Finite Difference Computing with Partial Differential Equations

Hans Petter Langtangen[1,2]

Svein Linge[3,1]

[1]Center for Biomedical Computing, Simula Research Laboratory

[2]Department of Informatics, University of Oslo

[3]Department of Process, Energy and Environmental Technology, Telemark University College

This easy-to-read book introduces the basics of solving partial differential equations by finite difference methods. The emphasis is on constructing finite difference schemes, formulating algorithms, implementing algorithms, verifying implementations, analyzing the physical behavior of the numerical solutions, and applying the methods and software to solve problems from physics and biology.

Oct 21, 2015

Preface

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 the 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.

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.* [4, p. 86] 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" [1]. 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 has 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).

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 problem to the symbol `u` that is naturally used 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.

Independent chapters. Most book authors are careful with avoiding repetitions of material. The chapters in this book, however, contain some overlap, because I want the chapters to 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 the most important information is explicitly stated, even if it was already met in another chapter.

Acknowledgments. The author is particularly thankful to all the very detailed criticism of the text provided by Professor Svein Linge. 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.

Contents

List of Exercises, Problems, and Projects

Vibration ODEs

1

Vibration problems lead to differential equations with solutions that oscillate in time, typically in a damped or undamped sinusoidal fashion. Such solutions put certain demands on the numerical methods compared to other phenomena whose solutions are monotone or very smooth. Both the frequency and amplitude of the oscillations need to be accurately handled by the numerical schemes. Most of the reasoning and specific building blocks introduced in the forthcoming text can be reused to construct sound methods for partial differential equations of wave nature in multiple spatial dimensions.

hpl 1: Need to discuss errors also for the damped and nonlinear models. At least the frequency errors must be illustrated here as well and investigated numerically, either in text or exercises.

1.1 Finite difference discretization

Many of the numerical challenges faced when computing oscillatory solutions to ODEs and PDEs can be captured by the very simple ODE $u'' + u = 0$. This ODE is thus chosen as our starting point for method development, implementation, and analysis.

1.1.1 A basic model for vibrations

A system that vibrates without damping and external forcing can be described by the ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,\ t \in (0, T]\,. \tag{1.1}$$

Here, ω and I are given constants. The exact solution of (1.1) is

$$u(t) = I\cos(\omega t)\,. \tag{1.2}$$

That is, u oscillates with constant amplitude I and angular frequency ω. The corresponding period of oscillations (i.e., the time between two neighboring peaks in the cosine function) is $P = 2\pi/\omega$. The number of periods per second is $f = \omega/(2\pi)$ and measured in the unit Hz. Both f and ω are referred to as frequency, but ω is more precisely named *angular frequency*, measured in rad/s.

In vibrating mechanical systems modeled by (1.1), $u(t)$ very often represents a position or a displacement of a particular point in the system. The derivative $u'(t)$ then has the interpretation of velocity, and $u''(t)$ is the associated acceleration. The model (1.1) is not only applicable to vibrating mechanical systems, but also to oscillations in electrical circuits.

1.1.2 A centered finite difference scheme

To formulate a finite difference method for the model problem (1.1) we follow the four steps explained in Section 1.1.2 in [1].

Step 1: Discretizing the domain. The domain is discretized by introducing a uniformly partitioned time mesh. The points in the mesh are $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, where $\Delta t = T/N_t$ is the constant length of the time steps. We introduce a mesh function u^n for $n = 0, 1, \ldots, N_t$, which approximates the exact solution at the mesh points. The mesh function will be computed from algebraic equations derived from the differential equation problem.

Step 2: Fulfilling the equation at discrete time points. The ODE is to be satisfied at each mesh point:

$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \ldots, N_t\,. \tag{1.3}$$

Step 3: Replacing derivatives by finite differences. The derivative $u''(t_n)$ is to be replaced by a finite difference approximation. A common second-order accurate approximation to the second-order derivative is

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}. \tag{1.4}$$

Inserting (1.4) in (1.3) yields

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n. \tag{1.5}$$

We also need to replace the derivative in the initial condition by a finite difference. Here we choose a centered difference, whose accuracy is similar to the centered difference we used for u'':

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0. \tag{1.6}$$

Step 4: Formulating a recursive algorithm. To formulate the computational algorithm, we assume that we have already computed u^{n-1} and u^n such that u^{n+1} is the unknown value, which we can readily solve for:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n. \tag{1.7}$$

The computational algorithm is simply to apply (1.7) successively for $n = 1, 2, \ldots, N_t - 1$. This numerical scheme sometimes goes under the name Störmer's method or Verlet integration.

Computing the first step. We observe that (1.7) cannot be used for $n = 0$ since the computation of u^1 then involves the undefined value u^{-1} at $t = -\Delta t$. The discretization of the initial condition then comes to our rescue: (1.6) implies $u^{-1} = u^1$ and this relation can be combined with (1.7) for $n = 1$ to yield a value for u^1:

$$u^1 = 2u^0 - u^1 - \Delta t^2\omega^2 u^0,$$

which reduces to

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 u^0. \tag{1.8}$$

Exercise 1.5 asks you to perform an alternative derivation and also to generalize the initial condition to $u'(0) = V \neq 0$.

The computational algorithm. The steps for solving (1.1) becomes

1. $u^0 = I$
2. compute u^1 from (1.8)
3. for $n = 1, 2, \ldots, N_t - 1$:
   a. compute u^{n+1} from (1.7)

The algorithm is more precisely expressed directly in Python:

```
t = linspace(0, T, Nt+1)  # mesh points in time
dt = t[1] - t[0]          # constant time step
u = zeros(Nt+1)           # solution

u[0] = I
u[1] = u[0] - 0.5*dt**2*w**2*u[0]
for n in range(1, Nt):
    u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
```

Remark on using `w` for ω

In the code, we use `w` as the symbol for ω. The reason is that this author prefers `w` for readability and comparison with the mathematical ω instead of the full word `omega` as variable name.

Operator notation. We may write the scheme using a compact difference notation listed in Appendix A.1 (see also Section 1.1.8 in [1]). The difference (1.4) has the operator notation $[D_tD_tu]^n$ such that we can write:

$$[D_tD_tu + \omega^2 u = 0]^n. \tag{1.9}$$

Note that $[D_tD_tu]^n$ means applying a central difference with step $\Delta t/2$ twice:

$$[D_t(D_tu)]^n = \frac{[D_tu]^{n+\frac{1}{2}} - [D_tu]^{n-\frac{1}{2}}}{\Delta t}$$

which is written out as

$$\frac{1}{\Delta t}\left(\frac{u^{n+1} - u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t}\right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}.$$

The discretization of initial conditions can in the operator notation be expressed as

$$[u = I]^0, \quad [D_{2t}u = 0]^0, \tag{1.10}$$

where the operator $[D_{2t}u]^n$ is defined as

$$[D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}. \tag{1.11}$$

1.2 Implementation

1.2.1 Making a solver function

The algorithm from the previous section is readily translated to a complete Python function for computing and returning $u^0, u^1, \ldots, u^{N_t}$ and $t_0, t_1, \ldots, t_{N_t}$, given the input I, ω, Δt, and T:

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

def solver(I, w, dt, T):
    """
    Solve u'' + w**2*u = 0 for t in (0,T], u(0)=I and u'(0)=0,
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
    return u, t
```

We do a simple `from module import *` to make the code as close as possible to MATLAB, although good programming habits would prefix the `numpy` and `matplotlib` calls by (abbreviations of) the module name.

hpl 2: Refer to right section in decay book for prefix discussion.

A function for plotting the numerical and the exact solution is also convenient to have:

```
def u_exact(t, I, w):
    return I*np.cos(w*t)

def visualize(u, t, I, w):
    plt.plot(t, u, 'r--o')
    t_fine = np.linspace(0, t[-1], 1001)  # very fine mesh for u_e
    u_e = u_exact(t_fine, I, w)
    plt.hold('on')
    plt.plot(t_fine, u_e, 'b-')
    plt.legend(['numerical', 'exact'], loc='upper left')
    plt.xlabel('t')
    plt.ylabel('u')
```



```
    dt = t[1] - t[0]
    plt.title('dt=%g' % dt)
    umin = 1.2*u.min();  umax = -umin
    plt.axis([t[0], t[-1], umin, umax])
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
```

A corresponding main program calling these functions for a simulation of a given number of periods (`num_periods`) may take the form

```
I = 1
w = 2*pi
dt = 0.05
num_periods = 5
P = 2*pi/w    #  one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

Adjusting some of the input parameters via the command line can be handy. Here is a code segment using the `ArgumentParser` tool in the `argparse` module to define option value (`-option value`) pairs on the command line:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

Such parsing of the command line is explained in more detailed in Section 5.2 in [1].

hpl 3: Fix reference to web document.

A typical execution goes like

Terminal

```
Terminal> python vib_undamped.py --num_periods 20 --dt 0.1
```

Computing u'. In mechanical vibration applications one is often interested in computing the velocity $v(t) = u'(t)$ after $u(t)$ has been computed. This can be done by a central difference,

$$v(t_n) = u'(t_n) \approx v^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n. \tag{1.12}$$

This formula applies for all inner mesh points, $n = 1, \ldots, N_t - 1$. For $n = 0$, $v(0)$ is given by the initial condition on $u'(0)$, and for $n = N_t$ we can use a one-sided, backward difference:

$$v^n = [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t}.$$

Typical (scalar) code is

```
v = np.zeros_like(u)  # or v = np.zeros(len(u))
# Use central difference for internal points
for i in range(1, len(u)-1):
    v[i] = (u[i+1] - u[i-1])/(2*dt)
# Use initial condition for u'(0) when i=0
v[0] = 0
# Use backward difference at the final mesh point
v[-1] = (u[-1] - u[-2])/dt
```

We can get rid of the loop, which is slow for large N_t, by vectorizing the central difference. The above code segment goes as follows in its vectorized version:

```
v = np.zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*dt)  # central difference
v[0] = 0                           # boundary condition u'(0)
v[-1] = (u[-1] - u[-2])/dt         # backward difference
```

1.2.2 Verification

Manual calculation. The simplest type of verification, which is also instructive for understanding the algorithm, is to compute u^1, u^2, and u^3 with the aid of a calculator and make a function for comparing these results with those from the `solver` function. The `test_three_steps` function in the file `vib_undamped.py` shows the details how we use the hand calculations to test the code:

```
def test_three_steps():
    from math import pi
    I = 1;  w = 2*pi;  dt = 0.1;  T = 1
    u_by_hand = np.array([1.000000000000000,
                          0.802607911978213,
                          0.288358920740053])
    u, t = solver(I, w, dt, T)
    diff = np.abs(u_by_hand - u[:3]).max()
    tol = 1E-14
    assert diff < tol
```

Testing very simple solutions. Constructing test problems where the exact solution is constant or linear helps initial debugging and verification as one expects any reasonable numerical method to reproduce such solutions to machine precision. Second-order accurate methods will often

also reproduce a quadratic solution. Here $[D_t D_t t^2]^n = 2$, which is the exact result. A solution $u = t^2$ leads to $u'' + \omega^2 u = 2 + (\omega t)^2 \neq 0$. We must therefore add a source in the equation: $u'' + \omega^2 u = f$ to allow a solution $u = t^2$ for $f = (\omega t)^2$. By simple insertion we can show that the mesh function $u^n = t_n^2$ is also a solution of the discrete equations. Problem 1.1 asks you to carry out all details to show that linear and quadratic solutions are solutions of the discrete equations. Such results are very useful for debugging and verification. You are strongly encouraged to do this problem now!

Checking convergence rates. Empirical computation of convergence rates yields a good method for verification. The method and its computational are explained in detail in Section 3.1.6 in [1]. Readers not familiar with the concept should look up this reference before proceeding.

In the present problem, computing convergence rates means that we must

- perform m simulations with halved time steps: $\Delta t_i = 2^{-i}\Delta t_0$, $i = 0, \ldots, m-1$,
- compute the L^2 norm of the error, $E_i = \sqrt{\Delta t_i \sum_{n=0}^{N_t-1}(u^n - u_{\mathrm{e}}(t_n))^2}$ in each case,
- estimate the convergence rates r_i based on two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C(\Delta t_i)^r$ and $E_{i-1} = C(\Delta t_{i-1})^r$. From these equations it follows that $r = \ln(E_{i-1}/E_i)/\ln(\Delta t_{i-1}/\Delta t_i)$. Since this r will vary with i, we equip it with an index and call it r_{i-1}, where i runs from 1 to $m-1$.

The computed rates $r_0, r_1, \ldots, r_{m-2}$ hopefully converges to a number, which hopefully is 2, the right one, in the present problem. The convergence of the rates demands that the time steps Δt_i are sufficiently small for the error model $E_i = (\Delta t_i)^r$ to be valid.

All the implementational details of computing the sequence $r_0, r_1, \ldots, r_{m-2}$ appear below.

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    from math import pi
    w = 0.35; I = 0.3       # just chosen values
```

```
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = np.sqrt(dt*np.sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [np.log(E_values[i-1]/E_values[i])/
         np.log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r
```

The expected convergence rate is 2, because we have used a second-order finite difference approximations $[D_tD_tu]^n$ to the ODE and a second-order finite difference formula for the initial condition for u'. Other theoretical error measures also points to $r = 2$.

In the present problem, when Δt_0 corresponds to 30 time steps per period, the returned `r` list has all its values equal to 2.00 (if rounded to two decimals). This amazing result means that all Δt_i values are well into the asymptotic regime where the error model $E_i = C(\Delta t_i)^r$ is valid.

We can now construct a test function that computes convergence rates and checks that the final (and usually the best) estimate is sufficiently close to 2. Here, a rough tolerance of 0.1 is enough. This unit test goes like

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver, num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol
```

The complete code appears in the file `vib_undamped.py`.

1.2.3 Scaled model

hpl 4: Need reference to scaling book and maybe also decay book.

It is advantageous to use dimensionless variables in simulations, because fewer parameters need to be set. The present problem is made dimensionless by introducing dimensionless variables $\bar{t} = t/t_c$ and $\bar{u} = u/u_c$, where t_c and u_c are characteristic scales for t and u, respectively. The scaled ODE problem reads

$$\frac{u_c}{t_c^2}\frac{d^2\bar{u}}{d\bar{t}^2} + u_c\bar{u} = 0, \quad u_c\bar{u}(0) = I, \ \frac{u_c}{t_c}\frac{d\bar{u}}{d\bar{t}}(0) = 0\,.$$

A common choice is to take t_c as one period of the oscillations, $t_c = 2\pi/w$, and $u_c = I$. This gives the dimensionless model

$$\frac{d^2\bar{u}}{\bar{t}^2} + 4\pi^2\bar{u} = 0, \quad \bar{u}(0) = 1, \ \bar{u}'(0) = 0\,. \tag{1.13}$$

Observe that there are no physical parameters in (1.13)! We can therefore perform a single numerical simulation $\bar{u}(\bar{t})$ and afterwards recover any $u(t;\omega,I)$ by

$$u(t;\omega,I) = u_c\bar{u}(t/t_c) = I\bar{u}(omegat/(2\pi))\,.$$

We can easily check this assertion: the solution of the scaled problem is $\bar{u}(\bar{t}) = \cos(2\pi\bar{t})$. The formula for u in terms of $\bar{u}$ gives $u = I\cos(\omega t)$, which is nothing but the solution of the original problem with dimensions.

The scaled model can by run by calling `solver(I=1, w=2*pi, dt, T)`. Each period is now 1 and `T` simply counts the number of periods. Choosing `dt` as `1./M` gives `M` time steps per period.

1.3 Long time simulations

Figure 1.1 shows a comparison of the exact and numerical solution for the scaled model (1.13) with $\Delta t = 0.1, 0.05$. From the plot we make the following observations:

- The numerical solution seems to have correct amplitude.
- There is a angular frequency error which is reduced by reducing the time step.
- The total angular frequency error grows with time.

By angular frequency error we mean that the numerical angular frequency differs from the exact ω. This is evident by looking at the peaks of the numerical solution: these have incorrect positions compared with the peaks of the exact cosine solution. The effect can be mathematical expressed by writing the numerical solution as $I\cos\tilde{\omega}t$, where $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quantify this numerical angular frequency $\tilde{\omega}$.

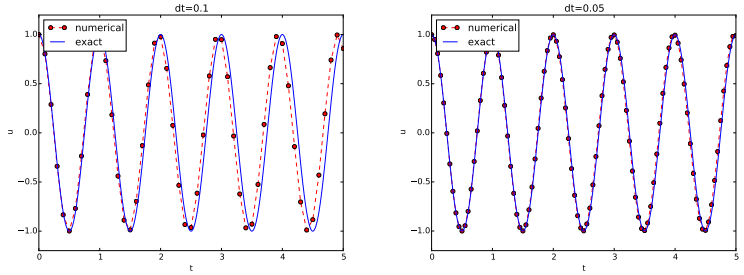


Fig. 1.1 Effect of halving the time step.

1.3.1 Using a moving plot window

In vibration problems it is often of interest to investigate the system's behavior over long time intervals. Errors in the angular frequency accumulate and become more visible as time grows. We can investigate long time series by introducing a moving plot window that can move along with the p most recently computed periods of the solution. The SciTools package contains a convenient tool for this: `MovingPlotWindow`. Typing `pydoc scitools.MovingPlotWindow` shows a demo and a description of its use. The function below utilizes the moving plot window and is in fact called by the `main` function the `vib_undamped` module if the number of periods in the simulation exceeds 10.

```
def visualize_front(u, t, I, w, savefig=False, skip_frames=1):
    """
    Visualize u and the exact solution vs t, using a
    moving plot window and continuous drawing of the
    curves as they evolve in time.
    Makes it easy to plot very long time series.
    Plots are saved to files if savefig is True.
    Only each skip_frames-th plot is saved (e.g., if
    skip_frame=10, only each 10th plot is saved to file;
    this is convenient if plot files corresponding to
    different time steps are to be compared).
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow
    from math import pi

    # Remove all old plot files tmp_*.png
    import glob, os
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    dt = t[1] - t[0]
    plot_manager = MovingPlotWindow(
```



```
        window_width=8*P,
        dt=dt,
        yaxis=[umin, umax],
        mode='continuous drawing')
    frame_counter = 0
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], I*cos(w*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # drop window if savefig
            if savefig and n % skip_frames == 0:
                filename = 'tmp_%04d.png' % frame_counter
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t[n]
                frame_counter += 1
        plot_manager.update(n)
```

We run the scaled problem (the default values for the command-line arguments `-I` and `-w` correspond to the scaled problem) for 40 periods with 20 time steps per period:

Terminal

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

The moving plot window is invoked, and we can follow the numerical and exact solutions as time progresses. From this demo we see that the angular frequency error is small in the beginning, but it becomes more prominent with time. A new run with $\Delta t = 0.1$ (i.e., only 10 time steps per period) clearly shows that the phase errors become significant even earlier in the time series, deteriorating the solution further.

1.3.2 Making animations

Producing standard video formats. The `visualize_front` function stores all the plots in files whose names are numbered: `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so on. From these files we may make a movie. The Flash format is popular,

Terminal

```
Terminal> ffmpeg -r 12 -i tmp_%04d.png -c:v flv movie.flv
```

The `ffmpeg` program can be replaced by the `avconv` program in the above command if desired (but at the time of this writing it seems to be more momentum in the `ffmpeg` project). The `-r` option should come

first and describes the number of frames per second in the movie. The `-i` option describes the name of the plot files. Other formats can be generated by changing the video codec and equipping the video file with the right extension:

Format	Codec and filename
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
WebM	`-c:v libvpx movie.webm`
Ogg	`-c:v libtheora movie.ogg`

The video file can be played by some video player like `vlc`, `mplayer`, `gxine`, or `totem`, e.g.,

Terminal

```
Terminal> vlc movie.webm
```

A web page can also be used to play the movie. Today's standard is to use the HTML5 `video` tag:

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.webm'  type='video/webm; codecs="vp8, vorbis"'>
</video>
```

Modern browsers do not support all of the video formats. MP4 is needed to successfully play the videos on Apple devices that use the Safari browser. WebM is the preferred format for Chrome, Opera, Firefox, and Internet Explorer v9+. Flash was a popular format, but older browsers that required Flash can play MP4. All browsers that work with Ogg can also work with WebM. This means that to have a video work in all browsers, the video should be available in the MP4 and WebM formats. The proper HTML code reads

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.mp4'   type='video/mp4;
 codecs="avc1.42E01E, mp4a.40.2"'>
<source src='movie.webm'  type='video/webm;
 codecs="vp8, vorbis"'>
</video>
```

The MP4 format should appear first to ensure that Apple devices will load the video correctly.

Caution: number the plot files correctly

To ensure that the individual plot frames are shown in correct order, it is important to number the files with zero-padded numbers (0000, 0001, 0002, etc.). The printf format `%04d` specifies an integer in a field of width 4, padded with zeros from the left. A simple Unix wildcard file specification like `tmp_*.png` will then list the frames in the right order. If the numbers in the filenames were not zero-padded, the frame `tmp_11.png` would appear before `tmp_2.png` in the movie.

Paying PNG files in a web browser. The `scitools movie` command can create a movie player for a set of PNG files such that a web browser can be used to watch the movie. This interface has the advantage that the speed of the movie can easily be controlled, a feature that scientists often appreciate. The command for creating an HTML with a player for a set of PNG files `tmp_*.png` goes like

Terminal

```
Terminal> scitools movie output_file=vib.html fps=4 tmp_*.png
```

The `fps` argument controls the speed of the movie ("frames per second").

To watch the movie, load the video file `vib.html` into some browser, e.g.,

Terminal

```
Terminal> google-chrome vib.html  # invoke web page
```

Clicking on `Start movie` to see the result. Moving this movie to some other place requires moving `vib.html` *and all the PNG files* `tmp_*.png`:

Terminal

```
Terminal> mkdir vib_dt0.1
Terminal> mv tmp_*.png vib_dt0.1
Terminal> mv vib.html vib_dt0.1/index.html
```

Making animated GIF files. The `convert` program from the ImageMagick software suite can be used to produce animated GIF files from a set of PNG files:

Terminal

```
Terminal> convert -delay 25 tmp_vib*.png tmp_vib.gif
```

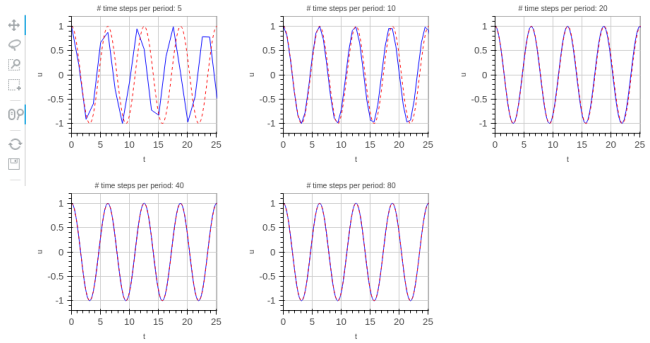
The `-delay` option needs an argument of the delay between each frame, measured in 1/100 s, so 4 frames/s here gives 25/100 s delay. Note, however, that in this particular example with $\Delta t = 0.05$ and 40 periods, making an animated GIF file out of the large number of PNG files is a very heavy process and not considered feasible. Animated GIFs are best suited for animations with not so many frames and where you want to see each frame and play them slowly.

hpl 5: Combine two simulations side by side!

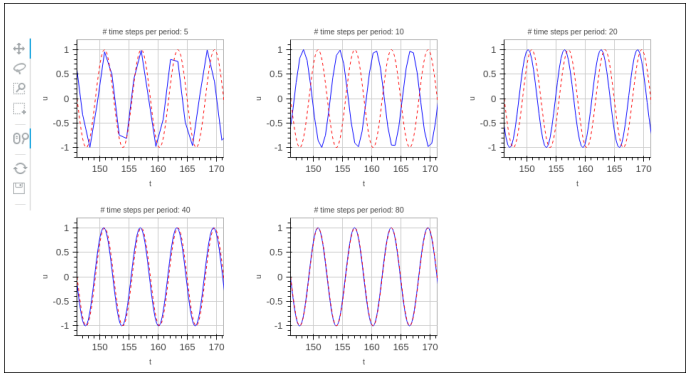
1.3.3 Using Bokeh to compare graphs

Instead of a moving plot frame, one can use tools that allows panning by the mouse. For example, we can show four periods of a signal in a plot and then scroll with the mouse through the rest of the simulation. The Bokeh plotting library offers such tools, but the plot must be displayed in a web browser. The documentation of Bokeh is excellent, so here we just show how the library can be used to compare a set of u curves corresponding to long time simulations.

Imagine we have performed experiments for a set of Δt values. We want each curve, together with the exact solution, to appear in a plot, and then arrange all plots in a grid-like fashion:



Furthermore, we want the axis to couple such that if we move into the future in one plot, all the other plots follows (note the displaced t axes!):



A function for creating a Bokeh plot, given a list of `u` arrays and corresponding `t` arrays, from different simulations, described compactly in a list of strings `legends`, takes the following form:

```
def bokeh_plot(u, t, legends, I, w, t_range, filename):
    """
    Make plots for u vs t using the Bokeh library.
    u and t are lists (several experiments can be compared).
    legens contain legend strings for the various u,t pairs.
    """
    if not isinstance(u, (list,tuple)):
        u = [u]  # wrap in list
    if not isinstance(t, (list,tuple)):
        t = [t]  # wrap in list
    if not isinstance(legends, (list,tuple)):
        legends = [legends]  # wrap in list

    import bokeh.plotting as plt
    plt.output_file(filename, mode='cdn', title='Comparison')
    # Assume that all t arrays have the same range
    t_fine = np.linspace(0, t[0][-1], 1001)  # fine mesh for u_e
    tools = 'pan,wheel_zoom,box_zoom,reset,'\
            'save,box_select,lasso_select'
    u_range = [-1.2*I, 1.2*I]
    font_size = '8pt'
    p = []  # list of plot objects
    # Make the first figure
    p_ = plt.figure(
        width=300, plot_height=250, title=legends[0],
        x_axis_label='t', y_axis_label='u',
        x_range=t_range, y_range=u_range, tools=tools,
        title_text_font_size=font_size)
    p_.xaxis.axis_label_text_font_size=font_size
    p_.yaxis.axis_label_text_font_size=font_size
    p_.line(t[0], u[0], line_color='blue')
    # Add exact solution
    u_e = u_exact(t_fine, I, w)
    p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
    p.append(p_)
    # Make the rest of the figures and attach their axes to
    # the first figure's axes
```

```
    for i in range(1, len(t)):
        p_ = plt.figure(
            width=300, plot_height=250, title=legends[i],
            x_axis_label='t', y_axis_label='u',
            x_range=p[0].x_range, y_range=p[0].y_range, tools=tools,
            title_text_font_size=font_size)
        p_.xaxis.axis_label_text_font_size = font_size
        p_.yaxis.axis_label_text_font_size = font_size
        p_.line(t[i], u[i], line_color='blue')
        p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
        p.append(p_)

    # Arrange all plots in a grid with 3 plots per row
    grid = [[]]
    for i, p_ in enumerate(p):
        grid[-1].append(p_)
        if (i+1) % 3 == 0:
            # New row
            grid.append([])
    plot = plt.gridplot(grid, toolbar_location='left')
    plt.save(plot)
    plt.show(plot)
```

A particular example using the `bokeh_plot` function appears below.

```
def demo_bokeh():
    """Solve a scaled ODE u'' + u = 0."""
    from math import pi
    w = 1.0        # Scaled problem (frequency)
    P = 2*np.pi/w  # Period
    num_steps_per_period = [5, 10, 20, 40, 80]
    T = 40*P       # Simulation time: 40 periods
    u = []         # List of numerical solutions
    t = []         # List of corresponding meshes
    legends = []
    for n in num_steps_per_period:
        dt = P/n
        u_, t_ = solver(I=1, w=w, dt=dt, T=T)
        u.append(u_)
        t.append(t_)
        legends.append('# time steps per period: %d' % n)
    bokeh_plot(u, t, legends, I=1, w=w, t_range=[0, 4*P],
               filename='tmp.html')
```

1.3.4 Using a line-by-line ascii plotter

Plotting functions vertically, line by line, in the terminal window using ascii characters only is a simple, fast, and convenient visualization technique for long time series. Note that the time axis then is positive downwards on the screen. The tool `scitools.avplotter.Plotter` makes it easy to create such plots:


```
def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    from math import pi
    P = 2*pi/w
    umin = 1.2*u.min();  umax = -umin

    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
    for n in range(len(u)):
        print p.plot(t[n], u[n], I*cos(w*t[n])), \
              '%.1f' % (t[n]/P)
        time.sleep(1/float(fps))
```

The call `p.plot` returns a line of text, with the t axis marked and a symbol `+` for the first function (`u`) and `o` for the second function (the exact solution). Here we append to this text a time counter reflecting how many periods the current time point corresponds to. A typical output ($\omega = 2\pi$, $\Delta t = 0.05$) looks like this:

```
                              |                       o+      14.0
                              |                      + o      14.0
                              |                  +    o      14.1
                              |              +     o          14.1
                              |     +         o               14.2
                             +|       o                      14.2
                      +       |                               14.2
                +       o     |                               14.3
           +     o            |                               14.4
         +   o                |                               14.4
        +o                    |                               14.5
        o +                   |                               14.5
         o    +               |                               14.6
             o      +         |                               14.6
                  o       +   |                               14.7
                        o     | +                             14.7
                              |       +                       14.8
                              |      o       +                14.8
                              |              o     +          14.9
                              |                   o   +       14.9
                              |                       o+      15.0
```

1.3.5 Empirical analysis of the solution

For oscillating functions like those in Figure 1.1 we may compute the amplitude and frequency (or period) empirically. That is, we run through the discrete solution points (t_n, u_n) and find all maxima and minima points. The distance between two consecutive maxima (or minima) points

can be used as estimate of the local period, while half the difference between the u value at a maximum and a nearby minimum gives an estimate of the local amplitude.

The local maxima are the points where

$$u^{n-1} < u^n > u^{n+1}, \quad n = 1, \ldots, N_t - 1, \tag{1.14}$$

and the local minima are recognized by

$$u^{n-1} > u^n < u^{n+1}, \quad n = 1, \ldots, N_t - 1\,. \tag{1.15}$$

In computer code this becomes

```
def minmax(t, u):
    minima = []; maxima = []
    for n in range(1, len(u)-1, 1):
        if u[n-1] > u[n] < u[n+1]:
            minima.append((t[n], u[n]))
        if u[n-1] < u[n] > u[n+1]:
            maxima.append((t[n], u[n]))
    return minima, maxima
```

Note that the two returned objects are lists of tuples.

Let (t_i, e_i), $i = 0, \ldots, M-1$, be the sequence of all the M maxima points, where t_i is the time value and e_i the corresponding u value. The local period can be defined as $p_i = t_{i+1} - t_i$. With Python syntax this reads

```
def periods(maxima):
    p = [extrema[n][0] - maxima[n-1][0]
         for n in range(1, len(maxima))]
    return np.array(p)
```

The list `p` created by a list comprehension is converted to an array since we probably want to compute with it, e.g., find the corresponding frequencies `2*pi/p`.

Having the minima and the maxima, the local amplitude can be calculated as the difference between two neighboring minimum and maximum points:

```
def amplitudes(minima, maxima):
    a = [(abs(maxima[n][1] - minima[n][1]))/2.0
         for n in range(min(len(minima),len(maxima)))]
    return np.array(a)
```

The code segments are found in the file `vib_empirical_analysis.py`.

Since `a[i]` and `p[i]` correspond to the i-th amplitude estimate and the i-th period estimate, respectively, it is most convenient to visualize

the `a` and `p` values with the index `i` on the horizontal axis. (There is no unique time point associated with either of these estimate since values at two different time points were used in the computations.)

In the analysis of very long time series, it is advantageous to compute and plot `p` and `a` instead of u to get an impression of the development of the oscillations. Let us do this for the scaled problem and $\Delta t = 0.1, 0.05, 0.01$. A ready-made function

```
plot_empirical_freq_and_amplitude(u, t, I, w)
```

computes the empirical amplitudes and periods, and creates a plot where the amplitudes and angular frequencies are visualized together with the exact amplitude `I` and the exact angular frequency `w`. We can make a little program for creating the plot:

```
from vib_undamped import solver, plot_empirical_freq_and_amplitude
from math import pi
dt_values = [0.1, 0.05, 0.01]
u_cases = []
t_cases = []
for dt in dt_values:
    # Simulate scaled problem for 40 periods
    u, t = solver(I=1, w=2*pi, dt=dt, T=40)
    u_cases.append(u)
    t_cases.append(t)
plot_empirical_freq_and_amplitude(u_cases, t_cases, I=1, w=2*pi)
```

Figure 1.2 shows the result: we clearly see that lowering Δt improves the angular frequency significantly, while the amplitude seems to be more accurate. The lines with $\Delta t = 0.01$, corresponding to 100 steps per period, can hardly be distinguished from the exact values. The next section shows how we can get mathematical insight into why amplitudes are good and frequencies are more inaccurate.

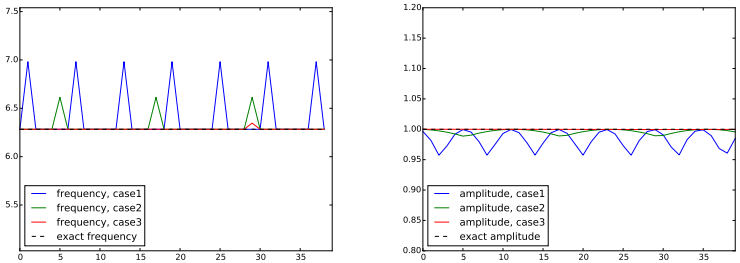


Fig. 1.2 Empirical amplitude and angular frequency for three cases of time steps.

1.4 Analysis of the numerical scheme

1.4.1 Deriving a solution of the numerical scheme

After having seen the phase error grow with time in the previous section, we shall now quantify this error through mathematical analysis. The key tool in the analysis will be to establish an exact solution of the discrete equations. The difference equation (1.7) has constant coefficients and is homogeneous. Such equations are known to have solutions on the form $u^n = CA^n$, where A is some number to be determined from the difference equation and C is found as the initial condition ($C = I$). Recall that n in u^n is a superscript labeling the time level, while n in A^n is an exponent.

With oscillating functions as solutions, the algebra will be considerably simplified if we seek an A on the form

$$A = e^{i\tilde{\omega}\Delta t},$$

and solve for the numerical frequency $\tilde{\omega}$ rather than A. Note that $i = \sqrt{-1}$ is the imaginary unit. (Using a complex exponential function gives simpler arithmetics than working with a sine or cosine function.) We have

$$A^n = e^{i\tilde{\omega}\Delta t\, n} = e^{i\tilde{\omega}t} = \cos(\tilde{\omega}t) + i\sin(\tilde{\omega}t)\,.$$

The physically relevant numerical solution can be taken as the real part of this complex expression.

The calculations go as

$$\begin{aligned}
[D_tD_tu]^n &= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= I\frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\
&= \frac{I}{\Delta t^2}(e^{i\tilde{\omega}(t+\Delta t)} - 2e^{i\tilde{\omega}t} + e^{i\tilde{\omega}(t-\Delta t)}) \\
&= Ie^{i\tilde{\omega}t}\frac{1}{\Delta t^2}\left(e^{i\tilde{\omega}\Delta t} + e^{i\tilde{\omega}(-\Delta t)} - 2\right) \\
&= Ie^{i\tilde{\omega}t}\frac{2}{\Delta t^2}\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) \\
&= Ie^{i\tilde{\omega}t}\frac{2}{\Delta t^2}\left(\cos(\tilde{\omega}\Delta t) - 1\right) \\
&= -Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2})
\end{aligned}$$

The last line follows from the relation $\cos x - 1 = -2\sin^2(x/2)$ (try `cos(x)-1` in wolframalpha.com to see the formula).

The scheme (1.7) with $u^n = Ie^{i\omega\tilde{\Delta}t\, n}$ inserted now gives

$$-Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 Ie^{i\tilde{\omega}t} = 0, \tag{1.16}$$

which after dividing by $Ie^{i\tilde{\omega}t}$ results in

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2\,. \tag{1.17}$$

The first step in solving for the unknown $\tilde{\omega}$ is

$$\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \left(\frac{\omega\Delta t}{2}\right)^2\,.$$

Then, taking the square root, applying the inverse sine function, and multiplying by $2/\Delta t$, results in

$$\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)\,. \tag{1.18}$$

The first observation of (1.18) tells that there is a phase error since the numerical frequency $\tilde{\omega}$ never equals the exact frequency ω. But how good is the approximation (1.18)? That is, what is the error $\omega - \tilde{\omega}$ or $\tilde{\omega}/\omega$? Taylor series expansion for small Δt may give an expression that is easier to understand than the complicated function in (1.18):

```
>>> from sympy import *
>>> dt, w = symbols('dt w')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> print w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

This means that

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^4)\,. \tag{1.19}$$

The error in the numerical frequency is of second-order in Δt, and the error vanishes as $\Delta t \to 0$. We see that $\tilde{\omega} > \omega$ since the term $\omega^3\Delta t^2/24 > 0$ and this is by far the biggest term in the series expansion for small $\omega\Delta t$. A numerical frequency that is too large gives an oscillating curve that oscillates too fast and therefore "lags behind" the exact oscillations, a feature that can be seen in the left plot in Figure 1.1.

Figure 1.3 plots the discrete frequency (1.18) and its approximation (1.19) for $\omega = 1$ (based on the program `vib_plot_freq.py`). Although $\tilde{\omega}$ is a function of Δt in (1.19), it is misleading to think of Δt as the important discretization parameter. It is the product $\omega\Delta t$ that is the key discretization parameter. This quantity reflects the *number of time steps per period* of the oscillations. To see this, we set $P = N_P\Delta t$, where P is the length of a period, and N_P is the number of time steps during a period. Since P and ω are related by $P = 2\pi/\omega$, we get that $\omega\Delta t = 2\pi/N_P$, which shows that $\omega\Delta t$ is directly related to N_P.

The plot shows that at least $N_P \sim 25 - 30$ points per period are necessary for reasonable accuracy, but this depends on the length of the simulation (T) as the total phase error due to the frequency error grows linearly with time (see Exercise 1.2).

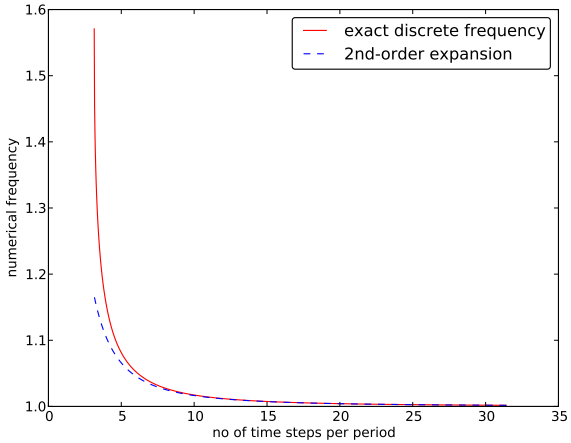


Fig. 1.3 Exact discrete frequency and its second-order series expansion.

1.4.2 Exact discrete solution

Perhaps more important than the $\tilde{\omega} = \omega + \mathcal{O}(\Delta t^2)$ result found above is the fact that we have an exact discrete solution of the problem:

$$u^n = I\cos\left(\tilde{\omega}n\Delta t\right), \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right). \tag{1.20}$$

We can then compute the error mesh function

$$e^n = u_{\mathrm{e}}(t_n) - u^n = I\cos\left(\omega n\Delta t\right) - I\cos\left(\tilde{\omega}n\Delta t\right). \tag{1.21}$$

From the formula $\cos 2x - \cos 2y = -2\sin(x-y)\sin(x+y)$ we can rewrite e^n so the expression is easier to interpret:

$$e^n = -2I\sin\left(t\frac{1}{2}\left(\omega - \tilde{\omega}\right)\right)\sin\left(t\frac{1}{2}\left(\omega + \tilde{\omega}\right)\right). \tag{1.22}$$

The error mesh function is ideal for verification purposes and you are strongly encouraged to make a test based on (1.20) by doing Exercise 1.10.

1.4.3 Convergence

We can use (1.19), (1.21), or (1.22) to show *convergence* of the numerical scheme, i.e., $e^n \to 0$ as $\Delta t \to 0$. We have that

$$\lim_{\Delta t\to 0}\tilde{\omega} = \lim_{\Delta t\to 0}\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) = \omega,$$

by L'Hopital's rule or simply asking `sympy` or WolframAlpha about the limit:

```
>>> import sympy as sym
>>> dt, w = sym.symbols('x w')
>>> sym.limit((2/dt)*sym.asin(w*dt/2), dt, 0, dir='+')
w
```

Also (1.19) can be used to establish this result that $\tilde{\omega} \to \omega$. It then follows from the expression(s) for e^n that $e^n \to 0$.

1.4.4 The global error

To achieve more analytical insight into the nature of the global error, we can Taylor expand the error mesh function (1.21). Since $\tilde{\omega}$ in (1.18) contains Δt in the denominator we use the series expansion for $\tilde{\omega}$ inside the cosine function. A relevant `sympy` session is

```
>>> from sympy import *
>>> dt, w, t = symbols('dt w t')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
```

```
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

Series expansions in `sympy` have the inconvenient `O()` term that prevents further calculations with the series. We can use the `removeO()` command to get rid of the `O()` term:

```
>>> w_tilde_series = w_tilde_series.removeO()
>>> w_tilde_series
dt**2*w**3/24 + w
```

Using this `w_tilde_series` expression for $\tilde{w}$ in (1.21), dropping I (which is a common factor), and performing a series expansion of the error yields

```
>>> error = cos(w*t) - cos(w_tilde_series*t)
>>> error.series(dt, 0, 6)
dt**2*t*w**3*sin(t*w)/24 + dt**4*t**2*w**6*cos(t*w)/1152 + O(dt**6)
```

Since we are mainly interested in the leading-order term in such expansions (the term with lowest power in Δt and goes most slowly to zero), we use the `.as_leading_term(dt)` construction to pick out this term:

```
>>> error.series(dt, 0, 6).as_leading_term(dt)
dt**2*t*w**3*sin(t*w)/24
```

The last result means that the leading order global (true) error at a point t is proportional to $\omega^3 t\Delta t^2$. Now, t is related to Δt through $t = n\Delta t$. The factor $\sin(\omega t)$ can at most be 1, so we use this value to bound the leading-order expression to its maximum value

$$e^n = \frac{1}{24} n\omega^3 \Delta t^3 \,.$$

This is the dominating term of the error *at a point*.

We are interested in the accumulated global error, which can be taken as the ℓ^2 norm of e^n. The norm is simply computed by summing contributions from all mesh points:

$$||e^n||_{\ell^2}^2 = \Delta t \sum_{n=0}^{N_t} \frac{1}{24^2} n^2\omega^6\Delta t^6 = \frac{1}{24^2}\omega^6\Delta t^7 \sum_{n=0}^{N_t} n^2 \,.$$

The sum $\sum_{n=0}^{N_t} n^2$ is approximately equal to $\frac{1}{3}N_t^3$. Replacing N_t by $T/\Delta t$ and taking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{24}\sqrt{\frac{T^3}{3}}\omega^3\Delta t^2 \,.$$

This is our expression for the global (or integrated) error. The main result from this expression is that also the global error is proportional to Δt^2.

1.4.5 Stability

Looking at (1.20), it appears that the numerical solution has constant and correct amplitude, but an error in the angular frequency. A constant amplitude is not necessarily the case, however! To see this, note that if only Δt is large enough, the magnitude of the argument to $\sin^{-1}$ in (1.18) may be larger than 1, i.e., $\omega\Delta t/2 > 1$. In this case, $\sin^{-1}(\omega\Delta t/2)$ has a complex value and therefore $\tilde{\omega}$ becomes complex. Type, for example, `asin(x)` in wolframalpha.com to see basic properties of $\sin^{-1}(x)$).

A complex $\tilde{\omega}$ can be written $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Since $\sin^{-1}(x)$ has a *negative* imaginary part for $x > 1$, $\tilde{\omega}_i < 0$, which means that $e^{i\tilde{\omega}t} = e^{-\tilde{\omega}_i t}e^{i\tilde{\omega}_r t}$ will lead to exponential growth in time because $e^{-\tilde{\omega}_i t}$ with $\tilde{\omega}_i < 0$ has a positive exponent.

Stability criterion

We do not tolerate growth in the amplitude since such growth is not present in the exact solution. Therefore, we must impose a *stability criterion* that the argument in the inverse sine function leads to real and not complex values of $\tilde{\omega}$. The stability criterion reads

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega} \,. \tag{1.23}$$

With $\omega = 2\pi$, $\Delta t > \pi^{-1} = 0.3183098861837907$ will give growing solutions. Figure 1.4 displays what happens when $\Delta t = 0.3184$, which is slightly above the critical value: $\Delta t = \pi^{-1} + 9.01 \cdot 10^{-5}$.

1.4.6 About the accuracy at the stability limit

An interesting question is whether the stability condition $\Delta t < 2/\omega$ is unfortunate, or more precisely: would it be meaningful to take larger time steps to speed up computations? The answer is a clear no. At the stability limit, we have that $\sin^{-1}\omega\Delta t/2 = \sin^{-1} 1 = \pi/2$, and

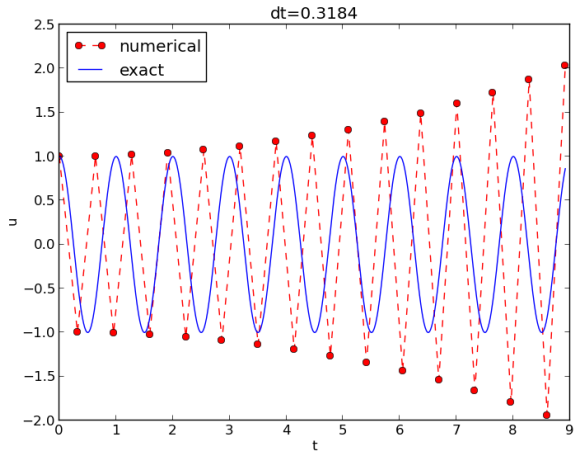


Fig. 1.4 Growing, unstable solution because of a time step slightly beyond the stability limit.

therefore $\tilde{\omega} = \pi/\Delta t$. (Note that the approximate formula (1.19) is very inaccurate for this value of Δt as it predicts $\tilde{\omega} = 2.34/pi$, which is a 25 percent reduction.) The corresponding period of the numerical solution is $\tilde{P} = 2\pi/\tilde{\omega} = 2\Delta t$, which means that there is just one time step Δt between a peak (maximum) and a through (minimum) in the numerical solution. This is the shortest possible wave that can be represented in the mesh! In other words, it is not meaningful to use a larger time step than the stability limit.

Also, the error in angular frequency when $\Delta t = 2/\omega$ is severe: Figure 1.5 shows a comparison of the numerical and analytical solution with $\omega = 2\pi$ and $\Delta t = 2/\omega = \pi^{-1}$. Already after one period, the numerical solution has a through while the exact solution has a peak (!). The error in frequency when Δt is at the stability limit becomes $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. The corresponding error in the period is $P - \tilde{P} \approx 0.36P$. The error after m periods is then $0.36mP$. This error has reached half a period when $m = 1/(2 \cdot 0.36) \approx 1.38$, which theoretically confirms the observations in Figure 1.5 that the numerical solution is a through ahead of a peak already after one and a half period. Consequently, Δt should be chosen much less than the stability limit to achieve meaningful numerical computations.

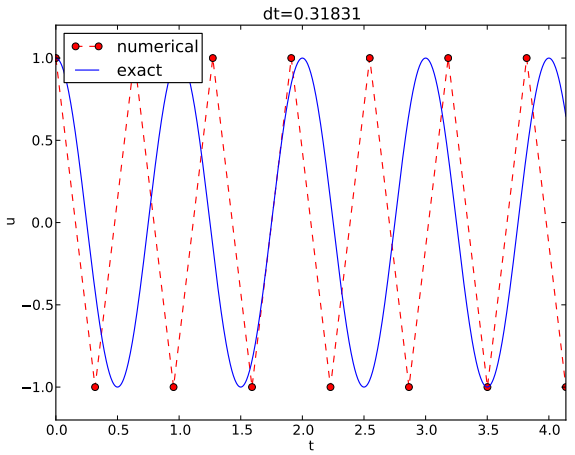


Fig. 1.5 Numerical solution with Δt exactly at the stability limit.

Summary

From the accuracy and stability analysis we can draw three important conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$. The period of oscillations is $P = 2\pi/\omega$, and the number of time steps per period is $N_P = P/\Delta t$. Therefore, $p = \omega\Delta t = 2\pi N_P$, showing that the critical parameter is the number of time steps per period. The smallest possible N_P is 2, showing that $p \in (0, \pi]$.
2. Provided $p \leq 2$, the amplitude of the numerical solution is constant.
3. The ratio of the numerical angular frequency and the exact one is $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$. The error $\frac{1}{24}p^2$ leads to wrongly displaced peaks of the numerical solution, and the error in peak location grows linearly with time (see Exercise 1.2).

1.5 Alternative schemes based on 1st-order equations

A standard technique for solving second-order ODEs is to rewrite them as a system of first-order ODEs and then choose a solution strategy from the vast collection of methods for first-order ODE systems. Given the second-order ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,$$

we introduce the auxiliary variable $v = u'$ and express the ODE problem in terms of first-order derivatives of u and v:

$$u' = v, \tag{1.24}$$
$$v' = -\omega^2 u\,. \tag{1.25}$$

The initial conditions become $u(0) = I$ and $v(0) = 0$.

1.5.1 The Forward Euler scheme

A Forward Euler approximation to our 2×2 system of ODEs (1.24)-(1.25) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{1.26}$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{1.27}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n\,. \tag{1.28}$$

Let us briefly compare this Forward Euler method with the centered difference scheme for the second-order differential equation. We have from (1.27) and (1.28) applied at levels n and $n-1$ that

$$u^{n+1} = u^n + \Delta t v^n = u^n + \Delta t(v^{n-1} - \Delta t \omega^2 u^{n-1}\,.$$

Since from (1.27)

$$v^{n-1} = \frac{1}{\Delta t}(u^n - u^{n-1}),$$

it follows that

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^{n-1},$$

which is very close to the centered difference scheme, but the last term is evaluated at t_{n-1} instead of t_n. Dividing by Δt^2, the left-hand side is an approximation to u'' at t_n, while the right-hand side is sampled at t_{n-1}. All terms should be sampled at the same mesh point, so using $\omega^2 u^{n-1}$ instead of $\omega^2 u^n$ is an inconsistency in the scheme. This inconsistency turns out to be rather crucial for the accuracy of the Forward Euler method applied to vibration problems.

1.5.2 The Backward Euler scheme

A Backward Euler approximation the ODE system is equally easy to write up in the operator notation:

$$[D_t^- u = v]^{n+1}, \tag{1.29}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{1.30}$$

This becomes a coupled system for u^{n+1} and v^{n+1}:

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{1.31}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n\,. \tag{1.32}$$

We can compare (1.31)-(1.32) with the centered scheme (1.7) for the second-order differential equation. To this end, we eliminate v^{n+1} in (1.31) using (1.32) solved with respect to v^{n+1}. Thereafter, we eliminate v^n using (1.31) solved with respect to v^{n+1} and replacing $n+1$ by n. The resulting equation involving only u^{n+1}, u^n, and u^{n-1} can be ordered as

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^{n+1},$$

which has almost the same form as the centered scheme for the second-order differential equation, but the right-hand side is evaluated at u^{n+1} and not u^n. This inconsistent sampling of terms has a dramatic effect on the numerical solution.

1.5.3 The Crank-Nicolson scheme

The Crank-Nicolson scheme takes this form in the operator notation:

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{1.33}$$

$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}}\,. \tag{1.34}$$

Writing the equations out shows that this is also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \tag{1.35}$$

$$v^{n+1} + \frac{1}{2}\Delta t\omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t\omega^2 u^n\,. \tag{1.36}$$

To see the nature of this approximation, and that it is actually very promising, we write the equations as follows

$$u^{n+1} - u^n = \frac{1}{2}\Delta t(v^{n+1} + v^n), \tag{1.37}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t(u^{n+1} + u^n), \tag{1.38}$$

and add the latter at the previous time level as well:

$$v^n = v^{n-1} - \frac{1}{2}\Delta t(u^n + u^{n-1}) \tag{1.39}$$

We can also rewrite (1.37) at the previous time level as

$$v^{n+1} + v^n = \frac{2}{\Delta t}(u^{n+1} - u^n)\,. \tag{1.40}$$

Inserting (1.38) for v^{n+1} in (1.37) and (1.39) for v^n in (1.37) yields after some reordering:

$$u^{n+1} - n^n = \frac{1}{2}(-\frac{1}{2}\Delta t\omega^2(u^{n+1} + 2u^n + u^{n-1}) + v^+ v^{n-1})\,.$$

Now, $v^n + v^{n-1}$ can be eliminated by means of (1.40). The result becomes

$$u^{n+1} - 2u^n + u^{n-1} = \Delta t^2\omega^2\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1})\,. \tag{1.41}$$

We have that

$$\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1}) \approx u^n + \mathcal{O}(\Delta t^2),$$

meaning that (1.41) is an approximation to the centered scheme (1.7) for the second-order ODE where the sampling error in the term $\Delta t^2\omega^2 u^n$ is of the same order as the approximation errors in the finite differences, i.e., $\mathcal{O}(\Delta t^2)$. The Crank-Nicolson scheme written as (1.41) therefore has consistent sampling of all terms at the same time point t_n. The implication is a much better method than the Forward and Backward Euler schemes.

1.5.4 Comparison of schemes

We can easily compare methods like the ones above (and many more!) with the aid of the Odespy package. Below is a sketch of the code.

```
import odespy
import numpy as np

def f(u, t, w=1):
    u, v = u  # u is array of length 2 holding our [u, v]
    return [v, -w**2*u]

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                         num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w  # duration of one period
    dt = P/timesteps_per_period
    Nt = num_periods*timesteps_per_period
    T = Nt*dt
    t_mesh = np.linspace(0, T, Nt+1)

    legends = []
    for solver in solvers:
        solver.set(f_kwargs={'w': w})
        solver.set_initial_condition([I, 0])
        u, t = solver.solve(t_mesh)
```

There is quite some more code dealing with plots also, and we refer to the source file `vib_undamped_odespy.py` for details. Observe that keyword arguments in `f(u,t,w=1)` can be supplied through a solver parameter `f_kwargs` (dictionary of additional keyword arguments to `f`).

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson schemes is done like this:

```
solvers = [
    odespy.ForwardEuler(f),
    # Implicit methods must use Newton solver to converge
    odespy.BackwardEuler(f, nonlinear_solver='Newton'),
    odespy.CrankNicolson(f, nonlinear_solver='Newton'),
    ]
```

The `vib_undamped_odespy.py` program makes two plots of the computed solutions with the various methods in the `solvers` list: one plot with $u(t)$ versus t, and one *phase plane plot* where v is plotted against u. That is, the phase plane plot is the curve $(u(t), v(t))$ parameterized by t. Analytically, $u = I\cos(\omega t)$ and $v = u' = -\omega I \sin(\omega t)$. The exact curve $(u(t), v(t))$ is therefore an ellipse, which often looks like a circle in a plot if the axes are automatically scaled. The important feature, however, is that exact curve $(u(t), v(t))$ is closed and repeats itself for every period. Not all numerical schemes are capable of doing that, meaning that the amplitude instead shrinks or grows with time.

Figure 1.6 show the results. Note that Odespy applies the label MidpointImplicit for what we have specified as `CrankNicolson` in the code (`CrankNicolson` is just a synonym for class `MidpointImplicit` in the Odespy code). The Forward Euler scheme in Figure 1.6 has a pronounced spiral curve, pointing to the fact that the amplitude steadily grows, which is also evident in Figure 1.7. The Backward Euler scheme has a similar feature, except that the spriral goes inward and the amplitude is significantly damped. The changing amplitude and the sprial form decreases with decreasing time step. The Crank-Nicolson scheme looks much more accurate. In fact, these plots tell that the Forward and Backward Euler schemes are not suitable for solving our ODEs with oscillating solutions.

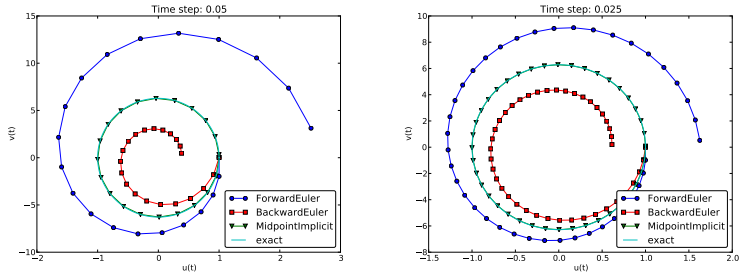


Fig. 1.6 Comparison of classical schemes in the phase plane for two time step values.

1.5.5 Runge-Kutta methods

We may run two popular standard methods for first-order ODEs, the 2nd- and 4th-order Runge-Kutta methods, to see how they perform.

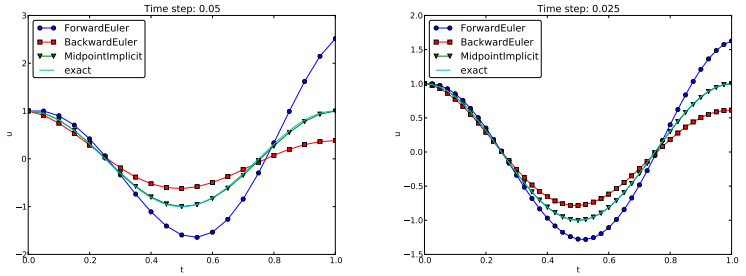


Fig. 1.7 Comparison of solution curves for classical schemes.

Figures 1.8 and 1.9 show the solutions with larger Δt values than what was used in the previous two plots.

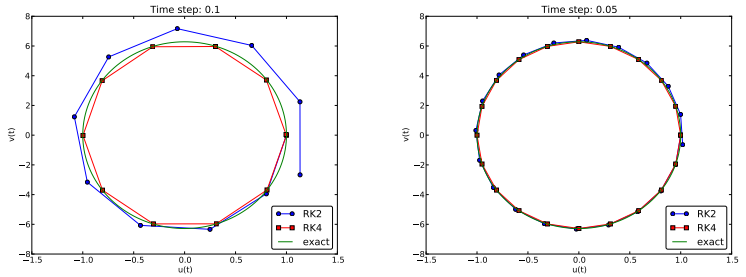


Fig. 1.8 Comparison of Runge-Kutta schemes in the phase plane.

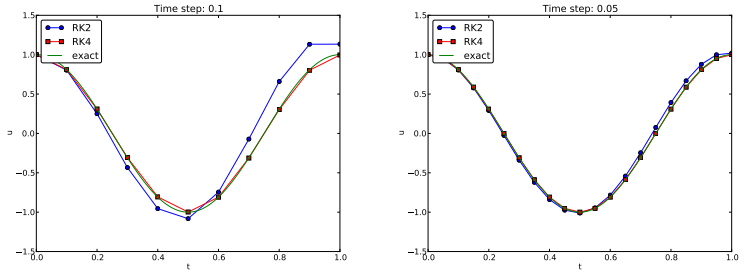


Fig. 1.9 Comparison of Runge-Kutta schemes.

The visual impression is that the 4th-order Runge-Kutta method is very accurate, under all circumstances in these tests, while the 2nd-order scheme suffers from amplitude errors unless the time step is very small.

The corresponding results for the Crank-Nicolson scheme are shown in Figure 1.10. It is clear that the Crank-Nicolson scheme outperforms the 2nd-order Runge-Kutta method. Both schemes have the same order of accuracy $\mathcal{O}(\Delta t^2)$, but their differences in the accuracy that matters in a real physical application is very clearly pronounced in this example. Exercise 1.12 invites you to investigate how the amplitude is computed by a series of famous methods for first-order ODEs.

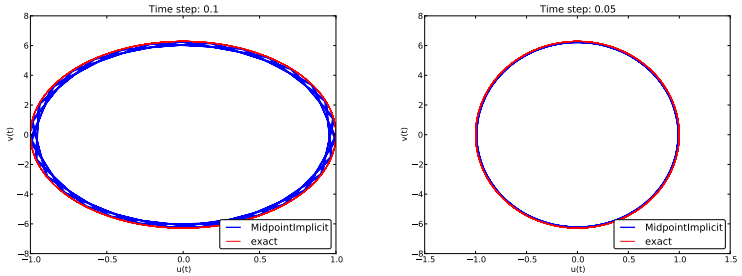


Fig. 1.10 Long-time behavior of the Crank-Nicolson scheme in the phase plane.

1.5.6 Analysis of the Forward Euler scheme

We may try to find exact solutions of the discrete equations (1.27)-(1.28) in the Forward Euler method. An "ansatz" is

$$u^n = IA^n,$$
$$v^n = qIA^n,$$

where q and A are unknown numbers. We could have used a complex exponential form $e^{i\tilde{\omega}n\Delta t}$ since we get oscillatory form, but the oscillations grow in the Forward Euler method, so the numerical frequency $\tilde{\omega}$ will be complex anyway (producing an exponentially growing amplitude). Therefore, it is easier to just work with potentially complex A and q as introduced above.

The Forward Euler scheme leads to

$$A = 1 + \Delta t q,$$
$$A = 1 - \Delta t \omega^2 q^{-1}.$$

We can easily eliminate A, get $q^2 + \omega^2 = 0$, and solve for

$$q = \pm i\omega,$$

which gives

$$A = 1 \pm \Delta t i\omega.$$

We shall take the real part of A^n as the solution. The two values of A are complex conjugates, and the real part of A^n will be the same for both roots. This is easy to realize if we rewrite the complex numbers in polar form, which is also convenient for further analysis and understanding. The polar form $re^{i\theta}$ of a complex number $x + iy$ has $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}(y/x)$. Hence, the polar form of the two values for A become

$$1 \pm \Delta t i\omega = \sqrt{1 + \omega^2\Delta t^2} e^{\pm i \tan^{-1}(\omega\Delta t)}.$$

Now it is very easy to compute A^n:

$$(1 \pm \Delta t i\omega)^n = (1 + \omega^2\Delta t^2)^{n/2} e^{\pm ni \tan^{-1}(\omega\Delta t)}.$$

Since $\cos(\theta n) = \cos(-\theta n)$, the real part of the two numbers become the same. We therefore continue with the solution that has the plus sign.

The general solution is $u^n = CA^n$, where C is a constant determined from the initial condition: $u^0 = C = I$. We have $u^n = IA^n$ and $v^n = qIA^n$. The final solutions are just the real part of the expressions in polar form:

$$u^n = I(1 + \omega^2\Delta t^2)^{n/2} \cos(n \tan^{-1}(\omega\Delta t)), \tag{1.42}$$
$$v^n = -\omega I(1 + \omega^2\Delta t^2)^{n/2} \sin(n \tan^{-1}(\omega\Delta t)). \tag{1.43}$$

The expression $(1 + \omega^2\Delta t^2)^{n/2}$ causes growth of the amplitude, since a number greater than one is raised to a positive exponent $n/2$. We can develop a series expression to better understand the formula for the amplitude. Introducing $p = \omega\Delta t$ as the key variable and using `sympy` gives

```
>>> from sympy import *
>>> p = symbols('p', real=True)
>>> n = symbols('n', integer=True, positive=True)
>>> amplitude = (1 + p**2)**(n/2)
>>> amplitude.series(p, 0, 4)
1 + n*p**2/2 + O(p**4)
```

The amplitude goes like $1 + \frac{1}{2}n\omega^2\Delta t^2$, clearly growing linearly in time (with n).

We can also investigate the error in the angular frequency by a series expansion:

```
>>> n*atan(p).series(p, 0, 4)
n*(p - p**3/3 + O(p**4))
```

This means that the solution for u^n can be written as

$$u^n = (1 + \frac{1}{2}n\omega^2\Delta t^2 + \mathcal{O}(()\Delta t^4))\cos\left(\omega t - \frac{1}{3}\omega t\Delta t^2 + \mathcal{O}(\Delta t^4)\right) .$$

The error in the angular frequency is of the same order as in the scheme (1.7) for the second-order ODE, but error in the amplitude is severe.

1.6 Energy considerations

The observations of various methods in the previous section can be better interpreted if we compute a quantity reflecting the total *energy of the system*. It turns out that this quantity,

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2u^2,$$

is *constant* for all t. Checking that $E(t)$ really remains constant brings evidence that the numerical computations are sound. It turns out that E is proportional to the mechanical energy in the system. Conservation of energy is much used to check numerical simulations.

1.6.1 Derivation of the energy expression

We start out with multiplying

$$u'' + \omega^2u = 0,$$

by u' and integrating from 0 to T:

$$\int_0^T u''u'dt + \int_0^T \omega^2uu'dt = 0 .$$

Observing that

$$u''u' = \frac{d}{dt}\frac{1}{2}(u')^2, \quad uu' = \frac{d}{dt}\frac{1}{2}u^2,$$

we get

$$\int_0^T (\frac{d}{dt}\frac{1}{2}(u')^2 + \frac{d}{dt}\frac{1}{2}\omega^2u^2)dt = E(T) - E(0) = 0,$$

where we have introduced

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2u^2 . \tag{1.44}$$

The important result from this derivation is that the total energy is constant:

$$E(t) = E(0) .$$

$E(t)$ is closely related to the system's energy

The quantity $E(t)$ derived above is physically not the mechanical energy of a vibrating mechanical system, but the energy per unit mass. To see this, we start with Newton's second law $F = ma$ (F is the sum of forces, m is the mass of the system, and a is the acceleration). The displacement u is related to a through $a = u''$. With a spring force as the only force we have $F = -ku$, where k is a spring constant measuring the stiffness of the spring. Newton's second law then implies the differential equation

$$-ku = mu'' \quad \Rightarrow mu'' + ku = 0 .$$

This equation of motion can be turned into an energy balance equation by finding the work done by each term during a time interval $[0, T]$. To this end, we multiply the equation by $du = u'dt$ and integrate:

$$\int_0^T muu'dt + \int_0^T kuu'dt = 0\,.$$

The result is

$$\tilde{E}(t) = E_k(t) + E_p(t) = 0,$$

where

$$E_k(t) = \frac{1}{2}mv^2, \quad v = u', \tag{1.45}$$

is the *kinetic energy* of the system, and

$$E_p(t) = \frac{1}{2}ku^2 \tag{1.46}$$

is the *potential energy*. The sum $\tilde{E}(t)$ is the total mechanical energy. The derivation demonstrates the famous energy principle that, under the right physical circumstances, any change in the kinetic energy is due to a change in potential energy and vice versa. (This principle breaks down when we introduce damping in system, as we do in Section 1.8.)

The equation $mu'' + ku = 0$ can be divided by m and written as $u'' + \omega^2 u = 0$ for $\omega = \sqrt{k/m}$. The energy expression $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ derived earlier is then $\tilde{E}(t)/m$, i.e., mechanical energy per unit mass.

Energy of the exact solution. Analytically, we have $u(t) = I\cos\omega t$, if $u(0) = I$ and $u'(0) = 0$, so we can easily check that the energy evolution and confirm that $E(t)$ is constant:

$$E(t) = \frac{1}{2}I^2(-\omega\sin\omega t)^2 + \frac{1}{2}\omega^2 I^2\cos^2\omega t = \frac{1}{2}\omega^2(\sin^2\omega t + \cos^2\omega t) = \frac{1}{2}\omega^2\,.$$

1.6.2 An error measure based on energy

The constant energy is well expressed by its initial value $E(0)$, so that the error in mechanical energy can be computed as a mesh function by

$$e_E^n = \frac{1}{2}\left(\frac{u^{n+1} - u^{n-1}}{2\Delta t}\right)^2 + \frac{1}{2}\omega^2(u^n)^2 - E(0), \quad n = 1, \ldots, N_t - 1, \tag{1.47}$$

where

$$E(0) = \frac{1}{2}V^2 + \frac{1}{2}\omega^2 I^2,$$

if $u(0) = I$ and $u'(0) = V$. Note that we have used a centered approximation to u': $\boldsymbol{u}'(t_n) \approx [D_{2t}u]^n$.

A useful norm of the mesh function e_E^n for the discrete mechanical energy can be the maximum absolute value of e_E^n:

$$||e_E^n||_{\ell^\infty} = \max_{1\leq n<N_t} |e_E^n|\,.$$

Alternatively, we can compute other norms involving integration over all mesh points, but we are often interested in worst case deviation of the energy, and then the maximum value is of particular relevance.

A vectorized Python implementation takes the form

```
# import numpy as np and compute u, t
dt = t[1]-t[0]
E = 0.5*((u[2:] - u[:-2])/(2*dt))**2 + 0.5*w**2*u[1:-1]**2
E0 = 0.5*V**2 + 0.5**w**2*I**2
e_E = E - E0
e_E_norm = np.abs(e_E).max()
```

The convergence rates of the quantity `e_E_norm` can be used for verification. The value of `e_E_norm` is also useful for comparing schemes through their ability to preserve energy. Below is a table demonstrating the error in total energy for various schemes. We clearly see that the Crank-Nicolson and 4th-order Runge-Kutta schemes are superior to the 2nd-order Runge-Kutta method and better compared to the Forward and Backward Euler schemes.

Method	T	Δt	$\max\|e_E^n\|$
Forward Euler	1	0.05	$1.113 \cdot 10^{2}$
Forward Euler	1	0.025	$3.312 \cdot 10^{1}$
Backward Euler	1	0.05	$1.683 \cdot 10^{1}$
Backward Euler	1	0.025	$1.231 \cdot 10^{1}$
Runge-Kutta 2nd-order	1	0.1	8.401
Runge-Kutta 2nd-order	1	0.05	$9.637 \cdot 10^{-1}$
Crank-Nicolson	1	0.05	$9.389 \cdot 10^{-1}$
Crank-Nicolson	1	0.025	$2.411 \cdot 10^{-1}$
Runge-Kutta 4th-order	1	0.1	2.387
Runge-Kutta 4th-order	1	0.05	$6.476 \cdot 10^{-1}$
Crank-Nicolson	10	0.1	3.389
Crank-Nicolson	10	0.05	$9.389 \cdot 10^{-1}$
Runge-Kutta 4th-order	10	0.1	3.686
Runge-Kutta 4th-order	10	0.05	$6.928 \cdot 10^{-1}$

hpl 6: The error reductions are not directly in accordance with the order of the schemes, probably caused by Δt not being in the asympotic regime.

1.7 The Euler-Cromer method

While the 4th-order Runge-Kutta method and a Crank-Nicolson scheme work well for vibration equation modeled as a first-order ODE system, both were inferior to the straightforward centered difference scheme for the second-order equation $u'' + \omega^2 u = 0$. However, there is a similarly successful scheme available for the first-order system $u' = v$, $v' = -\omega^2 u$, to be presented next.

1.7.1 Forward-backward discretization

The idea is to apply a Forward Euler discretization to the first equation and a Backward Euler discretization to the second. In operator notation this is stated as

$$[D_t^+ u = v]^n, \tag{1.48}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{1.49}$$

We can write out the formulas and collect the unknowns on the left-hand side:

$$u^{n+1} = u^n + \Delta t v^n, \tag{1.50}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}\,. \tag{1.51}$$

We realize that after u^{n+1} has been computed from (1.50), it may be used directly in (1.51) to compute v^{n+1}.

In physics, it is more common to update the v equation first, with a forward difference, and thereafter the u equation, with a backward difference that applies the most recently computed v value:

$$v^{n+1} = v^n - \Delta t \omega^2 u^n, \tag{1.52}$$
$$u^{n+1} = u^n + \Delta t v^{n+1}\,. \tag{1.53}$$

The advantage of ordering the ODEs as in (1.52)-(1.53) becomes evident when consider complicated models. Such models are included if we write our vibration ODE more generally as

$$\ddot{u} + g(u, u', t) = 0\,.$$

We can rewrite this second-order ODE as two first-order ODEs,

$$v' = -g(u, v, t),$$
$$u' = v\,.$$

This rewrite allows the following scheme to be used:

$$v^{n+1} = v^n - \Delta t\, g(u^n, v^n, t),$$
$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,.$$

We realize that the first update works well with any g since old values u^n and v^n are used. Switching the equations would demand u^n+1 and v^{n+1} values in g.

The scheme (1.52)-(1.53) goes under several names: forward-backward scheme, semi-implicit Euler method, semi-explicit Euler, symplectic Euler, Newton-Störmer-Verlet, and Euler-Cromer. We shall stick to the latter name. Since both time discretizations are based on first-order difference approximation, one may think that the scheme is only of first-order, but this is not true: the use of a forward and then a backward difference make errors cancel so that the overall error in the scheme is $\mathcal{O}(\Delta t^2)$. This is explained below.

1.7.2 Equivalence with the scheme for the second-order ODE

We may eliminate the v^n variable from (1.50)-(1.51) or (1.52)-(1.53). The v^{n+1} term in (1.52) can be eliminated from (1.53):

$$u^{n+1} = u^n + \Delta t(v^n - \omega^2 \Delta t^2 u^n)\,. \tag{1.54}$$

The v^n quantity can be expressed by u^n and u^{n-1} using (1.53):

$$v^n = \frac{u^n - u^{n-1}}{\Delta t},$$

and when this is inserted in (1.54) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n, \tag{1.55}$$

which is nothing but the centered scheme (1.7)! The two seemingly different numerical methods are mathematically equivalent. Consequently, the previous analysis of (1.7) also applies to the Euler-Cromer method. In particular, the amplitude is constant, given that the stability criterion is fulfilled, but there is always an angular frequency error (1.19). Exercise 1.17 gives guidance on how to derive the exact discrete solution of the two equations in the Euler-Cromer method.

Although the Euler-Cromer scheme and the method (1.7) are equivalent, there could be differences in the way they handle the initial conditions. Let is look into this topic. The initial condition $u' = 0$ means $u' = v = 0$. From (1.53) we get $v^1 = -\omega^2 u^0$ and $u^1 = u^0 - \omega^2\Delta t^2 u^0$. When using a centered approximation of $u'(0) = 0$ combined with the discretization (1.7) of the second-order ODE, we get $u^1 = u^0 - \frac{1}{2}\omega^2\Delta t^2 u^0$. The difference is $\frac{1}{2}\omega^2\Delta t^2 u^0$, which is of second order in Δt, seemingly consistent with the overall error in the scheme for the differential equation model.

A different view can also be taken. If we approximate $u'(0) = 0$ by a backward difference, $(u^0 - u^{-1})/\Delta t = 0$, we get $u^{-1} = u^0$, and when combined with (1.7), it results in $u^1 = u^0 - \omega^2\Delta t^2 u^0$. This means that the Euler-Cromer method based on (1.53)-(1.52) corresponds to using only a first-order approximation to the initial condition in the method from Section 1.1.2.

Correspondingly, using the formulation (1.50)-(1.51) with $v^n = 0$ leads to $u^1 = u^0$, which can be interpreted as using a forward difference approximation for the initial condition $u'(0) = 0$. Both Euler-Cromer formulations lead to slightly different values for u^1 compared to the method in Section 1.1.2. The error is $\frac{1}{2}\omega^2\Delta t^2 u^0$ and of the same order as the overall scheme.

1.7.3 Implementation

The function below, found in `vib_EulerCromer.py` implements the Euler-Cromer scheme (1.52)-(1.53):


```
import numpy as np

def solver(I, w, dt, T):
    """
    Solve v' = - w**2*u, u'=v for t in (0,T], u(0)=I and v(0)=0,
    by an Euler-Cromer method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    v[0] = 0
    u[0] = I
    for n in range(0, Nt):
        v[n+1] = v[n] - dt*w**2*u[n]
        u[n+1] = u[n] + dt*v[n+1]
    return u, v, t
```

Since the Euler-Cromer scheme is equivalent to the finite difference method for the second-order ODE $u'' + \omega^2 u = 0$ (see Section 1.7.2), the performance of the above `solver` function is the same as for the `solver` function in Section 1.2. The only difference is the formula for the first time step, as discussed above. This deviation in the Euler-Cromer scheme means that the discrete solution listed in Section 1.4.2 is not a solution of the Euler-Cromer scheme!

To verify the implementation of the Euler-Cromer method we can adjust `v[1]` so that the computer-generated values can be compared with the formula (1.20) from in Section 1.4.2. This adjustment is done in an alternative solver function, `solver_ic_fix` in `vib_EulerCromer.py`. Since we now have an exact solution of the discrete equations available, we can write a test function `test_solver` for checking the equality of computed values with the formula (1.20):

```
def test_solver():
    """
    Test solver with fixed initial condition against
    equivalent scheme for the 2nd-order ODE u'' + u = 0.
    """
    I = 1.2; w = 2.0; T = 5
    dt = 2/w  # longest possible time step
    u, v, t = solver_ic_fix(I, w, dt, T)
    from vib_undamped import solver as solver2  # 2nd-order ODE
    u2, t2 = solver2(I, w, dt, T)
    error = np.abs(u - u2).max()
    tol = 1E-14
    assert error < tol
```

Another function, `demo`, visualizes the difference between Euler-Cromer scheme and the scheme (1.7) for the second-oder ODE, arising from the mismatch in the first time level.

hpl 7: Odespy's Euler-Cromer, but it needs more work with the example code.

1.7.4 The velocity Verlet algorithm

Another very popular algorithm for vibration problems $u'' + \omega^2 u = 0$ can be derived as follows. First, we step u forward from t_n to t_{n+1} using a three-term Taylor series,

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 .$$

Using $u' = v$ and $u'' = -\omega^2 u$, we get the updating formula

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n .$$

Second, the first-order equation for v,

$$v' = -\omega^2 u,$$

is discretized by a centered difference in a Crank-Nicolson fashion at $t_{n+\frac{1}{2}}$:

$$\frac{v^{n+1} - v^n}{\Delta t} = -\omega^2\frac{1}{2}(u^n + u^{n+1}) .$$

To summarize, we have the scheme

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n \tag{1.56}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t\omega^2(u^n + u^{n+1}), \tag{1.57}$$

known as the *velocity Verlet* algorithm. Observe that this scheme is explicit since u^{n+1} in (1.57) is already computed from (1.56).

The algorithm can be straightforwardly implemented as shown below (the code appears in the file `vib_undamped_velocity_Verlet.py`).

```
from vib_undamped import convergence_rates, main

def solver(I, w, dt, T, return_v=False):
    """
    Solve u'=v, v'=-w**2*u for t in (0,T], u(0)=I and v(0)=0,
    by the velocity Verlet method with time step dt.
    """
```



```
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    v[0] = 0
    for n in range(Nt):
        u[n+1] = u[n] + v[n]*dt - 0.5*dt**2*w**2*u[n]
        v[n+1] = v[n] - 0.5*dt*w**2*(u[n] + u[n+1])
    if return_v:
        return u, v, t
    else:
        # Return just u and t as in the vib_undamped.py's solver
        return u, t
```

We provide the option that this `solver` function returns the same data as the `solver` function from Section 1.2.1 (if `return_v` is `False`), but we may return `v` along with `u` and `t`.

The error in the Taylor series expansion behind (1.56) is $\mathcal{O}(\Delta t^3)$, while the error in the central difference for v is $\mathcal{O}(\Delta t^2)$. The overall error is then no better than $\mathcal{O}(\Delta t^2)$, which can be verified empirically using the `convergence_rates` function from 1.2.2:

```
>>> import vib_undamped_velocity_Verlet as m
>>> m.convergence_rates(4, solver_function=m.solver)
[2.0036366687367346, 2.0009497328124835, 2.000240105995295]
```

1.8 Generalization: damping, nonlinear spring, and external excitation

We shall now generalize the simple model problem from Section 1.1 to include a possibly nonlinear damping term $f(u')$, a possibly nonlinear spring (or restoring) force $s(u)$, and some external excitation $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T] . \tag{1.58}$$

We have also included a possibly nonzero initial value of $u'(0)$. The parameters m, $f(u')$, $s(u)$, $F(t)$, I, V, and T are input data.

There are two main types of damping (friction) forces: linear $f(u') = bu$, or quadratic $f(u') = bu'|u'|$. Spring systems often feature linear damping, while air resistance usually gives rise to quadratic damping. Spring forces are often linear: $s(u) = cu$, but nonlinear versions are also common, the

most famous is the gravity force on a pendulum that acts as a spring with $s(u) \sim \sin(u)$.

1.8.1 A centered scheme for linear damping

Sampling (1.58) at a mesh point t_n, replacing $u''(t_n)$ by $[D_tD_tu]^n$, and $u'(t_n)$ by $[D_{2t}u]^n$ results in the discretization

$$[mD_tD_tu + f(D_{2t}u) + s(u) = F]^n, \tag{1.59}$$

which written out means

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n, \tag{1.60}$$

where F^n as usual means $F(t)$ evaluated at $t = t_n$. Solving (1.60) with respect to the unknown u^{n+1} gives a problem: the u^{n+1} inside the f function makes the equation *nonlinear* unless $f(u')$ is a linear function, $f(u') = bu'$. For now we shall assume that f is linear in u'. Then

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + s(u^n) = F^n, \tag{1.61}$$

which gives an explicit formula for u at each new time level:

$$u^{n+1} = (2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n)))(m + \frac{b}{2}\Delta t)^{-1}. \tag{1.62}$$

For the first time step we need to discretize $u'(0) = V$ as $[D_{2t}u = V]^0$ and combine with (1.62) for $n = 0$. The discretized initial condition leads to

$$u^{-1} = u^1 - 2\Delta t V, \tag{1.63}$$

which inserted in (1.62) for $n = 0$ gives an equation that can be solved for u^1:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0). \tag{1.64}$$

1.8.2 A centered scheme for quadratic damping

When $f(u') = bu'|u'|$, we get a quadratic equation for u^{n+1} in (1.60). This equation can be straightforwardly solved by the well-known formula for the roots of a quadratic equation. However, we can also avoid the nonlinearity by introducing an approximation with an error of order no higher than what we already have from replacing derivatives with finite differences.

We start with (1.58) and only replace u'' by D_tD_tu, resulting in

$$[mD_tD_tu + bu'|u'| + s(u) = F]^n. \tag{1.65}$$

Here, $u'|u'|$ is to be computed at time t_n. The idea is now to introduce a *geometric mean*, defined by

$$(w^2)^n \approx w^{n-\frac{1}{2}}w^{n+\frac{1}{2}},$$

for some quantity w depending on time. The error in the geometric mean approximation is $\mathcal{O}(\Delta t^2)$, the same as in the approximation $u'' \approx D_tD_tu$. With $w = u'$ it follows that

$$[u'|u'|]^n \approx u'(t_{n+\frac{1}{2}})|u'(t_{n-\frac{1}{2}})|.$$

The next step is to approximate u' at $t_{n\pm1/2}$, and fortunately a centered difference fits perfectly into the formulas since it involves u values at the mesh points only. With the approximations

$$u'(t_{n+1/2}) \approx [D_tu]^{n+\frac{1}{2}}, \quad u'(t_{n-1/2}) \approx [D_tu]^{n-\frac{1}{2}}, \tag{1.66}$$

we get

$$[u'|u'|]^n \approx [D_tu]^{n+\frac{1}{2}}|[D_tu]^{n-\frac{1}{2}}| = \frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t}. \tag{1.67}$$

The counterpart to (1.60) is then

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t} + s(u^n) = F^n, \tag{1.68}$$

which is linear in the unknown u^{n+1}. Therefore, we can easily solve (1.68) with respect to u^{n+1} and achieve the explicit updating formula

$$u^{n+1} = \left(m + b|u^n - u^{n-1}|\right)^{-1} \times \left(2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))\right). \tag{1.69}$$

In the derivation of a special equation for the first time step we run into some trouble: inserting (1.63) in (1.69) for $n = 0$ results in a complicated nonlinear equation for u^1. By thinking differently about the problem we can easily get away with the nonlinearity again. We have for $n = 0$ that $b[u'|u'|]^0 = bV|V|$. Using this value in (1.65) gives

$$[mD_tD_tu + bV|V| + s(u) = F]^0. \tag{1.70}$$

Writing this equation out and using (1.63) results in the special equation for the first time step:

$$u^1 = u^0 + \Delta tV + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right). \tag{1.71}$$

1.8.3 A forward-backward discretization of the quadratic damping term

The previous section first proposed to discretize the quadratic damping term $|u'|u'$ using centered differences: $[|D_{2t}|D_{2t}u]^n$. As this gives rise to a nonlinearity in u^{n+1}, it was instead proposed to use a geometric mean combined with centered differences. But there are other alternatives. To get rid of the nonlinearity in $[|D_{2t}|D_{2t}u]^n$, one can think differently: apply a backward difference to $|u'|$, such that the term involves known values, and apply a forward difference to u' to make the term linear in the unknown u^{n+1}. With mathematics,

$$[\beta|u'|u']^n \approx \beta|[D_t^-u]^n|[D_t^+u]^n = \beta\left|\frac{u^n - u^{n-1}}{\Delta t}\right|\frac{u^{n+1} - u^n}{\Delta t}. \tag{1.72}$$

The forward and backward differences have both an error proportional to Δt so one may think the discretization above leads to a first-order scheme. However, by looking at the formulas, we realize that the forward-backward differences in (1.72) result in exactly the same scheme as in (1.68) where we used a geometric mean and centered differences and committed errors of size $\mathcal{O}(\Delta t^2)$. Therefore, the forward-backward differences in (1.72) act in a symmetric way and actually produce a second-order accurate discretization of the quadratic damping term.

1.8.4 Implementation

The algorithm arising from the methods in Sections 1.8.1 and 1.8.2 is very similar to the undamped case in Section 1.1.2. The difference is basically a question of different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The equation (1.58) is normally impossible to solve by pen and paper, but possible for some special choices of F, s, and f. On the contrary, the complexity of the nonlinear generalized model (1.58) versus the simple undamped model is not a big deal when we solve the problem numerically!

The computational algorithm takes the form

1. $u^0 = I$
2. compute u^1 from (1.64) if linear damping or (1.71) if quadratic damping
3. for $n = 1, 2, \ldots, N_t - 1$:
   a. compute u^{n+1} from (1.62) if linear damping or (1.69) if quadratic damping

Modifying the `solver` function for the undamped case is fairly easy, the big difference being many more terms and if tests on the type of damping:

```
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    """
    Solve m*u'' + f(u') + s(u) = F(t) for t in (0,T],
    u(0)=I and u'(0)=V,
    by a central finite difference method with time step dt.
    If damping is 'linear', f(u')=b*u, while if damping is
    'quadratic', f(u')=b*u'*abs(u').
    F(t) and s(u) are Python functions.
    """
    dt = float(dt); b = float(b); m = float(m) # avoid integer div.
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    if damping == 'linear':
        u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + dt*V + \
               dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, Nt):
        if damping == 'linear':
```

```
            u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                      dt**2*(F(t[n]) - s(u[n])))/(m + b*dt/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                      + dt**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```

The complete code resides in the file `vib.py`.

1.8.5 Verification

Constant solution. For debugging and initial verification, a constant solution is often very useful. We choose $u_{\mathrm{e}}(t) = I$, which implies $V = 0$. Inserted in the ODE, we get $F(t) = s(I)$ for any choice of f. Since the discrete derivative of a constant vanishes (in particular, $[D_{2t}I]^n = 0$, $[D_tI]^n = 0$, and $[D_tD_tI]^n = 0$), the constant solution also fulfills the discrete equations. The constant should therefore be reproduced to machine precision. The function `test_constant` in `vib.py` implements this test.

hpl 8: Add verification tests for constant, linear, quadratic. Check how many bugs that are caught by these tests.

Linear solution. Now we choose a linear solution: $u_{\mathrm{e}} = ct + d$. The initial condition $u(0) = I$ implies $d = I$, and $u'(0) = V$ forces c to be V. Inserting $u_{\mathrm{e}} = Vt + I$ in the ODE with linear damping results in

$$0 + bV + s(Vt + I) = F(t),$$

while quadratic damping requires the source term

$$0 + b|V|V + s(Vt + I) = F(t)\,.$$

Since the finite difference approximations used to compute u' all are exact for a linear function, it turns out that the linear u_{e} is also a solution of the discrete equations. Exercise 1.9 asks you to carry out all the details.

Quadratic solution. Choosing $u_{\mathrm{e}} = bt^2 + Vt + I$, with b arbitrary, fulfills the initial conditions and fits the ODE if F is adjusted properly. The solution also solves the discrete equations with linear damping. However, this quadratic polynomial in t does not fulfill the discrete equations in case of quadratic damping, because the geometric mean used in the approximation of this term introduces an error. Doing Exercise 1.9 will reveal the details. One can fit F^n in the discrete equations such that

the quadratic polynomial is reproduced by the numerical method (to machine precision).

1.8.6 Visualization

The functions for visualizations differ significantly from those in the undamped case in the `vib_undamped.py` program because, in the present general case, we do not have an exact solution to include in the plots. Moreover, we have no good estimate of the periods of the oscillations as there will be one period determined by the system parameters, essentially the approximate frequency $\sqrt{s'(0)/m}$ for linear s and small damping, and one period dictated by $F(t)$ in case the excitation is periodic. This is, however, nothing that the program can depend on or make use of. Therefore, the user has to specify T and the window width to get a plot that moves with the graph and shows the most recent parts of it in long time simulations.

The `vib.py` code contains several functions for analyzing the time series signal and for visualizing the solutions.

1.8.7 User interface

The `main` function is changed substantially from the `vib_undamped.py` code, since we need to specify the new data c, $s(u)$, and $F(t)$. In addition, we must set T and the plot window width (instead of the number of periods we want to simulate as in `vib_undamped.py`). To figure out whether we can use one plot for the whole time series or if we should follow the most recent part of u, we can use the `plot_empricial_freq_and_amplitude` function's estimate of the number of local maxima. This number is now returned from the function and used in `main` to decide on the visualization technique.

```
def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', type=float, default=1.0)
    parser.add_argument('--V', type=float, default=0.0)
    parser.add_argument('--m', type=float, default=1.0)
    parser.add_argument('--c', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='u')
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=140)
    parser.add_argument('--damping', type=str, default='linear')
```

```
    parser.add_argument('--window_width', type=float, default=30)
    parser.add_argument('--savefig', action='store_true')
    a = parser.parse_args()
    from scitools.std import StringFunction
    s = StringFunction(a.s, independent_variable='u')
    F = StringFunction(a.F, independent_variable='t')
    I, V, m, c, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.c, a.dt, a.T, a.window_width, a.savefig, \
       a.damping

    u, t = solver(I, V, m, c, s, F, dt, T)
    num_periods = empirical_freq_and_amplitude(u, t)
    if num_periods <= 15:
        figure()
        visualize(u, t)
    else:
        visualize_front(u, t, window_width, savefig)
    show()
```

The program `vib.py` contains the above code snippets and can solve the model problem (1.58). As a demo of `vib.py`, we consider the case $I = 1$, $V = 0$, $m = 1$, $c = 0.03$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\Delta t = 0.05$, and $T = 140$. The relevant command to run is

Terminal

```
Terminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --c 0.03
```

This results in a moving window following the function on the screen. Figure 1.11 shows a part of the time series.

1.8.8 The Euler-Cromer scheme for the generalized model

The ideas of the Euler-Cromer method from Section 1.7 carry over to the generalized model. We write (1.58) as two equations for u and $v = u'$. The first equation is taken as the one with v' on the left-hand side:

$$v' = \frac{1}{m}(F(t) - s(u) - f(v)), \tag{1.73}$$

$$u' = v\,. \tag{1.74}$$

The idea is to step (1.73) forward using a standard Forward Euler method, while we update u from (1.74) with a Backward Euler method, utilizing the recent, computed v^{n+1} value. In detail,

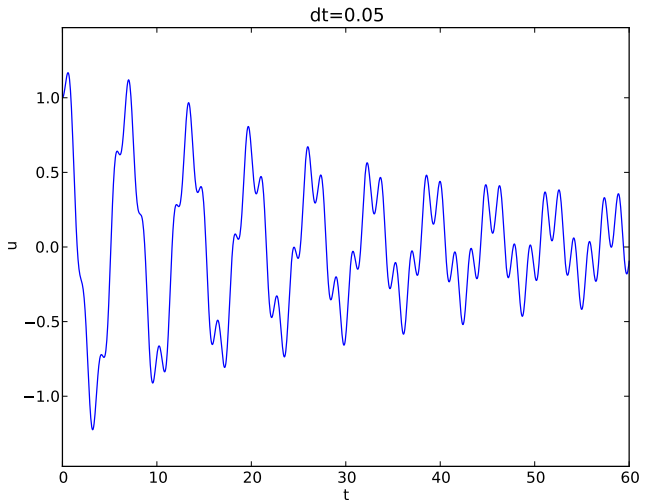


Fig. 1.11 Damped oscillator excited by a sinusoidal function.

$$\frac{v^{n+1} - v^n}{\Delta t} = \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{1.75}$$

$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1}, \tag{1.76}$$

resulting in the explicit scheme

$$v^{n+1} = v^n + \Delta t\frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{1.77}$$

$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,. \tag{1.78}$$

We immediately note one very favorable feature of this scheme: all the nonlinearities in $s(u)$ and $f(v)$ are evaluated at a previous time level. This makes the Euler-Cromer method easier to apply and hence much more convenient than the centered scheme for the second-order ODE (1.58).

The initial conditions are trivially set as

$$v^0 = V, \tag{1.79}$$
$$u^0 = I\,. \tag{1.80}$$

hpl 9: odespy for the generalized problem

1.9 Exercises and Problems

Problem 1.1: Use linear/quadratic functions for verification

Consider the ODE problem

$$u'' + \omega^2 u = f(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,.$$

Discretize this equation according to $[D_t D_t u + \omega^2 u = f]^n$.

a) Derive the equation for the first time step (u^1).

b) For verification purposes, we use the method of manufactured solutions (MMS) with the choice of $u_{\mathrm{e}}(x,t) = ct + d$. Find restrictions on c and d from the initial conditions. Compute the corresponding source term f by term. Show that $[D_t D_t t]^n = 0$ and use the fact that the $D_t D_t$ operator is linear, $[D_t D_t(ct + d)]^n = c[D_t D_t t]^n + [D_t D_t d]^n = 0$, to show that u_{e} is also a perfect solution of the discrete equations.

c) Use `sympy` to do the symbolic calculations above. Here is a sketch of the program `vib_undamped_verify_mms.py`:

```
import sympy as sym
V, t, I, w, dt = sym.symbols('V t I w dt')  # global symbols
f = None  # global variable for the source term in the ODE

def ode_source_term(u):
    """Return the terms in the ODE that the source term
    must balance, here u'' + w**2*u.
    u is symbolic Python function of t."""
    return sym.diff(u(t), t, t) + w**2*u(t)

def residual_discrete_eq(u):
    """Return the residual of the discrete eq. with u inserted."""
    R = ...
    return sym.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    R = ...
    return sym.simplify(R)

def DtDt(u, dt):
    """Return 2nd-order finite difference for u_tt.
```



```
    u is a symbolic Python function of t.
    """
    return ...

def main(u):
    """
    Given some chosen solution u (as a function of t, implemented
    as a Python function), use the method of manufactured solutions
    to compute the source term f, and check if u also solves
    the discrete equations.
    """
    print '=== Testing exact solution: %s ===' % u
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting f
    global f  # source term in the ODE
    f = sym.simplify(ode_lhs(u))

    # Residual in discrete equations (should be 0)
    print 'residual step1:', residual_discrete_eq_step1(u)
    print 'residual:', residual_discrete_eq(u)

def linear():
    main(lambda t: V*t + I)

if __name__ == '__main__':
    linear()
```

Fill in the various functions such that the calls in the `main` function works.

d) The purpose now is to choose a quadratic function $u_{\mathrm{e}} = bt^2 + ct + d$ as exact solution. Extend the `sympy` code above with a function `quadratic` for fitting `f` and checking if the discrete equations are fulfilled. (The function is very similar to `linear`.)

e) Will a polynomial of degree three fulfill the discrete equations?

f) Implement a `solver` function for computing the numerical solution of this problem.

g) Write a nose test for checking that the quadratic solution is computed to correctly (too machine precision, but the round-off errors accumulate and increase with T) by the `solver` function.
Filename: `vib_undamped_verify_mms`.

Exercise 1.2: Show linear growth of the phase with time

Consider an exact solution $I\cos(\omega t)$ and an approximation $I\cos(\tilde{\omega} t)$. Define the phase error as time lag between the peak I in the exact solution and the corresponding peak in the approximation after m periods

of oscillations. Show that this phase error is linear in m. Filename: `vib_phase_error_growth`.

Exercise 1.3: Improve the accuracy by adjusting the frequency

According to (1.19), the numerical frequency deviates from the exact frequency by a (dominating) amount $\omega^3\Delta t^2/24 > 0$. Replace the `w` parameter in the algorithm in the `solver` function in `vib_undamped.py` by `w*(1 - (1./24)*w**2*dt**2` and test how this adjustment in the numerical algorithm improves the accuracy (use $\Delta t = 0.1$ and simulate for 80 periods, with and without adjustment of ω). Filename: `vib_adjust_w`.

Exercise 1.4: See if adaptive methods improve the phase error

Adaptive methods for solving ODEs aim at adjusting Δt such that the error is within a user-prescribed tolerance. Implement the equation $u'' + u = 0$ in the Odespy software. Use the example from Section 3.2.11 in [1]. Run the scheme with a very low tolerance (say 10^{-14}) and for a long time, check the number of time points in the solver's mesh (`len(solver.t_all)`), and compare the phase error with that produced by the simple finite difference method from Section 1.1.2 with the same number of (equally spaced) mesh points. The question is whether it pays off to use an adaptive solver or if equally many points with a simple method gives about the same accuracy. Filename: `vib_undamped_adaptive`.

Exercise 1.5: Use a Taylor polynomial to compute u^1

As an alternative to the derivation of (1.8) for computing u^1, one can use a Taylor polynomial with three terms for u^1:

$$u(t_1) \approx u(0) + u'(0)\Delta t + \frac{1}{2}u''(0)\Delta t^2$$

With $u'' = -\omega^2 u$ and $u'(0) = 0$, show that this method also leads to (1.8). Generalize the condition on $u'(0)$ to be $u'(0) = V$ and compute u^1 in this case with both methods. Filename: `vib_first_step`.

Exercise 1.6: Find the minimal resolution of an oscillatory function

Sketch the function on a given mesh which has the highest possible frequency. That is, this oscillatory "cos-like" function has its maxima and minima at every two grid points. Find an expression for the frequency of this function, and use the result to find the largest relevant value of $\omega\Delta t$ when ω is the frequency of an oscillating function and Δt is the mesh spacing. Filename: `vib_largest_wdt`.

Exercise 1.7: Visualize the accuracy of finite differences for a cosine function

We introduce the error fraction

$$E = \frac{[D_tD_tu]^n}{u''(t_n)}$$

to measure the error in the finite difference approximation D_tD_tu to u''. Compute E for the specific choice of a cosine/sine function of the form $u = \exp(i\omega t)$ and show that

$$E = \left(\frac{2}{\omega\Delta t}\right)^2 \sin^2(\frac{\omega\Delta t}{2}).$$

Plot E as a function of $p = \omega\Delta t$. The relevant values of p are $[0, \pi]$ (see Exercise 1.6 for why $p > \pi$ does not make sense). The deviation of the curve from unity visualizes the error in the approximation. Also expand E as a Taylor polynomial in p up to fourth degree (use, e.g., `sympy`). Filename: `vib_plot_fd_exp_error`.

Exercise 1.8: Verify convergence rates of the error in energy

We consider the ODE problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = V$, for $t \in (0, T]$. The total energy of the solution $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2u^2$ should stay constant. The error in energy can be computed as explained in Section 1.6.

Make a nose test in a file `test_error_conv.py`, where code from `vib_undamped.py` is imported, but the `convergence_rates` and `test_convergence_rates` functions are copied and modified to also

incorporate computations of the error in energy and the convergence rate of this error. The expected rate is 2. Filename: `test_error_conv`.

Exercise 1.9: Use linear/quadratic functions for verification

This exercise is a generalization of Problem 1.1 to the extended model problem (1.58) where the damping term is either linear or quadratic. Solve the various subproblems and see how the results and problem settings change with the generalized ODE in case of linear or quadratic damping. By modifying the code from Problem 1.1, `sympy` will do most of the work required to analyze the generalized problem. Filename: `vib_verify_mms`.

Exercise 1.10: Use an exact discrete solution for verification

Write a nose test function in a separate file that employs the exact discrete solution (1.20) to verify the implementation of the `solver` function in the file `vib_undamped.py`. Filename: `test_vib_undamped_exact_discrete_sol`.

Exercise 1.11: Use analytical solution for convergence rate tests

The purpose of this exercise is to perform convergence tests of the problem (1.58) when $s(u) = \omega^2 u$ and $F(t) = A\sin\phi t$. Find the complete analytical solution to the problem in this case (most textbooks on mechanics or ordinary differential equations list the various elements you need to write down the exact solution). Modify the `convergence_rate` function from the `vib_undamped.py` program to perform experiments with the extended model. Verify that the error is of order Δt^2. Filename: `vib_conv_rate`.

Exercise 1.12: Investigate the amplitude errors of many solvers

Use the program `vib_undamped_odespy.py` from Section 1.5.4 and the amplitude estimation from the `amplitudes` function in the `vib_undamped.py` file (see Section 1.3.5) to investigate how well famous methods for 1st-order ODEs can preserve the amplitude of

u in undamped oscillations. Test, for example, the 3rd- and 4th-order Runge-Kutta methods (`RK3`, `RK4`), the Crank-Nicolson method (`CrankNicolson`), the 2nd- and 3rd-order Adams-Bashforth methods (`AdamsBashforth2`, `AdamsBashforth3`), and a 2nd-order Backwards scheme (`Backward2Step`). The relevant governing equations are listed in the beginning of Section 1.5. Filename: `vib_amplitude_errors`.

Exercise 1.13: Minimize memory usage of a vibration solver

The program `vib.py` store the complete solution $u^0, u^1, \ldots, u^{N_t}$ in memory, which is convenient for later plotting. Make a memory minimizing version of this program where only the last three u^{n+1}, u^n, and u^{n-1} values are stored in memory. Write each computed (t_{n+1}, u^{n+1}) pair to file. Visualize the data in the file (a cool solution is to read one line at a time and plot the u value using the line-by-line plotter in the `visualize_front_ascii` function - this technique makes it trivial to visualize very long time simulations). Filename: `vib_memsave`.

Exercise 1.14: Implement the solver via classes

Reimplement the `vib.py` program using a class `Problem` to hold all the physical parameters of the problem, a class `Solver` to hold the numerical parameters and compute the solution, and a class `Visualizer` to display the solution.

Hint. Use the ideas and examples from Section ?? and ??in [1]. More specifically, make a superclass `Problem` for holding the scalar physical parameters of a problem and let subclasses implement the $s(u)$ and $F(t)$ functions as methods. Try to call up as much existing functionality in `vib.py` as possible.
Filename: `vib_class`.

Exercise 1.15: Interpret $[D_tD_tu]^n$ as a forward-backward difference

Show that the difference $[D_tD_tu]^n$ is equal to $[D_t^+D_t^-u]^n$ and $D_t^-D_t^+u]^n$. That is, instead of applying a centered difference twice one can alternatively apply a mixture forward and backward differences. Filename: `vib_DtDt_fw_bw`.

Exercise 1.16: Use a backward difference for the damping term

As an alternative to discretizing the damping terms $\beta u'$ and $\beta|u'|u'$ by centered differences, we may apply backward differences:

$$
\begin{aligned}
[u']^n &\approx [D_t^- u]^n, \\
[|u'|u']^n &\approx [|D_t^- u| D_t^- u]^n = |[D_t^- u]^n|[D_t^- u]^n \,.
\end{aligned}
$$

The advantage of the backward difference is that the damping term is evaluated using known values u^n and u^{n-1} only. Extend the `vib.py` code with a scheme based on using backward differences in the damping terms. Add statements to compare the original approach with centered difference and the new idea launched in this exercise. Perform numerical experiments to investigate how much accuracy that is lost by using the backward differences. Filename: `vib_gen_bwdamping`.

Exercise 1.17: Analysis of the Euler-Cromer scheme

The Euler-Cromer scheme for the model problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = 0$, is given in (1.53)-(1.52). Find the exact discrete solutions of this scheme and show that the solution for u^n coincides with that found in Section 1.4.

Hint. Use an "ansatz" $u^n = I\exp{(i\tilde{\omega}\Delta t\, n)}$ and $v^n = qu^n$, where $\tilde{\omega}$ and q are unknown parameters. The following formula is handy:

$$
e^{i\tilde{\omega}\Delta t} + e^{i\tilde{\omega}(-\Delta t)} - 2 = 2\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) = -4\sin^2(\frac{\tilde{\omega}\Delta t}{2}) \,.
$$

Wave equations

2

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 deformed 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

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

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

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

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

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

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*, give 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:

$$u_t = \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2}, \tag{2.6}$$

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.

2.1.1 Discretizing the domain

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

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{2.7}$$

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

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L\,. \tag{2.8}$$

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 $n = 0, \ldots, N_t$.

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

$$x_i = i\Delta x,\ i = 0, \ldots, N_x, \quad t_n = n\Delta t,\ n = 0, \ldots, N_t\,. \tag{2.9}$$

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 2.1 displays a mesh in the x,t plane with $N_t = 5$, $N_x = 5$, and constant mesh spacings.

2.1.2 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.

2.1.3 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:

$$\frac{\partial^2}{\partial t^2}u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2}u(x_i, t_n), \tag{2.10}$$

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 $u = 0$.

2.1.4 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 (2.10) and get

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{2.11}$$

or written more compactly using the operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n\,. \tag{2.12}$$

Interpretation of the equation as a stencil. A typical feature of (2.11) 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 2.1 illustrate such neighboring mesh points that contributes to an algebraic equation. In this particular case, we have sampled the PDE at the point $(2, 2)$ and constructed (2.11), which then involves a coupling of u_2^1, u_1^2, u_2^2, u_3^2, and u_2^3. 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 2.1. One also often refers to the algebraic equations as *discrete equations*, *(finite) difference equations* or a *finite difference scheme*.

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

$$\frac{\partial}{\partial t}u(x_i, t_n) \approx \frac{u_i^1 - u_i^{-1}}{2\Delta t} = [D_{2t}u]_i^0,$$

seems appropriate. In operator notation the initial condition is written as

$$[D_{2t}u]_i^n = 0, \quad n = 0\,.$$

Writing out this equation and ordering the terms give

$$u_i^{n-1} = u_i^{n+1}, \quad i = 0, \ldots, N_x,\ n = 0\,. \tag{2.13}$$

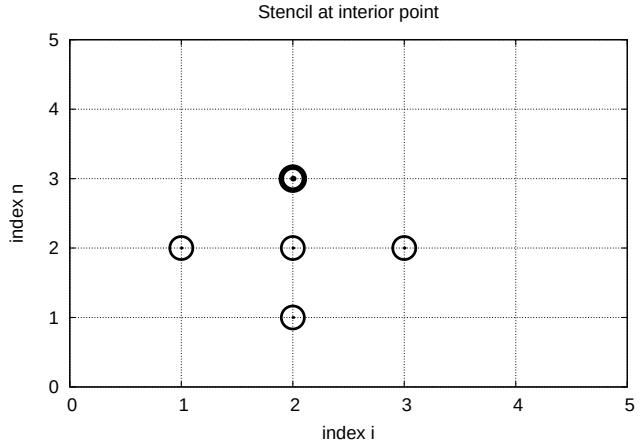


Fig. 2.1 Mesh in space and time. The circles show points connected in a finite difference equation.

The other initial condition can be computed by

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

2.1.5 Formulating a recursive algorithm

We assume that u_i^n and u_i^{n-1} are already computed for $i = 0, \ldots, N_x$. The only unknown quantity in (2.11) is therefore u_i^{n+1}, which we can solve for:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \tag{2.14}$$

where we have introduced the parameter

$$C = c\frac{\Delta t}{\Delta x}, \tag{2.15}$$

known as the *Courant number*.

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 that governs the quality of the numerical solution (see Section 2.10 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 computed for $i = 0, \ldots, N_x$, we find new values at the next time level by applying the formula (2.14) for $i = 1, \ldots, N_x - 1$. 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 $u_i^{n+1} = 0$.

A problem with (2.14) arises when $n = 0$ since the formula for u_i^1 involves u_i^{-1}, which is an undefined quantity outside the time mesh (and the time domain). However, we can use the initial condition (2.13) in combination with (2.14) when $n = 0$ to eliminate u_i^{-1} and arrive at a special formula for u_i^1:

$$u_i^1 = u_i^0 - \frac{1}{2}C^2\left(u_{i+1}^0 - 2u_i^0 + u_{i-1}^0\right) \,. \tag{2.16}$$

Figure 2.2 illustrates how (2.16) 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 (2.16) and set $u_i^1 = 0$ for the boundary points $i = 0$ and $i = N_x$, for $n = 1, 2, \ldots, N - 1$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
   a. apply (2.14) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
   b. set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

The algorithm essentially consists of moving a finite difference stencil through all the mesh points, which can be seen as an animation in a web page or a movie file.

2.1.6 Sketch of an implementation

In a Python implementation of this algorithm, we use the array elements `u[i]` to store u_i^{n+1}, `u_1[i]` to store u_i^n, and `u_2[i]` to store u_i^{n-1}. Our

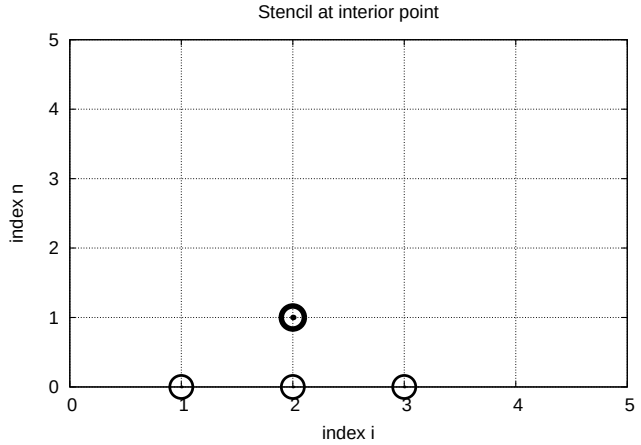


Fig. 2.2 Modified stencil for the first time step.

naming convention is use `u` for the unknown new spatial field to be computed, `u_1` as the solution at one time step back in time, `u_2` as the solution two time steps back in time and so forth.

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.

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

```
# Given mesh points as arrays x and t (x[i], t[n])
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

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_1[i] - 0.5*C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])
```



```
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

# Switch variables before next step
u_2[:], u_1[:] = u_1, u

for n in range(1, Nt):
    # Update all inner mesh points at time t[n+1]
    for i in range(1, Nx):
        u[i] = 2u_1[i] - u_2[i] - \
               C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0

    # Switch variables before next step
    u_2[:], u_1[:] = u_1, u
```

2.2 Verification

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

2.2.1 A slightly generalized model problem

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

$$u_{tt} = c^2 u_{xx} + f(x,t), \qquad x \in (0,L),\ t \in (0,T] \tag{2.17}$$

$$u(x,0) = I(x), \qquad x \in [0,L] \tag{2.18}$$

$$u_t(x,0) = V(x), \qquad x \in [0,L] \tag{2.19}$$

$$u(0,t) = 0, \qquad t > 0 \tag{2.20}$$

$$u(L,t) = 0, \qquad t > 0 \tag{2.21}$$

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

$$[D_t D_t u = c^2 D_x D_x u + f]_i^n \,. \tag{2.22}$$

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

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n \,. \tag{2.23}$$

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 (2.23) for $n = 0$, gives the special formula

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2}C^2\left(u_{i+1}^0 - 2u_i^0 + u_{i-1}^0\right) + \frac{1}{2}\Delta t^2 f_i^0 \,. \tag{2.24}$$

2.2.2 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

$$u_{\mathrm{e}}(x,t)) = A\sin\left(\frac{\pi}{L}x\right)\cos\left(\frac{\pi}{L}ct\right) \,. \tag{2.25}$$

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

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 a plot of the computed solution u_i^n and the exact one (2.25) looks 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 (2.25) for serious and thorough verification is to run a series of 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 convergence rates is given in Section 3.1.6 in [1]. There is also a detailed example on computing convergence rates in Section 1.2.2.

In the present problem, one expects the method to have a convergence rate of 2 (see Section 2.10), so if the computed rates are close to 2 on a sufficiently mesh, we have good evidence that the implementation is free of programming mistakes.

2.2.3 Manufactured solution

One problem with the exact solution (2.25) is that it requires a simplification ($V = 0, f = 0$) of the implemented problem (2.17)-(2.21). 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 $u_{tt} = c^2 u_{xx} + f$ 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

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

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 tests rely 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.

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:

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

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. A standard choice of error measure is the ℓ^2 or ℓ^∞ norm of the error mesh function e_i^n:

$$E = ||e_i^n||_{\ell^2} = \left(\Delta t\Delta x \sum_{n=0}^{N_t}\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n, \quad (2.26)$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{i,n}|e_n^i| . \quad (2.27)$$

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 ($n = N_t$):

$$E = ||e_i^n||_{\ell^2} = \left(\Delta x \sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n, \quad (2.28)$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{0\le i\le N_x}|e_i^n| . \quad (2.29)$$

The important issue is that our error measure E must be one number that represents the error in the simulation.

Let E_i be the error measure in experiment (mesh) number 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.

2.2.4 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 2.10 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

$$u_{\mathrm{e}}(x,t) = x(L-x)(1+\frac{1}{2}t), \tag{2.30}$$

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 $t_n = n\Delta t$ before we establish that

$$[D_tD_t t^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - 2n^2 + (n-1)^2 = 2, \tag{2.31}$$

$$[D_tD_t t]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - 2n + (n-1))\Delta t}{\Delta t^2} = 0\,. \tag{2.32}$$

Hence,

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

Similarly, we get that

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

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

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

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. We can use this result to check that the computed u_i^n values from an implementation equals $u_{\mathrm{e}}(x_i,t_n)$ within machine precision, *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).

Notice

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 $u_{tt} = c^2u_{xx}$ 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 2.10).

2.3 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 2.1.5 and 2.1.6 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.

2.3.1 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 do to 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-suppled 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 call 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 make 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-dependent actions (like visualization), it is a common technique to leave those actions to user-supplied callback functions.

2.3.2 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
    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_1 = np.zeros(Nx+1)   # Solution at 1 time level back
    u_2 = np.zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # for measuring CPU time

    # Load initial condition into u_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)

    # Special formula for first time step
    n = 0
    for i in range(1, Nx):
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[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)

    # Switch variables before next step
    u_2[:] = u_1;  u_1[:] = u

    for n in range(1, Nt):
        # Update all inner points at time t[n+1]
        for i in range(1, Nx):
            u[i] = - u_2[i] + 2*u_1[i] + \
                     C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                     dt**2*f(x[i], t[n])

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

        # Switch variables before next step
        u_2[:] = u_1;  u_1[:] = u

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

2.3.3 Verification: exact quadratic solution

We use the test problem derived in Section 2.2.1 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 ether `py.test` or `nosetests`,

Terminal

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

to automatically run all test functions with name `test_*()`.

2.3.4 Visualization: animating the solution

Now that we have verified the implementation it is time to do a real computation where we also display the 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.

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 paramteres
    umin, umax,               # Interval for u in plots
    animate=True,             # Simulation with animation?
    tool='matplotlib',        # 'matplotlib' or 'scitools'
    solver_function=solver,   # Function with numerical algorithm
    ):
    """Run solver and visualize u at each time level."""

    def plot_u_st(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, L, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Let the initial condition stay on the screen for 2
        # seconds, else insert a pause of 0.2 s between each plot
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n)  # for movie making

    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

    if tool == 'matplotlib':
        import matplotlib.pyplot as plt
        plot_u = PlotMatplotlib()
    elif tool == 'scitools':
        import scitools.std as plt  # scitools.easyviz interface
        plot_u = plot_u_st
    import time, glob, os

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

    # Call solver and do the simulaton
    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(
        I, V, f, c, L, dt, C, T, user_action)

    # Make video files
    fps = 4  # frames per second
    codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                     libtheora='ogg')  # video formats
    filespec = 'tmp_%04d.png'
    movie_program = 'ffmpeg'  # or 'avconv'
    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)

    if tool == 'scitools':
        # Make an HTML play for showing the animation in a browser
        plt.movie('tmp_*.png', encoder='html', fps=fps,
                  output_file='movie.html')
    return cpu
```

Dissection of the code. The `viz` function can either use SciTools or Matplotlib for visualizing the solution. The `user_action` function based on SciTools is called `plot_u_st`, while the `user_action` function based on Matplotlib is a bit more complicated as it is realized as a class and needs statements that differ from those for making static plots. SciTools can utilize both Matplotlib and Gnuplot (and many other plotting programs) for doing the graphics, but Gnuplot is a relevant choice for large N_x or in two-dimensional problems as Gnuplot is significantly faster than Matplotlib for screen animations.

A function inside another function, like `plot_u_st` in the above code segment, has access to *and remembers* all the local variables in the surrounding code inside the `viz` function (!). This is known in computer science as a *closure* and is very convenient to program with. For example, the `plt` and `time` modules defined outside `plot_u` are accessible for `plot_u_st` when the function is called (as `user_action`) in the `solver` function. Some may think, however, that a class instead of a closure is a cleaner and easier-to-understand implementation of the user action function, see Section 2.8.

The `plot_u_st` function just makes a standard SciTools `plot` command for plotting `u` as a function of `x` at time `t[n]`. To achieve a smooth animation, the `plot` command should take keyword arguments instead of being broken into separate calls to `xlabel`, `ylabel`, `axis`, `time`, and `show`. Several `plot` calls will automatically cause an animation on the screen. In addition, 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`. Section 1.3.2 contains more basic information on making animations.

The solver is called with an argument `plot_u` as `user_function`. If the user chooses to use SciTools, `plot_u` is the `plot_u_st` callback function, but for Matplotlib it is an instance of the class `PlotMatplotlib`. Also this class makes use of variables defined in the `viz` function: `plt`

and `time`. 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`.

Making movie files. From the `frame_*.png` files containing the frames in the animation we can make video files. Section 1.3.2 presents basic information on how to use the `ffmpeg` (or `avconv`) program for producing video files in different modern formats: Flash, MP4, Webm, and Ogg.

The `viz` function creates a `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. As mentioned in Section 1.3.2, only two formats are actually needed to ensure that all browsers can successfully play the video: MP4 and WebM.

Some animations consisting of a large number of plot files may not be properly combined into a video using `ffmpeg` or `avconv`. A method that always works is to play the PNG files as an animation in a browser using JavaScript code in an HTML file. The SciTools package has a function `movie` (or a stand-alone command `scitools movie`) for creating such an HTML player. The `plt.movie` call in the `viz` function shows how the function is used. The file `movie.html` can be loaded into a browser and features a user interface where the speed of the animation can be controlled. Note that the movie in this case consists of the `movie.html` file and all the frame files `tmp_*.png`.

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`) 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. Then we need to plot every 801/60 frame, which with integer division yields 13 as `every`. Using the mod function, `n % every`, 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 in the file `wave1D_u0v.py`.

2.3.5 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:

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{2.33}$$

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 form half a wavelength. There is no external force, so $f = 0$, and the string is at rest initially so that $V = 0$.

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*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./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, tool='scitools')
```

The associated program has the name `wave1D_u0.py`. Run the program and watch the movie of the vibrating string.

hpl 10: Must recompute these movies as Δt is different when $C < 1$.

2.3.6 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 value of these is not very large or small, but preferably around unity in size. We introduce the dimensionless variables

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

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: $a = \max_x |I(x)|$.

Inserting these new variables in the PDE and noting that

$$\frac{\partial u}{\partial t} = \frac{aL}{c}\frac{\partial \bar{u}}{\partial \bar{t}},$$

by the chain rule, one gets

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

in case $f = 0$. Dropping the bars, we arrive at the scaled PDE

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \tag{2.34}$$

which has not 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 (2.33), 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 as u behaves as $\cos(\omega t)$ in time in variables with dimension. 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 $u(0, t) = u(0, L)$, 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 is $\tilde{P}$ 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 $\tilde{P} = 2/n$ ($n > 1$).

2.4 Vectorization

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. Vectorization not only speeds up the code on serial computers, but also makes it easy to exploit parallel computing.

2.4.1 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 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*.

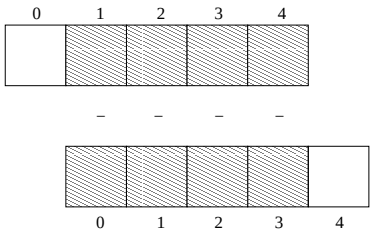


Fig. 2.3 Illustration of subtracting two slices of two arrays.

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 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 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 in 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.

2.4.2 Finite difference schemes expressed as slices

We now have the necessary tools to vectorize the wave equation algorithm as described mathematically in Section 2.1.5 and through code in Section 2.3.2. 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:

```
for i in range(1, Nx):
    u[i] = 2*u_1[i] - u_2[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

The vectorized version becomes


```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

or

```
u[1:Nx] = 2*u_1[1:Nx]- u_2[1:Nx] + \
          C2*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[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).

2.4.3 Verification

We may reuse the quadratic solution $u_{\mathrm{e}}(x,t) = x(L-x)(1+\frac{1}{2}t)$ 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 work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    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 is a scalar (zeros_like(x) works for scalar x too)
    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, ...)
```

2.4.4 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 2.3.4. 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 quite complicated animation code we then duplicate, this is not a good idea. Introducing the `version` argument in `wave1D_u0.viz` is not a good solution since `version` has no meaning in that file.

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='vectorized'`. 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 paramteres
    umin, umax,               # Interval for u in plots
    animate=True,             # Simulation with animation?
    tool='matplotlib',        # 'matplotlib' or 'scitools'
    solver_function=solver,   # Function with numerical algorithm
    version='vectorized',     # 'scalar' or 'vectorized'
    ):
    import wave1D_u0
    if version == 'vectorized':
        # Reuse viz from wave1D_u0, but with the present
        # modules' new vectorized solver (which has
        # version='vectorized' as default argument;
        # wave1D_u0.viz does not feature this argument)
        cpu = wave1D_u0.viz(
            I, V, f, c, L, dt, C, T, umin, umax,
            animate, tool, solver_function=solver)
    elif version == 'scalar':
        # Call wave1D_u0.viz with a solver with
        # scalar code and use wave1D_u0.solver.
        cpu = wave1D_u0.viz(
            I, V, f, c, L, dt, C, T, umin, umax,
            animate, tool,
            solver_function=wave1D_u0.solver)
```

Solution 2. There is a more advanced, 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
# Call scalar with version fixed to 'scalar'
scalar_solver = functools.partial(scalar, 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'`.

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!

2.4.5 Remark on the updating of arrays

At the end of each time step we need to update the `u_2` and `u_1` arrays such that they have the right content for the next time step:

```
u_2[:] = u_1
u_1[:] = u
```

The order here is important! (Updating `u_1` first, makes `u_2` equal to `u`, which is wrong.)

The assignment `u_1[:] = u` copies the content of the `u` array into the elements of the `u_1` 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_2` and `u_1` 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_2` refer

to the `u_1` object and `u_1` refer to the `u` object. This is a very efficiency operation (like switching pointers in C). A naive implementation like

```
u_2 = u_1
u_1 = u
```

will fail, however, because now `u_2` refers to the `u_1` object, but then the name `u_1` refers to `u`, so that this `u` object has two references, `u_1` and `u`, while our third array, originally referred to by `u_2` has no more references and is lost. This means that the variables `u`, `u_1`, and `u_2` 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_1` too so the solution at the previous time step, which is crucial in our formulas, is destroyed.

While `u_2 = u_1` is fine, `u_1 = u` is problematic, so the solution to this problem is to ensure that `u` points to the `u_2` 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_2
u_2 = u_1
u_1 = u
u = tmp
```

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

```
u_2, u_1, u = u_1, u, u_2
```

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

Caution:

The update `u_2, u_1, u = u_1, u, u_2` 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_2`, which is obviously wrong. It is therefore important to adjust the content of `u` to `u = u_1` before returning `u`.

2.5 Exercises

Exercise 2.1: 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.

b) Import the `solver` function `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.

c) Make animations where you illustrate how the error $e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n$ develops and increases in time. Also make animations of u and u_{e} simultaneously.

Hint 1. Quite long time simulations are needed in order to display significant discrepancies between the numerical and exact solution.

Hint 2. 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.
Filename: `wave_standing`.

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 2.10.4 for explanation).

Exercise 2.2: 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. Filename: `wave1D_u0_s_store`.

Exercise 2.3: Use a class for the user action function

Redo Exercise 2.2 using a class for the user action function. That is, define a class `Action` where the `all_u` list is an attribute, and implement the user action function as a method (the special method `__call__` is a natural choice). The class versions avoids that the user action function depends on parameters defined outside the function (such as `all_u` in Exercise 2.2). Filename: `wave1D_u0_s2c`.

Exercise 2.4: 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. Import the `solver` function from the `wave1D_u0_s` movie in a new file `wave_compare.py`. Reimplement the `viz` function such that it 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 2.2 or 2.3 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 corresponds to the same time point. The easiest remedy is to keep the time and space resolution constant and change the wave velocity c to change the Courant number. Filename: `wave_numerics_comparison`.

Project 2.5: 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.

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

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. S how 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.

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

Hint. 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]`.

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
```

Filename: `mesh_calculus_1D`.

2.6 Generalization: reflecting boundaries

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, see a web page or a movie file for demonstration.

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.

2.6.1 Neumann boundary condition

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

$$\frac{\partial u}{\partial n} \equiv \boldsymbol{n}\cdot\nabla u = 0\,. \tag{2.35}$$

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

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

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.

2.6.2 Discretization of derivatives at the boundary

How can we incorporate the condition (2.35) 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 (2.35) at $x = 0$ and $t = t_n$ by the difference

$$[D_{2x}u]_0^n = \frac{u_{-1}^n - u_1^n}{2\Delta x} = 0\,. \tag{2.36}$$

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 (2.36) with the scheme for $i = 0$,

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \tag{2.37}$$

we can eliminate the fictitious value u_{-1}^n. We see that $u_{-1}^n = u_1^n$ from (2.36), which can be used in (2.37) to arrive at a modified scheme for the boundary point u_0^{n+1}:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u_{i+1}^n - u_i^n\right), \quad i = 0\,. \tag{2.38}$$

Figure 2.4 visualizes this equation for computing u_0^3 in terms of u_0^2, u_0^1, and u_1^2.

Similarly, (2.35) applied at $x = L$ is discretized by a central difference

$$\frac{u_{N_x+1}^n - u_{N_x-1}^n}{2\Delta x} = 0\,. \tag{2.39}$$

Combined with the scheme for $i = N_x$ we get a modified scheme for the boundary value $u_{N_x}^{n+1}$:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u_{i-1}^n - u_i^n\right), \quad i = N_x\,. \tag{2.40}$$

The modification of the scheme at the boundary is also required for the special formula for the first time step. How the stencil moves through the mesh and is modified at the boundary can be illustrated by an animation in a web page or a movie file.

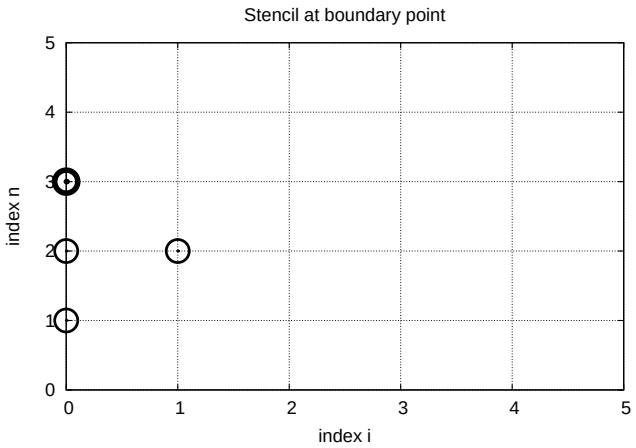


Fig. 2.4 Modified stencil at a boundary with a Neumann condition.

2.6.3 Implementation of Neumann conditions

We have seen in the preceding section that the special formulas for the boundary points arise from replacing u_{i-1}^n by u_{i+1}^n when computing u_i^{n+1} from the stencil formula for $i = 0$. Similarly, we replace u_{i+1}^n by u_{i-1}^n 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_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[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_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[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 2.4.3. However, the Neumann conditions requires the polynomial variation in x direction to be of third degree, which causes challenging problems when designing a test where the numerical solution is known exactly. Exercise 2.14 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 2.10.4).

2.6.4 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, $i \in \mathcal{I}_x$, 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 of 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

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\}, \tag{2.41}$$

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

$$
\begin{aligned}
u_i^{n+1} &= u_i^n - \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \quad , i \in \mathcal{I}_x^i,\ n = 0,\\
u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \quad i \in \mathcal{I}_x^i,\ n \in \mathcal{I}_t^i,\\
u_i^{n+1} &= 0, \quad i = \mathcal{I}_x^0,\ n \in \mathcal{I}_t^-,\\
u_i^{n+1} &= 0, \quad i = \mathcal{I}_x^{-1},\ n \in \mathcal{I}_t^- .
\end{aligned}
$$

The corresponding implementation becomes

```
# Initial condition
for i in Ix[1:-1]:
    u[i] = u_1[i] - 0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])

# Time loop
for n in It[1:-1]:
    # Compute internal points
    for i in Ix[1:-1]:
        u[i] = - u_2[i] + 2*u_1[i] + \
               C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
    # Compute boundary conditions
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Notice

The program `wave1D_dn.py` applies the index set notation and solves the 1D wave equation $u_{tt} = c^2 u_{xx} + f(x,t)$ with quite general boundary and initial conditions:

- $x = 0$: $u = U_0(t)$ or $u_x = 0$
- $x = L$: $u = U_L(t)$ or $u_x = 0$
- $t = 0$: $u = I(x)$
- $t = 0$: $u_t = I(x)$

The program combines Dirichlet and Neumann conditions, scalar and vectorized implementation of schemes, and the index 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 exact 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.

hpl 11: Should include some experiments here or make exercises. Qualitative behavior of the wave equation can be exemplified.

2.6.5 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 and Neumann conditions at $x = 0$ and $x = L$. It is tempting to apply a quadratic solution as described in Sections 2.2.1 and 2.3.3, 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 work 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 though, 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.

2.6.6 Alternative implementation via ghost cells

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 $[L, L+\Delta x]$, often referred to 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 $i = -1$ 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 $u_x = 0$.

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_1` and `u_2` 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

$$u^{n+1}_i = \cdots, \quad i = 1, \ldots, N_x + 1,$$

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 2.6.4: 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_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[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_1[i] + dt*V(x[i-Ix[0]]) + \
           0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[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`.

Warning

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_1[i] = I(x[i])
```

becomes wrong, since `u_1` 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_1[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_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[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]`.

2.7 Generalization: variable wave velocity

Our next generalization of the 1D wave equation (2.1) or (2.17) 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 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 passes 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).

2.7.1 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

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{2.42}$$

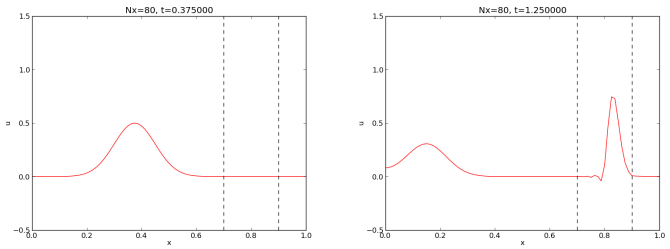


Fig. 2.5 Left: wave entering another medium; right: transmitted and reflected wave.

This is the most frequent form of a wave equation with variable wave velocity, but other forms also appear, see Section 2.15.1 and equation (2.125).

As usual, we sample (2.42) 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

$$\frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) = \left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]^n_i .$$

2.7.2 Discretizing the variable coefficient

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:

$$\left[\frac{\partial \phi}{\partial x}\right]^n_i \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x\phi]^n_i .$$

Then discretize

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]^n_{i+\frac{1}{2}} \approx q_{i+\frac{1}{2}}\frac{u^n_{i+1} - u^n_i}{\Delta x} = [qD_xu]^n_{i+\frac{1}{2}} .$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]^n_{i-\frac{1}{2}} \approx q_{i-\frac{1}{2}}\frac{u^n_i - u^n_{i-1}}{\Delta x} = [qD_xu]^n_{i-\frac{1}{2}} .$$

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]^n_i \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u^n_{i+1} - u^n_i\right) - q_{i-\frac{1}{2}}\left(u^n_i - u^n_{i-1}\right)\right) . \tag{2.43}$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]^n_i \approx [D_xqD_xu]^n_i . \tag{2.44}$$

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 (2.43) or $[D_x(qD_xu)]^n_i$.

When using the chain rule, we get two terms $qu_{xx} + q_xu_x$. The typical discretization is

$$qD_xD_xu + D_{2x}qD_{2x}u]^n_i, \tag{2.45}$$

Writing this out shows that it is different from $[D_x(qD_xu)]^n_i$ 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, $[D_x(qD_xu)]^n_i$ with harmonic averaging of q yields a better solution than arithmetic averaging or (2.45). In the literature, the discretization

$[D_x(qD_xu)]_i^n$ totally dominant and very few mention the possibility of (2.45).

2.7.3 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} can then be done by averaging in three ways:

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}\left(q_i + q_{i+1}\right) = [\overline{q}^x]_i \qquad \text{(arithmetic mean)} \tag{2.46}$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1} \qquad \text{(harmonic mean)} \tag{2.47}$$

$$q_{i+\frac{1}{2}} \approx \left(q_i q_{i+1}\right)^{1/2} \qquad \text{(geometric mean)} \tag{2.48}$$

The arithmetic mean in (2.46) 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 (see Section 1.8.2 for an example).

With the operator notation from (2.46) we can specify the discretization of the complete variable-coefficient wave equation in a compact way:

$$[D_tD_tu = D_x\overline{q}^xD_xu + f]_i^n\,. \tag{2.49}$$

From this 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 $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i + q_{i+1})/2$. With the notation $[D_xqD_xu]_i^n$, we specify that q is evaluated directly, as a function, between the mesh points: $q(x_{i-\frac{1}{2}})$ and $q(x_{i+\frac{1}{2}})$.

Before any implementation, it remains to solve (2.49) with respect to u_i^{n+1}:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2\left(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n\,. \tag{2.50}$$

2.7.4 How a variable coefficient affects the stability

The stability criterion derived in Section 2.10.3 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 $\Delta x/c(x) > \Delta t$. That is, we must use the largest c value in the criterion:

$$\Delta t \leq \beta\frac{\Delta x}{\max_{x\in[0,L]} c(x)}\,. \tag{2.51}$$

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 using a constant Δt and a worst case value of $c(x)$ (with a safety factor β).

2.7.5 Neumann condition and a variable coefficient

Consider a Neumann condition $\partial u/\partial x = 0$ at $x = L = N_x\Delta x$, discretized as

$$[D_{2x}u]_i^n = \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0 \quad u_{i+1}^n = u_{i-1}^n,$$

for $i = N_x$. Using the scheme (2.50) at the end point $i = N_x$ with $u_{i+1}^n = u_{i-1}^n$ results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \quad (2.52)$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \quad (2.53)$$

$$\approx -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_i(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \,. \quad (2.54)$$

Here we used the approximation

$$q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} = q_i + \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots + q_i - \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots = 2q_i + 2\left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \mathcal{O}(\Delta x^4) \approx 2q_i \,. \quad (2.55)$$

An alternative derivation may apply the arithmetic mean of q in (2.52), 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

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \quad (2.56)$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \,. \quad (2.57)$$

2.7.6 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 (2.50):

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

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 2.7.5. 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 2.6.3. 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_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])
```

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_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])
```

2.7.7 A more general PDE model with variable coefficients

Sometimes a wave PDE has a variable coefficient in front of the time-derivative term:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{2.58}$$

One example appears when modeling elastic waves in a rod with varying density, cf. (2.15.1) with $\varrho(x)$.

A natural scheme for (2.58) is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n\,. \tag{2.59}$$

We realize that the ϱ coefficient poses no particular difficulty, since ϱ enters the formula just 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:

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]_i^n\,. \tag{2.60}$$

2.7.8 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):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{2.61}$$

where $b \geq 0$ is a prescribed damping coefficient.

A typical discretization of (2.61) in terms of centered differences reads

$$[D_t D_t u + bD_{2t} u = c^2 D_x D_x u + f]_i^n\,. \tag{2.62}$$

Writing out the equation and solving for the unknown u_i^{n+1} gives the scheme

$$u_i^{n+1} = (1+\frac{1}{2}b\Delta t)^{-1}((\frac{1}{2}b\Delta t - 1)u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t^2 f_i^n), \tag{2.63}$$

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.

2.8 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

$$u_{tt} = (c^2(x)u_x)_x + f(x,t), \qquad x \in (0,L),\ t \in (0,T] \tag{2.64}$$
$$u(x,0) = I(x), \qquad x \in [0,L] \tag{2.65}$$
$$u_t(x,0) = V(t), \qquad x \in [0,L] \tag{2.66}$$
$$u(0,t) = U_0(t) \text{ or } u_x(0,t) = 0, \qquad t \in (0,T] \tag{2.67}$$
$$u(L,t) = U_L(t) \text{ or } u_x(L,t) = 0, \qquad t \in (0,T] \tag{2.68}$$

The only new feature here is the time-dependent Dirichlet conditions. These 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 ($u_x = 0$), a the 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.

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.

2.8.1 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.

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
        backend='matplotlib',       # or 'gnuplot' or None
        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
        self.backend = backend
        if backend is None:
            # Use native matplotlib
            import matplotlib.pyplot as plt
        elif backend in ('matplotlib', 'gnuplot'):
            module = 'scitools.easyviz.' + backend + '_'
            exec('import %s as plt' % module)
        self.plt = plt
        self.screen_movie = screen_movie
        self.title = title
        self.skip_frame = skip_frame
        self.filename = filename
        if filename is not None:
            # Store time points when u is written to file
            self.t = []
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            for filename in filenames:
                os.remove(filename)

        # Clean up old movie frames
```



```
        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.
        """
        # Save solution u to a file using numpy.savez
        if self.filename is not None:
            name = 'u%04d' % n  # array name
            kwargs = {name: u}
            fname = '.' + self.filename + '_' + name + '.dat'
            savez(fname, **kwargs)
            self.t.append(t[n])  # store corresponding time value
            if n == 0:           # save x once
                savez('.' + self.filename + '_x.dat', x=x)

        # Animate
        if n % self.skip_frame != 0:
            return
        title = 't=%.3f' % t[n]
        if self.title:
            title = self.title + ' ' + title
        if self.backend is None:
            # native matplotlib animation
            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:
                # Update new solution
                self.lines[0].set_ydata(u)
                self.plt.legend(['t=%.3f' % t[n]])
                self.plt.draw()
        else:
            # scitools.easyviz animation
            self.plt.plot(x, u, 'r-',
                          xlabel='x', ylabel='u',
                          axis=[x[0], x[-1],
                                self.yaxis[0], self.yaxis[1]],
                          title=title,
                          show=self.screen_movie)
        # pause
        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))
```

Dissection. Understanding this class requires quite some familiarity with Python in general and class programming in particular. The

class supports plotting with Matplotlib (`backend=None`) or SciTools (`backend=matplotlib` or `backend=gnuplot`) for maximum flexibility.

The constructor shows how we can flexibly import the plotting engine as (typically) `scitools.easyviz.gnuplot_` or `scitools.easyviz.matplotlib_` (note the trailing underscore - it is required). 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 C.3.1.

2.8.2 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 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 location of a medium with a different wave velocity.

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,            # aximum 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, # 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.
    """
    # Use scaled parameters: L=1 for domain length, c_0=1
    # for wave velocity outside the domain.
    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 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):
            # One period of a cosine
            w = 2
            a = w*sigma
            return 0.5*(1 + cos(pi*(x-xc)/a)) \
                   if xc - a <= x <= xc + a else 0

    elif pulse_tp == 'half-cosinehat':
        def I(x):
            # Half a period of a cosine
            w = 4
            a = w*sigma
            return cos(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')

    # Choose the stability limit with given Nx, worst case c
    # (lower C will then use this dt, but smaller Nx)
    dt = (L/Nx)/c_0
    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)
    action.make_movie_file()
    action.file_close()
```

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 varies when the Courant number varies).

The reader is encouraged to play around with the `pulse` function:

```
>>> import wave1D_dn_vc as w
>>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, every_frame=10)
```

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

```
Terminal> python -c 'import wave1D_dn_vc as w; w.pulse(...)'
```


2.9 Exercises

Exercise 2.6: Find the analytical solution to a damped wave equation

Consider the wave equation with damping (2.61). The goal is to find an exact solution to a wave problem with damping. A starting point is the standing wave solution from Exercise 2.1. It becomes necessary to include a damping term e^{-ct} and also have both a sine and cosine component in time:

$$u_{\mathrm{e}}(x,t) = e^{-\beta t}\sin kx\left(A\cos\omega t + B\sin\omega t\right) .$$

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. Filename: `damped_waves`.

Problem 2.7: 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 the half of the domain $[0, L]$.

a) Argue why the symmetry boundary condition is $u_x = 0$ at $x = 0$.

Hint. Symmetry of a function about $x = x_0$ means that $f(x_0 + h) = f(x_0 - h)$.

b) Perform simulations of the complete wave problem from 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 the two solutions and make sure that they 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.
Filename: `wave1D_symmetric`.

Exercise 2.8: 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 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 $T = 2$. Filename: `pulse1D`.

Exercise 2.9: Explain why numerical noise occurs

The experiments performed in Exercise 2.8 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" pulses for $s_f = 1$, $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion relation to explain the observations. Filename: `pulse1D_analysis`.

Exercise 2.10: 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 2.8? Filename: `pulse1D_harmonic`.

Problem 2.11: 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*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0\,. \tag{2.69}$$

The parameter c is the wave velocity.

Show that (2.69) accepts a solution $u = g_R(x - ct)$ (right-going wave), but not $u = g_L(x + ct)$ (left-going wave). This means that (2.69) 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

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0\,. \tag{2.70}$$

a) A natural idea for discretizing the condition (2.69) at the spatial end point $i = N_x$ is to apply centered differences in time and space:

$$[D_{2t}u + cD_{2x}u = 0]^n_i, \quad i = N_x\,. \tag{2.71}$$

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^1_i, 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

$$[D^+_t u + cD^-_x u = 0]^n_i, \quad i = N_x\,. \tag{2.72}$$

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 $u_{tt} = c^2 u_{xx}$ in a domain $[0, L]$, where you have $u_x = 0$ at $x = 0$, the condition (2.69) 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).

Hint. 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

$$[D_t^+ u - cD_x^+ u = 0]_i^n, \quad i = 0, \tag{2.73}$$

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 nose tests 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 (2.72). Make a movies to illustrate what happens.
Filename: `wave1D_open_BC`.

Remarks. The condition (2.69) 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.

Exercise 2.12: 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 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 2.11) 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 2.7) $u_x = 0$, or an open boundary condition.

This initial wave will split in two and either 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 2.11. 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: $u(x, 0) = \exp(-\frac{1}{2}(x - 0.5)^2/0.05)$. 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. Filename: `periodic`.

Exercise 2.13: 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 (2.54) and (2.57).

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 (2.54).

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 (2.57). 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).
Filename: `Neumann_discr`.

Exercise 2.14: 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 $u_{tt} = c^2u_{xx} + f$ 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

$$[D_{2x}u]_i^n = u_x(x_i,t_n) + \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,. \tag{2.74}$$

At the boundary two boundary points, $X_x(x) = 0$ 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 $\sum_{j=0}^3 a_jx^j$. 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)

    # Assume discrete solution is a polynomial of degree 3 in x
    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:])
# s is dictionary with the unknowns a[2] and a[3] as keys
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.

Hint. To turn a `sympy` expression `e`, depending on a series of symbols, say `x`, `t`, `dx`, `dt`, `L`, and `c`, into 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 $u = T(t)\sum_{j=0}^{3} a_jx^j$, 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 `sumpy` 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
    # Set u as a 3rd-degree polynomial in space
    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)
    # Force discrete boundary condition to be zero by adding
    # a correction term the analytical suggestion x*(L-x)*T
    # u_x = x*(L-x)*T(t) - 1/6*u_xxx*dx**2
    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 is a polynomial: force all coefficients to vanish.
    # Turn R to Poly to extract coefficients:
    R = sm.poly(R, x)
    coeff = R.all_coeffs()
    s = sm.solve(coeff, a[1:])  # a[0] is not present in R
    # s is dictionary with a[i] as keys
    # Fix a[0] as 1
    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. Filename: `wave1D_n0_test_cubic`.

2.10 Analysis of the difference equations

2.10.1 Properties of the solution of the wave equation

The wave equation

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

has solutions of the form

$$u(x,t) = g_R(x - ct) + g_L(x + ct), \tag{2.75}$$

for any functions g_R and g_L sufficiently smooth to be differentiated twice. The result follows from inserting (2.75) 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 x axis with coordinates $\xi = x - ct$, 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 (2.75),

$$g_R(x) + g_L(x) = I(x), \quad -cg_R'(x) + cg_L'(x) = 0,$$

which have the solution $g_R = g_L = I/2$, and consequently

$$u(x,t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct)\,. \tag{2.76}$$

The interpretation of (2.76) 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 (2.76) 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 (2.76) 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}:

$$I(x) \approx \sum_{k \in K} b_k e^{ikx}\,. \tag{2.77}$$

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 (2.77) converge to $I(x)$. This sum is known as a *Fourier series* representation of $I(x)$. Looking at (2.76), we see that the solution $u(x,t)$, when $I(x)$ is represented as in (2.77), is also built of basic complex exponential wave components of the form $e^{ik(x \pm ct)}$ according to

$$u(x,t) = \frac{1}{2}\sum_{k\in K} b_k e^{ik(x-ct)} + \frac{1}{2}\sum_{k\in K} b_k e^{ik(x+ct)}\,. \tag{2.78}$$

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.)

2.10.2 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*:

$$I(x) = \int_{-\infty}^{\infty} A(k)e^{ikx}dk, \tag{2.79}$$

$$A(k) = \int_{-\infty}^{\infty} I(x)e^{-ikx}dx\,. \tag{2.80}$$

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^n_q, 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 "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:

$$I_q = \frac{1}{N_x+1}\sum_{k=0}^{N_x} A_k e^{i2\pi kj/(N_x+1)}, \quad i = 0,\ldots,N_x, \tag{2.81}$$

$$A_k = \sum_{q=0}^{N_x} I_q e^{-i2\pi kq/(N_x+1)}, \quad k = 0,\ldots,N_x+1\,. \tag{2.82}$$

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 sin, pi

def I(x):
    return sin(2*pi*x) + 0.5*sin(4*pi*x) + 0.1*sin(6*pi*x)

# Mesh
L = 10; Nx = 100
x = np.linspace(0, L, Nx+1)
dx = L/float(Nx)

# Discrete Fourier transform
A = np.fft.rfft(I(x))
A_amplitude = np.abs(A)

# Compute the corresponding frequencies
freqs = np.linspace(0, pi/dx, A_amplitude.size)

import matplotlib.pyplot as plt
plt.plot(freqs, A_amplitude)
plt.show()
```

2.10.3 Stability

The scheme

$$[D_tD_tu = c^2D_xD_xu]^n_q \tag{2.83}$$

for the wave equation $u_t = c^2u_{xx}$ allows basic wave components

$$u^n_q = e^{i(kx_q - \tilde{\omega}t_n)}$$

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 counter for the mesh point in x direction.

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:

$$[D_tD_te^{i\omega t}]^n = -\frac{4}{\Delta t^2}\sin^2\left(\frac{\omega\Delta t}{2}\right)e^{i\omega n\Delta t}.$$

By just changing symbols ($\omega \to k$, $t \to x$, $n \to q$) it follows that

$$[D_xD_xe^{ikx}]_q = -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)e^{ikq\Delta x}.$$

Numerical wave propagation. Inserting a basic wave component $u^n_q = e^{i(kx_q - \tilde{\omega}t_n)}$ in (2.83) results in the need to evaluate two expressions:

$$\begin{aligned}[D_tD_te^{ikx}e^{-i\tilde{\omega}t}]^n_q &= [D_tD_te^{-i\tilde{\omega}t}]^n e^{ikq\Delta x}\\ &= -\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right)e^{-i\tilde{\omega}n\Delta t}e^{ikq\Delta x} \qquad (2.84)\\ [D_xD_xe^{ikx}e^{-i\tilde{\omega}t}]^n_q &= [D_xD_xe^{ikx}]_q e^{-i\tilde{\omega}n\Delta t}\\ &= -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)e^{ikq\Delta x}e^{-i\tilde{\omega}n\Delta t}. \qquad (2.85)\end{aligned}$$

Then the complete scheme,

$$[D_tD_te^{ikx}e^{-i\tilde{\omega}t} = c^2D_xD_xe^{ikx}e^{-i\tilde{\omega}t}]^n_q$$

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

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2\sin^2\left(\frac{k\Delta x}{2}\right), \tag{2.86}$$

where

$$C = \frac{c\Delta t}{\Delta x} \tag{2.87}$$

is the Courant number. Taking the square root of (2.86) yields

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C\sin\left(\frac{k\Delta x}{2}\right), \tag{2.88}$$

Since the exact ω is real it is reasonable to look for a real solution $\tilde{\omega}$ of (2.88). The right-hand side of (2.88) 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 sine function on the right-hand side can attain the value 1 when

$$\frac{k\Delta x}{2} = m\frac{\pi}{2}, \quad m \in \mathbb{Z}.$$

With $m = 1$ 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

$$C \leq 1. \tag{2.89}$$

Consider a right-hand side in (2.88) of magnitude larger than unity. The solution $\tilde{\omega}$ of (2.88) 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, (2.89) 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 leads to exponential growth of the shortest possible wave component in the mesh. This noise will therefore grow with time and destroy the whole solution.

2.10.4 Numerical dispersion relation

Equation (2.88) can be solved with respect to $\tilde{\omega}$:

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(C\sin\left(\frac{k\Delta x}{2}\right)\right). \tag{2.90}$$

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 (2.90) 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:

$$r(C,p) = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right), \quad C \in (0,1],\ p \in (0,\pi/2]\,. \tag{2.91}$$

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 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:

```
>>> 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
```

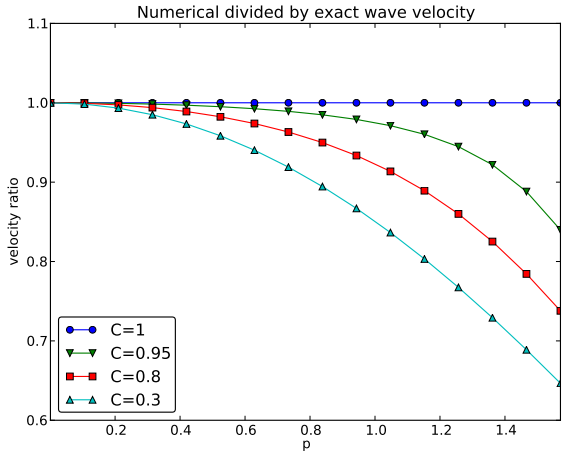


Fig. 2.6 The fractional error in the wave velocity for different Courant numbers.

```
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 get 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

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2 (C^2-1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right), \tag{2.92}$$

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 $C=1$.

2.10.5 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. Such a wave can be expressed by

$$u(x,y,t) = g(k_xx + k_yy - kct)$$

for some twice differentiable function g, or with $\omega = kc$, $k = |\boldsymbol{k}|$:

$$u(x,y,t) = g(k_xx + k_yy - \omega t)\,.$$

We can, in particular, build a solution by adding complex Fourier components of the form

$$\exp\left(i(k_xx + k_yy - \omega t)\right).$$

A discrete 2D wave equation can be written as

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu)]^n_{q,r}\,. \tag{2.93}$$

This equation admits a Fourier component

$$u^n_{q,r} = \exp\left(i(k_xq\Delta x + k_yr\Delta y - \tilde{\omega}n\Delta t)\right), \tag{2.94}$$

as solution. Letting the operators D_tD_t, D_xD_x, and D_yD_y act on $u^n_{q,r}$ from (2.94) transforms (2.93) to

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) + c^2\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right). \tag{2.95}$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2\sin^2 p_x + C_y^2\sin^2 p_y, \tag{2.96}$$

where we have eliminated the factor 4 and introduced the symbols

$$C_x = \frac{c^2\Delta t^2}{\Delta x^2},\quad C_y = \frac{c^2\Delta t^2}{\Delta y^2},\quad p_x = \frac{k_x\Delta x}{2},\quad p_y = \frac{k_y\Delta y}{2}\,.$$

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

$$C_x^2 + C_y^2 \leq 1\,. \tag{2.97}$$

This gives the stability criterion, more commonly expressed directly in an inequality for the time step:

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{2.98}$$

A similar, straightforward analysis for the 3D case leads to

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{2.99}$$

In the case of a variable coefficient $c^2 = c^2(\boldsymbol{x})$, we must use the worst-case value

$$\bar{c} = \sqrt{\max_{\boldsymbol{x}\in\Omega} c^2(\boldsymbol{x})} \tag{2.100}$$

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:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{2.101}$$

The exact numerical dispersion relations in 2D and 3D becomes, for constant c,

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin^2 p_y\right)^{\frac{1}{2}}\right), \tag{2.102}$$

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin^2 p_y + C_z^2\sin^2 p_z\right)^{\frac{1}{2}}\right). \tag{2.103}$$

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 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 (2.97) 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 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.

```
def dispersion_relation_2D(p, theta, C):
    arg = C*sqrt(sin(p*cos(theta))**2 +
                 sin(p*sin(theta))**2)
    c_frac = 2./(C*p)*arcsin(arg)

    return c_frac

import numpy as np
from numpy import \
     cos, sin, arcsin, sqrt, pi  # for nicer math formulas

r = p = np.linspace(0.001, pi/2, 101)
theta = np.linspace(0, 2*pi, 51)
r, theta = np.meshgrid(r, theta)

# Make 2x2 filled contour plots for 4 values of C
import matplotlib.pyplot as plt
C_max = 1/sqrt(2)
C = [[C_max, 0.9*C_max], [0.5*C_max, 0.2*C_max]]
fix, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
for row in range(2):
    for column in range(2):
        error = 1 - dispersion_relation_2D(
            p, theta, C[row][column])
        print error.min(), error.max()
        # use vmin=error.min(), vmax=error.max()
        cax = axes[row][column].contourf(
            theta, r, error, 50, vmin=-1, vmax=-0.28)
        axes[row][column].set_xticks([])
        axes[row][column].set_yticks([])

# Add colorbar to the last plot
cbar = plt.colorbar(cax)
cbar.ax.set_ylabel('error in wave velocity')
plt.savefig('disprel2D.png');  plt.savefig('disprel2D.pdf')
plt.show()
```

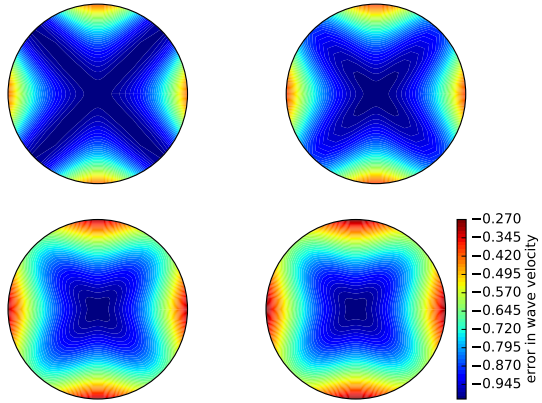



Fig. 2.7 Error in numerical dispersion in 2D.

2.11 Finite difference methods for 2D and 3D 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.

2.11.1 Multi-dimensional wave equations

The general wave equation in d space dimensions, with constant wave velocity c, can be written in the compact form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0, T], \tag{2.104}$$

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

$$\varrho \frac{\partial^2 u}{\partial t^2} = \nabla \cdot (q \nabla u) + f \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0, T], \tag{2.105}$$

which in, e.g., 2D becomes

$$\varrho(x,y) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t)\,. \tag{2.106}$$

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,

$$\frac{\partial^2 u}{\partial t^2} = u_{tt},$$
$$\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) = (q u_y)_y\,.$$

These comments extend straightforwardly to 3D, which means that the 3D versions of the two wave PDEs, with and without variable coefficients, can with be stated as

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f, \tag{2.107}$$
$$\varrho u_{tt} = (q u_x)_x + (q u_z)_z + (q u_z)_z + f, \tag{2.108}$$

where the index notation for differentiation has been used.

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 2.11 for details.

All the listed wave equations with *second-order* derivatives in time need *two* initial conditions:

1. $u = I$,
2. $u_t = V$.

2.11.2 Mesh

We introduce a mesh in time and in space. The mesh in time consists of time points

$$t_0 = 0 < t_1 < \cdots < t_{N_t},$$

often 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 domains. More complicated shapes of the domain require substantially more advanced techniques and implementational efforts. On a rectangle- or box-shaped domain, mesh points are introduced separately in the various space directions:

$$x_0 < x_1 < \cdots < x_{N_x} \text{ in the } x \text{ direction},$$
$$y_0 < y_1 < \cdots < y_{N_y} \text{ in the } y \text{ direction},$$
$$z_0 < z_1 < \cdots < z_{N_z} \text{ in the } z \text{ direction}\,.$$

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 $u^n_{i,j}$.

2.11.3 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.

Discretizing the PDEs. Equation (2.107) can be discretized as

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu + D_zD_zu) + f]^n_{i,j,k}\,. \tag{2.109}$$

A 2D version might be instructive to write out in detail:

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu) + f]^n_{i,j,k},$$

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

$$u^{n+1}_{i,j} = 2u^n_{i,j} + u^{n-1}_{i,j} + c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{2.110}$$

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:

$$[D_{2t}u = V]^0_{i,j} \quad \Rightarrow \quad u^{-1}_{i,j} = u^1_{i,j} - 2\Delta tV_{i,j}\,.$$

The result becomes, in compact form,

$$u^{n+1}_{i,j} = u^n_{i,j} - 2\Delta V_{i,j} + \frac{1}{2}c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{2.111}$$

The PDE (2.108) with variable coefficients is discretized term by term using the corresponding elements from the 1D case:

$$[\varrho D_tD_tu = (D_x\overline{q}^xD_xu + D_y\overline{q}^yD_yu + D_z\overline{q}^zD_zu) + f]^n_{i,j,k}\,. \tag{2.112}$$

When written out and solved for the unknown $u^{n+1}_{i,j,k}$, one gets the scheme

$$\begin{aligned}
u^{n+1}_{i,j,k} = {} & -u^{n-1}_{i,j,k} + 2u^n_{i,j,k} + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i+1,j,k})(u^n_{i+1,j,k} - u^n_{i,j,k}) - \\
& \qquad \frac{1}{2}(q_{i-1,j,k} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i-1,j,k})) + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j+1,k})(u^n_{i,j+1,k} - u^n_{i,j,k}) - \\
& \qquad \frac{1}{2}(q_{i,j-1,k} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i,j-1,k})) + \\
& \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j,k+1})(u^n_{i,j,k+1} - u^n_{i,j,k}) - \\
& \qquad \frac{1}{2}(q_{i,j,k-1} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i,j,k-1})) + \\
& \Delta t^2 f^n_{i,j,k}\,.
\end{aligned}$$

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 $u^{-1}_{i,j,k} = u^1_{i,j,k} - 2\Delta tV_{i,j,k}$ in the scheme and solving for $u^1_{i,j,k}$.

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.

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 2.6.3.

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 $u^{n+1}_{i,-1} = u^{n+1}_{i,1}$.

2.12 Implementation

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

$$u_{tt} = c^2(u_{xx} + u_{yy}) + f(x, y, t), \qquad (x, y) \in \Omega,\ t \in (0, T], \tag{2.113}$$

$$u(x, y, 0) = I(x, y), \qquad (x, y) \in \Omega, \tag{2.114}$$

$$u_t(x, y, 0) = V(x, y), \qquad (x, y) \in \Omega, \tag{2.115}$$

$$u = 0, \qquad (x, y) \in \partial\Omega,\ t \in (0, T], \tag{2.116}$$

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

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu) + f]^n_{i,j},$$

which leads to an explicit updating formula to be implemented in a program:

$$u^{n+1} = -u^{n-1}_{i,j} + 2u^n_{i,j} + C_x^2(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}) + C_y^2(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}) + \Delta t^2 f^n_{i,j}, \tag{2.117}$$

for all interior mesh points $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$, and for $n \in \mathcal{I}^+_t$. The constants C_x and C_y are defined as

$$C_x = c\frac{\Delta t}{\Delta x}, \quad C_x = c\frac{\Delta t}{\Delta y}\,.$$

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, $n = 0$, (2.117) 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:

$$u^1 = u^0_{i,j} + \Delta t V_{i,j} + \frac{1}{2}C_x^2(u^0_{i+1,j} - 2u^0_{i,j} + u^0_{i-1,j}) + \frac{1}{2}C_y^2(u^0_{i,j+1} - 2u^0_{i,j} + u^0_{i,j-1}) + \frac{1}{2}\Delta t^2 f^n_{i,j}, \tag{2.118}$$

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 (2.117)
3. Set $u^1_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$
4. For $n = 1, 2, \ldots, N_t$:
   a. Find $u^{n+1}_{i,j}$ from (2.117) for all internal mesh points, $i \in \mathcal{I}^i_x$, $j \in \mathcal{I}^i_y$
   b. Set $u^{n+1}_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

2.12.1 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`.

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
```

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_1 = zeros((Nx+1,Ny+1))   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1))   # solution at t-2*dt
```

where $u^{n+1}_{i,j}$ corresponds to `u[i,j]`, $u^n_{i,j}$ to `u_1[i,j]`, and $u^{n-1}_{i,j}$ to `u_2[i,j]`

Index sets. It is also convenient to introduce the index sets (cf. Section 2.6.4)

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])
```

Computing the solution. Inserting the initial condition `I` in `u_1` and making a callback to the user in terms of the `user_action` function is a straightforward generalization of the 1D code from Section 2.1.6:

```
for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, 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 2.12.2.

The key finite difference formula (2.110) for updating the solution at a time level is implemented in a separate function as

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = 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 Iy[1:-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] = D1*u_1[i,j] - D2*u_2[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])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
```

```
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

The `step1` variable has been introduced to allow the formula to be reused for first step $u^1_{i,j}$:

```
u = advance_scalar(u, u_1, u_2, 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.

Finally, we remark that the `solver` function in the `wave2D_u0.py` code updates arrays for the next time step by switching references as described in Section 2.4.5. If the solution `u` is returned from `solver`, which it is not, it is important to set `u = u_1` after the time loop, otherwise `u` actually contains `u_2`.

2.12.2 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 (!).

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,:]
# or
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_1[:,:] = I(xv, yv)
```

One could also have written `u_1 = I(xv, yv)` and let `u_1` point to a new object, but vectorized operations often make use of direct insertion in the original array through `u_1[:,:]`, 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 $u^{n+1}_{i,j}$:

```
def advance_vectorized(u, u_1, u_2, 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_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
    u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
    u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[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]
    # Boundary condition u=0
    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

def quadratic(Nx, Ny, version):
    """Exact discrete solution of the scheme."""

    def exact_solution(x, y, t):
        return x*(Lx - x)*y*(Ly - y)*(1 + 0.5*t)

    def I(x, y):
        return exact_solution(x, y, 0)

    def V(x, y):
        return 0.5*exact_solution(x, y, 0)

    def f(x, y, t):
        return 2*c**2*(1 + 0.5*t)*(y*(Ly - y) + x*(Lx - x))

    Lx = 5;  Ly = 2
    c = 1.5
    dt = -1 # use longest possible steps
    T = 18

    def assert_no_error(u, x, xv, y, yv, t, n):
        u_e = exact_solution(xv, yv, t[n])
```

```
        diff = abs(u - u_e).max()
        tol = 1E-12
        msg = 'diff=%g, step %d, time=%g' % (diff, n, t[n])
        assert diff < tol, msg

    new_dt, cpu = solver(
        I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
        user_action=assert_no_error, version=version)
    return new_dt, cpu


def test_quadratic():
    # Test a series of meshes where Nx > Ny and Nx < Ny
    versions = 'scalar', 'vectorized', 'cython', 'f77', 'c_cy', 'c_f2py'
    for Nx in range(2, 6, 2):
        for Ny in range(2, 6, 2):
            for version in versions:
                print 'testing', version, 'for %dx%d mesh' % (Nx, Ny)
                quadratic(Nx, Ny, version)

def run_efficiency(nrefinements=4):
    def I(x, y):
        return sin(pi*x/Lx)*sin(pi*y/Ly)

    Lx = 10;  Ly = 10
    c = 1.5
    T = 100
    versions = ['scalar', 'vectorized', 'cython', 'f77',
                'c_f2py', 'c_cy']
    print ' '*15, ''.join(['%-13s' % v for v in versions])
    for Nx in 15, 30, 60, 120:
        cpu = {}
        for version in versions:
            dt, cpu_ = solver(I, None, None, c, Lx, Ly, Nx, Nx,
                              -1, T, user_action=None,
                              version=version)
            cpu[version] = cpu_
        cpu_min = min(list(cpu.values()))
        if cpu_min < 1E-6:
            print 'Ignored %dx%d grid (too small execution time)' \
                  % (Nx, Nx)
        else:
            cpu = {version: cpu[version]/cpu_min for version in cpu}
            print '%-15s' % '%dx%d' % (Nx, Nx),
            print ''.join(['%13.1f' % cpu[version] for version in versions])

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, =0 means no plot.
    """
    # Clean up plot files
    for name in glob('tmp_*.png'):
        os.remove(name)

    Lx = 10
    Ly = 10
    c = 1.0

    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)
```



```
    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):
        if t[n] == 0:
            time.sleep(2)
        if plot_method == 1:
            mesh(x, y, u, title='t=%g' % t[n], zlim=[-1,1],
                 caxis=[-1,1])
        elif plot_method == 2:
            surfc(xv, yv, u, title='t=%g' % t[n], zlim=[-1, 1],
                  colorbar=True, colormap=hot(), caxis=[-1,1],
                  shading='flat')
        elif plot_method == 3:
            print 'Experimental 3D matplotlib...under development...'
            #plt.clf()
            ax = fig.add_subplot(111, projection='3d')
            u_surf = ax.plot_surface(xv, yv, u, alpha=0.3)
            #ax.contourf(xv, yv, u, zdir='z', offset=-100, cmap=cm.coolwarm)
            #ax.set_zlim(-1, 1)
            # Remove old surface before drawing
            if u_surf is not None:
                ax.collections.remove(u_surf)
            plt.draw()
            time.sleep(1)
        if plot_method > 0:
            time.sleep(0) # pause between frames
            if save_plot:
                filename = 'tmp_%04d.png' % n
                savefig(filename)  # time consuming!

    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)



if __name__ == '__main__':
    test_quadratic()
```

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_1[1:,j] - u_1[:-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 makes 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])`.

2.12.3 Verification

Testing a quadratic solution. The 1D solution from Section 2.2.4 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 (2.30):

$$u_{\mathrm{e}}(x,y,t) = x(L_x-x)y(L_y-y)(1+\frac{1}{2}t)\,. \tag{2.119}$$

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 $[D_tD_t1]^n = 0$, $[D_tD_tt]^n = 0$, and $[D_tD_tt^2] = 2$, and use these to compute

$$\begin{aligned}[D_xD_xu_{\mathrm{e}}]^n_{i,j} &= [y(L_y-y)(1+\frac{1}{2}t)D_xD_xx(L_x-x)]^n_{i,j}\\ &= y_j(L_y-y_j)(1+\frac{1}{2}t_n)(-2)\,.\end{aligned}$$

A similar calculation must be carried out for the $[D_yD_yu_{\mathrm{e}}]^n_{i,j}$ and $[D_tD_tu_{\mathrm{e}}]^n_{i,j}$ terms. One must also show that the quadratic solution fits the special formula for $u^1_{i,j}$. The details are left as Exercise 2.15. The `test_quadratic` function in the `wave2D_u0.py` program implements this verification as a nose test.

2.13 Using classes to implement a simulator

- Introduce classes `Mesh`, `Function`, `Problem`, `Solver`, `Visualizer`, `File`

2.14 Exercises

Exercise 2.15: Check that a solution fulfills the discrete model

Carry out all mathematical details to show that (2.119) 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. Filename: `check_quadratic_solution`.

Project 2.16: Calculus with 2D mesh functions

The goal of this project is to redo Project 2.5 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. Filename: `mesh_calculus_2D`.

Exercise 2.17: 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 2.8 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). Filename: `wave2D_dn`.

Exercise 2.18: 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. Filename: `wave3D_u0`.

2.15 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 2.15.7). 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 information to fill in the details, while other readers will hopefully get some impression of the physics and approximations involved when establishing wave equation models.

2.15.1 Waves on a string

Figure 2.8 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 y direction is denoted by $u_i(t)$.

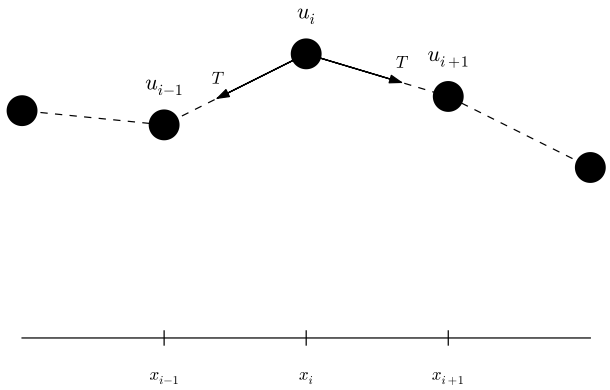


Fig. 2.8 Discrete string model with point masses connected by elastic strings.

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 2.8. 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

$$T\frac{\Delta x}{\Delta s_i} = T\frac{\Delta x}{\Delta s_{i+1}}, \tag{2.120}$$

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta s_{i+1}} - T\frac{\Delta u_i}{\Delta s_i}\,. \tag{2.121}$$

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 (2.120) is then simply the identity $T = T$, while (2.121) 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

$$\varrho_i u_i''(t) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,. \tag{2.122}$$

We can now choose to approximate u_i'' by a finite difference in time and get the discretized wave equation,

$$\varrho_i \frac{1}{\Delta t^2}\left(u_i^{n+1} - 2u_i^n - u_i^{n-1}\right) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right). \quad (2.123)$$

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 (2.122) approaches $\partial^2 u/\partial x^2$ as $\Delta x \to 0$. We end up with the continuous model for waves on a string:

$$\varrho\frac{\partial^2 u}{\partial t^2} = T\frac{\partial^2 u}{\partial x^2}. \quad (2.124)$$

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

$$\frac{\partial^2 u}{\partial t^2} = c^2(x)\frac{\partial^2 u}{\partial x^2}. \quad (2.125)$$

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 $u = 0$.

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

$$\varrho\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = T\frac{\partial^2 u}{\partial x^2}. \quad (2.126)$$

The parameter $b \geq 0$ is small for most wave phenomena, but the damping effect may become significant in long time simulations.

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. 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 (2.124). In the model (2.125) we would add a term $f(x,t) = \tilde{f}(x,y)/\varrho(x)$.

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.

2.15.2 Waves on a membrane

hpl 12: Must write the memberane model. Easiest to use Navier.

2.15.3 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,

$$\varrho\boldsymbol{u}_{tt} = \nabla\cdot\boldsymbol{\sigma} + \varrho\boldsymbol{f}, \quad (2.127)$$

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, 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, (2.127) simplifies to

$$\varrho\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(E\frac{\partial u}{\partial x}\right). \quad (2.128)$$

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.

2.15.4 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,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{2.129}$$

and a constitutive law relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$, often Hooke's generalized law,

$$\boldsymbol{\sigma} = K\nabla \cdot \boldsymbol{u}\, \boldsymbol{I} + G(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3}\nabla \cdot \boldsymbol{u}\, \boldsymbol{I}) . \tag{2.130}$$

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 $\boldsymbol{f}$, (2.129) becomes

$$\varrho \boldsymbol{u}_{tt} = \nabla(K\nabla \cdot \boldsymbol{u}) \tag{2.131}$$

Introducing p as a pressure via

$$p = -K\nabla \cdot \boldsymbol{u}, \tag{2.132}$$

and dividing (2.131) by ϱ, we get

$$\boldsymbol{u}_{tt} = -\frac{1}{\varrho}\nabla p . \tag{2.133}$$

Taking the divergence of this equation, using $\nabla \cdot \boldsymbol{u} = -p/K$ from (2.132), gives the *acoustic approximation to elastic waves*:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) . \tag{2.134}$$

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:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) + s . \tag{2.135}$$

A common additional approximation of (2.135) 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

$$p_{tt} = \frac{K}{\varrho}\nabla^2 p + s . \tag{2.136}$$

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 $p = p_0$.

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 (2.136) to

$$p_{tt} = c_z^2 p_{zz} + c_h^2(p_{xx} + p_{yy}) + s . \tag{2.137}$$

2.15.5 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:

$$\varrho_t + \nabla\cdot(\varrho\boldsymbol{u}) = 0, \tag{2.138}$$
$$\varrho\boldsymbol{u}_t + \varrho\boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p, \tag{2.139}$$
$$\varrho = \varrho(p)\,. \tag{2.140}$$

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 (2.139) reflects mass balance, (2.138) is Newton's second law for a fluid, with frictional and body forces omitted, and (2.140) is a constitutive law relating density to pressure by thermodynamic considerations. A typical model for (2.140) is the so-called isentropic relation, valid for adiabatic processes where there is no heat transfer:

$$\varrho = \varrho_0\left(\frac{p}{p_0}\right)^{1/\gamma}. \tag{2.141}$$

Here, p_0 and ϱ_0 are references 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 (2.138) and (2.139), 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:

$$\varrho_t + \varrho_0\nabla\cdot\boldsymbol{u} = 0, \tag{2.142}$$
$$\varrho_0\boldsymbol{u}_t = -\nabla p\,. \tag{2.143}$$

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

$$p_t + \gamma p_0\nabla\cdot\boldsymbol{u} = 0, \tag{2.144}$$
$$\boldsymbol{u}_t = -\frac{1}{\varrho_0}\nabla p,\,. \tag{2.145}$$

Taking the divergence of (2.145) and differentiating (2.144) 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:

$$p_{tt} = c^2\nabla^2 p, \tag{2.146}$$

where $c = \sqrt{\gamma p_0/\varrho_0}$ is the speed of sound in the fluid.

2.15.6 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

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right) + f(r,t), \quad r\in(0,R),\ t>0\,. \tag{2.147}$$

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

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right)$$

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

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial r}\left(c^2(r)\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{dc^2}{dr}v + rf(r,t), \quad r\in(0,R),\ t>0\,. \tag{2.148}$$

In the case of constant wave velocity c, this equation reduces to the wave equation in a single Cartesian coordinate called r:

$$\frac{\partial^2 v}{\partial t^2} = c^2\frac{\partial^2 v}{\partial r^2} + rf(r,t), \quad r\in(0,R),\ t>0\,. \tag{2.149}$$

That is, any program for solving the one-dimensional wave equation in a Cartesian coordinate system can be used to solve (2.149), 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

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r=0,$$

implying $v(0,t)=0$ as a necessary condition. For practical applications, we exclude $r=0$ from the domain and assume that some boundary condition is assigned at $r=\epsilon$, for some $\epsilon>0$.

2.15.7 The linear shallow water equations

The next example considers water waves whose wavelengths are much lager 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:

$$\eta_t = -(Hu)_x - (Hv)_x \tag{2.150}$$
$$u_t = -g\eta_x, \tag{2.151}$$
$$v_t = -g\eta_y, \tag{2.152}$$

where g is the acceleration of gravity. Equation (2.150) corresponds to mass balance while the other two are derived from momentum balance (Newton's second law).

The initial conditions associated with (2.150)-(2.152) 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 (2.150), (2.151), and (2.152) can be transformed to a standard, linear wave equation. First, multiply (2.151) and (2.152) by H, differentiate (2.151)) with respect to x and (2.152) with respect to y. Sec-

ond, differentiate (2.150) 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 results in a standard, linear wave equation for η:

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla\cdot(gH\nabla\eta)\,. \tag{2.153}$$

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 (2.150) 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 (2.150)-(2.152) 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 (2.150). 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 (2.153), because of the moving (accelerating) bottom:

$$\eta_{tt} = \nabla\cdot(gH\nabla\eta) + B_{tt}\,. \tag{2.154}$$

The reduction of (2.154) 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 ($\partial/\partial y = 0$):

$$\eta_{tt} = (gH\eta_x)_x + B_{tt}\,. \tag{2.155}$$

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 (2.151) and (2.152), respectively, on the right-hand sides.

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 (2.151) and $C_B\sqrt{u^2+v^2}v$ on the right-hand side of (2.152). Note that in this case the PDEs (2.151) and (2.152) become nonlinear and the elimination of u and v to arrive at a 2nd-order wave equation for η is not possible anymore.

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 (2.151) and $-fu$ on the right-hand side of (2.152). 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.

2.15.8 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

$$A(x,t) = 2\pi\int_0^{R(x,t)} r\,dr, \tag{2.156}$$

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

$$Q(x,t) = A(x,t)v_x(x,t) \tag{2.157}$$

Mass balance and Newton's second law lead to the PDEs

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{2.158}$$

$$\frac{\partial Q}{\partial t} + \frac{\gamma+2}{\gamma+1}\frac{\partial}{\partial x}\left(\frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} = -2\pi(\gamma+2)\frac{\mu}{\varrho}\frac{Q}{A}, \tag{2.159}$$

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 (2.158) and (2.159). A third equation is needed to relate the flow to the deformations of the wall. A common form for this equation is

$$\frac{\partial P}{\partial t} + \frac{1}{C}\frac{\partial Q}{\partial x} = 0, \tag{2.160}$$

where C is the compliance of the wall, given by the constitutive relation

$$C = \frac{\partial A}{\partial P} + \frac{\partial A}{\partial t}, \tag{2.161}$$

which require 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

$$C = \frac{\partial A}{\partial P} = \frac{2(1-\nu^2)A_0}{\pi h E}\sqrt{A_0} + 2\left(\frac{(1-\nu^2)A_0}{\pi h E}\right)^2 (P - P_0)\,. \tag{2.162}$$

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

$$C = \frac{\partial A}{\partial P} = \frac{A_0}{\beta P}\,. \tag{2.163}$$

Reduction to the standard wave equation. It is not uncommon to neglect the viscous term on the right-hand side of (2.159) and also the quadratic term with Q^2 on the left-hand side. The reduced equations (2.159) and (2.160) form a first-order linear wave equation system:

$$C\frac{\partial P}{\partial t} = -\frac{\partial Q}{\partial x}, \tag{2.164}$$

$$\frac{\partial Q}{\partial t} = -\frac{A}{\varrho}\frac{\partial P}{\partial x}\,. \tag{2.165}$$

These can be combined into standard 1D wave equation PDE by differentiating the first equation with respect 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

$$\frac{\partial^2 Q}{\partial t^2} = c^2\frac{\partial^2 Q}{\partial x^2}, \quad c = \sqrt{\frac{A}{\varrho C}}, \tag{2.166}$$

where the A and C in the expression for c are taken as constant reference values.

2.15.9 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:

$$\frac{\partial^2 \boldsymbol{E}}{\partial t^2} = c^2\nabla^2\boldsymbol{E}, \tag{2.167}$$

$$\frac{\partial^2 \boldsymbol{B}}{\partial t^2} = c^2\nabla^2\boldsymbol{B}, \tag{2.168}$$

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.

2.16 Exercises

Exercise 2.19: 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.

Hint. According to Section 2.15.1, 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.
Filename: `wave1D_u0_sv_discont`.

Exercise 2.20: Simulate damped waves on a string

Formulate a mathematical model for damped waves on a string. Use data from Section 2.3.5, 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.
Filename: `wave1D_u0_sv_damping`.

Exercise 2.21: 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 (2.128) from Section 2.15.3. 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). Filename: `wave_rod`.

Exercise 2.22: Simulate spherical waves

Implement a model for spherically symmetric waves using the method described in Section 2.15.6. 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 2.11. 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.

Hint. 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.
Filename: `wave1D_spherical`.

Problem 2.23: Earthquake-generated tsunami over a subsea hill

A subsea earthquake leads to an immediate lift of the water surface, see Figure 2.9. The lifted water surface splits into two tsunamis, one

traveling to the right and one to the left, as depicted in Figure 2.10. Since tsunamis are normally very long waves, compared to the depth, with a small amplitude, compared to the wave length, the wave equation model described in Section 2.15.7 is relevant:

$$\eta_{tt} = (gH(x)\eta_x)_x,$$

where g is the acceleration of gravity, and $H(x)$ is the still water depth.

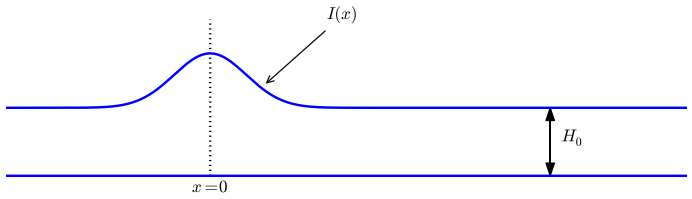


Fig. 2.9 Sketch of initial water surface due to a subsea earthquake.

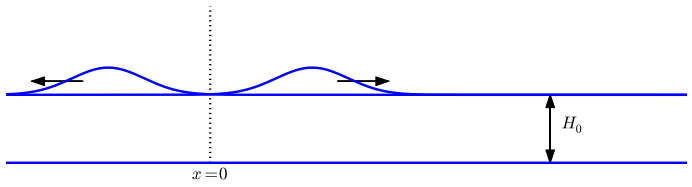


Fig. 2.10 An initial surface elevation is split into two waves.

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 2.11 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,

$$I(x; I_0, I_a, I_m, I_s) = I_0 + I_a \exp\left(-\left(\frac{x - I_m}{I_s}\right)^2\right), \tag{2.169}$$

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 2.11. The wave speed $c = \sqrt{gH(x)} = \sqrt{g(H_0 - B(x))}$ will then be reduced in the shallow water above the hill.

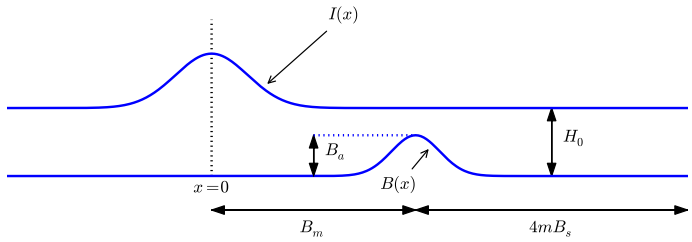


Fig. 2.11 Sketch of an earthquake-generated tsunami passing over a subsea hill.

One possible form of the hill is a Gaussian function,

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \exp\left(-\left(\frac{x - B_m}{B_s}\right)^2\right), \tag{2.170}$$

but many other shapes are also possible, e.g., a "cosine hat" where

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_m}{2B_s}\right), \tag{2.171}$$

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:

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a, \tag{2.172}$$

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: (2.170), (2.171), (2.172), 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. Filename: `tsunami1D_hill`.

Problem 2.24: Earthquake-generated tsunami over a 3D hill

This problem extends Problem 2.23 to a three-dimensional wave phenomenon, governed by the 2D PDE (2.153). We assume that the earthquake arise 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 2.23. 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 (2.170):

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s, b) = B_0 + B_a \exp\left(-\left(\frac{x - B_{mx}}{B_s}\right)^2 - \left(\frac{y - B_{my}}{bB_s}\right)^2\right), \tag{2.173}$$

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.

The "cosine hat" (2.171) can also be generalized to

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_{mx}}{2B_s}\right) \cos\left(\pi \frac{y - B_{my}}{2B_s}\right), \tag{2.174}$$

when $0 \leq \sqrt{x^2 + y^2} \leq B_s$ and $B = B_0$ outside this circle.

A box-shaped obstacle means that

$$B(x; B_0, B_a, B_m, B_s, b) = B_0 + B_a \tag{2.175}$$

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. Filename: `tsunami2D_hill`.

Problem 2.25: Investigate Matplotlib for visualization

Play with native Matplotlib 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. Filename: `tsunami2D_hill_mpl`.

Problem 2.26: Investigate visualization packages

Create some fancy 3D visualization of the water waves *and* the subsea hill in Problem 2.24. Try to make the hill transparent. Possible visualization tools are Mayavi, Paraview, and OpenDX. Filename: `tsunami2D_hill_viz`.

Problem 2.27: Implement loops in compiled languages

Extend the program from Problem 2.24 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. Filename: `tsunami2D_hill_compiled`.

Exercise 2.28: Simulate seismic waves in 2D

The goal of this exercise is to simulate seismic waves using the PDE model (2.137) 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. Filename: `seismic2D`.

Project 2.29: Model 3D acoustic waves in a room

The equation for sound waves in air is derived in Section 2.15.5 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

$$p_t t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f, \tag{2.176}$$

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:

$$p_t t + bp_t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f\,. \tag{2.177}$$

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. Filename: `acoustics`.

Project 2.30: Solve a 1D transport equation

We shall study the wave equation

$$u_t + cu_x = 0, \quad x \in (0, L],\ t \in (0, T], \tag{2.178}$$

with initial condition

$$u(x, 0) = I(x), \quad x \in [0, L], \tag{2.179}$$

and *one* periodic boundary condition

$$u(0, t) = u(L, t)\,. \tag{2.180}$$

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 of the spatial derivative is of first order only.

Physical interpretation. The parameter c can be constant or variable, $c = c(x)$. The equation (2.178) 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 $a = \text{const}$,

$$u(x, t) = I(x - ct) \tag{2.181}$$

fulfills the PDE as well as the initial and boundary condition (provided $I(0) = I(L)$).

A widely used numerical scheme for (2.178) applies a forward difference in time and a backward difference in space when $c > 0$:

$$[D_t^+ u + cD_x^- u = 0]_i^n\,. \tag{2.182}$$

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 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:

$$I(x) = \left[\sin\left(\pi\frac{x}{L}\right)\right]^{2n} \tag{2.183}$$

where n is an integer, typically $n = 5$, and

$$I(x) = \exp\left(-\frac{(x - L/2)^2}{2\sigma 2}\right). \tag{2.184}$$

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 the $\Delta t/(c\Delta x) = 1$, 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

$$u(x,t) = f(C(x) - t), \tag{2.185}$$

where

$$C'(x) = \frac{1}{c(x)},$$

is a solution of (2.178) for any differentiable function f.

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 $[0, L]$.

To compute (2.185) 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 seeks 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 `wrap2callable` in `scitools.std` 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 scitools.std import wrap2callable
    g_inverse = wrap2callable((y, x))
    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

$$C(x_{i+1}) = C(x_i) + \int_{x_i}^{x_{i+1}} \frac{dx}{c} \approx h\left(\frac{1}{2}\frac{1}{c(x_i)} + \frac{1}{2}\frac{1}{c(x_{i+1})} + \sum_{j=1}^{m-1}\frac{1}{c(x_i + jh)}\right), \tag{2.186}$$

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 (2.186) is a *difference equation* which we can solve by repeatedly applying (2.186) 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 (2.183) and (2.30). 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 (2.183) or (2.30).

The PDE $u_t + cu_x = 0$ expresses that the initial condition $I(x)$ is transported with velocity $c(x)$.
Filename: `advec1D`.

Problem 2.31: General analytical solution of a 1D damped wave equation

We consider an initial-boundary value problem for the damped wave equation:

$$
\begin{aligned}
u_{tt} + bu_t &= c^2 u_{xx}, \qquad && x \in (0,L),\ t \in (0,T] \\
u(0,t) &= 0, \\
u(L,t) &= 0, \\
u(x,0) &= I(x), \\
u_t(x,0) &= V(x)\,.
\end{aligned}
$$

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

$$
I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{2.187}
$$

g) Implement the series for $u(x,t)$ in a function `u_series(x, t, tol=1E-10)`, 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 $u_x(L,t) = 0$?
Filename: `damped_wave1D`.

Problem 2.32: General analytical solution of a 2D damped wave equation

Carry out Problem 2.31 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 $u(x,y,t) = X(x)Y(y)T(t)$. Filename: `damped_wave2D`.

Diffusion equations 3

3.1 An explicit method for the 1D diffusion equation

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, but 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 - the solution is basically a moving initial condition. The standard wave equation $u_{tt} = c^2 u_{xx}$ has solutions that propagates 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 a explicit scheme, but the time step restrictions soon become much less favorable than for an explicit scheme for 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.

3.1.1 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

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f, \quad x \in (0,L),\ t \in (0,T] \tag{3.1}$$

$$u(x,0) = I(x), \qquad x \in [0,L] \tag{3.2}$$

$$u(0,t) = 0, \qquad t > 0, \tag{3.3}$$

$$u(L,t) = 0, \qquad t > 0\,. \tag{3.4}$$

Equation (3.1) is known as a one-dimensional *diffusion equation*, also often referred to as a *heat equation*. 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*. The parameter α must be given and is referred to as the *diffusion coefficient*. We have added a source term $f = f(x,t)$ for convenience 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.

3.1.2 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

$$\frac{\partial}{\partial t}u(x_i, t_n) = \alpha\frac{\partial^2}{\partial x^2}u(x_i, t_n) + f(x_i, t_n), \tag{3.5}$$

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:

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n\,. \tag{3.6}$$

Written out,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n\,. \tag{3.7}$$

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 (3.7). Solving with respect to this unknown is easy:

$$u_i^{n+1} = u_i^n + F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t f_i^n, \tag{3.8}$$

where we have introduced the *mesh Fourier number*:

$$F = \alpha\frac{\Delta t}{\Delta x^2}\,. \tag{3.9}$$

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. All the properties of the numerical method are critically dependent upon the value of F (see Section 3.3 for details).

The computational algorithm then becomes

1. compute $u_i^0 = I(x_i)$for $i = 0, \ldots, N_x$
2. for $n = 0, 1, \ldots, N_t$:
   a. apply (3.8) for all the internal spatial points $i = 1, \ldots, N_x - 1$
   b. set the boundary values $u_i^{n+1} = 0$ for $i = 0$ and $i = N_x$

The algorithm is compactly fully specified in Python:

```
x = linspace(0, L, Nx+1)    # mesh points in space
dx = x[1] - x[0]
t = linspace(0, T, Nt+1)    # mesh points in time
dt = t[1] - t[0]
F = a*dt/dx**2
u   = zeros(Nx+1)           # unknown u at new time level
u_1 = zeros(Nx+1)           # u at the previous time level

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

for n in range(0, Nt):
    # Compute u at inner mesh points
    for i in range(1, Nx):
        u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               f(x[i], t[n])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0

    # Update u_1 before next step
    u_1[:]= 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 3.3.

3.1.3 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 time

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.
    f must be a Python function of x and t. If None, a
    default f=0 is used.
    """
    import time
    t0 = time.clock()

    dt = float(dt)                # avoid integer division
    Nt = int(round(T/dt))
    t = np.linspace(0, T, 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

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

    u   = np.zeros(Nx+1)
    u_1 = np.zeros(Nx+1)

    # Set initial condition u(x,0) = I(x)
    for i in range(0, Nx+1):
        u_1[i] = I(x[i])

    for n in range(0, Nt):
        # Compute u at inner mesh points
        for i in range(1, Nx):
            u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                   dt*f(x[i], t[n])

        # Insert boundary conditions
        u[0] = 0;  u[Nx] = 0

        # Switch variables before next step
        u_1, u = u, u_1

    t1 = time.clock()
    # Return u_1 as u since we set u_1=u above
    return u_1, x, t, t1-t0
```

A faster version, based on vectorization of the finite difference scheme, is available in the function solver_FE. The vectorized version replaces the explicit loop


```
for i in range(1, Nx):
    u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) \
           + dt*f(x[i], t[n])
```

by arithmetics on displaced slices of the u array:

```
u[1:Nx] = u_1[1:Nx] + F*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1]) \
          + dt*f(x[1:Nx], t[n])
# or
u[1:-1] = u_1[1:-1] + F*(u_1[0:-2] - 2*u_1[1:-1] + u_1[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. Apart from the vectorized loop over the spatial mesh points, the callback function, and a bit more complicated setting of the source f it is not specified (None), the solver_FE is identical to solver_FE_simple above:

```
def solver_FE(I, a, f, L, dt, F, T,
              user_action=None, version='scalar'):
    """
    Vectorized implementation of solver_FE_simple.
    If version='vectorized', f must be a vectorized
    function of x and t (if f is None, a default version
    f=0 is made).
    """
    import time
    t0 = time.clock()

    dt = float(dt)                # avoid integer division
    Nt = int(round(T/dt))
    t = np.linspace(0, T, 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

    if f is None:
        if version == 'scalar':
            f = lambda x, t: 0
        else:
            f = lambda x, t: np.zeros(len(x))

    u   = np.zeros(Nx+1)   # solution array
    u_1 = np.zeros(Nx+1)   # solution at t-dt

    # Set initial condition
    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 n in range(0, Nt):
        # Update all inner points
        if version == 'scalar':
            for i in range(1, Nx):
                u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                       dt*f(x[i], t[n])

        elif version == 'vectorized':
            u[1:Nx] = u_1[1:Nx] +  \
                      F*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1]) + \
                      dt*f(x[1:Nx], t[n])
        else:
            raise ValueError('version=%s' % version)

        # Insert boundary conditions
        u[0] = 0;  u[Nx] = 0
        if user_action is not None:
            user_action(u, x, t, n+1)

        # Update u_1 before next step
        #u_1[:]= u
        u_1, u = u, u_1

    t1 = time.clock()
    # Return u_1 as solution since we set u_1=u above
    return u_1, x, t, t1-t0
```

3.1.4 Verification

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 A.4 we can easily check that the manufactured `u` fulfills the scheme:

$$\begin{aligned}[D_t^+ u = \alpha D_x D_x u + f]_i^n &= [5x(L-x)D_t^+ t = 5t\alpha D_x D_x(xL - x^2) + 10\alpha t + 5x(L-x)]_i^n \\ &= [5x(L-x) = 5t\alpha(-2) + 10\alpha t + 5x(L-x)]_i^n\,.\end{aligned}$$

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`.

The numerical code will need to access the `u` and `f` above as Python function. 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():
    # Define u_exact, f, I as explained above

    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
```

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 $F > 1/2$).

3.1.5 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 [2] 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(solver='solver_FE', F=0.5, dt=0.0002):
    """Plug profile as initial condition."""
    L = 1.
    a = 1
    T = 0.1

    def I(x):
        return 0 if abs(x-L/2.0) > 0.1 else 1

    u, cpu = viz(I, a, None, L, dt, F, T, umin=-0.1, umax=1.1,
                 solver=solver, animate=True, framefiles=True)
    return u

def gaussian(solver='solver_FE', F=0.5, dt=0.0002, sigma=0.1):
    """Gaussian profile as initial condition."""
    L = 1.
    a = 1
    T = 0.1

    def I(x):
        return np.exp(-0.5*((x-L/2.0)**2)/sigma**2)

    u, cpu = viz(I, a, None, L, dt, F, T, umin=-0.1, umax=1.1,
                 solver=solver, animate=True, framefiles=True)
    return u
```

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, f, L, dt, F, T, umin, umax,
        solver='solver_FE', animate=True, framefiles=True):

    solutions = []
    from scitools.std import plot, savefig

    def plot_u(u, x, t, n):
        if n == 0:
            # Store x and t first in solutions
            solutions.append(x)
            solutions.append(t)
        solutions.append(u.copy())
        plot(x, u, 'r-', axis=[0, L, umin, umax], title='t=%f' % t[n])
        if framefiles:
            savefig('tmp_frame%04d.png' % n)
            if n in [0, 2, 5, 10, 25, 50, 250, 500]: savefig('tmp_frame%04d.pdf' % n)
        if t[n] == 0:
            time.sleep(2)
        elif not framefiles:
            # It takes time to write files so pause is needed
            # for screen only animation
            time.sleep(0.2)

    user_action = plot_u if animate else lambda u,x,t,n: None

    u, x, t, cpu = eval(solver)(I, a, f, L, dt, F, T,
                                user_action=user_action)
    return solutions, 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.

hpl 13: Better to show the scalable file solution here?

Our experiments employs 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

```
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 3.1 at four time levels (see also a movie).

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 3.2 (and a movie).

Increasing F slightly beyond the limit 0.5, to $F = 0.51$, gives growing, non-physical instabilities, as seen in 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 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 ($F \geq \frac{1}{2}$).

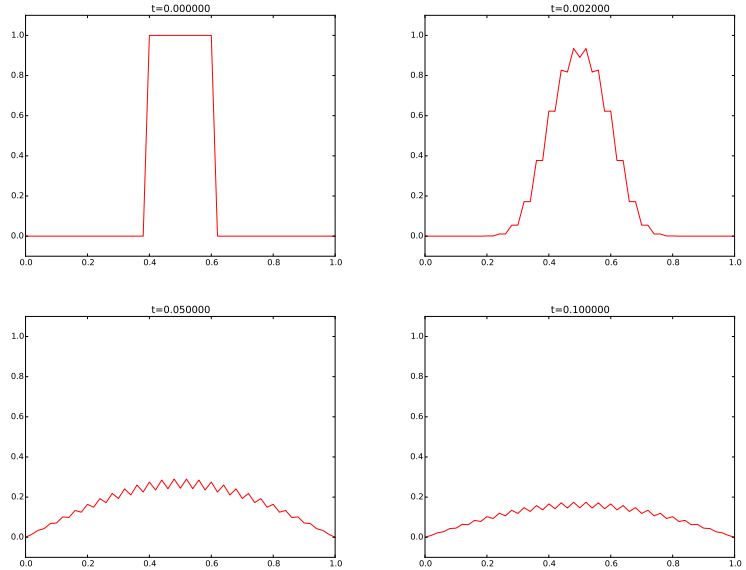


Fig. 3.1 Forward Euler scheme for $F = 0.5$.

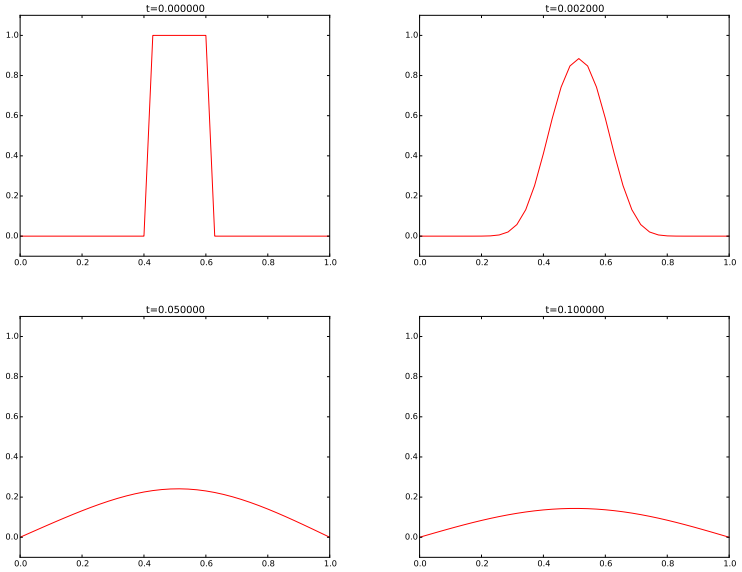


Fig. 3.2 Forward Euler scheme for $F = 0.25$.

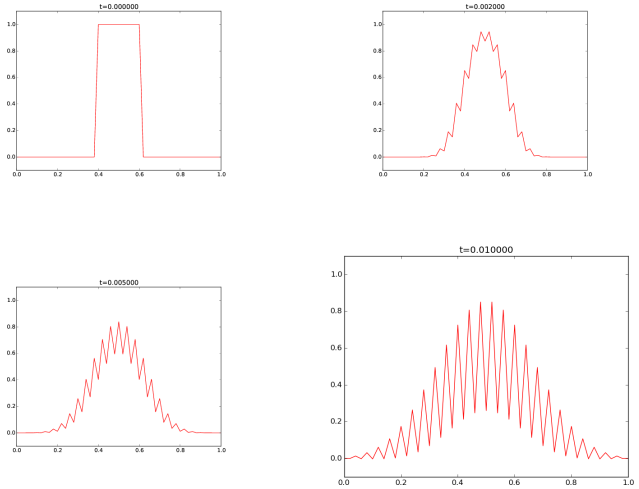


Fig. 3.3 Forward Euler scheme for $F = 0.51$.

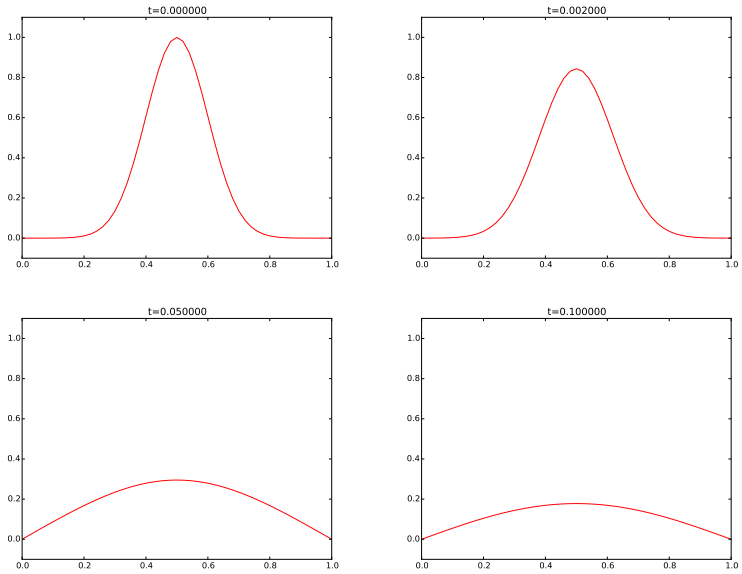


Fig. 3.4 Forward Euler scheme for $F = 0.5$.

3.2 Implicit methods for the 1D diffusion equation

Simulations with the Forward Euler scheme shows 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. By using *implicit schemes*, which lead to a coupled system 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.

3.2.1 Backward Euler scheme

We now apply a backward difference in time in (3.5), but the same central difference in space:

$$[D_t^- u = D_x D_x u + f]_i^n, \tag{3.10}$$

which written out reads

$$\frac{u_i^n - u_i^{n-1}}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \,. \tag{3.11}$$

Now we assume u_i^{n-1} is computed, but 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 (3.11) written for $i = 1, \ldots, Nx - 1 = 1, 2$ becomes

$$\frac{u_1^n - u_1^{n-1}}{\Delta t} = \alpha \frac{u_2^n - 2u_1^n + u_0^n}{\Delta x^2} + f_1^n \tag{3.12}$$

$$\frac{u_2^n - u_2^{n-1}}{\Delta t} = \alpha \frac{u_3^n - 2u_2^n + u_1^n}{\Delta x^2} + f_2^n \tag{3.13}$$

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

$$(1+2F)\,u_1^n - Fu_2^n = u_1^{n-1} + \Delta t f_1^n, \tag{3.14}$$
$$-Fu_1^n + (1+2F)\,u_2^n = u_2^{n-1} + \Delta t f_2^n\,. \tag{3.15}$$

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

$$\begin{pmatrix} 1+2F & -F \\ -F & 1+2F \end{pmatrix}\begin{pmatrix} u_1^n \\ u_2^n \end{pmatrix} = \begin{pmatrix} u_1^{n-1} + \Delta t f_1^n \\ u_2^{n-1} + \Delta t f_2^n \end{pmatrix}$$

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.

In the general case, (3.11) gives rise to a coupled $(Nx-1)\times(Nx-1)$ system of algebraic equations for all the unknown u_i^n at the interior spatial points $i = 1,\ldots,Nx-1$. Collecting the unknowns on the left-hand side, (3.11) can be written

$$-Fu_{i-1}^n + (1+2F)\,u_i^n - Fu_{i+1}^n = u_{i-1}^{n-1}, \tag{3.16}$$

for $i = 1,\ldots,Nx-1$. One can either view these equations as a system for 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 unknown and we must add the boundary equations to the N_x-1 equations in (3.16):

$$u_0^n = 0, \tag{3.17}$$
$$u_{N_x}^n = 0\,. \tag{3.18}$$

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 (3.16) and (3.17)–(3.18) 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:

$$A = \begin{pmatrix}
A_{0,0} & A_{0,1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
A_{1,0} & A_{1,1} & 0 & \ddots & & & & & \vdots \\
0 & A_{2,1} & A_{2,2} & A_{2,3} & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & A_{i,i-1} & A_{i,i} & A_{i,i+1} & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & \ddots & \ddots & A_{N_x-1,N_x} \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & A_{N_x,N_x-1} & A_{N_x,N_x}
\end{pmatrix} \tag{3.19}$$

The nonzero elements are given by

$$A_{i,i-1} = -F \tag{3.20}$$
$$A_{i,i} = 1+2F \tag{3.21}$$
$$A_{i,i+1} = -F \tag{3.22}$$

for the equations for internal points, $i = 1,\ldots,N_x-1$. The equations for the boundary points correspond to

$$A_{0,0} = 1, \tag{3.23}$$
$$A_{0,1} = 0, \tag{3.24}$$
$$A_{N_x,N_x-1} = 0, \tag{3.25}$$
$$A_{N_x,N_x} = 1\,. \tag{3.26}$$

The right-hand side b is written as

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{N_x} \end{pmatrix} \tag{3.27}$$

with

$$b_0 = 0, \tag{3.28}$$
$$b_i = u_i^{n-1}, \quad i = 1, \ldots, N_x - 1, \tag{3.29}$$
$$b_{N_x} = 0\,. \tag{3.30}$$

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 = linspace(0, L, Nx+1)   # mesh points in space
dx = x[1] - x[0]
t = linspace(0, T, N+1)    # mesh points in time
u   = zeros(Nx+1)          # unknown u at new time level
u_1 = zeros(Nx+1)          # u at the previous time level

# Data structures for the linear system
A = zeros((Nx+1, Nx+1))
b = 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

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

import scipy.linalg

for n in range(0, Nt):
    # Compute b and solve linear system
    for i in range(1, Nx):
        b[i] = -u_1[i]
    b[0] = b[Nx] = 0
    u[:] = scipy.linalg.solve(A, b)

    # Update u_1 before next step
    u_1[:] = u
```


3.2.2 Sparse matrix implementation

We have seen from (3.19) 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, the work and storage demands can be proportional to N_x only.

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 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.

```
# Representation of sparse matrix and right-hand side
main  = zeros(Nx+1)
lower = zeros(Nx-1)
upper = zeros(Nx-1)
b     = zeros(Nx+1)

# Precompute sparse matrix
main[:] = 1 + 2*F
lower[:] = -F  #1
upper[:] = -F  #1
# Insert boundary conditions
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

# Set initial condition
for i in range(0,Nx+1):
    u_1[i] = I(x[i])

for n in range(0, Nt):
    b = u_1
    b[0] = b[-1] = 0.0  # boundary conditions
    u[:] = scipy.sparse.linalg.spsolve(A, b)
    u_1[:] = 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 3.1.2, the functions `plug` and `gaussian` runs 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.

3.2.3 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 in 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

$$\frac{1}{\Delta x^2}\left(u_{i-1}^{n+\frac{1}{2}} - 2u_i^{n+\frac{1}{2}} + u_{i+1}^{n+\frac{1}{2}}\right) + f_i^{n+\frac{1}{2}}\,.$$

This expression is problematic since $u_i^{n+\frac{1}{2}}$ is not one of the unknown we compute. A possibility is to replace $u_i^{n+\frac{1}{2}}$ by an arithmetic average:

$$u_i^{n+\frac{1}{2}} \approx \frac{1}{2}\left(u_i^n + u_i^{n+1}\right)\,.$$

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

$$u_i^{n+1} - \frac{1}{2}F(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}) = u_i^n + \frac{1}{2}F(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \\ \frac{1}{2}f_i^{n+1} + \frac{1}{2}f_i^n\,. \tag{3.31}$$

Also here, as in the Backward Euler scheme, the new unknowns u_{i-1}^{n+1}, u_i^{n+1}, and u_{i+1}^{n+1} are coupled in a linear system $AU = b$, where A has the same structure as in (3.19), but with slightly different entries:

$$A_{i,i-1} = -\frac{1}{2}F \tag{3.32}$$

$$A_{i,i} = \frac{1}{2} + F \tag{3.33}$$

$$A_{i,i+1} = -\frac{1}{2}F \tag{3.34}$$

for the equations for internal points, $i = 1, \ldots, N_x - 1$. The equations for the boundary points correspond to

$$A_{0,0} = 1, \tag{3.35}$$

$$A_{0,1} = 0, \tag{3.36}$$

$$A_{N_x,N_x-1} = 0, \tag{3.37}$$

$$A_{N_x,N_x} = 1\,. \tag{3.38}$$

The right-hand side b has entries

$$b_0 = 0, \tag{3.39}$$

$$b_i = u_i^{n-1} + \frac{1}{2}(f_i^n + f_i^{n+1}), \quad i = 1, \ldots, N_x - 1, \tag{3.40}$$

$$b_{N_x} = 0\,. \tag{3.41}$$

3.2.4 The θ rule

For the equation

$$\frac{\partial u}{\partial t} = G(u),$$

where $G(u)$ is some a 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

Applied to the 1D diffusion problem, the θ-rule gives

$$\frac{u_i^{n+1}-u_i^n}{\Delta t} = \alpha\left(\theta\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{\Delta x^2} + (1-\theta)\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2}\right) + \theta f_i^{n+1} + (1-\theta)f_i^n\,.$$

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.

3.2.5 Experiments

We can repeat the experiments from Section 3.1.5 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 in practice often means choosing larger time steps.

The Backward Euler scheme always produces smooth solutions for any F. Figure 3.5 shows one example. The Crank-Nicolson method produces smooth solutions for small F, $F \leq \frac{1}{2}$, but small noise is more and more evident as F increases. Figures 3.6 and 3.7 demonstrates the effect for

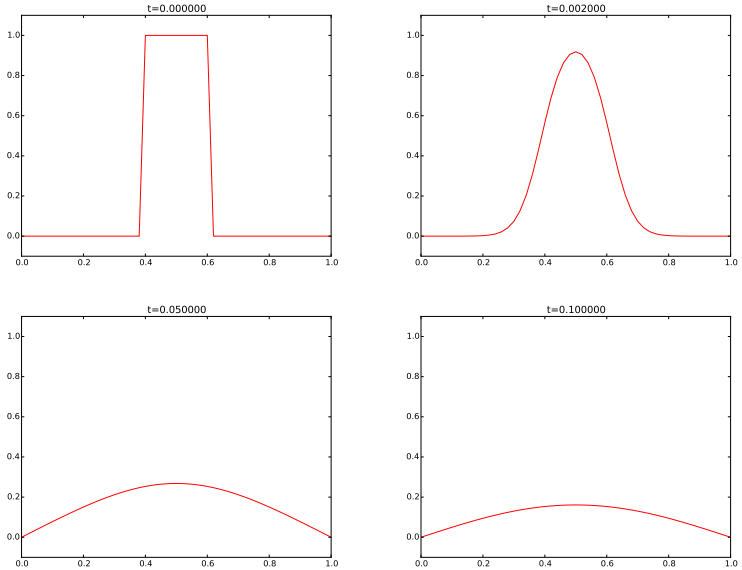


Fig. 3.5 Backward Euler scheme for $F = 0.5$.

$F = 3$ and $F = 10$, respectively. Section 3.3 explains why such noise occur.

3.2.6 The Laplace and Poisson equation

The Laplace equation, $\nabla^2 u = 0$, or 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 reach 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 $t \to$ 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 the limit the Backward Euler scheme gives

$$-\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{\Delta x^2} = f_i^{n+1},$$

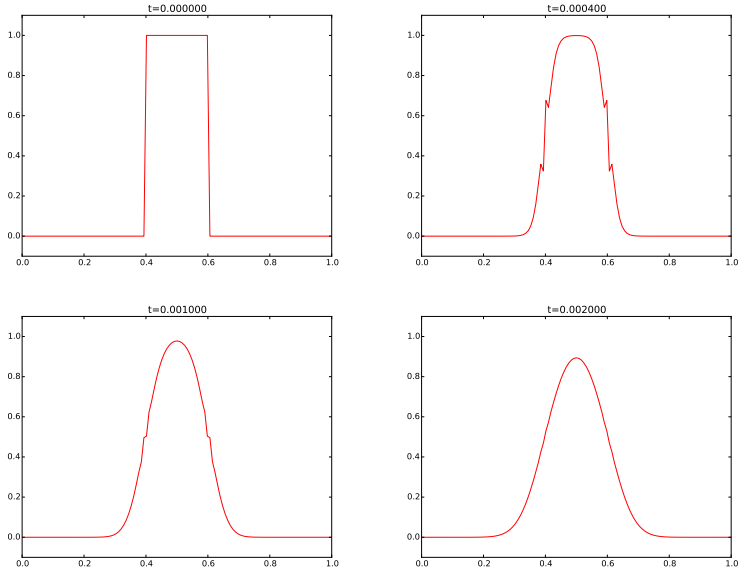


Fig. 3.6 Crank-Nicolson scheme for $F = 3$.

which is nothing but the discretization $[-D_xD_xu = f]_i^{n+1} = 0$ 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 time pseudo time variable.

hpl 14: Better to say the last sentence when we treat iterative methods.

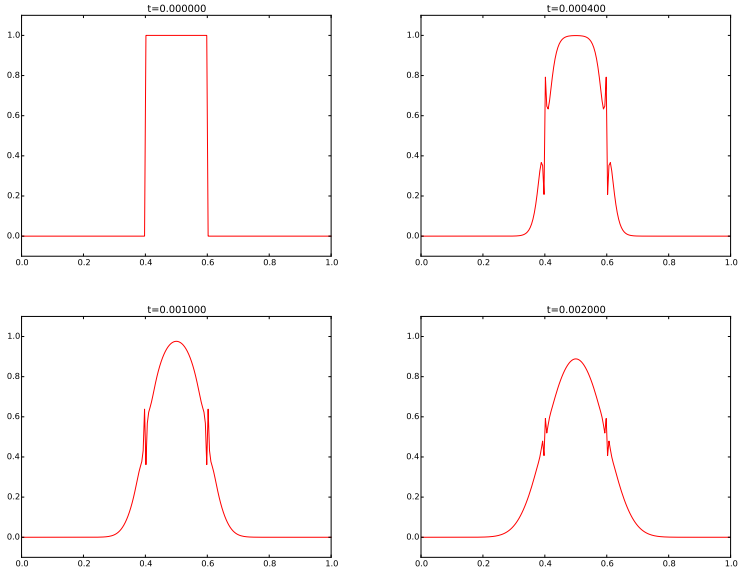


Fig. 3.7 Crank-Nicolson scheme for $F = 10$.

3.3 Analysis of schemes for the diffusion equation

The numerical experiments in Sections 3.2.5 and 3.1.5 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 3.1, 3.4, 3.2, 3.3, 3.5, 3.6, and 3.7.

3.3.1 Properties of the solution

A particular characteristic of diffusive processes, governed by an equation like

$$u_t = \alpha u_{xx}, \tag{3.42}$$

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.

Similarity solution. The diffusion equation (3.42) admits solutions that depend on $\eta = (x-c)/\sqrt{4\alpha t}$ for a given value of c. One particular solution is

$$u(x,t) = a\,\mathrm{erf}(\eta) + b, \tag{3.43}$$

where

$$\mathrm{erf}(\eta) = \frac{2}{\sqrt{\pi}}\int_0^\eta e^{-\zeta^2}d\zeta, \tag{3.44}$$

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

$$u(x,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right)\right) = \frac{1}{2}\mathrm{erfc}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right), \tag{3.45}$$

where we have introduced the *complementary error function* $\mathrm{erfc}(\eta) = 1 - \mathrm{erf}(\eta)$. The solution (3.45) implies the boundary conditions

$$u(0,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{-1/2}{\sqrt{4\alpha t}}\right)\right), \tag{3.46}$$

$$u(1,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{1/2}{\sqrt{4\alpha t}}\right)\right). \tag{3.47}$$

For small enough t, $u(0,t) \approx 1$ and $u(1,t) \approx 1$, but as $t \to \infty$, $u(x,t) \to 1/2$ on $[0,1]$.

Solution for a Gaussian pulse. The standard diffusion equation $u_t = \alpha u_{xx}$ admits a Gaussian function as solution:

$$u(x,t) = \frac{1}{\sqrt{4\pi\alpha t}}\exp\left(-\frac{(x-c)^2}{4\alpha t}\right). \tag{3.48}$$

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 (3.48) 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 (3.48) 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.

Solution for a sine component. For example, (3.42) admits a solution of the form

$$u(x,t) = Qe^{-at}\sin(kx)\,. \tag{3.49}$$

The parameters Q and k can be freely chosen, while inserting (3.49) in (3.42) 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.

The following examples illustrates the damping properties of (3.49). 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 (3.49) constructs an analytical solution to the problem:

$$
u(x,t) = e^{-\pi^2 t}\sin(\pi x) + 0.1e^{-\pi^2 10^4 t}\sin(100\pi x)\,. \tag{3.50}
$$

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.

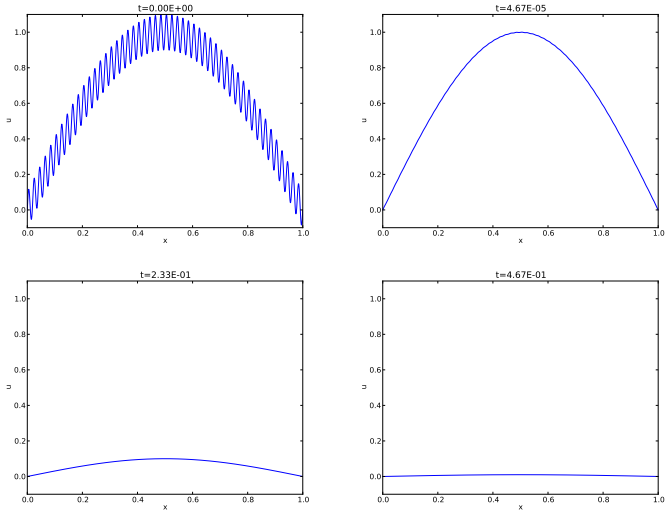


Fig. 3.8 Evolution of the solution of a diffusion problem: initial condition (upper left), 1/100 reduction of the small waves (upper right), 1/10 reduction of the long wave (lower left), and 1/100 reduction of the long wave (lower right).

3.3.2 Example: Diffusion of a discontinues profile

We shall see how different schemes predict the evolution of a discontinuous initial condition:

$$
u(x,0) = \begin{cases} U_L, & x < L/2 \\ U_R, & x \geq L/2 \end{cases}
$$

Such a discontinuous initial condition may arise when two insulated blocks of metals at different temperature are brought in contact at $t = 0$. Alternatively, signaling in the brain is based on release of a huge ion concentration on one side of a synapse, which implies diffusive transport of a discontinuous concentration function.

More to be written...

3.3.3 Analysis of discrete equations

A counterpart to (3.49) 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:

$$
u(x,t) \approx \sum_{k \in K} b_k e^{-\alpha k^2 t}e^{ikx}, \tag{3.51}
$$

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 need build a good approximate solution. Note that (3.50) is a special case of (3.51) 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

$$
I(x) \approx \sum_{k \in K} b_k e^{ikx}\,. \tag{3.52}
$$

(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 $A_{\mathrm{e}} = \exp(-\alpha k^2 \Delta t)$.

3.3.4 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

$$u_q^n = A^n e^{ikq\Delta x} = A^n e^{ikx}, \tag{3.53}$$

where the amplification factor A must be determined by inserting the component into an actual scheme.

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.

Accuracy. To determine how accurately a finite difference scheme treats one wave component (3.53), 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.

3.3.5 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 (3.53) 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

$$A = 1 - 4F \sin^2\left(\frac{k\Delta x}{2}\right), \tag{3.54}$$

where

$$F = \frac{\alpha \Delta t}{\Delta x^2} \tag{3.55}$$

is the *numerical Fourier number*. The complete numerical solution is then

$$u_q^n = \left(1 - 4F \sin^2\left(\frac{k\Delta x}{2}\right)\right)^n e^{ikq\Delta x}. \tag{3.56}$$

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

$$F \leq \frac{1}{2}, \tag{3.57}$$

or expressed as a condition on Δt:

$$\Delta t \leq \frac{\Delta x^2}{2\alpha}\,. \tag{3.58}$$

Note that halving the spatial mesh size, $\Delta x \rightarrow \frac{1}{2}\Delta x$, requires Δt to be reduced by a factor of 1/4. The method hence becomes very expensive for fine spatial meshes.

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 $F = \alpha\Delta t/\Delta x$, $p = k\Delta x/2$, and that $\sin^2 p \leq 1$, we realize that the dominating error terms are at most

$$1 - 4\alpha\frac{\Delta t}{\Delta x^2} + \alpha\Delta t - 4\alpha^2\Delta t^2 + \alpha^2\Delta t^2\Delta x^2 + \cdots\,.$$

3.3.6 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 (3.53), 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

$$A = \left(1 + 4F\sin^2 p\right)^{-1}\,. \tag{3.59}$$

The complete numerical solution can be written

$$u_q^n = \left(1 + 4F\sin^2 p\right)^{-n} e^{ikq\Delta x}\,. \tag{3.60}$$

Stability. We see from (3.59) that $0 < A < 1$, which means that all numerical wave components are stable and non-oscillatory for any $\Delta t > 0$.

3.3.7 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 (3.53) 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 (3.53) 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,

$$A = \frac{1-2F\sin^2 p}{1+2F\sin^2 p}. \tag{3.61}$$

The exact numerical solution is hence

$$u_q^n = \left(\frac{1-2F\sin^2 p}{1+2F\sin^2 p}\right)^n e^{ikp\Delta x}. \tag{3.62}$$

Stability. The criteria $A > -1$ and $A < 1$ are fulfilled for any $\Delta t > 0$.

3.3.8 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 3.9, 3.10, and 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 $F > 0.5$.

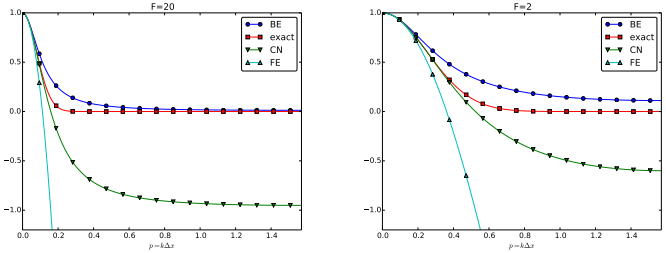


Fig. 3.9 Amplification factors for large time steps.

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 3.9 that for $F = 20$, waves with $p \geq \pi/2$ undergo a damping close to -1, which means that the amplitude does not decay and that the wave component jumps up and down 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.

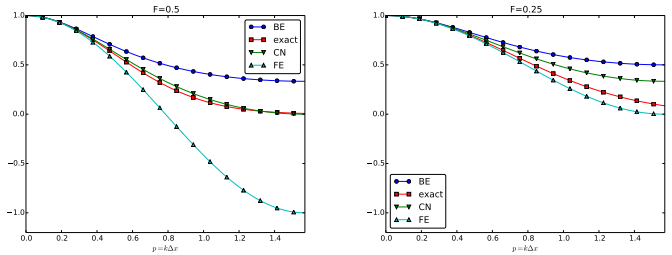


Fig. 3.10 Amplification factors for time steps around the Forward Euler stability limit.

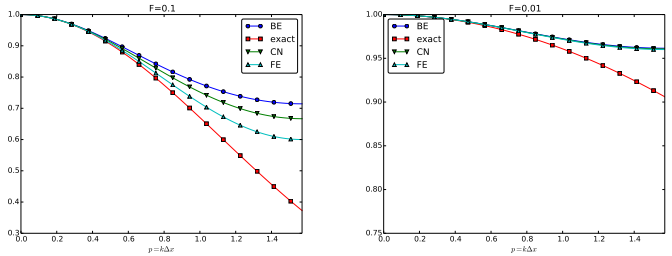


Fig. 3.11 Amplification factors for small time steps.

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.

Run $F =$ .....

Exercise 3.1: Explore symmetry in a 1D problem

This exercise simulates the exact solution (3.48). Suppose for simplicity that $c = 0$.

a) Formulate an initial-boundary value problem that has (3.48) as solution in the domain $[-L, L]$. Use the exact solution (3.48) 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 (3.48) 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).
Filename: `diffu_symmetric_gaussian`.

Exercise 3.2: Investigate approximation errors from a $u_x = 0$ boundary condition

We consider the problem solved in Exercise 3.1 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.

Hint. 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.
Filename: `diffu_onesided_fd`.

Exercise 3.3: Experiment with open boundary conditions in 1D

We address diffusion of a Gaussian function as in Exercise 3.1, 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

$$-\alpha u_x = q(u - u_S), \tag{3.63}$$

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 (3.63) as boundary condition. Assume one can take $u_S = 0$ "outside the domain" since $u_\mathrm{e} \to 0$ as $x \to \infty$. Find a function $q = q(t)$ such that the exact solution obeys the condition (3.63). Test some constant values of q and animate how

the corresponding error function behaves. Also compute $E(t)$ curves as defined above.
Filename: `diffu_open_BC`.

Exercise 3.4: Simulate a diffused Gaussian peak in 2D/3D

a) Generalize (3.48) 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 3.3. 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).
Filename: `diffu_symmetric_gaussian_2D`.

Exercise 3.5: 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 a Forward Euler scheme, a Backward Euler scheme, and a Crank-Nicolson for this type of diffusion model. Thereafter, formulate a θ-rule to summarize the three schemes.

b) Assume a solution like (3.49) and find the relation between a, k, α, and β.

Hint. Insert (3.49) 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 (3.49) 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?

Hint. For analysis of the accuracy, compare the numerical and exact amplification factors, in graphs and/or by Taylor series expansion.
Filename: `diffu_stability_uterm`.

3.4 Diffusion in heterogeneous media

Diffusion in heterogeneous media will normally imply a non-constant diffusion coefficient $\alpha = \alpha(x)$. A 1D diffusion model with such a variable diffusion coefficient reads

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(x)\frac{\partial u}{\partial x^2}\right), \quad x\in(0,L),\ t\in(0,T] \tag{3.64}$$

$$u(x,0) = I(x), \qquad x\in[0,L] \tag{3.65}$$

$$u(0,t) = U_L, \qquad t>0, \tag{3.66}$$

$$u(L,t) = U_0, \qquad t>0\,. \tag{3.67}$$

A short form of the diffusion equation with variable coefficients is $u_t = (\alpha u_x)_x$.

3.4.1 Stationary solution

As $t\to\infty$, the solution of the above problem will approach a stationary limit where $\partial u/\partial t = 0$. The governing equation is then

$$\frac{d}{dx}\left(\alpha\frac{du}{dx}\right) = 0, \tag{3.68}$$

with boundary conditions $u(0) = U_0$ and $u(L) = u_L$. It is possible to obtain an exact solution of (3.68) for any α. Integrating twice and applying the boundary conditions to determine the integration constants gives

$$u(x) = U_0 + (U_L - U_0)\frac{\int_0^x (\alpha(\xi))^{-1}d\xi}{\int_0^L (\alpha(\xi))^{-1}d\xi}. \tag{3.69}$$

3.4.2 Piecewise constant medium

Consider a medium built of M layers. The boundaries between the layers are denoted by $b_0, \ldots, b_M$, where $b_0 = 0$ and $b_M = L$. If the material in each layer potentially differs from the others, but is otherwise constant, we can express α as a *piecewise constant function* according to

$$\alpha(x) = \begin{cases} \alpha_0, \, b_0 \leq x < b_1, \\ \vdots \\ \alpha_i, \, b_i \leq x < b_{i+1}, \\ \vdots \\ \alpha_0, \, b_{M-1} \leq x \leq b_M. \end{cases} \tag{3.70}$$

The exact solution (3.69) 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:

$$u(x) = U_0 + (U_L - U_0)\frac{\sum_{j=0}^{m-1}(b_{j+1} - b_j)/\alpha(b_j) + (x - b_m)/\alpha(b_m)}{\sum_{j=0}^{M-1}(b_{j+1} - b_j)/\alpha(b_j)} \tag{3.71}$$

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 continues and therefore can be differentiated as $(\alpha u_x)_x$.

3.4.3 Implementation

Programming with piecewise function definition quickly becomes cumbersome as the most naive approach is to test for which interval x lies, and then start evaluating a formula like (3.71). In Python, vectorized expressions may help to speed up the computations. The convenience classes `PiecewiseConstant` and `IntegratedPiecewiseConstant` were

made to simplify programming with functions like (3.4.2) and expressions like (3.71). 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. This is advantageous in many computational contexts (although seldom for pure finite difference computations of the solution u).

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 function, 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 = 1/a is needed in formulas
        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)
        # Denominator in the exact formula is constant
        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 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
```

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 3.13 shows the result.

3.4.4 Diffusion equation in axi-symmetric geometries

Suppose we have a diffusion process taking care in a straight tube with radius R. We assume axi-symmetry such that u is just a function of r and t. A model problem is

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha(r)\frac{\partial u}{\partial r}\right) + f(t), \quad r \in (0,R),\ t \in (0,T], \tag{3.72}$$

$$\frac{\partial u}{\partial r}(0,t) = 0, \qquad t \in (0,T], \tag{3.73}$$

$$u(R,t) = 0, \qquad t \in (0,T], \tag{3.74}$$

$$u(r,0) = I(r), \qquad r \in [0,R]. \tag{3.75}$$

The condition (3.73) is a necessary symmetry condition at $r = 0$, while (3.74) could be any Dirichlet or Neumann condition (or Robin condition in case of cooling or heating).

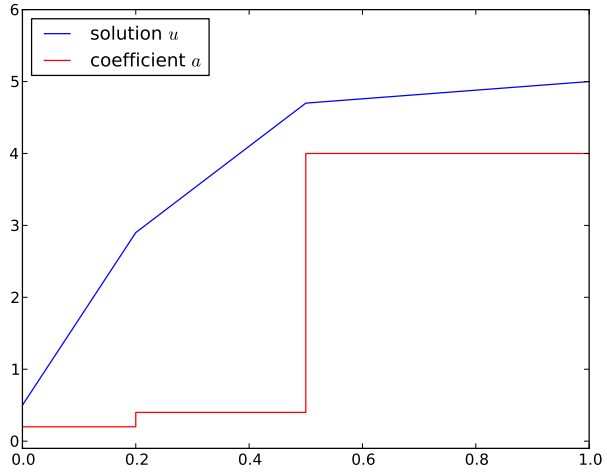


Fig. 3.12 Solution of the stationary diffusion equation corresponding to a piecewise constant diffusion coefficient.

The finite difference approximation at $r = 0$ of the spatial derivative term is the only new challenge in this problem. Let us in case of constant α expand the derivative to

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}.$$

The last term faces a difficulty at $r = 0$ since it becomes a $0/0$ expression because of the symmetry condition. L'Hosptial's rule can be used:

$$\lim_{r\to 0}\frac{1}{r}\frac{\partial u}{\partial r} = \lim_{r\to 0}\frac{\partial^2 u}{\partial r^2}.$$

The PDE at $r = 0$ therefore becomes

$$\frac{\partial u}{\partial t} = 2\alpha\frac{\partial^2 u}{\partial r^2} + f(t). \tag{3.76}$$

For a variable coefficient $\alpha(r)$ the expanded derivative reads

$$\alpha(r)\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}(\alpha(r) + r\alpha'(r))\frac{\partial u}{\partial r}.$$

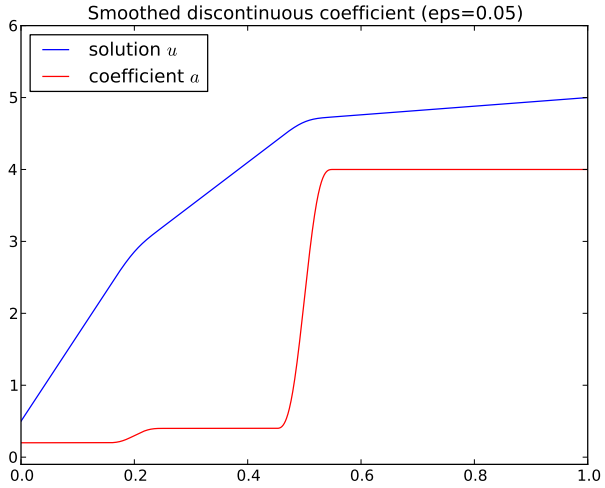


Fig. 3.13 Solution of the stationary diffusion equation corresponding to a *smoothed* piecewise constant diffusion coefficient.

We have that the limit of a product is

$$\lim_{r\to 0}\frac{1}{r}(\alpha(r)+r\alpha'(r))\frac{\partial u}{\partial r} = \lim_{r\to 0}(\alpha(r)+r\alpha'(r))\ \lim_{x\to c}\frac{1}{r}\frac{\partial u}{\partial r}\,.$$

The second limit becomes as above, so the PDE at $r=0$, assuming $(\alpha(0)+r\alpha'(0))\neq 0$, looks like

$$\frac{\partial u}{\partial t} = (2\alpha + r\alpha')\frac{\partial^2 u}{\partial r^2} + f(t)\,. \tag{3.77}$$

The second-order derivative is discretized in the usual way. Consider first constant α:

$$2\alpha\frac{\partial^2}{\partial r^2}u(r_0,t_n) \approx [2\alpha 2D_rD_ru]_0^n = 2\alpha\frac{u_1^n - 2u_0^n + u_{-1}^n}{\Delta r^2}\,.$$

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 second-order derivative of u in r at $r=0$:

$$4\alpha\frac{u_1^n - u_0^n}{\Delta r^2}\,. \tag{3.78}$$

With variable α we simply get

$$(2\alpha + r\alpha')2D_rD_ru]_0^n = (2\alpha(0)+r\alpha'(0))\frac{u_1^n - 2u_0^n + u_{-1}^n}{\Delta r^2}\,.$$

The discretization of the second-order derivative in r at another internal mesh point is straightforward:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha\frac{\partial u}{\partial r}\right)\Big|_{r=r_i}^{t=t_n} \approx [r^{-1}D_r(r\alpha D_ru)]_i^n = \frac{1}{\Delta r^2}\left(r_{i+\frac{1}{2}}\alpha_{i+\frac{1}{2}}(u_{i+1}^n - u_i^n) - r_{i-\frac{1}{2}}\alpha_{i-\frac{1}{2}}(u_i^n - u\right.$$

θ-rule in time...

3.4.5 Diffusion equation in spherically-symmetric geometries

Discretization in spherical coordinates. Let us now pose the problem from Section 3.4.4 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

$$\frac{\partial u}{\partial t} = \frac{\alpha}{r^\gamma}\frac{\partial}{\partial r}\left(r^\gamma\frac{\partial u}{\partial r}\right) + f(t), \tag{3.79}$$

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 (3.72)-(3.75).

Since the PDE in spherical coordinates has the same form as the PDE in Section 3.4.4, 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'Hosptial's rule shows that this term equals $\gamma\partial^2 u/\partial r^2$, and the PDE at $r=0$ becomes

$$\frac{\partial u}{\partial t} = (\gamma+1)\alpha\frac{\partial^2 u}{\partial r^2} + f(t)\,. \tag{3.80}$$

Same discretization, write up with γ.

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 the PDE yields

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(\alpha(r)r^2\frac{\partial u}{\partial t}\right),$$

and then

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + \alpha\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(\alpha\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one Cartesian coordinate.

3.5 Random walk

3.6 Exercises

Exercise 3.6: 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 3.3.7). Rannacher [3] 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 (3.45) with the boundary conditions (3.46)-(3.47) 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.

Project 3.7: 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:

$$u_t = \alpha u_x x, \qquad x\in\Omega = (0,L),\ t\in(0,T], \tag{3.81}$$

$$u(0,t) = u(L,t) = 0, \qquad t\in(0,T], \tag{3.82}$$

$$u(x,0) = I(x), \qquad x\in[0,L]\,. \tag{3.83}$$

The energy estimate for this problem reads

$$||u||_{L^2} \leq ||I||_{L^2}, \tag{3.84}$$

where the $||\cdot||_{L^2}$ norm is defined by

$$||g||_{L^2} = \sqrt{\int_0^L g^2 dx}\,. \tag{3.85}$$

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 (3.84) says that the "size of u" never exceeds that of the initial condition, or more equivalently, that the area under the u curve decreases with time.

To show (3.84), 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_xxu 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 (3.84).

b) Now we address a slightly different problem,

$$u_t = \alpha u_x x + f(x,t), \quad x \in \Omega = (0,L), \ t \in (0,T], \tag{3.86}$$
$$u(0,t) = u(L,t) = 0, \qquad t \in (0,T], \tag{3.87}$$
$$u(x,0) = 0, \qquad x \in [0,L]\,. \tag{3.88}$$

The associated energy estimate is

$$||u||_{L^2} \leq ||f||_{L^2}\,. \tag{3.89}$$

(This result is more difficult to derive.)

Now consider the compound problem with an initial condition $I(x)$ and a right-hand side $f(x,t)$:

$$u_t = \alpha u_x x + f(x,t), \quad x \in \Omega = (0,L), \ t \in (0,T], \tag{3.90}$$
$$u(0,t) = u(L,t) = 0, \qquad t \in (0,T], \tag{3.91}$$
$$u(x,0) = I(x), \qquad x \in [0,L]\,. \tag{3.92}$$

Show that if w_1 fulfills (3.81)-(3.83) and w_2 fulfills (3.86)-(3.88), then $u = w_1 + w_2$ is the solution of (3.90)-(3.92). Using the triangle inequality for norms,

$$||a+b|| \leq ||a|| + ||b||,$$

show that the energy estimate for (3.90)-(3.92) becomes

$$||u||_{L^2} \leq ||I||_{L^2} + ||f||_{L^2}\,. \tag{3.93}$$

c) One application of (3.93) is to prove uniqueness of the solution. Suppose u_1 and u_2 both fulfill (3.90)-(3.92). Show that $u = u_1 - u_2$ then fulfills (3.90)-(3.92) with $f = 0$ and $I = 0$. Use (3.93) 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 (3.93) to a 2D/3D diffusion equation $u_t = \nabla\cdot(\alpha\nabla u)$ for $x \in \Omega$.

Hint. 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

$$\int_\Omega u\,\mathrm{d}x = \int_\Omega I\,\mathrm{d}x\,. \tag{3.94}$$

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 (3.93) and (3.94) as a partial verification of the code. Choose some functions f and I and check that (3.93) is obeyed at any time when zero Dirichlet conditions are used. Iterate over the same I functions and check that (3.94) 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 (3.94) are for pointing out errors in the implementation.
Filename: `diffu_energy`.

Convection-diffusion equations 4

Staggered mesh discretization

5

5.0.1 The Euler-Cromer scheme on a standard mesh

Consider the fundamental model problem for simple harmonic oscillations,

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0, \tag{5.1}$$

where ω is the frequency of the oscillations (the exact solution is $u(t) = I\cos\omega t$). This model can equivalently be formulated as two first-order equations,

$$v' = -\omega^2 u, \tag{5.2}$$
$$u' = v\,. \tag{5.3}$$

The popular Euler-Cromer scheme for this 2×2 system of ODEs applies an explicit forward difference in (5.2) and a backward difference in (5.3):

$$\frac{v^{n+1} - v^n}{\Delta t} = -\omega^2 u^n, \tag{5.4}$$
$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1}\,. \tag{5.5}$$

For a time domain $[0, T]$, we have introduced a mesh with points $0 = t_0 < t_1 < \cdots < t_n = T$. The most common case is a mesh with uniform spacing Δt: $t_n = n\Delta t$. Then v^n is an approximation to $v(t)$ at mesh point t_n, and u^n is an approximation to $u(t)$ at the same point. Note that the

backward difference in (5.7) leads to an explicit updating formula for u^{n+1} since v^{n+1} is already computed:

$$v^{n+1} = v^n - \Delta t\omega^2 u^n, \tag{5.6}$$
$$u^{n+1} = u^n + \Delta t v^{n+1}\,. \tag{5.7}$$

The Euler-Cromer scheme is equivalent with the standard second-order accurate scheme for (5.1):

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n,\ n = 1, 2, \ldots, \tag{5.8}$$

but for the first time step, the method for (5.1) leads to

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 u^0, \tag{5.9}$$

while Euler-Cromer gives

$$u^1 = u^0 - \Delta t^2\omega^2 u^0, \tag{5.10}$$

which can be interpreted as a first-order, backward difference approximation of $u'(0) = 0$ combined with (5.8). At later time steps, however, the alternating use of forward and backward differences in (5.6)-(5.7) leads to a method with error $\mathcal{O}(\Delta t^2)$.

5.0.2 The Euler-Cromer scheme on a staggered mesh

hpl 15: Do the equations in different sequence, first vel, then pos.

The fact that the forward and backward differences used the Euler-Cromer method yield a second-order accurate method is not obvious from intuition. A much more intuitive discretization employs solely centered differences and leads to a scheme that is equivalent to the Euler-Cromer scheme. It is in fact fully equivalent to the second-order scheme for (5.1), also for the first time step. This alternative scheme is based on using a *staggered* (or alternating) mesh in time.

In a staggered mesh, the unknowns are sought at different points in the mesh. Specifically, u is sought at integer time points t_n and v is sought at $t_{n+1/2}$ *between* two u points. The unknowns are then $u^1, v^{3/2}, u^2, v^{5/2}$, and so on. We typically use the notation u^n and $v^{n+\frac{1}{2}}$ for the two unknown mesh functions.

On a staggered mesh it is natural to use centered difference approximations, expressed in operator notation as

$$[D_t u = v]^{n+\frac{1}{2}}, \tag{5.11}$$
$$[D_t v = -\omega u]^{n+1}\,. \tag{5.12}$$

Writing out the formulas gives

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}, \tag{5.13}$$
$$v^{n+\frac{3}{2}} = v^{n+\frac{1}{2}} - \Delta t\omega^2 u^{n+1}\,. \tag{5.14}$$

Of esthetic reasons one often writes these equations at the previous time level to replace the $\frac{3}{2}$ by $\frac{1}{2}$ in the left-most term in the last equation,

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \tag{5.15}$$
$$v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t\omega^2 u^n\,. \tag{5.16}$$

Such a rewrite is only mathematical cosmetics. The important thing is that this centered scheme has exactly the same computational structure as the forward-backward scheme. We shall use the names *forward-backward Euler-Cromer* and *staggered Euler-Cromer* to distinguish the two schemes.

We can eliminate the v values and get back the centered scheme based on the second-order differential equation, so all these three schemes are equivalent. However, they differ somewhat in the treatment of the initial conditions.

Suppose we have $u(0) = I$ and $u'(0) = v(0) = 0$ as mathematical initial conditions. This means $u^0 = I$ and

$$v(0) \approx \frac{1}{2}(v^{-\frac{1}{2}} + v^{\frac{1}{2}}) = 0, \quad \Rightarrow \quad v^{-\frac{1}{2}} = -v^{\frac{1}{2}}\,.$$

Using the discretized equation (5.16) for $n = 0$ yields

$$v^{\frac{1}{2}} = v^{-\frac{1}{2}} - \Delta t\omega^2 I,$$

and eliminating $v^{-\frac{1}{2}} = -v^{\frac{1}{2}}$ results in $v^{\frac{1}{2}} = -\frac{1}{2}\Delta t\omega^2 I$ and

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 I,$$

which is exactly the same equation for u^1 as we had in the centered scheme based on the second-order differential equation (and hence corresponds to a centered difference approximation of the initial condition for $u'(0)$). The conclusion is that a staggered mesh is fully equivalent with that scheme, while the forward-backward version gives a slight deviation in the computation of u^1.

We can redo the derivation of the initial conditions when $u'(0) = V$:

$$v(0) \approx \frac{1}{2}(v^{-\frac{1}{2}} + v^{\frac{1}{2}}) = V, \quad \Rightarrow \quad v^{-\frac{1}{2}} = 2V - v^{\frac{1}{2}}\,.$$

Using this $v^{-\frac{1}{2}}$ in

$$v^{\frac{1}{2}} = v^{-\frac{1}{2}} - \Delta t\omega^2 I,$$

then gives $v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I$. The general initial conditions are therefore

$$u^0 = I, \tag{5.17}$$
$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I\,. \tag{5.18}$$

5.0.3 Implementation of the scheme on a staggered mesh

The algorithm goes like this:

1. Set the initial values (5.17) and (5.18).
2. For $n = 1, 2, \ldots$:
   a. Compute u^n from (5.15).
   b. Compute $v^{n+\frac{1}{2}}$ from (5.16).

Implementation with integer indices. Translating the schemes (5.15) and (5.16) to computer code faces the problem of how to store and access $v^{n+\frac{1}{2}}$, since arrays only allow integer indices with base 0. We must then introduce a convention: $v^{1+\frac{1}{2}}$ is stored in `v[n]` while $v^{1-\frac{1}{2}}$ is stored in `v[n-1]`. We can then write the algorithm in Python as

```
def solver(I, w, dt, T):
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    v = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)  # mesh for u
```

```
    t_v = t + dt/2                # mesh for v

    u[0] = I
    v[0] = 0 - 0.5*dt*w**2*u[0]
    for n in range(1, Nt+1):
        u[n] = u[n-1] + dt*v[n-1]
        v[n] = v[n-1] - dt*w**2*u[n]
    return u, t, v, t_v
```

Note that the return u and v together with the mesh points such that the complete mesh function for u is described by `u` and `t`, while `v` and `t_v` represents the mesh function for v.

Implementation with half-integer indices. Some prefer to see a closer relationship between the code and the mathematics for the quantities with half-integer indices. For example, we would like to replace the updating equation for `v[n]` by

```
v[n+half] = v[n-half] - dt*w**2*u[n]
```

This is easy to do if we could be sure that `n+half` means `n` and `n-half` means `n-1`. A possible solution is to define `half` as a special object such that an integer plus `half` results in the integer, while an integer minus `half` equals the integer minus 1. A simple Python class may realize the `half` object:

```
class HalfInt:
    def __radd__(self, other):
        return other

    def __rsub__(self, other):
        return other - 1

half = HalfInt()
```

The `__radd__` function is invoked for all expressions `n+half` ("right add" with `self` as `half` and `other` as `n`). Similarly, the `__rsub__` function is invoked for `n-half` and results in `n-1`.

Using the `half` object, we can implement the algorithms in an even more readable way:

```
def solver(I, w, dt, T):
    """
    Solve u'=v, v' = - w**2*u for t in (0,T], u(0)=I and v(0)=0,
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    v = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)  # mesh for u
```



```
    t_v = t + dt/2                # mesh for v

    u[0] = I
    v[0+half] = 0 - 0.5*dt*w**2*u[0]
    for n in range(1, Nt+1):
        print n, n+half, n-half
        u[n] = u[n-1] + dt*v[n-half]
        v[n+half] = v[n-half] - dt*w**2*u[n]
    return u, t, v, t_v

def test_staggered():
    I = 1.2; w = 2.0; T = 5; dt = 2/w
    u, t, v, t_v = solver(I, w, dt, T)
    from vib_undamped import solver as solver2
    u2, t2 = solver2(I, w, dt, T)
    error = abs(u - u2).max()
    tol = 1E-14
    assert error < tol

if __name__ == '__main__':
    test_staggered()
```

Verification of this code is easy as we can just compare the computed `u` with the `u` produced by the `solver` function in `vib_undamped.py` (which solves $u'' + \omega^2 u = 0$ directly). The values should coincide to machine precision since the two numerical methods are mathematically equivalent. We refer to the file `vib_undamped_staggered.py` for the details of a nose test that checks this property.

5.0.4 A staggered Euler-Cromer scheme for a generalized model

The more general model for vibration problems,

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T], \quad (5.19)$$

can be rewritten as a first-order ODE system

$$u' = v, \quad (5.20)$$
$$v' = m^{-1}\left(F(t) - f(v) - s(u)\right). \quad (5.21)$$

It is natural to introduce a staggered mesh (see Section 5.0.2) and seek u at mesh points t_n (the numerical value is denoted by u^n) and v between mesh points at $t_{n+1/2}$ (the numerical value is denoted by $v^{n+\frac{1}{2}}$). A centered difference approximation to (5.20)-(5.21) can then be written in operator notation as

$$[D_t u = v]^{n-\frac{1}{2}}, \tag{5.22}$$
$$[D_t v = m^{-1}\left(F(t) - f(v) - s(u)\right)]^n\,. \tag{5.23}$$

Written out,

$$\frac{u^n - u^{n-1}}{\Delta t} = v^{n-\frac{1}{2}}, \tag{5.24}$$
$$\frac{v^{n+\frac{1}{2}} - v^{n-\frac{1}{2}}}{\Delta t} = m^{-1}\left(F^n - f(v^n) - s(u^n)\right)\,. \tag{5.25}$$

With linear damping, $f(v) = bv$, we can use an arithmetic mean for $f(v^n)$: $f(v^n) \approx= \frac{1}{2}(f(v^{n-\frac{1}{2}}) + f(v^{n+\frac{1}{2}}))$. The system (5.24)-(5.25) can then be solved with respect to the unknowns u^n and $v^{n+\frac{1}{2}}$:

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \tag{5.26}$$
$$v^{n+\frac{1}{2}} = \left(1 + \frac{b}{2m}\Delta t\right)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - \frac{1}{2}f(v^{n-\frac{1}{2}}) - s(u^n)\right)\right)\,. \tag{5.27}$$

In case of quadratic damping, $f(v) = b|v|v$, we can use a geometric mean: $f(v^n) \approx b|v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}}$. Inserting this approximation in (5.24)-(5.25) and solving for the unknowns u^n and $v^{n+\frac{1}{2}}$ results in

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \tag{5.28}$$
$$v^{n+\frac{1}{2}} = (1 + \frac{b}{m}|v^{n-\frac{1}{2}}|\Delta t)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - s(u^n)\right)\right)\,. \tag{5.29}$$

The initial conditions are derived at the end of Section 5.0.2:

$$u^0 = I, \tag{5.30}$$
$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I\,. \tag{5.31}$$

5.1 Exercises

Exercise 5.1: Use the forward-backward scheme with quadratic damping

We consider the generalized model with quadratic damping, expressed as a system of two first-order equations as in Section 5.0.4:

$$u' = v,$$
$$v' = \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right)\,.$$

However, contrary to what is done in Section 5.0.4, we want to apply the idea of a forward-backward discretization: u is marched forward by a one-sided Forward Euler scheme applied to the first equation, and thereafter v can marched forward by a Backward Euler scheme in the second equation, see in Section 1.7. Express the idea in operator notation and write out the scheme. Unfortunately, the backward difference for the v equation creates a nonlinearity $|v^{n+1}|v^n$. To linearize this nonlinearity, use the known value v^n inside the absolute value factor, i.e., $|v^{n+1}|v^n \approx |v^n|v^{n+1}$. Show that the resulting scheme is equivalent to the one in Section 5.0.4 for some time level $n \geq 1$.

What we learn from this exercise is that the first-order differences and the linearization trick play together in "the right way" such that the scheme is as good as when we (in Section 5.0.4) carefully apply centered differences and a geometric mean on a staggered mesh to achieve second-order accuracy. There is a difference in the handling of the initial conditions, though, as explained at the end of Section 1.7. Filename: `vib_gen_bwdamping`.

Useful formulas

A

A.1 Finite difference operator notation

$$u'(t_n) \approx [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \tag{A.1}$$

$$u'(t_n)a \approx [D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} + \tag{A.2}$$

$$u'(t_n) = [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \tag{A.3}$$

$$u'(t_n) \approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \tag{A.4}$$

$$u'(t_{n+\theta}) = [\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} \tag{A.5}$$

$$u'(t_n) \approx [D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} \tag{A.6}$$

$$u''(t_n) \approx [D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \tag{A.7}$$

$$u(t_{n+\frac{1}{2}}) \approx [\overline{u}^t]^{n+\frac{1}{2}} = \frac{1}{2}(u^{n+1} + u^n) \tag{A.8}$$

$$u(t_{n+\frac{1}{2}})^2 \approx [\overline{u^2}^{t,g}]^{n+\frac{1}{2}} = u^{n+1}u^n \tag{A.9}$$

$$u(t_{n+\frac{1}{2}}) \approx [\overline{u}^{t,h}]^{n+\frac{1}{2}} = \frac{2}{\frac{1}{u^{n+1}} + \frac{1}{u^n}} \tag{A.10}$$

$$u(t_{n+\theta}) \approx [\overline{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1-\theta)u^n, \quad t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_{n-1} \tag{A.11}$$

A.2 Truncation errors of finite difference approximations

$$u'_{\mathrm{e}}(t_n) = [D_t u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+\frac{1}{2}} - u_{\mathrm{e}}^{n-\frac{1}{2}}}{\Delta t} + R^n,$$
$$R^n = -\frac{1}{24}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.12}$$

$$u'_{\mathrm{e}}(t_n) = [D_{2t} u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n-1}}{2\Delta t} + R^n,$$
$$R^n = -\frac{1}{6}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.13}$$

$$u'_{\mathrm{e}}(t_n) = [D_t^- u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n} - u_{\mathrm{e}}^{n-1}}{\Delta t} + R^n,$$
$$R^n = -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{A.14}$$

$$u'_{\mathrm{e}}(t_n) = [D_t^+ u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n}}{\Delta t} + R^n,$$
$$R^n = -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{A.15}$$

$$u'_{\mathrm{e}}(t_{n+\theta}) = [\bar{D}_t u_{\mathrm{e}}]^{n+\theta} + R^{n+\theta} = \frac{u_{\mathrm{e}}^{n+1} - u_{\mathrm{e}}^{n}}{\Delta t} + R^{n+\theta},$$
$$R^{n+\theta} = -\frac{1}{2}(1-2\theta)u''_{\mathrm{e}}(t_{n+\theta})\Delta t + \frac{1}{6}((1-\theta)^3 - \theta^3)u'''_{\mathrm{e}}(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{A.16}$$

$$u'_{\mathrm{e}}(t_n) = [D_t^{2-} u_{\mathrm{e}}]^n + R^n = \frac{3u_{\mathrm{e}}^{n} - 4u_{\mathrm{e}}^{n-1} + u_{\mathrm{e}}^{n-2}}{2\Delta t} + R^n,$$
$$R^n = \frac{1}{3}u'''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{A.17}$$

$$u''_{\mathrm{e}}(t_n) = [D_t D_t u_{\mathrm{e}}]^n + R^n = \frac{u_{\mathrm{e}}^{n+1} - 2u_{\mathrm{e}}^{n} + u_{\mathrm{e}}^{n-1}}{\Delta t^2} + R^n,$$
$$R^n = -\frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.18}$$

$$u_{\mathrm{e}}(t_{n+\theta}) = [\overline{u_{\mathrm{e}}}^{t,\theta}]^{n+\theta} + R^{n+\theta} = \theta u_{\mathrm{e}}^{n+1} + (1-\theta)u_{\mathrm{e}}^{n} + R^{n+\theta},$$
$$R^{n+\theta} = -\frac{1}{2}u''_{\mathrm{e}}(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3)\,. \tag{A.19}$$

A.3 Finite differences of exponential functions

Complex exponentials. Let $u^n = \exp(i\omega n\Delta t) = e^{i\omega t}$.

$$[D_t D_t u]^n = u^n \frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right), \tag{A.20}$$
$$[D_t^+ u]^n = u^n \frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1), \tag{A.21}$$
$$[D_t^- u]^n = u^n \frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)), \tag{A.22}$$
$$[D_t u]^n = u^n \frac{2}{\Delta t} i\sin\left(\frac{\omega\Delta t}{2}\right), \tag{A.23}$$
$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i\sin(\omega\Delta t)\,. \tag{A.24}$$

Real exponentials. Let $u^n = \exp(\omega n\Delta t) = e^{\omega t}$.

$$[D_t D_t u]^n = u^n \frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right), \tag{A.25}$$
$$[D_t^+ u]^n = u^n \frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1), \tag{A.26}$$
$$[D_t^- u]^n = u^n \frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)), \tag{A.27}$$
$$[D_t u]^n = u^n \frac{2}{\Delta t} i\sin\left(\frac{\omega\Delta t}{2}\right), \tag{A.28}$$
$$[D_{2t} u]^n = u^n \frac{1}{\Delta t} i\sin(\omega\Delta t)\,. \tag{A.29}$$

A.4 Finite differences of t^n

The following results are useful when checking if a polynomial term in a solution fulfills the discrete equation for the numerical method.

$$[D_t^+ t]^n = 1, \tag{A.30}$$
$$[D_t^- t]^n = 1, \tag{A.31}$$
$$[D_t t]^n = 1, \tag{A.32}$$
$$[D_{2t} t]^n = 1, \tag{A.33}$$
$$[D_t D_t t]^n = 0\,. \tag{A.34}$$

The next formulas concern the action of difference operators on a t^2 term.

$$[D_t^+ t^2]^n = (2n+1)\Delta t, \tag{A.35}$$
$$[D_t^- t^2]^n = (2n-1)\Delta t, \tag{A.36}$$
$$[D_t t^2]^n = 2n\Delta t, \tag{A.37}$$
$$[D_{2t} t^2]^n = 2n\Delta t, \tag{A.38}$$
$$[D_t D_t t^2]^n = 2, \tag{A.39}$$

Finally, we present formulas for a t^3 term: **These must be controlled** against `lib.py`. Use t_n instead of $n\Delta t$??

$$[D_t^+ t^3]^n = 3(n\Delta t)^2 + 3n\Delta t^2 + \Delta t^2, \tag{A.40}$$
$$[D_t^- t^3]^n = 3(n\Delta t)^2 - 3n\Delta t^2 + \Delta t^2, \tag{A.41}$$
$$[D_t t^3]^n = 3(n\Delta t)^2 + \frac{1}{4}\Delta t^2, \tag{A.42}$$
$$[D_{2t} t^3]^n = 3(n\Delta t)^2 + \Delta t^2, \tag{A.43}$$
$$[D_t D_t t^3]^n = 6n\Delta t, \tag{A.44}$$

Truncation error analysis

B

Summary

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.

B.1 Overview of truncation error analysis

B.1.1 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.

B.1.2 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 is to ask 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 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.

B.2 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.

B.2.1 Example: The backward difference for $u'(t)$

Consider a backward finite difference approximation of the first-order derivative u':

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx u'(t_n)\,. \tag{B.1}$$

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 not exactly equal to the derivative $u'(t_n)$. The error in the approximation is

$$R^n = [D_t^- u]^n - u'(t_n)\,. \tag{B.2}$$

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 (B.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,

$$u(t_{n-1}) = u(t-\Delta t) = \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i u}{dt^i}(t_n)(-\Delta t)^i$$
$$= u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3),$$

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 left-hand side of1 (B.2) gives rise to some algebra:

$$[D_t^- u]^n - u'(t_n) = \frac{u(t_n) - u(t_{n-1})}{\Delta t} - u'(t_n)$$
$$= \frac{u(t_n) - (u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3))}{\Delta t} - u'(t_n)$$
$$= -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2)),$$

which is, according to (B.2), the truncation error:

$$R^n = -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2))\,. \tag{B.3}$$

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.

B.2.2 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.

B.2.3 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 $u(t_{n-1/2})$ in Taylor series around the point t_n where the derivative is evaluated. We have

$$\begin{aligned}
u(t_{n+\frac{1}{2}}) =& u(t_n) + u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)(\frac{1}{2}\Delta t)^2 + \\
& \frac{1}{6}u'''(t_n)(\frac{1}{2}\Delta t)^3 + \frac{1}{24}u''''(t_n)(\frac{1}{2}\Delta t)^4 + \\
& \frac{1}{120}u'''''(t_n)(\frac{1}{2}\Delta t)^5 + \mathcal{O}(\Delta t^6), \\
u(t_{n-1/2}) =& u(t_n) - u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)(\frac{1}{2}\Delta t)^2 - \\
& \frac{1}{6}u'''(t_n)(\frac{1}{2}\Delta t)^3 + \frac{1}{24}u''''(t_n)(\frac{1}{2}\Delta t)^4 - \\
& \frac{1}{120}u'''''(t_n)(\frac{1}{2}\Delta t)^5 + \mathcal{O}(\Delta t^6)\,.
\end{aligned}$$

Now,

$$u(t_{n+\frac{1}{2}}) - u(t_{n-1/2}) = u'(t_n)\Delta t + \frac{1}{24}u'''(t_n)\Delta t^3 + \frac{1}{960}u'''''(t_n)\Delta t^5 + \mathcal{O}(\Delta t^7)\,.$$

By collecting terms in $[D_t u]^n - u(t_n)$ we find the truncation error to be

$$R^n = \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4), \tag{B.4}$$

with only even powers of Δt. Since $R \sim \Delta t^2$ we say the centered difference is of *second order* in Δt.

B.2.4 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.

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} = u'(t_n) + R^n, \tag{B.5}$$

$$R^n = \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.6}$$

$$[D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} = u'(t_n) + R^n, \tag{B.7}$$

$$R^n = \frac{1}{6}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.8}$$

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} = u'(t_n) + R^n, \tag{B.9}$$

$$R^n = -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{B.10}$$

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} = u'(t_n) + R^n, \tag{B.11}$$

$$R^n = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{B.12}$$

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} = u'(t_{n+\theta}) + R^{n+\theta}, \tag{B.13}$$

$$R^{n+\theta} = \frac{1}{2}(1-2\theta)u''(t_{n+\theta})\Delta t - \frac{1}{6}((1-\theta)^3 - \theta^3)u'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{B.14}$$

$$[D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} = u'(t_n) + R^n, \tag{B.15}$$

$$R^n = -\frac{1}{3}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{B.16}$$

$$[D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = u''(t_n) + R^n, \tag{B.17}$$

$$R^n = \frac{1}{12}u''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.18}$$

It will also be convenient to have the truncation errors for various means or averages. The weighted arithmetic mean leads to

$$[\overline{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1-\theta)u^n = u(t_{n+\theta}) + R^{n+\theta}, \tag{B.19}$$

$$R^{n+\theta} = \frac{1}{2}u''(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3)\,. \tag{B.20}$$

The standard arithmetic mean follows from this formula when $\theta = 1/2$. Expressed at point t_n we get

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) = u(t_n) + R^n, \tag{B.21}$$

$$R^n = \frac{1}{8}u''(t_n)\Delta t^2 + \frac{1}{384}u''''(t_n)\Delta t^4 + \mathcal{O}(\Delta t^6)\,. \tag{B.22}$$

The geometric mean also has an error $\mathcal{O}(\Delta t^2)$:

$$[\overline{u^2}^{t,g}]^n = u^{n-\frac{1}{2}}u^{n+\frac{1}{2}} = (u^n)^2 + R^n, \tag{B.23}$$

$$R^n = -\frac{1}{4}u'(t_n)^2\Delta t^2 + \frac{1}{4}u(t_n)u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,. \tag{B.24}$$

The harmonic mean is also second-order accurate:

$$[\overline{u}^{t,h}]^n = u^n = \frac{2}{\frac{1}{u^{n-\frac{1}{2}}} + \frac{1}{u^{n+\frac{1}{2}}}} + R^{n+\frac{1}{2}}, \tag{B.25}$$

$$R^n = -\frac{u'(t_n)^2}{4u(t_n)}\Delta t^2 + \frac{1}{8}u''(t_n)\Delta t^2\,. \tag{B.26}$$

B.2.5 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
        # Introduce symbols for the derivatives
        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.

B.3 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 the linear ODE modeling exponential decay,

$$u'(t) = -au(t)\,. \tag{B.27}$$

B.3.1 Truncation error of the Forward Euler scheme

We begin with the Forward Euler scheme for discretizing (B.27):

$$[D_t^+ u = -au]^n\,. \tag{B.28}$$

The idea behind the truncation error computation is to insert the exact solution u_{e} of the differential equation problem (B.27) in the discrete equations (B.28) 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:

$$[D_t^+ u_{\mathrm{e}} + au_{\mathrm{e}} = R]^n\,. \tag{B.29}$$

From (B.11)-(B.12) 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 (B.29) 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:

$$R^n = \frac{1}{2}u_{\mathrm{e}}''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,. \tag{B.30}$$

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.

B.3.2 Truncation error of the Crank-Nicolson scheme

For the Crank-Nicolson scheme,

$$[D_t u = -au]^{n+\frac{1}{2}}, \tag{B.31}$$

we compute the truncation error by inserting the exact solution of the ODE and adding a residual R,

$$[D_t u_{\mathrm{e}} + a\overline{u_{\mathrm{e}}}^t = R]^{n+\frac{1}{2}}\,. \tag{B.32}$$

The term $[D_t u_{\mathrm{e}}]^{n+\frac{1}{2}}$ is easily computed from (B.5)-(B.6) by replacing n with $n+\frac{1}{2}$ in the formula,

$$[D_t u_{\mathrm{e}}]^{n+\frac{1}{2}} = u'(t_{n+\frac{1}{2}}) + \frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

The arithmetic mean is related to $u(t_{n+\frac{1}{2}})$ by (B.21)-(B.22) so

$$[a\overline{u_{\mathrm{e}}}^t]^{n+\frac{1}{2}} = u(t_{n+\frac{1}{2}}) + \frac{1}{8}u''(t_n)\Delta t^2 + +\mathcal{O}(\Delta t^4)\,.$$

Inserting these expressions in (B.32) 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

$$R^{n+\frac{1}{2}} = \left(\frac{1}{24}u_{\mathrm{e}}'''(t_{n+\frac{1}{2}}) + \frac{1}{8}u''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.33}$$

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 B.7.4.

B.3.3 Truncation error of 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 (B.13)-(B.14) and (B.19)-(B.20) 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

$$\begin{aligned} R^{n+\theta} =& (\frac{1}{2} - \theta)u_{\mathrm{e}}''(t_{n+\theta})\Delta t + \frac{1}{2}\theta(1-\theta)u_{\mathrm{e}}''(t_{n+\theta})\Delta t^2 + \\ & \frac{1}{2}(\theta^2 - \theta + 3)u_{\mathrm{e}}'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned} \tag{B.34}$$

For $\theta = 1/2$ the first-order term vanishes and the scheme is of second order, while for $\theta \neq 1/2$ we only have a first-order scheme.

B.3.4 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

    # Define schemes
    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))
    # Residuals (truncation errors)
    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.

B.3.5 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

$$R^n = [D_t^+ u_{\mathrm{e}} + a u_{\mathrm{e}}]^n \,. \tag{B.35}$$

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

$$r_{i-1} = \frac{\ln(R_{i-1}/R_i)}{\ln(\Delta t_{i-1}/\Delta t_i)}, \tag{B.36}$$

for $i = 1, \ldots, m-1$. Note that the truncation error R_i varies through the mesh, so (B.36) 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 (B.36), where n counts mesh points in the coarsest mesh, $n = 0, \ldots, N_0$, leads to the formula

$$r_{i-1}^n = \frac{\ln(R_{i-1}^n/R_i^n)}{\ln(\Delta t_{i-1}/\Delta t_i)} \,. \tag{B.37}$$

Experiments are most conveniently defined by N_0 and a number of refinements m. Suppose each mesh have 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,

$$R_{I,i} = \left(\Delta t_i \sum_{n=0}^{N_i} (R_i^n)^2\right)^{\frac{1}{2}} \approx \int_0^T R_i(t)dt \,. \tag{B.38}$$

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
import scitools.std as plt

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], log='y')
            legends_R.append('N=%d' % N[i])
            plt.hold('on')

            plt.figure(2)
            plt.plot(t_coarse, R_a[i] - R[i], log='y')
            plt.hold('on')
            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')

    # Convergence rates
    r_R_I = convergence_rates(dt, R_I)
    print 'R integrated in time; r:',
    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 (B.30) 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 (B.37), 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 (B.35) 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]

    # Theoretical expression for the trunction error
    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 (B.30) and the computed quantity (ref(B.35)) is very good, as illustrated in Figures B.1 and B.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 B.7.5).

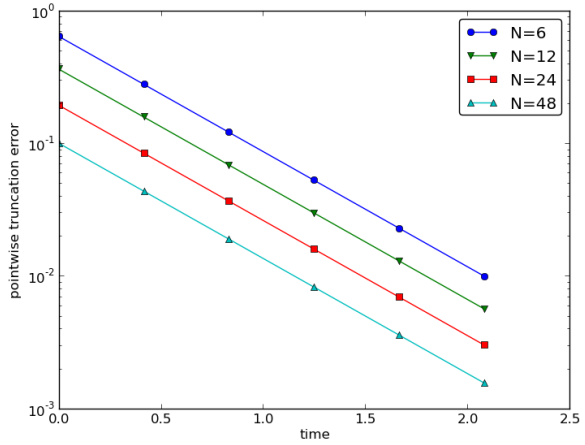


Fig. B.1 Estimated truncation error at mesh points for different meshes.

B.3.6 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:

$$[D_t^+ u_\mathrm{e} + a u_\mathrm{e} = C + R]^n \,. \tag{B.39}$$

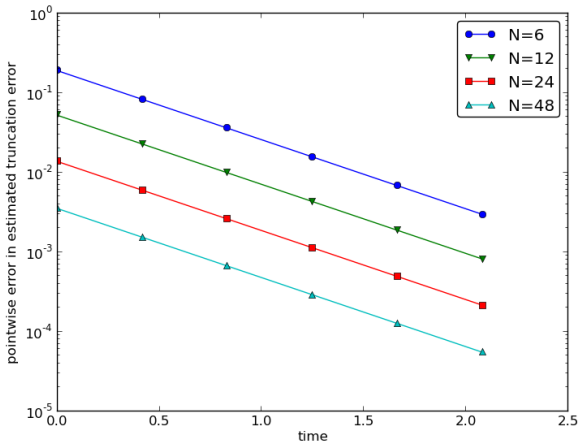


Fig. B.2 Difference between theoretical and estimated truncation error at mesh points for different meshes.

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

$$[D_t^+ u = -au + \frac{1}{2}a^2\Delta t u]^n, \tag{B.40}$$

or we can equivalently discretize the perturbed ODE

$$u' = -\hat{a}u, \quad \hat{a} = a(1 - \frac{1}{2}a\Delta t), \tag{B.41}$$

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 (B.41) 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 $a^2 u$, 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 now recommended to do Exercise B.7.6 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 (B.5)-(B.6) and (B.21)-(B.22), 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 $u'' = a^2 u$, and we find that $u''' = -a^3 u$. 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 B.7.7 encourages you to implement these correction terms and calculate empirical convergence rates to verify that higher-order accuracy is indeed obtained in real computations.

B.3.7 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,

$$[D_t^+ u = -au + b]^n \,. \tag{B.42}$$

The truncation error R is as always found by inserting the exact solution $u_{\mathrm{e}}(t)$ in the discrete scheme:

$$[D_t^+ u_{\mathrm{e}} + au_{\mathrm{e}} - b = R]^n \,. \tag{B.43}$$

Using (B.11)-(B.12),

$$u'_{\mathrm{e}}(t_n) - \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) + a(t_n)u_{\mathrm{e}}(t_n) - b(t_n) = R^n \,.$$

Because of the ODE,

$$u'_{\mathrm{e}}(t_n) + a(t_n)u_{\mathrm{e}}(t_n) - b(t_n) = 0,$$

so we are left with the result

$$R^n = -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \mathcal{O}(\Delta t^2) \,. \tag{B.44}$$

We see that the variable coefficients do not pose any additional difficulties in this case. Exercise B.7.8 takes the analysis above one step further to the Crank-Nicolson scheme.

B.3.8 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 (B.5)-(B.6) and (B.25)-(B.26) in Section B.2.4, we see that all but two of the R expressions contains 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 B.2.4 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}.

B.3.9 Computing truncation errors in nonlinear problems

The general nonlinear ODE

$$u' = f(u, t), \tag{B.45}$$

can be solved by a Crank-Nicolson scheme

$$[D_t u = \overline{f}^t]^{n+\frac{1}{2}} \,. \tag{B.46}$$

The truncation error is as always defined as the residual arising when inserting the exact solution u_{e} in the scheme:

$$[D_t u_{\mathrm{e}} - \overline{f}^t = R]^{n+\frac{1}{2}} \,. \tag{B.47}$$

Using (B.21)-(B.22) 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 (B.5)-(B.6) the discrete equations (B.47) lead to

$$u'_{\mathrm{e}}(t_{n+\frac{1}{2}})+\frac{1}{24}u'''_{\mathrm{e}}(t_{n+\frac{1}{2}})\Delta t^2-f(u_{\mathrm{e}}^{n+\frac{1}{2}},t_{n+\frac{1}{2}})-\frac{1}{8}u''(t_{n+\frac{1}{2}})\Delta t^2+\mathcal{O}(\Delta t^4)=R^{n+\frac{1}{2}}\,.$$

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

$$R^{n+\frac{1}{2}}=(\frac{1}{24}u'''_{\mathrm{e}}(t_{n+\frac{1}{2}})-\frac{1}{8}u''(t_{n+\frac{1}{2}}))\Delta t^2\,.$$

The computational techniques worked well even for this nonlinear ODE.

B.4 Truncation errors in vibration ODEs

B.4.1 Linear model without damping

The next example on computing the truncation error involves the following ODE for vibration problems:

$$u''(t)+\omega^2u(t)=0\,. \tag{B.48}$$

Here, ω is a given constant.

The truncation error of a centered finite difference scheme. Using a standard, second-ordered, central difference for the second-order derivative time, we have the scheme

$$[D_tD_tu+\omega^2u=0]^n\,. \tag{B.49}$$

Inserting the exact solution u_{e} in this equation and adding a residual R so that u_{e} can fulfill the equation results in

$$[D_tD_tu_{\mathrm{e}}+\omega^2u_{\mathrm{e}}=R]^n\,. \tag{B.50}$$

To calculate the truncation error R^n, we use (B.17)-(B.18), i.e.,

$$[D_tD_tu_{\mathrm{e}}]^n=u''_{\mathrm{e}}(t_n)+\frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2,$$

and the fact that $u''_{\mathrm{e}}(t)+\omega^2u_{\mathrm{e}}(t)=0$. The result is

$$R^n=\frac{1}{12}u''''_{\mathrm{e}}(t_n)\Delta t^2+\mathcal{O}(\Delta t^4)\,. \tag{B.51}$$

The truncation error of approximating $u'(0)$. The initial conditions for (B.48) 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 (B.7)-(B.8). 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 B.7.11 asks you to quickly perform an experiment to investigate this question.

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 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 $u^{-1}=u^1-2\Delta tV$ 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 $[D_tD_t+\omega^2u=0]^0$ to eliminate u^{-1} in the disretized initial condition. From $[D_tD_t+\omega^2u=0]^0$ we have

$$u^{-1}=2u^0-u^1-\Delta t^2\omega^2u^0,$$

which inserted in $[D_{2t}u=V]^0$ gives

$$\frac{u^1-u^0}{\Delta t}+\frac{1}{2}\omega^2\Delta tu^0=V\,. \tag{B.52}$$

The first term can be recognized as a forward difference such that the equation can be written in operator notation as

$$[D_t^+u+\frac{1}{2}\omega^2\Delta tu=V]^0\,.$$

The truncation error is defined as

$$[D_t^+u_{\mathrm{e}}+\frac{1}{2}\omega^2\Delta tu_{\mathrm{e}}-V=R]^0\,.$$

Using (B.11)-(B.12) 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

$$[D_{2t}u + \Delta t(D_tD_tu - \omega^2 u) = V]^0\,. \tag{B.53}$$

Writing out (B.53) shows that the equation is equivalent to (B.52). The truncation error is defined by

$$[D_{2t}u_{\mathrm{e}} + \Delta t(D_tD_tu_{\mathrm{e}} - \omega^2 u_{\mathrm{e}}) = V + R]^0\,.$$

Replacing the difference via (B.7)-(B.8) and (B.17)-(B.18), as well as using $u'_{\mathrm{e}}(0) = V$ and $u''_{\mathrm{e}}(0) = -\omega^2 u_{\mathrm{e}}(0)$, gives

$$R^n = \frac{1}{6}u'''(0)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.$$

Computing correction terms. The idea of using correction terms to increase the order of R^n can be applied as described in Section B.3.6. We look at

$$[D_tD_tu_{\mathrm{e}} + \omega^2 u_{\mathrm{e}} = C + R]^n,$$

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

$$u'' + \omega^2(1 - \frac{1}{12}\omega^2\Delta t^2)u = 0\,. \tag{B.54}$$

Solving this equation by the standard scheme

$$[D_tD_tu + \omega^2(1 - \frac{1}{12}\omega^2\Delta t^2)u = 0]^n,$$

will result in a scheme with truncation error $\mathcal{O}(\Delta t^4)$.

We can use another set of arguments to justify that (B.54) leads to a higher-order method. Mathematical analysis of the scheme (B.49) 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:

$$[u'' + (\omega(1 - \frac{1}{24}\omega^2\Delta t^2))^2 u = 0\,.$$

Expanding the squared term and omitting the higher-order term Δt^4 gives exactly the ODE (B.54). Experiments show that u^n is computed to 4th order in Δt.

B.4.2 Model with damping and nonlinearity

The model (B.48) can be extended to include damping $\beta u'$, a nonlinear restoring (spring) force $s(u)$, and some known excitation force $F(t)$:

$$mu'' + \beta u' + s(u) = F(t)\,. \tag{B.55}$$

The coefficient m usually represents the mass of the system. This governing equation can by discretized by centered differences:

$$[mD_tD_tu + \beta D_{2t}u + s(u) = F]^n\,. \tag{B.56}$$

The exact solution u_{e} fulfills the discrete equations with a residual term:

$$[mD_tD_tu_{\mathrm{e}} + \beta D_{2t}u_{\mathrm{e}} + s(u_{\mathrm{e}}) = F + R]^n\,. \tag{B.57}$$

Using (B.17)-(B.18) and (B.7)-(B.8) 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

$$R^n = \left(\frac{m}{12}u''''_{\mathrm{e}}(t_n) + \frac{\beta}{6}u'''_{\mathrm{e}}(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4), \tag{B.58}$$

so the scheme is of second order.

According to (B.58), 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.

B.4.3 Extension to quadratic damping

Instead of the linear damping term $\beta u'$ in (B.55) we now consider quadratic damping $\beta|u'|u'$:

$$mu'' + \beta|u'|u' + s(u) = F(t)\,. \tag{B.59}$$

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

$$[mD_tD_tu]^n + \beta|[D_tu]^{n-\frac{1}{2}}|[D_tu]^{n+\frac{1}{2}} + s(u^n) = F^n\,. \tag{B.60}$$

The truncation error is defined through

$$[mD_tD_tu_{\mathrm{e}}]^n + \beta|[D_tu_{\mathrm{e}}]^{n-\frac{1}{2}}|[D_tu_{\mathrm{e}}]^{n+\frac{1}{2}} + s(u^n_{\mathrm{e}}) - F^n = R^n\,. \tag{B.61}$$

We start with expressing the truncation error of the geometric mean. According to (B.23)-(B.24),

$$|[D_tu_{\mathrm{e}}]^{n-\frac{1}{2}}|[D_tu_{\mathrm{e}}]^{n+\frac{1}{2}} = [|D_tu_{\mathrm{e}}|D_tu_{\mathrm{e}}]^n - \frac{1}{4}u'(t_n)^2\Delta t^2 + \frac{1}{4}u(t_n)u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Using (B.5)-(B.6) 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 of $\mathcal{O}(\Delta t^2)$.

B.4.4 The general model formulated as first-order ODEs

The second-order model (B.59) can be formulated as a first-order system,

$$u' = v, \tag{B.62}$$

$$v' = \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right)\,. \tag{B.63}$$

The system (B.62)-(B.62) can be solved either by a forward-backward scheme or a centered scheme on a staggered mesh.

The forward-backward scheme. The discretization is based on the idea of stepping (B.62) forward in time and then using a backward difference in (B.63) with the recently computed (and therefore known) u:

$$[D_t^+ u = v]^n, \tag{B.64}$$

$$[D_t^- v = \frac{1}{m}(F(t) - \beta|v|v - s(u))]^{n+1}\,. \tag{B.65}$$

The term $|v|v$ gives rise to a nonlinearity $|v^{n+1}|v^{n+1}$, which can be linearized as $|v^n|v^{n+1}$:

$$[D_t^+ u = v]^n, \tag{B.66}$$

$$[D_t^- v]^{n+1} = \frac{1}{m}(F(t_{n+1}) - \beta|v^n|v^{n+1} - s(u^{n+1}))\,. \tag{B.67}$$

Each ODE will have a truncation error when inserting the exact solutions u_e and v_e in (B.64)-(B.65):

$$[D_t^+ u_\mathrm{e} = v_\mathrm{e} + R_u]^n, \tag{B.68}$$

$$[D_t^- v_\mathrm{e}]^{n+1} = \frac{1}{m}(F(t_{n+1}) - \beta|v_\mathrm{e}(t_n)|v_\mathrm{e}(t_{n+1}) - s(u_\mathrm{e}(t_{n+1}))) + R_v^{n+1}\,. \tag{B.69}$$

Application of (B.11)-(B.12) and (B.9)-(B.10) in (B.68) and (B.69), respectively, gives

$$u_\mathrm{e}'(t_n) + \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t + \mathcal{O}(\Delta t^2) = v_\mathrm{e}(t_n) + R_u^n, \tag{B.70}$$

$$v_\mathrm{e}'(t_{n+1}) - \frac{1}{2}v_\mathrm{e}''(t_{n+1})\Delta t + \mathcal{O}(\Delta t^2) = \frac{1}{m}(F(t_{n+1}) - \beta|v_\mathrm{e}(t_n)|v_\mathrm{e}(t_{n+1}) + s(u_\mathrm{e}(t_{n+1})) + R_v^n\,. \tag{B.71}$$

Since $u_\mathrm{e}' = v_\mathrm{e}$, (B.70) gives

$$R_u^n = \frac{1}{2}u_\mathrm{e}''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,.$$

In (B.71) 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.

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 (B.62)-(B.62):

$$[D_t u = v]^{n-\frac{1}{2}}, \tag{B.72}$$

$$[D_t v = \frac{1}{m}(F(t) - \beta|v|v - s(u))]^n\,. \tag{B.73}$$

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

$$[D_t u]^{n-\frac{1}{2}} = v^{n-\frac{1}{2}}, \tag{B.74}$$

$$[D_t v]^n = \frac{1}{m}(F(t_n) - \beta|v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}} - s(u^n))\,. \tag{B.75}$$

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 (B.5)-(B.6), and the geometric mean approximation analysis can be taken from (B.23)-(B.24). 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:

$$R_u^{n-\frac{1}{2}} = \mathcal{O}(\Delta t^2), \quad R_v^n = \mathcal{O}(\Delta t^2)\,.$$

Comparing (B.74)-(B.75) with (B.66)-(B.67), we can hopefully realize that these schemes are equivalent (which becomes clear when we implement both). The obvious advantage with the staggered mesh approach is that we can all the way use second-order accurate building blocks and in this way convince ourselves that the resulting scheme has an error of $\mathcal{O}(\Delta t^2)$.

B.5 Truncation errors in wave equations

B.5.1 Linear wave equation in 1D

The standard, linear wave equation in 1D for a function $u(x,t)$ reads

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{B.76}$$

where c is the constant wave velocity of the physical medium $[0,L]$. The equation can also be more compactly written as

$$u_{tt} = c^2 u_{xx} + f, \quad x \in (0,L),\ t \in (0,T], \tag{B.77}$$

Centered, second-order finite differences are a natural choice for discretizing the derivatives, leading to

$$[D_t D_t u = c^2 D_x D_x u + f]_i^n\,. \tag{B.78}$$

Inserting the exact solution $u_{\mathrm{e}}(x,t)$ in (B.78) makes this function fulfill the equation if we add the term R:

$$[D_t D_t u_{\mathrm{e}} = c^2 D_x D_x u_{\mathrm{e}} + f + R]_i^n \tag{B.79}$$

Our purpose is to calculate the truncation error R. From (B.17)-(B.18) we have that

$$[D_t D_t u_{\mathrm{e}}]_i^n = u_{\mathrm{e},tt}(x_i,t_n) + \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

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:

$$[D_x D_x u_{\mathrm{e}}]_i^n = u_{\mathrm{e},xx}(x_i,t_n) + \frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4),$$

Equation (B.79) now becomes

$$u_{\mathrm{e},tt} + \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 = c^2 u_{\mathrm{e},xx} + c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + f(x_i,t_n) + \mathcal{O}(\Delta t^4,\Delta x^4) + R_i^n\,.$$

Because u_{e} fulfills the partial differential equation (PDE) (B.77), the first, third, and fifth terms cancel out, and we are left with

$$R_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta t^4,\Delta x^4), \tag{B.80}$$

showing that the scheme (B.78) is of second order in the time and space mesh spacing.

B.5.2 Finding correction terms

Can we add correction terms to the PDE and increase the order of R_i^n in (B.80)? The starting point is

$$[D_tD_tu_\mathrm{e} = c^2D_xD_xu_\mathrm{e} + f + C + R]_i^n \tag{B.81}$$

From the previous analysis we simply get (B.80) again, but now with C:

$$R_i^n + C_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta t^4,\Delta x^4)\,. \tag{B.82}$$

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

$$C = \frac{1}{12}\left(u_{tttt}\Delta t^2 - c^2u_{xxxx}\Delta x^2\right),$$

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 that

$$\frac{\partial^2}{\partial t^2} = c^2\frac{\partial^2}{\partial x^2},$$

so

$$u_{tttt} = c^2u_{xxtt},\quad u_{xxxx} = c^{-2}u_{ttxx}\,.$$

Assuming u is smooth enough 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

$$\frac{1}{\Delta t^2}\Bigg(\frac{u_{i+1}^{n+1} - 2u_{i+1}^n + u_{i+1}^{n-1}}{\Delta x^2} - 2\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta x^2} + \frac{u_{i-1}^{n+1} - 2u_{i-1}^n + u_{i-1}^{n-1}}{\Delta x^2}\Bigg)$$

Now the unknown values u_{i+1}^{n+1}, u_i^{n+1}, 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 schemes, while we had a convenient explicit scheme without the correction terms.

B.5.3 Extension to variable coefficients

Now we address the variable coefficient version of the linear 1D wave equation,

$$\frac{\partial^2u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right),$$

or written more compactly as

$$u_{tt} = (\lambda u_x)_x\,. \tag{B.83}$$

The discrete counterpart to this equation, using arithmetic mean for λ and centered differences, reads

$$[D_tD_tu = D_x\overline{\lambda}^xD_xu]_i^n\,. \tag{B.84}$$

The truncation error is the residual R in the equation

$$[D_tD_tu_\mathrm{e} = D_x\overline{\lambda}^xD_xu_\mathrm{e} + R]_i^n\,. \tag{B.85}$$

The difficulty in the present is how to compute the truncation error of the term $[D_x\overline{\lambda}^xD_xu_\mathrm{e}]_i^n$.

We start by writing out the outer operator:

$$[D_x\overline{\lambda}^xD_xu_\mathrm{e}]_i^n = \frac{1}{\Delta x}\left([\overline{\lambda}^xD_xu_\mathrm{e}]_{i+\frac{1}{2}}^n - [\overline{\lambda}^xD_xu_\mathrm{e}]_{i-\frac{1}{2}}^n\right). \tag{B.86}$$

With the aid of (B.5)-(B.6) and (B.21)-(B.22) we have

$$
\begin{aligned}
[D_x u_{\mathrm{e}}]^n_{i+\frac{1}{2}} &= u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \frac{1}{24} u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4),\\
[\overline{\lambda}^x]_{i+\frac{1}{2}} &= \lambda(x_{i+\frac{1}{2}}) + \frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4),\\
[\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_{i+\frac{1}{2}} &= (\lambda(x_{i+\frac{1}{2}}) + \frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4))\times\\
&\quad (u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \frac{1}{24} u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4))\\
&= \lambda(x_{i+\frac{1}{2}}) u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n) + \lambda(x_{i+\frac{1}{2}})\frac{1}{24} u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\Delta x^2 +\\
&\quad u_{\mathrm{e},x}(x_{i+\frac{1}{2}})\frac{1}{8}\lambda''(x_{i+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4)\\
&= [\lambda u_{\mathrm{e},x}]^n_{i+\frac{1}{2}} + G^n_{i+\frac{1}{2}}\Delta x^2 + \mathcal{O}(\Delta x^4),
\end{aligned}
$$

where we have introduced the short form

$$
G^n_{i+\frac{1}{2}} = (\frac{1}{24} u_{\mathrm{e},xxx}(x_{i+\frac{1}{2}}, t_n)\lambda((x_{i+\frac{1}{2}}) + u_{\mathrm{e},x}(x_{i+\frac{1}{2}}, t_n)\frac{1}{8}\lambda''(x_{i+\frac{1}{2}}))\Delta x^2\,.
$$

Similarly, we find that

$$
[\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_{i-\frac{1}{2}} = [\lambda u_{\mathrm{e},x}]^n_{i-\frac{1}{2}} + G^n_{i-\frac{1}{2}}\Delta x^2 + \mathcal{O}(\Delta x^4)\,.
$$

Inserting these expressions in the outer operator (B.86) results in

$$
\begin{aligned}
[D_x\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_i &= \frac{1}{\Delta x}([\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_{i+\frac{1}{2}} - [\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_{i-\frac{1}{2}})\\
&= \frac{1}{\Delta x}([\lambda u_{\mathrm{e},x}]^n_{i+\frac{1}{2}} + G^n_{i+\frac{1}{2}}\Delta x^2 - [\lambda u_{\mathrm{e},x}]^n_{i-\frac{1}{2}} - G^n_{i-\frac{1}{2}}\Delta x^2 + \mathcal{O}(\Delta x^4))\\
&= [D_x\lambda u_{\mathrm{e},x}]^n_i + [D_x G]^n_i\Delta x^2 + \mathcal{O}(\Delta x^4)\,.
\end{aligned}
$$

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 $[D_x H]^n_i\Delta x^4$.

We can now use (B.5)-(B.6) to express the D_x operator in $[D_x\lambda u_{\mathrm{e},x}]^n_i$ as a derivative and a truncation error:

$$
[D_x\lambda u_{\mathrm{e},x}]^n_i = \frac{\partial}{\partial x}\lambda(x_i)u_{\mathrm{e},x}(x_i, t_n) + \frac{1}{24}(\lambda u_{\mathrm{e},x})_{xxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.
$$

Expressions like $[D_x G]^n_i\Delta x^2$ can be treated in an identical way,

$$
[D_x G]^n_i\Delta x^2 = G_x(x_i, t_n)\Delta x^2 + \frac{1}{24} G_{xxx}(x_i, t_n)\Delta x^4 + \mathcal{O}(\Delta x^4)\,.
$$

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

$$
[D_x\overline{\lambda}^x D_x u_{\mathrm{e}}]^n_i = \frac{\partial}{\partial x}\lambda(x_i)u_{\mathrm{e},x}(x_i, t_n) + \mathcal{O}(\Delta x^2)\,.
$$

After having treated the $[D_t D_t u_{\mathrm{e}}]^n_i$ term as well, we achieve

$$
R^n_i = \mathcal{O}(\Delta x^2) + \frac{1}{12} u_{\mathrm{e},tttt}(x_i, t_n)\Delta t^2\,.
$$

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.

B.5.4 1D wave equation on a staggered mesh

B.5.5 Linear wave equation in 2D/3D

The two-dimensional extension of (B.76) takes the form

$$
\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x, y, t), \quad (x, y) \in (0, L)\times(0, H),\ t \in (0, T], \tag{B.87}
$$

where now $c(x, y)$ is the constant wave velocity of the physical medium $[0, L]\times[0, H]$. In the compact notation, the PDE (B.87) can be written

$$
u_{tt} = c^2(u_{xx} + u_{yy}) + f(x, y, t), \quad (x, y) \in (0, L)\times(0, H),\ t \in (0, T], \tag{B.88}
$$

in 2D, while the 3D version reads

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f(x, y, z, t), \tag{B.89}$$

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 (B.17)-(B.18) yields the scheme

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]^n_{i,j,k}\,. \tag{B.90}$$

The truncation error is found from

$$[D_t D_t u_{\mathrm{e}} = c^2(D_x D_x u_{\mathrm{e}} + D_y D_y u_{\mathrm{e}}) + f + R]^n\,. \tag{B.91}$$

The calculations from the 1D case can be repeated to the terms in the y and z directions. Collecting terms that fulfill the PDE, we end up with

$$R^n_{i,j,k} = [\frac{1}{12}u_{\mathrm{e},tttt}\Delta t^2 - c^2\frac{1}{12}\left(u_{\mathrm{e},xxxx}\Delta x^2 + u_{\mathrm{e},yyyy}\Delta x^2 + u_{\mathrm{e},zzzz}\Delta z^2\right)]^n_{i,j,k} + \mathcal{O}(\Delta t^4, \Delta x^4, \Delta y^4, \Delta z^4)\,. \tag{B.92}$$

B.6 Truncation errors in diffusion equations

B.6.1 Linear diffusion equation in 1D

The standard, linear, 1D diffusion equation takes the form

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2} + f(x, t), \quad x \in (0, L),\ t \in (0, T], \tag{B.93}$$

where $\alpha > 0$ is the 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, $\alpha u_x x$, is commonly discretized as $[D_x D_x u]^n_i$. The time-derivative, however, can be treated by a variety of methods.

The Forward Euler scheme in time. Let us start with the simple Forward Euler scheme:

$$[D_t^+ u = \alpha D_x D_x u + f]^n\,.$$

The truncation error arises as the residual R when inserting the exact solution u_{e} in the discrete equations:

$$[D_t^+ u_{\mathrm{e}} = \alpha D_x D_x u_{\mathrm{e}} + f + R]^n_i\,.$$

Now, using (B.11)-(B.12) and (B.17)-(B.18), we can transform the difference operators to derivatives:

$$u_{\mathrm{e},t}(x_i, t_n) + \frac{1}{2}u_{\mathrm{e},tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) = \alpha u_{\mathrm{e},xx}(x_i, t_n) + \frac{\alpha}{12}u_{\mathrm{e},xxxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4) + f(x_i, t_n) + R^n_i\,.$$

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

$$R^n_i = \frac{1}{2}u_{\mathrm{e},tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) - \frac{\alpha}{12}u_{\mathrm{e},xxxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

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 $u_x x$ term:

$$[D_t u]^{n+\frac{1}{2}}_i = \alpha\frac{1}{2}([D_x D_x u]^n_i + [D_x D_x u]^{n+1}_i + f^{n+\frac{1}{2}}_i\,.$$

The equation for the truncation error is

$$[D_t u_{\mathrm{e}}]^{n+\frac{1}{2}}_i = \alpha\frac{1}{2}([D_x D_x u_{\mathrm{e}}]^n_i + [D_x D_x u_{\mathrm{e}}]^{n+1}_i) + f^{n+\frac{1}{2}}_i + R^{n+\frac{1}{2}}_i\,.$$

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 (B.21)-(B.22),

$$\frac{1}{2}([D_x D_x u_{\mathrm{e}}]^n_i + [D_x D_x u_{\mathrm{e}}]^{n+1}_i) = [D_x D_x u_{\mathrm{e}}]^{n+\frac{1}{2}}_i + \frac{1}{8}[D_x D_x u_{\mathrm{e},tt}]^{n+\frac{1}{2}}_i\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

With (B.17)-(B.18) we can express the difference operator $D_x D_x u$ in terms of a derivative:

$$[D_x D_x u_{\mathrm{e}}]^{n+\frac{1}{2}}_i = u_{\mathrm{e},xx}(x_i, t_{n+\frac{1}{2}}) + \frac{1}{12}u_{\mathrm{e},xxxx}(x_i, t_{n+\frac{1}{2}})\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

The error term from the arithmetic mean is similarly expanded,

$$\frac{1}{8}[D_xD_xu_{\mathrm{e},tt}]_i^{n+\frac{1}{2}}\Delta t^2 = \frac{1}{8}u_{\mathrm{e},ttxx}(x_i,t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^2\Delta x^2)$$

The time derivative is analyzed using (B.5)-(B.6):

$$[D_tu]_i^{n+\frac{1}{2}} = u_{\mathrm{e},t}(x_i,t_{n+\frac{1}{2}}) + \frac{1}{24}u_{\mathrm{e},ttt}(x_i,t_{n+\frac{1}{2}})\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

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

$$R_i^{n+\frac{1}{2}} = \frac{1}{8}u_{\mathrm{e},xx}(x_i,t_{n+\frac{1}{2}})\Delta t^2 + \frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_{n+\frac{1}{2}})\Delta x^2 + \frac{1}{24}u_{\mathrm{e},ttt}(x_i,t_{n+\frac{1}{2}})\Delta t^2 + +\mathcal{O}(\Delta x^4) + \mathcal{O}(\Delta t^4) + \mathcal{O}(\Delta t^2\Delta x^2)$$

B.6.2 Linear diffusion equation in 2D/3D

B.6.3 A nonlinear diffusion equation in 2D

B.7 Exercises

B.7.1 Exercise B.1: Truncation error of a weighted mean

Derive the truncation error of the weighted mean in (B.19)-(B.20).

Hint. Expand u_{e}^{n+1} and u_{e}^n around $t_{n+\theta}$.
Filename: `trunc_weighted_mean`.

B.7.2 Exercise B.2: 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 $u_{\mathrm{e}}(t)$ 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 (B.19)-(B.20). Filename: `trunc_theta_avg`.

B.7.3 Exercise B.3: Verify a truncation error formula

Set up a numerical experiment as explained in Section B.3.5 for verifying the formulas (B.15)-(B.16). Filename: `trunc_backward_2level`.

B.7.4 Exercise B.4: 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 $u' = -a(t)u + b(t)$. Filename: `trunc_decay_BE`.

B.7.5 Exercise B.5: Empirical estimation of truncation errors

Use the ideas and tools from Section B.3.5 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]`.
Filename: `trunc_decay_BNCN`.

B.7.6 Exercise B.6: Correction term for a Backward Euler scheme

Consider the model $u' = -au$, $u(0) = I$. Use the ideas of Section B.3.6 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. Filename: `trunc_decay_BE_corr`.

B.7.7 Exercise B.7: Verify the effect of correction terms

The program `decay_convrate.py` solves $u' = -au$, $u(0) = I$, by the θ-rule and computes convergence rates. Copy this file and adjust a in the `solver` function 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 $\mathcal{O}(\Delta t^4)$. Filename: `trunc_decay_corr_verify`.

B.7.8 Exercise B.8: 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 $t_{n+\frac{1}{2}}$:

$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{B.94}$$

$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}}\,. \tag{B.95}$$

Compute the truncation error in both cases. Filename: `trunc_decay_CN_vc`.

B.7.9 Exercise B.9: 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 $f(u,t) = -au$. Filename: `trunc_nonlinear_ODE`.

B.7.10 Exercise B.10: Truncation error of $[D_t D_t u]^n$

Derive the truncation error of the finite difference approximation (B.17)-(B.18) to the second-order derivative. Filename: `trunc_d2u`.

B.7.11 Exercise B.11: Investigate the impact of approximating $u'(0)$

Section B.4.1 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. Filename: `trunc_vib_ic_fw`.

B.7.12 Exercise B.12: 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

$$[mD_t D_t u + \beta|D_t^- u|D_t^- u + s(u) = F]^n\,.$$

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? Filename: `trunc_vib_bw_damping`.

Software engineering; wave equation model

C

C.1 A 1D wave equation simulator

C.1.1 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

$$u_{tt} = (q(x)u_x)_x + f(x,t), \quad x \in (0,L),\ t \in (0,T] \tag{C.1}$$
$$u(x,0) = I(x), \quad x \in [0,L] \tag{C.2}$$
$$u_t(x,0) = V(t), \quad x \in [0,L] \tag{C.3}$$
$$u(0,t) = U_0(t) \text{ or } u_x(0,t) = 0, \quad t \in (0,T] \tag{C.4}$$
$$u(L,t) = U_L(t) \text{ or } u_x(L,t) = 0, \quad t \in (0,T] \tag{C.5}$$

We allow variable wave velocity $c^2(x) = q(x)$, and Dirichlet or homogeneous Neumann conditions at the boundaries.

C.1.2 Numerical discretization

The PDE is discretized by second-order finite differences in time and space, with arithmetic mean for the variable coefficient

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]_i^n . \tag{C.6}$$

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 Sections 2.6 and 2.7.

C.1.3 A solver function

The general initial-boundary value problem (C.1)-(C.5) solved by finite difference methods can be implemented in the following `solver` function (taken from the file `wave1D_dn_vc.py`). This function builds on simpler versions described in Sections 2.3, 2.4 2.6, and 2.7. There are several quite advanced constructs that will be commented upon later.

```
def solver(I, V, f, c, U_0, U_L, L, dt, C, T,
           user_action=None, version='scalar',
           stability_safety_factor=1.0):
    """Solve u_tt=(c^2*u_x)_x + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = np.linspace(0, Nt*dt, Nt+1)      # Mesh points in time

    # Find max(c) using a fake mesh and adapt dx to C and dt
    if isinstance(c, (float,int)):
        c_max = c
    elif callable(c):
        c_max = max([c(x_) for x_ in linspace(0, L, 101)])
    dx = dt*c_max/(stability_safety_factor*C)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)          # Mesh points in space

    # Treat c(x) as array
    if isinstance(c, (float,int)):
        c = np.zeros(x.shape) + c
    elif callable(c):
        # Call c(x) and fill array 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

    # Wrap user-given f, I, V, U_0, U_L if None or 0
    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

    # Make hash of all input data
    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()
    if os.path.isfile('.' + hashed_input + '_archive.npz'):
        # Simulation is already run
        return -1, hashed_input

    u   = np.zeros(Nx+1)   # Solution array at new time level
    u_1 = np.zeros(Nx+1)   # Solution at 1 time level back
    u_2 = np.zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # CPU time measurement

    Ix = range(0, Nx+1)
    It = range(0, Nt+1)

    # Load initial condition into u_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)

    # Special formula for the first step
    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:
        # Set boundary values (x=0: i-1 -> i+1 since u[i-1]=u[i+1]
        # when du/dn = 0, on x=L: i+1 -> i-1 since u[i+1]=u[i-1])
        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)

    # Update data structures for next step
    #u_2[:] = u_1;  u_1[:] = u  # safe, but slower
    u_2, u_1, u = u_1, u, u_2

    for n in It[1:-1]:
        # Update all inner points
        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)

        # Insert boundary conditions
        i = Ix[0]
        if U_0 is None:
            # Set boundary values
            # x=0: i-1 -> i+1 since u[i-1]=u[i+1] when du/dn=0
            # x=L: i+1 -> i-1 since u[i+1]=u[i-1] when du/dn=0
            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

        # Update data structures for next step
        #u_2[:] = u_1;  u_1[:] = u  # safe, but slower
        u_2, u_1, u = u_1, u, u_2
```

```
    # Important to correct the mathematically wrong u=u_2 above
    # before returning u
    u = u_1
    cpu_time = t0 - time.clock()
    return cpu_time, hashed_input
```

Or maybe copy section by section...?

C.2 Saving large arrays 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`.

C.2.1 Using `savez` to store arrays in files

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.

Merging zip archives. Each separate call to `np.savez` creates a new file (zip archive) with extension `.npz`. It is very convenient if 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)

    # Open each archive and write to the common archive
    for filename in filenames:
        f = zipfile.ZipFile(filename,  'r',
                            zipfile.ZIP_DEFLATED)
        for name in f.namelist():
            data = f.open(name, 'r')
            # Save under name without .npy
            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.

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:
    # array_names[array_name] is the array itself
    # e.g. plot(array_names['t'], array_names[array_name])
```

C.2.2 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 from its name. The name of the directory under which the arrays are stored by `joblib` can be given by the user.

```
class Storage(object):
    """
    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 `retrive` 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 example of a memoize function, see Section 2.1.4in [2]).

C.2.3 Using a hash to create a file or directory name

The user of array storage techniques like those outlined in Sections C.2.2 and C.2.1 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):
    # Convert arguments to hash
    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` to turn 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 of x <= x0 else 1
```

The source code will be `f = lambda x: 0 of 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 to hash to test if a zip archive or `joblib` subdirectory exists and if so avoids recomputation).

C.3 Software for the 1D wave equation

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:
            # Save all the time points where solutions are saved
            savez('.' + self.filename + '_t.dat',
                  t=array(self.t, dtype=float))
            # Merge all savez files to one zip archive
            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 C.9.1.

C.3.1 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`:

```
# Make hash of all input data
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()
```

NOTE: All this is now explained!

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!

C.3.2 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)
```

C.3.3 Verification

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 an exact solution of the scheme, while with Neumann conditions are left with testing just a constant solution (see comments in Section 2.6.5).

A more general method for verification is to check the convergence rates.

Do convergence rates here! It is general...

C.4 Programming the solver with classes

Many who knows about class programming prefer to organize their software in terms of classes. We can easily port our function-based code in ... to a class version.

We will create a class `Problem` to hold the physical parameters of the problem and a class `Solver` to hold the numerical parameters and the solver function. 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).

C.4.1 Class Problem

C.4.2 Class Mesh

The `Mesh` class can be made valid for a space-time mesh in any number of space dimensions. To make 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(object):
    """
    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:
            # No spatial mesh
            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')

        # Allow 1D interface without nested lists with one element
```



```
        if L is not None and isinstance(L[0], (float,int)):
            # Only an interval was given
            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]

        # Set all attributes to None
        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
```

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.

C.4.3 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(object):
    """
    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 = []

        # Create array(s) to store mesh point values
        if (self.mesh.has_space() and not self.mesh.has_time()) or \
           (self.mesh.has_space() and self.mesh.has_time() and \
            space_only):
            # Space mesh 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():
            # Time mesh only
            if num_comp == 1:
                self.u = np.zeros(self.mesh.Nt+1)
                self.indices = ['time']
            else:
                # Need num_comp entries per time step
                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:
            # Space-time mesh
            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)
```


C.4.4 Class Solver

With the `Mesh` and `Function` classes in place, we can rewrite the `solver` function, but we put it as a method in class `Solver`:

hpl 16: Rewrite solver!

C.5 Migrating loops to Cython

We now consider the `wave2D_u0.py` code for solving the 2D linear wave equation with constant wave velocity and homogeneous Dirichlet boundary conditions $u = 0$. This code contains a `solver` function, which calls and `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 2.12, in particular Sections 2.12.1 and 2.12.2.

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`.

C.5.1 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_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = 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 Iy[1:-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] = D1*u_1[i,j] - D2*u_2[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])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: 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 numpy as np
cimport cython
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 Iy_start = 0
        int Ix_end = u.shape[0]-1
        int Iy_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(Iy_start+1, Iy_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]
    # Boundary condition u=0
    j = Iy_start
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
```



```
    j = Iy_end
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    i = Ix_start
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    i = Ix_end
    for j in range(Iy_start, Iy_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'`).
3. Functions declared with `cpdef` are translated to C but are also accessible from Python.
4. 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.
5. 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.
6. 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.
7. The use of index sets `Ix` and `Iy` 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.

C.5.2 Visual inspection of the C translation

Cython can visually explain how successfully it translated a code from Python to C. The command

Terminal

```
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 C.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.

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).

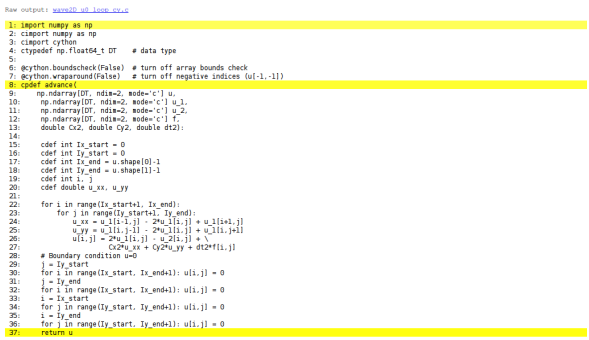


Fig. C.1 Visual illustration of Cython's ability to translate Python to C.

C.5.3 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

```
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.py` code we do not use `pyximport` and require an explicit build process of this and many other modules.

C.5.4 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_1, u_2, f_a, x, y, t, Cx2, Cy2, dt2)
```

Efficiency. For a mesh consisting of 120×120 cells, the scalar Python code require 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.

C.6 Migrating loops to Fortran

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++). Or more precisely, we can with ease program with 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 requires us to reduce multi-dimensional indices to a single index.

C.6.1 The Fortran subroutine

We write a Fortran subroutine `advance` in a file `wave2D_u0_loop_f77.f` for implementing the updating formula (2.117) 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.

Remark

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_1, u_2, 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_1, u_2, f, Cx2, Cy2, dt2)
```

C.6.2 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

```
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_1,u_2,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_1,u_2,f,cx2,cy2,dt2,
              nx=(shape(u,0)-1),ny=(shape(u,1)-1))
.
advance - Function signature:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,[nx,ny])
Required arguments:
  u : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_1 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_2 : 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.

C.6.3 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_1 = zeros((Nx+1,Ny+1), order=order)   # solution at t-dt
u_2 = 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

```
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.

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.

C.7 Migrating loops to C via Cython

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_1`, and `u_2` 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.

C.7.1 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,


```
#define idx(i,j) (i)*(Ny+1) + j
```

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 parenthesis 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 requires careful use. In C++, inline functions are safer and replace the need for macros.

C.7.2 The complete C code

The C version of our function `advance` can be coded as follows.

```
#define idx(i,j) (i)*(Ny+1) + j

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;
}
```

C.7.3 The Cython interface file

All the code above appears in a 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`. Here this file, called `wave2D_u0_loop_c_cy.pyx`, looks like

```
import numpy as np
cimport numpy as np
cimport cython

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_1, double* u_2, 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.

C.7.4 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

```
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_1, u_2, f_a, Cx2, Cy2, dt2)
```

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 we do not work with very small meshes and consequently little numerical work in the `advance` function.

C.8 Migrating loops to C via f2py

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

```
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

```
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`.

C.8.1 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.

C.9 Exercises

C.9.1 Exercise C.1: 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 C.2 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')
```

Hint. Study the source code for function `savez` (or more precisely, function `_savez`).
Filename: `Savez`.

References

[1] H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models*. Springer, 2015. http://tinyurl.com/nclmcng/web.

[2] H. P. Langtangen. *Scaling of Differential Equations*. 2015. http://tinyurl.com/qfjgxmf/web.

[3] R. Rannacher. Finite element solution of diffusion problems with irregular data. *Numerische Mathematik*, 43:309–327, 1984.

[4] L. N. Trefethen. *Trefethen's index cards - Forty years of notes about People, Words and Mathematics*. World Scientific, 2011.

Index