Study Guide: Finite difference methods for wave motion

Hans Petter Langtangen[1,2]

Center for Biomedical Computing, Simula Research Laboratory[1]

Department of Informatics, University of Oslo[2]

Sep 18, 2013

Finite difference methods for waves on a string

Waves on a string can be modeled by the *wave equation*

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

$u(x, t)$ is the displacement of the string

Initial-boundary value problem

$$
\begin{aligned}
u_{tt} &= c^2 u_{xx}, \quad x\in(0,L),\ t\in(0,T] && (1)\\
u(x,0) &= I(x), \quad x\in[0,L] && (2)\\
u_t(x,0) &= 0, \quad x\in[0,L] && (3)\\
u(0,t) &= 0, \quad t\in(0,T], && (4)\\
u(L,t) &= 0, \quad t\in(0,T]\,. && (5)
\end{aligned}
$$

Input data in the problem

- Initial condition $u(x,0) = I(x)$: initial string shape
- Initial condition $u_t(x,0) = 0$: string starts from rest
- $c = \sqrt{T/\varrho}$: velocity of waves on the string
- (T is the tension in the string, ϱ is density of the string)
- Two boundary conditions on u: $u = 0$ means fixed ends (no displacement)

Rule for no of initial and boundary conditions:

- u_{tt} in the PDE: two initial conditions, on u and u_t
- u_t, not u_{tt}, in the PDE: one initial conditions, on u
- u_{xx} in the PDE: one boundary condition on u at each boundary point

Demo of a vibrating string ($C = 0.8$)

- Our numerical method is sometimes exact (!)
- Our numerical method is sometimes subject to serious non-physical effects

Demo of a vibrating string ($C = 1.0012$)

Ooops!

Step 1: Discretizing the domain

Mesh in time:

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{6}$$

Mesh in space:

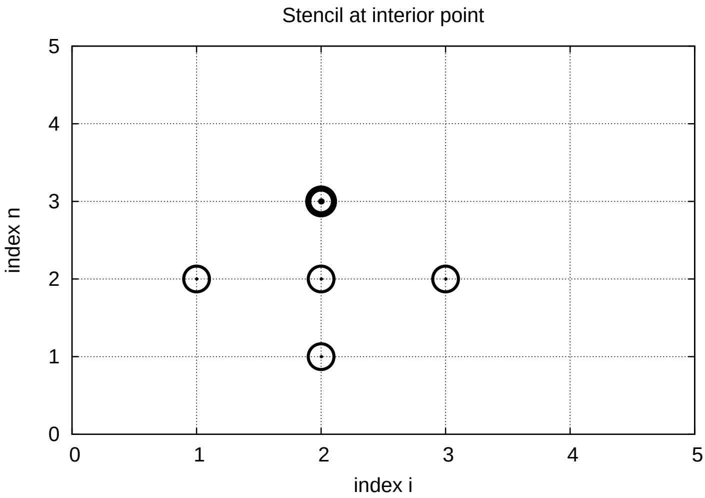
$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L\,. \tag{7}$$

Uniform mesh with constant mesh spacings Δt and Δx:

$$x_i = i\Delta x,\; i = 0,\ldots,N_x, \quad t_i = n\Delta t,\; n = 0,\ldots,N_t\,. \tag{8}$$

The discrete solution

- The numerical solution is a mesh function: $u_i^n \approx u_{\rm e}(x_i, t_n)$
- Finite difference stencil (or scheme): equation for u_i^n involving neighboring space-time points



Step 2: Fulfilling the equation at the mesh points

Let the PDE be satisfied at all *interior* mesh points:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \tag{9}$$

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 $u = 0$.

Step 3: Replacing derivatives by finite differences

Widely used finite difference formula for the second-order derivative:

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

and

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

Step 3: Algebraic version of the PDE

Replace derivatives by differences:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{10}$$

In operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n . \tag{11}$$

Step 3: Algebraic version of the initial conditions

- Need to replace the derivative in the initial condition $u_t(x, 0) = 0$ by a finite difference approximation
- The differences for u_{tt} and u_{xx} have second-order accuracy
- Use a centered difference for $u_t(x, 0)$

$$[D_{2t}u]^n_i = 0, \quad n = 0 \quad \Rightarrow \quad u^{n-1}_i = u^{n+1}_i, \quad i = 0, \ldots, N_x$$

The other initial condition $u(x, 0) = I(x)$ can be computed by

$$u^0_i = I(x_i), \quad i = 0, \ldots, N_x$$

Step 4: Formulating a recursive algorithm

- Nature of the algorithm: compute u in space at $t = \Delta t, 2\Delta t, 3\Delta t, ...$
- Three time levels are involved in the general discrete equation: $n+1$, n, $n-1$
- u_i^n and u_i^{n-1} are then already computed for $i = 0, \ldots, N_x$, and u_i^{n+1} is the unknown quantity

Write out $[D_t D_t u = c^2 D_x D_x]_i^n$ and solve for u_i^{n+1},

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) \tag{12}$$

The Courant number

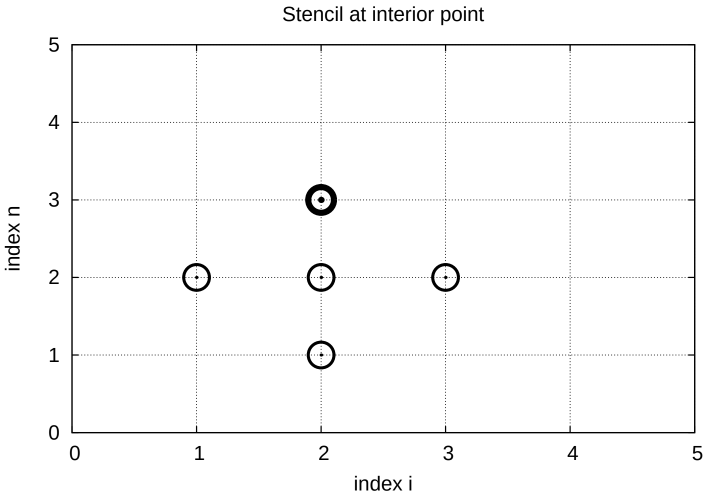
$$C = c\frac{\Delta t}{\Delta x}, \tag{13}$$

is known as the (dimensionless) *Courant number*

Notice.

There is only one parameter, C, in the discrete model: C lumps mesh parameters with the wave velocity c. The value C and the smoothness of $I(x)$ govern the quality of the numerical solution.

The finite difference stencil



The stencil for the first time level

- Problem: the stencil for $n = 1$ involves u_i^{-1}, but time $t = -\Delta t$ is outside the mesh
- Remedy: use the initial condition $u_t = 0$ together with the stencil to eliminate u_i^{-1}

Initial condition:

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

Insert in stencil $[D_tD_tu = c^2D_xD_x]_i^0$ to get

$$u_i^1 = u_i^0 - \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) . \tag{14}$$

The algorithm

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. Compute u_i^1 by (14) 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 $n = 1, 2, \ldots, N_t - 1$
    1. apply (12) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
    2. set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

Moving finite difference stencil

web page or a movie file.

Sketch of an implementation (1)

- Arrays:
  - `u[i]` stores u_i^{n+1}
  - `u_1[i]` stores u_i^n
  - `u_2[i]` stores u_i^{n-1}

Naming convention.

`u` is the unknown to be computed (a spatial mesh function), `u_k` is the computed spatial mesh function `k` time steps back in time.

PDE solvers should save memory

Important to minimize the memory usage.

The algorithm only needs to access 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 not in large 2D problems and not even in small 3D problems.

Sketch of an implementation (2)

```
# 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
```

Verification

- Think about testing and verification before you start implementing the algorithm!
- Powerful testing tool: method of manufactured solutions and computation of convergence rates
- Will need a source term in the PDE and $u_t(x, 0) \neq 0$
- Even more powerful method: exact solution of the scheme

A slightly generalized model problem

Add source term f and nonzero initial condition $u_t(x, 0)$:

$$
\begin{aligned}
u_{tt} &= c^2 u_{xx} + f(x,t), && (15)\\
u(x,0) &= I(x), \quad x \in [0, L] && (16)\\
u_t(x,0) &= V(x), \quad x \in [0, L] && (17)\\
u(0,t) &= 0, \quad t > 0, && (18)\\
u(L,t) &= 0, \quad t > 0\,. && (19)
\end{aligned}
$$

Discrete model for the generalized model problem

$$[D_t D_t u = c^2 D_x D_x + f]^n_i \,. \tag{20}$$

Writing out and solving for the unknown u^{n+1}_i:

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + C^2(u^n_{i+1} - 2u^n_i + u^n_{i-1}) + \Delta t^2 f^n_i \,. \tag{21}$$

Modified equation for the first time level

Centered difference for $u_t(x,0) = V(x)$:

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

which, when inserted in the stencil (21) for $n = 0$, gives

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2}C^2\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right) + \frac{1}{2}\Delta t^2 f^n_i\,. \qquad (22)$$

Using an analytical solution of physical significance

- Standing waves occur in real life on a string
- Can be analyzed mathematically (known exact solution)

$$u_e(x, y, t)) = A\sin\left(\frac{\pi}{L}x\right)\cos\left(\frac{\pi}{L}ct\right) . \tag{23}$$

- PDE data: $f = 0$, boundary conditions $u_e(0, t) = u_e(L, 0) = 0$, initial conditions $I(x) = A\sin\left(\frac{\pi}{L}x\right)$ and $V = 0$
- Note: $u_i^{n+1} \neq u_e(x_i, t_{n+1}$, and we do not know the error, so testing must aim at reproducing the expected convergence rates

Manufactured solution: principles

- Disadvantage with the previous physical solution: it does not test $V \neq 0$ and $f \neq 0$
- Method of manufactured solution:
  - Choose some $u_{\rm e}(x, t)$
  - Insert in PDE and fit f
  - Set boundary and initial conditions compatible with the chosen $u_{\rm e}(x, t)$

Manufactured solution: example

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

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

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

Initial conditions become

$$\begin{aligned} u(x,0) &= I(x) = 0,\\ u_t(x,0) &= V(x) = (2 - x(L-x))\cos t\,. \end{aligned}$$

Boundary conditions:

$$u(x,0) = u(x,L) = 0\,.$$

Testing a manufactured solution

- Introduce common mesh parameter: $h = \Delta t$, $\Delta x = ch/C$
- This h keeps C and $\Delta t/\Delta x$ constant
- Select coarse mesh h: h_0
- Run experiments with $h_i = 2^{-i}h_0$ (halving the cell size), $i = 0, \ldots, m$
- Record the error E_i and h_i in each experiment
- Compute pariwise convergence rates $r_i = \ln E_{i+1}/E_i/\ln h_{i+1}/h_i$
- Verification: $r_i \rightarrow 2$ as i increases

Constructing an exact solution of the discrete equations

- Manufactured solution with computation of convergence rates: significant manual work
- Simpler and more powerful: use an exact solution for u_i^n
- A linear or quadratic u_{e} in x and t is often a good candidate

Analytical work with the PDE problem

Here, choose u_e such that $u_e(x, 0) = u_e(L, 0) = 0$:

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

Insert in the PDE and find f:

$$f(x, t) = 2(1 + t)c^2 .$$

Initial conditions:

$$I(x) = x(L - x), \quad V(x) = \frac{1}{2}x(L - x) .$$

Analytical work with the discrete equations (1)

We want to show that u_{e} also solves the discrete equations!
Useful preliminary result:

$$[D_tD_tt^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - n^2 + (n-1)^2 = 2, \tag{24}$$

$$[D_tD_tt]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - n + (n-1))\Delta t}{\Delta t^2} = 0\,. \tag{25}$$

Hence,

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

Analytical work with the discrete equations (1)

$$
\begin{aligned}
[D_xD_xu_{\rm e}]^n_i &= (1+\frac{1}{2}t_n)[D_xD_x(xL-x^2)]_i = (1+\frac{1}{2}t_n)[LD_xD_xx - D_xD_xx^2\\
&= -2(1+\frac{1}{2}t_n)\,.
\end{aligned}
$$

Now, $f^n_i = 2(1+\frac{1}{2}t_n)c^2$ and we get

$$
[D_tD_tu_{\rm e} - c^2D_xD_xu_{\rm e} - f]^n_i = 0 - c^2(-1)2(1+\frac{1}{2}t_n + 2(1+\frac{1}{2}t_n)c^2 = 0\,.
$$

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

Testing with the exact discrete solution

- We have established that $u_i^{n+1} = u_{\mathrm{e}}(x_i, t_{n+1}) = x_i(L - x_i)(1 + t_{n+1}/2)$
- Run *one* simulation with one choice of c, Δt, and Δx
- Check that $\max_i |u_i^{n+1} - u_{\mathrm{e}}(x_i, t_{n+1})| < \epsilon$, $\epsilon \sim 10^{-14}$ (machine precision + some round-off errors)
- This is the simplest and best verification test

Later we show that the exact solution of the discrete equations can be obtained by $C = 1$ (!)

Testing with the exact discrete solution

- We have established that $u_i^{n+1} = u_e(x_i, t_{n+1}) = x_i(L - x_i)(1 + t_{n+1}/2)$
- Run *one* simulation with one choice of c, Δt, and Δx
- Check that $\max_i |u_i^{n+1} - u_e(x_i, t_{n+1})| < \epsilon$, $\epsilon \sim 10^{-14}$ (machine precision + some round-off errors)
- This is the simplest and best verification test

Later we show that the exact solution of the discrete equations can be obtained by $C = 1$ (!)

Implementation

The algorithm

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. Compute u_i^1 by (14) 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 $n = 1, 2, \ldots, N_t - 1$
   1. apply (12) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
   2. set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

What do to with the solution?

- Different problem settings demand different actions with the computed u_i^{n+1} at each time step
- Solution: let the solver function make a callback to a user function where the user can do whatever is desired with the solution
- Advantage: solver just solves and user uses the solution

```
def user_action(u, x, t, n):
    # u[i] at spatial mesh points x[i] at time t[n]
    # plot u
    # or store u
```

Making a solver function (1)

```python
def solver(I, V, f, c, L, Nx, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    x = linspace(0, L, 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                     # Help variable in the scheme
    if f is None or f == 0 :
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    u   = 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

    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)
```

Making a solver function (2)

```
def solver(I, V, f, c, L, Nx, C, T, user_action=None):
    ...
    # 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


===== Making a solver function (3) =====

\begin{minted}[fontsize=\fontsize{9pt}{9pt},linenos=false,mathescap
def solver(I, V, f, c, L, Nx, C, T, user_action=None):
    ...
    # Time loop

    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] + \
```

Verification: exact quadratic solution

Exact solution of the PDE problem *and* the discrete equations:

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

```python
import nose.tools as nt

def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""
    def exact_solution(x, t):
        return x*(L-x)*(1 + 0.5*t)

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

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

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

    L = 2.5
    c = 1.5
    Nx = 3  # Very coarse mesh
    C = 0.75
    T = 18

    u, x, t, cpu = solver(I, V, f, c, L, Nx, C, T)
    u_e = exact_solution(x, t[-1])
    diff = abs(u - u_e).max()
```

Visualization: animating $u(x, t)$

Make a viz function for animating the curve, with plotting in a `user_action` function `plot_u`:

```
def viz(I, V, f, c, L, Nx, C, T, umin, umax, animate=True):
    """Run solver and visualize u at each time level."""
    import scitools.std as plt
    import 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=[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

    # 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, L, Nx, C, T, user_action)

    # Make movie files
    fps = 4  # Frames per second
```

Making movie files

- Store spatial curve in a file, for each time level
- Name files like `'something_%04d.png' % frame_counter`
- Combine files to a movie

```
Terminal> scitools movie encoder=html output_file=movie.html \
          fps=4 frame_*.png  # web page with a player
Terminal> avconv -r 4 -i frame_%04d.png -vcodec flv       movie.flv
Terminal> avconv -r 4 -i frame_%04d.png -vcodec libtheora movie.ogg
Terminal> avconv -r 4 -i frame_%04d.png -vcodec libx264   movie.mp4
Terminal> avconv -r 4 -i frame_%04d.png -vcodec libtheora movie.ogg
Terminal> avconv -r 4 -i frame_%04d.png -vcodec libpvx    movie.webm
```

Important.

- Zero padding (`%04d`) is essential for correct sequence of frames in `something_*.png` (Unix alphanumeric sort)
- Remove old `frame_*.png` files before making a new movie

Running a case

- Vibrations of a guitar string
- Triangular initial shape (at rest)

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{26}$$

Appropriate data:

- $L = 75$ cm, $x_0 = 0.8L$, $a = 5$ mm, $N_x = 50$, time frequency $\nu = 440$ Hz

Implementation of the case

```python
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
    Nx = 50

    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, Nx, C, T, umin, umax, animate=True)
```

Program: `wave1D_u0_s.py`.

Resulting movie for $C = 0.8$

Movie of the vibrating string

The benefits of scaling

- It is difficult to figure out all the physical parameters of a case
- And it is not necessary because of a powerful: *scaling*

Introduce new x, t, and u without dimension:

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

Insert this in the PDE ($with f = 0$) and dropping bars

$$u_{tt} = u_{tt}$$

Initial condition: set $a = 1$, $L = 1$, and $x_0 \in [0, 1]$ in (26).
In the code: set `a=c=L=1`, `x0=0.8`, and there is no need to calculate with wavelengths and frequencies to estimate c!
Just one challenge: determine the period of the waves and an appropriate end time (see the text for details).

Vectorization

- Problem: Python loops over long arrays are slow
- One remedy: use vectorized (`numpy`) code instead of explicit loops
- Other remedies: use Cython, port spatial loops to Fortran or C
- Speedup: 100-1000 (varies with N_x)

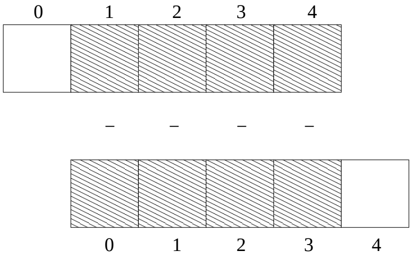
Next: vectorized loops

Operations on slices of arrays

- Introductory example: compute $d_i = u_{i+1} - u_i$

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

- Note: all the differences here are independent of each other.
- Therefore $d = (u_1, u_2, \ldots, u_n) - (u_0, u_1, \ldots, u_{n-1})$
- In numpy code: `u[1:n] - u[0:n-1]` or just `u[1:] - u[:-1]`



Test the 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.

Vectorization of finite difference schemes (1)

Finite difference schemes basically contains differences between array elements with shifted indices. Consider the updating formula

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

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

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

Note: `u2` gets length `n-2`.
If `u2` is already an array of length `n`, do update on "inner" elements

```
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
```

Vectorization of finite difference schemes (2)

Include a function evaluation too:

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

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

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

Vectorized implementation in the solver function

Scalar 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])
```

Vectorized loop:

```
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])
```

Program: `wave1D_u0_sv.py`

Verification of the vectorized version

```python
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    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 is a scalar (zeros_like(x) works for scalar x too)
    f = lambda x, t: zeros_like(x) + 2*c**2*(1 + 0.5*t)

    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, L, Nx, C, T,
           user_action=assert_no_error, version='scalar')
    solver(I, V, f, c, L, Nx, C, T,
           user_action=assert_no_error, version='vectorized')
```

Efficiency measurements

- Run `wave1D_u0_sv.py` for $N_x = 50, 100, 200, 400, 800$ and measuring the CPU time (cf. `run_efficiency_experiments` function)
- Observe substantial speed-up: vectorized version is about $N_x/5$ times faster

Much bigger improvements for 2D and 3D codes!

Generalization: reflecting boundaries

- Boundary condition $u = 0$: u changes sign
- Boundary condition $u_x = 0$: wave is perfectly reflected
- How can we implement u_x?
- It is more complicated than $u = 0$

Neumann boundary condition

$$\frac{\partial u}{\partial n} \equiv \mathbf{n} \cdot \nabla u = 0\,. \tag{27}$$

For a 1D domain $[0, L]$:

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

Boundary condition terminology:

- u_x specified: Neumann condition
- u specified: Dirichlet condition

Discretization of derivatives at the boundary (1)

- How can we incorporate the condition $u_x = 0$ in the finite difference scheme?
- We used centeral differences for u_{tt} and u_{xx}: $\mathcal{O}(\Delta t^2, \Delta x^2)$ accuracy
- Also for $u_t(x, 0)$
- Should use central difference for u_x to preserve second order accuracy

$$\frac{u^n_{-1} - u^n_1}{2\Delta x} = 0\,. \tag{28}$$

Discretization of derivatives at the boundary (2)

$$\frac{u_{-1}^n - u_1^n}{2\Delta x} = 0$$

- Problem: u_{-1}^n is outside the mesh (fictitious value)
- Remedy: use the stencil at the boundary to eliminate u_{-1}^n; just replace u_{-1}^n by u_1^n

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u_{i+1}^n - u_i^n\right), \quad i = 0\,. \tag{29}$$

Visualization of modified boundary stencil

Discrete equation for computing u_0^3 in terms of u_0^2, u_0^1, and u_1^2:
Animation in a web page or a movie file.

Implementation of Neumann conditions

- Use the general stencil for interior points also on the boundary
- Replace u^n_{i-1} by u^n_{i+1} for $i = 0$
- Replace u^n_{i+1} by u^n_{i-1} for $i = N_x$

```
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])

# Or just one loop over all points

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])
```

Program `wave1D_dn0.py`

Index set notation

- Tedious to write index sets like $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$
- Notation not valid if i or n starts at 1 instead...
- Both in math and code it is advantageous to use *index sets*
- $i \in \mathcal{I}_x$ instead of $i = 0, \ldots, N_x$
- Definition: $\mathcal{I}_x = \{0, \ldots, N_x\}$
- The first index: $i = \mathcal{I}_x^0$
- The last index: $i = \mathcal{I}_x^{-1}$
- All interior points: $i \in \mathcal{I}_x^+$, $\mathcal{I}_x^i = \{1, \ldots, N_x - 1\}$
- $\mathcal{I}_x^-$ means $\{0, \ldots, N_x - 1\}$
- $\mathcal{I}_x^+$ means $\{1, \ldots, N_x\}$

Index set notation in code

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]`

Index sets in action (1)

Index sets for a problem in the x, t plane:

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\}, \tag{30}$$

defined in Python as

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

Index sets in action (2)

A finite difference scheme can with the index set notation be specified as

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

and implemented by code like

```
for n in It[1:-1]:
    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])
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Program `wave1D_dn.py`

Alternative implementation via ghost cells

- Instead of modifying the stencil at the boundary, we extend the mesh to cover u^n_{-1} and $u^n_{N_x+1}$
- The extra left and right cell are called *ghost cells*
- The extra points are called *ghost points*
- The u^n_{-1} and $u^n_{N_x+1}$ values are called *ghost values*

The important idea is to ensure that

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

because then the stencil becomes right at the boundary.

Implementation of ghost cells (1)

Add ghost points:

```
u   = zeros(Nx+3)
u_1 = zeros(Nx+3)
u_2 = zeros(Nx+3)

x = linspace(0, L, Nx+1)  # Mesh points without ghost points
```

- A major indexing problem arises with ghost cells since Python indices *must* start at 0.
- `u[-1]` will always mean the last element in `u`
- Math indexing: $-1, 0, 1, 2, \ldots, N_x + 1$
- Python indexing: `0,..,Nx+2`
- Remedy: use index sets

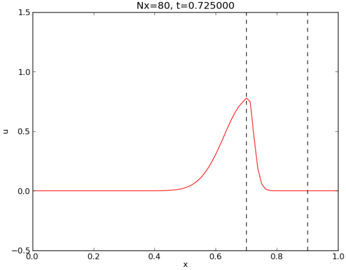
Implementation of ghost cells (2)

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

# Boundary values: u[Ix[0]], u[Ix[-1]]

# Set initial conditions
for i in Ix:
    u_1[i] = I(x[i-Ix[0]])  # Note i-Ix[0]

# Loop over all physical mesh points
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])

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

Program: `wave1D_dn0_ghost.py`.

Generalization: variable wave velocity

Heterogeneous media: varying $c = c(x)$



The model PDE with a variable coefficient

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{31}$$

This equation sampled at a mesh point (x_i, t_n):

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

Discretizing the variable coefficient (1)

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:

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

Discretizing the variable coefficient (2)

Then discretize the inner operators:

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

Similarly,

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

Discretizing the variable coefficient (3)

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right). \tag{32}$$

In operator notation:

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_x q D_x u]_i^n. \tag{33}$$

Remark.

Remark. 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 in this context.

Computing the coefficient between mesh points

- Given $q(x)$: compute $q_{i+\frac{1}{2}}$ as $q(x_{i+\frac{1}{2}})$
- Given q at the mesh points: q_i, use an average

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}(q_i + q_{i+1}) = [\overline{q}^x]_i, \qquad \text{(arithmetic mean)} \qquad (34)$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1}, \qquad \text{(harmonic mean)} \qquad (35)$$

$$q_{i+\frac{1}{2}} \approx (q_i q_{i+1})^{1/2}, \qquad \text{(geometric mean)} \qquad (36)$$

The arithmetic mean in (34) is by far the most used averaging technique.

Discretization of variable-coefficient wave equation in operator notation

$$[D_tD_t u = D_x\overline{q}^{x}D_x u + f]^n_i \,. \tag{37}$$

We clearly see the type of finite differences and averaging!
Write out and solve wrt u_i^{n+1}:

$$\begin{aligned}
u^{n+1}_i &= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 \times\\
&\quad \left(\frac{1}{2}(q_{i} + q_{i+1})(u_{i+1}^n - u_{i}^n) - \frac{1}{2}(q_{i} + q_{i-1})(u_{i}^n - u_{i-1}^n)\right) + \\
&\quad \Delta t^2 f^n_i \,.
\end{aligned} \tag{38}$$