Finite difference methods for wave equations

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

May 4, 2016

Contents

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.

1   Simulation of waves on a string

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

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \qquad\qquad x \in (0, L), \ t \in (0, T] \qquad (1)$$

$$u(x, 0) = I(x), \qquad\qquad x \in [0, L] \qquad (2)$$

$$\frac{\partial}{\partial t} u(x, 0) = 0, \qquad\qquad x \in [0, L] \qquad (3)$$

$$u(0, t) = 0, \qquad\qquad t \in (0, T] \qquad (4)$$

$$u(L, t) = 0, \qquad\qquad t \in (0, T] \qquad (5)$$

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

Equation (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) specifies the initial shape of the string, $I(x)$, and (3) expresses that the initial velocity of the string is zero. In addition, PDEs need *boundary conditions*, given here as (4) and (5). These two conditions specify that the string is fixed at the ends, i.e., that the displacement u is zero.

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

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

$$u_t = \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2}, \tag{6}$$

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 (1)-(5) will now be discretized in space and time by a finite difference method.

1.1 Discretizing the domain

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

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T. \tag{7}$$

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

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L. \tag{8}$$

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

$$x_i = i\Delta x, \ i = 0, \ldots, N_x, \quad t_n = n\Delta t, \ n = 0, \ldots, N_t. \tag{9}$$

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

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.

1.3 Fulfilling the equation at the mesh points

In the finite difference method, we relax the condition that (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:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \tag{10}$$

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 $u = 0$.

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 (10) and get

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{11}$$

or written more compactly using the operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n. \tag{12}$$

Interpretation of the equation as a stencil. A typical feature of (11) is that it involves u values from neighboring points only: u_i^{n+1}, $u_{i\pm1}^n$, u_i^n, and u_i^{n-1}. The circles in Figure 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 (11), which then involves a coupling of u_2^1, u_1^2, u_2^2, u_3^2, and u_2^3. The term *stencil* is often used about the algebraic equation at a mesh point, and the geometry of a typical stencil is illustrated in Figure 1. One also often refers to the algebraic equations as *discrete equations*, *(finite) difference equations* or a *finite difference scheme*.
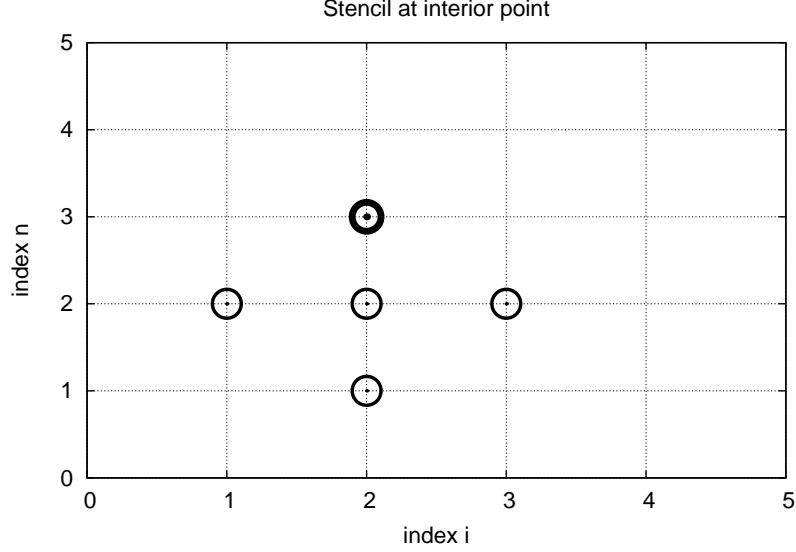
Figure 1: Mesh in space and time. The circles show points connected in a finite difference equation.

Algebraic version of the initial conditions. We also need to replace the derivative in the initial condition (3) by a finite difference approximation. A centered difference of the type

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

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

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

Writing out this equation and ordering the terms give

$$u_i^{-1} = u_i^1, \quad i = 0, \ldots, N_x\,. \tag{13}$$

The other initial condition can be computed by

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

1.5 Formulating a recursive algorithm

We assume that u_i^n and u_i^{n-1} are already computed for $i = 0, \ldots, N_x$. The only unknown quantity in (11) is therefore u_i^{n+1}, which we can solve for:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right), \tag{14}$$

where we have introduced the parameter

$$C = c\frac{\Delta t}{\Delta x},$$ (15)

known as the *Courant number*.

> **C is the key parameter in the discrete wave equation.**
>
> We see that the discrete version of the PDE features only one parameter, C, which is therefore the key parameter that governs the quality of the numerical solution (see Section 10 for details). Both the primary physical parameter c and the numerical parameters Δx and Δt are lumped together in C. Note that C is a dimensionless parameter.

Given that u_i^{n-1} and u_i^n are computed for $i = 0, \ldots, N_x$, we find new values at the next time level by applying the formula (14) for $i = 1, \ldots, N_x - 1$. Figure 1 illustrates the points that are used to compute u_2^3. For the boundary points, $i = 0$ and $i = N_x$, we apply the boundary conditions $u_i^{n+1} = 0$.

A problem with (14) arises when $n = 0$ since the formula for u_i^1 involves u_i^{-1}, which is an undefined quantity outside the time mesh (and the time domain). However, we can use the initial condition (13) in combination with (14) when $n = 0$ to eliminate u_i^{-1} and arrive at a special formula for u_i^1:

$$u_i^1 = u_i^0 - \frac{1}{2}C^2 \left(u_{i+1}^0 - 2u_i^0 + u_{i-1}^0 \right).$$ (16)

Figure 2 illustrates how (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 (16) and set $u_i^1 = 0$ for the boundary points $i = 0$ and $i = N_x$, for $n = 1, 2, \ldots, N - 1$,

3. For each time level $n = 1, 2, \ldots, N_t - 1$

    (a) apply (14) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$

    (b) set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

The algorithm essentially consists of moving a finite difference stencil through all the mesh points, which can be seen as an animation in a web page[1] or a movie file[2].

---

[1]`http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Dirichlet_stencil_gpl/index.html`
[2]`http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Dirichlet_stencil_gpl/movie.ogg`
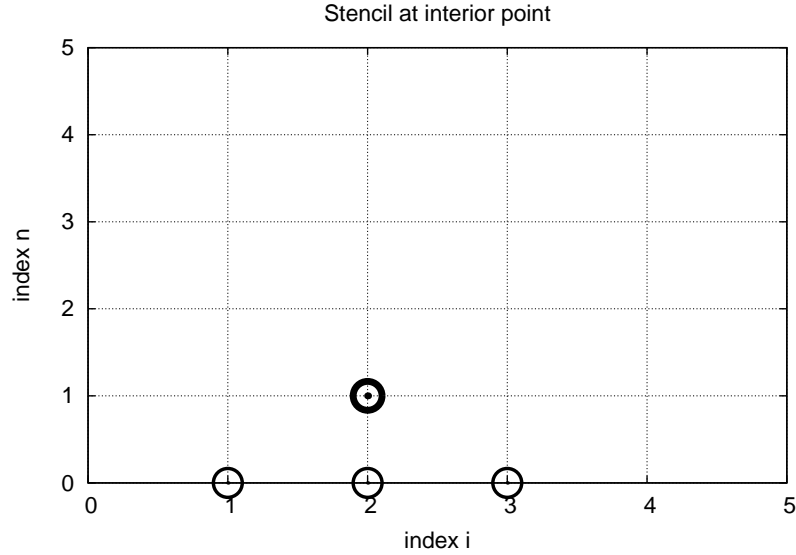
Figure 2: Modified stencil for the first time step.

1.6 Sketch of an implementation

In a Python implementation of this algorithm, we use the array elements `u[i]` to store u_i^{n+1}, `u_1[i]` to store u_i^n, and `u_2[i]` to store u_i^{n-1}. Our naming convention is to use `u` for the unknown new spatial field to be computed, `u_1` as the solution one time step back in time, `u_2` as the solution two time steps back in time and so forth.

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

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

```
# Given mesh points as arrays x and t (x[i], t[n])
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx              # Courant number
Nt = len(t)-1
C2 = C**2                # Help variable in the scheme

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

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

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

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

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

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

2   Verification

Before implementing the algorithm, it is convenient to add a source term to the PDE (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.

2.1   A slightly generalized model problem

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

$$u_{tt} = c^2 u_{xx} + f(x,t), \qquad\qquad x \in (0,L), \ t \in (0,T] \qquad (17)$$
$$u(x,0) = I(x), \qquad\qquad x \in [0,L] \qquad (18)$$
$$u_t(x,0) = V(x), \qquad\qquad x \in [0,L] \qquad (19)$$
$$u(0,t) = 0, \qquad\qquad t > 0 \qquad (20)$$
$$u(L,t) = 0, \qquad\qquad t > 0 \qquad (21)$$

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

$$[D_t D_t u = c^2 D_x D_x u + f]_i^n . \qquad (22)$$

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

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n . \qquad (23)$$

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

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2}C^2\left(u_{i+1}^0 - 2u_i^0 + u_{i-1}^0\right) + \frac{1}{2}\Delta t^2 f_i^0 . \tag{24}$$

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 (1)-(5) allows an exact solution

$$u_{\mathrm{e}}(x,t) = A \sin\left(\frac{\pi}{L}x\right)\cos\left(\frac{\pi}{L}ct\right) . \tag{25}$$

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

How to use exact solutions for verification.

It is common to use such exact solutions of physical interest to verify implementations. However, the numerical solution u_i^n will only be an approximation to $u_{\mathrm{e}}(x_i, t_n)$. We have no knowledge of the precise size of the error in this approximation, and therefore we can never know if discrepancies between u_i^n and $u_{\mathrm{e}}(x_i, t_n)$ are caused by mathematical approximations or programming errors. In particular, if a plot of the computed solution u_i^n and the exact one (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 (25) for serious and thorough verification is to run a series of finer and finer meshes, measure the integrated error in each mesh, and from this information estimate the empirical convergence rate of the method.

An introduction to the computing of convergence rates is given in Section **??** in [2].

In the present problem, one expects the method to have a convergence rate of 2 (see Section 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.

2.3 Manufactured solution

One problem with the exact solution (25) is that it requires a simplification ($V = 0, f = 0$) of the implemented problem (17)-(21). An advantage of using a *manufactured solution* is that we can test all terms in the PDE problem.

The idea of this approach is to set up some chosen solution and fit the source term, boundary conditions, and initial conditions to be compatible with the chosen solution. Given that our boundary conditions in the implementation are $u(0,t) = u(L,t) = 0$, we must choose a solution that fulfills these conditions. One example is

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

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

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

The initial conditions become

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

To verify the code, we compute the convergence rates in a series of simulations, letting each simulation use a finer mesh than the previous one. Such empirical estimation of convergence rates tests 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.

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

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

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

$$E = ||e_i^n||_{\ell^2} = \left(\Delta t \Delta x \sum_{n=0}^{N_t} \sum_{i=0}^{N_x} (e_i^n)^2 \right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n, \quad (26)$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{i,n} |e_n^i| . \quad (27)$$

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

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

$$E = ||e_i^n||_{\ell^2} = \left(\Delta x \sum_{i=0}^{N_x} (e_i^n)^2 \right)^{\frac{1}{2}}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - u_i^n, \quad (28)$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{0 \leq i \leq N_x} |e_i^n| . \quad (29)$$

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

Let E_i be the error measure in experiment (mesh) number i and let h_i be the corresponding discretization parameter (h). With the error model $E_i = D h_i^r$, we can estimate r by comparing two consecutive experiments:

$$E_{i+1} = D h_{i+1}^r,$$
$$E_i = D h_i^r .$$

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.

2.4 Constructing an exact solution of the discrete equations

With a manufactured or known analytical solution, as outlined above, we can estimate convergence rates and see if they have the correct asymptotic behavior.

Experience shows that this is a quite good verification technique in that many common bugs will destroy the convergence rates. A significantly better test though, would be to check that the numerical solution is exactly what it should be. This will in general require exact knowledge of the numerical error, which we do not normally have (although we in Section 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

$$u_e(x,t) = x(L-x)(1+\frac{1}{2}t), \tag{30}$$

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

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

$$[D_t D_t t^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - 2n^2 + (n-1)^2 = 2, \tag{31}$$

$$[D_t D_t t]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - 2n + (n-1))\Delta t}{\Delta t^2} = 0. \tag{32}$$

Hence,

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

Similarly, we get that

$$[D_x D_x u_e]_i^n = (1+\frac{1}{2}t_n)[D_x D_x(xL - x^2)]_i$$
$$= (1+\frac{1}{2}t_n)[LD_x D_x x - D_x D_x x^2]_i$$
$$= -2(1+\frac{1}{2}t_n).$$

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

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

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

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

Notice.

A product of quadratic or linear expressions in the various independent variables, as shown above, will often fulfill both the PDE problem and the discrete equations, and can therefore be very useful solutions for verifying implementations.

However, for 1D wave equations of the type $u_{tt} = c^2 u_{xx}$ we shall see that there is always another much more powerful way of generating exact solutions (which consists in just setting $C = 1$ (!), as shown in Section 10).

---

3   Implementation

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

A real implementation of the basic computational algorithm from Sections 1.5 and 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.

3.1 Callback function for user-specific actions

The solution at all spatial points at a new time level is stored in an array `u` of length $N_x + 1$. We need to decide what do to with this solution, e.g., visualize the curve, analyze the values, or write the array to file for later use. The decision about what to do is left to the user in the form of a user-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-dependent actions (like visualization), it is a common technique to leave those actions to user-supplied callback functions.

3.2 The solver function

A first attempt at a solver function is listed below.

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

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

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

    if user_action is not None:
        user_action(u_1, x, t, 0)
```

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

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

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

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

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

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

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

A remark on the computation of `dx` and `dt` is probably 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.

3.3   Verification: exact quadratic solution

We use the test problem derived in Section 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
```

16

```
    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
```

3.4 Visualization: animating the solution

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

Function for administering the simulation.   The following `viz` function

1. defines a `user_action` callback function for plotting the solution at each time level,

2. calls the `solver` function, and

3. combines all the plots (in files) to video in different formats.

```
def viz(
    I, V, f, c, L, dt, C, T,  # PDE 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 simulaton
    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(
        I, V, f, c, L, dt, C, T, user_action)

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

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

Dissection of the code. The `viz` function can either use SciTools or Matplotlib for visualizing the solution. The `user_action` function based on SciTools is called `plot_u_st`, while the `user_action` function based on Matplotlib is

a bit more complicated as it is realized as a class and needs statements that differ from those for making static plots. SciTools can utilize both Matplotlib and Gnuplot (and many other plotting programs) for doing the graphics, but Gnuplot is a relevant choice for large N_x or in two-dimensional problems as Gnuplot is significantly faster than Matplotlib for screen animations.

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

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. We use the `ffmpeg` (or `avconv`) program to combine individual plot files to movies in modern formats: Flash, MP4, Webm, and Ogg. A typical `ffmpeg` (or `avconv`) command for creating a movie file in Ogg format with 4 frames per second built from a collection of plot files with names generated by `frame_%04d.png`, look like

```Terminal
Terminal> ffmpeg -r 25 -i frame_%04d.png -c:v libtheora movie.ogg
```

The different formats require different video encoders (`-c:v`) to be installed: Flash applies `flv`, WebM applies `libvpx`, and MP4 applies `libx264`:

```
Terminal> ffmpeg -r 25 -i frame_%04d.png -c:v flv movie.flv
Terminal> ffmpeg -r 25 -i frame_%04d.png -c:v libvpx movie.webm
Terminal> ffmpeg -r 25 -i frame_%04d.png -c:v libx264 movie.mp4
```

Players like `vlc`, `mplayer`, `gxine`, and `totem` can be used to play these movie files.

Note that padding the frame counter with zeros in the `frame_*.png` files, as specified by the `%04d` format, is essential so that the wildcard notation `frame_*.png` expands to the correct set of files.

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

A simple choice of numbers may illustrate the formulas: say we have 801 frames in total (`t.size`) and we allow only 60 frames to be plotted. Then we need to plot every 801/60 frame, which with integer division yields 13 as `every`.

Using the mod function, `n % every`, this operation is zero every time `n` can be divided by 13 without a remainder. That is, the `if` test is true when `n` equals $0, 13, 26, 39, ..., 780, 801$. The associated code is included in the `plot_u` function in the file `wave1D_u0v.py`[3].

3.5 Running a case

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

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{33}$$

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

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

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

```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

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

    umin = -1.2*a;  umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax,
              animate=True, tool='scitools')
```

[3]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0v.py`

The associated program has the name `wave1D_u0.py`[4]. Run the program and watch the movie of the vibrating string[5].

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

3.6   Working with a scaled PDE model

Depending on the model, it may be a substantial job to establish consistent and relevant physical parameter values for a case. The guitar string example illustrates the point. However, by *scaling* the mathematical problem we can often reduce the need to estimate physical parameters dramatically. The scaling technique consists of introducing new independent and dependent variables, with the aim that the absolute values of these lie in $[0, 1]$. We introduce the dimensionless variables (details are found in Section **??** in [3]

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

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

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial \bar{t}}\left(\frac{\bar{u}}{a}\right)\frac{d\bar{t}}{dt} = \frac{\partial \bar{u}}{\partial \bar{t}}\frac{1}{a}\frac{c}{L} = \frac{ac}{L}\frac{\partial \bar{u}}{\partial \bar{t}}.$$

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 $f = 0$,

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

Dropping the bars, we arrive at the scaled PDE

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, 1),\ t \in (0, cT/L), \tag{34}$$

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

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

and

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

resulting in

---

[4] `http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0.py`
[5] `http://tinyurl.com/opdfafk/pub/mov-wave/guitar_C0.8/index.html`

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

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 (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 $-\omega^2 = -c^2 k^2$, i.e., $\omega = kc$. The period is therefore $P = 2\pi/(kc)$. If the boundary conditions are $u(0, t) = u(0, L)$, we need to have $kL = n\pi$ for integer n. The period becomes $P = 2L/nc$. The longest period is $P = 2L/c$. The dimensionless period $\tilde{P}$ 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 $\tilde{P} = 2/n$ $(n > 1)$.

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[6] that can automatically turn vectorized code into parallel code.

4.1 Operations on slices of arrays

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

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

All the differences here are independent of each other. The computation of `d` can therefore alternatively be done by subtracting the array $(u_0, u_1, \ldots, u_{n-1})$ from the array where the elements are shifted one index upwards: $(u_1, u_2, \ldots, u_n)$, see Figure 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

---

[6]`http://numba.pydata.org`
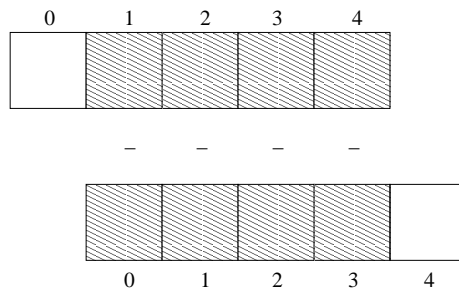
Figure 3: Illustration of subtracting two slices of two arrays.

```
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 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.

4.2 Finite difference schemes expressed as slices

We now have the necessary tools to vectorize the wave equation algorithm as described mathematically in Section 1.5 and through code in Section 3.2. There are three loops: one for the initial condition, one for the first time step, and finally the loop that is repeated for all subsequent time levels. Since only the latter is repeated a potentially large number of times, we limit our vectorization efforts to this loop:

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

The vectorized version becomes

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

or

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

The program `wave1D_u0v.py`[7] 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).

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')
```

---

[7]http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0v.py

4.4   Efficiency measurements

The `wave1D_u0v.py` contains our new `solver` function with both scalar and vectorized code. For comparing the efficiency of scalar versus vectorized code, we need a `viz` function as discussed in Section 3.4. All of this `viz` function can be reused, except the call to `solver_function`. This call lacks the parameter `version`, which we want to set to `vectorized` and `scalar` for our efficiency measurements.

One solution is to copy the `viz` code from `wave1D_u0` into `wave1D_u0v.py` and add a `version` argument to the `solver_function` call. Taking into account how much 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='vectorized'`. The simplest solution is then to use `wave1D_u0.solver` instead. We make a new `viz` function in `wave1D_u0v.py` that has a `version` argument and that just calls `wave1D_u0.viz`:

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

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

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

4.5   Remark on the updating of arrays

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

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

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

The assignment `u_1[:] = u` copies the content of the `u` array into the elements of the `u_1` array. Such copying takes time, but that time is negligible compared to the time needed for computing `u` from the finite difference formula, even when the formula has a vectorized implementation. However, efficiency of program code is a key topic when solving PDEs numerically (particularly when there are two or three space dimensions), so it must be mentioned that there exists a much more efficient way of making the arrays `u_2` and `u_1` ready for the next time step. The idea is based on *switching references* and explained as follows.

A Python variable is actually a reference to some object (C programmers may think of pointers). Instead of copying data, we can let `u_2` refer to the `u_1` object and `u_1` refer to the `u` object. This is a very efficient operation (like switching pointers in C). A naive implementation like

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

will fail, however, because now `u_2` refers to the `u_1` object, but then the name `u_1` refers to `u`, so that this `u` object has two references, `u_1` and `u`, while our third array, originally referred to by `u_2`, has no more references and is lost. This means that the variables `u`, `u_1`, and `u_2` refer to two arrays and not three. Consequently, the computations at the next time level will be messed up since

updating the elements in `u` will imply updating the elements in `u_1` too so the solution at the previous time step, which is crucial in our formulas, is destroyed.

While `u_2 = u_1` is fine, `u_1 = u` is problematic, so the solution to this problem is to ensure that `u` points to the `u_2` array. This is mathematically wrong, but new correct values will be filled into `u` at the next time step and make it right.

The correct switch of references is

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

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

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

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

> **Caution:**
>
> The update `u_2, u_1, u = u_1, u, u_2` leaves wrong content in `u` at the final time step. This means that if we return `u`, as we do in the example codes here, we actually return `u_2`, which is obviously wrong. It is therefore important to adjust the content of `u` to `u = u_1` before returning `u`.

5   Exercises

Exercise 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

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

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

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

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

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

The wave length in space becomes

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

The period in time becomes

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

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

Solution. See code below.

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

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.

Solution. The code:

```
import sys, os
sys.path.insert(0, os.path.join(os.pardir, os.pardir,
                                'src-wave', 'wave1D'))

from wave1D_u0 import solver
#from wave1D_u0v import solver  # allows faster vectorized operations
import numpy as np

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

    class Plot:
        def __init__(self, ymax, frame_name='frame'):
```

```
                self.max_error = []    # hold max amplitude errors
                self.max_error_t = [] # time points corresp. to max_error
                self.frame_name = frame_name
                self.ymax = ymax

        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if animate == 'u and u_exact':
                plt.plot(x, u, 'r-',
                         x, u_exact(x, t[n]), 'b--',
                         xlabel='x', ylabel='u',
                         axis=[0, L, -self.ymax, self.ymax],
                         title='t=%f' % t[n], show=True)
            else:
                error = u_exact(x, t[n]) - u
                local_max_error = np.abs(error).max()
                # self.max_error holds the increasing amplitude error
                if self.max_error == [] or \
                        local_max_error > max(self.max_error):
                    self.max_error.append(local_max_error)
                    self.max_error_t.append(t[n])
                # Use user's ymax until the error exceeds that value.
                # This gives a growing max value of the yaxis (but
                # never shrinking)
                self.ymax = max(self.ymax, max(self.max_error))
                plt.plot(x, error, 'r-',
                         xlabel='x', ylabel='error',
                         axis=[0, L, -self.ymax, self.ymax],
                         title='t=%f' % t[n], show=True)
            plt.savefig('%s_%04d.png' % (self.frame_name, n))

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

    plot = Plot(ymax)
    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T, plot)

    # Make plot of max error versus time
    plt.figure()
    plt.plot(plot.max_error_t, plot.max_error)
    plt.xlabel('time'); plt.ylabel('max abs(error)')
    plt.savefig('error.png')
    plt.savefig('error.pdf')

    # Make .flv movie file
    fps = 4  # Frames per second
    codec2ext = dict(flv='flv') #, libx64='mp4',
                                #libvpx='webm', libtheora='ogg')

    filespec = 'frame_%04d.png'
    movie_program = 'avconv'  # or 'ffmpeg'
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
              '-vcodec %(codec)s %(movie_filename)s.%(ext)s' % vars()
        os.system(cmd)

def simulations():
    from numpy import sin, cos, pi
    L = 12   # length of domain
    m = 9    # 2L/m: wave length or period in space (2*pi/k, k=pi*m/L)
    c = 2    # wave velocity
    A = 1    # amplitude
```

```
    Nx = 80
    C = 0.8
    P = 2*pi/(pi*m*c/L)  # 1 period in time
    #T = 10*P
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

    def u_exact(x, t):
        return A*sin(pi*m*x/L)*cos(pi*m*c*t/L)

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

    V = 0
    f = 0

    viz(I, V, f, c, L, dt, C, 10.5*P,
        0.1, u_exact,
        animate='error',
        movie_filename='error')

    # Very long simulation to demonstrate different curves
    viz(I, V, f, c, L, dt, C, 30*P,
        1.2*A, u_exact,
        animate='u and u_exact',
        movie_filename='solution')

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

Filename: `wave_standing`.

Remarks. The important parameters for numerical quality are C and $k\Delta x$, where $C = c\Delta t/\Delta x$ is the Courant number and k is defined above ($k\Delta x$ is proportional to how many mesh points we have per wave length in space, see Section 10.4 for explanation).

Exercise 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.

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

```python
#!/usr/bin/env python
"""
1D wave equation with u=0 at the boundary.
Simplest possible implementation.

The key function is::

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

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

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

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

import numpy as np

def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = np.linspace(0, Nt*dt, Nt+1)    # Mesh points in time
    dx = dt*c/float(C)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)        # Mesh points in space
    C2 = C**2                          # Help variable in the scheme
    # 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_1 = np.zeros(Nx+1)   # Solution at 1 time level back
    u_2 = np.zeros(Nx+1)   # Solution at 2 time levels back

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

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

    if user_action is not None:
        user_action(u_1, x, t, 0)

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

    if user_action is not None:
```

```python
            user_action(u, x, t, 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

def viz(
    I, V, f, c, L, dt, C, T,   # PDE parameters
    umin, umax,                # Interval for u in plots
    animate=True,              # Simulation with animation?
    tool='matplotlib',         # 'matplotlib' or 'scitools'
    solver_function=solver,    # Function with numerical algorithm
    ):
    """Run solver, store and visualize u at each time level."""

    all_u = []
    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
        plt.savefig('tmp_%04d.png' % n)  # for movie making
        all_u.append(u.copy())

    class PlotMatplotlib:
        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if n == 0:
                plt.ion()
                self.lines = plt.plot(x, u, 'r-')
                plt.xlabel('x');  plt.ylabel('u')
                plt.axis([0, L, umin, umax])
                plt.legend(['t=%f' % t[n]], loc='lower left')
            else:
                self.lines[0].set_ydata(u)
                plt.legend(['t=%f' % t[n]], loc='lower left')
                plt.draw()
            time.sleep(2) if t[n] == 0 else time.sleep(0.2)
            plt.savefig('tmp_%04d.png' % n)  # for movie making

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

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

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

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

```
        for codec in codec2ext:
            ext = codec2ext[codec]
            cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
                  '-vcodec %(codec)s movie.%(ext)s' % vars()
            os.system(cmd)

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

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
    from math import pi
    w = 2*pi*freq
    num_periods = 1
    T = 2*pi/w*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, all_u = viz(I, 0, 0, c, L, dt, C, T, umin, umax,
                     animate=True, tool='scitools')
    # checking
    #for e in all_u:
    #    print e[int(len(all_u[1])/2)]

if __name__ == '__main__':
    #test_quadratic()
    import sys
    try:
        C = float(sys.argv[1])
        print 'C=%g' % C
    except IndexError:
        C = 0.85
    print 'Courant number: %.2f' % C
    guitar(C)
```

Filename: `wave1D_u0_s_store`.

Exercise 3: Use a class for the user action function

Redo Exercise 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).

Solution. Using a class, we get

```python
#!/usr/bin/env python
"""
1D wave equation with u=0 at the boundary.
Simplest possible implementation.

The key function is::

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

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

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

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

import numpy as np

def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = np.linspace(0, Nt*dt, Nt+1)    # Mesh points in time
    dx = dt*c/float(C)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)         # Mesh points in space
    C2 = C**2                           # Help variable in the scheme
    # 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_1 = np.zeros(Nx+1)    # Solution at 1 time level back
    u_2 = np.zeros(Nx+1)    # Solution at 2 time levels back

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

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

    if user_action is not None:
        user_action(u_1, x, t, 0)

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

    if user_action is not None:
```

```
            user_action(u, x, t, 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

def viz(
    I, V, f, c, L, dt, C, T,   # PDE parameters
    umin, umax,               # Interval for u in plots
    animate=True,             # Simulation with animation?
    tool='matplotlib',        # 'matplotlib' or 'scitools'
    solver_function=solver,   # Function with numerical algorithm
    ):
    """Run solver, store and visualize u at each time level."""

    class PlotUst:
        def __init__(self):
            self.all_u = []
        def __call__(self, 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('tmp_%04d.png' % n)  # for movie making
            self.all_u.append(u.copy())

    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 = PlotUst()
    import time, glob, os

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

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

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

```
        movie_program = 'ffmpeg'  # or 'avconv'
        for codec in codec2ext:
            ext = codec2ext[codec]
            cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
                  '-vcodec %(codec)s movie.%(ext)s' % vars()
            os.system(cmd)

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

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
    from math import pi
    w = 2*pi*freq
    num_periods = 1
    T = 2*pi/w*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, all_u = viz(I, 0, 0, c, L, dt, C, T, umin, umax,
                     animate=True, tool='scitools')
    # checking
    #for e in all_u:
    #    print e[int(len(all_u[1])/2)]

if __name__ == '__main__':
    #test_quadratic()
    import sys
    try:
        C = float(sys.argv[1])
        print 'C=%g' % C
    except IndexError:
        C = 0.85
    print 'Courant number: %.2f' % C
    guitar(C)
```

Filename: `wave1D_u0_s2c`.

Exercise 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 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 or 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.

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

```python
#!/usr/bin/env python
"""
1D wave equation with u=0 at the boundary.
Simplest possible implementation.

The key function is::

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

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

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

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

import numpy as np

def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = np.linspace(0, Nt*dt, Nt+1)   # Mesh points in time
    dx = c*dt/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
    # Recompute to 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_1 = np.zeros(Nx+1)    # Solution at 1 time level back
    u_2 = np.zeros(Nx+1)    # Solution at 2 time levels back

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

    # Load initial condition into u_1
```

43

```
            for i in range(0,Nx+1):
                u_1[i] = I(x[i])

            if user_action is not None:
                user_action(u_1, x, t, 0)

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

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

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

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

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

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

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

    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, store and viz. u at each time level with all C values.
        """

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

                    if len(self.all_u_for_all_C) == len(C):  # all C done
```

```python
                        print 'Finished all C. Proceed with plots...'
                        # note: n will here be the last index in t[n]
                        for n_ in range(0, n+1):       # for each tn
                            plt.plot(self.x_mesh[0],
                                         self.all_u_for_all_C[0][n_],
                                         axis=[0, L, umin, umax],
                                         title='Solutions for all \
                                                 C at t=%f' % t[n_])
                            plt.hold('on')

                            for j in range(1, len(C)):
                                # build plot at this tn with each
                                # sol. from the different C values
                                plt.plot(self.x_mesh[j],
                                             self.all_u_for_all_C[j][n_],
                                             axis=[0, L, umin, umax])
                            plt.xlabel('x'); plt.ylabel('u')
                            plt.hold('off')
                            plt.show()
                            # Let the init. cond. stay on the screen for
                            # 2 sec, 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('tmp_%04d.png' % n_)  # for movie


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

                if len(self.all_u_for_all_C) == len(C):  # all C done
                    print 'Finished all C. Proceed with plots...'
                    plt.ion()
                    # note: n will here be the last index in t[n]
                    for n_ in range(0, n+1):        # for each tn
                        plt.plot(x, self.all_u_for_all_C[0][n_])
                        plt.axis([0, L, umin, umax])
                        plt.hold(True)
                        for j in range(1, len(C)):
                            # build plot at this tn with each
                            # sol. from the different C values
                            plt.plot(x, self.all_u_for_all_C[j][n_])
                        plt.axis([0, L, umin, umax])
                        plt.xlabel('x'); plt.ylabel('u')
                        plt.title('Solutions for all \
                                     C at t=%f' % t[n_])
                        plt.hold(False)
                        plt.draw()
                        # Let the init. cond. stay on the screen for
                        # 2 sec, 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('tmp_%04d.png' % n_)  # for movie

    if tool == 'matplotlib':
```

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

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

    # Call solver and do the simulaton
    user_action = plot_u if animate else None
    for C_value in C:
        print 'C_value ------------------------------- ', C_value
        u, x, t, cpu = solver_function(
                    I, V, f, c, L, dt, C_value, T, user_action)

    # 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

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
    from math import pi
    w = 2*pi*freq
    num_periods = 1
    T = 2*pi/w*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c
    dx = dt*c/float(C)
    # Now dt is considered fixed and a list of C
    # values is made by reducing increasing the dx value
    # in steps of 10%.
    all_C = [C]
    all_C.append(c*dt/(1.1*dx))
    all_C.append(c*dt/(1.2*dx))

    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, all_C, T, umin, umax,
              animate=True, tool='scitools')
```

```
        #cpu = viz(I, 0, 0, c, L, dt, all_C, T, umin, umax,
        #                animate=True, tool='matplotlib')
        print 'cpu = ', cpu

    if __name__ == '__main__':
        import sys
        try:
            C = float(sys.argv[1])
            print 'C=%g' % C
        except IndexError:
            C = 0.85
        print 'Courant number: %.2f' % C
        # The list of C values will be generated from this C value
        guitar(C)
```

Filename: `wave_numerics_comparison`.

Project 5: Calculus with 1D mesh functions

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

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

Solution. See code below.

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

Solution. See code below.

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

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

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

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

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

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

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

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

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

This may further be written as

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

Finally, this gives

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

See code below for implementation.

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

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]`.

Solution. See code below.

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

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

Solution. See code below.

Solution. The final version of the code reads

```
# -*- coding: utf-8 -*-
"""
Calculus with a 1D mesh function.
"""
import numpy as np

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

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

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

def test_differentiate():
    def f(x):
        return 4*x - 2.5
    def dfdx(x):
        derivatives = np.zeros(len(x))
        derivatives[:] = 4
        return derivatives

    a = 0; b = 1; Nx = 10
    x = np.linspace(a, b, Nx+1)
    exact_dfdx = dfdx(x)          # compute exact derivatives
    # test vectorized version
    calc_v = MeshCalculus(vectorized=True)
    num_dfdx  = calc_v.differentiate(f, x)
    print np.abs(num_dfdx - exact_dfdx)
    diff = np.abs(num_dfdx - exact_dfdx).max()
    tol = 1E-14
```

```
        assert diff < tol
        # test scalar version
        calc = MeshCalculus(vectorized=False)
        num_dfdx  = calc.differentiate(f, x)
        print np.abs(num_dfdx - exact_dfdx)
        diff = np.abs(num_dfdx - exact_dfdx).max()
        assert diff < tol

    def test_integrate():
        def f(x):
            return 4*x - 2.5
        a = 0; b = 1; Nx = 10
#        a = 2.5/4; b = 10; Nx = 2
        x = np.linspace(a, b, Nx+1)
        # The exact integral amounts to the total area of two triangles
        I_exact = 0.5*abs(2.5/4 - a)*f(a) + 0.5*abs(b - 2.5/4)*f(b)
        # test vectorized version
        calc_v = MeshCalculus(vectorized=True)
        F = calc_v.integrate(f, x)
        print F, I_exact
        diff = np.abs(F[-1] - I_exact)
        print diff
        tol = 1E-14
        assert diff < tol
        # test scalar version
        calc = MeshCalculus(vectorized=False)
        F = calc.integrate(f, x)
        print F, I_exact
        diff = np.abs(F[-1] - I_exact)
        print diff
        assert diff < tol


    if __name__ == '__main__':
        test_differentiate()
        test_integrate()
```

Filename: `mesh_calculus_1D`.

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[8] or a movie file[9] 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.

---

[8]`http://tinyurl.com/opdfafk/pub/mov-wave/demo_BC_gaussian/index.html`
[9]`http://tinyurl.com/opdfafk/pub/mov-wave/demo_BC_gaussian/movie.flv`

6.1 Neumann boundary condition

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

$$\frac{\partial u}{\partial n} \equiv \boldsymbol{n} \cdot \nabla u = 0 \,. \tag{35}$$

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[a] conditions, while Dirichlet conditions[b] 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.

---
[a]http://en.wikipedia.org/wiki/Neumann_boundary_condition
[b]http://en.wikipedia.org/wiki/Dirichlet_conditions

6.2 Discretization of derivatives at the boundary

How can we incorporate the condition (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 (35) at $x = 0$ and $t = t_n$ by the difference

$$[D_{2x}u]_0^n = \frac{u_{-1}^n - u_1^n}{2\Delta x} = 0 \,. \tag{36}$$

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

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \tag{37}$$

we can eliminate the fictitious value u_{-1}^n. We see that $u_{-1}^n = u_1^n$ from (36), which can be used in (37) to arrive at a modified scheme for the boundary point u_0^{n+1}:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2 \left(u_{i+1}^n - u_i^n\right), \quad i = 0 \,. \tag{38}$$

Figure 4 visualizes this equation for computing u_0^3 in terms of u_0^2, u_0^1, and u_1^2.

Similarly, (35) applied at $x = L$ is discretized by a central difference

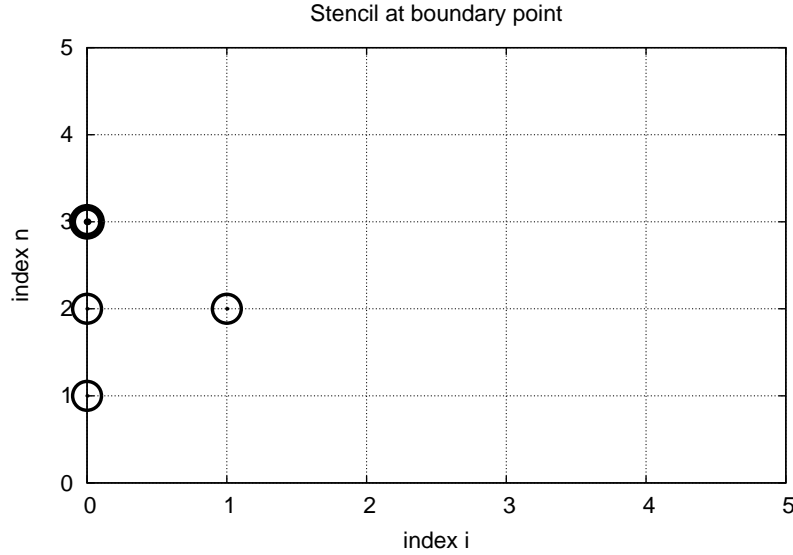$$\frac{u_{N_x+1}^n - u_{N_x-1}^n}{2\Delta x} = 0 \,. \tag{39}$$

Figure 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 $u_{N_x}^{n+1}$:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2 \left(u_{i-1}^n - u_i^n \right), \quad i = N_x.\qquad(40)$$

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[10] or a movie file[11].

6.3   Implementation of Neumann conditions

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

---

[10]`http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Neumann_stencil_gpl/index.html`
[11]`http://tinyurl.com/opdfafk/pub/mov-wave/wave1D_PDE_Neumann_stencil_gpl/movie.ogg`

```
i = 0
ip1 = i+1
im1 = ip1   # i-1 -> i+1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1   # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

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

```
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

The program `wave1D_n0.py`[12] contains a complete implementation of the 1D wave equation with boundary conditions $u_x = 0$ at $x = 0$ and $x = L$.

It would be nice to modify the `test_quadratic` test case from the `wave1D_u0.py` with Dirichlet conditions, described in Section 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 14 outlines ideas and code for this purpose. The only test in `wave1D_n0.py` is to start with a plug wave at rest and see that the initial condition is reached again perfectly after one period of motion, but such a test requires $C = 1$ (so the numerical solution coincides with the exact solution of the PDE, see Section 10.4).

6.4   Index set notation

To improve our mathematical writing and our implementations, it is wise to introduce a special notation for index sets. This means that we write x_i, $i \in \mathcal{I}_x$, instead of $i = 0, \ldots, N_x$. Obviously, $\mathcal{I}_x$ must be the index set $\mathcal{I}_x = \{0, \ldots, N_x\}$, but it is often advantageous to have a symbol for this set rather than specifying all its elements (all the time, as we have done up to now). This new notation saves writing and makes specifications of algorithms and their implementation of computer code simpler.

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

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

---

[12]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0.py`

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

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\}, \tag{41}$$

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

$$u_i^{n+1} = u_i^n - \frac{1}{2}C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right), \quad , i \in \mathcal{I}_x^i, \; n = 0,$$

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right), \quad i \in \mathcal{I}_x^i, \; n \in \mathcal{I}_t^i,$$

$$u_i^{n+1} = 0, \quad i = \mathcal{I}_x^0, \; n \in \mathcal{I}_t^-,$$

$$u_i^{n+1} = 0, \quad i = \mathcal{I}_x^{-1}, \; n \in \mathcal{I}_t^- .$$

The corresponding implementation becomes

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

# Time loop
```

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

Notice.

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

- $x = 0$: $u = U_0(t)$ or $u_x = 0$

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

- $t = 0$: $u = I(x)$

- $t = 0$: $u_t = I(x)$

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

- A rectangular plug-shaped initial condition. (For $C = 1$ the solution
  will be a rectangle that jumps one cell per time step, making the case
  well suited for verification.)

- A Gaussian function as initial condition.

- A triangular profile as initial condition, which resembles the typical
  initial shape of a guitar string.

- A sinusoidal variation of u at $x = 0$ and either $u = 0$ or $u_x = 0$ at
  $x = L$.

- An exact analytical solution $u(x,t) = \cos(m\pi t/L)\sin(\frac{1}{2}m\pi x/L)$,
  which can be used for convergence rate tests.

———————————

[a]http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn.py

hpl 2: Should include some experiments here or make exercises. Qualitative
behavior of the wave equation can be exemplified.

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 Sections 2.1 and 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: $u = \text{const}$.

```python
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 though, but it requires Dirichlet conditions at both ends.

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

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

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

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

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_{-1}^n and $u_{N_x+1}^n$ are defined in the mesh. Adding the intervals $[-\Delta x, 0]$ and $[L, L+\Delta x]$, often referred to as *ghost cells*, to the mesh gives us all the needed mesh points, corresponding to $i = -1, 0, \ldots, N_x, N_x + 1$. The extra points $i = -1$ and $i = N_x + 1$ are known as *ghost points*, and values at these points, u_{-1}^n and $u_{N_x+1}^n$, are called *ghost values*.

The important idea is to ensure that we always have

$$u_{-1}^n = u_1^n \text{ and } u_{N_x+1}^n = u_{N_x-1}^n,$$

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

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

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

The arrays `u_1` and `u_2` must be defined accordingly.

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

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

instead of $i = 0, \ldots, N_x$ as we have previously used. The ghost points now correspond to $i = 0$ and $i = N_x + 1$. A better solution is to use the ideas of Section 6.4: we hide the specific index value in an index set and operate with inner and boundary points using the index set notation.

To this end, we define u with proper length and Ix to be the corresponding indices for the real physical mesh points $(1, 2, \ldots, N_x + 1)$:

```
u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)
```

That is, the boundary points have indices Ix[0] and Ix[-1] (as before). We first update the solution at all physical mesh points (i.e., interior points in the mesh):

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

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

```
for i in Ix:
    u[i] = u_1[i] + dt*V(x[i-Ix[0]]) + \
           0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
           0.5*dt2*f(x[i-Ix[0]], t[0])
```

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

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

The physical solution to be plotted is now in u[1:-1], or equivalently u[Ix[0]:Ix[-1]+1], so this slice is the quantity to be returned from a solver function. A complete implementation appears in the program wave1D_n0_ghost.py[13].

> **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,

---

[13]http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0_ghost.py

```
    x = linspace(0, L, Nx+1)
```

"Standard coding" of the initial condition,

```
    for i in Ix:
        u_1[i] = I(x[i])
```

becomes wrong, since `u_1` and `x` have different lengths and the index `i` corresponds to two different mesh points. In fact, `x[i]` corresponds to `u[1+i]`. A correct implementation is

```
    for i in Ix:
        u_1[i] = I(x[i-Ix[0]])
```

Similarly, a source term usually coded as `f(x[i], t[n])` is incorrect if `x` is defined to be the physical points, so `x[i]` must be replaced by `x[i-Ix[0]]`.
    An alternative remedy is to let `x` also cover the ghost points such that `u[i]` is the value at `x[i]`.

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

```
u = zeros(Nx+2)
Ix = range(1, u.shape[0])
...
for i in Ix[:-1]:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
           dt2*f(x[i-Ix[0]], t[n])
i = Ix[-1]
u[i] = U_0        # set Dirichlet value
i = Ix[0]
u[i-1] = u[i+1]  # update ghost value
```

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

7   Generalization: variable wave velocity

Our next generalization of the 1D wave equation (1) or (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 5 shows a wave propagating in one medium $[0, 0.7] \cup [0.9, 1]$ with wave velocity c_1 (left) before it enters a second medium $(0.7, 0.9)$ with wave velocity c_2 (right). When the wave passes the boundary where c jumps from c_1 to c_2, a part of the wave is reflected back into the first

59

medium (the *reflected* wave), while one part is transmitted through the second medium (the *transmitted* wave).
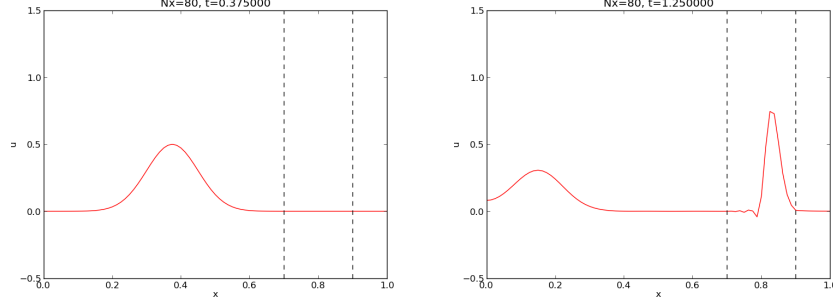

Figure 5: Left: wave entering another medium; right: transmitted and reflected wave.

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

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t). \tag{42}$$

This is the most frequent form of a wave equation with variable wave velocity, but other forms also appear, see Section 14.1 and equation (145).

As usual, we sample (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

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

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:

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

60

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

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right) . \qquad (43)$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_x q D_x u]_i^n . \qquad (44)$$

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 (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

$$D_x q D_x u + D_{2x} q D_{2x} u]_i^n , \qquad (45)$$

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, $[D_x(qD_x u)]_i^n$ with harmonic averaging of q yields a better solution than arithmetic averaging or (45). In the literature, the discretization $[D_x(qD_x u)]_i^n$ totally dominates and very few mention the possibility of (45).

7.3 Computing the coefficient between mesh points

If q is a known function of x, we can easily evaluate $q_{i+\frac{1}{2}}$ simply as $q(x_{i+\frac{1}{2}})$ with $x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$. However, in many cases c, and hence q, is only known as a discrete function, often at the mesh points x_i. Evaluating q between two mesh points x_i and x_{i+1} can then be done by averaging in three ways:

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}\left(q_i + q_{i+1}\right) = [\overline{q}^x]_i \qquad \text{(arithmetic mean)} \qquad (46)$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1} \qquad \text{(harmonic mean)} \qquad (47)$$

$$q_{i+\frac{1}{2}} \approx \left(q_i q_{i+1}\right)^{1/2} \qquad \text{(geometric mean)} \qquad (48)$$

The arithmetic mean in (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.

With the operator notation from (46) we can specify the discretization of the complete variable-coefficient wave equation in a compact way:

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n . \qquad (49)$$

From this notation we immediately see what kind of differences that each term is approximated with. The notation $\overline{q}^x$ also specifies that the variable coefficient is approximated by an arithmetic mean, the definition being $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i + q_{i+1})/2$. With the notation $[D_x q D_x u]_i^n$, we specify that q is evaluated directly, as a function, between the mesh points: $q(x_{i-\frac{1}{2}})$ and $q(x_{i+\frac{1}{2}})$.

Before any implementation, it remains to solve (49) with respect to u_i^{n+1}:

$$
\begin{aligned}
u_i^{n+1} = &-u_i^{n-1} + 2u_i^n + \\
&\left(\frac{\Delta t}{\Delta x}\right)^2 \left(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\right) + \\
&\Delta t^2 f_i^n .
\end{aligned}
\qquad (50)
$$

7.4 How a variable coefficient affects the stability

The stability criterion derived in Section 10.3 reads $\Delta t \le \Delta x/c$. If $c = c(x)$, the criterion will depend on the spatial location. We must therefore choose a Δt that is small enough such that no mesh cell has $\Delta x/c(x) > \Delta t$. That is, we must use the largest c value in the criterion:

$$\Delta t \le \beta \frac{\Delta x}{\max_{x \in [0,L]} c(x)} . \qquad (51)$$

The parameter β is included as a safety factor: in some problems with a significantly varying c it turns out that one must choose $\beta < 1$ to have stable solutions ($\beta = 0.9$ may act as an all-round value).

A different strategy to handle the stability criterion with variable wave velocity is to use a spatially varying Δt. While the idea is mathematically attractive at first sight, the implementation quickly becomes very complicated, so we stick to using a constant Δt and a worst case value of $c(x)$ (with a safety factor β).

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 (50) at the end point $i = N_x$ with $u_{i+1}^n = u_{i-1}^n$ results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n +$$

$$\left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \Delta t^2 f_i^n \tag{52}$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n \tag{53}$$

$$\approx -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_i(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n . \tag{54}$$

Here we used the approximation

$$q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} = q_i + \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2 q}{dx^2}\right)_i \Delta x^2 + \cdots +$$

$$q_i - \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2 q}{dx^2}\right)_i \Delta x^2 + \cdots$$

$$= 2q_i + 2\left(\frac{d^2 q}{dx^2}\right)_i \Delta x^2 + \mathcal{O}(\Delta x^4)$$

$$\approx 2q_i . \tag{55}$$

An alternative derivation may apply the arithmetic mean of q in (53), leading to the term

$$(q_i + \frac{1}{2}(q_{i+1} + q_{i-1}))(u_{i-1}^n - u_i^n) .$$

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

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

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n +$$

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

$$\Delta t^2 f_i^n \tag{56}$$

$$= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta t}{\Delta x}\right)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n . \tag{57}$$

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 (50):

```
for i in range(1, Nx):
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i])  - \
               0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
           dt2*f(x[i], t[n])
```

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

With Neumann conditions $u_x = 0$ at the boundary, we need to combine this scheme with the discrete version of the boundary condition, as shown in Section 7.5. Nevertheless, it would be convenient to reuse the formula for the interior points and just modify the indices `ip1=i+1` and `im1=i-1` as we did in Section 6.3. Assuming $dq/dx = 0$ at the boundaries, we can implement the scheme at the boundary with the following code.

```
i = 0
ip1 = i+1
im1 = ip1
u[i] = - u_2[i] + 2*u_1[i] + \
       C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
       dt2*f(x[i], t[n])
```

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

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

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

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:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t). \tag{58}$$

One example appears when modeling elastic waves in a rod with varying density, cf. (14.1) with $\varrho(x)$.

A natural scheme for (58) is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n. \tag{59}$$

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:

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]_i^n. \tag{60}$$

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):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{61}$$

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

A typical discretization of (61) in terms of centered differences reads

$$[D_t D_t u + b D_{2t} u = c^2 D_x D_x u + f]_i^n. \tag{62}$$

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

$$u_i^{n+1} = (1+\frac{1}{2}b\Delta t)^{-1}((\frac{1}{2}b\Delta t-1)u_i^{n-1}+2u_i^n+C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)+\Delta t^2 f_i^n), \tag{63}$$

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.

8 Building a general 1D wave equation solver

The program `wave1D_dn_vc.py`[14] is a fairly general code for 1D wave propagation problems that targets the following initial-boundary value problem

$$u_{tt} = (c^2(x)u_x)_x + f(x,t), \qquad x \in (0, L), \ t \in (0, T] \qquad (64)$$
$$u(x,0) = I(x), \qquad x \in [0, L] \qquad (65)$$
$$u_t(x,0) = V(t), \qquad x \in [0, L] \qquad (66)$$
$$u(0,t) = U_0(t) \ \text{or} \ u_x(0,t) = 0, \qquad t \in (0, T] \qquad (67)$$
$$u(L,t) = U_L(t) \ \text{or} \ u_x(L,t) = 0, \qquad t \in (0, T] \qquad (68)$$

The only new feature here is the time-dependent Dirichlet conditions. These are trivial to implement:

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

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

The `solver` function is a natural extension of the simplest `solver` function in the initial `wave1D_u0.py` program, extended with Neumann boundary conditions ($u_x = 0$), 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.

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.

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`

---

[14]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

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
```

67

```
                    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__`.

68

The `__call__` method plots the solution on the screen, saves the plot to file, and stores the solution in a file for later retrieval.

More details on storing the solution in files appear in Section **??** in [1].

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 5 shows a typical simulation scenario.

Four types of initial conditions are available:

1. a rectangular pulse (`plug`),

2. a Gaussian function (`gaussian`),

3. a "cosine hat" consisting of one period of the cosine function (`cosinehat`),

4. half a period of a "cosine hat" (`half-cosinehat`)

These peak-shaped initial conditions can be placed in the middle (`loc='center'`) or at the left end (`loc='left'`) of the domain. With the pulse in the middle, it splits in two parts, each with half the initial amplitude, traveling in opposite directions. With the pulse at the left end, centered at $x = 0$, and using the symmetry condition $\partial u/\partial x = 0$, only a right-going pulse is generated. There is also a left-going pulse, but it travels from $x = 0$ in negative x direction and is not visible in the domain $[0, L]$.

The `pulse` function is a flexible tool for playing around with various wave shapes and location of a medium with a different wave velocity.

The code is shown to demonstrate how easy it is to reach this flexibility with the building blocks we have already developed:

```python
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.
```

69

```
        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
```

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(...)'
```

9 Exercises

Exercise 6: Find the analytical solution to a damped wave equation

Consider the wave equation with damping (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 1. It becomes necessary to include a damping term $e^{-\beta t}$ and also have both a sine and cosine component in time:

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

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

Solution. Mathematical model:

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

$b \geq 0$ is a prescribed damping coefficient.
    Ansatz:

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

Boundary condition: $u = 0$ for $x = 0, L$. Fulfilled for $x = 0$. Requirement at $x = L$ gives

$$kL = m\pi,$$

for an arbitrary integer m. Hence, $k = m\pi/L$.

Inserting the ansatz in the PDE and dividing by $e^{-\beta t}$ results in

$$(\beta^2 sinkx - \omega^2 sinkx - b\beta sinkx)(A \cos \omega t + B \sin \omega t)+$$
$$(b\omega sinkx - 2\beta\omega sinkx)(-A \sin \omega t + B \cos \omega t) = -(A \cos \omega t + B \sin \omega t)k^2 c^2$$

This gives us two requirements:

$$\beta^2 - \omega^2 + b\beta + k^2 c^2 = 0$$

and

$$-2\beta\omega + b\omega = 0$$

Since b, c and k are to be given in advance, we may solve these two equations to get

$$\beta = \frac{b}{2}$$
$$\omega = \sqrt{c^2 k^2 - \frac{b^2}{4}}$$

From the initial condition on the derivative, i.e. $\frac{\partial u_e}{\partial t} = 0$, we find that

$$B\omega = \beta A$$

Inserting the expression for ω, we find that

$$B = \frac{b}{2\sqrt{c^2 k^2 - \frac{b^2}{4}}} A$$

for A prescribed.

Using $t = 0$ in the expression for u_e gives us the initial condition as

$$I(x) = A sinkx$$

Summarizing, the PDE problem can then be states as

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2}, \qquad\qquad x \in (0, L), \; t \in (0, T]$$
$$u(x, 0) = I(x), \qquad\qquad\qquad\qquad x \in [0, L]$$
$$\frac{\partial}{\partial t}u(x, 0) = 0, \qquad\qquad\qquad\qquad x \in [0, L]$$
$$u(0, t) = 0, \qquad\qquad\qquad\qquad\qquad t \in (0, T]$$
$$u(L, t) = 0, \qquad\qquad\qquad\qquad\qquad t \in (0, T]$$

where constants c, A, b and k, as well as $I(x)$, are prescribed.

The solution to the problem is then given as

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

with $k = m\pi/L$ for arbitrary integer m, $\beta = \frac{b}{2}$, $\omega = \sqrt{c^2 k^2 - \frac{b^2}{4}}$, $B = \frac{b}{2\sqrt{c^2 k^2 - \frac{b^2}{4}}}A$ and $I(x) = A\sin kx$.

Filename: `damped_waves`.

Problem 7: Explore symmetry boundary conditions

Consider the simple "plug" wave where $\Omega = [-L, L]$ and

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

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

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

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

Solution. A symmetric u around $x = 0$ means that $u(-x, t) = u(x, t)$. Let $x_0 = 0$ and $x = x_0 + h$. Then we can use a *centered* finite difference definition of the derivative:

$$\frac{\partial}{\partial x}u(x_0, t) = \lim_{h\to 0}\frac{u(x_0 + h, t) - u(x_0 - h)}{2h} = \lim_{h\to 0}\frac{u(h, t) - u(-h, t)}{2h} = 0,$$

since $u(h, t) = u(-h, t)$ for any h. Symmetry around a point $x = x_0$ therefore always implies $u_x(x_0, t) = 0$.

73

b) Perform simulations of the complete wave problem on $[-L, L]$. Thereafter, utilize the symmetry of the solution and run a simulation in half of the domain $[0, L]$, using a boundary condition at $x = 0$. Compare plots from the two solutions and confirm that they are the same.

Solution. We can utilize the `wave1D_dn.py` code which allows Dirichlet and Neumann conditions. The `solver` and `viz` functions must take x_0 and x_L as parameters instead of just L such that we can solve the wave equation in $[x_0, x_L]$. The we can call up `solver` for the two problems on $[-L, L]$ and $[0, L]$ with boundary conditions $u(-L, t) = u(L, t) = 0$ and $u_x(0, t) = u(L, t) = 0$, respectively.

The original `wave1D_dn.py` code makes a movie by playing all the `.png` files in a browser. It can then be wise to let the `viz` function create a movie directory and place all the frames and HTML player file in that directory. Alternatively, one can just make some ordinary movie file (Ogg, WebM, MP4, Flash) with `avconv` or `ffmpeg` and give it a name. It is a point that the name is transferred to `viz` so it is easy to call `viz` twice and get two separate movie files or movie directories.

The plots produced by the code (below) shows that the solutions indeed are the same.

c) Prove the symmetry property of the solution by setting up the complete initial-boundary value problem and showing that if $u(x, t)$ is a solution, then also $u(-x, t)$ is a solution.

Solution. The plan in this proof is to introduce $v(x, t) = u(-x, t)$ and show that v fulfills the same initial-boundary value problem as u. If the problem has a unique solution, then $v = u$. Or, in other words, the solution is symmetric: $u(-x, t) = u(x, t)$.

We can work with a general initial-boundary value problem on the form

$$u_t t(x, t) = c^2 u_{xx}(x, t) + f(x, t) \tag{69}$$

$$u(x, 0) = I(x) \tag{70}$$

$$u_t(x, 0) = V(x) \tag{71}$$

$$u(-L, 0) = 0 \tag{72}$$

$$u(L, 0) = 0 \tag{73}$$

Introduce a new coordinate $\bar{x} = -x$. We have that

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \bar{x}}\frac{\partial \bar{x}}{\partial x}\right) = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \bar{x}}(-1)\right) = (-1)^2\frac{\partial^2 u}{\partial \bar{x}^2}$$

The derivatives in time are unchanged.

Substituting x by $-\bar{x}$ leads to

$$u_{tt}(-\bar{x}, t) = c^2 u_{\bar{x}\bar{x}}(-\bar{x}, t) + f(-\bar{x}, t) \tag{74}$$
$$u(-\bar{x}, 0) = I(-\bar{x}) \tag{75}$$
$$u_t(-\bar{x}, 0) = V(-\bar{x}) \tag{76}$$
$$u(L, 0) = 0 \tag{77}$$
$$u(-L, 0) = 0 \tag{78}$$

Now, dropping the bars and introducing $v(x, t) = u(-x, t)$, we find that

$$v_{tt}(x, t) = c^2 v_{xx}(x, t) + f(-x, t) \tag{79}$$
$$v(x, 0) = I(-x) \tag{80}$$
$$v_t(x, 0) = V(-x) \tag{81}$$
$$v(-L, 0) = 0 \tag{82}$$
$$v(L, 0) = 0 \tag{83}$$

Provided that I, f, and V are all symmetric around $x = 0$ such that $I(x) = I(-x)$, $V(x) = V(-x)$, and $f(x, t) = f(-x, t)$, we can express the initial-boundary value problem as

$$v_{tt}(x, t) = c^2 v_{xx}(x, t) + f(x, t) \tag{84}$$
$$v(x, 0) = I(x) \tag{85}$$
$$v_t(x, 0) = V(x) \tag{86}$$
$$v(-L, 0) = 0 \tag{87}$$
$$v(L, 0) = 0 \tag{88}$$

This is the same problem as the one that u fulfills. If the solution is unique, which can be proven, then $v = u$, and $u(-x, t) = u(x, t)$.

To summarize, the necessary conditions for symmetry are that

- all involved functions I, V, and f must be symmetric, and

- the boundary conditions are symmetric in the sense that they can be flipped (the condition at $x = -L$ can be applied at $x = L$ and vice versa).

d) If the code works correctly, the solution $u(x, t) = x(L - x)(1 + \frac{t}{2})$ should be reproduced exactly. Write a test function `test_quadratic` that checks whether this is the case. Simulate for x in $[0, \frac{L}{2}]$ with a symmetry condition at the end $x = \frac{L}{2}$.

Solution. Running the code below, shows that the test case indeed is reproduced exactly.

75

```python
#!/usr/bin/env python
from scitools.std import *

# Add an x0 coordinate for solving the wave equation on [x0, xL]

def solver(I, V, f, c, U_0, U_L, x0, xL, Nx, C, T,
           user_action=None, version='scalar'):
    """
    Solve u_tt=c^2*u_xx + f on (0,L)x(0,T].
    u(0,t)=U_0(t) or du/dn=0 (U_0=None), u(L,t)=U_L(t) or du/dn=0 (u_L=None).
    """
    x = linspace(x0, xL, Nx+1)       # Mesh points in space
    dx = x[1] - x[0]
    dt = C*dx/c
    Nt = int(round(T/dt))
    t = linspace(0, Nt*dt, Nt+1)     # Mesh points in time
    C2 = C**2; dt2 = dt*dt           # Help variables in the scheme

    # Wrap user-given f, V, U_0, U_L
    if f is None or f == 0:
        f = (lambda x, t: 0) if version == 'scalar' else \
            lambda x, t: zeros(x.shape)
    if V is None or V == 0:
        V = (lambda x: 0) if version == 'scalar' else \
            lambda x: zeros(x.shape)
    if U_0 is not None:
        if isinstance(U_0, (float,int)) and U_0 == 0:
            U_0 = lambda t: 0
    if U_L is not None:
        if isinstance(U_L, (float,int)) and U_L == 0:
            U_L = lambda t: 0

    u   = zeros(Nx+1)    # Solution array at new time level
    u_1 = zeros(Nx+1)    # Solution at 1 time level back
    u_2 = zeros(Nx+1)    # Solution at 2 time levels back

    Ix = range(0, Nx+1)
    It = range(0, Nt+1)

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

    # Load initial condition into u_1
    for i in Ix:
        u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, 0)

    # Special formula for the first step
    for i in Ix[1:-1]:
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               0.5*dt2*f(x[i], t[0])

    i = Ix[0]
    if U_0 is None:
        # Set boundary values du/dn = 0
        # x=0: i-1 -> i+1 since u[i-1]=u[i+1]
        # x=L: i+1 -> i-1 since u[i+1]=u[i-1])
        ip1 = i+1
        im1 = ip1  # i-1 -> i+1
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[im1] - 2*u_1[i] + u_1[ip1]) + \
```

```
                    0.5*dt2*f(x[i], t[0])
        else:
            u[0] = U_0(dt)

        i = Ix[-1]
        if U_L is None:
            im1 = i-1
            ip1 = im1   # i+1 -> i-1
            u[i] = u_1[i] + dt*V(x[i]) + \
                    0.5*C2*(u_1[im1] - 2*u_1[i] + u_1[ip1]) + \
                    0.5*dt2*f(x[i], t[0])
        else:
            u[i] = U_L(dt)

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

        # Update data structures for next step
        u_2[:], u_1[:] = u_1, u

        for n in It[1:-1]:
            # Update all inner points
            if version == 'scalar':
                for i in Ix[1:-1]:
                    u[i] = - u_2[i] + 2*u_1[i] + \
                            C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                            dt2*f(x[i], t[n])

            elif version == 'vectorized':
                u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
                            C2*(u_1[0:-2] - 2*u_1[1:-1] + u_1[2:]) + \
                            dt2*f(x[1:-1], t[n])
            else:
                raise ValueError('version=%s' % version)

            # Insert boundary conditions
            i = Ix[0]
            if U_0 is None:
                # Set boundary values
                # x=0: i-1 -> i+1 since u[i-1]=u[i+1] when du/dn=0
                # x=L: i+1 -> i-1 since u[i+1]=u[i-1] when du/dn=0
                ip1 = i+1
                im1 = ip1
                u[i] = - u_2[i] + 2*u_1[i] + \
                        C2*(u_1[im1] - 2*u_1[i] + u_1[ip1]) + \
                        dt2*f(x[i], t[n])
            else:
                u[0] = U_0(t[n+1])

            i = Ix[-1]
            if U_L is None:
                im1 = i-1
                ip1 = im1
                u[i] = - u_2[i] + 2*u_1[i] + \
                        C2*(u_1[im1] - 2*u_1[i] + u_1[ip1]) + \
                        dt2*f(x[i], t[n])
            else:
                u[i] = U_L(t[n+1])

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

            # Update data structures for next step
```

77

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

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


def viz(I, V, f, c, U_0, U_L, x0, xL, Nx, C, T, umin, umax,
        version='scalar', animate=True,
        movie_dir='tmp'):
    """Run solver and visualize u at each time level."""
    import scitools.std as plt, time, glob, os

    def plot_u(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[x0, xL, umin, umax],
                 title='t=%f' % t[n])
        # 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

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

    user_action = plot_u if animate else None
    u, x, t, cpu = solver(I, V, f, c, U_0, U_L, L, Nx, C, T,
                          user_action, version)
    if animate:
        # Make a directory with the frames
        if os.path.isdir(movie_dir):
            shutil.rmtree(movie_dir)
        os.mkdir(movie_dir)
        os.chdir(movie_dir)
        # Move all frame_*.png files to this subdirectory
        for filename in glob.glob(os.path.join(os.pardir, 'frame_*.png')):
            os.renamve(os.path.join(os.pardir, filename), filename)
        plt.movie('frame_*.png', encoder='html', fps=4,
                  output_file='movie.html')
        # Invoke movie.html in a browser to steer the movie

    return cpu

import nose.tools as nt

def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    We simulate in [0, L/2] and apply a symmetry condition
    at the end x=L/2.
    """
    exact_solution = lambda x, t: x*(L-x)*(1+0.5*t)
    I = lambda x: exact_solution(x, 0)
    V = lambda x: 0.5*exact_solution(x, 0)
    f = lambda x, t: 2*(1+0.5*t)*c**2
    U_0 = lambda t: exact_solution(0, t)
    U_L = None
    L = 2.5
    c = 1.5
    Nx = 3  # very coarse mesh
    C = 1
```

```
        T = 18  # long time integration

        def assert_no_error(u, x, t, n):
            u_e = exact_solution(x, t[n])
            diff = abs(u - u_e).max()
            nt.assert_almost_equal(diff, 0, places=13)

        solver(I, V, f, c, U_0, U_L, 0, L/2, Nx, C, T,
               user_action=assert_no_error, version='scalar')
        solver(I, V, f, c, U_0, U_L, 0, L/2, Nx, C, T,
               user_action=assert_no_error, version='vectorized')


def plug(C=1, Nx=50, animate=True, version='scalar', T=2):
    """Plug profile as initial condition."""
    L = 1.
    c = 1
    delta = 0.1

    def I(x):
        if abs(x) > delta:
            return 0
        else:
            return 1

    # Solution on [-L,L]
    cpu = viz(I=I, V=0, f=0, c, U_0=0, U_L=0, -L, L, 2*Nx, C, T,
              umin=-1.1, umax=1.1, version=version, animate=animate,
              movie_dir='full')

    # Solution on [0,L]
    cpu = viz(I=I, V=0, f=0, c, U_0=None, U_L=0, 0, L, Nx, C, T,
              umin=-1.1, umax=1.1, version=version, animate=animate,
              movie_dir='half')


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

Filename: `wave1D_symmetric`.

Exercise 8: Send pulse waves through a layered medium

Use the `pulse` function in `wave1D_dn_vc.py` to investigate sending a pulse, located with its peak at $x = 0$, through two media with different wave velocities. The (scaled) velocity in the left medium is 1 while it is $\frac{1}{s_f}$ in the right medium. Report what happens with a Gaussian pulse, a "cosine hat" pulse, half a "cosine hat" pulse, and a plug pulse for resolutions $N_x = 40, 80, 160$, and $s_f = 2, 4$. Simulate until $T = 2$.

Solution. In all cases, the change in velocity causes some of the wave to be reflected back (while the rest is let through). When the waves go from higher to lower velocity, the amplitude builds, and vice versa.

```
import wave1D_dn_vc as wave
import os, sys, shutil, glob
```

```
for pulse_tp in 'gaussian', 'cosinehat', 'half-cosinehat', 'plug':
    for Nx in 40, 80, 160:
        for sf in 2, 4:
            if sf == 1 and Nx > 40:
                continue  # homogeneous medium with C=1: Nx=40 enough
            print 'wave1D.pulse:', pulse_tp, Nx, sf

            wave.pulse(C=1, Nx=Nx, animate=False, # just hardcopies
                       version='vectorized',
                       T=2, loc='left', pulse_tp=pulse_tp,
                       slowness_factor=sf, medium=[0.7, 0.9],
                       skip_frame = 1,
                       sigma=0.05)
```

Filename: `pulse1D`.

Exercise 9: Explain why numerical noise occurs

The experiments performed in Exercise 8 shows considerable numerical noise in the form of non-physical waves, especially for $s_f = 4$ and the plug pulse or the half a "cosinehat" pulse. The noise is much less visible for a Gaussian pulse. Run the case with the plug and half a "cosinehat" pulses for $s_f = 1$, $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion relation to explain the observations. Filename: `pulse1D_analysis`.

Exercise 10: Investigate harmonic averaging in a 1D model

Harmonic means are often used if the wave velocity is non-smooth or discontinuous. Will harmonic averaging of the wave velocity give less numerical noise for the case $s_f = 4$ in Exercise 8? Filename: `pulse1D_harmonic`.

Problem 11: Implement open boundary conditions

To enable a wave to leave the computational domain and travel undisturbed through the boundary $x = L$, one can in a one-dimensional problem impose the following condition, called a *radiation condition* or *open boundary condition*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \,. \tag{89}$$

The parameter c is the wave velocity.

Show that (89) accepts a solution $u = g_R(x - ct)$ (right-going wave), but not $u = g_L(x + ct)$ (left-going wave). This means that (89) 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

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0 \,. \tag{90}$$

a) A natural idea for discretizing the condition (89) at the spatial end point $i = N_x$ is to apply centered differences in time and space:

$$[D_{2t}u + cD_{2x}u = 0]_i^n, \quad i = N_x. \tag{91}$$

Eliminate the fictitious value $u_{N_x+1}^n$ by using the discrete equation at the same point.

The equation for the first step, u_i^1, is in principle also affected, but we can then use the condition $u_{N_x} = 0$ since the wave has not yet reached the right boundary.

b) A much more convenient implementation of the open boundary condition at $x = L$ can be based on an explicit discretization

$$[D_t^+ u + cD_x^- u = 0]_i^n, \quad i = N_x. \tag{92}$$

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 (89) at $x = L$, and an initial disturbance in the middle of the domain, e.g., a plug profile like

$$u(x, 0) = \begin{cases} 1, & L/2 - \ell \le x \le L/2 + \ell, \\ 0, \text{otherwise} \end{cases}$$

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`[15].

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

$$[D_t^+ u - cD_x^+ u = 0]_i^n, \quad i = 0, \tag{93}$$

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:

---

[15]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn.py`

1. Disturbance in the middle of the domain, $I(x) = g(x; L/2, s)$, and open boundary condition at the left end.

2. Disturbance at the left end, $I(x) = g(x; 0, s)$, and $u_x = 0$ as symmetry boundary condition at this end.

Make test functions for both cases, testing that the solution is zero after the waves have left the domain.

d) In 2D and 3D it is difficult to compute the correct wave velocity normal to the boundary, which is needed in generalizations of the open boundary conditions in higher dimensions. Test the effect of having a slightly wrong wave velocity in (92). Make movies to illustrate what happens.
Filename: `wave1D_open_BC`.

Remarks. The condition (89) 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 12: Implement periodic boundary conditions

It is frequently of interest to follow wave motion over large distances and long times. A straightforward approach is to work with a very large domain, but 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 11) to let a right-going wave out of the domain. At the left end, $x = 0$, we apply, in the beginning of the simulation, either a symmetry boundary condition (see Exercise 7) $u_x = 0$, 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 11. Implement the described type of boundary conditions and test them on two different initial shapes: a plug $u(x, 0) = 1$ for $x \leq 0.1$, $u(x, 0) = 0$ for $x > 0.1$, and a Gaussian function in the middle of the domain:

$u(x, 0) = \exp\left(-\frac{1}{2}(x - 0.5)^2/0.05\right)$. The domain is the unit interval $[0, 1]$. Run these two shapes for Courant numbers 1 and 0.5. Assume constant wave velocity. Make movies of the four cases. Reason why the solutions are correct. Filename: `periodic`.

Exercise 13: Compare discretizations of a Neumann condition

We have a 1D wave equation with variable wave velocity: $u_{tt} = (qu_x)_x$. A Neumann condition u_x at $x = 0, L$ can be discretized as shown in (54) and (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 (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 (57). Now, $q_{i-1/2}$ is a 2nd-order approximation to q_i, $q_{i-1/2} = q_i + 0.25q_i''\Delta x^2 + \cdots$, because $q_i' = 0$ for $i = N_x$ (a similar argument can be applied to the case $i = 0$).

c) A third discretization can be based on a simple and convenient, but less accurate, one-sided difference: $u_i - u_{i-1} = 0$ at $i = N_x$ and $u_{i+1} - u_i = 0$ at $i = 0$. Derive the resulting scheme in detail and implement it. Run experiments with q from a) or b) to establish the rate of convergence of the scheme.

d) A fourth technique is to view the scheme as

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

and place the boundary at $x_{i+\frac{1}{2}}$, $i = N_x$, instead of exactly at the physical boundary. With this idea of approximating (moving) the boundary, we can just set $[qD_x u]_{i+\frac{1}{2}}^n = 0$. Derive the complete scheme using this technique. The implementation of the boundary condition at $L - \Delta x/2$ is $\mathcal{O}(\Delta x^2)$ accurate, but the interesting question is what impact the movement of the boundary has on the convergence rate. Compute the errors as usual over the entire mesh and use q from a) or b).
Filename: `Neumann_discr`.

Exercise 14: Verification by a cubic polynomial in space

The purpose of this exercise is to verify the implementation of the `solver` function in the program `wave1D_n0.py`[16] by using an exact numerical solution

---

[16]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_n0.py`

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`[17], 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

$$[D_{2x}u]_i^n = u_x(x_i, t_n) + \frac{1}{6}u_{xxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4). \tag{94}$$

At the boundary 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

---

[17]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_u0.py`

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`.

10   Analysis of the difference equations

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

$$u(x,t) = g_R(x - ct) + g_L(x + ct), \tag{95}$$

for any functions g_R and g_L sufficiently smooth to be differentiated twice. The result follows from inserting (95) in the wave equation. A function of the form $g_R(x - ct)$ represents a signal moving to the right in time with constant velocity c. This feature can be explained as follows. At time $t = 0$ the signal looks like $g_R(x)$. Introducing a moving horizontal coordinate $\xi = x - ct$, we see the function $g_R(\xi)$ is "at rest" in the ξ coordinate system, and the shape is always the same. Say the $g_R(\xi)$ function has a peak at $\xi = 0$. This peak is located at $x = ct$, which means that it moves with the velocity $dx/dt = c$ in the x coordinate system. Similarly, $g_L(x + ct)$ is a function, initially with shape $g_L(x)$, that moves in the negative x direction with constant velocity c (introduce $\xi = x + ct$, look at the point $\xi = 0$, $x = -ct$, which has velocity $dx/dt = -c$).

With the particular initial conditions

$$u(x,0) = I(x), \quad \frac{\partial}{\partial t} u(x,0) = 0,$$

we get, with u as in (95),

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

The former suggests $g_R = g_L$, and the former then leads to $g_R = g_L = I/2$. Consequently,

$$u(x,t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct) \,. \tag{96}$$

The interpretation of (96) is that the initial shape of u is split into two parts, each with the same shape as I but half of the initial amplitude. One part is traveling to the left and the other one to the right.

The solution has two important physical features: constant amplitude of the left and right wave, and constant velocity of these two waves. It turns out that the numerical solution will also preserve the constant amplitude, but the velocity depends on the mesh parameters Δt and Δx.

The solution (96) will be influenced by boundary conditions when the parts $\frac{1}{2}I(x - ct)$ and $\frac{1}{2}I(x + ct)$ hit the boundaries and get, e.g., reflected back into the domain. However, when $I(x)$ is nonzero only in a small part in the middle of the spatial domain $[0, L]$, which means that the boundaries are placed far away from the initial disturbance of u, the solution (96) is very clearly observed in a simulation.

A useful representation of solutions of wave equations is a linear combination of sine and/or cosine waves. Such a sum of waves is a solution if the governing PDE is linear and each sine or cosine wave fulfills the equation. To ease analytical calculations by hand we shall work with complex exponential functions instead of real-valued sine or cosine functions. The real part of complex expressions will typically be taken as the physical relevant quantity (whenever a physical relevant quantity is strictly needed). The idea now is to build $I(x)$ of complex wave components e^{ikx}:

$$I(x) \approx \sum_{k \in K} b_k e^{ikx} \, . \tag{97}$$

Here, k is the frequency of a component, K is some set of all the discrete k values needed to approximate $I(x)$ well, and b_k are constants that must be determined. We will very seldom need to compute the b_k coefficients: most of the insight we look for, and the understanding of the numerical methods we want to establish, come from investigating how the PDE and the scheme treat a single component e^{ikx} wave.

Letting the number of k values in K tend to infinity, makes the sum (97) converge to $I(x)$. This sum is known as a *Fourier series* representation of $I(x)$. Looking at (96), we see that the solution $u(x, t)$, when $I(x)$ is represented as in (97), is also built of basic complex exponential wave components of the form $e^{ik(x \pm ct)}$ according to

$$u(x, t) = \frac{1}{2} \sum_{k \in K} b_k e^{ik(x - ct)} + \frac{1}{2} \sum_{k \in K} b_k e^{ik(x + ct)} \, . \tag{98}$$

It is common to introduce the frequency in time $\omega = kc$ and assume that $u(x, t)$ is a sum of basic wave components written as $e^{ikx - \omega t}$. (Observe that inserting such a wave component in the governing PDE reveals that $\omega^2 = k^2 c^2$, or $\omega = \pm kc$, reflecting the two solutions: one $(+kc)$ traveling to the right and the other $(-kc)$ traveling to the left.)

10.2 More precise definition of Fourier representations

The above introduction to function representation by sine and cosine waves was quick and intuitive, but will suffice as background knowledge for the following material of single wave component analysis. However, to understand all details of how different wave components sum up to the analytical and numerical solutions, a more precise mathematical treatment is helpful and therefore summarized below.

It is well known that periodic functions can be represented by Fourier series. A generalization of the Fourier series idea to non-periodic functions defined on the real line is the *Fourier transform*:

$$I(x) = \int_{-\infty}^{\infty} A(k)e^{ikx}dk, \tag{99}$$

$$A(k) = \int_{-\infty}^{\infty} I(x)e^{-ikx}dx. \tag{100}$$

The function $A(k)$ reflects the weight of each wave component e^{ikx} in an infinite sum of such wave components. That is, $A(k)$ reflects the frequency content in the function $I(x)$. Fourier transforms are particularly fundamental for analyzing and understanding time-varying signals.

The solution of the linear 1D wave PDE can be expressed as

$$u(x,t) = \int_{-\infty}^{\infty} A(k)e^{i(kx-\omega(k)t)}dx.$$

In a finite difference method, we represent u by a mesh function u_q^n, where n counts temporal mesh points and q counts the spatial ones (the usual counter for spatial points, i, is here already used as imaginary unit). Similarly, $I(x)$ is approximated by the mesh function I_q, $q = 0, \ldots, N_x$. On a mesh, it does not make sense to work with wave components e^{ikx} for very large k, because the shortest possible sine or cosine wave that can be represented uniquely on a mesh with spacing Δx is the wave with wavelength $2\Delta x$. This wave has its peaks and throughs at every two mesh points. That is, the wave "jumps up and down" between the mesh points.

The corresponding k value for the shortest possible wave in the mesh is $k = 2\pi/(2\Delta x) = \pi/\Delta x$. This maximum frequency is known as the *Nyquist frequency*. Within the range of relevant frequencies $(0, \pi/\Delta x]$ one defines the discrete Fourier transform[18], using $N_x + 1$ discrete frequencies:

$$I_q = \frac{1}{N_x + 1} \sum_{k=0}^{N_x} A_k e^{i2\pi kj/(N_x+1)}, \quad i = 0, \ldots, N_x, \tag{101}$$

$$A_k = \sum_{q=0}^{N_x} I_q e^{-i2\pi kq/(N_x+1)}, \quad k = 0, \ldots, N_x + 1. \tag{102}$$

The A_k values represent the discrete Fourier transform of the I_q values, which themselves are the inverse discrete Fourier transform of the A_k values.

The discrete Fourier transform is efficiently computed by the *Fast Fourier transform* algorithm. For a real function $I(x)$, the relevant Python code for computing and plotting the discrete Fourier transform appears in the example below.

---

[18]http://en.wikipedia.org/wiki/Discrete_Fourier_transform

```
import numpy as np
from numpy import sin, pi

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

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

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

# Compute the corresponding frequencies
freqs = np.linspace(0, pi/dx, A_amplitude.size)

import matplotlib.pyplot as plt
plt.plot(freqs, A_amplitude)
plt.show()
```

10.3   Stability

The scheme

$$[D_t D_t u = c^2 D_x D_x u]_q^n \tag{103}$$

for the wave equation $u_{tt} = c^2 u_{xx}$ allows basic wave components

$$u_q^n = e^{i(kx_q - \tilde{\omega}t_n)}$$

as solution, but it turns out that the frequency in time, $\tilde{\omega}$, is not equal to the exact frequency $\omega = kc$. The goal now is to find exactly what $\tilde{\omega}$ is. We ask two key questions:

- How accurate is $\tilde{\omega}$ compared to ω?

- Does the amplitude of such a wave component preserve its (unit) amplitude, as it should, or does it get amplified or damped in time (because of a complex $\tilde{\omega}$)?

The following analysis will answer these questions. We shall continue using q as an identifier for a certain mesh point in the x direction.

Preliminary results.   A key result needed in the investigations is the finite difference approximation of a second-order derivative acting on a complex wave component:

$$[D_t D_t e^{i\omega t}]^n = -\frac{4}{\Delta t^2} \sin^2\left(\frac{\omega \Delta t}{2}\right) e^{i\omega n \Delta t} .$$

By just changing symbols ($\omega \to k$, $t \to x$, $n \to q$) it follows that

$$[D_x D_x e^{ikx}]_q = -\frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x}.$$

Numerical wave propagation. Inserting a basic wave component $u_q^n = e^{i(kx_q - \tilde{\omega} t_n)}$ in (103) results in the need to evaluate two expressions:

$$[D_t D_t e^{ikx} e^{-i\tilde{\omega} t}]_q^n = [D_t D_t e^{-i\tilde{\omega} t}]^n e^{ikq\Delta x}$$

$$= -\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) e^{-i\tilde{\omega} n\Delta t} e^{ikq\Delta x} \quad (104)$$

$$[D_x D_x e^{ikx} e^{-i\tilde{\omega} t}]_q^n = [D_x D_x e^{ikx}]_q e^{-i\tilde{\omega} n\Delta t}$$

$$= -\frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x} e^{-i\tilde{\omega} n\Delta t}. \quad (105)$$

Then the complete scheme,

$$[D_t D_t e^{ikx} e^{-i\tilde{\omega} t} = c^2 D_x D_x e^{ikx} e^{-i\tilde{\omega} t}]_q^n$$

leads to the following equation for the unknown numerical frequency $\tilde{\omega}$ (after dividing by $-e^{ikx} e^{-i\tilde{\omega} t}$):

$$\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2 \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right),$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2 \sin^2\left(\frac{k\Delta x}{2}\right), \quad (106)$$

where

$$C = \frac{c\Delta t}{\Delta x} \quad (107)$$

is the Courant number. Taking the square root of (106) yields

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C \sin\left(\frac{k\Delta x}{2}\right), \quad (108)$$

Since the exact ω is real it is reasonable to look for a real solution $\tilde{\omega}$ of (108). The right-hand side of (108) must then be in $[-1, 1]$ because the sine function on the left-hand side has values in $[-1, 1]$ for real $\tilde{\omega}$. The sine function on the right-hand side can attain the value 1 when

$$\frac{k\Delta x}{2} = m\frac{\pi}{2}, \quad m \in \mathbb{Z}.$$

With $m = 1$ we have $k\Delta x = \pi$, which means that the wavelength $\lambda = 2\pi/k$ becomes $2\Delta x$. This is the absolutely shortest wavelength that can be represented

on the mesh: the wave jumps up and down between each mesh point. Larger values of $|m|$ are irrelevant since these correspond to k values whose waves are too short to be represented on a mesh with spacing Δx. For the shortest possible wave in the mesh, $\sin(k\Delta x/2) = 1$, and we must require

$$C \le 1. \tag{109}$$

Consider a right-hand side in (108) of magnitude larger than unity. The solution $\tilde{\omega}$ of (108) must then be a complex number $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$ because the sine function is larger than unity for a complex argument. One can show that for any ω_i there will also be a corresponding solution with $-\omega_i$. The component with $\omega_i > 0$ gives an amplification factor $e^{\omega_i t}$ that grows exponentially in time. We cannot allow this and must therefore require $C \le 1$ as a *stability criterion*.

Remark on the stability requirement.

For smoother wave components with longer wave lengths per length Δx, (109) can in theory be relaxed. However, small round-off errors are always present in a numerical solution and these vary arbitrarily from mesh point to mesh point and can be viewed as unavoidable noise with wavelength $2\Delta x$. As explained, $C > 1$ will for this very small noise lead to exponential growth of the shortest possible wave component in the mesh. This noise will therefore grow with time and destroy the whole solution.

10.4 Numerical dispersion relation

Equation (108) can be solved with respect to $\tilde{\omega}$:

$$\tilde{\omega} = \frac{2}{\Delta t} \sin^{-1}\left(C \sin\left(\frac{k\Delta x}{2} \right) \right). \tag{110}$$

The relation between the numerical frequency $\tilde{\omega}$ and the other parameters k, c, Δx, and Δt is called a *numerical dispersion relation*. Correspondingly, $\omega = kc$ is the *analytical dispersion relation*. In general, dispersion refers to the phenomenon where the wave velocity depends on the spatial frequency (k, or the wave length $\lambda = 2\pi/k$) of the wave. Since the wave velocity is $\omega/k = c$, we realize that the analytical dispersion relation reflects the fact that there is no dispersion. However, in a numerical scheme we have dispersive waves where the wave velocity depends on k.

The special case $C = 1$ deserves attention since then the right-hand side of (110) reduces to

$$\frac{2}{\Delta t} \frac{k\Delta x}{2} = \frac{1}{\Delta t} \frac{\omega\Delta x}{c} = \frac{\omega}{C} = \omega.$$

That is, $\tilde{\omega} = \omega$ and the numerical solution is exact at all mesh points regardless of Δx and Δt! This implies that the numerical solution method is also an analytical

solution method, at least for computing u at discrete points (the numerical method says nothing about the variation of u *between* the mesh points, and employing the common linear interpolation for extending the discrete solution gives a curve that in general deviates from the exact one).

For a closer examination of the error in the numerical dispersion relation when $C < 1$, we can study $\tilde{\omega} - \omega$, $\tilde{\omega}/\omega$, or the similar error measures in wave velocity: $\tilde{c} - c$ and $\tilde{c}/c$, where $c = \omega/k$ and $\tilde{c} = \tilde{\omega}/k$. It appears that the most convenient expression to work with is $\tilde{c}/c$, since it can be written as a function of just two parameters:

$$\frac{\tilde{c}}{c} = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right),$$

with $p = k\Delta x/2$ as a non-dimensional measure of the spatial frequency. In essence, p tells how many spatial mesh points we have per wave length in space for the wave component with frequency k (recall that the wave length is $2\pi/k$). That is, p reflects how well the spatial variation of the wave component is resolved in the mesh. Wave components with wave length less than $2\Delta x$ ($2\pi/k < 2\Delta x$) are not visible in the mesh, so it does not make sense to have $p > \pi/2$.

We may introduce the function $r(C, p) = \tilde{c}/c$ for further investigation of numerical errors in the wave velocity:

$$r(C,p) = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right), \quad C \in (0,1], \ p \in (0, \pi/2]. \tag{111}$$

This function is very well suited for plotting since it combines several parameters in the problem into a dependence on two dimensionless numbers, C and p.

Defining

```
def r(C, p):
    return 2/(C*p)*asin(C*sin(p))
```

we can plot $r(C, p)$ as a function of p for various values of C, see Figure 6. Note that the shortest waves have the most erroneous velocity, and that short waves move more slowly than they should.

We can also easily make a Taylor series expansion in the discretization parameter p:

```
>>> import sympy as sym
>>> C, p = sym.symbols('C p')
>>> # Compute the 7 first terms around p=0 with no O() term
>>> rs = r(C, p).series(p, 0, 7).removeO()
>>> rs
p**6*(5*C**6/112 - C**4/16 + 13*C**2/720 - 1/5040) +
p**4*(3*C**4/40 - C**2/12 + 1/120) +
p**2*(C**2/6 - 1/6) + 1

>>> # Pick out the leading order term, but drop the constant 1
>>> rs_error_leading_order = (rs - 1).extract_leading_order(p)
>>> rs_error_leading_order
p**2*(C**2/6 - 1/6)
```
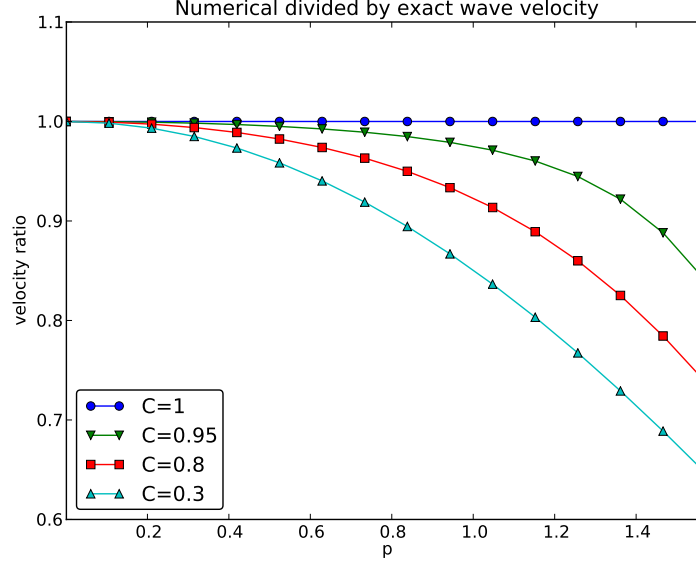
Figure 6: The fractional error in the wave velocity for different Courant numbers.

```
>>> # Turn the series expansion into a Python function
>>> rs_pyfunc = lambdify([C, p], rs, modules='numpy')

>>> # Check: rs_pyfunc is exact (=1) for C=1
>>> rs_pyfunc(1, 0.1)
1.0
```

Note that without the `.removeO()` call the series gets an `O(x**7)` term that makes it impossible to convert the series to a Python function (for, e.g., plotting).

From the `rs_error_leading_order` expression above, we see that the leading order term in the error of this series expansion is

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2(C^2 - 1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right), \tag{112}$$

pointing to an error $\mathcal{O}(\Delta t^2, \Delta x^2)$, which is compatible with the errors in the difference approximations ($D_t D_t u$ and $D_x D_x u$).

We can do more with a series expansion, e.g., factor it to see how the factor $C - 1$ plays a significant role. To this end, we make a list of the terms, factor each term, and then sum the terms:

```
>>> rs = r(C, p).series(p, 0, 4).removeO().as_ordered_terms()
>>> rs
[1, C**2*p**2/6 - p**2/6,
 3*C**4*p**4/40 - C**2*p**4/12 + p**4/120,
```

94

```
   5*C**6*p**6/112 - C**4*p**6/16 + 13*C**2*p**6/720 - p**6/5040]
>>> rs = [factor(t) for t in rs]
>>> rs
[1, p**2*(C - 1)*(C + 1)/6,
 p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120,
 p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040]
>>> rs = sum(rs)   # Python's sum function sums the list
>>> rs
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

We see from the last expression that $C = 1$ makes all the terms in `rs` vanish. Since we already know that the numerical solution is exact for $C = 1$, the remaining terms in the Taylor series expansion will also contain factors of $C - 1$ and cancel for $C = 1$.

10.5   Extending the analysis to 2D and 3D

The typical analytical solution of a 2D wave equation

$$u_{tt} = c^2(u_{xx} + u_{yy}),$$

is a wave traveling in the direction of $\boldsymbol{k} = k_x\boldsymbol{i} + k_y\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively ($\boldsymbol{i}$ must not be confused with $i = \sqrt{-1}$). Such a wave can be expressed by

$$u(x, y, t) = g(k_x x + k_y y - kct)$$

for some twice differentiable function g, or with $\omega = kc$, $k = |\boldsymbol{k}|$:

$$u(x, y, t) = g(k_x x + k_y y - \omega t).$$

We can, in particular, build a solution by adding complex Fourier components of the form

$$e^{(i(k_x x + k_y y - \omega t))}.$$

A discrete 2D wave equation can be written as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]^n_{q,r}. \tag{113}$$

This equation admits a Fourier component

$$u^n_{q,r} = e^{(i(k_x q \Delta x + k_y r \Delta y - \tilde{\omega} n \Delta t))}, \tag{114}$$

as solution. Letting the operators $D_t D_t$, $D_x D_x$, and $D_y D_y$ act on $u^n_{q,r}$ from (114) transforms (113) to

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) + c^2\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right). \tag{115}$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y, \tag{116}$$

where we have eliminated the factor 4 and introduced the symbols

$$C_x = \frac{c\Delta t}{\Delta x}, \quad C_y = \frac{c\Delta t}{\Delta y}, \quad p_x = \frac{k_x \Delta x}{2}, \quad p_y = \frac{k_y \Delta y}{2}\,.$$

For a real-valued $\tilde{\omega}$ the right-hand side must be less than or equal to unity in absolute value, requiring in general that

$$C_x^2 + C_y^2 \leq 1\,. \tag{117}$$

This gives the stability criterion, more commonly expressed directly in an inequality for the time step:

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{118}$$

A similar, straightforward analysis for the 3D case leads to

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{119}$$

In the case of a variable coefficient $c^2 = c^2(\boldsymbol{x})$, we must use the worst-case value

$$\bar{c} = \sqrt{\max_{\boldsymbol{x}\in\Omega} c^2(\boldsymbol{x})} \tag{120}$$

in the stability criteria. Often, especially in the variable wave velocity case, it is wise to introduce a safety factor $\beta \in (0, 1]$ too:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{121}$$

The exact numerical dispersion relations in 2D and 3D becomes, for constant c,

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y\right)^{\frac{1}{2}}\right), \tag{122}$$

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y + C_z^2 \sin^2 p_z\right)^{\frac{1}{2}}\right)\,. \tag{123}$$

We can visualize the numerical dispersion error in 2D much like we did in 1D. To this end, we need to reduce the number of parameters in $\tilde{\omega}$. The direction of the wave is parameterized by the polar angle θ, which means that

$$k_x = k\sin\theta, \quad k_y = k\cos\theta\,.$$

A simplification is to set $\Delta x = \Delta y = h$. Then $C_x = C_y = c\Delta t/h$, which we call C. Also,

$$p_x = \frac{1}{2}kh\cos\theta, \quad p_y = \frac{1}{2}kh\sin\theta.$$

The numerical frequency $\tilde{\omega}$ is now a function of three parameters:

- C, reflecting the number of cells a wave is displaced during a time step,

- $p = \frac{1}{2}kh$, reflecting the number of cells per wave length in space,

- θ, expressing the direction of the wave.

We want to visualize the error in the numerical frequency. To avoid having Δt as a free parameter in $\tilde{\omega}$, we work with $\tilde{c}/c = \tilde{\omega}/(kc)$. The coefficient in front of the $\sin^{-1}$ factor is then

$$\frac{2}{kc\Delta t} = \frac{2}{2kc\Delta th/h} = \frac{1}{Ckh} = \frac{2}{Cp},$$

and

$$\frac{\tilde{c}}{c} = \frac{2}{Cp}\sin^{-1}\left(C\left(\sin^2(p\cos\theta) + \sin^2(p\sin\theta)\right)^{\frac{1}{2}}\right).$$

We want to visualize this quantity as a function of p and θ for some values of $C \leq 1$. It is instructive to make color contour plots of $1 - \tilde{c}/c$ in *polar coordinates* with θ as the angular coordinate and p as the radial coordinate.

The stability criterion (117) becomes $C \leq C_{\max} = 1/\sqrt{2}$ in the present 2D case with the C defined above. Let us plot $1 - \tilde{c}/c$ in polar coordinates for $C_{\max}, 0.9C_{\max}, 0.5C_{\max}, 0.2C_{\max}$. The program below does the somewhat tricky work in Matplotlib, and the result appears in Figure 7. From the figure we clearly see that the maximum C value gives the best results, and that waves whose propagation direction makes an angle of 45 degrees with an axis are the most accurate.

```
def dispersion_relation_2D(p, theta, C):
    arg = C*sqrt(sin(p*cos(theta))**2 +
                 sin(p*sin(theta))**2)
    c_frac = 2./(C*p)*arcsin(arg)

    return c_frac

import numpy as np
from numpy import \
    cos, sin, arcsin, sqrt, pi  # for nicer math formulas

r = p = np.linspace(0.001, pi/2, 101)
theta = np.linspace(0, 2*pi, 51)
r, theta = np.meshgrid(r, theta)

# Make 2x2 filled contour plots for 4 values of C
import matplotlib.pyplot as plt
C_max = 1/sqrt(2)
C = [[C_max, 0.9*C_max], [0.5*C_max, 0.2*C_max]]
fix, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
for row in range(2):
```

```
    for column in range(2):
        error = 1 - dispersion_relation_2D(
            p, theta, C[row][column])
        print error.min(), error.max()
        # use vmin=error.min(), vmax=error.max()
        cax = axes[row][column].contourf(
            theta, r, error, 50, vmin=-1, vmax=-0.28)
        axes[row][column].set_xticks([])
        axes[row][column].set_yticks([])

# Add colorbar to the last plot
cbar = plt.colorbar(cax)
cbar.ax.set_ylabel('error in wave velocity')
plt.savefig('disprel2D.png');  plt.savefig('disprel2D.pdf')
plt.show()
```
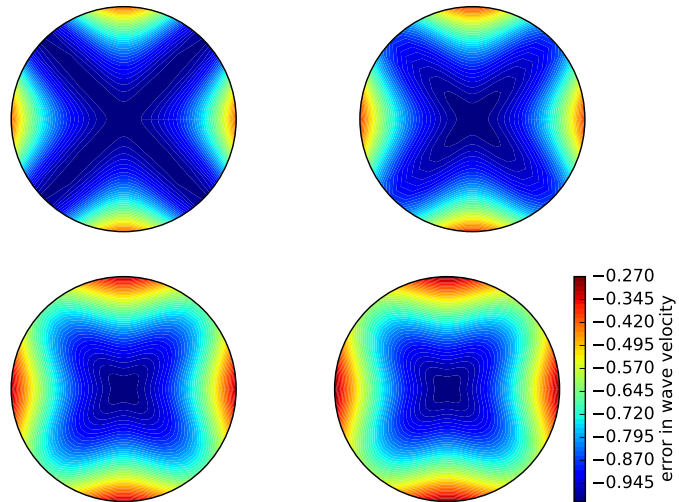

Figure 7: Error in numerical dispersion in 2D.

11 Finite difference methods for 2D and 3D wave equations

A natural next step is to consider extensions of the methods for various variants of the one-dimensional wave equation to two-dimensional (2D) and three-dimensional (3D) versions of the wave equation.

98

11.1 Multi-dimensional wave equations

The general wave equation in d space dimensions, with constant wave velocity c, can be written in the compact form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d, \ t \in (0, T], \tag{124}$$

where

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

in a 2D problem $(d = 2)$ and

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2},$$

in three space dimensions $(d = 3)$.

Many applications involve variable coefficients, and the general wave equation in d dimensions is in this case written as

$$\varrho \frac{\partial^2 u}{\partial t^2} = \nabla \cdot (q \nabla u) + f \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d, \ t \in (0, T], \tag{125}$$

which in, e.g., 2D becomes

$$\varrho(x, y) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x, y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x, y)\frac{\partial u}{\partial y}\right) + f(x, y, t). \tag{126}$$

To save some writing and space we may use the index notation, where subscript t, x, or y means differentiation with respect to that coordinate. For example,

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

$$\frac{\partial}{\partial y}\left(q(x, y)\frac{\partial u}{\partial y}\right) = (q u_y)_y.$$

These comments extend straightforwardly to 3D, which means that the 3D versions of the two wave PDEs, with and without variable coefficients, can be stated as

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f, \tag{127}$$

$$\varrho u_{tt} = (q u_x)_x + (q u_z)_z + (q u_z)_z + f. \tag{128}$$

At *each point* of the boundary $\partial\Omega$ (of Ω) we need *one* boundary condition involving the unknown u. The boundary conditions are of three principal types:

1. u is prescribed ($u = 0$ or a known time variation of u at the boundary points, e.g., modeling an incoming wave),

2. $\partial u / \partial n = \boldsymbol{n} \cdot \nabla u$ is prescribed (zero for reflecting boundaries),

3. an open boundary condition (also called radiation condition) is specified to let waves travel undisturbed out of the domain, see Exercise 11 for details.

All the listed wave equations with *second-order* derivatives in time need *two* initial conditions:

1. $u = I$,

2. $u_t = V$.

11.2 Mesh

We introduce a mesh in time and in space. The mesh in time consists of time points

$$t_0 = 0 < t_1 < \cdots < t_{N_t},$$

often with a constant spacing $\Delta t = t_{n+1} - t_n$, $n \in \mathcal{I}_t^-$.

Finite difference methods are easy to implement on simple rectangle- or box-shaped domains. More complicated shapes of the domain require substantially more advanced techniques and implementational efforts. On a rectangle- or box-shaped domain, mesh points are introduced separately in the various space directions:

$$x_0 < x_1 < \cdots < x_{N_x} \text{ in the } x \text{ direction},$$
$$y_0 < y_1 < \cdots < y_{N_y} \text{ in the } y \text{ direction},$$
$$z_0 < z_1 < \cdots < z_{N_z} \text{ in the } z \text{ direction}.$$

We can write a general mesh point as (x_i, y_j, z_k, t_n), with $i \in \mathcal{I}_x$, $j \in \mathcal{I}_y$, $k \in \mathcal{I}_z$, and $n \in \mathcal{I}_t$.

It is a very common choice to use constant mesh spacings: $\Delta x = x_{i+1} - x_i$, $i \in \mathcal{I}_x^-$, $\Delta y = y_{j+1} - y_j$, $j \in \mathcal{I}_y^-$, and $\Delta z = z_{k+1} - z_k$, $k \in \mathcal{I}_z^-$. With equal mesh spacings one often introduces $h = \Delta x = \Delta y = \Delta z$.

The unknown u at mesh point (x_i, y_j, z_k, t_n) is denoted by $u_{i,j,k}^n$. In 2D problems we just skip the z coordinate (by assuming no variation in that direction: $\partial / \partial z = 0$) and write $u_{i,j}^n$.

11.3 Discretization

Two- and three-dimensional wave equations are easily discretized by assembling building blocks for discretization of 1D wave equations, because the multi-dimensional versions just contain terms of the same type as those in 1D.

Discretizing the PDEs. Equation (127) can be discretized as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u + D_z D_z u) + f]_{i,j,k}^n \,. \qquad (129)$$

A 2D version might be instructive to write out in detail:

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]_{i,j,k}^n,$$

which becomes

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^n + u_{i,j}^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + c^2 \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} + f_{i,j}^n,$$

Assuming, as usual, that all values at time levels n and $n-1$ are known, we can solve for the only unknown $u_{i,j}^{n+1}$. The result can be compactly written as

$$u_{i,j}^{n+1} = 2u_{i,j}^n + u_{i,j}^{n-1} + c^2 \Delta t^2 [D_x D_x u + D_y D_y u]_{i,j}^n \,. \qquad (130)$$

As in the 1D case, we need to develop a special formula for $u_{i,j}^1$ where we combine the general scheme for $u_{i,j}^{n+1}$, when $n = 0$, with the discretization of the initial condition:

$$[D_{2t} u = V]_{i,j}^0 \quad \Rightarrow \quad u_{i,j}^{-1} = u_{i,j}^1 - 2\Delta t V_{i,j} \,.$$

The result becomes, in compact form,

$$u_{i,j}^1 = u_{i,j}^0 - 2\Delta V_{i,j} + \frac{1}{2} c^2 \Delta t^2 [D_x D_x u + D_y D_y u]_{i,j}^0 \,. \qquad (131)$$

The PDE (128) with variable coefficients is discretized term by term using the corresponding elements from the 1D case:

$$[\varrho D_t D_t u = (D_x \overline{q}^x D_x u + D_y \overline{q}^y D_y u + D_z \overline{q}^z D_z u) + f]_{i,j,k}^n \,. \qquad (132)$$

When written out and solved for the unknown $u_{i,j,k}^{n+1}$, one gets the scheme

$$
\begin{aligned}
u_{i,j,k}^{n+1} = {}& -u_{i,j,k}^{n-1} + 2u_{i,j,k}^n + \\
& \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \big(\frac{1}{2}(q_{i,j,k} + q_{i+1,j,k})(u_{i+1,j,k}^n - u_{i,j,k}^n) - \\
& \quad \frac{1}{2}(q_{i-1,j,k} + q_{i,j,k})(u_{i,j,k}^n - u_{i-1,j,k}^n) \big) + \\
& \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \big(\frac{1}{2}(q_{i,j,k} + q_{i,j+1,k})(u_{i,j+1,k}^n - u_{i,j,k}^n) - \\
& \quad \frac{1}{2}(q_{i,j-1,k} + q_{i,j,k})(u_{i,j,k}^n - u_{i,j-1,k}^n) \big) + \\
& \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \big(\frac{1}{2}(q_{i,j,k} + q_{i,j,k+1})(u_{i,j,k+1}^n - u_{i,j,k}^n) - \\
& \quad \frac{1}{2}(q_{i,j,k-1} + q_{i,j,k})(u_{i,j,k}^n - u_{i,j,k-1}^n) \big) + \\
& \Delta t^2 f_{i,j,k}^n \,.
\end{aligned}
$$

Also here we need to develop a special formula for $u_{i,j,k}^1$ by combining the scheme for $n = 0$ with the discrete initial condition, which is just a matter of inserting $u_{i,j,k}^{-1} = u_{i,j,k}^1 - 2\Delta t V_{i,j,k}$ in the scheme and solving for $u_{i,j,k}^1$.

Handling boundary conditions where u is known. The schemes listed above are valid for the internal points in the mesh. After updating these, we need to visit all the mesh points at the boundaries and set the prescribed u value.

Discretizing the Neumann condition. The condition $\partial u / \partial n = 0$ was implemented in 1D by discretizing it with a $D_{2x} u$ centered difference, followed by eliminating the fictitious u point outside the mesh by using the general scheme at the boundary point. Alternatively, one can introduce ghost cells and update a ghost value for use in the Neumann condition. Exactly the same ideas are reused in multiple dimensions.

Consider the condition $\partial u / \partial n = 0$ at a boundary $y = 0$ of a rectangular domain $[0, L_x] \times [0, L_y]$ in 2D. The normal direction is then in $-y$ direction, so

$$
\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y},
$$

and we set

$$
[-D_{2y} u = 0]_{i,0}^n \quad \Rightarrow \quad \frac{u_{i,1}^n - u_{i,-1}^n}{2\Delta y} = 0 \,.
$$

From this it follows that $u_{i,-1}^n = u_{i,1}^n$. The discretized PDE at the boundary point $(i, 0)$ reads

$$\frac{u_{i,0}^{n+1} - 2u_{i,0}^n + u_{i,0}^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1,0}^n - 2u_{i,0}^n + u_{i-1,0}^n}{\Delta x^2} + c^2 \frac{u_{i,1}^n - 2u_{i,0}^n + u_{i,-1}^n}{\Delta y^2} + f_{i,j}^n,$$

We can then just insert $u_{i,1}^n$ for $u_{i,-1}^n$ in this equation and solve for the boundary value $u_{i,0}^{n+1}$, just as was done in 1D.

From these calculations, we see a pattern: the general scheme applies at the boundary $j = 0$ too if we just replace $j - 1$ by $j + 1$. Such a pattern is particularly useful for implementations. The details follow from the explained 1D case in Section 6.3.

The alternative approach to eliminating fictitious values outside the mesh is to have $u_{i,-1}^n$ available as a ghost value. The mesh is extended with one extra line (2D) or plane (3D) of ghost cells at a Neumann boundary. In the present example it means that we need a line with ghost cells below the y axis. The ghost values must be updated according to $u_{i,-1}^{n+1} = u_{i,1}^{n+1}$.

12 Implementation

We shall now describe in detail various Python implementations for solving a standard 2D, linear wave equation with constant wave velocity and $u = 0$ on the boundary. The wave equation is to be solved in the space-time domain $\Omega \times (0, T]$, where $\Omega = (0, L_x) \times (0, L_y)$ is a rectangular spatial domain. More precisely, the complete initial-boundary value problem is defined by

$$u_{tt} = c^2(u_{xx} + u_{yy}) + f(x, y, t), \qquad (x, y) \in \Omega, \ t \in (0, T], \qquad (133)$$
$$u(x, y, 0) = I(x, y), \qquad (x, y) \in \Omega, \qquad (134)$$
$$u_t(x, y, 0) = V(x, y), \qquad (x, y) \in \Omega, \qquad (135)$$
$$u = 0, \qquad (x, y) \in \partial\Omega, \ t \in (0, T], \qquad (136)$$

where $\partial\Omega$ is the boundary of Ω, in this case the four sides of the rectangle $\Omega = [0, L_x] \times [0, L_y]$: $x = 0$, $x = L_x$, $y = 0$, and $y = L_y$.

The PDE is discretized as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]_{i,j}^n,$$

which leads to an explicit updating formula to be implemented in a program:

$$u_{i,j}^{n+1} = -u_{i,j}^{n-1} + 2u_{i,j}^n +$$
$$C_x^2(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + C_y^2(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n) + \Delta t^2 f_{i,j}^n, \qquad (137)$$

for all interior mesh points $i \in \mathcal{I}_x^i$ and $j \in \mathcal{I}_y^i$, for $n \in \mathcal{I}_t^+$. The constants C_x and C_y are defined as

$$C_x = c\frac{\Delta t}{\Delta x}, \quad C_x = c\frac{\Delta t}{\Delta y} \ .$$

At the boundary, we simply set $u_{i,j}^{n+1} = 0$ for $i = 0$, $j = 0, \ldots, N_y$; $i = N_x$, $j = 0, \ldots, N_y$; $j = 0$, $i = 0, \ldots, N_x$; and $j = N_y$, $i = 0, \ldots, N_x$. For the first step, $n = 0$, (137) is combined with the discretization of the initial condition $u_t = V$, $[D_{2t}u = V]_{i,j}^0$ to obtain a special formula for $u_{i,j}^1$ at the interior mesh points:

$$
\begin{aligned}
u_{i,j}^1 = u_{i,j}^0 &+ \Delta t V_{i,j} + \\
&\frac{1}{2}C_x^2(u_{i+1,j}^0 - 2u_{i,j}^0 + u_{i-1,j}^0) + \frac{1}{2}C_y^2(u_{i,j+1}^0 - 2u_{i,j}^0 + u_{i,j-1}^0) + \\
&\frac{1}{2}\Delta t^2 f_{i,j}^n,
\end{aligned}
\tag{138}
$$

The algorithm is very similar to the one in 1D:

1. Set initial condition $u_{i,j}^0 = I(x_i, y_j)$

2. Compute $u_{i,j}^1$ from (137)

3. Set $u_{i,j}^1 = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

4. For $n = 1, 2, \ldots, N_t$:

   (a) Find $u_{i,j}^{n+1}$ from (137) for all internal mesh points, $i \in \mathcal{I}_x^i$, $j \in \mathcal{I}_y^i$

   (b) Set $u_{i,j}^{n+1} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

12.1 Scalar computations

The `solver` function for a 2D case with constant wave velocity and boundary condition $u = 0$ is analogous to the 1D case with similar parameter values (see `wave1D_u0.py`), apart from a few necessary extensions. The code is found in the program `wave2D_u0.py`[19].

Domain and mesh. The spatial domain is now $[0, L_x] \times [0, L_y]$, specified by the arguments `Lx` and `Ly`. Similarly, the number of mesh points in the x and y directions, N_x and N_y, become the arguments `Nx` and `Ny`. In multi-dimensional problems it makes less sense to specify a Courant number since the wave velocity is a vector and mesh spacings may differ in the various spatial directions. We therefore give Δt explicitly. The signature of the `solver` function is then

---

[19]`http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Key parameters used in the calculations are created as

```
x = linspace(0, Lx, Nx+1)                 # mesh points in x dir
y = linspace(0, Ly, Ny+1)                 # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)                # mesh points in time
Cx2 = (c*dt/dx)**2;  Cy2 = (c*dt/dy)**2   # help variables
dt2 = dt**2
```

Solution arrays. We store $u_{i,j}^{n+1}$, $u_{i,j}^n$, and $u_{i,j}^{n-1}$ in three two-dimensional arrays,

```
u   = zeros((Nx+1,Ny+1))   # solution array
u_1 = zeros((Nx+1,Ny+1))   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1))   # solution at t-2*dt
```

where $u_{i,j}^{n+1}$ corresponds to `u[i,j]`, $u_{i,j}^n$ to `u_1[i,j]`, and $u_{i,j}^{n-1}$ to `u_2[i,j]`

Index sets. It is also convenient to introduce the index sets (cf. Section 6.4)

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])
```

Computing the solution. Inserting the initial condition I in `u_1` and making a callback to the user in terms of the **user_action** function is a straightforward generalization of the 1D code from Section 1.6:

```
for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, x, xv, y, yv, t, 0)
```

The **user_action** function has additional arguments compared to the 1D case. The arguments `xv` and `yv` will be commented upon in Section 12.2.

The key finite difference formula (130) for updating the solution at a time level is implemented in a separate function as

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
```

105

```
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

The `step1` variable has been introduced to allow the formula to be reused for first step $u_{i,j}^1$:

```
u = advance_scalar(u, u_1, u_2, f, x, y, t,
                   n, Cx2, Cy2, dt, V, step1=True)
```

Below, we will make many alternative implementations of the `advance_scalar` function to speed up the code since most of the CPU time in simulations is spent in this function.

Finally, we remark that the `solver` function in the `wave2D_u0.py` code updates arrays for the next time step by switching references as described in Section 4.5. If the solution `u` is returned from `solver`, which it is not, it is important to set `u = u_1` after the time loop, otherwise `u` actually contains `u_2`.

12.2 Vectorized computations

The scalar code above turns out to be extremely slow for large 2D meshes, and probably useless in 3D beyond debugging of small test cases. Vectorization is therefore a must for multi-dimensional finite difference computations in Python. For example, with a mesh consisting of 30×30 cells, vectorization brings down the CPU time by a factor of 70 (!). Equally important, vectorized code can also easily be parallelized to take (usually) optimal advantage of parallel computer platforms.

In the vectorized case, we must be able to evaluate user-given functions like $I(x,y)$ and $f(x,y,t)$ for the entire mesh in one operation (without loops). These user-given functions are provided as Python functions `I(x,y)` and `f(x,y,t)`, respectively. Having the one-dimensional coordinate arrays `x` and `y` is not sufficient when calling `I` and `f` in a vectorized way. We must extend `x` and `y` to their vectorized versions `xv` and `yv`:

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]
# or
xv = x.reshape((x.size, 1))
yv = y.reshape((1, y.size))
```

This is a standard required technique when evaluating functions over a 2D mesh, say `sin(xv)*cos(xv)`, which then gives a result with shape `(Nx+1,Ny+1)`. Calling `I(xv, yv)` and `f(xv, yv, t[n])` will now return `I` and `f` values for the entire set of mesh points.

With the `xv` and `yv` arrays for vectorized computing, setting the initial condition is just a matter of

```
u_1[:,:] = I(xv, yv)
```

One could also have written `u_1 = I(xv, yv)` and let `u_1` point to a new object, but vectorized operations often make use of direct insertion in the original array through `u_1[:,:]`, because sometimes not all of the array is to be filled by such a function evaluation. This is the case with the computational scheme for $u_{i,j}^{n+1}$:

```
def advance_vectorized(u, u_1, u_2, f_a, Cx2, Cy2, dt2,
                       V=None, step1=False):
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;   D2 = 0
    else:
        D1 = 2;   D2 = 1
    u_xx = u_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
    u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
    u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[1:-1,1:-1] + \
                   Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]
    if step1:
        u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
    # Boundary condition u=0
    j = 0
    u[:,j] = 0
    j = u.shape[1]-1
    u[:,j] = 0
    i = 0
    u[i,:] = 0
    i = u.shape[0]-1
    u[i,:] = 0
    return u
```

Array slices in 2D are more complicated to understand than those in 1D, but the logic from 1D applies to each dimension separately. For example, when doing $u_{i,j}^n - u_{i-1,j}^n$ for $i \in \mathcal{I}_x^+$, we just keep `j` constant and make a slice in the first index: `u_1[1:,j] - u_1[:-1,j]`, exactly as in 1D. The `1:` slice specifies all the indices $i = 1, 2, \ldots, N_x$ (up to the last valid index), while `:-1` specifies the relevant indices for the second term: $0, 1, \ldots, N_x - 1$ (up to, but not including the last index).

In the above code segment, the situation is slightly more complicated, because each displaced slice in one direction is accompanied by a `1:-1` slice in the other

107

direction. The reason is that we only work with the internal points for the index that is kept constant in a difference.

The boundary conditions along the four sides makes use of a slice consisting of all indices along a boundary:

```
u[: ,0] = 0
u[:,Ny] = 0
u[0 ,:] = 0
u[Nx,:] = 0
```

In the vectorized update of `u` (above), the function `f` is first computed as an array over all mesh points:

```
f_a = f(xv, yv, t[n])
```

We could, alternatively, have used the call `f(xv, yv, t[n])[1:-1,1:-1]` in the last term of the update statement, but other implementations in compiled languages benefit from having `f` available in an array rather than calling our Python function `f(x,y,t)` for every point.

Also in the `advance_vectorized` function we have introduced a boolean `step1` to reuse the formula for the first time step in the same way as we did with `advance_scalar`. We refer to the `solver` function in `wave2D_u0.py` for the details on how the overall algorithm is implemented.

The callback function now has the arguments `u, x, xv, y, yv, t, n`. The inclusion of `xv` and `yv` makes it easy to, e.g., compute an exact 2D solution in the callback function and compute errors, through an expression like `u - u_exact(xv, yv, t[n])`.

12.3   Verification

Testing a quadratic solution.   The 1D solution from Section 2.4 can be generalized to multi-dimensions and provides a test case where the exact solution also fulfills the discrete equations, such that we know (to machine precision) what numbers the solver function should produce. In 2D we use the following generalization of (30):

$$u_{\mathrm{e}}(x, y, t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t) \,. \tag{139}$$

This solution fulfills the PDE problem if $I(x, y) = u_{\mathrm{e}}(x, y, 0)$, $V = \frac{1}{2}u_{\mathrm{e}}(x, y, 0)$, and $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$. To show that u_{e} also solves the discrete equations, we start with the general results $[D_t D_t 1]^n = 0$, $[D_t D_t t]^n = 0$, and $[D_t D_t t^2] = 2$, and use these to compute

$$[D_x D_x u_{\mathrm{e}}]_{i,j}^n = [y(L_y - y)(1 + \frac{1}{2}t)D_x D_x x(L_x - x)]_{i,j}^n$$
$$= y_j(L_y - y_j)(1 + \frac{1}{2}t_n)(-2) \,.$$

108

A similar calculation must be carried out for the $[D_y D_y u_e]_{i,j}^n$ and $[D_t D_t u_e]_{i,j}^n$ terms. One must also show that the quadratic solution fits the special formula for $u_{i,j}^1$. The details are left as Exercise 15. The `test_quadratic` function in the `wave2D_u0.py`[20] program implements this verification as a proper test function for the pytest and nose frameworks.

13   Exercises

Exercise 15: Check that a solution fulfills the discrete model

Carry out all mathematical details to show that (139) is indeed a solution of the discrete model for a 2D wave equation with $u = 0$ on the boundary. One must check the boundary conditions, the initial conditions, the general discrete equation at a time level and the special version of this equation for the first time level. Filename: `check_quadratic_solution`.

Project 16: Calculus with 2D mesh functions

The goal of this project is to redo Project 5 with 2D mesh functions $(f_{i,j})$.

Differentiation.   The differentiation results in a discrete gradient function, which in the 2D case can be represented by a three-dimensional array `df[d,i,j]` where `d` represents the direction of the derivative, and `i,j` is a mesh point in 2D. Use centered differences for the derivative at inner points and one-sided forward or backward differences at the boundary points. Construct unit tests and write a corresponding test function.

Integration.   The integral of a 2D mesh function $f_{i,j}$ is defined as

$$F_{i,j} = \int_{y_0}^{y_j} \int_{x_0}^{x_i} f(x,y)dxdy,$$

where $f(x,y)$ is a function that takes on the values of the discrete mesh function $f_{i,j}$ at the mesh points, but can also be evaluated in between the mesh points. The particular variation between mesh points can be taken as bilinear, but this is not important as we will use a product Trapezoidal rule to approximate the integral over a cell in the mesh and then we only need to evaluate $f(x,y)$ at the mesh points.

Suppose $F_{i,j}$ is computed. The calculation of $F_{i+1,j}$ is then

$$F_{i+1,j} = F_{i,j} + \int_{x_i}^{x_{i+1}} \int_{y_0}^{y_j} f(x,y)dydx$$

$$\approx \Delta x \frac{1}{2} \left(\int_{y_0}^{y_j} f(x_i,y)dy + \int_{y_0}^{y_j} f(x_{i+1},y)dy \right)$$

---

[20] `http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`

109

The integrals in the y direction can be approximated by a Trapezoidal rule. A similar idea can be used to compute $F_{i,j+1}$. Thereafter, $F_{i+1,j+1}$ can be computed by adding the integral over the final corner cell to $F_{i+1,j} + F_{i,j+1} - F_{i,j}$. Carry out the details of these computations and implement a function that can return $F_{i,j}$ for all mesh indices i and j. Use the fact that the Trapezoidal rule is exact for linear functions and write a test function. Filename: `mesh_calculus_2D`.

Exercise 17: Implement Neumann conditions in 2D

Modify the `wave2D_u0.py`[21] program, which solves the 2D wave equation $u_{tt} = c^2(u_{xx} + u_{yy})$ with constant wave velocity c and $u = 0$ on the boundary, to have Neumann boundary conditions: $\partial u/\partial n = 0$. Include both scalar code (for debugging and reference) and vectorized code (for speed).

To test the code, use $u = 1.2$ as solution ($I(x, y) = 1.2$, $V = f = 0$, and c arbitrary), which should be exactly reproduced with any mesh as long as the stability criterion is satisfied. Another test is to use the plug-shaped pulse in the `pulse` function from Section 8 and the `wave1D_dn_vc.py`[22] program. This pulse is exactly propagated in 1D if $c\Delta t/\Delta x = 1$. Check that also the 2D program can propagate this pulse exactly in x direction ($c\Delta t/\Delta x = 1$, Δy arbitrary) and y direction ($c\Delta t/\Delta y = 1$, Δx arbitrary). Filename: `wave2D_dn`.

Exercise 18: Test the efficiency of compiled loops in 3D

Extend the `wave2D_u0.py` code and the Cython, Fortran, and C versions to 3D. Set up an efficiency experiment to determine the relative efficiency of pure scalar Python code, vectorized code, Cython-compiled loops, Fortran-compiled loops, and C-compiled loops. Normalize the CPU time for each mesh by the fastest version. Filename: `wave3D_u0`.

14 Applications of wave equations

This section presents a range of wave equation models for different physical phenomena. Although many wave motion problems in physics can be modeled by the standard linear wave equation, or a similar formulation with a system of first-order equations, there are some exceptions. Perhaps the most important is water waves: these are modeled by the Laplace equation with time-dependent boundary conditions at the water surface (long water waves, however, can be approximated by a standard wave equation, see Section 14.7). Quantum mechanical waves constitute another example where the waves are governed by the Schrödinger equation, i.e., not by a standard wave equation. Many wave phenomena also need to take nonlinear effects into account when the wave amplitude is significant. Shock waves in the air is a primary example.

---

[21]`http://tinyurl.com/nm5587k/wave/wave2D_u0/wave2D_u0.py`
[22]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

The derivations in the following are very brief. Those with a firm background in continuum mechanics will probably have enough knowledge to fill in the details, while other readers will hopefully get some impression of the physics and approximations involved when establishing wave equation models.
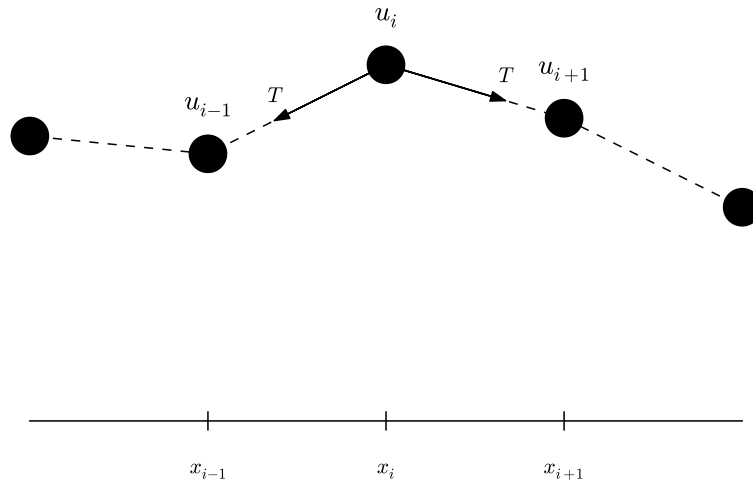
14.1 Waves on a string

Figure 8: Discrete string model with point masses connected by elastic strings.

Figure 8 shows a model we may use to derive the equation for waves on a string. The string is modeled as a set of discrete point masses (at mesh points) with elastic strings in between. The string has a large constant tension T. We let the mass at mesh point x_i be m_i. The displacement of this mass point in the y direction is denoted by $u_i(t)$.

The motion of mass m_i is governed by Newton's second law of motion. The position of the mass at time t is $x_i\boldsymbol{i} + u_i(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y direction, respectively. The acceleration is then $u_i''(t)\boldsymbol{j}$. Two forces are acting on the mass as indicated in Figure 8. The force $\boldsymbol{T}^-$ acting toward the point x_{i-1} can be decomposed as

$$\boldsymbol{T}^- = -T\sin\phi\boldsymbol{i} - T\cos\phi\boldsymbol{j},$$

where ϕ is the angle between the force and the line $x = x_i$. Let $\Delta u_i = u_i - u_{i-1}$ and let $\Delta s_i = \sqrt{\Delta u_i^2 + (x_i - x_{i-1})^2}$ be the distance from mass m_{i-1} to mass m_i. It is seen that $\cos\phi = \Delta u_i/\Delta s_i$ and $\sin\phi = (x_i - x_{i-1})/\Delta s$ or $\Delta x/\Delta s_i$ if we introduce a constant mesh spacing $\Delta x = x_i - x_{i-1}$. The force can then be written

$$\boldsymbol{T}^- = -T\frac{\Delta x}{\Delta s_i}\boldsymbol{i} - T\frac{\Delta u_i}{\Delta s_i}\boldsymbol{j}.$$

The force $\boldsymbol{T}^+$ acting toward x_{i+1} can be calculated in a similar way:

$$\boldsymbol{T}^+ = T\frac{\Delta x}{\Delta s_{i+1}}\boldsymbol{i} + T\frac{\Delta u_{i+1}}{\Delta s_{i+1}}\boldsymbol{j}.$$

Newton's second law becomes

$$m_i u_i''(t)\boldsymbol{j} = \boldsymbol{T}^+ + \boldsymbol{T}^-,$$

which gives the component equations

$$T\frac{\Delta x}{\Delta s_i} = T\frac{\Delta x}{\Delta s_{i+1}}, \tag{140}$$

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta s_{i+1}} - T\frac{\Delta u_i}{\Delta s_i}. \tag{141}$$

A basic reasonable assumption for a string is small displacements u_i and small displacement gradients $\Delta u_i/\Delta x$. For small $g = \Delta u_i/\Delta x$ we have that

$$\Delta s_i = \sqrt{\Delta u_i^2 + \Delta x^2} = \Delta x\sqrt{1 + g^2} + \Delta x(1 + \frac{1}{2}g^2 + \mathcal{O}(g^4) \approx \Delta x.$$

Equation (140) is then simply the identity $T = T$, while (141) can be written as

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta x} - T\frac{\Delta u_i}{\Delta x},$$

which upon division by Δx and introducing the density $\varrho_i = m_i/\Delta x$ becomes

$$\varrho_i u_i''(t) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right). \tag{142}$$

We can now choose to approximate u_i'' by a finite difference in time and get the discretized wave equation,

$$\varrho_i \frac{1}{\Delta t^2} \left(u_i^{n+1} - 2u_i^n - u_i^{n-1} \right) = T \frac{1}{\Delta x^2} \left(u_{i+1} - 2u_i + u_{i-1} \right) . \tag{143}$$

On the other hand, we may go to the continuum limit $\Delta x \to 0$ and replace $u_i(t)$ by $u(x,t)$, ϱ_i by $\varrho(x)$, and recognize that the right-hand side of (142) approaches $\partial^2 u / \partial x^2$ as $\Delta x \to 0$. We end up with the continuous model for waves on a string:

$$\varrho \frac{\partial^2 u}{\partial t^2} = T \frac{\partial^2 u}{\partial x^2} . \tag{144}$$

Note that the density ϱ may change along the string, while the tension T is a constant. With variable wave velocity $c(x) = \sqrt{T/\varrho(x)}$ we can write the wave equation in the more standard form

$$\frac{\partial^2 u}{\partial t^2} = c^2(x) \frac{\partial^2 u}{\partial x^2} . \tag{145}$$

Because of the way ϱ enters the equations, the variable wave velocity does *not* appear inside the derivatives as in many other versions of the wave equation. However, most strings of interest have constant ϱ.

The end points of a string are fixed so that the displacement u is zero. The boundary conditions are therefore $u = 0$.

Damping. Air resistance and non-elastic effects in the string will contribute to reduce the amplitudes of the waves so that the motion dies out after some time. This damping effect can be modeled by a term bu_t on the left-hand side of the equation

$$\varrho \frac{\partial^2 u}{\partial t^2} + b \frac{\partial u}{\partial t} = T \frac{\partial^2 u}{\partial x^2} . \tag{146}$$

The parameter $b \geq 0$ is small for most wave phenomena, but the damping effect may become significant in long time simulations.

External forcing. It is easy to include an external force acting on the string. Say we have a vertical force $\tilde{f}_i \boldsymbol{j}$ acting on mass m_i. This force affects the vertical component of Newton's law and gives rise to an extra term $\tilde{f}(x,t)$ on the right-hand side of (144). In the model (145) we would add a term $f(x,t) = \tilde{f}(x,y)/\varrho(x)$.

Modeling the tension via springs. We assumed, in the derivation above, that the tension in the string, T, was constant. It is easy to check this assumption by modeling the string segments between the masses as standard springs, where the force (tension T) is proportional to the elongation of the spring segment. Let k be the spring constant, and set $T_i = k\Delta \ell$ for the tension in the spring segment between x_{i-1} and x_i, where $\Delta \ell$ is the elongation of this segment from

the tension-free state. A basic feature of a string is that it has high tension in the equilibrium position $u = 0$. Let the string segment have an elongation $\Delta \ell_0$ in the equilibrium position. After deformation of the string, the elongation is $\Delta \ell = \Delta \ell_0 + \Delta s_i$: $T_i = k(\Delta \ell_0 + \Delta s_i) \approx k(\Delta \ell_0 + \Delta x)$. This shows that T_i is independent of i. Moreover, the extra approximate elongation Δx is very small compared to $\Delta \ell_0$, so we may well set $T_i = T = k \Delta \ell_0$. This means that the tension is completely dominated by the initial tension determined by the tuning of the string. The additional deformations of the spring during the vibrations do not introduce significant changes in the tension.

14.2 Waves on a membrane

hpl 3: Must write the memberane model. Easiest to use Navier.

14.3 Elastic waves in a rod

Consider an elastic rod subject to a hammer impact at the end. This experiment will give rise to an elastic deformation pulse that travels through the rod. A mathematical model for longitudinal waves along an elastic rod starts with the general equation for deformations and stresses in an elastic medium,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{147}$$

where ϱ is the density, $\boldsymbol{u}$ the displacement field, $\boldsymbol{\sigma}$ the stress tensor, and $\boldsymbol{f}$ body forces. The latter has normally no impact on elastic waves.

For stationary deformation of an elastic rod, one has that $\sigma_{xx} = E u_x$, with all other stress components being zero. The parameter E is known as Young's modulus. Moreover, we set $\boldsymbol{u} = u(x,t)\boldsymbol{i}$ and neglect the radial contraction and expansion (where Poisson's ratio is the important parameter). Assuming that this simple stress and deformation field is a good approximation, (147) simplifies to

$$\varrho \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(E \frac{\partial u}{\partial x} \right). \tag{148}$$

The associated boundary conditions are u or $\sigma_{xx} = E u_x$ known, typically $u = 0$ for a fixed end and $\sigma_{xx} = 0$ for a free end.

14.4 The acoustic model for seismic waves

Seismic waves are used to infer properties of subsurface geological structures. The physical model is a heterogeneous elastic medium where sound is propagated by small elastic vibrations. The general mathematical model for deformations in an elastic medium is based on Newton's second law,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{149}$$

and a constitutive law relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$, often Hooke's generalized law,

$$\boldsymbol{\sigma} = K\nabla \cdot \boldsymbol{u}\,\boldsymbol{I} + G(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3}\nabla \cdot \boldsymbol{u}\,\boldsymbol{I})\,. \tag{150}$$

Here, $\boldsymbol{u}$ is the displacement field, $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{I}$ is the identity tensor, ϱ is the medium's density, $\boldsymbol{f}$ are body forces (such as gravity), K is the medium's bulk modulus and G is the shear modulus. All these quantities may vary in space, while $\boldsymbol{u}$ and $\boldsymbol{\sigma}$ will also show significant variation in time during wave motion.

The acoustic approximation to elastic waves arises from a basic assumption that the second term in Hooke's law, representing the deformations that give rise to shear stresses, can be neglected. This assumption can be interpreted as approximating the geological medium by a fluid. Neglecting also the body forces $\boldsymbol{f}$, (149) becomes

$$\varrho \boldsymbol{u}_{tt} = \nabla(K\nabla \cdot \boldsymbol{u}) \tag{151}$$

Introducing p as a pressure via

$$p = -K\nabla \cdot \boldsymbol{u}, \tag{152}$$

and dividing (151) by ϱ, we get

$$\boldsymbol{u}_{tt} = -\frac{1}{\varrho}\nabla p\,. \tag{153}$$

Taking the divergence of this equation, using $\nabla \cdot \boldsymbol{u} = -p/K$ from (152), gives the *acoustic approximation to elastic waves*:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right)\,. \tag{154}$$

This is a standard, linear wave equation with variable coefficients. It is common to add a source term $s(x, y, z, t)$ to model the generation of sound waves:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) + s\,. \tag{155}$$

A common additional approximation of (155) is based on using the chain rule on the right-hand side,

$$K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) = \frac{K}{\varrho}\nabla^2 p + K\nabla\left(\frac{1}{\varrho}\right) \cdot \nabla p \approx \frac{K}{\varrho}\nabla^2 p,$$

under the assumption that the relative spatial gradient $\nabla\varrho^{-1} = -\varrho^{-2}\nabla\varrho$ is small. This approximation results in the simplified equation

$$p_{tt} = \frac{K}{\varrho}\nabla^2 p + s\,. \tag{156}$$

The acoustic approximations to seismic waves are used for sound waves in the ground, and the Earth's surface is then a boundary where p equals the atmospheric pressure p_0 such that the boundary condition becomes $p = p_0$.

Anisotropy. Quite often in geological materials, the effective wave velocity $c = \sqrt{K/\varrho}$ is different in different spatial directions because geological layers are compacted, and often twisted, in such a way that the properties in the horizontal and vertical direction differ. With z as the vertical coordinate, we can introduce a vertical wave velocity c_z and a horizontal wave velocity c_h, and generalize (156) to

$$p_{tt} = c_z^2 p_{zz} + c_h^2 (p_{xx} + p_{yy}) + s \,. \tag{157}$$

14.5 Sound waves in liquids and gases

Sound waves arise from pressure and density variations in fluids. The starting point of modeling sound waves is the basic equations for a compressible fluid where we omit viscous (frictional) forces, body forces (gravity, for instance), and temperature effects:

$$\varrho_t + \nabla \cdot (\varrho \boldsymbol{u}) = 0, \tag{158}$$
$$\varrho \boldsymbol{u}_t + \varrho \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\nabla p, \tag{159}$$
$$\varrho = \varrho(p) \,. \tag{160}$$

These equations are often referred to as the Euler equations for the motion of a fluid. The parameters involved are the density ϱ, the velocity $\boldsymbol{u}$, and the pressure p. Equation (159) reflects mass balance, (158) is Newton's second law for a fluid, with frictional and body forces omitted, and (160) is a constitutive law relating density to pressure by thermodynamic considerations. A typical model for (160) is the so-called isentropic relation[23], valid for adiabatic processes where there is no heat transfer:

$$\varrho = \varrho_0 \left(\frac{p}{p_0} \right)^{1/\gamma} \,. \tag{161}$$

Here, p_0 and ϱ_0 are reference values for p and ϱ when the fluid is at rest, and γ is the ratio of specific heat at constant pressure and constant volume ($\gamma = 5/3$ for air).

The key approximation in a mathematical model for sound waves is to assume that these waves are small perturbations to the density, pressure, and velocity. We therefore write

$$p = p_0 + \hat{p},$$
$$\varrho = \varrho_0 + \hat{\varrho},$$
$$\boldsymbol{u} = \hat{\boldsymbol{u}},$$

---

[23]http://en.wikipedia.org/wiki/Isentropic_process

where we have decomposed the fields in a constant equilibrium value, corresponding to $\boldsymbol{u} = 0$, and a small perturbation marked with a hat symbol. By inserting these decompositions in (158) and (159), neglecting all product terms of small perturbations and/or their derivatives, and dropping the hat symbols, one gets the following linearized PDE system for the small perturbations in density, pressure, and velocity:

$$\varrho_t + \varrho_0 \nabla \cdot \boldsymbol{u} = 0, \tag{162}$$

$$\varrho_0 \boldsymbol{u}_t = -\nabla p. \tag{163}$$

Now we can eliminate ϱ_t by differentiating the relation $\varrho(p)$,

$$\varrho_t = \varrho_0 \frac{1}{\gamma} \left(\frac{p}{p_0} \right)^{1/\gamma - 1} \frac{1}{p_0} p_t = \frac{\varrho_0}{\gamma p_0} \left(\frac{p}{p_0} \right)^{1/\gamma - 1} p_t.$$

The product term $p^{1/\gamma - 1} p_t$ can be linearized as $p_0^{1/\gamma - 1} p_t$, resulting in

$$\varrho_t \approx \frac{\varrho_0}{\gamma p_0} p_t.$$

We then get

$$p_t + \gamma p_0 \nabla \cdot \boldsymbol{u} = 0, \tag{164}$$

$$\boldsymbol{u}_t = -\frac{1}{\varrho_0} \nabla p, . \tag{165}$$

Taking the divergence of (165) and differentiating (164) with respect to time gives the possibility to easily eliminate $\nabla \cdot \boldsymbol{u}_t$ and arrive at a standard, linear wave equation for p:

$$p_{tt} = c^2 \nabla^2 p, \tag{166}$$

where $c = \sqrt{\gamma p_0 / \varrho_0}$ is the speed of sound in the fluid.

14.6 Spherical waves

Spherically symmetric three-dimensional waves propagate in the radial direction r only so that $u = u(r, t)$. The fully three-dimensional wave equation

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

then reduces to the spherically symmetric wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{r^2} \frac{\partial}{\partial r} \left(c^2(r) r^2 \frac{\partial u}{\partial r} \right) + f(r, t), \quad r \in (0, R), \ t > 0. \tag{167}$$

One can easily show that the function $v(r,t) = ru(r,t)$ fulfills a standard wave equation in Cartesian coordinates if c is constant. To this end, insert $u = v/r$ in

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial r}\right)$$

to obtain

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + c^2\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(c^2\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one Cartesian coordinate. The spherically symmetric wave equation in terms of $v(r,t)$ now becomes

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial r}\left(c^2(r)\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{dc^2}{dr}v + rf(r,t), \quad r \in (0,R),\ t > 0\,. \tag{168}$$

In the case of constant wave velocity c, this equation reduces to the wave equation in a single Cartesian coordinate called r:

$$\frac{\partial^2 v}{\partial t^2} = c^2\frac{\partial^2 v}{\partial r^2} + rf(r,t), \quad r \in (0,R),\ t > 0\,. \tag{169}$$

That is, any program for solving the one-dimensional wave equation in a Cartesian coordinate system can be used to solve (169), provided the source term is multiplied by the coordinate, and that we divide the Cartesian mesh solution by r to get the spherically symmetric solution. Moreover, if $r = 0$ is included in the domain, spherical symmetry demands that $\partial u/\partial r = 0$ at $r = 0$, which means that

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r = 0\,.$$

For this to hold in the limit $r \to 0$, we must have $v(0,t) = 0$ at least as a necessary condition. In most practical applications, we exclude $r = 0$ from the domain and assume that some boundary condition is assigned at $r = \epsilon$, for some $\epsilon > 0$.

hpl 4: Should work out the condition for v further. It is a necessary condition, but the expression is $0/0$, so we must differentiate. Rough calculation says it ends with $v_{rr} = 0$ as another condition. Could discuss with Hjorth-Jensen.

14.7 The linear shallow water equations

The next example considers water waves whose wavelengths are much larger than the depth and whose wave amplitudes are small. This class of waves may be generated by catastrophic geophysical events, such as earthquakes at the sea bottom, landslides moving into water, or underwater slides (or a combination, as earthquakes frequently release avalanches of masses). For example, a subsea earthquake will normally have an extension of many kilometers but lift the water only a few meters. The wave length will have a size dictated by the earthquake area, which is much lager than the water depth, and compared to this wave length, an amplitude of a few meters is very small. The water is essentially a thin film, and mathematically we can average the problem in the vertical direction and approximate the 3D wave phenomenon by 2D PDEs. Instead of a moving water domain in three space dimensions, we get a horizontal 2D domain with an unknown function for the surface elevation and the water depth as a variable coefficient in the PDEs.

Let $\eta(x, y, t)$ be the elevation of the water surface, $H(x, y)$ the water depth corresponding to a flat surface ($\eta = 0$), $u(x, y, t)$ and $v(x, y, t)$ the depth-averaged horizontal velocities of the water. Mass and momentum balance of the water volume give rise to the PDEs involving these quantities:

$$\eta_t = -(Hu)_x - (Hv)_x \tag{170}$$

$$u_t = -g\eta_x, \tag{171}$$

$$v_t = -g\eta_y, \tag{172}$$

where g is the acceleration of gravity. Equation (170) corresponds to mass balance while the other two are derived from momentum balance (Newton's second law).

The initial conditions associated with (170)-(172) are η, u, and v prescribed at $t = 0$. A common condition is to have some water elevation $\eta = I(x, y)$ and assume that the surface is at rest: $u = v = 0$. A subsea earthquake usually means a sufficiently rapid motion of the bottom and the water volume to say that the bottom deformation is mirrored at the water surface as an initial lift $I(x, y)$ and that $u = v = 0$.

Boundary conditions may be η prescribed for incoming, known waves, or zero normal velocity at reflecting boundaries (steep mountains, for instance): $un_x + vn_y = 0$, where (n_x, n_y) is the outward unit normal to the boundary. More sophisticated boundary conditions are needed when waves run up at the shore, and at open boundaries where we want the waves to leave the computational domain undisturbed.

Equations (170), (171), and (172) can be transformed to a standard, linear wave equation. First, multiply (171) and (172) by H, differentiate (171)) with respect to x and (172) with respect to y. Second, differentiate (170) with respect to t and use that $(Hu)_{xt} = (Hu_t)_x$ and $(Hv)_{yt} = (Hv_t)_y$ when H is independent of t. Third, eliminate $(Hu_t)_x$ and $(Hv_t)_y$ with the aid of the other

two differentiated equations. These manipulations result in a standard, linear wave equation for η:

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla \cdot (gH\nabla\eta)\,. \tag{173}$$

In the case we have an initial non-flat water surface at rest, the initial conditions become $\eta = I(x,y)$ and $\eta_t = 0$. The latter follows from (170) if $u = v = 0$, or simply from the fact that the vertical velocity of the surface is η_t, which is zero for a surface at rest.

The system (170)-(172) can be extended to handle a time-varying bottom topography, which is relevant for modeling long waves generated by underwater slides. In such cases the water depth function H is also a function of t, due to the moving slide, and one must add a time-derivative term H_t to the left-hand side of (170). A moving bottom is best described by introducing $z = H_0$ as the still-water level, $z = B(x,y,t)$ as the time- and space-varying bottom topography, so that $H = H_0 - B(x,y,t)$. In the elimination of u and v one may assume that the dependence of H on t can be neglected in the terms $(Hu)_{xt}$ and $(Hv)_{yt}$. We then end up with a source term in (173), because of the moving (accelerating) bottom:

$$\eta_{tt} = \nabla \cdot (gH\nabla\eta) + B_{tt}\,. \tag{174}$$

The reduction of (174) to 1D, for long waves in a straight channel, or for approximately plane waves in the ocean, is trivial by assuming no change in y direction ($\partial/\partial y = 0$):

$$\eta_{tt} = (gH\eta_x)_x + B_{tt}\,. \tag{175}$$

Wind drag on the surface. Surface waves are influenced by the drag of the wind, and if the wind velocity some meters above the surface is (U,V), the wind drag gives contributions $C_V\sqrt{U^2 + V^2}U$ and $C_V\sqrt{U^2 + V^2}V$ to (171) and (172), respectively, on the right-hand sides.

Bottom drag. The waves will experience a drag from the bottom, often roughly modeled by a term similar to the wind drag: $C_B\sqrt{u^2 + v^2}u$ on the right-hand side of (171) and $C_B\sqrt{u^2 + v^2}v$ on the right-hand side of (172). Note that in this case the PDEs (171) and (172) become nonlinear and the elimination of u and v to arrive at a 2nd-order wave equation for η is not possible anymore.

Effect of the Earth's rotation. Long geophysical waves will often be affected by the rotation of the Earth because of the Coriolis force. This force gives rise to a term fv on the right-hand side of (171) and $-fu$ on the right-hand side of (172). Also in this case one cannot eliminate u and v to work with a single equation for η. The Coriolis parameter is $f = 2\Omega\sin\phi$, where Ω is the angular velocity of the earth and ϕ is the latitude.

14.8 Waves in blood vessels

The flow of blood in our bodies is basically fluid flow in a network of pipes. Unlike rigid pipes, the walls in the blood vessels are elastic and will increase their diameter when the pressure rises. The elastic forces will then push the wall back and accelerate the fluid. This interaction between the flow of blood and the deformation of the vessel wall results in waves traveling along our blood vessels.

A model for one-dimensional waves along blood vessels can be derived from averaging the fluid flow over the cross section of the blood vessels. Let x be a coordinate along the blood vessel and assume that all cross sections are circular, though with different radii $R(x, t)$. The main quantities to compute is the cross section area $A(x, t)$, the averaged pressure $P(x, t)$, and the total volume flux $Q(x, t)$. The area of this cross section is

$$A(x, t) = 2\pi \int_0^{R(x,t)} r\, dr, \tag{176}$$

Let $v_x(x, t)$ be the velocity of blood averaged over the cross section at point x. The volume flux, being the total volume of blood passing a cross section per time unit, becomes

$$Q(x, t) = A(x, t) v_x(x, t) \tag{177}$$

Mass balance and Newton's second law lead to the PDEs

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{178}$$

$$\frac{\partial Q}{\partial t} + \frac{\gamma + 2}{\gamma + 1} \frac{\partial}{\partial x}\left(\frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} = -2\pi(\gamma + 2)\frac{\mu}{\varrho}\frac{Q}{A}, \tag{179}$$

where γ is a parameter related to the velocity profile, ϱ is the density of blood, and μ is the dynamic viscosity of blood.

We have three unknowns A, Q, and P, and two equations (178) and (179). A third equation is needed to relate the flow to the deformations of the wall. A common form for this equation is

$$\frac{\partial P}{\partial t} + \frac{1}{C}\frac{\partial Q}{\partial x} = 0, \tag{180}$$

where C is the compliance of the wall, given by the constitutive relation

$$C = \frac{\partial A}{\partial P} + \frac{\partial A}{\partial t}, \tag{181}$$

which requires a relationship between A and P. One common model is to view the vessel wall, locally, as a thin elastic tube subject to an internal pressure. This gives the relation

$$P = P_0 + \frac{\pi h E}{(1 - \nu^2)A_0}(\sqrt{A} - \sqrt{A_0}),$$

121

where P_0 and A_0 are corresponding reference values when the wall is not deformed, h is the thickness of the wall, and E and ν are Young's modulus and Poisson's ratio of the elastic material in the wall. The derivative becomes

$$C = \frac{\partial A}{\partial P} = \frac{2(1-\nu^2)A_0}{\pi h E}\sqrt{A_0} + 2\left(\frac{(1-\nu^2)A_0}{\pi h E}\right)^2 (P - P_0). \qquad (182)$$

Another (nonlinear) deformation model of the wall, which has a better fit with experiments, is

$$P = P_0 \exp\left(\beta(A/A_0 - 1)\right),$$

where β is some parameter to be estimated. This law leads to

$$C = \frac{\partial A}{\partial P} = \frac{A_0}{\beta P}. \qquad (183)$$

Reduction to the standard wave equation. It is not uncommon to neglect the viscous term on the right-hand side of (179) and also the quadratic term with Q^2 on the left-hand side. The reduced equations (179) and (180) form a first-order linear wave equation system:

$$C\frac{\partial P}{\partial t} = -\frac{\partial Q}{\partial x}, \qquad (184)$$

$$\frac{\partial Q}{\partial t} = -\frac{A}{\varrho}\frac{\partial P}{\partial x}. \qquad (185)$$

These can be combined into standard 1D wave PDE by differentiating the first equation with respect t and the second with respect to x,

$$\frac{\partial}{\partial t}\left(C\frac{\partial P}{\partial t}\right) = \frac{\partial}{\partial x}\left(\frac{A}{\varrho}\frac{\partial P}{\partial x}\right),$$

which can be approximated by

$$\frac{\partial^2 Q}{\partial t^2} = c^2\frac{\partial^2 Q}{\partial x^2}, \quad c = \sqrt{\frac{A}{\varrho C}}, \qquad (186)$$

where the A and C in the expression for c are taken as constant reference values.

14.9 Electromagnetic waves

Light and radio waves are governed by standard wave equations arising from Maxwell's general equations. When there are no charges and no currents, as in a vacuum, Maxwell's equations take the form

$$\nabla \cdot \boldsymbol{E} = 0,$$
$$\nabla \cdot \boldsymbol{B} = 0,$$
$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t},$$
$$\nabla \times \boldsymbol{B} = \mu_0 \epsilon_0 \frac{\partial \boldsymbol{E}}{\partial t},$$

where $\epsilon_0 = 8.854187817620 \cdot 10^{-12}$ (F/m) is the permittivity of free space, also known as the electric constant, and $\mu_0 = 1.2566370614 \cdot 10^{-6}$ (H/m) is the permeability of free space, also known as the magnetic constant. Taking the curl of the two last equations and using the mathematical identity

$$\nabla \times (\nabla \times \boldsymbol{E}) = \nabla(\nabla \cdot \boldsymbol{E}) - \nabla^2 \boldsymbol{E} = -\nabla^2 \boldsymbol{E} \text{ when } \nabla \cdot \boldsymbol{E} = 0,$$

gives the wave equation governing the electric and magnetic field:

$$\frac{\partial^2 \boldsymbol{E}}{\partial t^2} = c^2 \nabla^2 \boldsymbol{E}, \tag{187}$$

$$\frac{\partial^2 \boldsymbol{B}}{\partial t^2} = c^2 \nabla^2 \boldsymbol{B}, \tag{188}$$

with $c = 1/\sqrt{\mu_0 \epsilon_0}$ as the velocity of light. Each component of $\boldsymbol{E}$ and $\boldsymbol{B}$ fulfills a wave equation and can hence be solved independently.

15 Exercises

Exercise 19: Simulate waves on a non-homogeneous string

Simulate waves on a string that consists of two materials with different density. The tension in the string is constant, but the density has a jump at the middle of the string. Experiment with different sizes of the jump and produce animations that visualize the effect of the jump on the wave motion.

Hint. According to Section 14.1, the density enters the mathematical model as ϱ in $\varrho u_{tt} = T u_{xx}$, where T is the string tension. Modify, e.g., the `wave1D_u0v.py` code to incorporate the tension and two density values. Make a mesh function `rho` with density values at each spatial mesh point. A value for the tension may be 150 N. Corresponding density values can be computed from the wave velocity estimations in the `guitar` function in the `wave1D_u0v.py` file.
Filename: `wave1D_u0_sv_discont`.

Exercise 20: Simulate damped waves on a string

Formulate a mathematical model for damped waves on a string. Use data from Section 3.5, and tune the damping parameter so that the string is very close to the rest state after 15 s. Make a movie of the wave motion. Filename: `wave1D_u0_sv_damping`.

Exercise 21: Simulate elastic waves in a rod

A hammer hits the end of an elastic rod. The exercise is to simulate the resulting wave motion using the model (148) from Section 14.3. Let the rod have length L and let the boundary $x = L$ be stress free so that $\sigma_{xx} = 0$, implying that $\partial u / \partial x = 0$. The left end $x = 0$ is subject to a strong stress pulse (the hammer), modeled as

$$\sigma_{xx}(t) = \left\{ \begin{array}{ll} S, & 0 < t \leq t_s, \\ 0, & t > t_s \end{array} \right.$$

The corresponding condition on u becomes $u_x = S/E$ for $t \leq t_s$ and zero afterwards (recall that $\sigma_{xx} = E u_x$). This is a non-homogeneous Neumann condition, and you will need to approximate this condition and combine it with the scheme (the ideas and manipulations follow closely the handling of a non-zero initial condition $u_t = V$ in wave PDEs or the corresponding second-order ODEs for vibrations). Filename: `wave_rod`.

Exercise 22: Simulate spherical waves

Implement a model for spherically symmetric waves using the method described in Section 14.6. The boundary condition at $r = 0$ must be $\partial u / \partial r = 0$, while the condition at $r = R$ can either be $u = 0$ or a radiation condition as described in Problem 11. The $u = 0$ condition is sufficient if R is so large that the amplitude of the spherical wave has become insignificant. Make movie(s) of the case where the source term is located around $r = 0$ and sends out pulses

$$f(r, t) = \left\{ \begin{array}{ll} Q \exp\left(-\frac{r^2}{2 \Delta r^2}\right) \sin \omega t, & \sin \omega t \geq 0 \\ 0, & \sin \omega t < 0 \end{array} \right.$$

Here, Q and ω are constants to be chosen.

Hint. Use the program `wave1D_u0v.py` as a starting point. Let `solver` compute the v function and then set $u = v/r$. However, $u = v/r$ for $r = 0$ requires special treatment. One possibility is to compute `u[1:]  = v[1:]/r[1:]` and then set `u[0]=u[1]`. The latter makes it evident that $\partial u / \partial r = 0$ in a plot. Filename: `wave1D_spherical`.

Problem 23: Earthquake-generated tsunami over a subsea hill

A subsea earthquake leads to an immediate lift of the water surface, see Figure 9. The lifted water surface splits into two tsunamis, one traveling to the right and one to the left, as depicted in Figure 10. Since tsunamis are normally very long waves, compared to the depth, with a small amplitude, compared to the wave length, the wave equation model described in Section 14.7 is relevant:

$$\eta_{tt} = (gH(x)\eta_x)_x,$$

where g is the acceleration of gravity, and $H(x)$ is the still water depth.
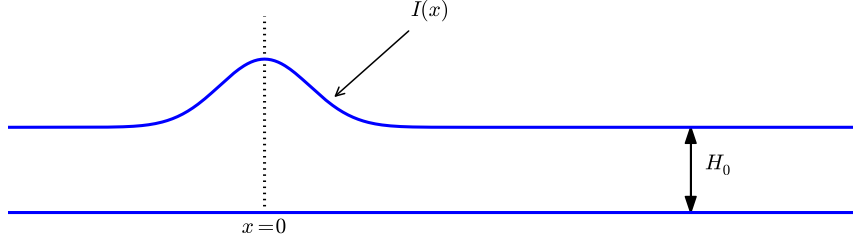

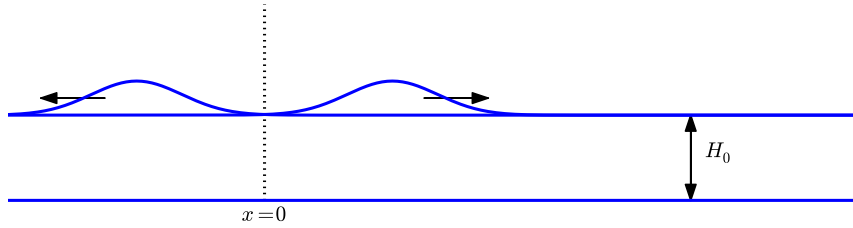Figure 9: Sketch of initial water surface due to a subsea earthquake.

Figure 10: An initial surface elevation is split into two waves.

To simulate the right-going tsunami, we can impose a symmetry boundary at $x = 0$: $\partial \eta / \partial x = 0$. We then simulate the wave motion in $[0, L]$. Unless the ocean ends at $x = L$, the waves should travel undisturbed through the boundary $x = L$. A radiation condition as explained in Problem 11 can be used for this purpose. Alternatively, one can just stop the simulations before the wave hits the boundary at $x = L$. In that case it does not matter what kind of boundary condition we use at $x = L$. Imposing $\eta = 0$ and stopping the simulations when $|\eta_i^n| > \epsilon$, $i = N_x - 1$, is a possibility (ϵ is a small parameter).

The shape of the initial surface can be taken as a Gaussian function,

125

$$I(x; I_0, I_a, I_m, I_s) = I_0 + I_a \exp\left(-\left(\frac{x - I_m}{I_s}\right)^2\right), \tag{189}$$

with $I_m = 0$ reflecting the location of the peak of $I(x)$ and I_s being a measure of the width of the function $I(x)$ (I_s is $\sqrt{2}$ times the standard deviation of the familiar normal distribution curve).

Now we extend the problem with a hill at the sea bottom, see Figure 11. The wave speed $c = \sqrt{gH(x)} = \sqrt{g(H_0 - B(x))}$ will then be reduced in the shallow water above the hill.
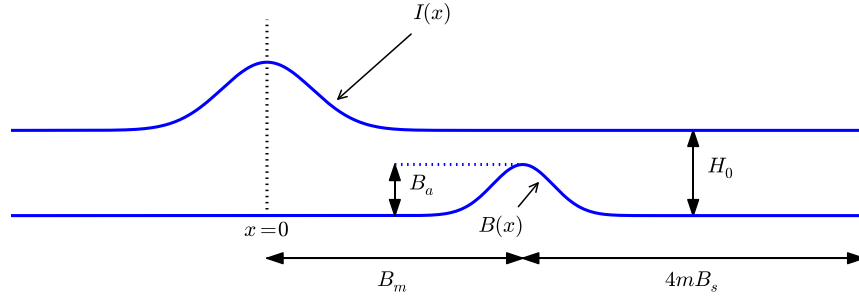
Figure 11: Sketch of an earthquake-generated tsunami passing over a subsea hill.

One possible form of the hill is a Gaussian function,

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \exp\left(-\left(\frac{x - B_m}{B_s}\right)^2\right), \tag{190}$$

but many other shapes are also possible, e.g., a "cosine hat" where

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_m}{2B_s}\right), \tag{191}$$

when $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

Also an abrupt construction may be tried:

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a, \tag{192}$$

for $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

The `wave1D_dn_vc.py`[24] program can be used as starting point for the implementation. Visualize both the bottom topography and the water surface

---

[24]`http://tinyurl.com/nm5587k/wave/wave1D/wave1D_dn_vc.py`

elevation in the same plot. Allow for a flexible choice of bottom shape: (190), (191), (192), or $B(x) = B_0$ (flat).

The purpose of this problem is to explore the quality of the numerical solution η_i^n for different shapes of the bottom obstruction. The "cosine hat" and the box-shaped hills have abrupt changes in the derivative of $H(x)$ and are more likely to generate numerical noise than the smooth Gaussian shape of the hill. Investigate if this is true. Filename: `tsunami1D_hill`.

Problem 24: Earthquake-generated tsunami over a 3D hill

This problem extends Problem 23 to a three-dimensional wave phenomenon, governed by the 2D PDE (173). We assume that the earthquake arises from a fault along the line $x = 0$ in the xy-plane so that the initial lift of the surface can be taken as $I(x)$ in Problem 23. That is, a plane wave is propagating to the right, but will experience bending because of the bottom.

The bottom shape is now a function of x and y. An "elliptic" Gaussian function in two dimensions, with its peak at (B_{mx}, B_{my}), generalizes (190):

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s, b) = B_0 + B_a \exp\left(-\left(\frac{x - B_{mx}}{B_s}\right)^2 - \left(\frac{y - B_{my}}{bB_s}\right)^2\right),$$

(193)

where b is a scaling parameter: $b = 1$ gives a circular Gaussian function with circular contour lines, while $b \neq 1$ gives an elliptic shape with elliptic contour lines.

The "cosine hat" (191) can also be generalized to

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_{mx}}{2B_s}\right) \cos\left(\pi \frac{y - B_{my}}{2B_s}\right),$$

(194)

when $0 \leq \sqrt{x^2 + y^2} \leq B_s$ and $B = B_0$ outside this circle.

A box-shaped obstacle means that

$$B(x; B_0, B_a, B_m, B_s, b) = B_0 + B_a$$

(195)

for x and y inside a rectangle

$$B_{mx} - B_s \leq x \leq B_{mx} + B_s, \quad B_{my} - bB_s \leq y \leq B_{my} + bB_s,$$

and $B = B_0$ outside this rectangle. The b parameter controls the rectangular shape of the cross section of the box.

Note that the initial condition and the listed bottom shapes are symmetric around the line $y = B_{my}$. We therefore expect the surface elevation also to be symmetric with respect to this line. This means that we can halve the computational domain by working with $[0, L_x] \times [0, B_{my}]$. Along the upper boundary, $y = B_{my}$, we must impose the symmetry condition $\partial\eta/\partial n = 0$. Such

a symmetry condition $(-\eta_x = 0)$ is also needed at the $x = 0$ boundary because the initial condition has a symmetry here. At the lower boundary $y = 0$ we also set a Neumann condition (which becomes $-\eta_y = 0$). The wave motion is to be simulated until the wave hits the reflecting boundaries where $\partial \eta / \partial n = \eta_x = 0$ (one can also set $\eta = 0$ - the particular condition does not matter as long as the simulation is stopped before the wave is influenced by the boundary condition).

Visualize the surface elevation. Investigate how different hill shapes, different sizes of the water gap above the hill, and different resolutions $\Delta x = \Delta y = h$ and Δt influence the numerical quality of the solution. Filename: `tsunami2D_hill`.

Problem 25: Investigate Matplotlib for visualization

Play with native Matplotlib code for visualizing 2D solutions of the wave equation with variable wave velocity. See if there are effective ways to visualize both the solution and the wave velocity. Filename: `tsunami2D_hill_mpl`.

Problem 26: Investigate visualization packages

Create some fancy 3D visualization of the water waves *and* the subsea hill in Problem 24. Try to make the hill transparent. Possible visualization tools are Mayavi[25], Paraview[26], and OpenDX[27]. Filename: `tsunami2D_hill_viz`.

Problem 27: Implement loops in compiled languages

Extend the program from Problem 24 such that the loops over mesh points, inside the time loop, are implemented in compiled languages. Consider implementations in Cython, Fortran via `f2py`, C via Cython, C via `f2py`, C/C++ via Instant, and C/C++ via `scipy.weave`. Perform efficiency experiments to investigate the relative performance of the various implementations. It is often advantageous to normalize CPU times by the fastest method on a given mesh. Filename: `tsunami2D_hill_compiled`.

Exercise 28: Simulate seismic waves in 2D

The goal of this exercise is to simulate seismic waves using the PDE model (157) in a 2D xz domain with geological layers. Introduce m horizontal layers of thickness h_i, $i = 0, \ldots, m - 1$. Inside layer number i we have a vertical wave velocity $c_{z,i}$ and a horizontal wave velocity $c_{h,i}$. Make a program for simulating such 2D waves. Test it on a case with 3 layers where

$$c_{z,0} = c_{z,1} = c_{z,2}, \quad c_{h,0} = c_{h,2}, \quad c_{h,1} \ll c_{h,0}.$$

Let s be a localized point source at the middle of the Earth's surface (the upper boundary) and investigate how the resulting wave travels through the medium.

---

[25]`http://code.enthought.com/projects/mayavi/`
[26]`http://www.paraview.org/`
[27]`http://www.opendx.org/`

The source can be a localized Gaussian peak that oscillates in time for some time interval. Place the boundaries far enough from the expanding wave so that the boundary conditions do not disturb the wave. Then the type of boundary condition does not matter, except that we physically need to have $p = p_0$, where p_0 is the atmospheric pressure, at the upper boundary. Filename: `seismic2D`.

Project 29: Model 3D acoustic waves in a room

The equation for sound waves in air is derived in Section 14.5 and reads

$$p_{tt} = c^2 \nabla^2 p,$$

where $p(x, y, z, t)$ is the pressure and c is the speed of sound, taken as 340 m/s. However, sound is absorbed in the air due to relaxation of molecules in the gas. A model for simple relaxation, valid for gases consisting only of one type of molecules, is a term $c^2 \tau_s \nabla^2 p_t$ in the PDE, where τ_s is the relaxation time. If we generate sound from, e.g., a loudspeaker in the room, this sound source must also be added to the governing equation.

The PDE with the mentioned type of damping and source then becomes

$$p_t t = c^2 \nabla^p + c^2 \tau_s \nabla^2 p_t + f, \tag{196}$$

where $f(x, y, z, t)$ is the source term.

The walls can absorb some sound. A possible model is to have a "wall layer" (thicker than the physical wall) outside the room where c is changed such that some of the wave energy is reflected and some is absorbed in the wall. The absorption of energy can be taken care of by adding a damping term bp_t in the equation:

$$p_t t + bp_t = c^2 \nabla^p + c^2 \tau_s \nabla^2 p_t + f. \tag{197}$$

Typically, $b = 0$ in the room and $b > 0$ in the wall. A discontinuity in b or c will give rise to reflections. It can be wise to use a constant c in the wall to control reflections because of the discontinuity between c in the air and in the wall, while b is gradually increased as we go into the wall to avoid reflections because of rapid changes in b. At the outer boundary of the wall the condition $p = 0$ or $\partial p / \partial n = 0$ can be imposed. The waves should anyway be approximately dampened to $p = 0$ this far out in the wall layer.

There are two strategies for discretizing the $\nabla^2 p_t$ term: using a center difference between times $n + 1$ and $n - 1$ (if the equation is sampled at level n), or use a one-sided difference based on levels n and $n - 1$. The latter has the advantage of not leading to any equation system, while the former is second-order accurate as the scheme for the simple wave equation $p_t t = c^2 \nabla^2 p$. To avoid an equation system, go for the one-sided difference such that the overall scheme becomes explicit and only of first order in time.

Develop a 3D solver for the specified PDE and introduce a wall layer. Test the solver with the method of manufactured solutions. Make some demonstrations

where the wall reflects and absorbs the waves (reflection because of discontinuity in b and absorption because of growing b). Experiment with the impact of the τ_s parameter. Filename: `acoustics`.

Project 30: Solve a 1D transport equation

We shall study the wave equation

$$u_t + cu_x = 0, \quad x \in (0, L], \ t \in (0, T], \tag{198}$$

with initial condition

$$u(x, 0) = I(x), \quad x \in [0, L], \tag{199}$$

and *one* periodic boundary condition

$$u(0, t) = u(L, t). \tag{200}$$

This boundary condition means that what goes out of the domain at $x = L$ comes in at $x = 0$. Roughly speaking, we need only one boundary condition because of the spatial derivative is of first order only.

Physical interpretation. The parameter c can be constant or variable, $c = c(x)$. The equation (198) arises in *transport* problems where a quantity u, which could be temperature or concentration of some contaminant, is transported with the velocity c of a fluid. In addition to the transport imposed by "travelling with the fluid", u may also be transported by diffusion (such as heat conduction or Fickian diffusion), but we have in the model $u_t + cu_x$ assumed that diffusion effects are negligible, which they often are.

a) Show that under the assumption of $a = \text{const}$,

$$u(x, t) = I(x - ct) \tag{201}$$

fulfills the PDE as well as the initial and boundary condition (provided $I(0) = I(L)$).

A widely used numerical scheme for (198) applies a forward difference in time and a backward difference in space when $c > 0$:

$$[D_t^+ u + cD_x^- u = 0]_i^n. \tag{202}$$

For $c < 0$ we use a forward difference in space: $[cD_x^+ u]_i^n$.

b) Set up a computational algorithm and implement it in a function. Assume a is constant and positive.

c) Test the implementation by using the remarkable property that the numerical solution is exact at the mesh points if $\Delta t = c^{-1} \Delta x$.

d) Make a movie comparing the numerical and exact solution for the following two choices of initial conditions:

$$I(x) = \left[\sin\left(\pi\frac{x}{L}\right)\right]^{2n} \tag{203}$$

where n is an integer, typically $n = 5$, and

$$I(x) = \exp\left(-\frac{(x - L/2)^2}{2\sigma2}\right). \tag{204}$$

Choose $\Delta t = c^{-1}\Delta x, 0.9c^{-1}\Delta x, 0.5c^{-1}\Delta x$.

e) The performance of the suggested numerical scheme can be investigated by analyzing the numerical dispersion relation. Analytically, we have that the *Fourier component*

$$u(x, t) = e^{i(kx - \omega t)},$$

is a solution of the PDE if $\omega = kc$. This is the *analytical dispersion relation.* A complete solution of the PDE can be built by adding up such Fourier components with different amplitudes, where the initial condition I determines the amplitudes. The solution u is then represented by a Fourier series.

A similar discrete Fourier component at (x_p, t_n) is

$$u_p^q = e^{i(kp\Delta x - \tilde{\omega}n\Delta t)},$$

where in general $\tilde{\omega}$ is a function of k, Δt, and Δx, and differs from the exact $\omega = kc$.

Insert the discrete Fourier component in the numerical scheme and derive an expression for $\tilde{\omega}$, i.e., the discrete dispersion relation. Show in particular that if $\Delta t/(c\Delta x) = 1$, the discrete solution coincides with the exact solution at the mesh points, regardless of the mesh resolution (!). Show that if the stability condition

$$\frac{\Delta t}{c\Delta x} \leq 1,$$

the discrete Fourier component cannot grow (i.e., $\tilde{\omega}$ is real).

f) Write a test for your implementation where you try to use information from the numerical dispersion relation.

We shall hereafter assume that $c(x) > 0$.

g) Set up a computational algorithm for the variable coefficient case and implement it in a function. Make a test that the function works for constant a.

h) It can be shown that for an observer moving with velocity $c(x)$, u is constant. This can be used to derive an exact solution when a varies with x. Show first that

$$u(x, t) = f(C(x) - t), \tag{205}$$

where

$$C'(x) = \frac{1}{c(x)},$$

is a solution of (198) for any differentiable function f.

Solution. Let $\xi = C(x) - t$. We have that

$$u_t = f'(\xi)(-1),$$

while

$$u_x = f'(\xi)C'(x) = f'(\xi)\frac{1}{c(x)},$$

implying that $au_x = f'(\xi)$. Then we have $u_t + cu_x = -f'(\xi) + f'(\xi) = 0$.

i) Use the initial condition to show that an exact solution is

$$u(x, t) = I(C^{-1}(C(x) - t)),$$

with C^{-1} being the inverse function of $C = \int c^1 dx$. Since $C(x)$ is an integral $\int_0^x (1/c)dx$, $C(x)$ is monotonically increasing and there exists hence an inverse function C^{-1} with values in $[0, L]$.

Solution. In general we have $u(x, t) = f(C(x) - t)$ and the solution is of this form with $f(\xi) = I(C^{-1}(\xi))$. Moreover, at $t = 0$ we have $I(C^{-1}(C(x))) = I(x)$, which is the required initial condition.

To compute (205) we need to integrate $1/c$ to obtain C and then compute the inverse of C.

The inverse function computation can be easily done if we first think discretely. Say we have some function $y = g(x)$ and seek its inverse. Plotting (x_i, y_i), where $y_i = g(x_i)$ for some mesh points x_i, displays g as a function of x. The inverse function is simply x as a function of g, i.e., the curve with points (y_i, x_i). We can therefore quickly compute points at the curve of the inverse function. One way of extending these points to a continuous function is to assume a linear variation (known as linear interpolation) between the points (which actually means to draw straight lines between the points, exactly as done by a plotting program).

The function `wrap2callable` in `scitools.std` can take a set of points and return a continuous function that corresponds to linear variation between the points. The computation of the inverse of a function g on $[0, L]$ can then be done by

```
def inverse(g, domain, resolution=101):
    x = linspace(domain[0], domain[L], resolution)
    y = g(x)
    from scitools.std import wrap2callable
    g_inverse = wrap2callable((y, x))
    return g_inverse
```

To compute $C(x)$ we need to integrate $1/c$, which can be done by a Trapezoidal rule. Suppose we have computed $C(x_i)$ and need to compute $C(x_{i+1})$. Using the Trapezoidal rule with m subintervals over the integration domain $[x_i, x_{i+1}]$ gives

$$C(x_{i+1}) = C(x_i) + \int_{x_i}^{x_{i+1}} \frac{dx}{c} \approx h \left(\frac{1}{2} \frac{1}{c(x_i)} + \frac{1}{2} \frac{1}{c(x_{i+1})} + \sum_{j=1}^{m-1} \frac{1}{c(x_i + jh)} \right),$$
(206)

where $h = (x_{i+1} - x_i)/m$ is the length of the subintervals used for the integral over $[x_i, x_{i+1}]$. We observe that (206) is a *difference equation* which we can solve by repeatedly applying (206) for $i = 0, 1, \ldots, N_x - 1$ if a mesh $x_0, x, \ldots, x_{N_x}$ is prescribed. Note that $C(0) = 0$.

j) Implement a function for computing $C(x_i)$ and one for computing $C^{-1}(x)$ for any x. Use these two functions for computing the exact solution $I(C^{-1}(C(x)-t))$. End up with a function `u_exact_variable_c(x, n, c, I)` that returns the value of $I(C^{-1}(C(x) - t_n))$.

k) Make movies showing a comparison of the numerical and exact solutions for the two initial conditions (203) and (30). Choose $\Delta t = \Delta x / \max_{0,L} c(x)$ and the velocity of the medium as

1. $c(x) = 1 + \epsilon \sin(k\pi x/L)$, $\epsilon < 1$,

2. $c(x) = 1 + I(x)$, where I is given by (203) or (30).

The PDE $u_t + cu_x = 0$ expresses that the initial condition $I(x)$ is transported with velocity $c(x)$.

Filename: `advec1D`.

Problem 31: General analytical solution of a 1D damped wave equation

We consider an initial-boundary value problem for the damped wave equation:

$$
\begin{aligned}
u_{tt} + bu_t &= c^2 u_{xx}, && x \in (0, L), \ t \in (0, T] \\
u(0, t) &= 0, \\
u(L, t) &= 0, \\
u(x, 0) &= I(x), \\
u_t(x, 0) &= V(x).
\end{aligned}
$$

133

Here, $b \geq 0$ and c are given constants. The aim is to derive a general analytical solution of this problem. Familiarity with the method of separation of variables for solving PDEs will be assumed.

a) Seek a solution on the form $u(x,t) = X(x)T(t)$. Insert this solution in the PDE and show that it leads to two differential equations for X and T:

$$T'' + bT' + \lambda T = 0, \quad c^2 X'' + \lambda X = 0,$$

with $X(0) = X(L) = 0$ as boundary conditions, and λ as a constant to be determined.

b) Show that $X(x)$ is on the form

$$X_n(x) = C_n \sin kx, \quad k = \frac{n\pi}{L}, \quad n = 1, 2, \ldots$$

where C_n is an arbitrary constant.

c) Under the assumption that $(b/2)^2 < k^2$, show that $T(t)$ is on the form

$$T_n(t) = e^{-\frac{1}{2}bt}(a_n \cos \omega t + b_n \sin \omega t), \quad \omega = \sqrt{k^2 - \frac{1}{4}b^2}, \quad n = 1, 2, \ldots$$

The complete solution is then

$$u(x,t) = \sum_{n=1}^{\infty} \sin kx e^{-\frac{1}{2}bt}(A_n \cos \omega t + B_n \sin \omega t),$$

where the constants A_n and B_n must be computed from the initial conditions.

d) Derive a formula for A_n from $u(x,0) = I(x)$ and developing $I(x)$ as a sine Fourier series on $[0, L]$.

e) Derive a formula for B_n from $u_t(x,0) = V(x)$ and developing $V(x)$ as a sine Fourier series on $[0, L]$.

f) Calculate A_n and B_n from vibrations of a string where $V(x) = 0$ and

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{207}$$

g) Implement the series for $u(x,t)$ in a function `u_series(x, t, tol=1E-10)`, where `tol` is a tolerance for truncating the series. Simply sum the terms until $|a_n|$ and $|b_b|$ both are less than `tol`.

h) What will change in the derivation of the analytical solution if we have $u_x(0,t) = u_x(L,t) = 0$ as boundary conditions? And how will you solve the problem with $u(0,t) = 0$ and $u_x(L,t) = 0$?
Filename: `damped_wave1D`.

Problem 32: General analytical solution of a 2D damped wave equation

Carry out Problem 31 in the 2D case: $u_{tt} + bu_t = c^2(u_{xx} + u_{yy})$, where $(x, y) \in (0, L_x) \times (0, L_y)$. Assume a solution on the form $u(x, y, t) = X(x)Y(y)T(t)$. Filename: `damped_wave2D`.

References

[1] H. P. Langtangen. Scientific software engineering; wave equation case. `http://tinyurl.com/k3sdbuv/pub/softeng2`.

[2] H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models.* Lecture Notes in Computational Science and Engineering. Springer, 2016. `http://tinyurl.com/nclmcng/web`.

[3] H. P. Langtangen and G. K. Pedersen. *Scaling of Differential Equations.* Simula Springer Brief Series. Springer, 2016. `http://tinyurl.com/qfjgxmf/web`.

Index