INF5620
Wave Project

Krister Stræte Karlsen

October 12, 2015

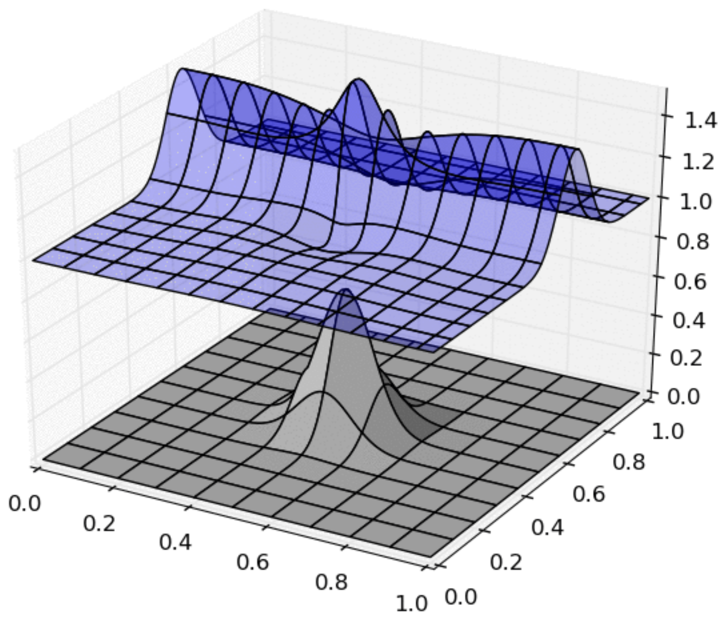


Figure 1: Simulation of a wave over a steep Gaussian Hill.

1 Introduction

In this project we will build a solver for a 2D wave equation with variable wave velocity and a linear damping term. Furthermore study if the numerical scheme is able to reproduce the known exact solution.

2 The mathematical problem

The boundary value problem to be studied in the project is a linear 2D wave equation

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t), \tag{2.1}$$

over the rectangular spatial domain $\Omega = [0, L_x] \times [0, L_y]$ with Neumann boundary conditions

$$\frac{\partial u}{\partial n} = 0, \tag{2.2}$$

and initial conditions

$$u(x,y,0) = I(x,y), \quad \frac{\partial}{\partial t}u(x,y,0) = V(x,y). \tag{2.3}$$

2.1 Discretization

To solve this equation numerically we must limit our domain to a finite number of discrete points. The discrete domain is built by sampling $x, y \in \Omega$ using N_x and N_y number of equally distanced points in both directions($\Delta x, \Delta y$: *constant*). That means, $x = x_1, x_2, .., x_{Nx}$ and $y = y_1, y_2, .., y_{Ny}$. The discrete time points will be defined in a similar way, $t = t_0, t_1, .., t_{Nt}$, also equally spaced(Δt : *constant*).

Discretizing the equation as follows, using Hans Petter Langtangen's operator notation:

$$[D_t D_t u + bD_{2t}u = D_x q D_x u + D_y q D_y u + f]^n_{i,j} \tag{2.4}$$

And the boundary/initial conditions:

$$\left.\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}\right|_{i=0,N_x} = 0 \quad and \quad \left.\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}\right|_{j=0,N_y} = 0 \tag{2.5}$$

$$u^0_{i,j} = I_{i,j} \quad and \quad \frac{u^1_{i,j} - u^1_{i,j}}{2\Delta t} = V_{i,j} \tag{2.6}$$

3 Implementation

3.1 A numerical scheme

Now we need to derive the scheme needed for solving the equation. Since we have two spatial dimensions we need special formulas for all the four boundaries and the four corners. We also need special formulas for the first step in time in addition to the general formula for inner spatial points mesh points. In total this adds up to 18 different formulas. Having derived all of them I realized that going with ghost points would be much easier to implement without mistakes.

Some comments about the implementation of ghost points can be found under the section with the obvious choice of name: *Ghost points.*

Using ghost points we don't run into problems at the boundary so we just need to derive two formulas for the inner spatial mesh points instead of a total of 18 for the other method. Solving (2.4) for $u^1_{i,j}$ and $u^{n+1}_{i,j}$ we obtain the following formulas:

$$k = \frac{b\Delta t}{2}, \quad E_x = \frac{\Delta t^2}{\Delta x^2}, \quad E_y = \frac{\Delta t^2}{\Delta y^2}$$

The scheme for all inner points (time and space) then becomes:

$$\begin{aligned} u^{n+1}_{i,j} =& \frac{E_x}{1+k}\left[\frac{1}{2}(q_i + q_{i+1})(u_{i+1} - u_i) - \frac{1}{2}(q_{i-1} + q_i)(u_i - u_{i-1})\right]^n_j \\ &+ \frac{E_y}{1+k}\left[\frac{1}{2}(q_j + q_{j+1})(u_{j+1} - u_j) - \frac{1}{2}(q_{j-1} + q_j)(u_j - u_{j-1})\right]^n_i \\ &+ \frac{2}{1+k}u^n_{i,j} - \frac{1-k}{1+k}u^{n-1}_{i,j} + \frac{\Delta t^2}{1+k}f^n_{i,j} \end{aligned} \tag{3.1}$$

Here we have used *arithmetic mean* approximations of q between to mesh points. In order to compute the solution at the first time-level we need the fictitious value $u^{-1}_{i,j}$. To avoid this we simply use the second initial condition(2.6) to obtain $u^{-1}_{i,j} = u^1_{i,j} - 2\Delta t V_{i,j}$. Now the formula for the first step is

$$
\begin{aligned}
u_{i,j}^{n+1} =& \frac{E_x}{2}\left[\frac{1}{2}(q_i+q_{i+1})(u_{i+1}-u_i)-\frac{1}{2}(q_{i-1}+q_i)(u_i-u_{i-1})\right]_j^0 \\
&+\frac{E_y}{2}\left[\frac{1}{2}(q_j+q_{j+1})(u_{j+1}-u_j)-\frac{1}{2}(q_{j-1}+q_j)(u_j-u_{j-1})\right]_i^0 \\
&+u_{i,j}^0+\Delta t V_{i,j}+\frac{\Delta t^2}{2}f_{i,j}^0 \qquad (3.2)
\end{aligned}
$$

3.1.1 Ghost points

To avoid running into problems at all boundaries we simply expand our computational domain with two more points in each direction, x and y. Now we have a physical domain of $N \times N$ points, and computational domain of $(N+2)\times(N+2)$ points. Lets say the physical domain with its indices is $\Omega_{(P)} = \{1, 2, 3, .., N\}$ and the ghost points are $\Omega_{(G)} = \{0, N+1\}$. We are now left with the question : *What values should these extra mesh points(ghost points) take?* To answer that question we must turn to Neumann, not the man, but the boundary condition carrying his name.

At all the boundaries we have from the discretized Neumann condition

$$u_{l+1}^n = u_{l-1}^n, \quad l = 1, N_x, N_y.$$

This means, for instance, that $u_{0,j} = u_{2,j}$ for all j in the physical domain. Similar relations for the other boundaries can easily be obtained from the Neumann condition above and an illustration of this an be found below.

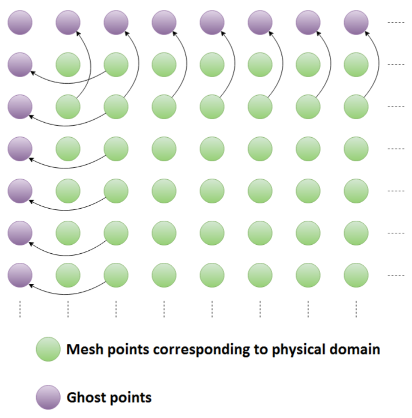


Figure 2: A corner in our computational domain illustrating which values the ghost points take.

The ghost points must of course be updated at each time-level.

3.2 The algorithm

```
                         Algorithm 1
Set:
    $$\left.\begin{aligned} I_{ij} &= I(x,y) \\ V_{ij} &= V(x,y) \\ q_{ij} &= q(x,y) \\ f_{ij}^0 &= f(x,y,0) \end{aligned}\right\} i,j \in \Omega_{(P)}$$
Set ghost points:
    $$\left.\begin{matrix} I_{0j} = I_{2j} \quad and \quad I_{(N_x+1)j} = I_{(N_x-1)j} \\ I_{i0} = I_{i2} \quad and \quad I_{i(N_y+1)} = I_{i(N_y-1)} \\ \vdots \end{matrix}\right\} i,j \in \Omega_{(P)}$$
Equally for $V_{ij}, f_{ij}$ and $q_{ij}$.
Compute first step:
    $u_{ij}^1 =< formula(3.2) >, \quad i,j \in \Omega_{(P)}$
Update ghost points:
    $u_{0j}^1 = u_{2j}^1 \quad and \quad u_{(N_x+1)j}^1 = u_{(N_x-1)j}^1$
    $u_{i0}^1 = u_{i2}^1 \quad and \quad u_{i(N_y+1)}^1 = u_{i(N_y-1)}^1$
Equally for $f_{ij}^1$.
Compute next(general) step:

$for\ n = 1, 2, 3, ..., N_t - 1$
        $u_{ij}^{n+1} =< formula(3.1) >, \quad i,j \in \Omega_{(P)}$
    Update ghost points:
        $u_{0j}^{n+1} = u_{2j}^{n+1} \quad and \quad u_{(N_x+1)j}^{n+1} = u_{(N_x-1)j}^{n+1}$
        $u_{i0}^{n+1} = u_{i2}^{n+1} \quad and \quad u_{i(N_y+1)}^{n+1} = u_{i(N_y-1)}^{n+1}$
    Equally for $f_{ij}^{n+1}$.
```

4 Verification

4.1 Constant solution

To verify that the scheme we derived and implemented above produces correct results we are going to construct a constant solution $u(x, y, t) = k, k \in \mathbb{R}$ to use as a test case.

Inserting $u(x, y, t) = k$ into (2.1)-(2.3) we obtain the following:

$$f(x,y,t) = 0, \quad V(x,y) = 0, \quad I(x,y) = k$$
$$q(x,y), b : \quad \textit{no restrictions}$$

Next we are going to make sure $u(x, y, t) = k$ satisfies the discrete equation (2.4).

$$\frac{k - 2k - k}{\Delta t^2} + b\frac{k - k}{2\Delta t} = \left[D_x q \frac{k-k}{\Delta x}\right]^n_{i,j} + \left[D_y q \frac{k-k}{\Delta y}\right]^n_{i,j} + f^n_{i,j}$$

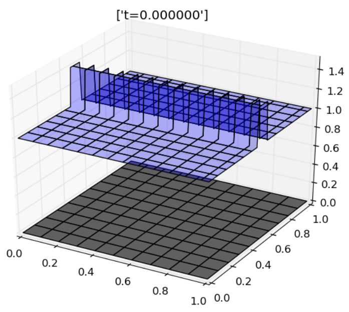
We see that the discrete equation is satisfied(zero on both sides) for the choices of $V(x, y), f(x, y, t)$ and $I(x, y)$ stated above.

To verify that program actually reproduces this result run: `constant_test.py`.

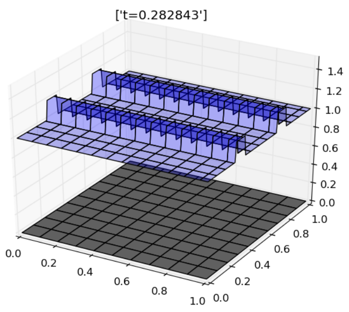
4.2 1D Plug wave

To see if the program could reproduce a one dimensional plug wave the initial condition was set to be $I = k$, for some constant k for $x \in (0.45, 55)$ and zero else.

For reproduction of the result run `plug_wave.py`.



(a) Initial condition.



(b) Later split into two plug waves.

Figure 3: Simulation of a one dimensional plug wave.

4.3 Standing, undamped waves

This section will investigate the convergence rate of the numerical solution as $\Delta x, \Delta y, \Delta t \rightarrow 0$. For this purpose we have chosen the known analytical solution of a undamped($b = 0$) standing wave:

$$u_e(x, y, t) = Acos(k_x x)cos(k_y y)cos(\omega t), \quad k_x = \frac{m_x \pi}{L_x}, \quad \frac{m_y \pi}{L_y}$$

In order for this solution to fulfill the undamped wave equation we must make a suitable choice of ω. Inserting u_e into the wave equation we obtain that the following relation must be satisfied: $\omega = \sqrt{q(k_x + k_y)}$.

To simulate this wave we need to adapt the conditions and parameters we use in the solver. These can be obtained from the exact solution above and the boundary value problem (2.1)-(2.3).

$$I(x, y) = u_e(x, y, 0) = Acos(k_x x)cos(k_y y), \quad V(x, y) = \frac{\partial}{\partial t}u(x, y, 0) = 0$$
$$q = c^2 : \quad \textit{no restrictions, but must remain constant}$$

4.3.1 Convergence rate

The expression used for the study of convergence rate is simply the maximum cell error over space and time for a simulation. This can be expressed as:

$$E = ||max_{i,j,t}(e^n_{i,j})|| \tag{4.1}$$

To study how this error behaves as $\Delta x, \Delta y, \Delta t \to 0$ we define $h = \Delta x \Delta y \Delta t$ and reduce it for each simulation. This gives us a sequence of errors, $E_1, E_2, ..., E_k$ and a corresponding sequence of h values, $h_1, h_2, .., h_k$.

The convergence rate can then be computed according to

$$r_i = \frac{log(E_{i-1}/E_i)}{log(h_{i-1}/h_i)}$$

Doing a set of simulations gave the following results.

Table 1: Data from numerical experiments. Computing maximum error according to (4.1) while reducing h_i.

h_i	E_i
0.1	0.0.36659
0.05	0.09938
0.025	0.02520

The corresponding convergence rates are: $r = 1.8830.., 1.9791...$

Even tough the error tends to zero and the convergence rate to 2, as we hoped and expected.

5 Investigation of a physical problem

In this section we will have a look at a model for so called *shallow water waves*. These waves travel with a velocity according to $c = \sqrt{gH}$, H being the depth of the water and ,g , gravity. It is when $\lambda >> H$ one uses this simple relationship between velocity and depth. We can easily include this effect in our program, since it can take variable wave velocities. This is done by introducing a model for the ocean floor elevation, $B(x, y)$, and an undisturbed water level, h_0. The water depth at a given point is then described by $H(x, y) = h_0 - B(x, y)$. We now have an expression for the wave velocity as a function of water depth: $c = \sqrt{q} = \sqrt{gH(x, y)}$ we can use in our program.

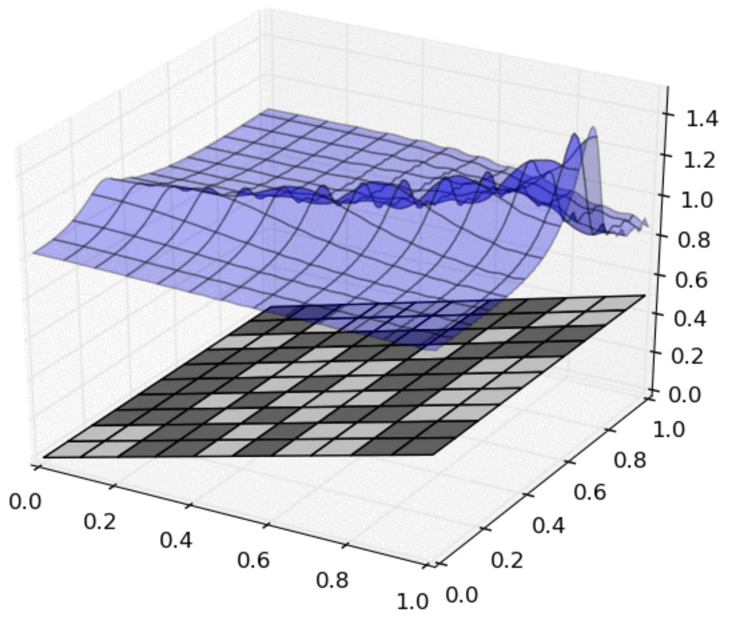


Figure 4: Wave propagation along a shore with rapidly increasing depth. This illustrates indeed the physical phenomenon $c = \sqrt{gH(x,y)}$, where we can see one part of the wave traveling faster (the wave is traveling to the left).

The effect discussed in the previous paragraph is well illustrated in figure 3, where we can observe a wave traveling left with one part slowing down due to shallower water.

One more phenomenon to be observed is the increasing wave amplitude. When the wave slows down, in order for the energy to be conserved the amplitude must increase, since the frequency is constant.