dt=dt / 2.0,
            F=F / 2.0,
            t=t2,
            T=dt / 2.0,
            step_no=2 * n + 1,
            user_action=None,
        )

        if user_action is not None:
            user_action(u, x, t, n + 1)
```

The second order version of the Strange splitting approach utilizes a second order Adams-Bashforth solver for the reaction part and a Crank-Nicolson scheme for the diffusion part. The solver has the same structure as the one for first order Strange splitting and was implemented as

```
def Strange_splitting_2andOrder(I, a, b, f, L, dt, dt_Rfactor, F, t, T, user_action=None):
    """Strange splitting using Crank-Nicolson for the diffusion
    step (theta-rule) and Adams-Bashforth 2 for the reaction step.
    Gives 2nd order scheme. Introduce an extra time mesh t2 for
    the diffusion part, since it steps dt/2.
    """
    import odespy

    Nt = int(round(T / float(dt)))
    t2 = np.linspace(0, Nt * dt, (Nt + 1) + Nt)  # Mesh points in diff
    dx = np.sqrt(a * dt / F)
    Nx = int(round(L / dx))
    x = np.linspace(0, L, Nx + 1)
    u = np.zeros(Nx + 1)

    for i in range(0, Nx + 1):
        u[i] = I(x[i])

    reaction_solver = odespy.AdamsBashforth2(f)

    for n in range(0, Nt):
        u_s = diffusion_theta(
            I=u,
            a=a,
            f=0,
            L=L,
            dt=dt / 2.0,
            F=F / 2.0,
            t=t2,
            T=dt / 2.0,
            step_no=2 * n,
            theta=0.5,
            u_L=0,
            u_R=0,
            user_action=None,
        )


        reaction_solver.set_initial_condition(u_s)
        t_points = np.linspace(0, dt, dt_Rfactor + 1)
        u_AB2, t_ = reaction_solver.solve(t_points)  # t_ not needed
        u_sss = u_AB2[-1, :]  # pick sol at last point in time
```

```
        u = diffusion_theta(
            I=u_sss,
            a=a,
            f=0,
            L=L,
            dt=dt / 2.0,
            F=F / 2.0,
            t=t2,
            T=dt / 2.0,
            step_no=2 * n + 1,
            theta=0.5,
            u_L=0,
            u_R=0,
            user_action=None,
        )


        if user_action is not None:
            user_action(u, x, t, n + 1)
```

When executing `split_diffu_react.py`, we find that the estimated convergence rates are as expected. The second order Strange splitting gives the least error (about $4e^{-5}$) and has second order convergence ($r = 2$), while the remaining three approaches have first order convergence ($r = 1$).

5.34. Analysis of the splitting method

Let us address a linear PDE problem for which we can develop analytical solutions of the discrete equations, with and without splitting, and discuss these. Choosing $f(u) = -\beta u$ for a constant β gives a linear problem. We use the Forward Euler method for both the PDE and ODE problems.

We seek a 1D Fourier wave component solution of the problem, assuming homogeneous Dirichlet conditions at $x = 0$ and $x = L$:

$$u = e^{-\alpha k^2 t - \beta t} \sin kx, \quad k = \frac{\pi}{L}.$$

This component fits the 1D PDE problem ($f = 0$). On complex form we can write

$$u = e^{-\alpha k^2 t - \beta t + ikx},$$

where $i = \sqrt{-1}$ and the imaginary part is taken as the physical solution.

We refer to Section 3.15 and to the book (Langtangen 2016b) for a discussion of exact numerical solutions to diffusion and decay problems, respectively. The key idea is to search for solutions $A^n e^{ikx}$ and determine A. For the diffusion problem solved by a Forward Euler method one has

$$A = 1 - 4F \sin^p,$$

where $F = \alpha\Delta t/\Delta x^2$ is the mesh Fourier number and $p = k\Delta x/2$ is a dimensionless number reflecting the spatial resolution (number of points per wave length in space). For the decay problem $u' = -\beta u$, we have $A = 1 - q$, where q is a dimensionless parameter for the resolution in the decay problem: $q = \beta\Delta t$.

The original model problem can also be discretized by a Forward Euler scheme,

$$[D_t^+ u = \alpha D_x D_x u - \beta u]_i^n \,.$$

Assuming $A^n e^{ikx}$ we find that

$$u_i^n = (1 - 4F\sin^p - q)^n \sin kx \,.$$

We are particularly interested in what happens at one time step. That is,

$$u_i^n = (1 - 4F\sin^2 p)u_i^{n-1} \,.$$

In the two stage algorithm, we first compute the diffusion step

$$u_i^{*,n+1} = (1 - 4F\sin^2 p)u_i^{n-1} \,.$$

Then we use this as input to the decay algorithm and arrive at

$$u^{**,n+1} = (1 - q)u^{*,n+1} = (1 - q)(1 - 4F\sin^2 p)u_i^{n-1} \,.$$

The splitting approximation over one step is therefore

$$E = 1 - 4F\sin^p - q - (1 - q)(1 - 4F\sin^2 p) = -q(2 - F\sin^2 p)) \,.$$

5.35. Problem: Determine if equations are nonlinear or not

Classify each term in the following equations as linear or nonlinear. Assume that u, $\boldsymbol{u}$, and p are unknown functions and that all other symbols are known quantities.

1. $mu'' + \beta|u'|u' + cu = F(t)$
2. $u_t = \alpha u_{xx}$
3. $u_{tt} = c^2\nabla^2 u$
4. $u_t = \nabla\cdot(\alpha(u)\nabla u) + f(x, y)$
5. $u_t + f(u)_x = 0$
6. $\boldsymbol{u}_t + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + r\nabla^2\boldsymbol{u}$, $\nabla\cdot\boldsymbol{u} = 0$ ($\boldsymbol{u}$ is a vector field)
7. $u' = f(u, t)$
8. $\nabla^2 u = \lambda e^u$

Solution

1. mu'' is linear; $\beta|u'|u'$ is nonlinear; cu is linear; $F(t)$ does not contain the unknown u and is hence constant in u, so the term is linear.
2. u_t is linear; αu_{xx} is linear.
3. u_{tt} is linear; $c^2\nabla^2 u$ is linear.

4. u_t is linear; $\nabla \cdot (\alpha(u)\nabla u)$ is nonlinear; $f(x,y)$ is constant in u and hence linear.
5. u_t is linear; $f(u)_x$ is nonlinear if f is nonlinear in u.
6. $\boldsymbol{u}_t$ is linear; $\boldsymbol{u}\cdot\nabla\boldsymbol{u}$ is nonlinear; $-\nabla p$ is linear (in p); $r\nabla^2\boldsymbol{u}$ is linear; $\nabla\cdot\boldsymbol{u}$ is linear.
7. u' is linear; $f(u,t)$ is nonlinear if f is nonlinear in u.
8. $\nabla^2 u$ is linear; λe^u is nonlinear.

5.36. Exercise: Derive a relaxation formula

Derive (5.9) in Section Section 5.9.

5.37. Problem: Derive and investigate a generalized logistic model

The logistic model for population growth is derived by assuming a nonlinear growth rate,

$$u' = a(u)u, \quad u(0) = I, \tag{5.71}$$

and the logistic model arises from the simplest possible choice of $a(u)$: $r(u) = \varrho(1 - u/M)$, where M is the maximum value of u that the environment can sustain, and ϱ is the growth under unlimited access to resources (as in the beginning when u is small). The idea is that $a(u) \sim \varrho$ when u is small and that $a(t) \to 0$ as $u \to M$.

An $a(u)$ that generalizes the linear choice is the polynomial form

$$a(u) = \varrho(1 - u/M)^p, \tag{5.72}$$

where $p > 0$ is some real number.

a)

Formulate a Forward Euler, Backward Euler, and a Crank-Nicolson scheme for (5.71).

Use a geometric mean approximation in the Crank-Nicolson scheme:

$[a(u)u]^{n+1/2} \approx a(u^n)u^{n+1}$.

Solution

The Forward Euler scheme reads

$$[D_t^+ u = a(u)u]^n,$$

or written out,

$$\frac{u^{n+1} - u^n}{\Delta t} = a(u^n)u^n\,.$$

The scheme is linear in the unknown u^{n+1}:

$$u^{n+1} = u^n + \Delta t a(u^n)u^n\,.$$

The Backward Euler scheme,

$$[D_t^- u = a(u)u]^n,$$

becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = a(u^n)u^n,$$

which is a nonlinear equation in the unknown u, here expressed as u^{n+1}:

$$u^{n+1} - \Delta t a(u^{n+1})u^{n+1} = u^n\,.$$

The standard Crank-Nicolson scheme,

$$D_t u = \overline{a(u)u}^t]^{n+\frac{1}{2}},$$

takes the form

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}a(u^n)u^n + \frac{1}{2}a(u^{n+1})u^{n+1}\,.$$

This is a nonlinear equation in the unknown u^{n+1},

$$u^{n+1} - \frac{1}{2}\Delta t a(u^{n+1})u^{n+1} = u^n + \frac{1}{2}\Delta t a(u^n)u^n\,.$$

However, with the suggested geometric mean, the $a(u)u$ term is linearized:

$$\frac{u^{n+1} - u^n}{\Delta t} = a(u^n)u^{n+1},$$

leading to a linear equation in u^{n+1}:

$$(1 - \Delta t a(u^n))u^{n+1} = u^n\,.$$

b)

Formulate Picard and Newton iteration for the Backward Euler scheme in a).

Solution

A Picard iteration for

$$u^{n+1} - \Delta t a(u^{n+1})u^{n+1} = u^n\,.$$

applies old values in for u^{n+1} in $a(u^{n+1})$. If u^- is the most recently computed approximation to u^{n+1}, we can write the Picard linearization as

$$(1 - \Delta t a(u^-))u^{n+1} = u^n\,.$$

Alternatively, with an iteration index k,

$$(1 - \Delta t a(u^{n+1,k}))u^{n+1,k+1} = u^n\,.$$

Newton's method starts with identifying the nonlinear equation as $F(u) = 0$, and here

$$F(u) = u - \Delta t a(u)u - u^n\,.$$

The Jacobian is

$$J(u) = \frac{F(u)}{du} = 1 - \Delta t(a'(u)u + a(u))\,.$$

The key equation in Newton's method is then

$$J(u^-)\delta u = -F(u^-), \quad u \leftarrow u - \delta u\,.$$

c)

Implement the numerical solution methods from a) and b). Use `logistic.py` to compare the case $p = 1$ and the choice (5.72).

Solution

We specialize the code for $a(u)$ to (5.72) since the code was developed from `logistic.py`. It is convenient to work with a dimensionless form of the problem. Choosing a time scale $t_c = 1\varrho$ and a scale for u, $u_c = M$, leads to

$$u' = \varrho(1-u)^p u, \quad u(0) = \alpha,$$

where α is a dimensionless number

$$\alpha = \frac{I}{M}\,.$$

The three schemes can be implemented as follows.

```python
import numpy as np


def FE_logistic(p, u0, dt, Nt):
    u = np.zeros(Nt + 1)
    u[0] = u0
    for n in range(Nt):
        u[n + 1] = u[n] + dt * (1 - u[n]) ** p * u[n]
    return u


def BE_logistic(p, u0, dt, Nt, choice="Picard", eps_r=1e-3, omega=1, max_iter=1000):
    if choice == "Picard1":
        choice = "Picard"
        max_iter = 1

    u = np.zeros(Nt + 1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt + 1):
        c = -u[n - 1]
        if choice == "Picard":

            def F(u):
                return -dt * (1 - u) ** p * u + u + c

            u_ = u[n - 1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = omega * (-c / (1 - dt * (1 - u_) ** p)) + (1 - omega) * u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == "Newton":

            def F(u):
                return -dt * (1 - u) ** p * u + u + c

            def dF(u):
                return dt * p * (1 - u) ** (p - 1) * u - dt * (1 - u) ** p + 1

            u_ = u[n - 1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_) / dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)
    return u, iterations


def CN_logistic(p, u0, dt, Nt):
```

A first verification is to choose $p = 1$ and compare the results with those from `logistic.py`. The number of iterations and the final numerical answers should be identical.

d)

Implement unit tests that check the asymptotic limit of the solutions: $u \to M$ as $t \to \infty$.

You need to experiment to find what "infinite time" is

(increases substantially with p) and what the appropriate tolerance is for testing the asymptotic limit.

Solution

The test function may look like

```
def test_asymptotic_value():
    T = 100
    dt = 0.1
    Nt = int(round(T / float(dt)))
    u0 = 0.1
    p = 1.8

    u_CN = CN_logistic(p, u0, dt, Nt)
    u_BE_Picard, iter_Picard = BE_logistic(
        p, u0, dt, Nt, choice="Picard", eps_r=1e-5, omega=1, max_iter=1000
    )
    u_BE_Newton, iter_Newton = BE_logistic(
        p, u0, dt, Nt, choice="Newton", eps_r=1e-5, omega=1, max_iter=1000
    )
    u_FE = FE_logistic(p, u0, dt, Nt)

    for arr in u_CN, u_BE_Picard, u_BE_Newton, u_FE:
        expected = 1
        computed = arr[-1]
        tol = 0.01
        msg = f"expected={expected}, computed={computed}"
        print(msg)
        assert abs(expected - computed) < tol
```

It is important with a sufficiently small `eps_r` tolerance for the asymptotic value to be accurate (using `eps_r=1E-3` leads to a value 0.92 at $t = T$ instead of 0.994 when `eps_r=1E-5`).

e)

Perform experiments with Newton and Picard iteration for the model (5.72). See how sensitive the number of iterations is to Δt and p.

Solution

Appropriate code is

```
import matplotlib.pyplot as plt


def demo():
    T = 12
    p = 1.2
    try:
        dt = float(sys.argv[1])
        eps_r = float(sys.argv[2])
        omega = float(sys.argv[3])
    except:
        dt = 0.8
        eps_r = 1e-3
        omega = 1
    N = int(round(T / float(dt)))

    u_FE = FE_logistic(p, 0.1, dt, N)
    u_BE31, iter_BE31 = BE_logistic(p, 0.1, dt, N, "Picard1", eps_r, omega)
    u_BE3, iter_BE3 = BE_logistic(p, 0.1, dt, N, "Picard", eps_r, omega)
    u_BE4, iter_BE4 = BE_logistic(p, 0.1, dt, N, "Newton", eps_r, omega)
    u_CN = CN_logistic(p, 0.1, dt, N)

    print(f"Picard mean no of iterations (dt={dt:g}):", int(round(np.mean(iter_BE3))))
    print(f"Newton mean no of iterations (dt={dt:g}):", int(round(np.mean(iter_BE4))))

    t = np.linspace(0, dt * N, N + 1)
    plt.figure()
    plt.plot(t, u_FE, label="FE")
    plt.plot(t, u_BE3, label="BE Picard")
    plt.plot(t, u_BE31, label="BE Picard1")
    plt.plot(t, u_BE4, label="BE Newton")
    plt.plot(t, u_CN, label="CN gm")
    plt.legend(loc="lower right")
    plt.title(f"dt={dt:g}, eps={eps_r:.0E}")
    plt.xlabel("t")
    plt.ylabel("u")
    filestem = "logistic_N%d_eps%03d" % (N, np.log10(eps_r))
    plt.savefig(filestem + "_u.png")
    plt.savefig(filestem + "_u.pdf")

    plt.figure()
    plt.plot(range(1, len(iter_BE3) + 1), iter_BE3, "r-o", label="Picard")
    plt.plot(range(1, len(iter_BE4) + 1), iter_BE4, "b-o", label="Newton")
    plt.legend()
    plt.title(f"dt={dt:g}, eps={eps_r:.0E}")
    plt.axis([1, N + 1, 0, max(iter_BE3 + iter_BE4) + 1])
    plt.xlabel("Time level")
    plt.ylabel("No of iterations")
    plt.savefig(filestem + "_iter.png")
    plt.savefig(filestem + "_iter.pdf")
```

5.38. Problem: Experience the behavior of Newton's method

The program Newton_demo.py illustrates graphically each step in Newton's method and is run like

```
Terminal> python Newton_demo.py f dfdx x0 xmin xmax
```

Use this program to investigate potential problems with Newton's method when solving $e^{-0.5x^2}\cos(\pi x) = 0$. Try a starting point $x_0 = 0.8$ and $x_0 = 0.85$ and watch the different behavior. Just run

```
Terminal>  python Newton_demo.py '0.2 + exp(-0.5*x**2)*cos(pi*x)' \
           '-x*exp(-x**2)*cos(pi*x) - pi*exp(-x**2)*sin(pi*x)' \
           0.85 -3 3
```

and repeat with 0.85 replaced by 0.8.

5.39. Exercise: Compute the Jacobian of a 2×2 system

Write up the system (5.17)-(5.18) in the form $F(u) = 0$, $F = (F_0, F_1)$, $u = (u_0, u_1)$, and compute the Jacobian $J_{i,j} = \partial F_i/\partial u_j$.

5.40. Problem: Solve nonlinear equations arising from a vibration ODE

Consider a nonlinear vibration problem

$$mu'' + bu'|u'| + s(u) = F(t),$$

where $m > 0$ is a constant, $b \geq 0$ is a constant, $s(u)$ a possibly nonlinear function of u, and $F(t)$ is a prescribed function. Such models arise from Newton's second law of motion in mechanical vibration problems where $s(u)$ is a spring or restoring force, mu'' is mass times acceleration, and $bu'|u'|$ models water or air drag.

a)

Rewrite the equation for u as a system of two first-order ODEs, and discretize this system by a Crank-Nicolson (centered difference) method. With $v = u'$, we get a nonlinear term $v^{n+\frac{1}{2}}|v^{n+\frac{1}{2}}|$. Use a geometric average for $v^{n+\frac{1}{2}}$.

b)

Formulate a Picard iteration method to solve the system of nonlinear algebraic equations.

c)

Explain how to apply Newton's method to solve the nonlinear equations at each time level. Derive expressions for the Jacobian and the right-hand side in each Newton iteration.

5.41. Exercise: Find the truncation error of arithmetic mean of products

In Section Section 5.21 we introduce alternative arithmetic means of a product. Say the product is $P(t)Q(t)$ evaluated at $t = t_{n+\frac{1}{2}}$. The exact value is

$$[PQ]^{n+\frac{1}{2}} = P^{n+\frac{1}{2}}Q^{n+\frac{1}{2}}$$

There are two obvious candidates for evaluating $[PQ]^{n+\frac{1}{2}}$ as a mean of values of P and Q at t_n and t_{n+1}. Either we can take the arithmetic mean of each factor P and Q,

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^n + P^{n+1})\frac{1}{2}(Q^n + Q^{n+1}), \tag{5.73}$$

or we can take the arithmetic mean of the product PQ:

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^nQ^n + P^{n+1}Q^{n+1}). \tag{5.74}$$

The arithmetic average of $P(t_{n+\frac{1}{2}})$ is $\mathcal{O}(\Delta t^2)$:

$$P(t_{n+\frac{1}{2}}) = \frac{1}{2}(P^n + P^{n+1}) + \mathcal{O}(\Delta t^2).$$

A fundamental question is whether (5.73) and (5.74) have different orders of accuracy in $\Delta t = t_{n+1} - t_n$. To investigate this question, expand quantities at t_{n+1} and t_n in Taylor series around $t_{n+\frac{1}{2}}$, and subtract the true value $[PQ]^{n+\frac{1}{2}}$ from the approximations (5.73) and (5.74) to see what the order of the error terms are.

You may explore `sympy` for carrying out the tedious calculations.

A general Taylor series expansion of $P(t + \frac{1}{2}\Delta t)$ around t involving just a general function $P(t)$ can be created as follows:

```
>>> from sympy import *
>>> t, dt = symbols('t dt')
>>> P = symbols('P', cls=Function)
>>> P(t).series(t, 0, 4)
P(0) + t*Subs(Derivative(P(_x), _x), (_x,), (0,)) +
t**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/2 +
t**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/6 + O(t**4)
>>> P_p = P(t).series(t, 0, 4).subs(t, dt/2)
>>> P_p
P(0) + dt*Subs(Derivative(P(_x), _x), (_x,), (0,))/2 +
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 +
dt**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/48 + O(dt**4)
```

The error of the arithmetic mean, $\frac{1}{2}(P(-\frac{1}{2}\Delta t) + P(-\frac{1}{2}\Delta t))$ for $t = 0$ is then

```
>>> P_m = P(t).series(t, 0, 4).subs(t, -dt/2)
>>> mean = Rational(1,2)*(P_m + P_p)
>>> error = simplify(expand(mean) - P(0))
>>> error
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 + O(dt**4)
```

Use these examples to investigate the error of (5.73) and (5.74) for $n = 0$. (Choosing $n = 0$ is necessary for not making the expressions too complicated for `sympy`, but there is of course no lack of generality by using $n = 0$ rather than an arbitrary n - the main point is the product and addition of Taylor series.)

5.42. Problem: Newton's method for linear problems

Suppose we have a linear system $F(u) = Au - b = 0$. Apply Newton's method to this system, and show that the method converges in one iteration.

5.43. Problem: Discretize a 1D problem with a nonlinear coefficient

We consider the problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0\,. \tag{5.75}$$

Discretize (5.75) by a centered finite difference method on a uniform mesh.

5.44. Problem: Linearize a 1D problem with a nonlinear coefficient

We have a two-point boundary value problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0\,. \tag{5.76}$$

a)

Construct a Picard iteration method for (5.76) without discretizing in space.

b)

Apply Newton's method to (5.76) without discretizing in space.

c)

Discretize (5.76) by a centered finite difference scheme. Construct a Picard method for the resulting system of nonlinear algebraic equations.

d)

Discretize (5.76) by a centered finite difference scheme. Define the system of nonlinear algebraic equations, calculate the Jacobian, and set up Newton's method for solving the system.

5.45. Problem: Finite differences for the 1D Bratu problem

5.46. Good: http://faculty.oxy.edu/ron/research/bratu/bratu.pdf

5.47. It has a collocation method too

We address the so-called Bratu problem

$$u'' + \lambda e^u = 0, \quad x \in (0,1), \quad u(0) = u(1) = 0, \tag{5.77}$$

where λ is a given parameter and u is a function of x. This is a widely used model problem for studying numerical methods for nonlinear differential equations. The problem (5.77) has an exact solution

$$u_{\mathrm{e}}(x) = -2\ln\left(\frac{\cosh((x-\frac{1}{2})\theta/2)}{\cosh(\theta/4)}\right),$$

where θ solves

$$\theta = \sqrt{2\lambda}\cosh(\theta/4)\,.$$

There are two solutions of (5.77) for $0 < \lambda < \lambda_c$ and no solution for $\lambda > \lambda_c$. For $\lambda = \lambda_c$ there is one unique solution. The critical value λ_c solves

$$1 = \sqrt{2\lambda_c}\frac{1}{4}\sinh(\theta(\lambda_c)/4)\,.$$

A numerical value is $\lambda_c = 3.513830719$.

a)

Discretize (5.77) by a centered finite difference method.

b)

Set up the nonlinear equations $F_i(u_0, u_1, \ldots, u_{N_x}) = 0$ from a). Calculate the associated Jacobian.

c)

Implement a solver that can compute $u(x)$ using Newton's method. Plot the error as a function of x in each iteration.

d)

Investigate whether Newton's method gives second-order convergence by computing $||u_{\mathrm{e}} - u||/||u_{\mathrm{e}} - u^-||^2$ in each iteration, where u is solution in the current iteration and u^- is the solution in the previous iteration.

5.48. Problem: Discretize a nonlinear 1D heat conduction PDE by finite differences

We address the 1D heat conduction PDE

$$\varrho c(T)T_t = (k(T)T_x)_x,$$

for $x \in [0, L]$, where ϱ is the density of the solid material, $c(T)$ is the heat capacity, T is the temperature, and $k(T)$ is the heat conduction coefficient. $T(x, 0) = I(x)$, and ends are subject to a cooling law:

$$k(T)T_x|_{x=0} = h(T)(T - T_s), \quad -k(T)T_x|_{x=L} = h(T)(T - T_s),$$

where $h(T)$ is a heat transfer coefficient and T_s is the given surrounding temperature.

a)

Discretize this PDE in time using either a Backward Euler or Crank-Nicolson scheme.

b)

Formulate a Picard iteration method for the time-discrete problem (i.e., an iteration method before discretizing in space).

c)

Formulate a Newton method for the time-discrete problem in b).

d)

Discretize the PDE by a finite difference method in space. Derive the matrix and right-hand side of a Picard iteration method applied to the space-time discretized PDE.

e)

Derive the matrix and right-hand side of a Newton method applied to the discretized PDE in d).

5.49. Problem: Differentiate a highly nonlinear term

The operator $\nabla \cdot (\alpha(u)\nabla u)$ with $\alpha(u) = |\nabla u|^q$ appears in several physical problems, especially flow of Non-Newtonian fluids. The expression $|\nabla u|$ is defined as the Euclidean norm of a vector: $|\nabla u|^2 = \nabla u \cdot \nabla u$. In a Newton method one has to carry out the differentiation $\partial\alpha(u)/\partial c_j$, for $u = \sum_k c_k\psi_k$. Show that

$$\frac{\partial}{\partial u_j}|\nabla u|^q = q|\nabla u|^{q-2}\nabla u \cdot \nabla\psi_j\,.$$

Solution

$$\frac{\partial}{\partial c_j}|\nabla u|^q = \frac{\partial}{\partial c_j}(\nabla u\cdot\nabla u)^{\frac{q}{2}} = \frac{q}{2}(\nabla u\cdot\nabla u)^{\frac{q}{2}-1}\frac{\partial}{\partial c_j}(\nabla u\cdot\nabla u)$$
$$= \frac{q}{2}|\nabla u|^{q-2}(\frac{\partial}{\partial c_j}(\nabla u)\cdot\nabla u + \nabla u\cdot\frac{\partial}{\partial c_j}(\nabla u))$$
$$= q|\nabla u|^{q-2}(\nabla u\cdot\nabla\frac{\partial u}{\partial c_j}) = q|\nabla u|^{q-2}(\nabla u\cdot\nabla\psi_j)$$

5.50. Exercise: Crank-Nicolson for a nonlinear 3D diffusion equation

Redo Section Section 5.26 when a Crank-Nicolson scheme is used to discretize the equations in time and the problem is formulated for three spatial dimensions.

Express the Jacobian as $J_{i,j,k,r,s,t} = \partial F_{i,j,k}/\partial u_{r,s,t}$ and observe, as in the 2D case, that $J_{i,j,k,r,s,t}$ is very sparse:

$J_{i,j,k,r,s,t} \neq 0$ only for $r = i \pm i$, $s = j \pm 1$, and $t = k \pm 1$ as well as $r = i$, $s = j$, and $t = k$.

5.51. Problem: Find the sparsity of the Jacobian

Consider a typical nonlinear Laplace term like $\nabla\cdot\alpha(u)\nabla u$ discretized by centered finite differences. Explain why the Jacobian corresponding to this term has the same sparsity pattern as the matrix associated with the corresponding linear term $\alpha\nabla^2 u$.

Set up the unknowns that enter the difference equation at a

point (i, j) in 2D or (i, j, k) in 3D, and identify the nonzero entries of the Jacobian that can arise from such a type of difference equation.

5.52. Problem: Investigate a 1D problem with a continuation method

Flow of a pseudo-plastic power-law fluid between two flat plates can be modeled by

$$\frac{d}{dx}\left(\mu_0\left|\frac{du}{dx}\right|^{n-1}\frac{du}{dx}\right) = -\beta, \quad u'(0) = 0,\ u(H) = 0,$$

where $\beta > 0$ and $\mu_0 > 0$ are constants. A target value of n may be $n = 0.2$.

a)

Formulate a Picard iteration method directly for the differential equation problem.

b)

Perform a finite difference discretization of the problem in each Picard iteration. Implement a solver that can compute u on a mesh. Verify that the solver gives an exact solution for $n = 1$ on a uniform mesh regardless of the cell size.

c)

Given a sequence of decreasing n values, solve the problem for each n using the solution for the previous n as initial guess for the Picard iteration. This is called a continuation method. Experiment with $n = (1, 0.6, 0.2)$ and $n = (1, 0.9, 0.8, \ldots, 0.2)$ and make a table of the number of Picard iterations versus n.

d)

Derive a Newton method at the differential equation level and discretize the resulting linear equations in each Newton iteration with the finite difference method.

e)

Investigate if Newton's method has better convergence properties than Picard iteration, both in combination with a continuation method.

5.53. Exercises: Nonlinear PDEs with Devito

These exercises explore nonlinear PDEs using Devito's symbolic finite difference framework.

5.53.1. Exercise 1: Nonlinear Diffusion Stability

The explicit scheme for nonlinear diffusion requires $F \leq 0.5$ where $F = D_{\max}\Delta t/\Delta x^2$.

a) Use `solve_nonlinear_diffusion_explicit` with $F = 0.4$ and verify stability.
b) Observe the solution behavior as F approaches 0.5.
c) Compare the decay rate for constant $D(u) = 1$ vs linear $D(u) = 1 + u$.

Solution

```
from src.nonlin import (
    solve_nonlinear_diffusion_explicit,
    constant_diffusion,
    linear_diffusion,
)
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(np.pi * x)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Constant diffusion
result_const = solve_nonlinear_diffusion_explicit(
    L=1.0, Nx=50, T=0.2, F=0.4, I=I,
    D_func=lambda u: constant_diffusion(u, D0=1.0),
    save_history=True,
)

# Linear diffusion
result_linear = solve_nonlinear_diffusion_explicit(
    L=1.0, Nx=50, T=0.2, F=0.4, I=I,
    D_func=lambda u: linear_diffusion(u, D0=1.0, alpha=0.5),
    save_history=True,
)

# Plot
for ax, result, title in [
    (axes[0], result_const, 'Constant D(u) = 1'),
    (axes[1], result_linear, 'Linear D(u) = 1 + 0.5u')
]:
    for i in range(0, len(result.t_history), len(result.t_history)//5):
        ax.plot(result.x, result.u_history[i],
                label=f't = {result.t_history[i]:.3f}')
    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(title)
    ax.legend()

plt.tight_layout()

# The linear diffusion case diffuses faster because D increases with u
print(f"Constant D: final max = {result_const.u.max():.4f}")
print(f"Linear D: final max = {result_linear.u.max():.4f}")
```

5.53.2. Exercise 2: Porous Medium Equation

The porous medium equation has $D(u) = mu^{m-1}$, giving:

$$u_t = \nabla \cdot (mu^{m-1}\nabla u) = \nabla \cdot \nabla(u^m)$$

a) Simulate with $m = 2$ (nonlinear diffusion).
b) Compare with $m = 1$ (linear diffusion).
c) Observe the "finite speed of propagation" for $m > 1$.

Solution

```
from src.nonlin import solve_nonlinear_diffusion_explicit, porous_medium_diffusion
import numpy as np
import matplotlib.pyplot as plt

# Compactly supported initial condition
def I(x):
    return np.maximum(0, 1 - 4*(x - 0.5)**2)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for ax, m, title in [
    (axes[0], 1.0, 'm = 1 (linear)'),
    (axes[1], 2.0, 'm = 2 (porous medium)')
]:
    result = solve_nonlinear_diffusion_explicit(
        L=1.0, Nx=100, T=0.1, F=0.3, I=I,
        D_func=lambda u, m=m: porous_medium_diffusion(u, m=m, D0=1.0),
        save_history=True,
    )

    for i in range(0, len(result.t_history), max(1, len(result.t_history)//5)):
        ax.plot(result.x, result.u_history[i],
                label=f't = {result.t_history[i]:.3f}')
    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(title)
    ax.legend()

plt.tight_layout()
```

For $m > 1$, the solution maintains compact support (finite speed of propagation), unlike linear diffusion which spreads instantly.

5.53.3. Exercise 3: Fisher-KPP Equation

The Fisher-KPP equation $u_t = Du_{xx} + ru(1-u)$ models population dynamics with logistic growth.

a) Simulate with a localized initial condition.
b) Observe the traveling wave behavior.
c) Measure the wave speed and compare with theory: $c = 2\sqrt{Dr}$.

Solution

```python
from src.nonlin import solve_reaction_diffusion_splitting, fisher_reaction
import numpy as np
import matplotlib.pyplot as plt

# Initial condition: localized population
def I(x):
    return np.where(x < 0.2, 1.0, 0.0)

D = 0.1
r = 1.0

result = solve_reaction_diffusion_splitting(
    L=5.0, a=D, Nx=200, T=5.0, F=0.3, I=I,
    R_func=lambda u: fisher_reaction(u, r=r),
    splitting="strang",
    save_history=True,
)

# Plot traveling wave
plt.figure(figsize=(10, 6))
for i in range(0, len(result.t_history), len(result.t_history)//10):
    plt.plot(result.x, result.u_history[i],
             label=f't = {result.t_history[i]:.1f}')
plt.xlabel('x')
plt.ylabel('u')
plt.title('Fisher-KPP Traveling Wave')
plt.legend()

# Theoretical wave speed
c_theory = 2 * np.sqrt(D * r)
print(f"Theoretical wave speed: {c_theory:.3f}")

# Estimate numerical wave speed from front position
threshold = 0.5
front_positions = []
for i, u in enumerate(result.u_history):
    idx = np.argmax(u < threshold)
    if idx > 0:
        front_positions.append((result.t_history[i], result.x[idx]))

if len(front_positions) > 2:
    t_vals = [p[0] for p in front_positions]
    x_vals = [p[1] for p in front_positions]
    c_numerical = np.polyfit(t_vals, x_vals, 1)[0]
    print(f"Numerical wave speed: {c_numerical:.3f}")
```

5.53.4. Exercise 4: Strang vs Lie Splitting

Compare the accuracy of Strang and Lie splitting.

a) Solve the reaction-diffusion equation with both methods.
b) Use a fine time step as reference solution.
c) Plot error vs time step size on a log-log scale.
d) Verify that Strang is second-order and Lie is first-order.

Solution

```
from src.nonlin import solve_reaction_diffusion_splitting, logistic_reaction
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return 0.5 * np.sin(np.pi * x)

# Reference solution with very fine time step
ref = solve_reaction_diffusion_splitting(
    L=1.0, a=0.1, Nx=100, T=0.1, F=0.1, I=I,
    R_func=lambda u: logistic_reaction(u, r=1.0, K=1.0),
    splitting="strang",
)

# Test different Fourier numbers (time step sizes)
F_values = [0.4, 0.3, 0.2, 0.1]
errors_lie = []
errors_strang = []

for F in F_values:
    for splitting, errors in [("lie", errors_lie), ("strang", errors_strang)]:
        result = solve_reaction_diffusion_splitting(
            L=1.0, a=0.1, Nx=100, T=0.1, F=F, I=I,
            R_func=lambda u: logistic_reaction(u, r=1.0, K=1.0),
            splitting=splitting,
        )
        error = np.max(np.abs(result.u - ref.u))
        errors.append(error)

# Plot
dt_values = [F * (1.0/100)**2 / 0.1 for F in F_values]
plt.figure(figsize=(8, 6))
plt.loglog(dt_values, errors_lie, 'bo-', label='Lie (O(dt))')
plt.loglog(dt_values, errors_strang, 'rs-', label='Strang (O(dt^2))')
plt.loglog(dt_values, [errors_lie[0]*(dt/dt_values[0]) for dt in dt_values],
           'b--', alpha=0.5)
plt.loglog(dt_values, [errors_strang[0]*(dt/dt_values[0])**2 for dt in dt_values],
           'r--', alpha=0.5)
plt.xlabel('Time step')
plt.ylabel('Error')
plt.legend()
plt.title('Splitting Method Comparison')
plt.grid(True)
```

Lie splitting shows first-order convergence ($O(\Delta t)$) while Strang splitting achieves second-order

($O(\Delta t^2)$).

5.53.5. Exercise 5: Burgers Shock Formation

Burgers' equation can develop steep gradients (shocks) for small viscosity.

a) Simulate with $\nu = 0.1, 0.01, 0.001$.
b) Observe the shock steepening for small ν.
c) Plot the maximum gradient vs time.

Solution

```
from src.nonlin import solve_burgers_equation
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(np.pi * x)

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

for col, nu in enumerate([0.1, 0.01, 0.001]):
    result = solve_burgers_equation(
        L=2.0, nu=nu, Nx=200, T=0.5, C=0.3, I=I,
        save_history=True,
    )

    # Plot solution at several times
    ax = axes[0, col]
    for i in range(0, len(result.t_history), max(1, len(result.t_history)//5)):
        ax.plot(result.x, result.u_history[i],
                label=f't = {result.t_history[i]:.2f}')
    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(f'nu = {nu}')
    ax.legend(fontsize=8)

    # Plot maximum gradient vs time
    ax = axes[1, col]
    max_grads = []
    for u in result.u_history:
        grad = np.abs(np.diff(u) / (result.x[1] - result.x[0]))
        max_grads.append(grad.max())
    ax.plot(result.t_history, max_grads)
    ax.set_xlabel('Time')
    ax.set_ylabel('Max |du/dx|')
    ax.set_title(f'Gradient evolution, nu = {nu}')

plt.tight_layout()
```

As viscosity decreases, the solution develops steeper gradients. For very small ν, the gradient can become large, approaching shock behavior.

5.53.6. Exercise 6: Allen-Cahn Equation

The Allen-Cahn equation $u_t = \epsilon^2 u_{xx} + u - u^3$ models phase transitions.

a) Start with random initial data in $[-1, 1]$.
b) Observe how the solution evolves toward ± 1.
c) Study the effect of ϵ on interface width.

Solution

```python
from src.nonlin import solve_reaction_diffusion_splitting, allen_cahn_reaction
import numpy as np
import matplotlib.pyplot as plt

# Random initial condition
np.random.seed(42)
x_init = np.linspace(0, 2.0, 101)
u_init = 0.2 * np.sin(3 * np.pi * x_init) + 0.1 * np.random.randn(101)

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for ax, epsilon in zip(axes, [0.1, 0.05, 0.02]):
    result = solve_reaction_diffusion_splitting(
        L=2.0, a=epsilon**2, Nx=100, T=1.0, F=0.3,
        I=lambda x, u_init=u_init: np.interp(x, x_init, u_init),
        R_func=lambda u: allen_cahn_reaction(u, epsilon=1.0),
        splitting="strang",
        save_history=True,
    )

    for i in range(0, len(result.t_history), max(1, len(result.t_history)//5)):
        ax.plot(result.x, result.u_history[i], alpha=0.7,
                label=f't = {result.t_history[i]:.2f}')
    ax.set_xlabel('x')
    ax.set_ylabel('u')
    ax.set_title(f'epsilon = {epsilon}')
    ax.axhline(1, color='k', linestyle='--', alpha=0.3)
    ax.axhline(-1, color='k', linestyle='--', alpha=0.3)
    ax.legend(fontsize=8)

plt.tight_layout()
```

The solution evolves toward ± 1 with sharp interfaces. Smaller ϵ gives sharper interfaces but requires finer resolution.

5.53.7. Exercise 7: Energy Decay in Nonlinear Diffusion

For nonlinear diffusion with homogeneous Dirichlet BCs, the "energy"

$$E(t) = \frac{1}{2}\int_0^L u^2\,dx$$

should decrease.

a) Compute $E(t)$ for nonlinear diffusion.
b) Verify monotonic decrease.
c) Compare decay rates for different $D(u)$.

Solution

```
from src.nonlin import (
    solve_nonlinear_diffusion_explicit,
    constant_diffusion,
    linear_diffusion,
)
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(np.pi * x)

plt.figure(figsize=(10, 6))

for D_func, label in [
    (lambda u: constant_diffusion(u, D0=1.0), 'Constant D=1'),
    (lambda u: linear_diffusion(u, D0=1.0, alpha=0.5), 'D=1+0.5u'),
    (lambda u: linear_diffusion(u, D0=1.0, alpha=1.0), 'D=1+u'),
]:
    result = solve_nonlinear_diffusion_explicit(
        L=1.0, Nx=100, T=0.5, F=0.4, I=I, D_func=D_func,
        save_history=True,
    )

    # Compute energy
    energies = []
    for u in result.u_history:
        E = 0.5 * np.trapz(u**2, result.x)
        energies.append(E)

    plt.semilogy(result.t_history, energies, label=label)

plt.xlabel('Time')
plt.ylabel('Energy E(t)')
plt.legend()
plt.title('Energy Decay in Nonlinear Diffusion')
plt.grid(True)

# Verify monotonic decrease
dE = np.diff(energies)
print(f"Energy monotonically decreasing: {np.all(dE <= 0)}")
```

The energy decreases monotonically. Nonlinear diffusion with $D(u)$ increasing with u can lead to faster decay.

5.53.8. Exercise 8: Convergence of Burgers Solver

Verify the spatial convergence of the Burgers equation solver.

a) Use grid sizes $N_x = 25, 50, 100, 200$.
b) Compare with a fine-grid reference solution.
c) Compute the observed convergence rate.

Solution

```python
from src.nonlin import solve_burgers_equation
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(np.pi * x)

# Reference solution
ref = solve_burgers_equation(
    L=2.0, nu=0.1, Nx=400, T=0.2, C=0.3, I=I,
)

grid_sizes = [25, 50, 100, 200]
errors = []

for Nx in grid_sizes:
    result = solve_burgers_equation(
        L=2.0, nu=0.1, Nx=Nx, T=0.2, C=0.3, I=I,
    )
    # Interpolate to reference grid for comparison
    u_interp = np.interp(ref.x, result.x, result.u)
    error = np.sqrt(np.mean((u_interp - ref.u)**2))
    errors.append(error)
    print(f"Nx = {Nx:3d}, error = {error:.4e}")

# Compute convergence rate
errors = np.array(errors)
dx = 2.0 / np.array(grid_sizes)
log_dx = np.log(dx)
log_err = np.log(errors)
rate = np.polyfit(log_dx, log_err, 1)[0]

print(f"\nObserved convergence rate: {rate:.2f}")

plt.figure(figsize=(8, 6))
plt.loglog(dx, errors, 'bo-', label=f'Observed (rate={rate:.2f})')
plt.loglog(dx, errors[0]*(dx/dx[0])**2, 'r--', label='O(dx^2)')
plt.xlabel('Grid spacing dx')
plt.ylabel('L2 error')
plt.legend()
plt.title('Convergence of Burgers Solver')
plt.grid(True)
```

5.53.9. Exercise 9: Picard Iteration Convergence

Study the convergence of Picard iteration for implicit nonlinear diffusion.

a) Track the number of iterations needed at each time step.
b) Study how the tolerance affects accuracy.
c) Compare with the explicit scheme for the same problem.

Solution

```
from src.nonlin import (
    solve_nonlinear_diffusion_picard,
    solve_nonlinear_diffusion_explicit,
)
import numpy as np
import matplotlib.pyplot as plt

def I(x):
    return np.sin(np.pi * x)

# Picard solver
result_picard = solve_nonlinear_diffusion_picard(
    L=1.0, Nx=50, T=0.05, dt=0.005,
    I=I,
)

# Explicit solver for comparison
result_explicit = solve_nonlinear_diffusion_explicit(
    L=1.0, Nx=50, T=0.05, F=0.4,
    I=I,
)

plt.figure(figsize=(10, 5))
plt.plot(result_picard.x, result_picard.u, 'b-', label='Picard (implicit)')
plt.plot(result_explicit.x, result_explicit.u, 'r--', label='Explicit')
plt.xlabel('x')
plt.ylabel('u')
plt.legend()
plt.title('Comparison: Picard vs Explicit')

diff = np.max(np.abs(result_picard.u - np.interp(result_picard.x,
                                                 result_explicit.x,
                                                 result_explicit.u)))
print(f"Maximum difference: {diff:.4e}")
```

The Picard method allows larger time steps but requires iteration. Both methods should give similar results for the same problem.

5.53.10. Exercise 10: Traveling Wave in Burgers

Study the traveling wave solution of the viscous Burgers equation.

a) Use initial condition $u(x,0) = -\tanh((x - L/2)/\delta)$ with boundary values $u(0) = 1$, $u(L) = -1$.
b) Observe the wave propagation.
c) Estimate the wave speed numerically.

Solution

```
from devito import Grid, TimeFunction, Eq, Operator, Constant
import numpy as np
import matplotlib.pyplot as plt

# Setup
L = 10.0
Nx = 200
nu = 0.1
T = 5.0
delta = 1.0  # Initial width

grid = Grid(shape=(Nx + 1,), extent=(L,))
x_dim = grid.dimensions[0]
t_dim = grid.stepping_dim

u = TimeFunction(name='u', grid=grid, time_order=1, space_order=2)
x_coords = np.linspace(0, L, Nx + 1)

# Initial condition: tanh profile
u.data[0, :] = -np.tanh((x_coords - L/2) / delta)
u.data[1, :] = u.data[0, :].copy()

dx = L / Nx
dt = 0.25 * min(0.5 * dx, 0.25 * dx**2 / nu)
Nt = int(T / dt)

dt_const = Constant(name='dt', value=np.float32(dt))
nu_const = Constant(name='nu', value=np.float32(nu))

u_plus = u.subs(x_dim, x_dim + x_dim.spacing)
u_minus = u.subs(x_dim, x_dim - x_dim.spacing)

advection = 0.25 * dt_const / dx * (u_plus**2 - u_minus**2)
diffusion = nu_const * dt_const / (dx**2) * (u_plus - 2*u + u_minus)
stencil = u - advection + diffusion

update = Eq(u.forward, stencil, subdomain=grid.interior)
bc_left = Eq(u[t_dim + 1, 0], 1.0)
bc_right = Eq(u[t_dim + 1, Nx], -1.0)

op = Operator([update, bc_left, bc_right])

# Run and save history
history = [u.data[0, :].copy()]
times = [0.0]

for n in range(Nt):
    op.apply(time_m=n, time_M=n, dt=np.float32(dt))
    if (n + 1) % (Nt // 10) == 0:
        history.append(u.data[(n+1) % 2, :].copy())
        times.append((n + 1) * dt)

# Plot
```

The wave propagates with a speed related to the average of the boundary values. For small viscosity, the wave develops a sharp front.

Part II.

Appendices

6. Formulas

6.1. Finite difference operator notation

$$
\begin{aligned}
u'(t_n) &\approx [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \\
u'(t_n) &\approx [D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} \\
u'(t_n) &= [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \\
u'(t_n) &\approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \\
u'(t_{n+\theta}) &= [\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} \\
u'(t_n) &\approx [D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} \\
u''(t_n) &\approx [D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
u(t_{n+\frac{1}{2}}) &\approx [\overline{u}^t]^{n+\frac{1}{2}} = \frac{1}{2}(u^{n+1} + u^n) \\
u(t_{n+\frac{1}{2}})^2 &\approx [\overline{u^2}^{t,g}]^{n+\frac{1}{2}} = u^{n+1}u^n \\
u(t_{n+\frac{1}{2}}) &\approx [\overline{u}^{t,h}]^{n+\frac{1}{2}} = \frac{2}{\frac{1}{u^{n+1}} + \frac{1}{u^n}} \\
u(t_{n+\theta}) &\approx [\overline{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1-\theta)u^n, \\
& \qquad t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_{n-1}
\end{aligned}
\tag{6.1}
$$

Some may wonder why θ is absent on the right-hand side of (6.1). The fraction is an approximation to the derivative at the point $t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n$.

6.2. Truncation errors of finite difference approximations

$$
\begin{aligned}
u'_{\mathrm{e}}(t_n) &= [D_t u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+\frac{1}{2}} - u_{\mathrm{e}}^{n-\frac{1}{2}}}{\Delta t} + R^n,\\
R^n &= -\frac{1}{24}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned}
$$

$$
\begin{aligned}
u'_{\mathrm{e}}(t_n) &= [D_{2t} u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n-1}}{2\Delta t} + R^n,\\
R^n &= -\frac{1}{6}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned}
$$

$$
\begin{aligned}
u'_{\mathrm{e}}(t_n) &= [D_t^- u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n} - u_{\mathrm{e}}^{n-1}}{\Delta t} + R^n,\\
R^n &= -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2)
\end{aligned}
$$

$$
\begin{aligned}
u'_{\mathrm{e}}(t_n) &= [D_t^+ u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n}}{\Delta t} + R^n,\\
R^n &= \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2)
\end{aligned}
$$

$$
\begin{aligned}
u'_{\mathrm{e}}(t_{n+\theta}) &= [\bar{D}_t u_{\mathrm{e}}]^{n+\theta} + R^{n+\theta} = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n}}{\Delta t} + R^{n+\theta},\\
R^{n+\theta} &= -\frac{1}{2}(1-2\theta)u''_{\mathrm{e}}(t_{n+\theta})\Delta t + \frac{1}{6}((1-\theta)^3 - \theta^3)u'''_{\mathrm{e}}(t_{n+\theta})\Delta t^2 +\\
&\quad \mathcal{O}(\Delta t^3)
\end{aligned}
$$

$$
\begin{aligned}
u'_{\mathrm{e}}(t_n) &= [D_t^{2-} u_{\mathrm{e}}]^n + R^n = \frac{3u_{\mathrm{e}}^{n} - 4u_{\mathrm{e}}^{n-1} + u_{\mathrm{e}}^{n-2}}{2\Delta t} + R^n,\\
R^n &= \frac{1}{3}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}
$$

$$
\begin{aligned}
u''_{\mathrm{e}}(t_n) &= [D_t D_t u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - 2u_{\mathrm{e}}^{n} + u_{\mathrm{e}}^{n-1}}{\Delta t^2} + R^n,\\
R^n &= -\frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned}
\tag{6.2}
$$

$$
\begin{aligned}
u_{\mathrm{e}}(t_{n+\theta}) &= [\overline{u_{\mathrm{e}}}^{t,\theta}]^{n+\theta} + R^{n+\theta} = \theta u_{\mathrm{e}}^{n+1} + (1-\theta)u_{\mathrm{e}}^{n} + R^{n+\theta},\\
R^{n+\theta} &= -\frac{1}{2}u''_{\mathrm{e}}(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3)\,.
\end{aligned}
\tag{6.3}
$$

6.2.1. Complex exponentials

Let $u^n = \exp(i\omega n\Delta t) = e^{i\omega t_n}$.

$$
[D_t D_t u]^n = u^n \frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right),
$$

$$
[D_t^+ u]^n = u^n \frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1),
$$

$$[D_t^- u]^n = u^n \frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)), \tag{6.4}$$

$$[D_t u]^n = u^n \frac{2}{\Delta t} i \sin\left(\frac{\omega\Delta t}{2}\right), \tag{6.5}$$

$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i \sin(\omega\Delta t)\,. \tag{6.6}$$

6.2.2. Real exponentials

Let $u^n = \exp(\omega n\Delta t) = e^{\omega t_n}$.

$$[D_t D_t u]^n = u^n \frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right),$$

$$[D_t^+ u]^n = u^n \frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1),$$

$$[D_t^- u]^n = u^n \frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)),$$

$$[D_t u]^n = u^n \frac{2}{\Delta t} i \sin\left(\frac{\omega\Delta t}{2}\right),$$

$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i \sin(\omega\Delta t)\,. \tag{6.7}$$

6.3. Finite difference formulas for powers of t

The following results are useful when checking if a polynomial term in a solution fulfills the discrete equation for the numerical method.

$$[D_t^+ t]^n = 1,$$
$$[D_t^- t]^n = 1,$$
$$[D_t t]^n = 1,$$
$$[D_{2t} t]^n = 1,$$
$$[D_t D_t t]^n = 0\,. \tag{6.8}$$

The next formulas concern the action of difference operators on a t^2 term.

$$[D_t^+ t^2]^n = (2n+1)\Delta t,$$
$$[D_t^- t^2]^n = (2n-1)\Delta t,$$
$$[D_t t^2]^n = 2n\Delta t,$$
$$[D_{2t} t^2]^n = 2n\Delta t,$$
$$[D_t D_t t^2]^n = 2, \tag{6.9}$$

Finally, we present formulas for a t^3 term:

$$
\begin{aligned}
[D_t^+ t^3]^n &= 3(n\Delta t)^2 + 3n\Delta t^2 + \Delta t^2, \\
[D_t^- t^3]^n &= 3(n\Delta t)^2 - 3n\Delta t^2 + \Delta t^2, \\
[D_t t^3]^n &= 3(n\Delta t)^2 + \frac{1}{4}\Delta t^2, \\
[D_{2t} t^3]^n &= 3(n\Delta t)^2 + \Delta t^2, \\
[D_t D_t t^3]^n &= 6n\Delta t,
\end{aligned} \tag{6.10}
$$

6.4. Software

Application of finite difference operators to polynomials and exponential functions, resulting in the formulas above, can easily be computed by some `sympy` code (from the file `src/formulas/lib.py` in this repository):

```
from sympy import *

t, dt, n, w = symbols("t dt n w", real=True)


def D_t_forward(u):
    return (u(t + dt) - u(t)) / dt


def D_t_backward(u):
    return (u(t) - u(t - dt)) / dt


def D_t_centered(u):
    return (u(t + dt / 2) - u(t - dt / 2)) / dt


def D_2t_centered(u):
    return (u(t + dt) - u(t - dt)) / (2 * dt)


def D_t_D_t(u):
    return (u(t + dt) - 2 * u(t) + u(t - dt)) / (dt**2)


op_list = [D_t_forward, D_t_backward, D_t_centered, D_2t_centered, D_t_D_t]


def ft1(t):
```

```
    return t


def ft2(t):
    return t**2


def ft3(t):
    return t**3


def f_expiwt(t):
    return exp(I * w * t)


def f_expwt(t):
    return exp(w * t)


func_list = [ft1, ft2, ft3, f_expiwt, f_expwt]
```

To see the results, one can now make a simple loop over the different types of functions and the various operators associated with them:

```
for func in func_list:
    for op in op_list:
        f = func
        e = op(f)
        e = simplify(expand(e))
        print e
        if func in [f_expiwt, f_expwt]:
            e = e/f(t)
        e = e.subs(t, n*dt)
        print expand(e)
        print factor(simplify(expand(e)))
```

7. Truncation Error Analysis

Truncation error analysis provides a widely applicable framework for analyzing the accuracy of finite difference schemes. This type of analysis can also be used for finite element and finite volume methods if the discrete equations are written in finite difference form. The result of the analysis is an asymptotic estimate of the error in the scheme on the form Ch^r, where h is a discretization parameter (Δt, Δx, etc.), r is a number, known as the convergence rate, and C is a constant, typically dependent on the derivatives of the exact solution.

Knowing r gives understanding of the accuracy of the scheme. But maybe even more important, a powerful verification method for computer codes is to check that the empirically observed convergence rates in experiments coincide with the theoretical value of r found from truncation error analysis.

The analysis can be carried out by hand, by symbolic software, and also numerically. All three methods will be illustrated. From examining the symbolic expressions of the truncation error we can add correction terms to the differential equations in order to increase the numerical accuracy.

In general, the term truncation error refers to the discrepancy that arises from performing a finite number of steps to approximate a process with infinitely many steps. The term is used in a number of contexts, including truncation of infinite series, finite precision arithmetic, finite differences, and differential equations. We shall be concerned with computing truncation errors arising in finite difference formulas and in finite difference discretizations of differential equations.

7.1. Abstract problem setting

Consider an abstract differential equation

$$\mathcal{L}(u) = 0,$$

where $\mathcal{L}(u)$ is some formula involving the unknown u and its derivatives. One example is $\mathcal{L}(u) = u'(t) + a(t)u(t) - b(t)$, where a and b are constants or functions of time. We can discretize the differential equation and obtain a corresponding discrete model, here written as

$$\mathcal{L}_\Delta(u) = 0\,.$$

The solution u of this equation is the *numerical solution*. To distinguish the numerical solution from the exact solution of the differential equation problem, we denote the latter by u_e and write the differential equation and its discrete counterpart as

$$\mathcal{L}(u_\mathrm{e}) = 0,$$
$$\mathcal{L}_\Delta(u) = 0\,.$$

Initial and/or boundary conditions can usually be left out of the truncation error analysis and are omitted in the following.

The numerical solution u is, in a finite difference method, computed at a collection of mesh points. The discrete equations represented by the abstract equation $\mathcal{L}_\Delta(u) = 0$ are usually algebraic equations involving u at some neighboring mesh points.

7.2. Error measures

A key issue is how accurate the numerical solution is. The ultimate way of addressing this issue would be to compute the error $u_\mathrm{e} - u$ at the mesh points. This is usually extremely demanding. In very simplified problem settings we may, however, manage to derive formulas for the numerical solution u, and therefore closed form expressions for the error $u_\mathrm{e} - u$. Such special cases can provide considerable insight regarding accuracy and stability, but the results are established for special problems.

The error $u_\mathrm{e} - u$ can be computed empirically in special cases where we know u_e. Such cases can be constructed by the method of manufactured solutions, where we choose some exact solution $u_\mathrm{e} = v$ and fit a source term f in the governing differential equation $\mathcal{L}(u_\mathrm{e}) = f$ such that $u_\mathrm{e} = v$ is a solution (i.e., $f = \mathcal{L}(v)$). Assuming an error model of the form Ch^r, where h is the discretization parameter, such as Δt or Δx, one can estimate the convergence rate r. This is a widely applicable procedure, but the validity of the results is, strictly speaking, tied to the chosen test problems.

Another error measure arises by asking to what extent the exact solution u_e fits the discrete equations. Clearly, u_e is in general not a solution of $\mathcal{L}_\Delta(u) = 0$, but we can define the residual

$$R = \mathcal{L}_\Delta(u_\mathrm{e}),$$

and investigate how close R is to zero. A small R means intuitively that the discrete equations are close to the differential equation, and then we are tempted to think that u^n must also be close to $u_\mathrm{e}(t_n)$.

The residual R is known as the truncation error of the finite difference scheme $\mathcal{L}_\Delta(u) = 0$. It appears that the truncation error is relatively straightforward to compute by hand or symbolic software *without specializing the differential equation and the discrete model to a special case*. The resulting R is found as a power series in the discretization parameters. The leading-order terms in the series provide an asymptotic measure of the accuracy of the numerical solution method (as the discretization parameters tend to zero). An advantage of truncation error analysis, compared to empirical estimation of convergence rates, or detailed analysis of a special problem with a mathematical expression for the numerical solution, is that the truncation error analysis reveals the accuracy of the various building blocks in the numerical method and how each building block impacts the overall accuracy. The analysis can therefore be used to detect building blocks with lower accuracy than the others.

Knowing the truncation error or other error measures is important for verification of programs by empirically establishing convergence rates. The forthcoming text will provide many examples on how to compute truncation errors for finite difference discretizations of ODEs and PDEs.

7.3. Truncation errors in finite difference formulas

The accuracy of a finite difference formula is a fundamental issue when discretizing differential equations. We shall first go through a particular example in detail and thereafter list the truncation error in the most common finite difference approximation formulas.

7.4. Example: The backward difference for $u'(t)$

Consider a backward finite difference approximation of the first-order derivative u':

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx u'(t_n)\,. \tag{7.1}$$

Here, u^n means the value of some function $u(t)$ at a point t_n, and $[D_t^- u]^n$ is the *discrete derivative* of $u(t)$ at $t = t_n$. The discrete derivative computed by a finite difference is, in general, not exactly equal to the derivative $u'(t_n)$. The error in the approximation is

$$R^n = [D_t^- u]^n - u'(t_n)\,. \tag{7.2}$$

The common way of calculating R^n is to

1. expand $u(t)$ in a Taylor series around the point where the derivative is evaluated, here t_n,
2. insert this Taylor series in (7.2), and
3. collect terms that cancel and simplify the expression.

The result is an expression for R^n in terms of a power series in Δt. The error R^n is commonly referred to as the *truncation error* of the finite difference formula.

The Taylor series formula often found in calculus books takes the form

$$f(x+h) = \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i f}{dx^i}(x)h^i\,.$$

In our application, we expand the Taylor series around the point where the finite difference formula approximates the derivative. The Taylor series of u^n at t_n is simply $u(t_n)$, while the Taylor series of u^{n-1} at t_n must employ the general formula,

$$\begin{aligned} u(t_{n-1}) = u(t - \Delta t) &= \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i u}{dt^i}(t_n)(-\Delta t)^i \\ &= u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3), \end{aligned}$$

where $\mathcal{O}(\Delta t^3)$ means a power-series in Δt where the lowest power is Δt^3. We assume that Δt is small such that $\Delta t^p \gg \Delta t^q$ if p is smaller than q. The details of higher-order terms in Δt are therefore not of much interest. Inserting the Taylor series above in the right-hand side of (7.2) gives rise to some algebra:

$$
\begin{aligned}
[D_t^- u]^n - u'(t_n) &= \frac{u(t_n) - u(t_{n-1})}{\Delta t} - u'(t_n) \\
&= \frac{u(t_n) - (u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3))}{\Delta t} - u'(t_n) \\
&= -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2)),
\end{aligned}
$$

which is, according to (7.2), the truncation error:

$$
R^n = -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2))\,.
$$

The dominating term for small Δt is $-\frac{1}{2}u''(t_n)\Delta t$, which is proportional to Δt, and we say that the truncation error is of *first order* in Δt.

7.5. Example: The forward difference for $u'(t)$

We can analyze the approximation error in the forward difference

$$
u'(t_n) \approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t},
$$

by writing

$$
R^n = [D_t^+ u]^n - u'(t_n),
$$

and expanding u^{n+1} in a Taylor series around t_n,

$$
u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.
$$

The result becomes

$$
R = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2),
$$

showing that also the forward difference is of first order.

7.6. Example: The central difference for $u'(t)$

For the central difference approximation,

$$
u'(t_n) \approx [D_t u]^n, \quad [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t},
$$

we write

$$
R^n = [D_t u]^n - u'(t_n),
$$

and expand $u(t_{n+\frac{1}{2}})$ and $u(t_{n-\frac{1}{2}}$ in Taylor series around the point t_n where the derivative is evaluated. We have

$$\begin{aligned}
u(t_{n+\frac{1}{2}}) =& u(t_n) + u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)(\frac{1}{2}\Delta t)^2 + \\
& \frac{1}{6}u'''(t_n)(\frac{1}{2}\Delta t)^3 + \frac{1}{24}u''''(t_n)(\frac{1}{2}\Delta t)^4 + \\
& \frac{1}{120}u'''''(t_n)(\frac{1}{2}\Delta t)^5 + \mathcal{O}(\Delta t^6), \\
u(t_{n-\frac{1}{2}}) =& u(t_n) - u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)(\frac{1}{2}\Delta t)^2 - \\
& \frac{1}{6}u'''(t_n)(\frac{1}{2}\Delta t)^3 + \frac{1}{24}u''''(t_n)(\frac{1}{2}\Delta t)^4 - \\
& \frac{1}{120}u'''''(t_n)(\frac{1}{2}\Delta t)^5 + \mathcal{O}(\Delta t^6)
\end{aligned}$$

.Now,

$$u(t_{n+\frac{1}{2}}) - u(t_{n-\frac{1}{2}}) = u'(t_n)\Delta t + \frac{1}{24}u'''(t_n)\Delta t^3 + \frac{1}{960}u'''''(t_n)\Delta t^5 + \mathcal{O}(\Delta t^7)\,.$$

By collecting terms in $[D_t u]^n - u'(t_n)$ we find the truncation error to be

$$R^n = \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

with only even powers of Δt. Since $R \sim \Delta t^2$ we say the centered difference is of *second order* in Δt.

7.7. Overview of leading-order error terms in finite difference formulas

Here we list the leading-order terms of the truncation errors associated with several common finite difference formulas for the first and second derivatives.

$$\begin{aligned}
[D_t u]^n &= \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} = u'(t_n) + R^n, \\
R^n &= \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned} \tag{7.3}$$

$$\begin{aligned}
[D_{2t} u]^n &= \frac{u^{n+1} - u^{n-1}}{2\Delta t} = u'(t_n) + R^n, \\
R^n &= \frac{1}{6}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned} \tag{7.4}$$

$$\begin{aligned}
[D_t^- u]^n &= \frac{u^n - u^{n-1}}{\Delta t} = u'(t_n) + R^n, \\
R^n &= -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2)
\end{aligned} \tag{7.5}$$

$$\begin{aligned}
[D_t^+ u]^n &= \frac{u^{n+1} - u^n}{\Delta t} = u'(t_n) + R^n, \\
R^n &= \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2)
\end{aligned} \tag{7.6}$$

$$
\begin{aligned}
[\bar{D}_t u]^{n+\theta} &= \frac{u^{n+1}-u^n}{\Delta t} = u'(t_{n+\theta}) + R^{n+\theta},\\
R^{n+\theta} &= \frac{1}{2}(1-2\theta)u''(t_{n+\theta})\Delta t - \frac{1}{6}((1-\theta)^3-\theta^3)u'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}
\tag{7.7}
$$

$$
\begin{aligned}
[D_t^{2-}u]^n &= \frac{3u^n - 4u^{n-1}+u^{n-2}}{2\Delta t} = u'(t_n) + R^n,\\
R^n &= -\frac{1}{3}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}
\tag{7.8}
$$

$$
\begin{aligned}
[D_t D_t u]^n &= \frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2} = u''(t_n) + R^n,\\
R^n &= \frac{1}{12}u''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)
\end{aligned}
\tag{7.9}
$$

It will also be convenient to have the truncation errors for various means or averages. The weighted arithmetic mean leads to

$$
\begin{aligned}
[\overline{u}^{t,\theta}]^{n+\theta} &= \theta u^{n+1} + (1-\theta)u^n = u(t_{n+\theta}) + R^{n+\theta},\\
R^{n+\theta} &= \frac{1}{2}u''(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3)\,.
\end{aligned}
\tag{7.10}
$$

The standard arithmetic mean follows from this formula when $\theta = \frac{1}{2}$. Expressed at point t_n we get

$$
\begin{aligned}
[\overline{u}^{t}]^{n} &= \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) = u(t_n) + R^n,\\
R^n &= \frac{1}{8}u''(t_n)\Delta t^2 + \frac{1}{384}u''''(t_n)\Delta t^4 + \mathcal{O}(\Delta t^6)\,.
\end{aligned}
\tag{7.11}
$$

The geometric mean also has an error $\mathcal{O}(\Delta t^2)$:

$$
\begin{aligned}
[\overline{u^2}^{t,g}]^n &= u^{n-\frac{1}{2}}u^{n+\frac{1}{2}} = (u^n)^2 + R^n,\\
R^n &= -\frac{1}{4}u'(t_n)^2\Delta t^2 + \frac{1}{4}u(t_n)u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.
\end{aligned}
\tag{7.12}
$$

The harmonic mean is also second-order accurate:

$$
\begin{aligned}
[\overline{u}^{t,h}]^n = u^n &= \frac{2}{\frac{1}{u^{n-\frac{1}{2}}} + \frac{1}{u^{n+\frac{1}{2}}}} + R^{n+\frac{1}{2}},\\
R^n &= -\frac{u'(t_n)^2}{4u(t_n)}\Delta t^2 + \frac{1}{8}u''(t_n)\Delta t^2\,.
\end{aligned}
\tag{7.13}
$$

7.8. Software for computing truncation errors

We can use `sympy` to aid calculations with Taylor series. The derivatives can be defined as symbols, say `D3f` for the 3rd derivative of some function f. A truncated Taylor series can then be written as `f + D1f*h + D2f*h**2/2`. The following class takes some symbol `f` for the function in question and makes a list of symbols for the derivatives. The `__call__` method computes the symbolic form of the series truncated at `num_terms` terms.

```
import sympy as sym

class TaylorSeries:
    """Class for symbolic Taylor series."""

    def __init__(self, f, num_terms=4):
        self.f = f
        self.N = num_terms
        self.df = [f]
        for i in range(1, self.N + 1):
            self.df.append(sym.Symbol("D%d%s" % (i, f.name)))

    def __call__(self, h):
        """Return the truncated Taylor series at x+h."""
        terms = self.f
        for i in range(1, self.N + 1):
            terms += sym.Rational(1, sym.factorial(i)) * self.df[i] * h**i
        return terms
```

We may, for example, use this class to compute the truncation error of the Forward Euler finite difference formula:

```
>>> from truncation_errors import TaylorSeries
>>> from sympy import *
>>> u, dt = symbols('u dt')
>>> u_Taylor = TaylorSeries(u, 4)
>>> u_Taylor(dt)
D1u*dt + D2u*dt**2/2 + D3u*dt**3/6 + D4u*dt**4/24 + u
>>> FE = (u_Taylor(dt) - u)/dt
>>> FE
(D1u*dt + D2u*dt**2/2 + D3u*dt**3/6 + D4u*dt**4/24)/dt
>>> simplify(FE)
D1u + D2u*dt/2 + D3u*dt**2/6 + D4u*dt**3/24
```

The truncation error consists of the terms after the first one (u').

The module file trunc/truncation_errors.py contains another class `DiffOp` with symbolic expressions for most of the truncation errors listed in the previous section. For example:

```
>>> from truncation_errors import DiffOp
>>> from sympy import *
>>> u = Symbol('u')
>>> diffop = DiffOp(u, independent_variable='t')
>>> diffop['geometric_mean']
-D1u**2*dt**2/4 - D1u*D3u*dt**4/48 + D2u**2*dt**4/64 + ...
>>> diffop['Dtm']
D1u + D2u*dt/2 + D3u*dt**2/6 + D4u*dt**3/24
>>> >>> diffop.operator_names()
['geometric_mean', 'harmonic_mean', 'Dtm', 'D2t', 'DtDt',
 'weighted_arithmetic_mean', 'Dtp', 'Dt']
```

The indexing of `diffop` applies names that correspond to the operators: `Dtp` for D_t^+, `Dtm` for D_t^-, `Dt` for D_t, `D2t` for D_{2t}, `DtDt` for D_tD_t.

7.9. Truncation errors in exponential decay ODE

We shall now compute the truncation error of a finite difference scheme for a differential equation. Our first problem involves the following linear ODE that models exponential decay,

$$u'(t) = -au(t)\,. \tag{7.14}$$

7.10. Forward Euler scheme

We begin with the Forward Euler scheme for discretizing (7.14):

$$[D_t^+ u = -au]^n\,. \tag{7.15}$$

The idea behind the truncation error computation is to insert the exact solution u_e of the differential equation problem (7.14) in the discrete equations (7.15) and find the residual that arises because u_e does not solve the discrete equations. Instead, u_e solves the discrete equations with a residual R^n:

$$[D_t^+ u_\mathrm{e} + au_\mathrm{e} = R]^n\,. \tag{7.16}$$

From 7.6 it follows that

$$[D_t^+ u_\mathrm{e}]^n = u_\mathrm{e}'(t_n) + \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t + \mathcal{O}(\Delta t^2),$$

which inserted in (7.16) results in

$$u_\mathrm{e}'(t_n) + \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t + \mathcal{O}(\Delta t^2) + au_\mathrm{e}(t_n) = R^n\,.$$

Now, $u_\mathrm{e}'(t_n) + au_\mathrm{e}^n = 0$ since u_e solves the differential equation. The remaining terms constitute the residual:

$$R^n = \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,. \tag{7.17}$$

This is the truncation error R^n of the Forward Euler scheme.

Because R^n is proportional to Δt, we say that the Forward Euler scheme is of first order in Δt. However, the truncation error is just one error measure, and it is not equal to the true error $u_{\mathrm{e}}^n - u^n$. For this simple model problem we can compute a range of different error measures for the Forward Euler scheme, including the true error $u_{\mathrm{e}}^n - u^n$, and all of them have dominating terms proportional to Δt.

7.11. Crank-Nicolson scheme

For the Crank-Nicolson scheme,

$$[D_t u = -au]^{n+\frac{1}{2}}, \tag{7.18}$$

we compute the truncation error by inserting the exact solution of the ODE and adding a residual R,

$$[D_t u_{\mathrm{e}} + a\overline{u_{\mathrm{e}}}^t = R]^{n+\frac{1}{2}}\,. \tag{7.19}$$

The term $[D_t u_{\mathrm{e}}]^{n+\frac{1}{2}}$ is easily computed from 7.9 by replacing n with $n+\frac{1}{2}$ in the formula,

$$[D_t u_{\mathrm{e}}]^{n+\frac{1}{2}} = u_{\mathrm{e}}'(t_{n+\frac{1}{2}}) + \frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

The arithmetic mean is related to $u(t_{n+\frac{1}{2}})$ by 7.11 so

$$[a\overline{u_{\mathrm{e}}}^t]^{n+\frac{1}{2}} = u_{\mathrm{e}}(t_{n+\frac{1}{2}}) + \frac{1}{8}u_{\mathrm{e}}''(t_n)\Delta t^2 + +\mathcal{O}(\Delta t^4)\,.$$

Inserting these expressions in (7.19) and observing that $u_{\mathrm{e}}'(t_{n+\frac{1}{2}}) + au_{\mathrm{e}}^{n+\frac{1}{2}} = 0$, because $u_{\mathrm{e}}(t)$ solves the ODE $u'(t) = -au(t)$ at any point t, we find that

$$R^{n+\frac{1}{2}} = \left(\frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}}) + \frac{1}{8}u_{\mathrm{e}}''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4)$$

Here, the truncation error is of second order because the leading term in R is proportional to Δt^2.

At this point it is wise to redo some of the computations above to establish the truncation error of the Backward Euler scheme, see Exercise Section 7.33.

7.12. The θ-rule

We may also compute the truncation error of the θ-rule,

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}\,.$$

Our computational task is to find $R^{n+\theta}$ in

$$[\bar{D}_t u_{\mathrm{e}} + a\overline{u_{\mathrm{e}}}^{t,\theta} = R]^{n+\theta}\,.$$

From 7.7 and 7.10 we get expressions for the terms with u_{e}. Using that $u_{\mathrm{e}}'(t_{n+\theta}) + au_{\mathrm{e}}(t_{n+\theta}) = 0$, we end up with

$$R^{n+\theta} =(\frac{1}{2}-\theta)u_{\mathrm{e}}''(t_{n+\theta})\Delta t + \frac{1}{2}\theta(1-\theta)u_{\mathrm{e}}''(t_{n+\theta})\Delta t^2 + \frac{1}{2}(\theta^2-\theta+3)u_{\mathrm{e}}'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{7.20}$$

For $\theta = \frac{1}{2}$ the first-order term vanishes and the scheme is of second order, while for $\theta \neq \frac{1}{2}$ we only have a first-order scheme.

7.13. Using symbolic software

The previously mentioned `truncation_error` module can be used to automate the Taylor series expansions and the process of collecting terms. Here is an example on possible use:

```
from truncation_error import DiffOp
from sympy import *

def decay():
    u, a = symbols('u a')
    diffop = DiffOp(u, independent_variable='t',
                    num_terms_Taylor_series=3)
    D1u = diffop.D(1)   # symbol for du/dt
    ODE = D1u + a*u     # define ODE

    FE = diffop['Dtp'] + a*u
    CN = diffop['Dt' ] + a*u
    BE = diffop['Dtm'] + a*u
    theta = diffop['barDt'] + a*diffop['weighted_arithmetic_mean']
    theta = sm.simplify(sm.expand(theta))
    R = {'FE': FE-ODE, 'BE': BE-ODE, 'CN': CN-ODE,
         'theta': theta-ODE}
    return R
```

The returned dictionary becomes

```
decay: {
 'BE': D2u*dt/2 + D3u*dt**2/6,
 'FE': -D2u*dt/2 + D3u*dt**2/6,
 'CN': D3u*dt**2/24,
 'theta': -D2u*a*dt**2*theta**2/2 + D2u*a*dt**2*theta/2 -
           D2u*dt*theta + D2u*dt/2 + D3u*a*dt**3*theta**3/3 -
           D3u*a*dt**3*theta**2/2 + D3u*a*dt**3*theta/6 +
           D3u*dt**2*theta**2/2 - D3u*dt**2*theta/2 + D3u*dt**2/6,
}
```

The results are in correspondence with our hand-derived expressions.

7.14. Empirical verification of the truncation error

The task of this section is to demonstrate how we can compute the truncation error R numerically. For example, the truncation error of the Forward Euler scheme applied to the decay ODE $u' = -ua$ is

$$R^n = [D_t^+ u_\mathrm{e} + a u_\mathrm{e}]^n \,. \tag{7.21}$$

If we happen to know the exact solution $u_\mathrm{e}(t)$, we can easily evaluate R^n from the above formula.

To estimate how R varies with the discretization parameter Δt, which has been our focus in the previous mathematical derivations, we first make the assumption that $R = C\Delta t^r$ for appropriate constants C and r and small enough Δt. The rate r can be estimated from a series of experiments where Δt is varied. Suppose we have m experiments $(\Delta t_i, R_i)$, $i = 0, \ldots, m-1$. For two consecutive experiments $(\Delta t_{i-1}, R_{i-1})$ and $(\Delta t_i, R_i)$, a corresponding r_{i-1} can be estimated by

$$r_{i-1} = \frac{\ln(R_{i-1}/R_i)}{\ln(\Delta t_{i-1}/\Delta t_i)}, \tag{7.22}$$

for $i = 1, \ldots, m-1$. Note that the truncation error R_i varies through the mesh, so (7.22) is to be applied pointwise. A complicating issue is that R_i and R_{i-1} refer to different meshes. Pointwise comparisons of the truncation error at a certain point in all meshes therefore requires any computed R to be restricted to the *coarsest mesh* and that all finer meshes contain all the points in the coarsest mesh. Suppose we have N_0 intervals in the coarsest mesh. Inserting a superscript n in (7.22), where n counts mesh points in the coarsest mesh, $n = 0, \ldots, N_0$, leads to the formula

$$r_{i-1}^n = \frac{\ln(R_{i-1}^n/R_i^n)}{\ln(\Delta t_{i-1}/\Delta t_i)} \,. \tag{7.23}$$

Experiments are most conveniently defined by N_0 and a number of refinements m. Suppose each mesh has twice as many cells N_i as the previous one:

$$N_i = 2^i N_0, \quad \Delta t_i = T N_i^{-1},$$

where $[0, T]$ is the total time interval for the computations. Suppose the computed R_i values on the mesh with N_i intervals are stored in an array `R[i]` (`R` being a list of arrays, one for each mesh). Restricting this R_i function to the coarsest mesh means extracting every N_i/N_0 point and is done as follows:

```
stride = N[i]/N_0
R[i] = R[i][::stride]
```

The quantity `R[i][n]` now corresponds to R_i^n.

In addition to estimating r for the pointwise values of $R = C\Delta t^r$, we may also consider an integrated quantity on mesh i,

$$R_{I,i} = \left(\Delta t_i \sum_{n=0}^{N_i} (R_i^n)^2 \right)^{\frac{1}{2}} \approx \int_0^T R_i(t) dt \,.$$

The sequence $R_{I,i}$, $i = 0, \ldots, m-1$, is also expected to behave as $C\Delta t^r$, with the same r as for the pointwise quantity R, as $\Delta t \to 0$.

The function below computes the R_i and $R_{I,i}$ quantities, plots them and compares with the theoretically derived truncation error (`R_a`) if available.

```
import numpy as np

def estimate(truncation_error, T, N_0, m, makeplot=True):
    """
    Compute the truncation error in a problem with one independent
    variable, using m meshes, and estimate the convergence
    rate of the truncation error.

    The user-supplied function truncation_error(dt, N) computes
    the truncation error on a uniform mesh with N intervals of
    length dt::

      R, t, R_a = truncation_error(dt, N)

    where R holds the truncation error at points in the array t,
    and R_a are the corresponding theoretical truncation error
    values (None if not available).

    The truncation_error function is run on a series of meshes
    with 2**i*N_0 intervals, i=0,1,...,m-1.
    The values of R and R_a are restricted to the coarsest mesh.
    and based on these data, the convergence rate of R (pointwise)
    and time-integrated R can be estimated empirically.
    """
    N = [2**i * N_0 for i in range(m)]

    R_I = np.zeros(m)  # time-integrated R values on various meshes
    R = [None] * m  # time series of R restricted to coarsest mesh
    R_a = [None] * m  # time series of R_a restricted to coarsest mesh
    dt = np.zeros(m)
    legends_R = []
    legends_R_a = []  # all legends of curves

    for i in range(m):
        dt[i] = T / float(N[i])
        R[i], t, R_a[i] = truncation_error(dt[i], N[i])

        R_I[i] = np.sqrt(dt[i] * np.sum(R[i] ** 2))

        if i == 0:
            t_coarse = t  # the coarsest mesh

        stride = N[i] / N_0
        R[i] = R[i][::stride]  # restrict to coarsest mesh
        R_a[i] = R_a[i][::stride]

        if makeplot:
```

```
            plt.figure(1)
            plt.plot(t_coarse, R[i])
            plt.yscale("log")
            legends_R.append("N=%d" % N[i])

            plt.figure(2)
            plt.plot(t_coarse, R_a[i] - R[i])
            plt.yscale("log")
            legends_R_a.append("N=%d" % N[i])

    if makeplot:
        plt.figure(1)
        plt.xlabel("time")
        plt.ylabel("pointwise truncation error")
        plt.legend(legends_R)
        plt.savefig("R_series.png")
        plt.savefig("R_series.pdf")
        plt.figure(2)
        plt.xlabel("time")
        plt.ylabel("pointwise error in estimated truncation error")
        plt.legend(legends_R_a)
        plt.savefig("R_error.png")
        plt.savefig("R_error.pdf")

    r_R_I = convergence_rates(dt, R_I)
    print("R integrated in time; r:", end=" ")
    print(" ".join(["%.1f" % r for r in r_R_I]))
    R = np.array(R)  # two-dim. numpy array
    r_R = [convergence_rates(dt, R[:, n])[-1] for n in range(len(t_coarse))]
```

The first `makeplot` block demonstrates how to build up two figures in parallel, using `plt.figure(i)` to create and switch to figure number `i`. Figure numbers start at 1. A logarithmic scale is used on the y axis since we expect that R as a function of time (or mesh points) is exponential. The reason is that the theoretical estimate (7.17) contains u_{e}'', which for the present model goes like e^{-at}. Taking the logarithm makes a straight line.

The code follows closely the previously stated mathematical formulas, but the statements for computing the convergence rates might deserve an explanation. The generic help function `convergence_rate(h, E)` computes and returns r_{i-1}, $i = 1, \ldots, m-1$ from (7.23), given Δt_i in `h` and R_i^n in `E`:

```
def convergence_rates(h, E):
    from math import log
    r = [log(E[i]/E[i-1])/log(h[i]/h[i-1])
         for i in range(1, len(h))]
    return r
```

Calling `r_R_I = convergence_rates(dt, R_I)` computes the sequence of rates $r_0, r_1, \ldots, r_{m-2}$ for the model $R_I \sim \Delta t^r$, while the statements

```
R = np.array(R)  # two-dim. numpy array
r_R = [convergence_rates(dt, R[:,n])[-1]
       for n in range(len(t_coarse))]
```

compute the final rate r_{m-2} for $R^n \sim \Delta t^r$ at each mesh point t_n in the coarsest mesh. This latter computation deserves more explanation. Since `R[i][n]` holds the estimated truncation error R_i^n on mesh i, at point t_n in the coarsest mesh, `R[:,n]` picks out the sequence R_i^n for $i = 0, \ldots, m-1$. The `convergence_rate` function computes the rates at t_n, and by indexing `[-1]` on the returned array from `convergence_rate`, we pick the rate r_{m-2}, which we believe is the best estimation since it is based on the two finest meshes.

The `estimate` function is available in a module `trunc_empir.py`. Let us apply this function to estimate the truncation error of the Forward Euler scheme. We need a function `decay_FE(dt, N)` that can compute (7.21) at the points in a mesh with time step `dt` and `N` intervals:

```
import numpy as np
import trunc_empir

def decay_FE(dt, N):
    dt = float(dt)
    t = np.linspace(0, N * dt, N + 1)
    u_e = I * np.exp(-a * t)  # exact solution, I and a are global
    u = u_e  # naming convention when writing up the scheme
    R = np.zeros(N)

    for n in range(0, N):
        R[n] = (u[n + 1] - u[n]) / dt + a * u[n]

    R_a = 0.5 * I * (-a) ** 2 * np.exp(-a * t) * dt

    return R, t[:-1], R_a[:-1]

if __name__ == "__main__":
    I = 1
    a = 2  # global variables needed in decay_FE
    trunc_empir.estimate(decay_FE, T=2.5, N_0=6, m=4, makeplot=True)
```

The estimated rates for the integrated truncation error R_I become 1.1, 1.0, and 1.0 for this sequence of four meshes. All the rates for R^n, computed as `r_R`, are also very close to 1 at all mesh points. The agreement between the theoretical formula (7.17) and the computed quantity (ref(7.21)) is very good, as illustrated in Figures Figure 7.1 and Figure 7.2. The program `trunc_decay_FE.py` was used to perform the simulations and it can easily be modified to test other schemes (see also Exercise Section 7.34).

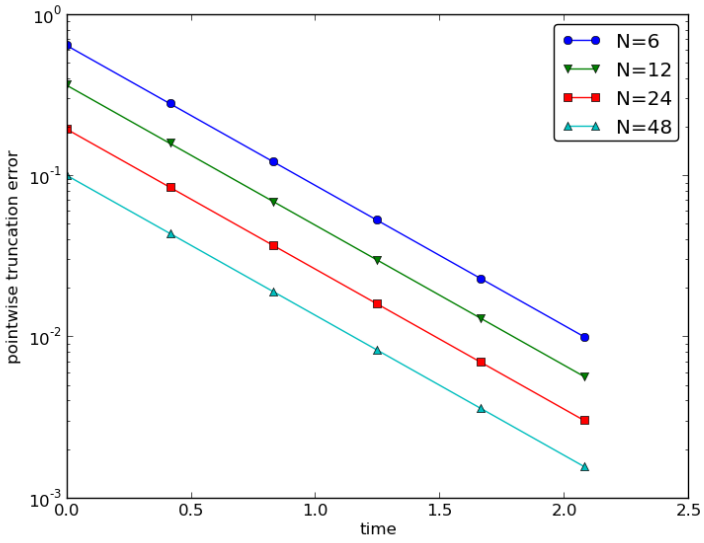


Figure 7.1.: Estimated truncation error at mesh points for different meshes.

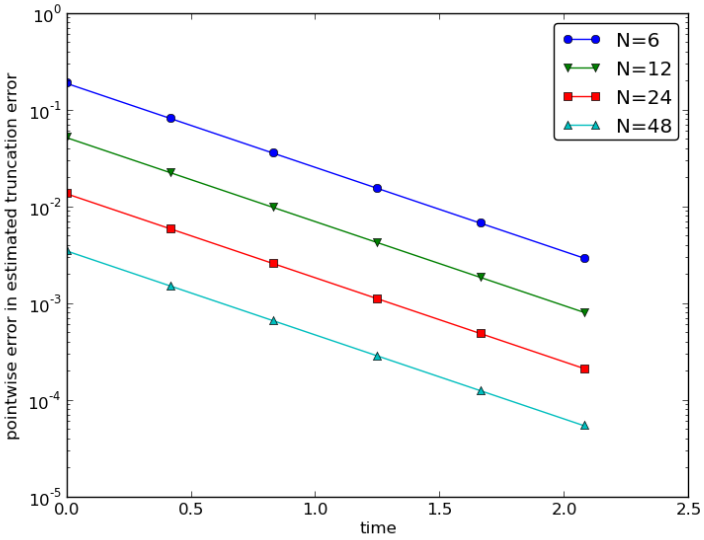


Figure 7.2.: Difference between theoretical and estimated truncation error at mesh points for different meshes.

7.15. Increasing the accuracy by adding correction terms

Now we ask the question: can we add terms in the differential equation that can help increase the order of the truncation error? To be precise, let us revisit the Forward Euler scheme for $u' = -au$, insert the exact solution u_e, include a residual R, but also include new terms C:

$$[D_t^+ u_\mathrm{e} + a u_\mathrm{e} = C + R]^n \,. \tag{7.24}$$

Inserting the Taylor expansions for $[D_t^+ u_\mathrm{e}]^n$ and keeping terms up to 3rd order in Δt gives the equation

$$\frac{1}{2}u_\mathrm{e}''(t_n)\Delta t - \frac{1}{6}u_\mathrm{e}'''(t_n)\Delta t^2 + \frac{1}{24}u_\mathrm{e}''''(t_n)\Delta t^3 + \mathcal{O}(\Delta t^4) = C^n + R^n \,.$$

Can we find C^n such that R^n is $\mathcal{O}(\Delta t^2)$? Yes, by setting

$$C^n = \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t,$$

we manage to cancel the first-order term and

$$R^n = \frac{1}{6}u_\mathrm{e}'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \,.$$

The correction term C^n introduces $\frac{1}{2}\Delta t u''$ in the discrete equation, and we have to get rid of the derivative u''. One idea is to approximate u'' by a second-order accurate finite difference formula, $u'' \approx (u^{n+1} - 2u^n + u^{n-1})/\Delta t^2$, but this introduces an additional time level with u^{n-1}. Another approach is to rewrite u'' in terms of u' or u using the ODE:

$$u' = -au \quad \Rightarrow \quad u'' = -au' = -a(-au) = a^2 u \,.$$

This means that we can simply set $C^n = \frac{1}{2}a^2\Delta t u^n$. We can then either solve the discrete equation

$$[D_t^+ u = -au + \frac{1}{2}a^2\Delta t u]^n, \tag{7.25}$$

or we can equivalently discretize the perturbed ODE

$$u' = -\hat{a}u, \quad \hat{a} = a(1 - \frac{1}{2}a\Delta t), \tag{7.26}$$

by a Forward Euler method. That is, we replace the original coefficient a by the perturbed coefficient $\hat{a}$. Observe that $\hat{a} \to a$ as $\Delta t \to 0$.

The Forward Euler method applied to (7.26) results in

$$[D_t^+ u = -a(1 - \frac{1}{2}a\Delta t)u]^n \,.$$

We can control our computations and verify that the truncation error of the scheme above is indeed $\mathcal{O}(\Delta t^2)$.

Another way of revealing the fact that the perturbed ODE leads to a more accurate solution is to look at the amplification factor. Our scheme can be written as

$$u^{n+1} = Au^n, \quad A = 1 - \hat{a}\Delta t = 1 - p + \frac{1}{2}p^2, \quad p = a\Delta t,$$

The amplification factor A as a function of $p = a\Delta t$ is seen to be the first three terms of the Taylor series for the exact amplification factor e^{-p}. The Forward Euler scheme for $u = -au$ gives only the first two terms $1 - p$ of the Taylor series for e^{-p}. That is, using $\hat{a}$ increases the order of the accuracy in the amplification factor.

Instead of replacing u'' by a^2u, we use the relation $u'' = -au'$ and add a term $-\frac{1}{2}a\Delta t u'$ in the ODE:

$$u' = -au - \frac{1}{2}a\Delta t u' \quad \Rightarrow \quad \left(1 + \frac{1}{2}a\Delta t\right) u' = -au\,.$$

Using a Forward Euler method results in

$$\left(1 + \frac{1}{2}a\Delta t\right) \frac{u^{n+1} - u^n}{\Delta t} = -au^n,$$

which after some algebra can be written as

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t} u^n\,.$$

This is the same formula as the one arising from a Crank-Nicolson scheme applied to $u' = -au$! It is now recommended to do Exercise Section 7.35 and repeat the above steps to see what kind of correction term is needed in the Backward Euler scheme to make it second order.

The Crank-Nicolson scheme is a bit more challenging to analyze, but the ideas and techniques are the same. The discrete equation reads

$$[D_t u = -au]^{n+\frac{1}{2}},$$

and the truncation error is defined through

$$[D_t u_{\mathrm{e}} + a\overline{u_{\mathrm{e}}}^t = C + R]^{n+\frac{1}{2}},$$

where we have added a correction term. We need to Taylor expand both the discrete derivative and the arithmetic mean with aid of 7.9 and 7.11, respectively. The result is

$$\frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4) + \frac{a}{8}u_{\mathrm{e}}''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4) = C^{n+\frac{1}{2}} + R^{n+\frac{1}{2}}\,.$$

The goal now is to make $C^{n+\frac{1}{2}}$ cancel the Δt^2 terms:

$$C^{n+\frac{1}{2}} = \frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}})\Delta t^2 + \frac{a}{8}u_{\mathrm{e}}''(t_n)\Delta t^2\,.$$

Using $u' = -au$, we have that $u'' = a^2u$, and we find that $u''' = -a^3u$. We can therefore solve the perturbed ODE problem

$$u' = -\hat{a}u, \quad \hat{a} = a(1 - \frac{1}{12}a^2\Delta t^2),$$

by the Crank-Nicolson scheme and obtain a method that is of fourth order in Δt. Exercise Section 7.36 encourages you to implement these correction terms and calculate empirical convergence rates to verify that higher-order accuracy is indeed obtained in real computations.

7.16. Extension to variable coefficients

Let us address the decay ODE with variable coefficients,

$$u'(t) = -a(t)u(t) + b(t),$$

discretized by the Forward Euler scheme,

$$[D_t^+ u = -au + b]^n .$$

The truncation error R is as always found by inserting the exact solution $u_{\mathrm{e}}(t)$ in the discrete scheme:

$$[D_t^+ u_{\mathrm{e}} + au_{\mathrm{e}} - b = R]^n .$$

Using 7.6,

$$u'_{\mathrm{e}}(t_n) - \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) + a(t_n)u_{\mathrm{e}}(t_n) - b(t_n) = R^n .$$

Because of the ODE,

$$u'_{\mathrm{e}}(t_n) + a(t_n)u_{\mathrm{e}}(t_n) - b(t_n) = 0,$$

we are left with the result

$$R^n = -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) . \tag{7.27}$$

We see that the variable coefficients do not pose any additional difficulties in this case. Exercise Section 7.37 takes the analysis above one step further to the Crank-Nicolson scheme.

7.17. Exact solutions of the finite difference equations

Having a mathematical expression for the numerical solution is very valuable in program verification, since we then know the exact numbers that the program should produce. Looking at the various formulas for the truncation errors in 7.9 and 7.13 in Section Section 7.7, we see that all but two of the R expressions contain a second or higher order derivative of u_{e}. The exceptions are the geometric and harmonic means where the truncation error involves u'_{e} and even u_{e} in case of the harmonic mean. So, apart from these two means, choosing u_{e} to be a linear function of t, $u_{\mathrm{e}} = ct + d$ for constants c and d, will make the truncation error vanish since $u''_{\mathrm{e}} = 0$. Consequently, the truncation error of a finite difference scheme will be zero since the various approximations used will all be exact. This means that the linear solution is an exact solution of the discrete equations.

In a particular differential equation problem, the reasoning above can be used to determine if we expect a linear u_{e} to fulfill the discrete equations. To actually prove that this is true, we can either compute the truncation error and see that it vanishes, or we can simply insert $u_{\mathrm{e}}(t) = ct + d$ in the scheme and see that it fulfills the equations. The latter method is usually the simplest. It will often be necessary to add some source term to the ODE in order to allow a linear solution.

Many ODEs are discretized by centered differences. From Section Section 7.7 we see that all the centered difference formulas have truncation errors involving u'''_{e} or higher-order derivatives. A quadratic solution, e.g., $u_{\mathrm{e}}(t) = t^2 + ct + d$, will then make the truncation errors vanish. This observation can be used to test if a quadratic solution will fulfill the discrete equations. Note that a quadratic solution will not obey the equations for a Crank-Nicolson scheme for $u' = -au + b$ because the approximation applies an arithmetic mean, which involves a truncation error with u''_{e}.

7.18. Computing truncation errors in nonlinear problems

The general nonlinear ODE

$$u' = f(u,t), \tag{7.28}$$

can be solved by a Crank-Nicolson scheme

$$[D_t u = \overline{f}^t]^{n+\frac{1}{2}}\,. \tag{7.29}$$

The truncation error is as always defined as the residual arising when inserting the exact solution u_e in the scheme:

$$[D_t u_\mathrm{e} - \overline{f}^t = R]^{n+\frac{1}{2}}\,. \tag{7.30}$$

Using 7.11 for $\overline{f}^t$ results in

$$\begin{aligned}[\overline{f}^t]^{n+\frac{1}{2}} &= \frac{1}{2}(f(u_\mathrm{e}^n, t_n) + f(u_\mathrm{e}^{n+1}, t_{n+1})) \\ &= f(u_\mathrm{e}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) + \frac{1}{8}u_\mathrm{e}''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4)\,.\end{aligned}$$

With 7.9 the discrete equations (7.30) lead to

$$u_\mathrm{e}'(t_{n+\frac{1}{2}}) + \frac{1}{24}u_\mathrm{e}'''(t_{n+\frac{1}{2}})\Delta t^2 + f(u_\mathrm{e}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) - \frac{1}{8}u_\mathrm{e}''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4) = R^{n+\frac{1}{2}}\,.$$

Since $u_\mathrm{e}'(t_{n+\frac{1}{2}}) - f(u_\mathrm{e}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) = 0$, the truncation error becomes

$$R^{n+\frac{1}{2}} = (\frac{1}{24}u_\mathrm{e}'''(t_{n+\frac{1}{2}}) + \frac{1}{8}u_\mathrm{e}''(t_{n+\frac{1}{2}}))\Delta t^2\,.$$

The computational techniques worked well even for this nonlinear ODE.

7.19. Linear model without damping

The next example on computing the truncation error involves the following ODE for vibration problems:

$$u''(t) + \omega^2 u(t) = 0\,. \tag{7.31}$$

Here, ω is a given constant.

7.19.1. The truncation error of a centered finite difference scheme

Using a standard, second-ordered, central difference for the second-order derivative in time, we have the scheme

$$[D_t D_t u + \omega^2 u = 0]^n\,. \tag{7.32}$$

Inserting the exact solution u_e in this equation and adding a residual R so that u_e can fulfill the equation results in

$$[D_t D_t u_\mathrm{e} + \omega^2 u_\mathrm{e} = R]^n\,.$$

To calculate the truncation error R^n, we use 7.4, i.e.,

$$[D_t D_t u_{\mathrm{e}}]^n = u''_{\mathrm{e}}(t_n) + \frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

and the fact that $u''_{\mathrm{e}}(t) + \omega^2 u_{\mathrm{e}}(t) = 0$. The result is

$$R^n = \frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

The truncation error of approximating $u'(0)$ The initial conditions for (7.31) are $u(0) = I$ and $u'(0) = V$. The latter involves a finite difference approximation. The standard choice

$$[D_{2t}u = V]^0,$$

where u^{-1} is eliminated with the aid of the discretized ODE for $n = 0$, involves a centered difference with an $\mathcal{O}(\Delta t^2)$ truncation error given by 7.3. The simpler choice

$$[D_t^+ u = V]^0,$$

is based on a forward difference with a truncation error $\mathcal{O}(\Delta t)$. A central question is if this initial error will impact the order of the scheme throughout the simulation. Exercise Section 7.40 asks you to perform an experiment to investigate this question.

7.19.2. Truncation error of the equation for the first step

We have shown that the truncation error of the difference used to approximate the initial condition $u'(0) = 0$ is $\mathcal{O}(\Delta t^2)$, but we can also investigate the difference equation used for the first step. In a truncation error setting, the right way to view this equation is not to use the initial condition $[D_{2t}u = V]^0$ to express $u^{-1} = u^1 - 2\Delta t V$ in order to eliminate u^{-1} from the discretized differential equation, but the other way around: the fundamental equation is the discretized initial condition $[D_{2t}u = V]^0$ and we use the discretized ODE $[D_t D_t + \omega^2 u = 0]^0$ to eliminate u^{-1} in the discretized initial condition. From $[D_t D_t + \omega^2 u = 0]^0$ we have

$$u^{-1} = 2u^0 - u^1 - \Delta t^2 \omega^2 u^0,$$

which inserted in $[D_{2t}u = V]^0$ gives

$$\frac{u^1 - u^0}{\Delta t} + \frac{1}{2}\omega^2 \Delta t u^0 = V\,. \tag{7.33}$$

The first term can be recognized as a forward difference such that the equation can be written in operator notation as

$$[D_t^+ u + \frac{1}{2}\omega^2 \Delta t u = V]^0\,.$$

The truncation error is defined as

$$[D_t^+ u_{\mathrm{e}} + \frac{1}{2}\omega^2 \Delta t u_{\mathrm{e}} - V = R]^0\,.$$

Using 7.6 with one more term in the Taylor series, we get that

$$u'_{\mathrm{e}}(0) + \frac{1}{2}u''_{\mathrm{e}}(0)\Delta t + \frac{1}{6}u'''_{\mathrm{e}}(0)\Delta t^2 + \mathcal{O}(\Delta t^3) + \frac{1}{2}\omega^2 \Delta t u_{\mathrm{e}}(0) - V = R^n\,.$$

Now, $u'_{\mathrm{e}}(0) = V$ and $u''_{\mathrm{e}}(0) = -\omega^2 u_{\mathrm{e}}(0)$ so we get

$$R^n = \frac{1}{6}u'''_{\mathrm{e}}(0)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.$$

There is another way of analyzing the discrete initial condition, because eliminating u^{-1} via the discretized ODE can be expressed as

$$[D_{2t}u + \Delta t(D_t D_t u - \omega^2 u) = V]^0\,. \tag{7.34}$$

Writing out (7.34) shows that the equation is equivalent to (7.33). The truncation error is defined by

$$[D_{2t}u_{\mathrm{e}} + \Delta t(D_t D_t u_{\mathrm{e}} - \omega^2 u_{\mathrm{e}}) = V + R]^0\,.$$

Replacing the difference via 7.3 and 7.4, as well as using $u'_{\mathrm{e}}(0) = V$ and $u''_{\mathrm{e}}(0) = -\omega^2 u_{\mathrm{e}}(0)$, gives

$$R^n = \frac{1}{6}u'''_{\mathrm{e}}(0)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.$$

Computing correction terms The idea of using correction terms to increase the order of R^n can be applied as described in Section Section 7.15. We look at

$$[D_t D_t u_{\mathrm{e}} + \omega^2 u_{\mathrm{e}} = C + R]^n,$$

and observe that C^n must be chosen to cancel the Δt^2 term in R^n. That is,

$$C^n = \frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2\,.$$

To get rid of the 4th-order derivative we can use the differential equation: $u'' = -\omega^2 u$, which implies $u'''' = \omega^4 u$. Adding the correction term to the ODE results in

$$u'' + \omega^2(1 - \frac{1}{12}\omega^2\Delta t^2)u = 0\,. \tag{7.35}$$

Solving this equation by the standard scheme

$$[D_t D_t u + \omega^2(1 - \frac{1}{12}\omega^2\Delta t^2)u = 0]^n,$$

will result in a scheme with truncation error $\mathcal{O}(\Delta t^4)$.

We can use another set of arguments to justify that (7.35) leads to a higher-order method. Mathematical analysis of the scheme (7.32) reveals that the numerical frequency $\tilde{\omega}$ is (approximately as $\Delta t \to 0$)

$$\tilde{\omega} = \omega(1 + \frac{1}{24}\omega^2\Delta t^2)\,.$$

One can therefore attempt to replace ω in the ODE by a slightly smaller ω since the numerics will make it larger:

$$[u'' + (\omega(1 - \frac{1}{24}\omega^2\Delta t^2))^2 u]^n = 0\,.$$

Expanding the squared term and omitting the higher-order term Δt^4 gives exactly the ODE (7.35). Experiments show that u^n is computed to 4th order in Δt. You can confirm this by running a little program in the `vib` directory:

```
from vib_undamped import convergence_rates, solver_adjust_w

r = convergence_rates(
      m=5, solver_function=solver_adjust_w, num_periods=8)
```

One will see that the rates `r` lie around 4.

7.20. Model with damping and nonlinearity

The model (7.31) can be extended to include damping $\beta u'$, a nonlinear restoring (spring) force $s(u)$, and some known excitation force $F(t)$:

$$mu'' + \beta u' + s(u) = F(t)\,. \tag{7.36}$$

The coefficient m usually represents the mass of the system. This governing equation can be discretized by centered differences:

$$[mD_tD_tu + \beta D_{2t}u + s(u) = F]^n\,.$$

The exact solution u_e fulfills the discrete equations with a residual term:

$$[mD_tD_tu_\mathrm{e} + \beta D_{2t}u_\mathrm{e} + s(u_\mathrm{e}) = F + R]^n\,.$$

Using 7.4 and 7.3 we get

$$[mD_tD_tu_\mathrm{e} + \beta D_{2t}u_\mathrm{e}]^n = mu_\mathrm{e}''(t_n) + \beta u_\mathrm{e}'(t_n) + \left(\frac{m}{12}u_\mathrm{e}''''(t_n) + \frac{\beta}{6}u_\mathrm{e}'''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4)$$

Combining this with the previous equation, we can collect the terms

$$mu_\mathrm{e}''(t_n) + \beta u_\mathrm{e}'(t_n) + \omega^2 u_\mathrm{e}(t_n) + s(u_\mathrm{e}(t_n)) - F^n,$$

and set this sum to zero because u_e solves the differential equation. We are left with the truncation error

$$R^n = \left(\frac{m}{12}u_\mathrm{e}''''(t_n) + \frac{\beta}{6}u_\mathrm{e}'''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4), \tag{7.37}$$

so the scheme is of second order.

According to (7.37), we can add correction terms

$$C^n = \left(\frac{m}{12}u_\mathrm{e}''''(t_n) + \frac{\beta}{6}u_\mathrm{e}'''(t_n)\right)\Delta t^2,$$

to the right-hand side of the ODE to obtain a fourth-order scheme. However, expressing u'''' and u''' in terms of lower-order derivatives is now harder because the differential equation is more complicated:

$$
\begin{aligned}
u''' &= \frac{1}{m}(F' - \beta u'' - s'(u)u'),\\
u'''' &= \frac{1}{m}(F'' - \beta u''' - s''(u)(u')^2 - s'(u)u''),\\
&= \frac{1}{m}(F'' - \beta\frac{1}{m}(F' - \beta u'' - s'(u)u') - s''(u)(u')^2 - s'(u)u'')\,.
\end{aligned}
$$

It is not impossible to discretize the resulting modified ODE, but it is up to debate whether correction terms are feasible and the way to go. Computing with a smaller Δt is usually always possible in these problems to achieve the desired accuracy.

7.21. Extension to quadratic damping

Instead of the linear damping term $\beta u'$ in (7.36) we now consider quadratic damping $\beta|u'|u'$:

$$mu'' + \beta|u'|u' + s(u) = F(t)\,. \tag{7.38}$$

A centered difference for u' gives rise to a nonlinearity, which can be linearized using a geometric mean: $[|u'|u']^n \approx |[u']^{n-\frac{1}{2}}|[u']^{n+\frac{1}{2}}$. The resulting scheme becomes

$$[mD_tD_tu]^n + \beta|[D_tu]^{n-\frac{1}{2}}|[D_tu]^{n+\frac{1}{2}} + s(u^n) = F^n\,.$$

The truncation error is defined through

$$[mD_tD_tu_{\mathrm{e}}]^n + \beta|[D_tu_{\mathrm{e}}]^{n-\frac{1}{2}}|[D_tu_{\mathrm{e}}]^{n+\frac{1}{2}} + s(u_{\mathrm{e}}^n) - F^n = R^n\,.$$

We start with expressing the truncation error of the geometric mean. According to 7.12,

$$
\begin{aligned}
|[D_tu_{\mathrm{e}}]^{n-\frac{1}{2}}|[D_tu_{\mathrm{e}}]^{n+\frac{1}{2}} = [|D_tu_{\mathrm{e}}|D_tu_{\mathrm{e}}]^n - \frac{1}{4}u_{\mathrm{e}}'(t_n)^2\Delta t^2 +\\
\frac{1}{4}u_{\mathrm{e}}(t_n)u_{\mathrm{e}}''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.
\end{aligned}
$$

Using 7.9 for the D_tu_{e} factors results in

$$[|D_tu_{\mathrm{e}}|D_tu_{\mathrm{e}}]^n = |u_{\mathrm{e}}' + \frac{1}{24}u_{\mathrm{e}}'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)|(u_{\mathrm{e}}' + \frac{1}{24}u_{\mathrm{e}}'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4))$$

We can remove the absolute value since it essentially gives a factor 1 or -1 only. Calculating the product, we have the leading-order terms

$$[D_tu_{\mathrm{e}}D_tu_{\mathrm{e}}]^n = (u_{\mathrm{e}}'(t_n))^2 + \frac{1}{12}u_{\mathrm{e}}(t_n)u_{\mathrm{e}}'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

With

$$m[D_tD_tu_{\mathrm{e}}]^n = mu_{\mathrm{e}}''(t_n) + \frac{m}{12}u_{\mathrm{e}}''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

and using the differential equation on the form $mu'' + \beta(u')^2 + s(u) = F$, we end up with

$$R^n = (\frac{m}{12}u_{\mathrm{e}}''''(t_n) + \frac{\beta}{12}u_{\mathrm{e}}(t_n)u_{\mathrm{e}}'''(t_n))\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

This result demonstrates that we have second-order accuracy also with quadratic damping. The key elements that lead to the second-order accuracy is that the difference approximations are $\mathcal{O}(\Delta t^2)$ *and* the geometric mean approximation is also $\mathcal{O}(\Delta t^2)$.

7.22. The general model formulated as first-order ODEs

The second-order model (7.38) can be formulated as a first-order system,

$$v' = \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right), \tag{7.39}$$

$$u' = v\,. \tag{7.40}$$

The system (7.39)-(7.40) can be solved either by a forward-backward scheme (the Euler-Cromer method) or a centered scheme on a staggered mesh.

7.22.1. The Euler-Cromer scheme

The discretization is based on the idea of stepping (7.39) forward in time and then using a backward difference in (7.40) with the recently computed v^{n+1}:

$$[D_t^+ v = \frac{1}{m}(F(t) - \beta|v|v - s(u))]^{n+1}\,. \tag{7.41}$$

$$[D_t^- u = v]^{n+1}, \tag{7.42}$$

For a truncation error analysis, we rewrite the system in vector-matrix form. Consider the linear case without damping,

$$v^{n+1} = v^n - \Delta t\omega^2 u^n, \quad u^{n+1} = u^n + \Delta t v^{n+1}\,.$$

We introduce the vector $w = (v, u)$ and write the system as

$$Aw^{n+1} = Bw^n, \quad A = \begin{bmatrix} 1 & 0 \\ 1 & -\Delta t \end{bmatrix}, \quad B = \begin{bmatrix} 1 & -\Delta t\omega^2 \\ 0 & 1 \end{bmatrix}.$$

The exact solution w_{e} satisfies

$$Aw_{\mathrm{e}}^{n+1} = Bw^n + \Delta t R^n,$$

where R^n is the residual, which has to be multiplied by Δt since we have already done that in the discrete equation.

The corresponding differential equation in w becomes

$$\frac{dw}{dt} = Cw, \quad C = \begin{bmatrix} 0 & -\omega^2 \\ 1 & 0 \end{bmatrix}.$$

We realize that $d^2w = C^2w$, and in general $d^mw = C^w$. Using these formulas to get rid of the derivatives in a Taylor expansion of w_{e}^{n+1} around t_n gives

$$A(w_{\mathrm{e}}^n + \Delta t C w_{\mathrm{e}}^n + \frac{1}{2}\Delta t^2 C^2 w_{\mathrm{e}}^n + \cdots) = Bw_{\mathrm{e}}^n + \Delta t R^n\,.$$

From this we get

$$R^n = \frac{1}{\Delta t}(A - B + \Delta t AC + \frac{1}{2}\Delta t^2 AC^2 + \cdots)w_{\mathrm{e}}^n$$
$$\sim (1)$$

This does not work out...

Each ODE will have a truncation error when inserting the exact solutions u_{e} and v_{e} in (7.41)-(7.42):

$$[D_t^+ u_{\mathrm{e}} = v_{\mathrm{e}} + R_u]^n, \tag{7.43}$$

$$[D_t^- v_{\mathrm{e}}]^{n+1} = \frac{1}{m}(F(t_{n+1}) - \beta|v_{\mathrm{e}}(t_n)|v_{\mathrm{e}}(t_{n+1}) - s(u_{\mathrm{e}}(t_{n+1}))) + R_v^{n+1}. \tag{7.44}$$

Application of 7.6 and 7.5 in (7.43) and (7.44), respectively, gives

$$u_{\mathrm{e}}'(t_n) + \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t + \mathcal{O}(\Delta t^2) = v_{\mathrm{e}}(t_n) + R_u^n, \tag{7.45}$$

$$v_{\mathrm{e}}'(t_{n+1}) - \frac{1}{2}v_{\mathrm{e}}''(t_{n+1})\Delta t + \mathcal{O}(\Delta t^2) = \frac{1}{m}(F(t_{n+1}) - \beta|v_{\mathrm{e}}(t_n)|v_{\mathrm{e}}(t_{n+1}) + s(u_{\mathrm{e}}(t_{n+1})) + R_v^n. \tag{7.46}$$

Since $u_{\mathrm{e}}' = v_{\mathrm{e}}$, (7.45) gives

$$R_u^n = \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t + \mathcal{O}(\Delta t^2).$$

In (7.46) we can collect the terms that constitute the ODE, but the damping term has the wrong form. Let us drop the absolute value in the damping term for simplicity. Adding a subtracting the right form $v^{n+1}v^{n+1}$ helps:

$$v_{\mathrm{e}}'(t_{n+1}) - \frac{1}{m}(F(t_{n+1}) - \beta v_{\mathrm{e}}(t_{n+1})v_{\mathrm{e}}(t_{n+1}) + s(u_{\mathrm{e}}(t_{n+1})) + (\beta v_{\mathrm{e}}(t_n)v_{\mathrm{e}}(t_{n+1}) - \beta v_{\mathrm{e}}(t_{n+1})v_{\mathrm{e}}(t_{n+1}))),$$

which reduces to

$$\frac{\beta}{m}v_{\mathrm{e}}(t_{n+1}(v_{\mathrm{e}}(t_n) - v_{\mathrm{e}}(t_{n+1})) = \frac{\beta}{m}v_{\mathrm{e}}(t_{n+1}[D_t^- v_{\mathrm{e}}]^{n+1}\Delta t$$
$$= \frac{\beta}{m}v_{\mathrm{e}}(t_{n+1}(v_{\mathrm{e}}'(t_{n+1})\Delta t + -\frac{1}{2}v_{\mathrm{e}}'''(t_{n+1})\Delta t^+ \mathcal{O}(\Delta t^3)).$$

We end with R_u^n and R_v^{n+1} as $\mathcal{O}(\Delta t)$, simply because all the building blocks in the schemes (the forward and backward differences and the linearization trick) are only first-order accurate. However, this analysis is misleading: the building blocks play together in a way that makes the scheme second-order accurate. This is shown by considering an alternative, yet equivalent, formulation of the above scheme.

7.22.2. A centered scheme on a staggered mesh

We now introduce a staggered mesh where we seek u at mesh points t_n and v at points $t_{n+\frac{1}{2}}$ in between the u points. The staggered mesh makes it easy to formulate centered differences in the system (7.39)-(7.40):

$$[D_t u = v]^{n-\frac{1}{2}},$$
$$[D_t v = \frac{1}{m}(F(t) - \beta|v|v - s(u))]^n\,. \tag{7.47}$$

The term $|v^n|v^n$ causes trouble since v^n is not computed, only $v^{n-\frac{1}{2}}$ and $v^{n+\frac{1}{2}}$. Using geometric mean, we can express $|v^n|v^n$ in terms of known quantities: $|v^n|v^n \approx |v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}}$. We then have

$$[D_t u]^{n-\frac{1}{2}} = v^{n-\frac{1}{2}},$$
$$[D_t v]^n = \frac{1}{m}(F(t_n) - \beta|v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}} - s(u^n))\,. \tag{7.48}$$

The truncation error in each equation fulfills

$$[D_t u_{\mathrm{e}}]^{n-\frac{1}{2}} = v_{\mathrm{e}}(t_{n-\frac{1}{2}}) + R_u^{n-\frac{1}{2}},$$
$$[D_t v_{\mathrm{e}}]^n = \frac{1}{m}(F(t_n) - \beta|v_{\mathrm{e}}(t_{n-\frac{1}{2}})|v_{\mathrm{e}}(t_{n+\frac{1}{2}}) - s(u^n)) + R_v^n\,.$$

The truncation error of the centered differences is given by 7.9, and the geometric mean approximation analysis can be taken from 7.12. These results lead to

$$u_{\mathrm{e}}'(t_{n-\frac{1}{2}}) + \frac{1}{24}u_{\mathrm{e}}'''(t_{n-\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4) = v_{\mathrm{e}}(t_{n-\frac{1}{2}}) + R_u^{n-\frac{1}{2}},$$

and

$$v_{\mathrm{e}}'(t_n) = \frac{1}{m}(F(t_n) - \beta|v_{\mathrm{e}}(t_n)|v_{\mathrm{e}}(t_n) + \mathcal{O}(\Delta t^2) - s(u^n)) + R_v^n\,.$$

The ODEs fulfilled by u_{e} and v_{e} are evident in these equations, and we achieve second-order accuracy for the truncation error in both equations:

$$R_u^{n-\frac{1}{2}} = \mathcal{O}(\Delta t^2), \quad R_v^n = \mathcal{O}(\Delta t^2)\,.$$

7.23. Linear wave equation in 1D

The standard, linear wave equation in 1D for a function $u(x,t)$ reads

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{7.49}$$

where c is the constant wave velocity of the physical medium in $[0,L]$. The equation can also be more compactly written as

$$u_{tt} = c^2 u_{xx} + f, \quad x \in (0,L),\ t \in (0,T], \tag{7.50}$$

Centered, second-order finite differences are a natural choice for discretizing the derivatives, leading to

$$[D_tD_tu = c^2D_xD_xu + f]_i^n\,. \tag{7.51}$$

Inserting the exact solution $u_{\mathrm{e}}(x,t)$ in (7.51) makes this function fulfill the equation if we add the term R:

$$[D_tD_tu_{\mathrm{e}} = c^2D_xD_xu_{\mathrm{e}} + f + R]_i^n \tag{7.52}$$

Our purpose is to calculate the truncation error R. From 7.4 we have that

$$[D_tD_tu_{\mathrm{e}}]_i^n = u_{\mathrm{e},tt}(x_i,t_n) + \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

when we use a notation taking into account that u_{e} is a function of two variables and that derivatives must be partial derivatives. The notation $u_{\mathrm{e},tt}$ means $\partial^2 u_{\mathrm{e}}/\partial t^2$.

The same formula may also be applied to the x-derivative term:

$$[D_xD_xu_{\mathrm{e}}]_i^n = u_{\mathrm{e},xx}(x_i,t_n) + \frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4),$$

Equation (7.52) now becomes

$$u_{\mathrm{e},tt} + \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 = c^2u_{\mathrm{e},xx} + c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + f(x_i,t_n) + \mathcal{O}(\Delta t^4,\Delta x^4) + R_i^n$$

.Because u_{e} fulfills the partial differential equation (PDE) (7.50), the first, third, and fifth term cancel out, and we are left with

$$R_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta t^4,\Delta x^4), \tag{7.53}$$

showing that the scheme (7.51) is of second order in the time and space mesh spacing.

7.24. Finding correction terms

Can we add correction terms to the PDE and increase the order of R_i^n in (7.53)? The starting point is

$$[D_tD_tu_{\mathrm{e}} = c^2D_xD_xu_{\mathrm{e}} + f + C + R]_i^n \tag{7.54}$$

From the previous analysis we simply get (7.53) again, but now with C:

$$R_i^n + C_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta t^4,\Delta x^4)\,. \tag{7.55}$$

The idea is to let C_i^n cancel the Δt^2 and Δx^2 terms to make $R_i^n = \mathcal{O}(\Delta t^4,\Delta x^4)$:

$$C_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2\,.$$

Essentially, it means that we add a new term

$$C = \frac{1}{12}\left(u_{tttt}\Delta t^2 - c^2 u_{xxxx}\Delta x^2\right),$$

to the right-hand side of the PDE. We must either discretize these 4th-order derivatives directly or rewrite them in terms of lower-order derivatives with the aid of the PDE. The latter approach is more feasible. From the PDE we have the operator equality

$$\frac{\partial^2}{\partial t^2} = c^2\frac{\partial^2}{\partial x^2},$$

so

$$u_{tttt} = c^2 u_{xxtt}, \quad u_{xxxx} = c^{-2}u_{ttxx}\,.$$

Assuming u is smooth enough, so that $u_{xxtt} = u_{ttxx}$, these relations lead to

$$C = \frac{1}{12}((c^2\Delta t^2 - \Delta x^2)u_{xx})_{tt}\,.$$

A natural discretization is

$$C_i^n = \frac{1}{12}((c^2\Delta t^2 - \Delta x^2)[D_xD_xD_tD_tu]_i^n\,.$$

Writing out $[D_xD_xD_tD_tu]_i^n$ as $[D_xD_x(D_tD_tu)]_i^n$ gives

$$\frac{1}{\Delta t^2}\Bigg(\frac{u^{n+1}**i+1-2u^n**i+1+u_{i+1}^{n-1}}{\Delta x^2} - 2$$
$$\frac{u^{n+1}**i-2u^n**i+u_i^{n-1}}{\Delta x^2}+ \qquad \frac{u^{n+1}**i-1-2u^n**i-1+u_{i-1}^{n-1}}{\Delta x^2}\Bigg)$$

Now the unknown values $u^{n+1}**i+1$, $u^{n+1}**i$, and u_{i-1}^{n+1} are *coupled*, and we must solve a tridiagonal system to find them. This is in principle straightforward, but it results in an implicit finite difference scheme, while we had a convenient explicit scheme without the correction terms.

7.25. Extension to variable coefficients

Now we address the variable coefficient version of the linear 1D wave equation,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right),$$

or written more compactly as

$$u_{tt} = (\lambda u_x)_x\,. \tag{7.56}$$

The discrete counterpart to this equation, using arithmetic mean for λ and centered differences, reads

$$[D_tD_tu = D_x\overline{\lambda}^x D_xu]_i^n\,. \tag{7.57}$$

The truncation error is the residual R in the equation

$$[D_tD_tu_{\mathrm{e}} = D_x\overline{\lambda}^x D_xu_{\mathrm{e}} + R]_i^n\,. \tag{7.58}$$

The difficulty with (7.58) is how to compute the truncation error of the term $[D_x\overline{\lambda}^x D_x u_\mathrm{e}]_i^n$.

We start by writing out the outer operator:

$$[D_x\overline{\lambda}^x D_x u_\mathrm{e}]_i^n = \frac{1}{\Delta x}\left([\overline{\lambda}^x D_x u_\mathrm{e}]_{i+\frac{1}{2}}^n - [\overline{\lambda}^x D_x u_\mathrm{e}]_{i-\frac{1}{2}}^n\right). \tag{7.59}$$

With the aid of 7.9 and 7.11 we have

$$\begin{aligned}
[D_x u_\mathrm{e}]_{i+\frac{1}{2}}^n &= u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \frac{1}{24}u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4),\\
[\overline{\lambda}^x]_{i+\frac{1}{2}} &= \lambda(x_{i+\frac{1}{2}}) + \frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4),\\
[\overline{\lambda}^x D_x u_\mathrm{e}]_{i+\frac{1}{2}}^n &= (\lambda(x_{i+\frac{1}{2}}) + \frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4))\times\\
&\quad (u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \frac{1}{24}u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4))\\
&= \lambda(x_{i+\frac{1}{2}})u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \lambda(x_{i+\frac{1}{2}})\frac{1}{24}u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 +\\
&\quad u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n)\frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4)\\
&= [\lambda u_{\mathrm{e},x}]_{i+\frac{1}{2}}^n + G_{i+\frac{1}{2}}^n\Delta x^2 + \mathcal{O}(\Delta x^4),
\end{aligned}$$

where we have introduced the short form

$$G_{i+\frac{1}{2}}^n = \frac{1}{24}u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\lambda(x_{i+\frac{1}{2}}) + u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n)\frac{1}{8}\lambda''(x_{i+\frac{1}{2}}).$$

Similarly, we find that

$$[\overline{\lambda}^x D_x u_\mathrm{e}]_{i-\frac{1}{2}}^n = [\lambda u_{\mathrm{e},x}]_{i-\frac{1}{2}}^n + G_{i-\frac{1}{2}}^n\Delta x^2 + \mathcal{O}(\Delta x^4).$$

Inserting these expressions in the outer operator (7.59) results in

$$\begin{aligned}
[D_x\overline{\lambda}^x D_x u_\mathrm{e}]_i^n &= \frac{1}{\Delta x}([\overline{\lambda}^x D_x u_\mathrm{e}]_{i+\frac{1}{2}}^n - [\overline{\lambda}^x D_x u_\mathrm{e}]_{i-\frac{1}{2}}^n)\\
&= \frac{1}{\Delta x}([\lambda u_{\mathrm{e},x}]_{i+\frac{1}{2}}^n + G_{i+\frac{1}{2}}^n\Delta x^2 - [\lambda u_{\mathrm{e},x}]_{i-\frac{1}{2}}^n - G_{i-\frac{1}{2}}^n\Delta x^2 + \mathcal{O}(\Delta x^4))\\
&= [D_x\lambda u_{\mathrm{e},x}]_i^n + [D_x G]_i^n\Delta x^2 + \mathcal{O}(\Delta x^4).
\end{aligned}$$

The reason for $\mathcal{O}(\Delta x^4)$ in the remainder is that there are coefficients in front of this term, say $H\Delta x^4$, and the subtraction and division by Δx results in $[D_x H]_i^n\Delta x^4$.

We can now use 7.9 to express the D_x operator in $[D_x\lambda u_{\mathrm{e},x}]_i^n$ as a derivative and a truncation error:

$$[D_x\lambda u_{\mathrm{e},x}]_i^n = \frac{\partial}{\partial x}\lambda(x_i)u_{\mathrm{e},x}(x_i, t_n) + \frac{1}{24}(\lambda u_{\mathrm{e},x})_{xxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4).$$

Expressions like $[D_x G]_i^n\Delta x^2$ can be treated in an identical way,

$$[D_x G]_i^n\Delta x^2 = G_x(x_i, t_n)\Delta x^2 + \frac{1}{24}G_{xxx}(x_i, t_n)\Delta x^4 + \mathcal{O}(\Delta x^4).$$

There will be a number of terms with the Δx^2 factor. We lump these now into $\mathcal{O}(\Delta x^2)$. The result of the truncation error analysis of the spatial derivative is therefore summarized as

$$[D_x\overline{\lambda}^x D_x u_{\mathrm{e}}]_i^n = \frac{\partial}{\partial x}\lambda(x_i)u_{\mathrm{e},x}(x_i,t_n) + \mathcal{O}(\Delta x^2)\,.$$

After having treated the $[D_tD_tu_{\mathrm{e}}]_i^n$ term as well, we achieve

$$R_i^n = \mathcal{O}(\Delta x^2) + \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2\,.$$

The main conclusion is that the scheme is of second-order in time and space also in this variable coefficient case. The key ingredients for second order are the centered differences and the arithmetic mean for λ: all those building blocks feature second-order accuracy.

[sl: HP planned 1D wave equation on a staggered mesh here, five equal signs heading]

7.26. Linear wave equation in 2D/3D

The two-dimensional extension of (7.49) takes the form

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x,y,t), \quad (x,y)\in(0,L)\times(0,H),\ t\in(0,T], \tag{7.60}$$

where now $c(x,y)$ is the constant wave velocity of the physical medium $[0,L]\times[0,H]$. In compact notation, the PDE (7.60) can be written

$$u_{tt} = c^2(u_{xx}+u_{yy}) + f(x,y,t), \quad (x,y)\in(0,L)\times(0,H),\ t\in(0,T], \tag{7.61}$$

in 2D, while the 3D version reads

$$u_{tt} = c^2(u_{xx}+u_{yy}+u_{zz}) + f(x,y,z,t), \tag{7.62}$$

for $(x,y,z)\in(0,L)\times(0,H)\times(0,B)$ and $t\in(0,T]$.

Approximating the second-order derivatives by the standard formulas 7.4 yields the scheme

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu + D_zD_zu) + f]_{i,j,k}^n\,.$$

The truncation error is found from

$$[D_tD_tu_{\mathrm{e}} = c^2(D_xD_xu_{\mathrm{e}} + D_yD_yu_{\mathrm{e}} + D_zD_zu_{\mathrm{e}}) + f + R]_{i,j,k}^n\,.$$

The calculations from the 1D case can be repeated with the terms in the y and z directions. Collecting terms that fulfill the PDE, we end up with

$$R_{i,j,k}^n = [\frac{1}{12}u_{\mathrm{e},tttt}\Delta t^2 - c^2\frac{1}{12}\left(u_{\mathrm{e},xxxx}\Delta x^2 + u_{\mathrm{e},yyyy}\Delta x^2 + u_{\mathrm{e},zzzz}\Delta z^2\right)]_{i,j,k}^n + \mathcal{O}(\Delta t^4,\Delta x^4,\Delta y^4,\Delta z^4) \tag{7.63}$$

.

7.27. Linear diffusion equation in 1D

The standard, linear, 1D diffusion equation takes the form

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{7.64}$$

where $\alpha > 0$ is a constant diffusion coefficient. A more compact form of the diffusion equation is $u_t = \alpha u_{xx} + f$.

The spatial derivative in the diffusion equation, αu_{xx}, is commonly discretized as $[D_x D_x u]_i^n$. The time-derivative, however, can be treated by a variety of methods.

7.27.1. The Forward Euler scheme in time

Let us start with the simple Forward Euler scheme:

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n \,.$$

The truncation error arises as the residual R when inserting the exact solution u_e in the discrete equations:

$$[D_t^+ u_\mathrm{e} = \alpha D_x D_x u_\mathrm{e} + f + R]_i^n \,.$$

Now, using 7.6 and 7.4, we can transform the difference operators to derivatives:

$$u_{\mathrm{e},t}(x_i,t_n) + \frac{1}{2}u_{\mathrm{e},tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) = \alpha u_{\mathrm{e},xx}(x_i,t_n) + \frac{\alpha}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4) + f(x_i,t_n) + R_i^n \,.$$

The terms $u_{\mathrm{e},t}(x_i,t_n) - \alpha u_{\mathrm{e},xx}(x_i,t_n) - f(x_i,t_n)$ vanish because u_e solves the PDE. The truncation error then becomes

$$R_i^n = \frac{1}{2}u_{\mathrm{e},tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) + \frac{\alpha}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4) \,.$$

The Crank-Nicolson scheme in time The Crank-Nicolson method consists of using a centered difference for u_t and an arithmetic average of the u_{xx} term:

$$[D_t u]_i^{n+\frac{1}{2}} = \alpha \frac{1}{2}([D_x D_x u]_i^n + [D_x D_x u]_i^{n+1}) + f_i^{n+\frac{1}{2}} \,.$$

The equation for the truncation error is

$$[D_t u_\mathrm{e}]_i^{n+\frac{1}{2}} = \alpha \frac{1}{2}([D_x D_x u_\mathrm{e}]_i^n + [D_x D_x u_\mathrm{e}]_i^{n+1}) + f_i^{n+\frac{1}{2}} + R_i^{n+\frac{1}{2}} \,.$$

To find the truncation error, we start by expressing the arithmetic average in terms of values at time $t_{n+\frac{1}{2}}$. According to 7.11,

$$\frac{1}{2}([D_x D_x u_\mathrm{e}]_i^n + [D_x D_x u_\mathrm{e}]_i^{n+1}) = [D_x D_x u_\mathrm{e}]_i^{n+\frac{1}{2}} + \frac{1}{8}[D_x D_x u_{\mathrm{e},tt}]_i^{n+\frac{1}{2}}\Delta t^2 + \mathcal{O}(\Delta t^4) \,.$$

With 7.4 we can express the difference operator $D_x D_x u$ in terms of a derivative:

$$[D_x D_x u_\mathrm{e}]_i^{n+\frac{1}{2}} = u_{\mathrm{e},xx}(x_i, t_{n+\frac{1}{2}}) + \frac{1}{12} u_{\mathrm{e},xxxx}(x_i, t_{n+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

The error term from the arithmetic mean is similarly expanded,

$$\frac{1}{8}[D_x D_x u_{\mathrm{e},tt}]_i^{n+\frac{1}{2}}\Delta t^2 = \frac{1}{8} u_{\mathrm{e},ttxx}(x_i, t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^2 \Delta x^2)$$

The time derivative is analyzed using 7.9:

$$[D_t u]_i^{n+\frac{1}{2}} = u_{\mathrm{e},t}(x_i, t_{n+\frac{1}{2}}) + \frac{1}{24} u_{\mathrm{e},ttt}(x_i, t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Summing up all the contributions and notifying that

$$u_{\mathrm{e},t}(x_i, t_{n+\frac{1}{2}}) = \alpha u_{\mathrm{e},xx}(x_i, t_{n+\frac{1}{2}}) + f(x_i, t_{n+\frac{1}{2}}),$$

the truncation error is given by

$$\begin{aligned} R_i^{n+\frac{1}{2}} = &\frac{1}{8} u_{\mathrm{e},xx}(x_i, t_{n+\frac{1}{2}})\Delta t^2 + \frac{1}{12} u_{\mathrm{e},xxxx}(x_i, t_{n+\frac{1}{2}})\Delta x^2 + \\ &\frac{1}{24} u_{\mathrm{e},ttt}(x_i, t_{n+\frac{1}{2}})\Delta t^2 + + \mathcal{O}(\Delta x^4) + \mathcal{O}(\Delta t^4) + \mathcal{O}(\Delta t^2 \Delta x^2) \end{aligned}$$

7.28. Nonlinear diffusion equation in 1D

We address the PDE

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(u)\frac{\partial u}{\partial x}\right) + f(u),$$

with two potentially nonlinear coefficients $q(u)$ and $\alpha(u)$. We use a Backward Euler scheme with arithmetic mean for $\alpha(u)$,

$$[D^- u = D_x \overline{\alpha(u)}^x D_x u + f(u)]_i^n\,.$$

Inserting u_e defines the truncation error R:

$$[D^- u_\mathrm{e} = D_x \overline{\alpha(u_\mathrm{e})}^x D_x u_\mathrm{e} + f(u_\mathrm{e}) + R]_i^n\,.$$

The most computationally challenging part is the variable coefficient with $\alpha(u)$, but we can use the same setup as in Section Section 7.25 and arrive at a truncation error $\mathcal{O}(\Delta x^2)$ for the x-derivative term. The nonlinear term $[f(u_\mathrm{e})]_i^n = f(u_\mathrm{e}(x_i, t_n))$ matches x and t derivatives of u_e in the PDE. We end up with

$$R_i^n = -\frac{1}{2}\frac{\partial^2}{\partial t^2} u_\mathrm{e}(x_i, t_n)\Delta t + \mathcal{O}(\Delta x^2)\,.$$

7.29. Devito and Truncation Errors

Devito's `space_order` parameter directly controls the truncation error of spatial derivatives. Understanding this connection is essential for choosing appropriate accuracy settings in your simulations.

7.29.1. The `space_order` Parameter

When you create a `TimeFunction` or `Function` in Devito, the `space_order` parameter specifies the accuracy of spatial derivative approximations:

```
from devito import Grid, TimeFunction

grid = Grid(shape=(101,), extent=(1.0,))
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
```

The `space_order=2` specifies that spatial derivatives should use stencils accurate to $O(\Delta x^2)$. Higher orders are available:

`space_order`	Stencil Points	Accuracy
2	3	$O(\Delta x^2)$
4	5	$O(\Delta x^4)$
6	7	$O(\Delta x^6)$
8	9	$O(\Delta x^8)$

The relationship between `space_order` and stencil width follows from the truncation error analysis in Section 7.3. To achieve $O(\Delta x^{2k})$ accuracy for a second derivative, we need a stencil with $2k + 1$ points.

7.29.2. Viewing Generated Stencils

Devito allows you to inspect the symbolic expressions for derivatives, which reveals the stencil coefficients:

```
from devito import Grid, TimeFunction

grid = Grid(shape=(11,), extent=(1.0,))
x, = grid.dimensions
h = x.spacing  # Grid spacing symbol

# Compare different space orders
for order in [2, 4, 6]:
    u = TimeFunction(name='u', grid=grid, space_order=order)
    print(f"space_order={order}: {u.dx2}")
```

For `space_order=2`, this produces the familiar three-point stencil:

$$[D_x D_x u]_i = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2}$$

For `space_order=4`, Devito generates the five-point stencil with coefficients derived from the formulas in Section 7.7:

$$[D_x D_x u]_i = \frac{-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}}{12\Delta x^2}$$

7.29.3. Trading Accuracy for Performance

Higher `space_order` means:

- **Wider stencils**: More memory bandwidth required
- **More operations**: Additional floating-point operations per grid point
- **Better accuracy**: Smaller truncation error per grid point

For many problems, `space_order=2` is sufficient, especially when combined with grid refinement studies. However, wave propagation problems often benefit from higher orders:

```
# Wave equation with high-order spatial accuracy
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=8)
```

In geophysics applications (seismic imaging, full waveform inversion), `space_order=8` is common because the accuracy gain outweighs the computational overhead for wave propagation over many wavelengths.

7.29.4. Matching Temporal and Spatial Accuracy

For a scheme to achieve its full accuracy potential, the truncation errors from time and space discretization should be balanced. As shown in Section 7.23, the standard leapfrog scheme for the wave equation has truncation error:

$$R = O(\Delta t^2) + O(\Delta x^2)$$

Using `space_order=4` with `time_order=2` means the spatial error $O(\Delta x^4)$ may be much smaller than the temporal error $O(\Delta t^2)$. This is acceptable when:

1. You want higher spatial accuracy for a fixed grid
2. The time step is limited by stability (CFL condition), not accuracy
3. You're doing convergence studies focused on spatial refinement

For problems where temporal accuracy is also critical, consider using higher-order time integration schemes or smaller time steps.

7.29.5. Verifying Convergence Rates

As discussed in the empirical verification section (Section 7.14), we can verify that Devito's stencils achieve the expected convergence rates:

```
import numpy as np
from devito import Grid, TimeFunction, Eq, Operator

def test_laplacian_accuracy(space_order):
    """Verify convergence rate of Laplacian approximation."""
    errors = []
```

```
    dx_values = []

    for N in [20, 40, 80, 160]:
        grid = Grid(shape=(N+1,), extent=(1.0,))
        x, = grid.dimensions

        u = TimeFunction(name='u', grid=grid, space_order=space_order)
        u.data[0, :] = np.sin(np.pi * np.linspace(0, 1, N+1))

        # Exact second derivative: -pi^2 * sin(pi*x)
        exact = -np.pi**2 * np.sin(np.pi * np.linspace(0, 1, N+1))

        # Devito approximation
        laplacian = u.dx2.evaluate
        numerical = np.array(laplacian.data[0, :])

        # Compute error (excluding boundary points)
        error = np.max(np.abs(numerical[2:-2] - exact[2:-2]))
        errors.append(error)
        dx_values.append(1.0/N)

    # Estimate convergence rate
    rates = [np.log(errors[i]/errors[i+1])/np.log(2)
             for i in range(len(errors)-1)]
    return np.mean(rates)

# Expected: rate ~ 2 for space_order=2, rate ~ 4 for space_order=4
```

The measured rates should match the theoretical orders from truncation error analysis, providing verification that both the theory and implementation are correct.

7.30. Exercise: Truncation error of a weighted mean

Derive the truncation error of the weighted mean in 7.10.

> Expand u_e^{n+1} and u_e^{n} around $t_{n+\theta}$.

7.31. Exercise: Simulate the error of a weighted mean

We consider the weighted mean

$$u_\mathrm{e}(t_n) \approx \theta u_\mathrm{e}^{n+1} + (1-\theta)u_\mathrm{e}^{n}\,.$$

Choose some specific function for $u_e(t)$ and compute the error in this approximation for a sequence of decreasing $\Delta t = t_{n+1} - t_n$ and for $\theta = 0, 0.25, 0.5, 0.75, 1$. Assuming that the error equals $C\Delta t^r$, for some constants C and r, compute r for the two smallest Δt values for each choice of θ and compare with the truncation error 7.10.

7.32. Exercise: Verify a truncation error formula

Set up a numerical experiment as explained in Section Section 7.14 for verifying the formulas 7.8.

7.33. Problem: Truncation error of the Backward Euler scheme

Derive the truncation error of the Backward Euler scheme for the decay ODE $u' = -au$ with constant a. Extend the analysis to cover the variable-coefficient case $u' = -a(t)u + b(t)$.

7.34. Exercise: Empirical estimation of truncation errors

Use the ideas and tools from Section Section 7.14 to estimate the rate of the truncation error of the Backward Euler and Crank-Nicolson schemes applied to the exponential decay model $u' = -au$, $u(0) = I$.

Hint

In the Backward Euler scheme, the truncation error can be estimated at mesh points $n = 1, \ldots, N$, while the truncation error must be estimated at midpoints $t_{n+\frac{1}{2}}$, $n = 0, \ldots, N-1$ for the Crank-Nicolson scheme. The `truncation_error(dt, N)` function to be supplied to the `estimate` function needs to carefully implement these details and return the right `t` array such that `t[i]` is the time point corresponding to the quantities `R[i]` and `R_a[i]`.

7.35. Exercise: Correction term for a Backward Euler scheme

Consider the model $u' = -au$, $u(0) = I$. Use the ideas of Section Section 7.15 to add a correction term to the ODE such that the Backward Euler scheme applied to the perturbed ODE problem is of second order in Δt. Find the amplification factor.

7.36. Problem: Verify the effect of correction terms

Make a program that solves $u' = -au$, $u(0) = I$, by the θ-rule and computes convergence rates. Adjust a such that it incorporates correction terms. Run the program to verify that the error from the Forward and Backward Euler schemes with perturbed a is $\mathcal{O}(\Delta t^2)$, while the error arising from the Crank-Nicolson scheme with perturbed a is $\mathcal{O}(\Delta t^4)$.

7.37. Problem: Truncation error of the Crank-Nicolson scheme

The variable-coefficient ODE $u' = -a(t)u + b(t)$ can be discretized in two different ways by the Crank-Nicolson scheme, depending on whether we use averages for a and b or compute them at the midpoint $t_{n+\frac{1}{2}}$:

$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{7.65}$$

$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}} \tag{7.66}$$

.Compute the truncation error in both cases.

7.38. Problem: Truncation error of $u' = f(u, t)$

Consider the general nonlinear first-order scalar ODE

$$u'(t) = f(u(t), t)\,.$$

Show that the truncation error in the Forward Euler scheme,

$$[D_t^+ u = f(u, t)]^n,$$

and in the Backward Euler scheme,

$$[D_t^- u = f(u, t)]^n,$$

both are of first order, regardless of what f is.

Showing the order of the truncation error in the Crank-Nicolson scheme,

$$[D_t u = f(u, t)]^{n+\frac{1}{2}},$$

is somewhat more involved: Taylor expand u_{e}^n, u_{e}^{n+1}, $f(u_{\mathrm{e}}^n, t_n)$, and $f(u_{\mathrm{e}}^{n+1}, t_{n+1})$ around $t_{n+\frac{1}{2}}$, and use that

$$\frac{df}{dt} = \frac{\partial f}{\partial u}u' + \frac{\partial f}{\partial t}\,.$$

Check that the derived truncation error is consistent with previous results for the case $f(u, t) = -au$.

7.39. Exercise: Truncation error of $[D_t D_t u]^n$

Derive the truncation error of the finite difference approximation 7.4 to the second-order derivative.

7.40. Exercise: Investigate the impact of approximating $u'(0)$

Section Section 7.19 describes two ways of discretizing the initial condition $u'(0) = V$ for a vibration model $u'' + \omega^2 u = 0$: a centered difference $[D_{2t}u = V]^0$ or a forward difference $[D_t^+ u = V]^0$. The program `vib_undamped.py` solves $u'' + \omega^2 u = 0$ with $[D_{2t}u = 0]^0$ and features a function `convergence_rates` for computing the order of the error in the numerical solution. Modify this program such that it applies the forward difference $[D_t^+ u = 0]^0$ and report how this simpler and more convenient approximation impacts the overall convergence rate of the scheme.

7.41. Problem: Investigate the accuracy of a simplified scheme

Consider the ODE

$$mu'' + \beta|u'|u' + s(u) = F(t)\,.$$

The term $|u'|u'$ quickly gives rise to nonlinearities and complicates the scheme. Why not simply apply a backward difference to this term such that it only involves known values? That is, we propose to solve

$$[mD_tD_tu + \beta|D_t^- u|D_t^- u + s(u) = F]^n\,.$$

Drop the absolute value for simplicity and find the truncation error of the scheme. Perform numerical experiments with the scheme and compared with the one based on centered differences. Can you illustrate the accuracy loss visually in real computations, or is the asymptotic analysis here mainly of theoretical interest?

8. Software Engineering

8.1. Mathematical model

Let u_t, u_{tt}, u_x, u_{xx} denote derivatives of u with respect to the subscript, i.e., u_{tt} is a second-order time derivative and u_x is a first-order space derivative. The initial-boundary value problem implemented in the `wave1D_dn_vc.py` code is

$$
\begin{aligned}
u_{tt} &= (q(x)u_x)_x + f(x,t), \quad x \in (0,L),\ t \in (0,T] \\
u(x,0) &= I(x), \quad x \in [0,L] \\
u_t(x,0) &= V(t), \quad x \in [0,L] \\
u(0,t) &= U_0(t) \text{ or } u_x(0,t) = 0, \quad t \in (0,T] \\
u(L,t) &= U_L(t) \text{ or } u_x(L,t) = 0, \quad t \in (0,T]
\end{aligned} \tag{8.1}
$$

We allow variable wave velocity $c^2(x) = q(x)$, and Dirichlet or homogeneous Neumann conditions at the boundaries.

8.2. Numerical discretization

The PDE is discretized by second-order finite differences in time and space, with arithmetic mean for the variable coefficient

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n .$$

The Neumann boundary conditions are discretized by

$$[D_{2x} u]_i^n = 0,$$

at a boundary point i. The details of how the numerical scheme is worked out are described in Section 2.35 and Section 2.42.

8.3. A solver function

The general initial-boundary value problem solved by finite difference methods can be implemented as shown in the following `solver` function (taken from the file `wave1D_dn_vc.py`). This function builds on simpler versions described in Section 2.14, Section 2.22, Section 2.35, and Section 2.42. There are several quite advanced constructs that will be commented upon later. The code is lengthy, but that is because we provide a lot of flexibility with respect to input arguments, boundary conditions, and optimization (scalar versus vectorized loops).

8.4. Storing simulation data in files

Numerical simulations produce large arrays as results and the software needs to store these arrays on disk. Several methods are available in Python. We recommend to use tailored solutions for large arrays and not standard file storage tools such as `pickle` (`cPickle` for speed in Python version 2) and `shelve`, because the tailored solutions have been optimized for array data and are hence much faster than the standard tools.

8.5. Using `savez` to store arrays in files

8.5.1. Storing individual arrays

The `numpy.storez` function can store a set of arrays to a named file in a zip archive. An associated function `numpy.load` can be used to read the file later. Basically, we call `numpy.storez(filename, **kwargs)`, where `kwargs` is a dictionary containing array names as keys and the corresponding array objects as values. Very often, the solution at a time point is given a natural name where the name of the variable and the time level counter are combined, e.g., `u11` or `v39`. Suppose `n` is the time level counter and we have two solution arrays, `u` and `v`, that we want to save to a zip archive. The appropriate code is

```
import numpy as np
u_name = 'u%04d' % n   # array name
v_name = 'v%04d' % n   # array name
kwargs = {u_name: u, v_name: v}   # keyword args for savez
fname = '.mydata%04d.dat' % n
np.savez(fname, **kwargs)
if n == 0:           # store x once
    np.savez('.mydata_x.dat', x=x)
```

Since the name of the array must be given as a keyword argument to `savez`, and the name must be constructed as shown, it becomes a little tricky to do the call, but with a dictionary `kwargs` and `**kwargs`, which sends each key-value pair as individual keyword arguments, the task gets accomplished.

8.5.2. Merging zip archives

Each separate call to `np.savez` creates a new file (zip archive) with extension `.npz`. It is very convenient to collect all results in one archive instead. This can be done by merging all the individual `.npz` files into a single zip archive:

```
def merge_zip_archives(individual_archives, archive_name):
    """
    Merge individual zip archives made with numpy.savez into
    one archive with name archive_name.
    The individual archives can be given as a list of names
    or as a Unix wild chard filename expression for glob.glob.
    The result of this function is that all the individual
    archives are deleted and the new single archive made.
    """
    import zipfile
    archive = zipfile.ZipFile(
        archive_name, 'w', zipfile.ZIP_DEFLATED,
        allowZip64=True)
    if isinstance(individual_archives, (list,tuple)):
        filenames = individual_archives
    elif isinstance(individual_archives, str):
        filenames = glob.glob(individual_archives)

    for filename in filenames:
        f = zipfile.ZipFile(filename,  'r',
                            zipfile.ZIP_DEFLATED)
        for name in f.namelist():
            data = f.open(name, 'r')
            archive.writestr(name[:-4], data.read())
        f.close()
        os.remove(filename)
    archive.close()
```

Here we remark that `savez` automatically adds the `.npz` extension to the names of the arrays we store. We do not want this extension in the final archive.

8.5.3. Reading arrays from zip archives

Archives created by `savez` or the merged archive we describe above with name of the form `myarchive.npz`, can be conveniently read by the `numpy.load` function:

```
import numpy as np
array_names = np.load(`myarchive.npz`)
for array_name in array_names:
```

8.6. Using `joblib` to store arrays in files

The Python package `joblib` has nice functionality for efficient storage of arrays on disk. The following class applies this functionality so that one can save an array, or in fact any Python data structure (e.g., a dictionary of arrays), to disk under a certain name. Later, we can retrieve the

object by use of its name. The name of the directory under which the arrays are stored by `joblib` can be given by the user.

```
class Storage:
    """
    Store large data structures (e.g. numpy arrays) efficiently
    using joblib.

    Use:

    >>> from Storage import Storage
    >>> storage = Storage(cachedir='tmp_u01', verbose=1)
    >>> import numpy as np
    >>> a = np.linspace(0, 1, 100000) # large array
    >>> b = np.linspace(0, 1, 100000) # large array
    >>> storage.save('a', a)
    >>> storage.save('b', b)
    >>> # later
    >>> a = storage.retrieve('a')
    >>> b = storage.retrieve('b')
    """

    def __init__(self, cachedir="tmp", verbose=1):
        """
        Parameters
        ----------
        cachedir: str
             Name of directory where objects are stored in files.
        verbose: bool, int
             Let joblib and this class speak when storing files
             to disk.
        """
        import joblib

        self.memory = joblib.Memory(cachedir=cachedir, verbose=verbose)
        self.verbose = verbose
        self.retrieve = self.memory.cache(self.retrieve, ignore=["data"])
        self.save = self.retrieve

    def retrieve(self, name, data=None):
        if self.verbose > 0:
            print("joblib save of", name)
        return data
```

The `retrieve` and `save` functions, which do the work, seem quite magic. The idea is that `joblib` looks at the `name` parameter and saves the return value `data` to disk if the `name` parameter has not been used in a previous call. Otherwise, if `name` is already registered, `joblib` fetches the `data` object

from file and returns it (this is an example of a memoize function, see Section 2.1.4 in (Langtangen and Pedersen 2016) for a brief explanation]).

8.7. Using a hash to create a file or directory name

Array storage techniques like those outlined in Sections Section 8.6 and Section 8.5.1 demand the user to assign a name for the file(s) or directory where the solution is to be stored. Ideally, this name should reflect parameters in the problem such that one can recognize an already run simulation. One technique is to make a hash string out of the input data. A hash string is a 40-character long hexadecimal string that uniquely reflects another potentially much longer string. (You may be used to hash strings from the Git version control system: every committed version of the files in Git is recognized by a hash string.)

Suppose you have some input data in the form of functions, `numpy` arrays, and other objects. To turn these input data into a string, we may grab the source code of the functions, use a very efficient hash method for potentially large arrays, and simply convert all other objects via `str` to a string representation. The final string, merging all input data, is then converted to an SHA1 hash string such that we represent the input with a 40-character long string.

```
def myfunction(func1, func2, array1, array2, obj1, obj2):
    import inspect, joblib, hashlib
    data = (inspect.getsource(func1),
            inspect.getsource(func2),
        joblib.hash(array1),
        joblib.hash(array2),
        str(obj1),
        str(obj2))
    hash_input = hashlib.sha1(data).hexdigest()
```

It is wise to use `joblib.hash` and not try to do a `str(array1)`, since that string can be *very* long, and `joblib.hash` is more efficient than `hashlib` when turning these data into a hash.

> ⚠ Remark: turning function objects into their source code is unreliable!
>
> The idea of turning a function object into a string via its source code may look smart, but is not a completely reliable solution. Suppose we have some function
>
> ```
> x0 = 0.1
> f = lambda x: 0 if x <= x0 else 1
> ```
>
> The source code will be `f = lambda x: 0 if x <= x0 else 1`, so if the calling code changes the value of `x0` (which `f` remembers - it is a closure), the source remains unchanged, the hash is the same, and the change in input data is unnoticed. Consequently, the technique above must be used with care. The user can always just remove the stored files in disk and thereby force a recomputation (provided the software applies a hash to test if a zip archive or `joblib` subdirectory exists, and if so, avoids recomputation).

We use `numpy.storez` to store the solution at each time level on disk. Such actions must be taken care of outside the `solver` function, more precisely in the `user_action` function that is called at every time level.

We have, in the wave1D_dn_vc.py code, implemented the `user_action` callback function as a class `PlotAndStoreSolution` with a `__call__(self, x, t, t, n)` method for the `user_action` function. Basically, `__call__` stores and plots the solution. The storage makes use of the `numpy.savez` function for saving a set of arrays to a zip archive. Here, in this callback function, we want to save one array, `u`. Since there will be many such arrays, we introduce the array names `'u%04d' % n` and closely related filenames. The usage of `numpy.savez` in `__call__` goes like this:

```
from numpy import savez
name = 'u%04d' % n   # array name
kwargs = {name: u}   # keyword args for savez
fname = '.' + self.filename + '_' + name + '.dat'
self.t.append(t[n])  # store corresponding time value
savez(fname, **kwargs)
if n == 0:           # store x once
    savez('.' + self.filename + '_x.dat', x=x)
```

For example, if `n` is 10 and `self.filename` is `tmp`, the above call to `savez` becomes `savez('.tmp_u0010.dat', u0010=u)`. The actual filename becomes `.tmp_u0010.dat.npz`. The actual array name becomes `u0010.npy`.

Each `savez` call results in a file, so after the simulation we have one file per time level. Each file produced by `savez` is a zip archive. It makes sense to merge all the files into one. This is done in the `close_file` method in the `PlotAndStoreSolution` class. The code goes as follows.

```
class PlotAndStoreSolution:
    ...
    def close_file(self, hashed_input):
        """
        Merge all files from savez calls into one archive.
        hashed_input is a string reflecting input data
        for this simulation (made by solver).
        """
        if self.filename is not None:
            savez('.' + self.filename + '_t.dat',
                  t=array(self.t, dtype=float))
            archive_name = '.' + hashed_input + '_archive.npz'
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            merge_zip_archives(filenames, archive_name)
```

We use various `ZipFile` functionality to extract the content of the individual files (each with name `filename`) and write it to the merged archive (`archive`). There is only one array in each individual file (`filename`) so strictly speaking, there is no need for the loop `for name in f.namelist()` (as `f.namelist()` returns a list of length 1). However, in other applications where we compute

more arrays at each time level, `savez` will store all these and then there is need for iterating over `f.namelist()`.

Instead of merging the archives written by `savez` we could make an alternative implementation that writes all our arrays into one archive. This is the subject of Exercise Section 8.31.

8.8. Making hash strings from input data

The `hashed_input` argument, used to name the resulting archive file with all solutions, is supposed to be a hash reflecting all import parameters in the problem such that this simulation has a unique name. The `hashed_input` string is made in the `solver` function, using the `hashlib` and `inspect` modules, based on the arguments to `solver`:

```
import hashlib, inspect
data = inspect.getsource(I) + '_' + inspect.getsource(V) + \
       '**' + inspect.getsource(f) + '**' + str(c) + '_' + \
       ('None' if U_0 is None else inspect.getsource(U_0)) + \
       ('None' if U_L is None else inspect.getsource(U_L)) + \
       '**' + str(L) + str(dt) + '**' + str(C) + '_' + str(T) + \
       '_' + str(stability_safety_factor)
hashed_input = hashlib.sha1(data).hexdigest()
```

To get the source code of a function `f` as a string, we use `inspect.getsource(f)`. All input, functions as well as variables, is then merged to a string `data`, and then `hashlib.sha1` makes a unique, much shorter (40 characters long), fixed-length string out of `data` that we can use in the archive filename.

> **Remark**
>
> Note that the construction of the `data` string is not fool proof: if, e.g., `I` is a formula with parameters and the parameters change, the source code is still the same and `data` and hence the hash remains unaltered. The implementation must therefore be used with care!

8.9. Avoiding rerunning previously run cases

If the archive file whose name is based on `hashed_input` already exists, the simulation with the current set of parameters has been done before and one can avoid redoing the work. The `solver` function returns the CPU time and `hashed_input`, and a negative CPU time means that no simulation was run. In that case we should not call the `close_file` method above (otherwise we overwrite the archive with just the `self.t` array). The typical usage goes like

```
action = PlotAndStoreSolution(...)
dt = (L/Nx)/C  # choose the stability limit with given Nx
cpu, hashed_input = solver(
    I=lambda x: ...,
```

```
    V=0, f=0, c=1, U_0=lambda t: 0, U_L=None, L=1,
    dt=dt, C=C, T=T,
    user_action=action, version='vectorized',
    stability_safety_factor=1)
action.make_movie_file()
if cpu > 0:  # did we generate new data?
    action.close_file(hashed_input)
```

8.10. Verification

8.10.1. Vanishing approximation error

Exact solutions of the numerical equations are always attractive for verification purposes since the software should reproduce such solutions to machine precision. With Dirichlet boundary conditions we can construct a function that is linear in t and quadratic in x that is also an exact solution of the scheme, while with Neumann conditions we are left with testing just a constant solution (see comments in Section 2.40).

8.10.2. Convergence rates

A more general method for verification is to check the convergence rates. We must introduce one discretization parameter h and assume an error model $E = Ch^r$, where C and r are constants to be determine (i.e., r is the rate that we are interested in). Given two experiments with different resolutions h_i and h_i-1, we can estimate r by

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})},$$

where E_i is the error corresponding to h_i and E_{i-1} corresponds to h_{i-1}. Section 2.9 explains the details of this type of verification and how we introduce the single discretization parameter $h = \Delta t = \hat{c}\Delta t$, for some constant $\hat{c}$. To compute the error, we had to rely on a global variable in the user action function. Below is an implementation where we have a more elegant solution in terms of a class: the `error` variable is not a class attribute and there is no need for a global error (which is always considered an advantage).

The returned sequence `r` should converge to 2 since the error analysis in Section 2.61 predicts various error measures to behave like $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2)$. We can easily run the case with standing waves and the analytical solution $u(x,t) = \cos(\frac{2\pi}{L}t)\sin(\frac{2\pi}{L}x)$. The call will be very similar to the one provided in the `test_convrate_sincos` function in Section 2.18, see the file `src/wave/wave1D/wave1D_dn_vc.py` for details.

Many who know about class programming prefer to organize their software in terms of classes. This gives a richer application programming interface (API) since a function solver must have all its input data in terms of arguments, while a class-based solver naturally has a mix of method arguments

and user-supplied methods. (Well, to be more precise, our solvers have demanded `user_action` to be a function provided by the user, so it is possible to mix variables and functions in the input also with a solver function.)

We will next illustrate how some of the functionality in `wave1D_dn_vc.py` may be implemented by using classes. Focusing on class implementation aspects, we restrict the example case to a simpler wave with constant wave speed c. Applying the method of manufactured solutions, we test whether the class based implementation is able to compute the known exact solution within machine precision.

We will create a class `Problem` to hold the physical parameters of the problem and a class `Solver` to hold the numerical solution parameters besides the solver function itself. As the number of parameters increases, so does the amount of repetitive code. We therefore take the opportunity to illustrate how this may be counteracted by introducing a super class `Parameters` that allows code to be parameterized. In addition, it is convenient to collect the arrays that describe the mesh in a special `Mesh` class and make a class `Function` for a mesh function (mesh point values and its mesh). All the following code is found in wave1D_oo.py.

8.11. Class Parameters

The classes `Problem` and `Solver` both inherit class `Parameters`, which handles reading of parameters from the command line and has methods for setting and getting parameter values. Since processing dictionaries is easier than processing a collection of individual attributes, the class `Parameters` requires each class `Problem` and `Solver` to represent their parameters by dictionaries, one compulsory and two optional ones. The compulsory dictionary, `self.prm`, contains all parameters, while a second and optional dictionary, `self.type`, holds the associated object types, and a third and optional dictionary, `self.help`, stores help strings. The `Parameters` class may be implemented as follows:

```
class Parameters:
    def __init__(self):
        """
        Subclasses must initialize self.prm with
        parameters and default values, self.type with
        the corresponding types, and self.help with
        the corresponding descriptions of parameters.
        self.type and self.help are optional, but
        self.prms must be complete and contain all parameters.
        """
        pass

    def ok(self):
        """Check if attr. prm, type, and help are defined."""
        if (
            hasattr(self, "prm")
            and isinstance(self.prm, dict)
            and hasattr(self, "type")
```

```
            and isinstance(self.type, dict)
            and hasattr(self, "help")
            and isinstance(self.help, dict)
        ):
            return True
        else:
            raise ValueError(
                "The constructor in class %s does not "
                "initialize the\ndictionaries "
                "self.prm, self.type, self.help!" % self.__class__.__name__
            )

    def _illegal_parameter(self, name):
        """Raise exception about illegal parameter name."""
        raise ValueError(
            'parameter "%s" is not registered.\nLegal '
            "parameters are\n%s" % (name, " ".join(list(self.prm.keys())))
        )

    def set(self, **parameters):
        """Set one or more parameters."""
        for name in parameters:
            if name in self.prm:
                self.prm[name] = parameters[name]
            else:
                self._illegal_parameter(name)

    def get(self, name):
        """Get one or more parameter values."""
        if isinstance(name, (list, tuple)):  # get many?
            for n in name:
                if n not in self.prm:
                    self._illegal_parameter(name)
            return [self.prm[n] for n in name]
        else:
            if name not in self.prm:
                self._illegal_parameter(name)
            return self.prm[name]

    def __getitem__(self, name):
        """Allow obj[name] indexing to look up a parameter."""
        return self.get(name)

    def __setitem__(self, name, value):
        """
        Allow obj[name] = value syntax to assign a parameter's value.
        """
```

```
        return self.set(name=value)

    def define_command_line_options(self, parser=None):
        self.ok()
        if parser is None:
            import argparse

            parser = argparse.ArgumentParser()

        for name in self.prm:
            tp = self.type[name] if name in self.type else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                "--" + name, default=self.get(name), metavar=name, type=tp, help=help
            )

        return parser

    def init_from_command_line(self, args):
        for name in self.prm:
            self.prm[name] = getattr(args, name)
```

8.12. Class Problem

Inheriting the `Parameters` class, our class `Problem` is defined as:

```
class Problem(Parameters):
    """
    Physical parameters for the wave equation
    u_tt = (c**2*u_x)_x + f(x,t) with t in [0,T] and
    x in (0,L). The problem definition is implied by
    the method of manufactured solution, choosing
    u(x,t)=x(L-x)(1+t/2) as our solution. This solution
    should be exactly reproduced when c is const.
    """

    def __init__(self):
        self.prm = dict(L=2.5, c=1.5, T=18)
        self.type = dict(L=float, c=float, T=float)
        self.help = dict(
            L="1D domain",
            c="coefficient (wave velocity) in PDE",
            T="end time of simulation",
        )
```

```
    def u_exact(self, x, t):
        L = self["L"]
        return x * (L - x) * (1 + 0.5 * t)

    def I(self, x):
        return self.u_exact(x, 0)

    def V(self, x):
        return 0.5 * self.u_exact(x, 0)

    def f(self, x, t):
        c = self["c"]
        return 2 * (1 + 0.5 * t) * c**2

    def U_0(self, t):
        return self.u_exact(0, t)

    U_L = None
```

8.13. Class Mesh

The `Mesh` class can be made valid for a space-time mesh in any number of space dimensions. To make the class versatile, the constructor accepts either a tuple/list of number of cells in each spatial dimension or a tuple/list of cell spacings. In addition, we need the size of the hypercube mesh as a tuple/list of 2-tuples with lower and upper limits of the mesh coordinates in each direction. For 1D meshes it is more natural to just write the number of cells or the cell size and not wrap it in a list. We also need the time interval from `t0` to `T`. Giving no spatial discretization information implies a time mesh only, and vice versa. The `Mesh` class with documentation and a doc test should now be self-explanatory:

```
import numpy as np


class Mesh:
    """
    Holds data structures for a uniform mesh on a hypercube in
    space, plus a uniform mesh in time.

    ======== ==================================================
    Argument                Explanation
    ======== ==================================================
    L        List of 2-lists of min and max coordinates
             in each spatial direction.
    T        Final time in time mesh.
    Nt       Number of cells in time mesh.
    dt       Time step. Either Nt or dt must be given.
```

```
    N        List of number of cells in the spatial directions.
    d        List of cell sizes in the spatial directions.
             Either N or d must be given.
    ======== ==================================================

    Users can access all the parameters mentioned above, plus
    ``x[i]`` and ``t`` for the coordinates in direction ``i``
    and the time coordinates, respectively.

    Examples:

    >>> from UniformFDMesh import Mesh
    >>>
    >>> # Simple space mesh
    >>> m = Mesh(L=[0,1], N=4)
    >>> print m.dump()
    space: [0,1] N=4 d=0.25
    >>>
    >>> # Simple time mesh
    >>> m = Mesh(T=4, dt=0.5)
    >>> print m.dump()
    time: [0,4] Nt=8 dt=0.5
    >>>
    >>> # 2D space mesh
    >>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1])
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1
    >>>
    >>> # 2D space mesh and time mesh
    >>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1], Nt=10, T=3)
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1 time: [0,3] Nt=10 dt=0.3

    """

    def __init__(self, L=None, T=None, t0=0, N=None, d=None, Nt=None, dt=None):
        if N is None and d is None:
            if Nt is None and dt is None:
                raise ValueError("Mesh constructor: either Nt or dt must be given")
            if T is None:
                raise ValueError("Mesh constructor: T must be given")
        if Nt is None and dt is None:
            if N is None and d is None:
                raise ValueError("Mesh constructor: either N or d must be given")
            if L is None:
                raise ValueError("Mesh constructor: L must be given")
```

```
        if L is not None and isinstance(L[0], (float, int)):
            L = [L]
        if N is not None and isinstance(N, (float, int)):
            N = [N]
        if d is not None and isinstance(d, (float, int)):
            d = [d]

        self.x = None
        self.t = None
        self.Nt = None
        self.dt = None
        self.N = None
        self.d = None
        self.t0 = t0

        if N is None and d is not None and L is not None:
            self.L = L
            if len(d) != len(L):
                raise ValueError(
                    "d has different size (no of space dim.) from L: %d vs %d",
                    len(d),
                    len(L),
                )
            self.d = d
            self.N = [
                int(round(float(self.L[i][1] - self.L[i][0]) / d[i]))
                for i in range(len(d))
            ]
        if d is None and N is not None and L is not None:
            self.L = L
            if len(N) != len(L):
                raise ValueError(
                    "N has different size (no of space dim.) from L: %d vs %d",
                    len(N),
                    len(L),
                )
            self.N = N
            self.d = [float(self.L[i][1] - self.L[i][0]) / N[i] for i in range(len(N))]

        if Nt is None and dt is not None and T is not None:
            self.T = T
            self.dt = dt
            self.Nt = int(round(T / dt))
        if dt is None and Nt is not None and T is not None:
            self.T = T
            self.Nt = Nt
            self.dt = T / float(Nt)
```

```
        if self.N is not None:
            self.x = [
                np.linspace(self.L[i][0], self.L[i][1], self.N[i] + 1)
                for i in range(len(self.L))
            ]
        if Nt is not None:
            self.t = np.linspace(self.t0, self.T, self.Nt + 1)

    def get_num_space_dim(self):
        return len(self.d) if self.d is not None else 0

    def has_space(self):
        return self.d is not None

    def has_time(self):
        return self.dt is not None

    def dump(self):
        s = ""
        if self.has_space():
            s += (
                "space: "
                + "x".join(
                    ["[%g,%g]" % (self.L[i][0], self.L[i][1]) for i in range(len(self.L))]
                )
                + " N="
            )
            s += "x".join([str(Ni) for Ni in self.N]) + " d="
            s += ",".join([str(di) for di in self.d])
        if self.has_space() and self.has_time():
            s += " "
        if self.has_time():
            s += (
                "time: "
                + "[%g,%g]" % (self.t0, self.T)
                + " Nt=%g" % self.Nt
                + " dt=%g" % self.dt
            )
        return s
```

i We rely on attribute access - not get/set functions!

Java programmers, in particular, are used to get/set functions in classes to access internal data. In Python, we usually apply direct access of the attribute, such as `m.N[i]` if `m` is a `Mesh` object. A widely used convention is to do this as long as access to an attribute does not require additional code. In that case, one applies a property construction. The original

interface remains the same after a property is introduced (in contrast to Java), so user will not notice a change to properties.
The only argument against direct attribute access in class `Mesh` is that the attributes are read-only so we could avoid offering a set function. Instead, we rely on the user that she does not assign new values to the attributes.

8.14. Class Function

A class `Function` is handy to hold a mesh and corresponding values for a scalar or vector function over the mesh. Since we may have a time or space mesh, or a combined time and space mesh, with one or more components in the function, some if tests are needed for allocating the right array sizes. To help the user, an `indices` attribute with the name of the indices in the final array `u` for the function values is made. The examples in the doc string should explain the functionality.

```
class Function:
    """
    A scalar or vector function over a mesh (of class Mesh).

    ========== ================================================
    Argument                       Explanation
    ========== ================================================
    mesh       Class Mesh object: spatial and/or temporal mesh.
    num_comp   Number of components in function (1 for scalar).
    space_only True if the function is defined on the space mesh
               only (to save space). False if function has values
               in space and time.
    ========== ================================================

    The indexing of ``u``, which holds the mesh point values of the
    function, depends on whether we have a space and/or time mesh.

    Examples:

    >>> from UniformFDMesh import Mesh, Function
    >>>
    >>> # Simple space mesh
    >>> m = Mesh(L=[0,1], N=4)
    >>> print m.dump()
    space: [0,1] N=4 d=0.25
    >>> f = Function(m)
    >>> f.indices
    ['x0']
    >>> f.u.shape
    (5,)
    >>> f.u[4]  # space point 4
```

```
0.0
>>>
>>> # Simple time mesh for two components
>>> m = Mesh(T=4, dt=0.5)
>>> print m.dump()
time: [0,4] Nt=8 dt=0.5
>>> f = Function(m, num_comp=2)
>>> f.indices
['time', 'component']
>>> f.u.shape
(9, 2)
>>> f.u[3,1]  # time point 3, comp=1 (2nd comp.)
0.0
>>>
>>> # 2D space mesh
>>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1])
>>> print m.dump()
space: [0,1]x[-1,1] N=2x2 d=0.5,1
>>> f = Function(m)
>>> f.indices
['x0', 'x1']
>>> f.u.shape
(3, 3)
>>> f.u[1,2]  # space point (1,2)
0.0
>>>
>>> # 2D space mesh and time mesh
>>> m = Mesh(L=[[0,1],[-1,1]], d=[0.5,1], Nt=10, T=3)
>>> print m.dump()
space: [0,1]x[-1,1] N=2x2 d=0.5,1 time: [0,3] Nt=10 dt=0.3
>>> f = Function(m, num_comp=2, space_only=False)
>>> f.indices
['time', 'x0', 'x1', 'component']
>>> f.u.shape
(11, 3, 3, 2)
>>> f.u[2,1,2,0]  # time step 2, space point (1,2), comp=0
0.0
>>> # Function with space data only
>>> f = Function(m, num_comp=1, space_only=True)
>>> f.indices
['x0', 'x1']
>>> f.u.shape
(3, 3)
>>> f.u[1,2]  # space point (1,2)
0.0
"""
```

```
    def __init__(self, mesh, num_comp=1, space_only=True):
        self.mesh = mesh
        self.num_comp = num_comp
        self.indices = []

        if (self.mesh.has_space() and not self.mesh.has_time()) or (
            self.mesh.has_space() and self.mesh.has_time() and space_only
        ):
            if num_comp == 1:
                self.u = np.zeros([self.mesh.N[i] + 1 for i in range(len(self.mesh.N))])
                self.indices = ["x" + str(i) for i in range(len(self.mesh.N))]
            else:
                self.u = np.zeros(
                    [self.mesh.N[i] + 1 for i in range(len(self.mesh.N))] + [num_comp]
                )
                self.indices = ["x" + str(i) for i in range(len(self.mesh.N))] + [
                    "component"
                ]
        if not self.mesh.has_space() and self.mesh.has_time():
            if num_comp == 1:
                self.u = np.zeros(self.mesh.Nt + 1)
                self.indices = ["time"]
            else:
                self.u = np.zeros((self.mesh.Nt + 1, num_comp))
                self.indices = ["time", "component"]
        if self.mesh.has_space() and self.mesh.has_time() and not space_only:
            size = [self.mesh.Nt + 1] + [
                self.mesh.N[i] + 1 for i in range(len(self.mesh.N))
            ]
            if num_comp > 1:
                self.indices = (
                    ["time"]
                    + ["x" + str(i) for i in range(len(self.mesh.N))]
                    + ["component"]
                )
                size += [num_comp]
            else:
                self.indices = ["time"] + ["x" + str(i) for i in range(len(self.mesh.N))]
            self.u = np.zeros(size)
```

8.15. Class Solver

With the `Mesh` and `Function` classes in place, we can rewrite the `solver` function, but we make it a method in class `Solver`:

```
class Solver(Parameters):
    """
    Numerical parameters for solving the wave equation
    u_tt = (c**2*u_x)_x + f(x,t) with t in [0,T] and
    x in (0,L). The problem definition is implied by
    the method of manufactured solution, choosing
    u(x,t)=x(L-x)(1+t/2) as our solution. This solution
    should be exactly reproduced, provided c is const.
    We simulate in [0, L/2] and apply a symmetry condition
    at the end x=L/2.
    """

    def __init__(self, problem):
        self.problem = problem
        self.prm = dict(C=0.75, Nx=3, stability_safety_factor=1.0)
        self.type = dict(C=float, Nx=int, stability_safety_factor=float)
        self.help = dict(
            C="Courant number",
            Nx="No of spatial mesh points",
            stability_safety_factor="stability factor",
        )

        from UniformFDMesh import Function, Mesh

        L_end = self.problem["L"]
        dx = (L_end / 2) / float(self["Nx"])
        t_interval = self.problem["T"]
        dt = dx * self["stability_safety_factor"] * self["C"] / float(self.problem["c"])
        self.m = Mesh(
            L=[0, L_end / 2], d=[dx], Nt=int(round(t_interval / float(dt))), T=t_interval
        )
        self.f = Function(self.m, num_comp=1, space_only=False)

    def solve(self, user_action=None, version="scalar"):
        L, c, T = self.problem[["L", "c", "T"]]
        L = L / 2  # compute with half the domain only (symmetry)
        C, Nx, stability_safety_factor = self[["C", "Nx", "stability_safety_factor"]]
        dx = self.m.d[0]
        I = self.problem.I
        V = self.problem.V
        f = self.problem.f
        U_0 = self.problem.U_0
        U_L = self.problem.U_L
        Nt = self.m.Nt
        t = np.linspace(0, T, Nt + 1)  # Mesh points in time
        x = np.linspace(0, L, Nx + 1)  # Mesh points in space
```

```
        dx = x[1] - x[0]
        dt = t[1] - t[0]

        if isinstance(c, (float, int)):
            c = np.zeros(x.shape) + c
        elif callable(c):
            c_ = np.zeros(x.shape)
            for i in range(Nx + 1):
                c_[i] = c(x[i])
            c = c_

        q = c**2
        C2 = (dt / dx) ** 2
        dt2 = dt * dt  # Help variables in the scheme

        if f is None or f == 0:
            f = (
                (lambda x, t: 0)
                if version == "scalar"
                else lambda x, t: np.zeros(x.shape)
            )
        if I is None or I == 0:
            I = (lambda x: 0) if version == "scalar" else lambda x: np.zeros(x.shape)
        if V is None or V == 0:
            V = (lambda x: 0) if version == "scalar" else lambda x: np.zeros(x.shape)
        if U_0 is not None:
            if isinstance(U_0, (float, int)) and U_0 == 0:
                U_0 = lambda t: 0
        if U_L is not None:
            if isinstance(U_L, (float, int)) and U_L == 0:
                U_L = lambda t: 0

        import hashlib
        import inspect

        data = (
            inspect.getsource(I)
            + "_"
            + inspect.getsource(V)
            + "_"
            + inspect.getsource(f)
            + "_"
            + str(c)
            + "_"
            + ("None" if U_0 is None else inspect.getsource(U_0))
            + ("None" if U_L is None else inspect.getsource(U_L))
            + "_"
```

```
            + str(L)
            + str(dt)
            + "_"
            + str(C)
            + "_"
            + str(T)
            + "_"
            + str(stability_safety_factor)
        )
        hashed_input = hashlib.sha1(data).hexdigest()
        if os.path.isfile("." + hashed_input + "_archive.npz"):
            return -1, hashed_input

        u_1 = self.f.u[0, :]
        u = self.f.u[1, :]


        t0 = time.perf_counter()  # CPU time measurement

        Ix = range(0, Nx + 1)
        It = range(0, Nt + 1)

        for i in range(0, Nx + 1):
            u_1[i] = I(x[i])

        if user_action is not None:
            user_action(u_1, x, t, 0)

        for i in Ix[1:-1]:
            u[i] = (
                u_1[i]
                + dt * V(x[i])
                + 0.5
                * C2
                * (
                    0.5 * (q[i] + q[i + 1]) * (u_1[i + 1] - u_1[i])
                    - 0.5 * (q[i] + q[i - 1]) * (u_1[i] - u_1[i - 1])
                )
                + 0.5 * dt2 * f(x[i], t[0])
            )

        i = Ix[0]
        if U_0 is None:
            ip1 = i + 1
            im1 = ip1  # i-1 -> i+1
            u[i] = (
                u_1[i]
```

```
                + dt * V(x[i])
                + 0.5
                * C2
                * (
                    0.5 * (q[i] + q[ip1]) * (u_1[ip1] - u_1[i])
                    - 0.5 * (q[i] + q[im1]) * (u_1[i] - u_1[im1])
                )
                + 0.5 * dt2 * f(x[i], t[0])
            )
        else:
            u[i] = U_0(dt)

        i = Ix[-1]
        if U_L is None:
            im1 = i - 1
            ip1 = im1  # i+1 -> i-1
            u[i] = (
                u_1[i]
                + dt * V(x[i])
                + 0.5
                * C2
                * (
                    0.5 * (q[i] + q[ip1]) * (u_1[ip1] - u_1[i])
                    - 0.5 * (q[i] + q[im1]) * (u_1[i] - u_1[im1])
                )
                + 0.5 * dt2 * f(x[i], t[0])
            )
        else:
            u[i] = U_L(dt)

        if user_action is not None:
            user_action(u, x, t, 1)

        for n in It[1:-1]:
            u_2 = self.f.u[n - 1, :]
            u_1 = self.f.u[n, :]
            u = self.f.u[n + 1, :]

            if version == "scalar":
                for i in Ix[1:-1]:
                    u[i] = (
                        -u_2[i]
                        + 2 * u_1[i]
                        + C2
                        * (
                            0.5 * (q[i] + q[i + 1]) * (u_1[i + 1] - u_1[i])
                            - 0.5 * (q[i] + q[i - 1]) * (u_1[i] - u_1[i - 1])
```

```
                    )
                    + dt2 * f(x[i], t[n])
                )

        elif version == "vectorized":
            u[1:-1] = (
                -u_2[1:-1]
                + 2 * u_1[1:-1]
                + C2
                * (
                    0.5 * (q[1:-1] + q[2:]) * (u_1[2:] - u_1[1:-1])
                    - 0.5 * (q[1:-1] + q[:-2]) * (u_1[1:-1] - u_1[:-2])
                )
                + dt2 * f(x[1:-1], t[n])
            )
        else:
            raise ValueError("version=%s" % version)

        i = Ix[0]
        if U_0 is None:
            ip1 = i + 1
            im1 = ip1
            u[i] = (
                -u_2[i]
                + 2 * u_1[i]
                + C2
                * (
                    0.5 * (q[i] + q[ip1]) * (u_1[ip1] - u_1[i])
                    - 0.5 * (q[i] + q[im1]) * (u_1[i] - u_1[im1])
                )
                + dt2 * f(x[i], t[n])
            )
        else:
            u[i] = U_0(t[n + 1])

        i = Ix[-1]
        if U_L is None:
            im1 = i - 1
            ip1 = im1
            u[i] = (
                -u_2[i]
                + 2 * u_1[i]
                + C2
                * (
                    0.5 * (q[i] + q[ip1]) * (u_1[ip1] - u_1[i])
                    - 0.5 * (q[i] + q[im1]) * (u_1[i] - u_1[im1])
                )
```

```
                    + dt2 * f(x[i], t[n])
                )
            else:
                u[i] = U_L(t[n + 1])

            if user_action is not None:
                if user_action(u, x, t, n + 1):
                    break

        cpu_time = time.perf_counter() - t0
        return cpu_time, hashed_input

    def assert_no_error(self):
        """Run through mesh and check error"""
        Nx = self["Nx"]
        Nt = self.m.Nt
        L, T = self.problem[["L", "T"]]
        L = L / 2  # only half the domain used (symmetry)
        x = np.linspace(0, L, Nx + 1)  # Mesh points in space
        t = np.linspace(0, T, Nt + 1)  # Mesh points in time

        for n in range(len(t)):
            u_e = self.problem.u_exact(x, t[n])
            diff = np.abs(self.f.u[n, :] - u_e).max()
            print("diff:", diff)
            tol = 1e-13
            assert diff < tol
```

Observe that the solutions from all time steps are stored in the mesh function, which allows error assessment (in `assert_no_error`) to take place after all solutions have been found. Of course, in 2D or 3D, such a strategy may place too high demands on available computer memory, in which case intermediate results could be stored on file.

Running `wave1D_oo.py` gives a printout showing that the class-based implementation performs as expected, i.e. that the known exact solution is reproduced (within machine precision).

8.16. Speeding up Cython code

We now consider the `wave2D_u0.py` code for solving the 2D linear wave equation with constant wave velocity and homogeneous Dirichlet boundary conditions $u = 0$. We shall in the present chapter extend this code with computational modules written in other languages than Python. This extended version is called `wave2D_u0_adv.py`.

The `wave2D_u0.py` file contains a `solver` function, which calls an `advance_*` function to advance the numerical scheme one level forward in time. The function `advance_scalar` applies standard Python loops to implement the scheme, while `advance_vectorized` performs corresponding vectorized

arithmetics with array slices. The statements of this solver are explained in Section 2.71, in particular Section 2.72 and Section 2.73.

Although vectorization can bring down the CPU time dramatically compared with scalar code, there is still some factor 5-10 to win in these types of applications by implementing the finite difference scheme in compiled code, typically in Fortran, C, or C++. This can quite easily be done by adding a little extra code to our program. Cython is an extension of Python that offers the easiest way to nail our Python loops in the scalar code down to machine code and achieve the efficiency of C.

Cython can be viewed as an extended Python language where variables are declared with types and where functions are marked to be implemented in C. Migrating Python code to Cython is done by copying the desired code segments to functions (or classes) and placing them in one or more separate files with extension `.pyx`.

8.17. Declaring variables and annotating the code

Our starting point is the plain `advance_scalar` function for a scalar implementation of the updating algorithm for new values $u_{i,j}^{n+1}$:

```
def advance_scalar(u, u_n, u_nm1, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  It = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in It[1:-1]:
            u_xx = u_n[i-1,j] - 2*u_n[i,j] + u_n[i+1,j]
            u_yy = u_n[i,j-1] - 2*u_n[i,j] + u_n[i,j+1]
            u[i,j] = D1*u_n[i,j] - D2*u_nm1[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    j = It[0]
    for i in Ix: u[i,j] = 0
    j = It[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in It: u[i,j] = 0
    i = Ix[-1]
    for j in It: u[i,j] = 0
    return u
```

We simply take a copy of this function and put it in a file `wave2D_u0_loop_cy.pyx`. The relevant Cython implementation arises from declaring variables with types and adding some important annotations to speed up array computing in Cython. Let us first list the complete code in the `.pyx` file:

```
import numpy as np

cimport cython
cimport numpy as np

ctypedef np.float64_t DT    # data type

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef advance(
    np.ndarray[DT, ndim=2, mode='c'] u,
    np.ndarray[DT, ndim=2, mode='c'] u_1,
    np.ndarray[DT, ndim=2, mode='c'] u_2,
    np.ndarray[DT, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):

    cdef:
        int Ix_start = 0
        int It_start = 0
        int Ix_end = u.shape[0]-1
        int It_end = u.shape[1]-1
        int i, j
        double u_xx, u_yy

    for i in range(Ix_start+1, Ix_end):
        for j in range(It_start+1, It_end):
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
    j = It_start
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    j = It_end
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    i = Ix_start
    for j in range(It_start, It_end+1): u[i,j] = 0
    i = Ix_end
    for j in range(It_start, It_end+1): u[i,j] = 0
    return u
```

This example may act as a recipe on how to transform array-intensive code with loops into Cython.

1. Variables are declared with types: for example, `double v` in the argument list instead of just

`v`, and `cdef double v` for a variable `v` in the body of the function. A Python `float` object is declared as `double` for translation to C by Cython, while an `int` object is declared by `int`.
2. Arrays need a comprehensive type declaration involving

- the type `np.ndarray`,
- the data type of the elements, here 64-bit floats, abbreviated as `DT` through `ctypedef np.float64_t DT` (instead of `DT` we could use the full name of the data type: `np.float64_t`, which is a Cython-defined type),
- the dimensions of the array, here `ndim=2` and `ndim=1`,
- specification of contiguous memory for the array (`mode='c'`).

1. Functions declared with `cpdef` are translated to C but are also accessible from Python.
2. In addition to the standard `numpy` import we also need a special Cython import of `numpy`: `cimport numpy as np`, to appear *after* the standard import.
3. By default, array indices are checked to be within their legal limits. To speed up the code one should turn off this feature for a specific function by placing `@cython.boundscheck(False)` above the function header.
4. Also by default, array indices can be negative (counting from the end), but this feature has a performance penalty and is therefore here turned off by writing `@cython.wraparound(False)` right above the function header.
5. The use of index sets `Ix` and `It` in the scalar code cannot be successfully translated to C. One reason is that constructions like `Ix[1:-1]` involve negative indices, and these are now turned off. Another reason is that Cython loops must take the form `for i in xrange` or `for i in range` for being translated into efficient C loops. We have therefore introduced `Ix_start` as `Ix[0]` and `Ix_end` as `Ix[-1]` to hold the start and end of the values of index i. Similar variables are introduced for the j index. A loop `for i in Ix` is with these new variables written as `for i in range(Ix_start, Ix_end+1)`.

> **Array declaration syntax in Cython**
>
> We have used the syntax `np.ndarray[DT, ndim=2, mode='c']` to declare `numpy` arrays in Cython. There is a simpler, alternative syntax, employing typed memory views, where the declaration looks like `double [:,:]`. However, the full support for this functionality is not yet ready, and in this text we use the full array declaration syntax.

8.18. Visual inspection of the C translation

Cython can visually explain how successfully it translated a code from Python to C. The command

```
Terminal> cython -a wave2D_u0_loop_cy.pyx
```

produces an HTML file `wave2D_u0_loop_cy.html`, which can be loaded into a web browser to illustrate which lines of the code that have been translated to C. Figure Figure 8.1 shows the illustrated code. Yellow lines indicate the lines that Cython did not manage to translate to efficient C code and that remain in Python. For the present code we see that Cython is able to translate all the loops with array computing to C, which is our primary goal.

Raw output: wave2D_u0_loop_cy.c

```
1: import numpy as np
2: cimport numpy as np
3: cimport cython
4: ctypedef np.float64_t DT    # data type
5:
6: @cython.boundscheck(False)  # turn off array bounds check
7: @cython.wraparound(False)   # turn off negative indices (u[-1,-1])
8: cpdef advance(
9:     np.ndarray[DT, ndim=2, mode='c'] u,
10:     np.ndarray[DT, ndim=2, mode='c'] u_1,
11:     np.ndarray[DT, ndim=2, mode='c'] u_2,
12:     np.ndarray[DT, ndim=2, mode='c'] f,
13:     double Cx2, double Cy2, double dt2):
14:
15:     cdef int Ix_start = 0
16:     cdef int Iy_start = 0
17:     cdef int Ix_end = u.shape[0]-1
18:     cdef int Iy_end = u.shape[1]-1
19:     cdef int i, j
20:     cdef double u_xx, u_yy
21:
22:     for i in range(Ix_start+1, Ix_end):
23:         for j in range(Iy_start+1, Iy_end):
24:             u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
25:             u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
26:             u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
27:                      Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
28:     # Boundary condition u=0
29:     j = Iy_start
30:     for i in range(Ix_start, Ix_end+1): u[i,j] = 0
31:     j = Iy_end
32:     for i in range(Ix_start, Ix_end+1): u[i,j] = 0
33:     i = Ix_start
34:     for j in range(Iy_start, Iy_end+1): u[i,j] = 0
35:     i = Iy_end
36:     for j in range(Iy_start, Iy_end+1): u[i,j] = 0
37:     return u
```

Figure 8.1.: Visual illustration of Cython's ability to translate Python to C.

You can also inspect the generated C code directly, as it appears in the file `wave2D_u0_loop_cy.c`. Nevertheless, understanding this C code requires some familiarity with writing Python extension modules in C by hand. Deep down in the file we can see in detail how the compute-intensive statements have been translated into some complex C code that is quite different from what a human would write (at least if a direct correspondence to the mathematical notation was intended).

8.19. Building the extension module

Cython code must be translated to C, compiled, and linked to form what is known in the Python world as a *C extension module*. This is usually done by making a `setup.py` script, which is the standard way of building and installing Python software. For an extension module arising from Cython code, the following `setup.py` script is all we need to build and install the module:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

cymodule = 'wave2D_u0_loop_cy'
setup(
  name=cymodule
  ext_modules=[Extension(cymodule, [cymodule + '.pyx'],)],
  cmdclass={'build_ext': build_ext},
)
```

We run the script by

```
Terminal> python setup.py build_ext --inplace
```

The `--inplace` option makes the extension module available in the current directory as the file `wave2D_u0_loop_cy.so`. This file acts as a normal Python module that can be imported and inspected:

```
>>> import wave2D_u0_loop_cy
>>> dir(wave2D_u0_loop_cy)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__test__', 'advance', 'np']
```

The important output from the `dir` function is our Cython function `advance` (the module also features the imported `numpy` module under the name `np` as well as many standard Python objects with double underscores in their names).

The `setup.py` file makes use of the `distutils` package in Python and Cython's extension of this package. These tools know how Python was built on the computer and will use compatible compiler(s) and options when building other code in Cython, C, or C++. Quite some experience with building large program systems is needed to do the build process manually, so using a `setup.py` script is strongly recommended.

Simplified build of a Cython module

When there is no need to link the C code with special libraries, Cython offers a shortcut for generating and importing the extension module:

```
import pyximport; pyximport.install()
```

This makes the `setup.py` script redundant. However, in the `wave2D_u0_adv.py` code we do not use `pyximport` and require an explicit build process of this and many other modules.

8.20. Calling the Cython function from Python

The `wave2D_u0_loop_cy` module contains our `advance` function, which we now may call from the Python program for the wave equation:

```
import wave2D_u0_loop_cy
advance = wave2D_u0_loop_cy.advance
...
for n in It[1:-1]:                    # time loop
    f_a[:,:] = f(xv, yv, t[n])      # precompute, size as u
    u = advance(u, u_n, u_nm1, f_a, x, y, t, Cx2, Cy2, dt2)
```

8.20.1. Efficiency

For a mesh consisting of 120×120 cells, the scalar Python code requires 1370 CPU time units, the vectorized version requires 5.5, while the Cython version requires only 1! For a smaller mesh with 60×60 cells Cython is about 1000 times faster than the scalar Python code, and the vectorized version is about 6 times slower than the Cython version.

Instead of relying on Cython's (excellent) ability to translate Python to C, we can invoke a compiled language directly and write the loops ourselves. Let us start with Fortran 77, because this is a language with more convenient array handling than C (or plain C++), because we can use the same multi-dimensional indices in the Fortran code as in the `numpy` arrays in the Python code, while in C these arrays are one-dimensional and require us to reduce multi-dimensional indices to a single index.

8.21. The Fortran subroutine

We write a Fortran subroutine `advance` in a file wave2D_u0_loop_f77.f for implementing the updating formula (2.111) and setting the solution to zero at the boundaries:

```
subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
      integer Nx, Ny
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx,0:Ny), Cx2, Cy2, dt2
      integer i, j
      real*8 u_xx, u_yy
Cf2py intent(in, out) u

C     Scheme at interior points
      do j = 1, Ny-1
         do i = 1, Nx-1
            u_xx = u_1(i-1,j) - 2*u_1(i,j) + u_1(i+1,j)
            u_yy = u_1(i,j-1) - 2*u_1(i,j) + u_1(i,j+1)
            u(i,j) = 2*u_1(i,j) - u_2(i,j) + Cx2*u_xx + Cy2*u_yy +
     &               dt2*f(i,j)
         end do
      end do

C     Boundary conditions
      j = 0
      do i = 0, Nx
         u(i,j) = 0
      end do
      j = Ny
      do i = 0, Nx
         u(i,j) = 0
      end do
      i = 0
```

```
      do j = 0, Ny
         u(i,j) = 0
      end do
      i = Nx
      do j = 0, Ny
         u(i,j) = 0
      end do
      return
      end
```

This code is plain Fortran 77, except for the special `Cf2py` comment line, which here specifies that `u` is both an input argument *and* an object to be returned from the `advance` routine. Or more precisely, Fortran is not able return an array from a function, but we need a *wrapper code* in C for the Fortran subroutine to enable calling it from Python, and from this wrapper code one can return `u` to the calling Python code.

Tip: Return all computed objects to the calling code

It is not strictly necessary to return `u` to the calling Python code since the `advance` function will modify the elements of `u`, but the convention in Python is to get all output from a function as returned values. That is, the right way of calling the above Fortran subroutine from Python is

```
u = advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2)
```

The less encouraged style, which works and resembles the way the Fortran subroutine is called from Fortran, reads

```
advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2)
```

8.22. Building the Fortran module with f2py

The nice feature of writing loops in Fortran is that, without much effort, the tool `f2py` can produce a C extension module such that we can call the Fortran version of `advance` from Python. The necessary commands to run are

```
Terminal> f2py -m wave2D_u0_loop_f77 -h wave2D_u0_loop_f77.pyf \
          --overwrite-signature wave2D_u0_loop_f77.f
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

The first command asks `f2py` to interpret the Fortran code and make a Fortran 90 specification of the extension module in the file `wave2D_u0_loop_f77.pyf`. The second command makes `f2py` generate all necessary wrapper code, compile our Fortran file and the wrapper code, and finally build the module. The build process takes place in the specified subdirectory `build_f77` so that files

can be inspected if something goes wrong. The option `-DF2PY_REPORT_ON_ARRAY_COPY=1` makes `f2py` write a message for every array that is copied in the communication between Fortran and Python, which is very useful for avoiding unnecessary array copying (see below). The name of the module file is `wave2D_u0_loop_f77.so`, and this file can be imported and inspected as any other Python module:

```
>>> import wave2D_u0_loop_f77
>>> dir(wave2D_u0_loop_f77)
['__doc__', '__file__', '__name__', '__package__',
 '__version__', 'advance']
>>> print wave2D_u0_loop_f77.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py....
Functions:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,
      nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

Examine the doc strings!

Printing the doc strings of the module and its functions is extremely important after having created a module with `f2py`. The reason is that `f2py` makes Python interfaces to the Fortran functions that are different from how the functions are declared in the Fortran code (!). The rationale for this behavior is that `f2py` creates *Pythonic* interfaces such that Fortran routines can be called in the same way as one calls Python functions. Output data from Python functions is always returned to the calling code, but this is technically impossible in Fortran. Also, arrays in Python are passed to Python functions without their dimensions because that information is packed with the array data in the array objects. This is not possible in Fortran, however. Therefore, `f2py` removes array dimensions from the argument list, and `f2py` makes it possible to return objects back to Python.

Let us follow the advice of examining the doc strings and take a close look at the documentation `f2py` has generated for our Fortran `advance` subroutine:

```
>>> print wave2D_u0_loop_f77.advance.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py
Functions:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,
              nx=(shape(u,0)-1),ny=(shape(u,1)-1))
.
advance - Function signature:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,[nx,ny])
Required arguments:
  u : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_n : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_nm1 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  f : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  cx2 : input float
  cy2 : input float
```

```
  dt2 : input float
Optional arguments:
  nx := (shape(u,0)-1) input int
  ny := (shape(u,1)-1) input int
Return objects:
  u : rank-2 array('d') with bounds (nx + 1,ny + 1)
```

Here we see that the `nx` and `ny` parameters declared in Fortran are optional arguments that can be omitted when calling `advance` from Python.

We strongly recommend to print out the documentation of *every* Fortran function to be called from Python and make sure the call syntax is exactly as listed in the documentation.

8.23. How to avoid array copying

Multi-dimensional arrays are stored as a stream of numbers in memory. For a two-dimensional array consisting of rows and columns there are two ways of creating such a stream: *row-major ordering*, which means that rows are stored consecutively in memory, or *column-major ordering*, which means that the columns are stored one after each other. All programming languages inherited from C, including Python, apply the row-major ordering, but Fortran uses column-major storage. Thinking of a two-dimensional array in Python or C as a matrix, it means that Fortran works with the transposed matrix.

Fortunately, `f2py` creates extra code so that accessing `u(i,j)` in the Fortran subroutine corresponds to the element `u[i,j]` in the underlying `numpy` array (without the extra code, `u(i,j)` in Fortran would access `u[j,i]` in the `numpy` array). Technically, `f2py` takes a copy of our `numpy` array and reorders the data before sending the array to Fortran. Such copying can be costly. For 2D wave simulations on a 60×60 grid the overhead of copying is a factor of 5, which means that almost the whole performance gain of Fortran over vectorized `numpy` code is lost!

To avoid having `f2py` to copy arrays with C storage to the corresponding Fortran storage, we declare the arrays with Fortran storage:

```
order = 'Fortran' if version == 'f77' else 'C'
u   = zeros((Nx+1,Ny+1), order=order)   # solution array
u_n = zeros((Nx+1,Ny+1), order=order)   # solution at t-dt
u_nm1 = zeros((Nx+1,Ny+1), order=order)   # solution at t-2*dt
```

In the compile and build step of using `f2py`, it is recommended to add an extra option for making `f2py` report on array copying:

```
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

It can sometimes be a challenge to track down which array that causes a copying. There are two principal reasons for copying array data: either the array does not have Fortran storage or the

element types do not match those declared in the Fortran code. The latter cause is usually effectively eliminated by using `real*8` data in the Fortran code and `float64` (the default `float` type in `numpy`) in the arrays on the Python side. The former reason is more common, and to check whether an array before a Fortran call has the right storage one can print the result of `isfortran(a)`, which is `True` if the array `a` has Fortran storage.

Let us look at an example where we face problems with array storage. A typical problem in the `wave2D_u0.py` code is to set

```
f_a = f(xv, yv, t[n])
```

before the call to the Fortran `advance` routine. This computation creates a new array with C storage. An undesired copy of `f_a` will be produced when sending `f_a` to a Fortran routine. There are two remedies, either direct insertion of data in an array with Fortran storage,

```
f_a = zeros((Nx+1, Ny+1), order='Fortran')
...
f_a[:,:] = f(xv, yv, t[n])
```

or remaking the `f(xv, yv, t[n])` array,

```
f_a = asarray(f(xv, yv, t[n]), order='Fortran')
```

The former remedy is most efficient if the `asarray` operation is to be performed a large number of times.

8.23.1. Efficiency

The efficiency of this Fortran code is very similar to the Cython code. There is usually nothing more to gain, from a computational efficiency point of view, by implementing the *complete* Python program in Fortran or C. That will just be a lot more code for all administering work that is needed in scientific software, especially if we extend our sample program `wave2D_u0.py` to handle a real scientific problem. Then only a small portion will consist of loops with intensive array calculations. These can be migrated to Cython or Fortran as explained, while the rest of the programming can be more conveniently done in Python.

The computationally intensive loops can alternatively be implemented in C code. Just as Fortran calls for care regarding the storage of two-dimensional arrays, working with two-dimensional arrays in C is a bit tricky. The reason is that `numpy` arrays are viewed as one-dimensional arrays when transferred to C, while C programmers will think of `u`, `u_n`, and `u_nm1` as two dimensional arrays and index them like `u[i][j]`. The C code must declare `u` as `double* u` and translate an index pair `[i][j]` to a corresponding single index when `u` is viewed as one-dimensional. This translation requires knowledge of how the numbers in `u` are stored in memory.

8.24. Translating index pairs to single indices

Two-dimensional `numpy` arrays with the default C storage are stored row by row. In general, multi-dimensional arrays with C storage are stored such that the last index has the fastest variation, then the next last index, and so on, ending up with the slowest variation in the first index. For a two-dimensional `u` declared as `zeros((Nx+1,Ny+1))` in Python, the individual elements are stored in the following order:

```
u[0,0], u[0,1], u[0,2], ..., u[0,Ny], u[1,0], u[1,1], ...,
u[1,Ny], u[2,0], ..., u[Nx,0], u[Nx,1], ..., u[Nx, Ny]
```

Viewing `u` as one-dimensional, the index pair (i, j) translates to $i(N_y + 1) + j$. So, where a C programmer would naturally write an index `u[i][j]`, the indexing must read `u[i*(Ny+1) + j]`. This is tedious to write, so it can be handy to define a C macro,

so that we can write `u[idx(i,j)]`, which reads much better and is easier to debug.

> **Be careful with macro definitions**
>
> Macros just perform simple text substitutions: `idx(hello,world)` is expanded to `(hello)*(Ny+1) + world`. The parentheses in `(i)` are essential - using the natural mathematical formula `i*(Ny+1) + j` in the macro definition, `idx(i-1,j)` would expand to `i-1*(Ny+1) + j`, which is the wrong formula. Macros are handy, but require careful use. In C++, inline functions are safer and replace the need for macros.

8.25. The complete C code

The C version of our function `advance` can be coded as follows.

```
void advance(double* u, double* u_1, double* u_2, double* f,
         double Cx2, double Cy2, double dt2, int Nx, int Ny)
{
  int i, j;
  double u_xx, u_yy;
  /* Scheme at interior points */
  for (i=1; i<=Nx-1; i++) {
    for (j=1; j<=Ny-1; j++) {
      u_xx = u_1[idx(i-1,j)] - 2*u_1[idx(i,j)] + u_1[idx(i+1,j)];
      u_yy = u_1[idx(i,j-1)] - 2*u_1[idx(i,j)] + u_1[idx(i,j+1)];
      u[idx(i,j)] = 2*u_1[idx(i,j)] - u_2[idx(i,j)] +
    Cx2*u_xx + Cy2*u_yy + dt2*f[idx(i,j)];
    }
  }
```

```
  /* Boundary conditions */
  j = 0;  for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  j = Ny; for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  i = 0;  for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
  i = Nx; for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
}
```

8.26. The Cython interface file

All the code above appears in the file wave2D_u0_loop_c.c. We need to compile this file together with C wrapper code such that `advance` can be called from Python. Cython can be used to generate appropriate wrapper code. The relevant Cython code for interfacing C is placed in a file with extension `.pyx`. This file, called wave2D_u0_loop_c_cy.pyx, looks like

```
import numpy as np

cimport cython
cimport numpy as np


cdef extern from "wave2D_u0_loop_c.h":
    void advance(double* u, double* u_1, double* u_2, double* f,
                 double Cx2, double Cy2, double dt2,
                 int Nx, int Ny)

@cython.boundscheck(False)
@cython.wraparound(False)
def advance_cwrap(
    np.ndarray[double, ndim=2, mode='c'] u,
    np.ndarray[double, ndim=2, mode='c'] u_1,
    np.ndarray[double, ndim=2, mode='c'] u_2,
    np.ndarray[double, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
    advance(&u[0,0], &u_1[0,0], &u_2[0,0], &f[0,0],
            Cx2, Cy2, dt2,
            u.shape[0]-1, u.shape[1]-1)
    return u
```

We first declare the C functions to be interfaced. These must also appear in a C header file, wave2D_u0_loop_c.h,

```
extern void advance(double* u, double* u_n, double* u_nm1, double* f,
            double Cx2, double Cy2, double dt2,
            int Nx, int Ny);
```

The next step is to write a Cython function with Python objects as arguments. The name `advance` is already used for the C function so the function to be called from Python is named `advance_cwrap`. The contents of this function is simply a call to the `advance` version in C. To this end, the right information from the Python objects must be passed on as arguments to `advance`. Arrays are sent with their C pointers to the first element, obtained in Cython as `&u[0,0]` (the `&` takes the address of a C variable). The `Nx` and `Ny` arguments in `advance` are easily obtained from the shape of the `numpy` array `u`. Finally, `u` must be returned such that we can set `u = advance(...)` in Python.

8.27. Building the extension module

It remains to build the extension module. An appropriate `setup.py` file is

```
from distutils.core import setup
from distutils.extension import Extension

from Cython.Distutils import build_ext

sources = ["wave2D_u0_loop_c.c", "wave2D_u0_loop_c_cy.pyx"]
module = "wave2D_u0_loop_c_cy"
setup(
    name=module,
    ext_modules=[
        Extension(
            module,
            sources,
            libraries=[],  # C libs to link with
        )
    ],
    cmdclass={"build_ext": build_ext},
)
```

All we need to specify is the `.c` file(s) and the `.pyx` interface file. Cython is automatically run to generate the necessary wrapper code. Files are then compiled and linked to an extension module residing in the file `wave2D_u0_loop_c_cy.so`. Here is a session with running `setup.py` and examining the resulting module in Python

```
Terminal> python setup.py build_ext --inplace
Terminal> python
>>> import wave2D_u0_loop_c_cy as m
>>> dir(m)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '__test__', 'advance_cwrap', 'np']
```

The call to the C version of `advance` can go like this in Python:

```
import wave2D_u0_loop_c_cy
advance = wave2D_u0_loop_c_cy.advance_cwrap
...
f_a[:,:] = f(xv, yv, t[n])
u = advance(u, u_n, u_nm1, f_a, Cx2, Cy2, dt2)
```

8.27.1. Efficiency

In this example, the C and Fortran code runs at the same speed, and there are no significant differences in the efficiency of the wrapper code. The overhead implied by the wrapper code is negligible as long as there is little numerical work in the `advance` function, or in other words, that we work with small meshes.

An alternative to using Cython for interfacing C code is to apply `f2py`. The C code is the same, just the details of specifying how it is to be called from Python differ. The `f2py` tool requires the call specification to be a Fortran 90 module defined in a `.pyf` file. This file was automatically generated when we interfaced a Fortran subroutine. With a C function we need to write this module ourselves, or we can use a trick and let `f2py` generate it for us. The trick consists in writing the signature of the C function with Fortran syntax and place it in a Fortran file, here `wave2D_u0_loop_c_f2py_signature.f`:

```
subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
Cf2py intent(c) advance
      integer Nx, Ny, N
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
Cf2py intent(in, out) u
Cf2py intent(c) u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny
      return
      end
```

Note that we need a special `f2py` instruction, through a `Cf2py` comment line, to specify that all the function arguments are C variables. We also need to tell that the function is actually in C: `intent(c) advance`.

Since `f2py` is just concerned with the function signature and not the complete contents of the function body, it can easily generate the Fortran 90 module specification based solely on the signature above:

```
Terminal> f2py -m wave2D_u0_loop_c_f2py \
          -h wave2D_u0_loop_c_f2py.pyf --overwrite-signature \
          wave2D_u0_loop_c_f2py_signature.f
```

The compile and build step is as for the Fortran code, except that we list C files instead of Fortran files:

```
Terminal> f2py -c wave2D_u0_loop_c_f2py.pyf \
          --build-dir tmp_build_c \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_c.c
```

As when interfacing Fortran code with `f2py`, we need to print out the doc string to see the exact call syntax from the Python side. This doc string is identical for the C and Fortran versions of `advance`.

8.28. Migrating loops to C++ via f2py

C++ is a much more versatile language than C or Fortran and has over the last two decades become very popular for numerical computing. Many will therefore prefer to migrate compute-intensive Python code to C++. This is, in principle, easy: just write the desired C++ code and use some tool for interfacing it from Python. A tool like SWIG can interpret the C++ code and generate interfaces for a wide range of languages, including Python, Perl, Ruby, and Java. However, SWIG is a comprehensive tool with a correspondingly steep learning curve. Alternative tools, such as Boost Python, SIP, and Shiboken are similarly comprehensive. Simpler tools include PyBindGen.

A technically much easier way of interfacing C++ code is to drop the possibility to use C++ classes directly from Python, but instead make a C interface to the C++ code. The C interface can be handled by `f2py` as shown in the example with pure C code. Such a solution means that classes in Python and C++ cannot be mixed and that only primitive data types like numbers, strings, and arrays can be transferred between Python and C++. Actually, this is often a very good solution because it forces the C++ code to work on array data, which usually gives faster code than if fancy data structures with classes are used. The arrays coming from Python, and looking like plain C/C++ arrays, can be efficiently wrapped in more user-friendly C++ array classes in the C++ code, if desired.

8.29. Software Engineering with Devito

The previous sections described traditional approaches to migrating Python loops to compiled languages. Devito provides an alternative paradigm: write the mathematics symbolically in Python, and let the framework generate optimized C code automatically.

8.29.1. The Devito Approach

Instead of manually writing C, Cython, or Fortran code, Devito:

1. Accepts symbolic PDE specifications in Python
2. Automatically generates optimized C/C++ code
3. Compiles and caches the generated code
4. Provides OpenMP parallelization and optional GPU support

This eliminates the need for manual low-level coding while achieving competitive performance with hand-tuned implementations.

8.29.2. Project Structure for Devito Applications

A well-organized Devito project follows standard Python package conventions:

```
my_pde_solver/
+-- src/
|   +-- __init__.py
|   +-- solvers/
|   |   +-- __init__.py
|   |   +-- wave.py          # Wave equation solvers
|   |   +-- diffusion.py     # Diffusion equation solvers
|   +-- utils/
|       +-- __init__.py
|       +-- visualization.py  # Plotting utilities
|       +-- convergence.py    # Convergence testing
+-- tests/
|   +-- conftest.py          # Pytest fixtures
|   +-- test_wave.py
|   +-- test_diffusion.py
+-- examples/
|   +-- run_simulation.py
+-- pyproject.toml
+-- README.md
```

8.29.3. Pytest Fixtures for Devito Testing

Devito's Grid and Function objects can be reused across tests using pytest fixtures:

```python
# tests/conftest.py
import pytest
import numpy as np
from devito import Grid, TimeFunction, Function


@pytest.fixture
def grid_1d():
    """Create a standard 1D grid for testing."""
    return Grid(shape=(101,), extent=(1.0,))


@pytest.fixture
def grid_2d():
    """Create a standard 2D grid for testing."""
    return Grid(shape=(101, 101), extent=(1.0, 1.0))
```

```python
@pytest.fixture
def wave_field(grid_2d):
    """Create a TimeFunction for wave equation testing."""
    return TimeFunction(name='u', grid=grid_2d,
                        time_order=2, space_order=4)


@pytest.fixture
def velocity_model(grid_2d):
    """Create a velocity model with constant value."""
    c = Function(name='c', grid=grid_2d)
    c.data[:] = 1500.0  # m/s
    return c
```

Usage in tests:

```python
# tests/test_wave.py
def test_wave_propagation(grid_2d, wave_field, velocity_model):
    """Test that wave equation solver runs without error."""
    from src.solvers.wave import solve_acoustic_wave

    result = solve_acoustic_wave(
        grid=grid_2d,
        u=wave_field,
        c=velocity_model,
        T=0.1,
    )

    assert result is not None
    assert not np.isnan(result.u.data).any()
```

8.29.4. Convergence Testing Pattern

Verifying numerical schemes against manufactured solutions is essential. Here's a reusable pattern:

```python
def convergence_test(solver_func, exact_solution, grid_sizes, **solver_kwargs):
    """
    Run a convergence test for a Devito solver.

    Parameters
    ----------
    solver_func : callable
        Solver function that returns a result with .u attribute
    exact_solution : callable
        Function(x, t) returning exact solution
    grid_sizes : list
```

```
        List of N values to test
    **solver_kwargs : dict
        Additional arguments passed to solver

    Returns
    -------
    rates : list
        Computed convergence rates between successive refinements
    """
    import numpy as np

    errors = []
    dx_values = []

    for N in grid_sizes:
        result = solver_func(Nx=N, **solver_kwargs)

        # Compute error at final time
        x = np.linspace(0, result.L, N + 1)
        u_exact = exact_solution(x, result.t)
        error = np.max(np.abs(result.u - u_exact))

        errors.append(error)
        dx_values.append(result.L / N)

    # Compute convergence rates
    rates = []
    for i in range(len(errors) - 1):
        rate = np.log(errors[i] / errors[i + 1]) / np.log(2)
        rates.append(rate)

    return rates


# Usage in test
def test_diffusion_convergence():
    from src.solvers.diffusion import solve_diffusion

    rates = convergence_test(
        solver_func=solve_diffusion,
        exact_solution=lambda x, t: np.exp(-np.pi**2 * t) * np.sin(np.pi * x),
        grid_sizes=[20, 40, 80, 160],
        T=0.01,
        a=1.0,
    )

    # Expect second-order convergence
```

```
    assert all(r > 1.9 for r in rates), f"Convergence rates {rates} < 2"
```

8.29.5. Performance Profiling with Devito

Devito provides built-in profiling through environment variables:

```
import os

# Enable performance logging
os.environ['DEVITO_LOGGING'] = 'PERF'

# Run your simulation
from src.solvers.wave import solve_acoustic_wave
result = solve_acoustic_wave(...)

# Output will include timing information for each operator
```

For more detailed analysis:

```
from devito import configuration

# Enable detailed profiling
configuration['profiling'] = 'advanced'

# Create and run operator
op = Operator([update_eq])
summary = op.apply(time=nt, dt=dt)

# Access timing information
print(f"Total time: {summary.globals['fdlike'].time:.3f} s")
print(f"GFLOPS: {summary.globals['fdlike'].gflopss:.2f}")
```

8.29.6. Caching and Compilation

Devito caches compiled operators to avoid recompilation:

```
from devito import configuration

# View cache location
print(configuration['cachedir'])

# Force recompilation (useful during development)
configuration['jit-backdoor'] = True
```

For production runs, ensure the cache is preserved between runs to avoid recompilation overhead.

8.29.7. Result Classes for Solver Output

Using dataclasses provides clean interfaces for solver results:

```
from dataclasses import dataclass, field
import numpy as np


@dataclass
class SolverResult:
    """Container for solver output."""
    u: np.ndarray              # Solution at final time
    x: np.ndarray              # Spatial grid
    t: float                   # Final time
    L: float                   # Domain length
    dx: float                  # Grid spacing
    dt: float                  # Time step used
    nsteps: int                # Number of time steps
    u_history: list = field(default_factory=list)  # Optional history
    t_history: list = field(default_factory=list)  # Time points


def solve_with_result(...):
    """Solver that returns a SolverResult."""
    # ... solver code ...

    return SolverResult(
        u=np.array(u.data[0, :]),
        x=x_values,
        t=t_final,
        L=L,
        dx=dx,
        dt=dt,
        nsteps=nt,
    )
```

8.29.8. Comparison with Manual Optimization

The following table compares Devito with manual optimization approaches:

Approach	Development Time	Performance	Portability	Maintainability
Pure Python	Low	Poor	High	High
NumPy vectorized	Medium	Medium	High	Medium
Cython	High	Good	Medium	Low
Fortran/f2py	High	Excellent	Low	Low

Approach	Development Time	Performance	Portability	Maintainability
C/C++	Very High	Excellent	Low	Low
Devito	Low	Excellent	High	High

Devito achieves performance comparable to hand-tuned code while maintaining the simplicity and portability of Python. This makes it an excellent choice for scientific computing projects where both productivity and performance matter.

8.30. Exercise: Explore computational efficiency of numpy.sum versus built-in sum

Using the task of computing the sum of the first `n` integers, we want to compare the efficiency of `numpy.sum` versus Python's built-in function `sum`. Use IPython's `%timeit` functionality to time these two functions applied to three different arguments: `range(n)`, `xrange(n)`, and `arrange(n)`.

Solution

Here are experiments in IPython:

```
In [1]: a = np.arrange(n)

In [2]: %timeit sum(range(n))
10 loops, best of 3: 25 ms per loop

In [3]: %timeit sum(xrange(n))
100 loops, best of 3: 9.91 ms per loop

In [4]: %timeit np.sum(range(n))
10 loops, best of 3: 73.7 ms per loop

In [5]: %timeit np.sum(xrange(n))
10 loops, best of 3: 119 ms per loop

In [6]: %timeit np.sum(a)
1000 loops, best of 3: 630 us per loop

In [7]: %timeit sum(a)
10 loops, best of 3: 95.5 ms per loop
```

We observe that `numpy.sum` applied to a `numpy` array is by far the fastest method. We also see that the plain `sum` function is slow when applied to arrays, but faster than `numpy.sum` when applied to the `range` list or the `xrange` sequence. There is almost a factor of 200 between the best and worst method!

8.31. Exercise: Make an improved `numpy.savez` function

The `numpy.savez` function can save multiple arrays to a zip archive. Unfortunately, if we want to use `savez` in time-dependent problems and call it multiple times (once per time level), each call leads to a separate zip archive. It is more convenient to have all arrays in one archive, which can be read by `numpy.load`. Section Section 8.4 provides a recipe for merging all the individual zip archives into one archive. An alternative is to write a new `savez` function that allows multiple calls and storage into the same archive prior to a final `close` method to close the archive and make it ready for reading. Implement such an improved `savez` function as a class `Savez`.

The class should pass the following unit test:

```
def test_Savez():
    import tempfile, os
    tmp = 'tmp_testarchive'
    database = Savez(tmp)
    for i in range(4):
        array = np.linspace(0, 5+i, 3)
        kwargs = {'myarray_%02d' % i: array}
        database.savez(**kwargs)
    database.close()

    database = np.load(tmp+'.npz')

    expected = {
        'myarray_00': np.array([ 0. ,  2.5,  5. ]),
        'myarray_01': np.array([ 0.,  3.,  6.])
        'myarray_02': np.array([ 0. ,  3.5,  7. ]),
        'myarray_03': np.array([ 0.,  4.,  8.]),
        }
    for name in database:
        computed = database[name]
        diff = np.abs(expected[name] - computed).max()
        assert diff < 1E-13
    database.close
    os.remove(tmp+'.npz')
```

Tip

Study the source code for function `savez` (or more precisely, function `_savez`).

Solution

Here is the code:

```
import numpy as np


class Savez:
    def __init__(self, zipfilename):
        import os
        import sys
        import tempfile
        import zipfile

        if isinstance(zipfilename, str):
            if not zipfilename.endswith(".npz"):
                zipfilename += ".npz"

        # original _savez has no compression
        compression = zipfile.ZIP_STORED

        if sys.version_info >= (2, 5):
            self.zip = zipfile.ZipFile(
                zipfilename, mode="w", allowZip64=True, compression=compression
            )

        # Stage arrays in a temporary file on disk,
        # before writing to zip.
        fd, tmpfile = tempfile.mkstemp(suffix="-numpy.npy")
        os.close(fd)
        self.tmpfile = tmpfile
        self.i = 0  # array counter

    def savez(self, *args, **kwds):
        import os

        import numpy.lib.format as format

        namedict = kwds
        for val in args:
            key = "arr_%d" % self.i
            if key in namedict.keys():
                raise ValueError("Cannot use un-named variables and keyword %s" % key)
            namedict[key] = val
            self.i += 1

        try:
            for key, val in namedict.items():
                fname = key + ".npy"
                fid = open(self.tmpfile, "wb")
                try:
                    format.write_array(fid, np.asanyarray(val))
                    fid.close()
                    fid = None
                    self.zip.write(self.tmpfile, arcname=fname)
                finally:
                    if fid:
```

8.32. Exercise: Visualize the impact of the Courant number

Use the `pulse` function in the `wave1D_dn_vc.py` to simulate a pulse through two media with different wave velocities. The aim is to visualize the impact of the Courant number C on the quality of the solution. Set `slowness_factor=4` and `Nx=100`.

Simulate for $C = 1, 0.9, 0.75$ and make an animation comparing the three curves (use the `animate_archives.py` program to combine the curves and make animations on the screen and video files). Perform the investigations for different types of initial profiles: a Gaussian pulse, a "cosine hat" pulse, half a"cosine hat" pulse, and a plug pulse.

Solution

We make a little Python script for running one "pulse" simulation:

```
import os
import sys

path = os.path.join(
    os.pardir, os.pardir, os.pardir, os.pardir, "wave", "src-wave", "wave1D"
)
sys.path.insert(0, path)
from wave1D_dn_vc import pulse

pulse_tp = sys.argv[1]
C = float(sys.argv[2])
pulse(pulse_tp=pulse_tp, C=C, Nx=100, animate=False, slowness_factor=4)
```

Then we can make another (Bash) script running through the different types of simulations and also making video files:

```
#!/bin/bash
for pulse_tp in gaussian cosinehat half-cosinehat plug; do
    mkdir -p $pulse_tp
    cd $pulse_tp
    for C in 1 0.9 0.75; do
        python ../pulse.py $pulse_tp $C
    done
    cd ..
done
```

Note that we make separate directories for the different type initial profiles.

8.33. Exercise: Visualize the impact of the resolution

We solve the same set of problems as in Exercise Section 8.32, except that we now fix $C = 1$ and instead study the impact of Δt and Δx by varying the `Nx` parameter: 20, 40, 160. Make animations comparing three such curves.

Axelsson, O. 1996. *Iterative Solution Methods*. Cambridge University Press.

Barrett, R., M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Second. SIAM. http://www.netlib.org/linalg/html_templates/Templates.html.

Duran, D. 2010. *Numerical Methods for Fluid Dynamics - with Applications to Geophysics*. Second. Springer.

Fletcher, C. A. J. 2013. *Computational Techniques for Fluid Dynamics, Vol. 1: Fundamental and General Techniques*. Second. Springer.

Grief, C., and U. M. Ascher. 2011. *A First Course in Numerical Methods*. Computational Science and Engineering. SIAM.

Hjorth-Jensen, M. 2016. *Computational Physics*. Institute of Physics Publishing. https://www.physics.udel.edu/~bnikolic/teaching/phys660/PDF/computational_physics.pdf.

Kelley, C. T. 1995. *Iterative Methods for Linear and Nonlinear Equations*. SIAM.

Langtangen, H. P. 2016a. *A Primer on Scientific Programming with Python*. Fifth. Texts in Computational Science and Engineering. Springer.

———. 2016b. *Finite Difference Computing with Exponential Decay Models*. Lecture Notes in Computational Science and Engineering. Springer. http://hplgit.github.io/decay-book/doc/web/.

Langtangen, H. P., and G. K. Pedersen. 2016. *Scaling of Differential Equations*. Simula Springer Brief Series. Springer. http://hplgit.github.io/scaling-book/doc/web/.

Lapidus, L., and G. F. Pinder. 1982. *Numerical Solution of Partial Differential Equations in Science and Engineering*. Wiley.

LeVeque, R. 2007. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM.

Rannacher, R. 1984. "Finite Element Solution of Diffusion Problems with Irregular Data." *Numerische Mathematik* 43: 309–27.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems*. Second. SIAM. http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

Strikwerda, J. 2007. *Numerical Solution of Partial Differential Equations in Science and Engineering*. Second. SIAM.