Wave project, INF5620

Arnfinn Aamodt

October 25, 2014

1 The core parts of the project

1.1 Mathematical problem

The project addresses the two-dimensional, standard, linear wave equation, with damping,

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

subject to the homogenous von Neumann boundary condition,

$$\frac{\partial u}{\partial n} = 0$$

in a rectangular spatial domain $\Omega = [0, L_x] \times [0, L_y]$. The initial conditions are

$$u(x,y,0) = I(x,y), \quad u_t(x,y,0) = V(x,y).$$

1.2 Discretization

We discretize the time derivatives as

$$\frac{\partial^2 u}{\partial t^2} \approx [D_t D_t u]^n_{i,j} = \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2},$$

$$\frac{\partial u}{\partial t} \approx [D_{2t} u]^n_{i,j} = \frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2\Delta t}.$$

The spatial derivatives are discretized as

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

$$\frac{\partial u}{\partial y}\left(q\frac{\partial u}{\partial y}\right) \approx [D_y q D_y u]^n_{i,j} = \frac{1}{\Delta y^2}\left(q_{i,j+\frac{1}{2}}(u^n_{i,j+1} - u^n_{i,j}) - q_{i,j-\frac{1}{2}}(u^n_{i,j} - u^n_{i,j-1})\right).$$

Substituting for the corresponding terms in the differential equations we get thee following difference equation

$$\begin{aligned}
&\frac{u_{i,j}^{n+1} - 2u_{i,j}^{n} + u_{i,j}^{n-1}}{\Delta t^2} + b\frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} \\
&\quad = \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2},j}(u_{i+1,j}^{n} - u_{i,j}^{n}) - q_{i-\frac{1}{2},j}(u_{i,j}^{n} - u_{i-1,j}^{n})\right) \\
&\quad + \frac{1}{\Delta y^2}\left(q_{i,j+\frac{1}{2}}(u_{i,j+1}^{n} - u_{i,j}^{n}) - q_{i,j-\frac{1}{2}}(u_{i,j}^{n} - u_{i,j-1}^{n})\right) + f_{i,j}^{n}.
\end{aligned}$$

Multiplying through by Δt^2, collecting the $u_{i,j}^{n+1}$-terms on the left side, dividing by the factor $1 + \frac{b}{2}\Delta t$ and rearranging gives us

$$\begin{aligned}
u_{i,j}^{n+1} &= (1 + \frac{b}{2}\Delta t)^{-1}\Big[2u_{i,j}^{n} - (1 - \frac{b}{2}\Delta t)u_{i,j}^{n-1} + f_{i,j}^{n}\Delta t^2 \\
&+ \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2},j}(u_{i+1,j}^{n} - u_{i,j}^{n}) - q_{i-\frac{1}{2},j}(u_{i,j}^{n} - u_{i-1,j}^{n})\right) \\
&+ \left(\frac{\Delta t}{\Delta y}\right)^2 \left(q_{i,j+\frac{1}{2}}(u_{i,j+1}^{n} - u_{i,j}^{n}) - q_{i,j-\frac{1}{2}}(u_{i,j}^{n} - u_{i,j-1}^{n})\right)\Big].
\end{aligned}$$

For the first time step, and at the boundaries, special care must be taken, since the scheme at these points asks for values of u outside the domain. The trick is to use the boundary conditions to express these values of u in terms of values inside the domain.

For the first time step we need an expression for the value of $u_{i,j}^{-1}$. Discretizing the boundary condition $u_t(x, y, 0) = V(x, y)$, we get

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

which gives

$$u_{i,j}^{-1} = u_{i,j}^{1} - 2V_{i,j}\Delta t.$$

Inserting $u_{i,j}^{n-1} = u_{i,j}^{n+1} - 2V_{i,j}\Delta t$ into the general scheme above, we obtain a modified scheme for $n = 0$, namely

$$\begin{aligned}
u_{i,j}^{n+1} &= \frac{1}{2}\Big[2u_{i,j}^{n} - (1 - \frac{b}{2}\Delta t)(-2V_{i,j}\Delta t) + f_{i,j}^{n}\Delta t^2 \\
&+ \left(\frac{\Delta t}{\Delta x}\right)^2 \left(q_{i+\frac{1}{2},j}(u_{i+1,j}^{n} - u_{i,j}^{n}) - q_{i-\frac{1}{2},j}(u_{i,j}^{n} - u_{i-1,j}^{n})\right) \\
&+ \left(\frac{\Delta t}{\Delta y}\right)^2 \left(q_{i,j+\frac{1}{2}}(u_{i,j+1}^{n} - u_{i,j}^{n}) - q_{i,j-\frac{1}{2}}(u_{i,j}^{n} - u_{i,j-1}^{n})\right)\Big].
\end{aligned}$$

Note that the schemes for both first and subsequent steps can be incorporated into the scheme

$$\begin{aligned} u_{i,j}^{n+1} = B\Big[& 2u_{i,j}^n - (1 - \frac{b}{2}\Delta t)(D_1 u_{i,j}^{n-1} - D_2 V_{i,j}) + f_{i,j}^n \Delta t^2 \\ & + \left(\frac{\Delta t}{\Delta x}\right)^2 \Big(q_{i+\frac{1}{2},j}(u_{i+1,j}^n - u_{i,j}^n) - q_{i-\frac{1}{2},j}(u_{i,j}^n - u_{i-1,j}^n) \Big) \\ & + \left(\frac{\Delta t}{\Delta y}\right)^2 \Big(q_{i,j+\frac{1}{2}}(u_{i,j+1}^n - u_{i,j}^n) - q_{i,j-\frac{1}{2}}(u_{i,j}^n - u_{i,j-1}^n) \Big) \Big], \end{aligned}$$

by adjusting B, D_1 and D_2 according to the current time step. This is the scheme that we will use to compute $u_{i,j}^{n+1}$ at all interior points. We will approximate the values of q between mesh points by the arithmetic mean, and we will assume

$$\frac{\partial q}{\partial n} = 0,$$

as discussed in the lecture notes.

At the boundaries we use the von Neumann boundary conditions $\frac{\partial u}{\partial n} = 0$, which gives us

$$u_{i-1,j}^n = u_{i+1,j}^n, \qquad i = 0, \quad i = N_x,$$

and

$$u_{i,j-1}^n = u_{i,j+1}^n, \qquad j = 0, \quad j = N_y.$$

Rather than inserting these relations into our general scheme, we will just substitute the indices at the boundaries accordingly in our implementation, thus achieving the same effect.

To speed up computations, we would like to use vector computations for each time step, to avoid looping over the spatial parameters i and j. For the inner points in the spatial domain we can easily modify the unified scheme above. The remaining question is what to do with the boundary conditions. The lecture notes suggest either using ghost cells, or updating cells on the boundary separately. We will handle this problem slightly differently. Instead of including ghost cells permanently, we will just extend u to include values outside the spatial domain whenever this is needed. For example, when the scheme calls for $u_{i-1,j}^n$ for all $i \in I_x$, $j \in I_y$, we will just vertically concatenate $u[1,:]$ and $u[:-1,:]$, which amounts to temporarily filling the cells at $i = -1$ with the values of u at $i = 1$, just like the boundary conditions require.

1.3 Implementation

An implementation of the scheme discussed in the previous section is given below (see the file `wave_project.py`). A nose test comparing the numerical solution to an exact constant solution is also defined, using the function `user_action` which is an optional parameter to the solver, executed at each time step if present. This test will be discussed in more detail in the next section.

```
"""
Solve the equation
    u_tt + bu_t = (q(x,y)u_x)_x + (q(x,y)u_y)_y + f(x,y,t)
for initial conditions
    u(x,y,0) = I(x,y)
    u_x(x,y,0) = V(x,y)
and homogenous von Neuman boundary conditions (du/dn=0) vy finite differences.
"""
from numpy import *
import matplotlib.pyplot as plt
import time


def solver(I, V, f, q, b, Lx, Ly, Nx, Ny, dt, T, user_action=None,
           version='scalar'):

    t0 = time.clock()

    x  = linspace(0, Lx, Nx+1)  # spatial mesh
    y  = linspace(0, Ly, Ny+1)
    xv = x[:,newaxis]            # for vector operations
    yv = y[newaxis,:]
    dx = x[1] - x[0]
    dy = y[1] - y[0]

    Nt = int(round(T/float(dt)))     #time discretization
    t  = linspace(0, Nt*dt, Nt+1)

    b = float(b);        A = 1/(1+b/2*dt)    # help variables
    Cx2 = (dt/dx)**2;   Cy2 = (dt/dy)**2

    u   = zeros((Nx+1, Ny+1))   # solution arrays
    u_1 = zeros((Nx+1, Ny+1))
    u_2 = zeros((Nx+1, Ny+1))

    Ix = range(0, u.shape[0])    # index sets
    Iy = range(0, u.shape[1])
    It = range(0, t.shape[0])

    if version == 'scalar':
        for i in Ix:
            for j in Iy:
                u_1[i,j] = I(x[i], y[j])
    else:
        u_1[:,:]  = I(xv,yv)
        V_a = V(xv, yv) #Evalute at startup for vectorized version
        q_a = q(xv, yv)
        #These will be used for the incrementation step.
        q_a_px = 0.5*(concatenate((q_a[1:,:], q_a[-2,:][None,:]),axis=0) + q_a)
        q_a_mx = 0.5*(q_a + concatenate((q_a[1,:][None,:],q_a[:-1,:]) ,axis=0))
        q_a_py = 0.5*(concatenate((q_a[:,1:], q_a[:,-2][:, None]),axis=1) + q_a)
        q_a_my = 0.5*(q_a + concatenate((q_a[:,1][:, None],q_a[:,:-1]) ,axis=1))
    if user_action is not None:
        user_action(u_1, x, xv, y, yv, t, 0)

    for n in It[0:-1]:
        if version == 'scalar':
            u = advance_scalar(u, u_1, u_2, V, f, q, b, x, y, t, Cx2, Cy2, A, dt, n)
        else:
            f_a=f(xv, yv, t[n])
            u = advance_vector(u, u_1, u_2, V_a, f_a, q_a_px, q_a_mx, q_a_py, q_a_my,
                               b, x, y, t, Cx2, Cy2, A, dt, n)
        u_2, u_1, u = u_1, u, u_2

        if user_action is not None:
            user_action(u_1, x, xv, y, yv, t, n)

    t1=time.clock()
    run_time=t1-t0
    print 'For version %s the run time was =%g' % (version,run_time)

def advance_scalar(u, u_1, u_2, V, f, q, b, x, y, t, Cx2, Cy2, A, dt, n):
    Ix = range(0, u.shape[0]); Iy = range(0, u.shape[1])
    if n==0:
        #Adjusting help variables for first step (n=0)
        D1 = 0; D2 = 2*dt; B = 0.5
    else:
        #Standard values for help variables
        D1 = 1; D2 = 0;    B = A
    for i in Ix:
        for j in Iy:
            #used to handle boundary conditions
            im1 = i-1; ip1 = i+1; jm1 = j-1; jp1 = j+1
            if i==Ix[0]:
                im1 = ip1
            if i==Ix[-1]:
                ip1 = im1
            if j==Iy[0]:
                jm1 = jp1
            if j==Iy[-1]:
                jp1 = jm1
            #incrementation scheme
            u_xx = Cx2*(0.5*(q(x[ip1],y[j])+q(x[i],y[j]))*(u_1[ip1,j]-u_1[i,j]) \
                    - 0.5*(q(x[i],y[j])+q(x[im1],y[j]))*(u_1[i,j]-u_1[im1,j]))
            u_yy = Cy2*(0.5*(q(x[i],y[jp1])+q(x[i],y[j]))*(u_1[i,jp1]-u_1[i,j]) \
                    - 0.5*(q(x[i],y[j])+q(x[i],y[jm1]))*(u_1[i,j]-u_1[i,jm1]))

            u[i,j] = B*(2*u_1[i,j] - (1-b*dt/2)*(D1*u_2[i,j]-D2*V(x[i],y[j])) \
                        + u_xx + u_yy + f(x[i],y[j],t[n])*dt**2)
    return u

def advance_vector(u, u_1, u_2, V_a, f_a, q_a_px, q_a_mx, q_a_py, q_a_my, b, x,
                   y, t, Cx2, Cy2, A, dt, n):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if n==0:
        #Adjusting first step (n=0)
        D1=0;  D2=2*dt;   B=0.5
    else:
        #Standard values for help variables
        D1=1;  D2=0;      B=A
    #incrementation scheme
    u_1_px = concatenate((u_1[1:,:], u_1[-2,:][None,:]),axis=0)
    u_1_mx = concatenate((u_1[1,:][None,:],u_1[:-1,:]),axis=0)
    u_1_py = concatenate((u_1[:,1:], u_1[:,-2][:, None]),axis=1)
    u_1_my = concatenate((u_1[:,1][:, None],u_1[:,:-1]),axis=1)

    u_xx = Cx2*(q_a_px*(u_1_px-u_1) - q_a_mx*(u_1-u_1_mx))
    u_yy = Cy2*(q_a_py*(u_1_py-u_1) - q_a_my*(u_1-u_1_my))

    u=B*(2*u_1 - (1-b*dt/2)*(D1*u_2-D2*V_a) + u_xx + u_yy + f_a*dt**2)
    return u

import nose.tools as nt

def test_constant_solution():
    u_exact = lambda x,y,t : zeros((size(x),size(y))) + 2
    I = lambda x,y: zeros((size(x),size(y))) + 2
    V = lambda x,y: zeros((size(x),size(y)))
    q = lambda x,y: 3 + x + y
    f = lambda x,y,t: zeros((size(x),size(y)))

    b  = 2
    Lx = 4
    Ly = 4
    Nx = 4
    Ny = 4
    dt = 1
    T  = 4

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

    solver(I, V, f, q, b, Lx, Ly, Nx, Ny, dt, T, user_action=assert_no_error,
           version='scalar')

    solver(I,V,f,q,b,Lx,Ly,Nx,Ny,dt,T,user_action=assert_no_error,
           version='vector')
```

Running nosetests on `wave_project.py` in IPython gives the output below. Note that the vectorized version of the solver in this case was more than ten times faster than the scalar version.

```
In [1]: !nosetests -s wave_project.py
For version scalar the run time was =0.016335
For version vector the run time was =0.001311
.
----------------------------------------------------------------------
Ran 1 test in 0.021s

OK
```

2 Verification

2.1 Constant solution

For a constant solution, $u(x, y, t) = c$, both derivatives and differences of u are zero, that is

$$u_z = \frac{\partial u(x, y, t)}{\partial z} = 0, \quad \text{and} \quad [D_z u]^n_{i,j} = 0 \quad ,$$

at any point, and for all $z \in \{x, y, t\}$. Hence, $u(x, y, t) = c$ satisfies both the differential equation

$$u_{tt} + bu_t = (q(x, y)u_x)_x + (q(x, y)u_y)_y + f(x, y, t) \quad ,$$

and the corresponding discrete equation

$$[D_t D_t u + bD_t u = D_x(\bar{q}^x D_x u) + D_y(\bar{q}^y D_y u) + f]^n_{i,j} \quad ,$$

provided $f(x, y, t) = 0$. Furthermore, the boundary conditions are of course satisfied, and the initial conditions are satisfied by setting

$$I(x, y) = u(x, y, 0) = c, \quad \text{and} \quad V(x, y) = u_x(x, y, 0) = 0 \quad .$$

There are no requirements on the parameters b and q for this solution to be valid, so for the implementation we just choose some arbitrary values. A nose test checking that the program reproduces this exact solution for each time step was implemented into `wave_project.py` and given in the previous section.

Now lets invent five bugs and see what happens when we run nosetests on the code.

Bug 1: Indenting the code for the incrementation step in the `advance_scalar`-function so that the incrementation is done inside the `elif`-loop entered when $n \neq 0$, rather than after the tests to check the current step. This means that for the first step, `advance_scalar` returns u without changing it. Since u was initially filled with zeros, while we have chosen $u(x, y, t) = c = 2$, the constant test catches this bug. Note however that the bug would have passed undetected for $c = 0$. IPython session:

Bug 2: Forgetting to switch indices for when $i == Ix[0]$ in the `advance_scalar`-function. This does not give an error, because calling $x[-1]$ returns the last element of x, and it does not matter where in the computational domain we evaluate q and u, since any difference between two values of the constant u gives a factor 0, causing the term in question to vanish. Forgetting to switch indices for $i == Ix[-1]$ will however produce an "index out of bounds" error.

Bug 3: Forgetting to convert b to `float`, causing division with other integers to be interpreted as integer division. This was discovered, because with $b = 2$ this causes (for example) the over all factor $B = 1/2$ of the first time step to be calculated as 0, and hence u becomes the zero matrix. However, for $b = 1$, this bug passes undetected.

Bug 4: Wrong sign for the f-term in the incrementation step. This bug is not discovered, since we for the constant solution must have $f(x, y, t) = 0$

Bug 5: Forgetting the factor 2 in front of the `u_1[i,j]` term in the formula for u (incrementation step). Since this term is non-zero, a wrong factor here changes the value of u (to 1) in the first step, and the test fails.

2.2 Exact 1D plug-wave solution in 2D

We assume no damping, $b = 0$, constant wave velocity $q = const$, and no source term $f(x, y, t) = 0$. Choosing the initial waveform to be a plug wave which is constant in one direction, we satisfy the boundary conditions, and effectively reduce the two dimensional problem to a one dimensional problem with known solution. Below is a tentative implementation of a test to check that the 2D solution behaves as the corresponding 1D solution. This implementation plots and saves the surfaces for each timestep, but could benefit from also making aquantitative comparison.

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
import glob, os
def test_1D_plug_wave():
        plt.ion()
    def plot_u_vector(u, x, xv, y, yv, t, n):
        fig  = plt.figure()
        ax = fig.add_subplot(1,1,1,projection='3d')
        p = ax.plot_surface(xv,yv,u, rstride=1,cstride=1,cmap=plt.cm.coolwarm)
        plt.title('u(x,y)')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.savefig('frame_vector_%04d.png' % n)
        plt.close("all")

    def plot_u_scalar(u, x, xv, y, yv, t, n):
        fig  = plt.figure()
        ax = fig.add_subplot(1,1,1,projection='3d')
        x,y=meshgrid(x,y)
        p = ax.plot_surface(xv,yv,u, rstride=1,cstride=1,cmap=plt.cm.coolwarm)
        plt.title('u(x,y)')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.savefig('frame_scalar_%04d.png' % n)
        plt.close("all")


    #for 'vector' version
    def Iv_x(x,y):
        Iv_x=zeros((size(x),size(y)))
        Iv_x[6:-6,:]+=1.0
        return Iv_x
    def Iv_y(x,y):
        Iv_y=zeros((size(x),size(y)))
        Iv_y[:,7:-7]+=1.0
        return Iv_y
    #for 'scalar' version
    def Is_x(x,y):
        return 1 if 5<x<8 else 0
    def Is_y(x,y):
        return 1 if 6<y<9 else 0
    V = lambda x,y: zeros((size(x), size(y)))
    q = lambda x,y: zeros((size(x), size(y)))+1.0
    f = lambda x,y,t: zeros((size(x), size(y)))

    #q*dt/dx=1 og q*dt/dx=1
    b  = 0.0
    Lx = 13.0
    Ly = 15.0
    Nx = 13.0
    Ny = 15.0
    dt = 1.0
    T  = 12.0

    plt.close("all")
    for filename in glob.glob('frame_*.png'):
            os.remove(filename)

#   solver(Iv_x,V,f,q,b,Lx,Ly,Nx,Ny,dt,T,user_action=plot_u_vector,
#           version='vector')

    solver(Iv_y,V,f,q,b,Lx,Ly,Nx,Ny,dt,T,user_action=plot_u_vector,
            version='vector')

#   solver(Is_x,V,f,q,b,Lx,Ly,Nx,Ny,dt,T,user_action=plot_u_scalar,
#           version='scalar')

    solver(Is_y,V,f,q,b,Lx,Ly,Nx,Ny,dt,T,user_action=plot_u_scalar,
            version='scalar')
```