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 undeformed state coincide with the interval $[0, L]$ on the x axis, and let $u(x,t)$ be the displacement at time t in the y direction of a point initially at x. The displacement function u is governed by the mathematical model

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

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

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

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

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

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*, given here as (2.4) and (2.5). These two conditions specify that the string is fixed at the ends, i.e., that the displacement u is zero.

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

H.P. Langtangen, S. Linge, *Finite Difference Computing with PDEs*,
Texts in Computational Science and Engineering 16, DOI 10.1007/978-3-319-55456-3_2

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 characteristic 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 Fig. 2.1 illustrate such neighboring mesh points that contribute to an algebraic equation. In this particular case, we have sampled the PDE at the point $(2, 2)$ and constructed (2.11), which then involves a coupling of u_1^2, u_2^3, u_2^2, u_2^1, and u_3^2. The term *stencil* is often used about the algebraic equation at a mesh point, and the geometry of a typical stencil is illustrated in Fig. 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

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

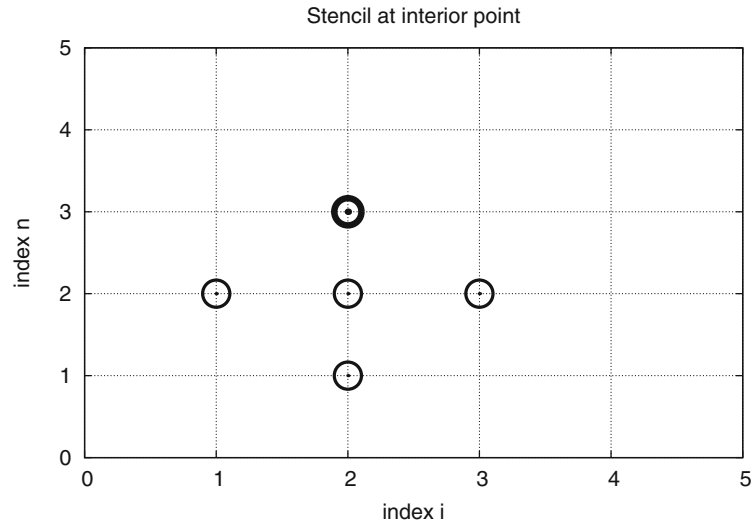

Fig. 2.1 Mesh in space and time. The circles show points connected in a finite difference equation

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

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

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 available for $i = 0, \ldots, N_x$. The only unknown quantity in (2.11) is therefore u_i^{n+1}, which we now can solve for:

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

We have here 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, together with N_x, that governs the quality of the numerical solution (see Sect. 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.

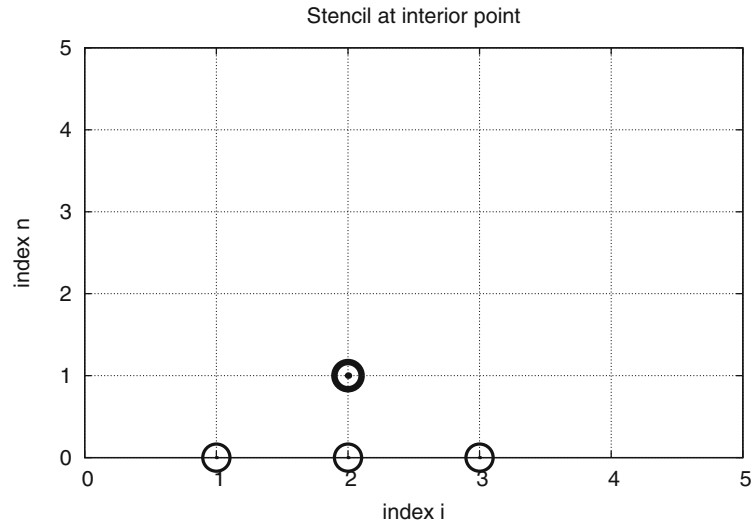

Fig. 2.2 Modified stencil for the first time step

Given that u_i^{n-1} and u_i^n are known for $i = 0, \ldots, N_x$, we find new values at the next time level by applying the formula (2.14) 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 $u_i^{n+1} = 0$.

Even though sound reasoning leads up to (2.14), there is still a minor challenge with it that needs to be resolved. Think of the very first computational step to be made. The scheme (2.14) is supposed to start at $n = 1$, which means that we compute u^2 from u^1 and u^0. Unfortunately, we do not know the value of u^1, so how to proceed? A standard procedure in such cases is to apply (2.14) also for $n = 0$. This immediately seems strange, since it involves u_i^{-1}, which is an undefined quantity outside the time mesh (and the time domain). However, we can use the initial condition (2.13) 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) for $i = 1, 2, \ldots, N_x - 1$ and set $u_i^1 = 0$ for the boundary points given by $i = 0$ and $i = N_x$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
   (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 having $i = 0$, $i = N_x$.

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[1] or a movie file[2].

2.1.6 Sketch of an Implementation

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

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

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

```
# 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_n[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_n[i] - \
           0.5*C**2(u_n[i+1] - 2*u_n[i] + u_n[i-1])
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

# Switch variables before next step
u_nm1[:], u_n[:] = u_n, 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_n[i] - u_nm1[i] - \
               C**2(u_n[i+1] - 2*u_n[i] + u_n[i-1])

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

    # Switch variables before next step
    u_nm1[:], u_n[:] = u_n, u
```

[1] http://tinyurl.com/hbcasmj/book/html/mov-wave/D_stencil_gpl/index.html
[2] http://tinyurl.com/gokgkov/mov-wave/D_stencil_gpl/movie.ogg

2.2 Verification

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

2.2.1 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 $u_{\mathrm{e}}(0,t) = u_{\mathrm{e}}(L,t) = 0$, as well as initial conditions $I(x) = A\sin\left(\frac{\pi}{L}x\right)$ and $V = 0$.

How to use exact solutions for verification

It is common to use such exact solutions of physical interest to verify implementations. However, the numerical solution u_i^n will only be an approximation to $u_e(x_i, t_n)$. 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 $u_e(x_i, t_n)$ are caused by mathematical approximations or programming errors. In particular, if plots of the computed solution u_i^n and the exact one (2.25) look similar, many are tempted to claim that the implementation works. However, even if color plots look nice and the accuracy is "deemed good", there can still be serious programming errors present!

The only way to use exact physical solutions like (2.25) for serious and thorough verification is to run a series of simulations on finer and finer meshes, measure the integrated error in each mesh, and from this information estimate the empirical convergence rate of the method.

An introduction to the computing of convergence rates is given in Section 3.1.6 in [9]. There is also a detailed example on computing convergence rates in Sect. 1.2.2.

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

2.2.3 Manufactured Solution and Estimation of Convergence Rates

Specifying the solution and computing corresponding data 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

$$u_e(x,t) = x(L-x)\sin t\,.$$

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

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

Defining a single discretization parameter To verify the code, we compute the convergence rates in a series of simulations, letting each simulation use a finer mesh than the previous one. Such empirical estimation of convergence rates relies on an

assumption that some measure E of the numerical error is related to the discretization parameters through

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

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

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

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

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

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

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

$$E = ||e_i^n||_{\ell^\infty} = \max_{i,n} |e_i^n| . \tag{2.27}$$

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 ($n = N_t$):

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

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

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

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

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

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

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

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

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

2.2.4 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 Sect. 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

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

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 $t_n = n\Delta t$ so that

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

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

Hence,

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

Similarly, we get that

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

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

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

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

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

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 $u_{tt} = c^2 u_{xx}$ we shall see that there is always another much more powerful way of generating exact solutions (which consists in just setting $C = 1$ (!), as shown in Sect. 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 Sect. 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 to do with this solution, e.g., visualize the curve, analyze the values, or write the array to file for later use. The decision about what to do is left to the user in the form of a user-supplied function

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

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

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

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

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

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

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

2.3.2 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
    # Make sure dx and dt are compatible with x and t
    dx = x[1] - x[0]
    dt = t[1] - t[0]

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

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

    import time;  t0 = time.clock()  # Measure CPU time

    # Load initial condition into u_n
    for i in range(0,Nx+1):
        u_n[i] = I(x[i])

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

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

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

    # Switch variables before next step
    u_nm1[:] = u_n;  u_n[:] = 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_nm1[i] + 2*u_n[i] + \
                     C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
                     dt**2*f(x[i], t[n])

        # 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_nm1[:] = u_n;  u_n[:] = u

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

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

- Although we give `dt` and compute `dx` via `C` and `c`, the resulting `t` and `x` meshes do not necessarily correspond exactly to these values because of rounding errors. To explicitly ensure that `dx` and `dt` correspond to the cell sizes in `x` and `t`, we recompute the values.
- According to the particular choice made in Sect. 2.3.1, a true value returned from `user_action` should terminate the simulation. This is here implemented by a `break` statement inside the for loop in the solver.

2.3.3 Verification: Exact Quadratic Solution

We use the test problem derived in Sect. 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 pytest to check that all test functions with names `test_*()` in this file work:

Terminal

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

2.3.4 Verification: Convergence Rates

A more general method, but not so reliable as a verification method, is to compute the convergence rates and see if they coincide with theoretical estimates. Here we expect a rate of 2 according to the various results in Sect. 2.10. A general function for computing convergence rates can be written like this:

```
def convergence_rates(
    u_exact,                # Python function for exact solution
    I, V, f, c, L,          # physical parameters
    dt0, num_meshes, C, T):  # numerical parameters
    """
    Half the time step and estimate convergence rates for
    for num_meshes simulations.
    """
    # First define an appropriate user action function
    global error
    error = 0  # error computed in the user action function

    def compute_error(u, x, t, n):
        global error  # must be global to be altered here
        # (otherwise error is a local variable, different
        # from error defined in the parent function)
        if n == 0:
            error = 0
        else:
            error = max(error, np.abs(u - u_exact(x, t[n])).max())
```

```
    # Run finer and finer resolutions and compute true errors
    E = []
    h = []  # dt, solver adjusts dx such that C=dt*c/dx
    dt = dt0
    for i in range(num_meshes):
        solver(I, V, f, c, L, dt, C, T,
               user_action=compute_error)
        # error is computed in the final call to compute_error
        E.append(error)
        h.append(dt)
        dt /= 2  # halve the time step for next simulation
    print 'E:', E
    print 'h:', h
    # Convergence rates for two consecutive experiments
    r = [np.log(E[i]/E[i-1])/np.log(h[i]/h[i-1])
         for i in range(1,num_meshes)]
    return r
```

Using the analytical solution from Sect. 2.2.2, we can call `convergence_rates` to see if we get a convergence rate that approaches 2 and use the final estimate of the rate in an `assert` statement such that this function becomes a proper test function:

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

    r = convergence_rates(
        u_exact=u_exact,
        I=lambda x: u_exact(x, 0),
        V=lambda x: 0,
        f=0,
        c=1,
        L=L,
        dt0=0.1,
        num_meshes=6,
        C=0.9,
        T=1)
    print 'rates sin(x)*cos(t) solution:', \
          [round(r_,2) for r_ in r]
    assert abs(r[-1] - 2) < 0.002
```

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

2.3.5 Visualization: Animating the Solution

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

Function for administering the simulation The following `viz` function

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

```
def viz(
    I, V, f, c, L, dt, C, T,  # PDE parameters
    umin, umax,               # Interval for u in plots
    animate=True,             # Simulation with animation?
    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 simulation
    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 Sect. 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 an `ffmpeg` or `avconv` command with the proper arguments for each of the formats Flash, MP4, WebM, and Ogg. The task is greatly simplified by having a `codec2ext` dictionary for mapping video codec names to filename extensions. As mentioned in Sect. 1.3.2, only two formats are actually needed to ensure that all browsers can successfully play the video: MP4 and WebM.

Some animations having a large number of plot files may not be properly combined into a video using `ffmpeg` or `avconv`. 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`) is included by `n % skip_frame == 0`, as well as every `skip_frame`-th frame. As `n % skip_frame == 0` will very seldom be true for the very final frame, we must also check if `n == t.size-1` to get the final frame included.

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

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

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

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

```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*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[3]. The string should ideally consist of straight segments, but these are somewhat wavy due to numerical approximation. Run the case with the `wave1D_u0.py` code and $C = 1$ to see the exact solution.

2.3.7 Working with a Scaled PDE Model

Depending on the model, it may be a substantial job to establish consistent and relevant physical parameter values for a case. The guitar string example illustrates the point. However, by *scaling* the mathematical problem we can often reduce the need to estimate physical parameters dramatically. The scaling technique consists of introducing new independent and dependent variables, with the aim that the absolute values of these lie in $[0, 1]$. We introduce the dimensionless variables (details are found in Section 3.1.1 in [11])

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

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

We get by the chain rule that

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

Similarly,

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

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

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

Dropping the bars, we arrive at the scaled PDE

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \quad x \in (0,1),\ t \in (0, cT/L), \tag{2.34}$$

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

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

and

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

[3] http://tinyurl.com/hbcasmj/wave/html/mov-wave/guitar_C0.8/movie.html

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

2.4 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. That is, vectorization not only speeds up the code on serial computers, but also makes it easy to exploit parallel computing. Actually, there are Python tools like Numba[4] that can automatically turn vectorized code into parallel code.

[4] http://numba.pydata.org

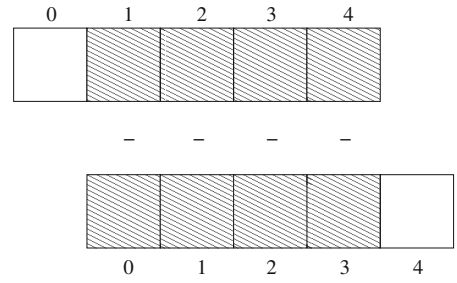

Fig. 2.3 Illustration of subtracting two slices of two arrays

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 Fig. 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*.

Test your understanding

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

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

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

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

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

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

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

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

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

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

Common mistakes with array slices

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

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

and write

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

Now `u[1:n]` has wrong length (`n-1`) compared to the other array slices, causing a `ValueError` and the message `could not broadcast input array from shape 103 into shape 104` (if `n` is 105). When such errors occur one must closely examine all the slices. Usually, it is easier to get upper limits of slices right when they use `-1` or `-2` or empty limit rather than expressions involving the length.

Another common mistake, when `u2` has length `n`, is to forget the slice in the array on the left-hand side,

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

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

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

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

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

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

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

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

2.4.2 Finite Difference Schemes Expressed as Slices

We now have the necessary tools to vectorize the wave equation algorithm as described mathematically in Sect. 2.1.5 and through code in Sect. 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. Within the time loop, the space loop reads:

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

The vectorized version becomes

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

or

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

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

2.4.3 Verification

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

```
def test_quadratic():
    """
    Check the scalar and vectorized versions for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # 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 Sect. 2.3.5. All of this `viz` function can be reused, except the call to `solver_function`. This call lacks the parameter `version`, which we want to set to `vectorized` and `scalar` for our efficiency measurements.

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

Solution 1 Calling `viz` in `wave1D_u0` with `solver_function` as our new solver in `wave1D_u0v` works fine, since this solver has `version='vectorized'` as default value. The problem arises when we want to test `version='scalar'`. The simplest solution is then to use `wave1D_u0.solver` instead. We make a new `viz` function in `wave1D_u0v.py` that has a `version` argument and that just calls `wave1D_u0.viz`:

```
def viz(
    I, V, f, c, L, dt, C, T,  # PDE parameters
    umin, umax,               # Interval for u in plots
    animate=True,             # Simulation with animation?
    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 and fancier solution featuring a very useful trick: we can make a new function that will always call `wave1D_u0v.solver` with `version='scalar'`. The `functools.partial` function from standard Python takes a function `func` as argument and a series of positional and keyword arguments and returns a new function that will call `func` with the supplied arguments, while the user can control all the other arguments in `func`. Consider a trivial example,

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

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

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

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

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

Back to our `viz` code, we can do

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

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

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_nm1` and `u_n` arrays such that they have the right content for the next time step:

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

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

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

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

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

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

updating the elements in u will imply updating the elements in u_n too, thereby destroying the solution at the previous time step.

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

The correct switch of references is

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

We can get rid of the temporary reference tmp by writing

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

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

Caution

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

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

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 Sect. 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. Let the `all_u` list be an attribute in this class and implement the user action function as a method (the special method `__call__` is a natural choice). The class versions avoid that the user action function depends on parameters defined outside the function (such as `all_u` in Exercise 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. Write a program that resembles `wave1D_u0_s2c.py` in Exercise 2.3, but with a `viz` function that can take a list of `C` values as argument and create a movie with solutions corresponding to the given `C` values. The `plot_u` function must be changed to store the solution in an array (see Exercise 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 correspond to the same time point. The easiest remedy is to keep the time resolution constant and change the space resolution to change the Courant number. Note that each spatial grid is needed for the final plotting, so it is an option to store those grids too.
Filename: `wave_numerics_comparison`.

Exercise 2.5: Implementing the solver function as a generator
The callback function `user_action(u, x, t, n)` is called from the `solver` function (in, e.g., `wave1D_u0.py`) at every time level and lets the user work perform desired actions with the solution, like plotting it on the screen. We have implemented the callback function in the typical way it would have been done in C and Fortran. Specifically, the code looks like

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

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

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

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

The program control is directed back to the calling code:

```
for u, x, t, n in solver(...):
    # Do something with u at t[n]
```

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

Implement the `solver` function as a generator, and plot the solution at each time step.
Filename: `wave1D_u0_generator`.

Project 2.6: 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. Show that if F_i is already computed, we can find F_{i+1} from

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

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

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[5] or a movie file[6] 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.

[5] http://tinyurl.com/hbcasmj/book/html/mov-wave/demo_BC_gaussian/index.html
[6] http://tinyurl.com/gokgkov/mov-wave/demo_BC_gaussian/movie.flv

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

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

Boundary condition terminology

Boundary conditions that specify the value of $\partial u/\partial n$ (or shorter u_n) are known as Neumann[7] conditions, while Dirichlet conditions[8] 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

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

for $i = 0$, 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

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

[7] http://en.wikipedia.org/wiki/Neumann_boundary_condition
[8] http://en.wikipedia.org/wiki/Dirichlet_conditions

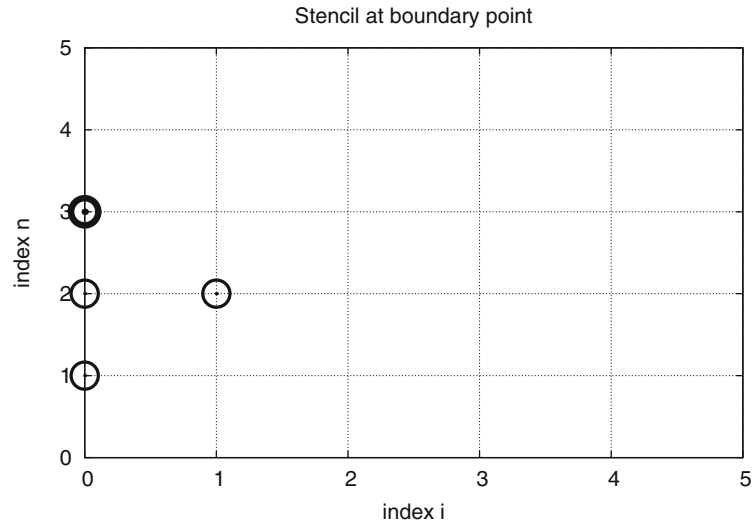

Fig. 2.4 Modified stencil at a boundary with a Neumann condition

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

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + 2C^2\left(u^n_{i-1} - u^n_i\right), \quad i = N_x\,. \tag{2.40}$$

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[9] or a movie file[10].

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^n_{i-1} by u^n_{i+1} when computing u^{n+1}_i from the stencil formula for $i = 0$. Similarly, we replace u^n_{i+1} by u^n_{i-1} in the stencil formula for $i = N_x$. This observation can conveniently be used in the coding: we just work with the general stencil formula, but write the code such that it is easy to replace `u[i-1]` by `u[i+1]` and vice versa. This is achieved by having the indices `i+1` and `i-1` as variables `ip1` (`i` plus 1) and `im1` (`i` minus 1), respectively. At the boundary we can easily define `im1=i+1` while we use `im1=i-1` in the internal parts of the mesh. Here are the details of the implementation (note that the updating formula for `u[i]` is the general stencil formula):

[9] http://tinyurl.com/hbcasmj/book/html/mov-wave/N_stencil_gpl/index.html
[10] http://tinyurl.com/gokgkov/mov-wave/N_stencil_gpl/movie.ogg

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])
```

We can in fact create one loop over both the internal and boundary points and use only one updating formula:

```
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_n[i] + C2*(u_n[im1] - 2*u_n[i] + u_n[ip1])
```

The program `wave1D_n0.py` contains a complete implementation of the 1D wave equation with boundary conditions $u_x = 0$ at $x = 0$ and $x = L$.

It would be nice to modify the `test_quadratic` test case from the `wave1D_u0.py` with Dirichlet conditions, described in Sect. 2.4.3. However, the Neumann conditions require the polynomial variation in the x direction to be of third degree, which causes challenging problems when designing a test where the numerical solution is known exactly. Exercise 2.15 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 Sect. 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, followed by $i \in \mathcal{I}_x$, instead of $i = 0, \ldots, N_x$. Obviously, $\mathcal{I}_x$ must be the index set $\mathcal{I}_x = \{0, \ldots, N_x\}$, but it is often advantageous to have a symbol for this set rather than specifying all its elements (all the time, as we have done up to now). This new notation saves writing and makes specifications of algorithms and their implementation as computer code simpler.

The first index in the set will be denoted $\mathcal{I}_x^0$ and the last $\mathcal{I}_x^{-1}$. When we need to skip the first element of the set, we use $\mathcal{I}_x^+$ for the remaining subset $\mathcal{I}_x^+ = \{1, \ldots, N_x\}$. Similarly, if the last element is to be dropped, we write $\mathcal{I}_x^- = \{0, \ldots, N_x - 1\}$ for the remaining indices. All the indices corresponding to inner grid points are specified by $\mathcal{I}_x^i = \{1, \ldots, N_x - 1\}$. For the time domain we find it natural to explicitly use 0 as the first index, so we will usually write $n = 0$ and t_0 rather than $n = \mathcal{I}_t^0$. We also avoid notation like $x_{\mathcal{I}_x^{-1}}$ and will instead use x_i, $i = \mathcal{I}_x^{-1}$.

The Python code associated with index sets applies the following conventions:

Notation	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}_x^0$	`Ix[0]`
$\mathcal{I}_x^{-1}$	`Ix[-1]`
$\mathcal{I}_x^-$	`Ix[:-1]`
$\mathcal{I}_x^+$	`Ix[1:]`
$\mathcal{I}_x^i$	`Ix[1:-1]`

Why index sets are useful

An important feature of the index set notation is that it keeps our formulas and code independent of how we count mesh points. For example, the notation $i \in \mathcal{I}_x$ or $i = \mathcal{I}_x^0$ remains the same whether $\mathcal{I}_x$ is defined as above or as starting at 1, i.e., $\mathcal{I}_x = \{1, \ldots, Q\}$. Similarly, we can in the code define `Ix=range(Nx+1)` or `Ix=range(1,Q)`, and expressions like `Ix[0]` and `Ix[1:-1]` remain correct. One application where the index set notation is convenient is conversion of code from a language where arrays has base index 0 (e.g., Python and C) to languages where the base index is 1 (e.g., MATLAB and Fortran). Another important application is implementation of Neumann conditions via ghost points (see next section).

For the current problem setting in the x, t plane, we work with the index sets

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

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_n[i] - 0.5*C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
```

```
# Time loop
for n in It[1:-1]:
    # Compute internal points
    for i in Ix[1:-1]:
        u[i] = - u_nm1[i] + 2*u_n[i] + \
               C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
    # 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 $u_x = 0$
- $t = 0$: $u = I(x)$
- $t = 0$: $u_t = V(x)$

The program combines Dirichlet and Neumann conditions, scalar and vectorized implementation of schemes, and the index set notation into one piece of code. A lot of test examples are also included in the program:

- A rectangular plug-shaped initial condition. (For $C = 1$ the solution will be a rectangle that jumps one cell per time step, making the case well suited for verification.)
- A Gaussian function as initial condition.
- A triangular profile as initial condition, which resembles the typical initial shape of a guitar string.
- A sinusoidal variation of u at $x = 0$ and either $u = 0$ or $u_x = 0$ at $x = L$.
- An analytical solution $u(x,t) = \cos(m\pi t/L)\sin(\frac{1}{2}m\pi x/L)$, which can be used for convergence rate tests.

2.6.5 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 or Neumann conditions at $x = 0$ and $x = L$. It is tempting to apply a quadratic solution as described in Sect. 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 for
    a constant u(x,t). We simulate in [0, L] and apply
    Neumann and Dirichlet conditions at both ends.
    """
    u_const = 0.45
    u_exact = lambda x, t: u_const
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0
    f = lambda x, t: 0

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(u - u_e).max()
        msg = 'diff=%E, t_%d=%g' % (diff, n, t[n])
        tol = 1E-13
        assert diff < tol, msg

    for U_0 in (None, lambda t: u_const):
        for U_L in (None, lambda t: u_const):
            L = 2.5
            c = 1.5
            C = 0.75
            Nx = 3  # Very coarse mesh for this exact test
            dt = C*(L/Nx)/c
            T = 18  # long time integration

            solver(I, V, f, c, U_0, U_L, L, dt, C, T,
                   user_action=assert_no_error,
                   version='scalar')
            solver(I, V, f, c, U_0, U_L, L, dt, C, T,
                   user_action=assert_no_error,
                   version='vectorized')
            print U_0, U_L
```

The quadratic solution is very useful for testing, but it requires Dirichlet conditions at both ends.

Another test may utilize the fact that the approximation error vanishes when the Courant number is unity. We can, for example, start with a plug profile as initial condition, let this wave split into two plug waves, one in each direction, and check that the two plug waves come back and form the initial condition again after "one period" of the solution process. Neumann conditions can be applied at both ends. A proper test function reads

```
def test_plug():
    """Check that an initial plug is correct back after one period."""
    L = 1.0
    c = 0.5
    dt = (L/10)/c  # Nx=10
    I = lambda x: 0 if abs(x-L/2.0) > 0.1 else 1
```

```
    u_s, x, t, cpu = solver(
        I=I,
        V=None, f=None, c=0.5, U_0=None, U_L=None, L=L,
        dt=dt, C=1, T=4, user_action=None, version='scalar')
    u_v, x, t, cpu = solver(
        I=I,
        V=None, f=None, c=0.5, U_0=None, U_L=None, L=L,
        dt=dt, C=1, T=4, user_action=None, version='vectorized')
    tol = 1E-13
    diff = abs(u_s - u_v).max()
    assert diff < tol
    u_0 = np.array([I(x_) for x_ in x])
    diff = np.abs(u_s - u_0).max()
    assert diff < tol
```

Other tests must rely on an unknown approximation error, so effectively we are left with tests on the convergence rate.

2.6.6 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 $[L, L + \Delta x]$, known as *ghost cells*, to the mesh gives us all the needed mesh points, corresponding to $i = -1, 0, \ldots, N_x, N_x + 1$. The extra points with $i = -1$ and $i = N_x + 1$ are known as *ghost points*, and values at these points, u^n_{-1} and $u^n_{N_x+1}$, are called *ghost values*.

The important idea is to ensure that we always have

$$u^n_{-1} = u^n_1 \quad \text{and} \quad u^n_{N_x+1} = u^n_{N_x-1},$$

because then the application of the standard scheme at a boundary point $i = 0$ or $i = N_x$ will be correct and guarantee that the solution is compatible with the boundary condition $u_x = 0$.

Some readers may find it strange to just extend the domain with ghost cells as a general technique, because in some problems there is a completely different medium with different physics and equations right outside of a boundary. Nevertheless, one should view the ghost cell technique as a purely mathematical technique, which is valid in the limit $\Delta x \to 0$ and helps us to implement derivatives.

Implementation The `u` array now needs extra elements corresponding to the ghost points. Two new point values are needed:

```
u   = zeros(Nx+3)
```

The arrays `u_n` and `u_nm1` must be defined accordingly.

Unfortunately, a major indexing problem arises with ghost cells. The reason is that Python indices *must* start at 0 and `u[-1]` will always mean the last element in `u`. This fact gives, apparently, a mismatch between the mathematical indices $i = -1, 0, \ldots, N_x + 1$ and the Python indices running over `u`: `0,..,Nx+2`. One remedy is to change the mathematical indexing of i in the scheme and write

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

instead of $i = 0, \ldots, N_x$ as we have previously used. The ghost points now correspond to $i = 0$ and $i = N_x + 1$. A better solution is to use the ideas of Sect. 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_nm1[i] + 2*u_n[i] + \
           C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1])
```

The indexing becomes a bit more complicated when we call functions like `V(x)` and `f(x, t)`, as we must remember that the appropriate x coordinate is given as `x[i-Ix[0]]`:

```
for i in Ix:
    u[i] = u_n[i] + dt*V(x[i-Ix[0]]) + \
           0.5*C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
           0.5*dt2*f(x[i-Ix[0]], t[0])
```

It remains to update the solution at ghost points, i.e., `u[0]` and `u[-1]` (or `u[Nx+2]`). For a boundary condition $u_x = 0$, the ghost value must equal the value at the associated inner mesh point. Computer code makes this statement precise:

```
i = Ix[0]          # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1]         # x=L boundary
u[i+1] = u[i-1]
```

The physical solution to be plotted is now in `u[1:-1]`, or equivalently `u[Ix[0]:Ix[-1]+1]`, so this slice is the quantity to be returned from a solver function. A complete implementation appears in the program `wave1D_n0_ghost.py`.

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

becomes wrong, since `u_n` and `x` have different lengths and the index `i` corresponds to two different mesh points. In fact, `x[i]` corresponds to `u[1+i]`. A correct implementation is

```
for i in Ix:
    u_n[i] = I(x[i-Ix[0]])
```

Similarly, a source term usually coded as `f(x[i], t[n])` is incorrect if `x` is defined to be the physical points, so `x[i]` must be replaced by `x[i-Ix[0]]`.

An alternative remedy is to let `x` also cover the ghost points such that `u[i]` is the value at `x[i]`.

The ghost cell is only added to the boundary where we have a Neumann condition. Suppose we have a Dirichlet condition at $x = L$ and a homogeneous Neumann condition at $x = 0$. One ghost cell $[-\Delta x, 0]$ is added to the mesh, so the index set for the physical points becomes $\{1, \ldots, N_x + 1\}$. A relevant implementation is

```
u = zeros(Nx+2)
Ix = range(1, u.shape[0])
...
for i in Ix[:-1]:
    u[i] = - u_nm1[i] + 2*u_n[i] + \
           C2*(u_n[i-1] - 2*u_n[i] + u_n[i+1]) + \
           dt2*f(x[i-Ix[0]], t[n])
i = Ix[-1]
u[i] = U_0       # set Dirichlet value
i = Ix[0]
u[i-1] = u[i+1]  # update ghost value
```

The physical solution to be plotted is now in `u[1:]` or (as always) `u[Ix[0]:Ix[-1]+1]`.

2.7 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 meets the boundary where c jumps from c_1 to c_2, a part of the wave is reflected back into the first medium (the *reflected* wave), while one part is transmitted through the second medium (the *transmitted* wave).

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

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

This is the most frequent form of a wave equation with variable wave velocity, but other forms also appear, see Sect. 2.14.1 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

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

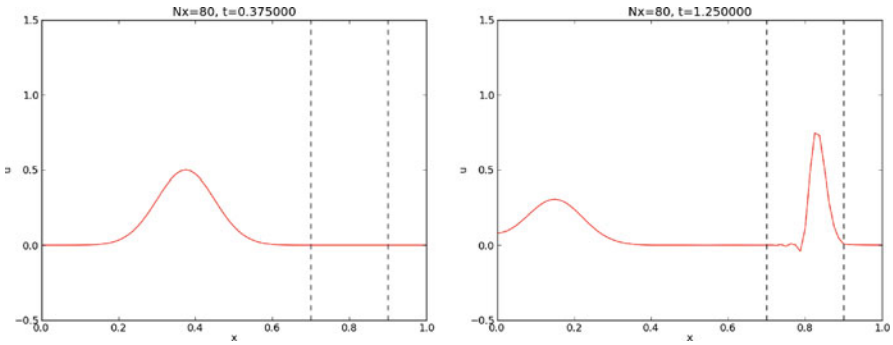

Fig. 2.5 *Left:* wave entering another medium; *right:* transmitted and reflected wave

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:

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

Then discretize

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}}\frac{u_{i+1}^n - u_i^n}{\Delta x} = [qD_x u]_{i+\frac{1}{2}}^n .$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}}\frac{u_i^n - u_{i-1}^n}{\Delta x} = [qD_x u]_{i-\frac{1}{2}}^n .$$

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right) . \quad (2.43)$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_x(\overline{q}^x D_x u)]_i^n . \quad (2.44)$$

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 $[D_x(qD_x u)]_i^n$.

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

$$[D_x q D_x u + D_{2x} q D_{2x} u]_i^n, \quad (2.45)$$

Writing this out shows that it is different from $[D_x(qD_x u)]_i^n$ and lacks the physical interpretation of net flux into a cell. With a smooth and slowly varying $q(x)$

the differences between the two discretizations are not substantial. However, when q exhibits (potentially large) jumps, $[D_x(qD_xu)]_i^n$ 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 dominates and very few mention the alternative in (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} must then be done by *interpolation* techniques, of which three are of particular interest in this context:

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

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

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

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 Sect. 1.10.2 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:

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n \,. \tag{2.49}$$

Strictly speaking, $[D_x \overline{q}^x D_x u]_i^n = [D_x(\overline{q}^x D_x u)]_i^n$.

From the compact difference notation we immediately see what kind of differences that each term is approximated with. The notation $\overline{q}^x$ also specifies that the variable coefficient is approximated by an arithmetic mean, the definition being $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i + q_{i+1})/2$.

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

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

2.7.4 How a Variable Coefficient Affects the Stability

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

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

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 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

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

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

$$\begin{aligned} u_i^{n+1} &= -u_i^{n-1} + 2u_i^n \\ &\quad + \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \qquad (2.52)\\ &= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \qquad (2.53)\\ &\approx -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_i(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n\,. \qquad (2.54) \end{aligned}$$

Here we used the approximation

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

An alternative derivation may apply the arithmetic mean of $q_{n-\frac{1}{2}}$ and $q_{n+\frac{1}{2}}$ in (2.53), leading to the term

$$\left(q_i + \frac{1}{2}(q_{i+1} + q_{i-1})\right)(u^n_{i-1} - u^n_i)\,.$$

Since $\frac{1}{2}(q_{i+1} + q_{i-1}) = q_i + \mathcal{O}(\Delta x^2)$, we can approximate with $2q_i(u^n_{i-1} - u^n_i)$ for $i = N_x$ and get the same term as we did above.

A common technique when implementing $\partial u/\partial x = 0$ boundary conditions, is to assume $dq/dx = 0$ as well. This implies $q_{i+1} = q_{i-1}$ and $q_{i+1/2} = q_{i-1/2}$ for $i = N_x$. The implications for the scheme are

$$\begin{aligned} u^{n+1}_i &= -u^{n-1}_i + 2u^n_i \\ &\quad + \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u^n_{i-1} - u^n_i) - q_{i-\frac{1}{2}}(u^n_i - u^n_{i-1})\right) \\ &\quad + \Delta t^2 f^n_i \qquad (2.56) \\ &= -u^{n-1}_i + 2u^n_i + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_{i-\frac{1}{2}}(u^n_{i-1} - u^n_i) + \Delta t^2 f^n_i\,. \qquad (2.57) \end{aligned}$$

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_nm1[i] + 2*u_n[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_n[i+1] - u_n[i])  - \
               0.5*(q[i] + q[i-1])*(u_n[i] - u_n[i-1])) + \
           dt2*f(x[i], t[n])
```

The coefficient `C2` is now defined as `(dt/dx)**2`, i.e., *not* as the squared Courant number, since the wave velocity is variable and appears inside the parenthesis.

With Neumann conditions $u_x = 0$ at the boundary, we need to combine this scheme with the discrete version of the boundary condition, as shown in Sect. 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 Sect. 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_nm1[i] + 2*u_n[i] + \
       C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
           0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
       dt2*f(x[i], t[n])
```

With ghost cells we can just reuse the formula for the interior points also at the boundary, provided that the ghost values of both u and q are correctly updated to ensure $u_x = 0$ and $q_x = 0$.

A vectorized version of the scheme with a variable coefficient at internal mesh points becomes

```
u[1:-1] = - u_nm1[1:-1] + 2*u_n[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_n[2:] - u_n[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_n[1:-1] - u_n[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

2.7.7 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. (2.14.1) 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 as a simple factor in front of a derivative. There is hence no need for any averaging of ϱ. Often, ϱ will be moved to the right-hand side, also without any difficulty:

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

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

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

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

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

The only new feature here is the time-dependent Dirichlet conditions, but they are trivial to implement:

```
i = Ix[0]  # x=0
u[i] = U_0(t[n+1])

i = Ix[-1] # x=L
u[i] = U_L(t[n+1])
```

The `solver` function is a natural extension of the simplest `solver` function in the initial `wave1D_u0.py` program, extended with Neumann boundary conditions ($u_x = 0$), time-varying Dirichlet conditions, as well as a variable wave velocity. The different code segments needed to make these extensions have been shown and commented upon in the preceding text. We refer to the `solver` function in the `wave1D_dn_vc.py` file for all the details. Note in that `solver` function, however, that the technique of "hashing" is used to check whether a certain simulation has been run before, or not. This technique is further explained in Sect. C.2.3.

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'
            np.savez(fname, **kwargs)
            self.t.append(t[n])  # store corresponding time value
            if n == 0:           # save x once
                np.savez('.' + self.filename + '_x.dat', x=x)

        # 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 Sect. C.2.

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 jumps in the wave velocity (i.e., discontinuous media). The code is shown to demonstrate how easy it is to reach this flexibility with the building blocks we have already developed:

```
def pulse(
    C=1,            # Maximum Courant number
    Nx=200,         # spatial resolution
    animate=True,
    version='vectorized',
    T=2,            # end time
    loc='left',     # location of initial condition
    pulse_tp='gaussian',  # pulse/init.cond. type
    slowness_factor=2, # inverse of wave vel. in right medium
    medium=[0.7, 0.9], # interval for right medium
    skip_frame=1,      # skip frames in animations
    sigma=0.05         # width measure of the pulse
    ):
    """
    Various peaked-shaped initial conditions on [0,1].
    Wave velocity is decreased by the slowness_factor inside
    medium. The loc parameter can be 'center' or 'left',
    depending on where the initial pulse is to be located.
    The sigma parameter governs the width of the pulse.
    """
    # 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 np.exp(-0.5*((x-xc)/sigma)**2)
    elif pulse_tp == 'plug':
        def I(x):
            return 0 if abs(x-xc) > sigma else 1
    elif pulse_tp == 'cosinehat':
        def I(x):
            # One period of a cosine
            w = 2
            a = w*sigma
            return 0.5*(1 + np.cos(np.pi*(x-xc)/a)) \
                   if xc - a <= x <= xc + a else 0

    elif pulse_tp == 'half-cosinehat':
        def I(x):
            # Half a period of a cosine
            w = 4
            a = w*sigma
            return np.cos(np.pi*(x-xc)/a) \
                   if xc - 0.5*a <= x <= xc + 0.5*a else 0
    else:
        raise ValueError('Wrong pulse_tp="%s"' % pulse_tp)

    def c(x):
        return c_0/slowness_factor \
               if medium[0] <= x <= medium[1] else c_0
```

```
    umin=-0.5; umax=1.5*I(xc)
    casename = '%s_Nx%s_sf%s' % \
               (pulse_tp, Nx, slowness_factor)
    action = PlotMediumAndSolution(
        medium, casename=casename, umin=umin, umax=umax,
        skip_frame=skip_frame, screen_movie=animate,
        backend=None, filename='tmpdata')

    # 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
    cpu, hashed_input = solver(
        I=I, V=None, f=None, c=c,
        U_0=None, U_L=None,
        L=L, dt=dt, C=C, T=T,
        user_action=action,
        version=version,
        stability_safety_factor=1)

    if cpu > 0:  # did we generate new data?
        action.close_file(hashed_input)
        action.make_movie_file()
    print 'cpu (-1 means no new data generated):', cpu

def convergence_rates(
    u_exact,
    I, V, f, c, U_0, U_L, L,
    dt0, num_meshes,
    C, T, version='scalar',
    stability_safety_factor=1.0):
    """
    Half the time step and estimate convergence rates for
    for num_meshes simulations.
    """
    class ComputeError:
        def __init__(self, norm_type):
            self.error = 0

        def __call__(self, u, x, t, n):
            """Store norm of the error in self.E."""
            error = np.abs(u - u_exact(x, t[n])).max()
            self.error = max(self.error, error)

    E = []
    h = []  # dt, solver adjusts dx such that C=dt*c/dx
    dt = dt0
    for i in range(num_meshes):
        error_calculator = ComputeError('Linf')
        solver(I, V, f, c, U_0, U_L, L, dt, C, T,
               user_action=error_calculator,
               version='scalar',
               stability_safety_factor=1.0)
        E.append(error_calculator.error)
        h.append(dt)
        dt /= 2  # halve the time step for next simulation
    print 'E:', E
    print 'h:', h
    r = [np.log(E[i]/E[i-1])/np.log(h[i]/h[i-1])
         for i in range(1,num_meshes)]
    return r
```

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

    r = convergence_rates(
        u_exact=u_exact,
        I=lambda x: u_exact(x, 0),
        V=lambda x: 0,
        f=0,
        c=1,
        U_0=0,
        U_L=0,
        L=L,
        dt0=0.1,
        num_meshes=6,
        C=0.9,
        T=1,
        version='scalar',
        stability_safety_factor=1.0)
    print 'rates sin(x)*cos(t) solution:', \
          [round(r_,2) for r_ in r]
    assert abs(r[-1] - 2) < 0.002
```

The `PlotMediumAndSolution` class used here is a subclass of `PlotAndStoreSolution` where the medium with reduced c value, as specified by the `medium` interval, is visualized in the plots.

Comment on the choices of discretization parameters

The argument N_x in the `pulse` function does not correspond to the actual spatial resolution of $C < 1$, since the `solver` function takes a fixed Δt and C, and adjusts Δx accordingly. As seen in the `pulse` function, the specified Δt is chosen according to the limit $C = 1$, so if $C < 1$, Δt remains the same, but the `solver` function operates with a larger Δx and smaller N_x than was specified in the call to `pulse`. The practical reason is that we always want to keep Δt fixed such that plot frames and movies are synchronized in time regardless of the value of C (i.e., Δx is varied when the Courant number varies).

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

```
>>> import wave1D_dn_vc as w
>>> w.pulse(Nx=50, loc='left', pulse_tp='cosinehat', slowness_factor=2)
```

To easily kill the graphics by Ctrl-C and restart a new simulation it might be easier to run the above two statements from the command line with

Terminal

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

2.9 Exercises

Exercise 2.7: 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 and zero source term. A starting point is the standing wave solution from Exercise 2.1. It becomes necessary to include a damping term $e^{-\beta t}$ and also have both a sine and cosine component in time:

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

Find k from the boundary conditions $u(0,t) = u(L,t) = 0$. Then use the PDE to find constraints on β, ω, A, and B. Set up a complete initial-boundary value problem and its solution.
Filename: `damped_waves`.

Problem 2.8: Explore symmetry boundary conditions
Consider the simple "plug" wave where $\Omega = [-L, L]$ and

$$I(x) = \begin{cases} 1, & x \in [-\delta, \delta], \\ 0, & \text{otherwise} \end{cases}$$

for some number $0 < \delta < L$. The other initial condition is $u_t(x,0) = 0$ and there is no source term f. The boundary conditions can be set to $u = 0$. The solution to this problem is symmetric around $x = 0$. This means that we can simulate the wave process in only half of the domain $[0, L]$.

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

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 on $[-L, L]$. Thereafter, utilize the symmetry of the solution and run a simulation in half of the domain $[0, L]$, using a boundary condition at $x = 0$. Compare plots from the two solutions and confirm that they are the same.
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.
d) If the code works correctly, the solution $u(x,t) = x(L-x)(1+\frac{t}{2})$ should be reproduced exactly. Write a test function `test_quadratic` that checks whether this is the case. Simulate for x in $[0, \frac{L}{2}]$ with a symmetry condition at the end $x = \frac{L}{2}$.

Filename: `wave1D_symmetric`.

Exercise 2.9: Send pulse waves through a layered medium
Use the `pulse` function in `wave1D_dn_vc.py` to investigate sending a pulse, located with its peak at $x = 0$, through two media with different wave velocities. The

(scaled) velocity in the left medium is 1 while it is $\frac{1}{s_f}$ in the right medium. Report what happens with a Gaussian pulse, a "cosine hat" pulse, half a "cosine hat" pulse, and a plug pulse for resolutions $N_x = 40, 80, 160$, and $s_f = 2, 4$. Simulate until $T = 2$.
Filename: `pulse1D`.

Exercise 2.10: Explain why numerical noise occurs
The experiments performed in Exercise 2.9 shows considerable numerical noise in the form of non-physical waves, especially for $s_f = 4$ and the plug pulse or the half a "cosinehat" pulse. The noise is much less visible for a Gaussian pulse. Run the case with the plug and half a "cosinehat" pulse for $s_f = 1$, $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion relation to explain the observations.
Filename: `pulse1D_analysis`.

Exercise 2.11: 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 2.9?
Filename: `pulse1D_harmonic`.

Problem 2.12: 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

$$[D_t^+ u + cD_x^- u = 0]_i^n, \quad i = N_x . \tag{2.72}$$

From this equation, one can solve for $u_{N_x}^{n+1}$ 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:
(a) Disturbance in the middle of the domain, $I(x) = g(x; L/2, s)$, and open boundary condition at the left end.
(b) Disturbance at the left end, $I(x) = g(x; 0, s)$, and $u_x = 0$ as symmetry boundary condition at this end.
Make test functions for both cases, testing that the solution is zero after the waves have left the domain.

d) In 2D and 3D it is difficult to compute the correct wave velocity normal to the boundary, which is needed in generalizations of the open boundary conditions in higher dimensions. Test the effect of having a slightly wrong wave velocity in (2.72). Make 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 2.13: Implement periodic boundary conditions
It is frequently of interest to follow wave motion over large distances and long times. A straightforward approach is to work with a very large domain, but that might lead to a lot of computations in areas of the domain where the waves cannot be noticed. A more efficient approach is to let a right-going wave out of the domain and at the same time let it enter the domain on the left. This is called a *periodic boundary condition*.

The boundary condition at the right end $x = L$ is an open boundary condition (see Exercise 2.12) 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 2.8) $u_x = 0$, or an open boundary condition.

This initial wave will split in two and either be reflected or transported out of the domain at $x = 0$. The purpose of the exercise is to follow the right-going wave. We can do that with a *periodic boundary condition*. This means that when the right-going wave hits the boundary $x = L$, the open boundary condition lets the wave out of the domain, but at the same time we use a boundary condition on the left end $x = 0$ that feeds the outgoing wave into the domain again. This periodic condition is simply $u(0) = u(L)$. The switch from $u_x = 0$ or an open boundary condition at the left end to a periodic condition can happen when $u(L, t) > \epsilon$, where $\epsilon = 10^{-4}$ might be an appropriate value for determining when the right-going wave hits the boundary $x = L$.

The open boundary conditions can conveniently be discretized as explained in Exercise 2.12. 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 2.14: 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 approxi-

mation 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

$$[D_t D_t u]_i^n = \frac{1}{\Delta x}\left([qD_x u]^n_{i+\frac{1}{2}} - [qD_x u]^n_{i-\frac{1}{2}}\right) + [f]_i^n,$$

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 $[qD_x u]^n_{i+\frac{1}{2}} = 0$. 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 2.15: Verification by a cubic polynomial in space

The purpose of this exercise is to verify the implementation of the `solver` function in the program `wave1D_n0.py` by using an exact numerical solution for the wave equation $u_{tt} = c^2 u_{xx} + f$ with Neumann boundary conditions $u_x(0,t) = u_x(L,t) = 0$.

A similar verification is used in the file `wave1D_u0.py`, which solves the same PDE, but with Dirichlet boundary conditions $u(0,t) = u(L,t) = 0$. The idea of the verification test in function `test_quadratic` in `wave1D_u0.py` is to produce a solution that is a lower-order polynomial such that both the PDE problem, the boundary conditions, and all the discrete equations are exactly fulfilled. Then the `solver` function should reproduce this exact solution to machine precision. More precisely, we seek $u = X(x)T(t)$, with $T(t)$ as a linear function and $X(x)$ as a parabola that fulfills the boundary conditions. Inserting this u in the PDE determines f. It turns out that u also fulfills the discrete equations, because the truncation error of the discretized PDE has derivatives in x and t of order four and higher. These derivatives all vanish for a quadratic $X(x)$ and linear $T(t)$.

It would be attractive to use a similar approach in the case of Neumann conditions. We set $u = X(x)T(t)$ and seek lower-order polynomials X and T. To force u_x to vanish at the boundary, we let X_x be a parabola. Then X is a cubic polynomial. The fourth-order derivative of a cubic polynomial vanishes, so $u = X(x)T(t)$ will fulfill the discretized PDE also in this case, if f is adjusted such that u fulfills the PDE.

However, the discrete boundary condition is not exactly fulfilled by this choice of u. The reason is that

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

At the two boundary points, we must demand that the derivative $X_x(x) = 0$ such that $u_x = 0$. However, u_{xxx} is a constant and not zero when $X(x)$ is a cubic polynomial. Therefore, our $u = X(x)T(t)$ fulfills

$$[D_{2x}u]_i^n = \frac{1}{6}u_{xxx}(x_i, t_n)\Delta x^2,$$

and not

$$[D_{2x}u]_i^n = 0, \quad i = 0, N_x,$$

as it should. (Note that all the higher-order terms $\mathcal{O}(\Delta x^4)$ also have higher-order derivatives that vanish for a cubic polynomial.) So to summarize, the fundamental problem is that u as a product of a cubic polynomial and a linear or quadratic polynomial in time is not an exact solution of the discrete boundary conditions.

To make progress, we assume that $u = X(x)T(t)$, where T for simplicity is taken as a prescribed linear function $1 + \frac{1}{2}t$, and $X(x)$ is taken as an *unknown* cubic polynomial $\sum_{j=0}^{3} a_j x^j$. There are two different ways of determining the coefficients $a_0, \ldots, a_3$ such that both the discretized PDE and the discretized boundary conditions are fulfilled, under the constraint that we can specify a function $f(x, t)$ for the PDE to feed to the `solver` function in `wave1D_n0.py`. Both approaches are explained in the subexercises.

a) One can insert u in the discretized PDE and find the corresponding f. Then one can insert u in the discretized boundary conditions. This yields two equations for the four coefficients $a_0, \ldots, a_3$. To find the coefficients, one can set $a_0 = 0$ and $a_1 = 1$ for simplicity and then determine a_2 and a_3. This approach will make a_2 and a_3 depend on Δx and f will depend on both Δx and Δt.
Use `sympy` to perform analytical computations. A starting point is to define u as follows:

```
def test_cubic1():
    import sympy as sm
    x, t, c, L, dx, dt = sm.symbols('x t c L dx dt')
    i, n = sm.symbols('i n', integer=True)

    # 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 $[D_x D_x u]_i^n$, $[D_t D_t u]_i^n$, 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: $f_i^n = [D_t D_t u - c^2 D_x D_x u]_i^n$. 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 $\Delta t = CL/(N_x c)$, 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 a plain Python function `e_exact(x,t,L,dx,dt,c)`, one can write

```
e_exact = sm.lambdify([x,t,L,dx,dt,c], e, 'numpy')
```

The `'numpy'` argument is a good habit as the `e_exact` function will then work with array arguments if it contains mathematical functions (but here we only do plain arithmetics, which automatically work with arrays).

b) An alternative way of determining $a_0, \ldots, a_3$ is to reason as follows. We first construct $X(x)$ such that the boundary conditions are fulfilled: $X = x(L - x)$. However, to compensate for the fact that this choice of X does not fulfill the discrete boundary condition, we seek u such that

$$u_x = \frac{\partial}{\partial x} x(L - x)T(t) - \frac{1}{6} u_{xxx} \Delta x^2,$$

since this u will fit the discrete boundary condition. Assuming $u = T(t) \sum_{j=0}^{3} a_j x^j$, we can use the above equation to determine the coefficients a_1, a_2, a_3. A value, e.g., 1 can be used for a_0. The following `sympy` code computes this u:

```
def test_cubic2():
    import sympy as sm
    x, t, c, L, dx = sm.symbols('x t c L dx')
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    # 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 horizontal coordinate $\xi = x - ct$, we see the function $g_R(\xi)$