INF 5620
Mandatory Assignment 2

Andreas Slyngstad

28. september 2015

Introduction and numerical implementation

In this exercise we were tasked to take a closer look on a wave equation with variable wave velocity on the form.

$$u_{tt} = (qu_x)_x + f(x,t)$$

Where $f(x,t)$ is the added source term, which is used to verify the implementation. For this exercise we were to compare different discretizations of Neumann conditions, for the manifactured solution

$$u(x,t) = cos(\frac{\pi x}{L})cos(\omega t)$$

To discretize the PDE probelm we have to take a closer look at the wave velocity. Introducing ϕ such that $\phi = q(x)\frac{\partial u}{\partial x}$, and using centered differentiating on this parameter we end up with

$$\Big[\frac{\partial \phi}{\partial x}\Big]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x}$$

Where **n** represents position in time, while **i** is the position in space. Using the relation for ϕ we get

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

Finally rewriting the discretized PDE with respect to u_i^{n+1} and using arithmetic mean for the q values, the numerical scheme for calculating the inner points yields

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = \frac{1}{\Delta x^2}\Big(q_{i+\frac{1}{2}}(u_{i+1} - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n) + f_i^n$$
$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \Big(\frac{\Delta t}{\Delta x}\Big)^2\Big(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\Big) + f_i^n$$

Now considering the first timestep for $n = 0$, we face a problem calculating the u_i^{-1} term due to the fact that this point lies outside the meshgrid. This can be solved by the centered discretized initial velocity condition

$$\frac{\partial u}{\partial t}(x,0) = V \qquad \frac{u_i^1 - u_i^{-1}}{2\Delta t} = V \qquad u_i^{-1} = u_i^1 - 2\Delta t V$$
$$u_i^{n+1} = \Delta t V + u_i^n + \frac{1}{2}\Big(\frac{\Delta t}{\Delta x}\Big)^2\Big(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\Big) + \frac{1}{2}f_i^n$$

For all timesteps, we observe we run upon points outside of meshpoints in space when $i = 0, L$. Here we use the Neumann condition to help us out, but not only for u but also q such that

$$\frac{\partial u}{\partial x}(x,t)\Big|_{i=0,L}^n = 0 \qquad \frac{\partial q}{\partial x}(x)\Big|_{i=0,L} = 0$$

In other words $q_{i+1} = q_{i-1}$. Using this result, the boundary calculations can be rewritten such that for $i = L$, we end up with

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + \Big(\frac{\Delta t}{\Delta x}\Big)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n$$
$$u_i^{n+1} = \Delta t V + u_i^n + \frac{1}{2}\Big(\frac{\Delta t}{\Delta x}\Big)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \frac{1}{2}\Delta t^2 f_i^n$$

Numerical results will be visually presented in gif files provided in my repository.

Exercise a

In this test problem we implement q as $q = 1 + (x - \frac{L}{2})^4$. I use sympy to calculate f in my python method fvalues provided in the code. I get the following convergence rate

```
Using V = 0, L = 2, dx = 0.0600, dt = 0.0300
Dividing dt by 2, 4 times
Convergence rate 0.25847 for dt = 0.03000
Convergence rate 0.26126 for dt = 0.01500
Convergence rate 0.34039 for dt = 0.00750
Convergence rate 0.42694 for dt = 0.00375
```

We observe that the solution looks more or less good. Looking at the convergence rate it seems that it's closing in towards 0.5, which is really low. So it seems even though the scheme solution looks good, the covergencerate is slow. It should be aroud 2.

Exercise b

In this test problem we implement q as $q(x) = cos(\frac{\pi x}{L})$. Again I use sympy to calculate f in my python method fvalues provided in the code. Now the convergence rate yields

```
Using V = 0, L = 2, dx = 0.0600, dt = 0.0300
Dividing dt by 2, 4 times
Convergence rate 0.51569 for dt = 0.03000
Convergence rate 0.50705 for dt = 0.01500
Convergence rate 0.50332 for dt = 0.00750
Convergence rate 0.50161 for dt = 0.00375
```

Here the convergence rate is better, stabilizing around 0.5 which is higher than in exercise a. But again, it should be around 2 so there might me some error in my scheme.

Exercise c

Here we use an approximation for the end points $i = 0, L$ in the following way

$$u_i - u_{i-1} = 0 \;\; i = Nx \qquad u_{i+1} - u_i = 0 \;\; i = 0$$

In other words we say that the end points is the same as the neighbor value within the mesh. Printing the convergence rate

```
Using V = 0, L = 2, dx = 0.0600, dt = 0.0300
Dividing dt by 2, 4 times
Convergence rate 0.46944 for dt = 0.03000
Convergence rate 0.48476 for dt = 0.01500
Convergence rate 0.49993 for dt = 0.00750
Convergence rate 0.49996 for dt = 0.00375
```

Even though we observe that this estimate gives a poor solution over time in the animation, we get convergence rate around 0.5.

Exercise d

As far as I can see, we get the same scheme as in b, only with a factor of $\frac{1}{2}$.

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sym
import os, glob

#os.remove('fig_*.png')
#convert -delay 10 -loop 0 frame_*.png animation.gif
def fvalues(task):
        x, t, L, w= sym.symbols('x, t, L, w')  # global symbols
        if task == "a":
                q = 1 + (x-L/2)**4
        else:
                q = 1 + sym.cos(sym.pi*x/L)

        u = sym.cos(x*sym.pi/L)*sym.cos(w*t)
        u_x = sym.diff(u,x)
        q_x = sym.diff(q,x)
        f = sym.diff(u,t,t) -q_x*u_x - q*sym.diff(u,x,x)
        sym.simplify(f)
        sym.simplify(q)
        f = sym.lambdify((x,t,L,w), f)
        q = sym.lambdify((x,L), q)
        return f, q


def fv(x,t,L,w):
        return (-16*L**2*w**2*np.cos(np.pi*x/L) - 8*np.pi*L*(L - 2*x)**3*np.sin(n
        np.pi**2.0*((L - 2*x)**4 + 16)*np.cos(np.pi*x/L))*np.cos(t*w)/(16*L**2)

def qval(x, L):
        return 1.0 + (x-L/2.0)**4

def exact(x, t, L, w):
        return np.cos(np.pi*x/L)*np.cos(w*t)

def solver(V, w, dx, L, dt, T, task):
        Nt = int(round(T/dt))
        t = np.linspace(0, T, Nt+1)
        Nx = int(round(L/dx))
        x = np.linspace(0, L, Nx+1)

        c = dt/dx
        u = np.zeros(Nx+1)
        u_1 = np.zeros(Nx+1)#last
        u1  = np.zeros(Nx+1)#next


        fval, q = fvalues(task) #Test of sympy implementation, remove if needed
        fval = np.vectorize(fval)
        q = np.vectorize(q); q = q(x, L)

        #Initial Condition
        u_1[:] = exact(x, 0, L, w)
        #First step
        u[1:-1] = dt*V + u_1[1:-1]+c**2*0.25*\
        ((q[1:-1]+q[2:])*(u_1[2:]-u_1[1:-1])-(q[1:-1]+q[:-2])*(u_1[1:-1]-u_1[:-2]
        0.5*dt**2*fval(x[1:-1],0,L,w)

        if task == "c":
                u[0] = u[1]
                u[-1] = u[-2]
        if task == "d":
                u[0] = dt*V + u_1[0]+c**2*0.25*(-(q[1]+q[0])*(u_1[1]-u_1[0])) + 0
                u[-1] = dt*V + u_1[-1]+c**2*0.25*(-(q[-1]+q[-2])*(u_1[-1]-u_1[-2]
        else: #Check brok
                u[0] = dt*V + u_1[0]+c**2 *0.5*(q[0]+q[1])*(u_1[1] -u_1[0]) + 0.5
                u[-1] = dt*V + u_1[-1]+c**2 *0.5*(q[-1]+q[-2])*(u_1[-2] -u_1[-1])

        error = u
        value = np.max(abs(u-exact(x, t[1], L, w)))
        time = dt


        for i in range(1,Nt+1):
                u1[1:-1] = -u_1[1:-1] + 2*u[1:-1]+c**2*0.5*\
                ((q[1:-1]+q[2:])*(u[2:]-u[1:-1])-(q[1:-1]+q[:-2])*(u[1:-1]-u[:-2]
                dt**2*fval(x[1:-1],t[i],L,w)

                if task == "a" or task == "b": #Can be changed to else
                        u1[0] = -u_1[0]+2.0*u[0]+ c**2 *(q[0]+q[1])*(u[1]-u[0]) +
                        u1[-1] = -u_1[-1]+ 2.0*u[-1]+c**2 *(q[-1]+q[-2])*(u[-2]-u

                if task == "c":
                        u1[0] = u1[1]
                        u1[-1] = u1[-2]
                        #u1[0] = -u_1[1] + 2*u[1]+c**2*0.5* ((q[1]+q[2])*(u[2]-u[
                        #                dt**2*fval(x[1],t[1],L,w)
                        #u1[-1] = -u_1[-2] + 2*u[-2]+c**2*0.5* ((q[-2]+q[-1])*(u[
                        #                dt**2*fval(x[-2],t[-2],L,w)

                if task == "d":
                        u1[0] = -u_1[0] +2*u[0]+c**2*0.5*(-(q[1]+q[0])*(u[1]-u[0]
                        u1[-1] = -u_1[-1] +2*u[-1]+c**2*0.5*((q[-1]+q[-2])*(u[-1]

                u_1[:] = u[:]
                u[:] = u1[:]


                if value < np.max(abs(u1[:]-exact(x, (i+1)*dt, L, w)[:])):
                        value = np.max(abs(u1[:]-exact(x, (i+1)*dt, L, w)[:]))
                        error[:] = u1[:]
                        time = (i+1)*dt

                plt.figure()
                plt.plot(x, exact(x,t[i],L,w),'g')
                plt.plot(x, u1)
                plt.axis([-0.1,2.1,-1.2,1.2])
                plt.xlabel('x')
                plt.ylabel('Values of u')
                plt.title("Exercise %s, numerical solution against analytical \n"
                 "Time = %d, dt = %.2f, L = %d, dx = %.2f" % (task, T,dt,L,dx))
                plt.savefig('frame_%04d.png' % i)
                return error, time, x

        def convergence_rates(m, w, dx, L, dt, T, V, solver_function, task):

            dt_values = []
            E_values = []
            for i in range(m):
                #u, t = solver_function(V,w,dx,L,dt,T)
                u, t, x = solver(V,w,dx,L,dt,T, task)
                u_e = exact(x, t, L, w)
                E = np.sqrt(dt*np.sum((u_e-u)**2))
                dt_values.append(dt)
                E_values.append(E)
                dt = dt/2

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

        def main():
                import sys
                V = 0.0
                L = 2; T = 6
                dx = 0.09; dt = 0.04
                if dt/dx > 0.5:
                        print "dt/dx must be lower than 0.5"
                        sys.exit(1)
                w = 1
                m = 5
                task = "d"
                #solver(V, w, dx, L, dt, T, task)
                r, Dt = convergence_rates(m, w, dx, L, dt, T, V, solver, task)
                print "Using V = %d, L = %d, dx = %.4f, dt = %.4f" % (V, L, dx, dt)
                print "Dividing dt by 2, %d times" % (m-1)
                for i in range(m-1):
                        print "Convergence rate %.5f for dt = %.5f" % (r[i], Dt[i])


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