Wave Project
-INF5620-

Håkon Vikør Treider
h.v.treider@fys.uio.no

Finite difference simulation of 2D waves

October 19, 2015

Introduction

In this project we are going to use finite difference method to solve the the two-dimensional, standard, linear wave equation with damping

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial^2 u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t), \quad (1)$$

..and standard Neumann boundary conditions

$$\frac{\partial u}{\partial n} = 0, \quad (2)$$

in the rectangular spatial domain $\Omega = [0, L_x]\times[0, L_y]$. Our initial conditions are as follows

$$u(x,y,0) = I(x,y) \quad (3)$$

$$u_t(x,y,0) = V(x,y) \quad (4)$$

To verify our implementation, we construct three test cases - a constant solution, a plug wave and a standing wave. Finally we investigate the physical problem of a wave that enters a medium with different wave velocity.

1 Numerical scheme and discretization

We find our discretization of (1), by using:

$$\frac{\partial^2 u}{\partial t^2} \approx [D_t D_t u]^n_{i,j} = \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2} \quad (5)$$

$$\frac{\partial^2 u}{\partial x^2} \approx [D_x D_x u]^n_{i,j} = \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} \quad (6)$$

$$\frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) \approx [D_x q D_x u]^n_{i,j} \quad (7)$$

$$= \frac{1}{\Delta x^2}(q_{i+1/2,j}(u^n_{i+1,j} - u^n_{i,j}) - q_{i-1/2,j}(u^n_{i,j} + u^n_{i-1,j})) \quad (8)$$

$$\frac{\partial^2 u}{\partial y^2} \approx [D_y D_y u]^n_{i,j} = \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2} \quad (9)$$

$$\frac{\partial u}{\partial t} \approx [D_t u]^n_{i,j} = \frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2\Delta t} \quad (10)$$

(1) can now be written as:

$$[D_t D_t u]^n_{i,j} + b[D_t u]^n_{i,j} = [D_x q D_x u]^n_{i,j} + [D_y q D_y u]^n_{i,j} + f^n_{i,j} \quad (11)$$

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

We solve this equation for $u_{i,j}^{n+1}$:

$$u_{i,j}^{n+1}(\frac{1}{\Delta t^2} + b\frac{1}{2\Delta t}) = \frac{2u_{i,j}^{n} - u_{i,j}^{n-1}}{\Delta t^2} + b\frac{u_{i,j}^{n-1}}{2\Delta t} + \frac{1}{\Delta x^2}u_{xx} + \frac{1}{\Delta y^2}u_{yy} + f_{i,j}^{n}$$

$$u_{i,j}^{n+1} = \frac{2u_{i,j}^{n} + (b\frac{\Delta t}{2} - 1)u_{i,j}^{n-1}}{1 + b\frac{\Delta t}{2}} + \frac{1}{1 + b\frac{\Delta t}{2}}(C_x^2 u_{xx} + C_y^2 u_{yy}) + \frac{\Delta t^2}{1 + b\frac{\Delta t}{2}}f_{i,j}^{n}, \tag{12}$$

where we use:

$$u_{xx} = \frac{1}{2}(q_{i,j} + q_{i+1,j})(u_{i+1,j}^{n} - u_{i,j}^{n}) - \frac{1}{2}(q_{i,j} + q_{i-1,j})(u_{i,j}^{n} - u_{i-1,j}^{n}) \tag{13}$$

$$u_{yy} = \frac{1}{2}(q_{i,j} + q_{i,j+1})(u_{i,j+1}^{n} - u_{i,j}^{n}) - \frac{1}{2}(q_{i,j} + q_{i,j-1})(u_{i,j}^{n} - u_{i,j-1}^{n}) \tag{14}$$

$$C_x = \frac{\Delta t^2}{\Delta x^2}, \; C_y = \frac{\Delta t^2}{\Delta y^2} \tag{15}$$

The boundary points are found using the fact that at the border, we have additional equations coming from the Neumann condition. We start by setting $x = 0, Nx$ and then $y = 0, Ny$ while running through the opposite index. The extra equations arise from taking the central difference at the boundary. We get:

$$\frac{\partial u}{\partial x} = 0 \;\; \rightarrow \;\; u_{i+1,j}^{n} = u_{i-1,j}^{n} \tag{16}$$

$$\frac{\partial u}{\partial y} = 0 \;\; \rightarrow \;\; u_{i,j+1}^{n} = u_{i,j-1}^{n} \tag{17}$$

The same also applies to $q_{i,j}$ at all four boundaries. At the four corner points, we can use both these (*actually all four when $q(x, y)$ is included*) equations to find the solution here.

To find the scheme for the first time step we use the initial condition (4):

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} = V(x, y) \tag{18}$$

Solving for $u_{i,j}^{n-1}$ and inserting in the general scheme above, we obtain our implemented first step.

Verification of algorithm

Constant solution

A first test, is a constant solution $u(x, y, t) = C$. We can then fit the parameters f, b, q, I, and V such that the solution fulfills the given PDE:

$$f = 0, \ b = 0.5, \ q = 1, \ I = 3, \ V = 0 \tag{19}$$

We then show that the constant solution is also a solution of the discrete equation (11), by inserting the solution into (12). With simple algebra, (the derivatives of a constant "tend to vanish"...) we see that $u = C$ is an acceptable solution.

In the code, I have implemented this as a test `constant_solution(Nx, Ny, version)`, found in the file `tests.py`. The version argument simply decides whether to use scalar or vectorized implementation. It is called by `test_constant()` that checks different (non-)square meshes. When run with a very small tolerance for error, the numerical solution is for all purposes equal to the analytical, C. This is also what we find.

To further investigate what kind of errors we would spot using a constant solution, we test different potential errors. When we add an extra factor to either f or V, we get an increasing solution (i.e. not constant) - while for I, the solution is still constant, but not equal to c, - off by the same factor we multiplied with. A change of the parameter q or b has no impact, so an important point is that this test cannot say anything about the correctness of the calculation of these in the code.

Exact 1D plug-wave solution in 2D

Simulating a plug wave where $I(x) = 0$ if $x - L_x/2.0 > 0.1$ and else 1. The initial wave splits into two waves moving exactly one unit cell per time step, in opposite directions (see `gifs/plug_wave.gif`) for the wave visualized). The implemented test, `test_plug()` can be run directly (or via nosetest) and it checks that the wave ends up in its initial configuration after one period. This is also the case, as the gif made from the program can visually confirm.

Standing, undamped waves

Using an exact analytical solution we may compute the error of our scheme, which should approach zero as $\Delta t, \Delta x, \Delta y \to 0$. However, what is most interesting is the rate at which these schemes tend to the exact solution. We call this the convergence rate r, and for this specific scheme we may expect a rate of 2 (can be found with a mathematical inspection of our finite approximations)

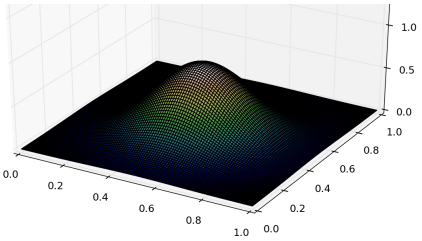
With no damping term b and a constant wave velocity, c,

$$u_{exact}(x, y, t) = A\cos(k_x x)\cos(k_y y)\cos(\omega t), \ \ k_x = \frac{\pi m_x}{L_x}, \ k_y = \frac{\pi m_y}{L_y}$$

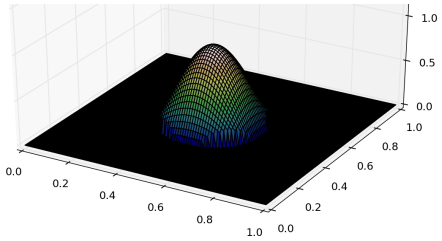
is an acceptable solution and actually an exact solution of the discrete equations. Here A is the amplitude of the wave and $\omega = \sqrt{c^2(k_x^2 + k_y^2)}$.

N(i)	N(i+1)	dt(i)	dt(i+1)	r(i)
5	10	1.4E-1	7.1E-2	2.46
10	20	7.1E-2	3.6E-2	2.37
20	40	3.6E-2	1.8E-2	1.96
40	80	1.8E-2	8.8E-3	2.09
80	160	8.8E-3	4.4E-3	1.99
160	320	4.4E-3	2.2E-3	2.02

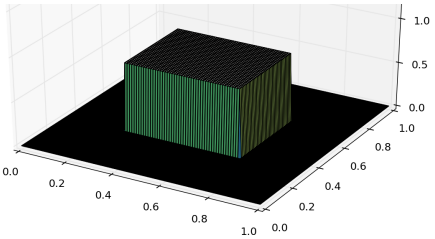
Table 1: Convergence rate for standing, undamped waves



(a) Subsea hill: Gaussian shape



(b) Subsea hill: "Cosine hat" shape



(c) Subsea hill: Box shape

Figure 1: Geometry of subsea floor

We define the error, $e^n_{i,j} = u(x_i, y_j, t_n) - u^n_{i,j}$ and $E = \text{norm}(e^n_{i,j}) = ||e^n_{i,j}||$ and $r = \frac{\log(E_{i+1}/E_i)}{\log(\Delta t_{i+1}/\Delta t_i)}$. A visualization of the standing wave, for arbitrary values for m_x and m_y can be seen in `gifs/stnd_undmp_cos.gif`. From the output of the program this is what we find for the convergence rate: (see table 1)

Investigating a physical problem

It is finally time to test our 2D finite difference solver on a physical problem. The case we are looking into is the propagation of a tsunami over a subsea hill with different shape / geometry. Since we already have tested the vectorized code to give equally good results as the scalar version, we use it for the additional speed it gives (approximately 100 times faster). If we run

```
>>> python main.py
```

we get a friendly user interface where we can choose between the three different hill shapes: Gaussian, "cosine hat" and a simple box. These can be seen in figure 1.

After a little while, we get a gif-file that can be viewed/investigated (the individual frames can also be used of course). What we see is the wave slowing

down over the subsea hill, and this means the amplitude of the wave has to go up. This is also what I expected from real world experience. When a wave is heading towards land and the water gets more and more shallow, it slows down and starts to rise. This effect makes the wave hit the entire shoreline "almost" at the same time, even when coming from an angle.

Since I am using a Δt in all of my calculations that are numerically stable, there arent any noise in either of the simulations. However, when using the "Box hill", the solution looks unnatural/unphysical at some points due to the harsh edges.

Conclusion

With all the tests implemented running correctly, and the convergence rates also agreeing - it makes it reasonable to conclude that my wave solver is working properly, for both the scalar and vectorized versions. The animations generated by the program also seems very plausible, even when using a rough / non-smooth ocean floor (i.e. wave velocity q(x,t))