# Finite Difference Simulation of 2D Waves

*Project in IN5270, University of Oslo*

Eina Jørgensen
einaj@mail.uio.no

6th of October 2019

## I. The Mathematical Problem

The equation we are looking to solve in this situation is

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} &= \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right)\\ &+ \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t)\end{aligned} \tag{1}$$

$u(x,y,t)$ hight og the wave at a given point in time, $q(x,y)$ a variable wave velocity, $b$ a constant ehich is the dampenig factor and $f(x,y,t)$ the source term. Our initial conditions are

$$\begin{aligned}u(x,y,0) &= I(x,y)\\ \frac{\partial u(x,y,0)}{\partial t} &= V(x,y)\end{aligned}$$

With boundry condtions

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

On the space domain $\Omega = [0,L_x] \times [0,L_y]$

## II. Discretization of the Problem

### A. Discrete Equations

For discretization we use the following discretizations:

$$\begin{aligned}[D_tD_tu]^n_{i,j} &= \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2}\\ [D_2tu]^n_{i,j} &= \frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2\Delta t}\\ [D_xqD_xu]^n_{i,j} &= \frac{1}{\Delta x^2}[q_{i+1/2,j}(u^n_{i+1,j} - u^n_{i,j})\\ &\quad - q_{i-1/2,j}(u^n_{i,j} - u^n_{i-1,j})]\\ [D_yqD_yu]^n_{i,j} &= \frac{1}{\Delta y^2}[q_{i,j+1/2}(u^n_{i,j+1} - u^n_{i,j})\\ &\quad - q_{i,j-1/2}(u^n_{i,j} - u^n_{i,j-1})]\end{aligned} \tag{2}$$

Where $q_{i\pm 1/2,j}$ is defined by

$$q_{i\pm 1/2,j} = \frac{1}{2}(q_{i,j} + q_{i\pm 1,j})$$

An likewise for $q_{i,j\pm 1/2}$.

### B. The General Scheme

Inserting the discretizations into (1) we get

$$\begin{aligned}&[D_tD_tu]^n_{i,j} + b[D_2tu]^n_{i,j}\\ &= [D_xqD_xu]^n_{i,j} + [D_yqD_yu]^n_{i,j} + f^n_{i,j}\end{aligned}$$

Temporarily, we will refer to the right hand side as $R$. Inserting for the left hand side of the equation and keeping $R$ as above, we get:

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^{n} + u_{i,j}^{n-1}}{\Delta t^2} + b\frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} = R$$

Multiplying by $\Delta t^2$ and solving for $u_{i,j}^{n+1}$:

$$u_{i,j}^{n+1}\left(1 + \frac{b\Delta t}{2}\right) - 2u_{i,j}^{n} + u_{i,j}^{n-1}\left(1 - \frac{b\Delta t}{2}\right) = R\Delta t^2$$
$$\Downarrow$$
$$u_{i,j}^{n+1} = R\frac{\Delta t^2}{\left(1 + \frac{b\Delta t}{2}\right)} + \frac{2u_{i,j}^{n}}{\left(1 + \frac{b\Delta t}{2}\right)} - u_{i,j}^{n-1}\frac{\left(1 - \frac{b\Delta t}{2}\right)}{\left(1 + \frac{b\Delta t}{2}\right)}$$

and finally, inserting back for $R$, we get the general scheme:

$$\begin{aligned} &u_{i,j}^{n+1} = \\ &\left([D_xqD_xu]_{i,j}^{n} + [D_yqD_yu]_{i,j}^{n} + f_{i,j}^{n}\right)\frac{\Delta t^2}{\left(1 + \frac{b\Delta t}{2}\right)} \\ &+ \frac{2u_{i,j}^{n}}{\left(1 + \frac{b\Delta t}{2}\right)} - u_{i,j}^{n-1}\frac{\left(1 - \frac{b\Delta t}{2}\right)}{\left(1 + \frac{b\Delta t}{2}\right)} \end{aligned} \tag{3}$$

with $[D_xqD_xu]_{i,j}^{n}$ and $[D_yqD_yu]_{i,j}^{n}$ as defined in (2).

## C. The Initial Step

We need to make a specialized scheme for the first step, that is, computing $u_{i,j}^{1}$, as we can not use $u_{i,j}^{-1}$. Knowing the time derivative at $n = 0$ to be $V(x, y)$, we can calculate a value for $u_{i,j}^{-1}$:

$$[D_{2t}u]_{i,j}^{0} = \frac{u^1 - u^{-1}}{2\Delta t} = V(x,y)$$
$$\Rightarrow -u_{i,j}^{-1} = V(x,y)2\Delta t - u_{i,j}^{1}$$

Introducing the help value

$$\beta = \frac{\left(1 - \frac{b\Delta t}{2}\right)}{\left(1 + \frac{b\Delta t}{2}\right)}$$

our and inserting $n = 1$ in our general scheme, we get:

$$\begin{aligned} &u_{i,j}^{1} = \\ &\left([D_xqD_xu]_{i,j}^{0} + [D_yqD_yu]_{i,j}^{0} + f_{i,j}^{0}\right)\frac{\Delta t^2}{\left(1 + \frac{b\Delta t}{2}\right)} \\ &+ \frac{2u_{i,j}^{0}}{\left(1 + \frac{b\Delta t}{2}\right)} + (2V(x,y)\Delta t - u_{i,j}^{1})\beta \end{aligned}$$
$$\Downarrow$$
$$\begin{aligned} &u_{i,j}^{1}(1+\beta) = \\ &\left([D_xqD_xu]_{i,j}^{0} + [D_yqD_yu]_{i,j}^{0} + f_{i,j}^{0}\right)\frac{\Delta t^2}{\left(1 + \frac{b\Delta t}{2}\right)} \\ &+ \frac{2u_{i,j}^{0}}{\left(1 + \frac{b\Delta t}{2}\right)} + 2V(x,y)\Delta t\beta \end{aligned}$$

And finally our special scheme for the initial step becomes:

$$\begin{aligned} &u_{i,j}^{1} = \\ &\frac{\left([D_xqD_xu]_{i,j}^{0} + [D_yqD_yu]_{i,j}^{0} + f_{i,j}^{0}\right)}{\left(1 + \frac{b\Delta t}{2}\right)(1+\beta)}\Delta t^2 \\ &+ \frac{2u_{i,j}^{0}}{\left(1 + \frac{b\Delta t}{2}\right)(1+\beta)} + \frac{2V(x,y)\Delta t\beta}{(1+\beta)} \end{aligned} \tag{4}$$

Again with $[D_xqD_xu]_{i,j}^{n}$ and $[D_yqD_yu]_{i,j}^{n}$ as defined in (2).

## D. Handeling Boundry Conditions

The boundry condtidtion in this problem is given by $\frac{\partial u}{\partial n} = 0$, meaning reflecting boundries in the space domain. There are different ways of implementing this in the discrtized solution. I have in this project solved it by the use of ghost cells. In both $x$ anf $y$ direction, we make one extra grid points in both ends, using this point to compute the true end points, which lies next to the ghost cell. This point shuld be equal to the point two cells in, imitating reflection at the end point. This is solved by setting:

$$\begin{aligned} u^n_{-1,j} &= u^n_{1,j} \\ u^n_{N_x+1,j} &= u^n_{N_x-1,j} \qquad \text{for all} \quad i,j,n \\ u^n_{i,-1} &= u^n_{i,1} \\ u^n_{i,N_y+1} &= u^n_{i,N_y-1} \end{aligned}$$

## III. Implementation

For the implementation of the numerical scheme, I have chosen to write a class, **wave2D** that can be found in `wave2D.py`. The class is initialized with the functions $q(x,y)$ and $f(x,y,t)$, and the constant $b$, required to define the equation (1). It is used by the call of the **solve** method, which takes the initial condition functions, the length of the space grid, and number of grid points in each direction, the time step $\Delta t$ and the total time T. Additionally, the key word argument **version** can be either "scalar" or "vectorized", and decides wether to solve the equations elementwise with nested loops, or vectoriwise with the use of numpy arrays. The method stores the wave in the multi-dimentional array `u`, that takes three indexes. The index `i` in x-direction, the index `j` in y-direction, and the index `n` of the point in time equal to `t[n]`. After the solve method has been called, a specific point in space and time of the wave can be accessed with the `call` method.

The solve methods makes use of several private help functions, in order to make the the class more readable. The `_set_initial_condtition` sets `u[:,:,0]` equal to the initial function $I(x,y)$. The `_initial_step`-function executes the first iteration of the solution, using the special scheme for the initial step (4). After that the one of the two advance functions `_advance_scalar` or `_advance_vectorized` calculates the rest of the solution according to (3), depening on the value of `version`. All the funtions computing the solution uses the two functions `_DxqDxu` and `_DxqDxu`, which are help functions to calculate $[D_xqD_xu]^n_{i,j}$ and $[D_yqD_yu]^n_{i,j}$ as defined in (2).

When the solve method has been called, the x,y,and t-array plus the discretization parameter $h$ and the solution $u$ can be accessed as properties of the class instance.

## IV. Verification

In order to test that the class behaves as it should, I have implemented several nosetests, checking for various erros that can occur. They are all in the script `test_wave2D`. The scirpt includes numerous tests, among other thigs testing that the vectorized and scalar solution is the same, and a test for a plug wave and a manufactured solution with convergence rate. For information on how to run the tests, see the code documentation.

### A. Constant Solution

The first thing we test is that the solution remains constant for for a case in which the true solution is $u(x,y,t) = c$. A simple way to achieve this is to choose $b = 0$, $f(x,y,z) = 0$, $V(x,y) = 0$, $q$ as an arbitrary constant and $I(x,y) = c$. By running this for a certain time, and then checking that the last time point in the solution is the same as the intial for the entire grid we can make sure that the discrete solution is stable, which is the case for the wave2D class.

By playing around with the class, we can however, invent errors and see if they lead to wrong, non-constant solution for the above constants and functions by checking if the test fails. It turns out that

- changing from `-beta*u[i,j,n-1]` to `+beta*u[i,j,n-1]` when calculationg `u[i,j,n+1]` makes the solution non-constant.
- changing `q_plus_half` to be `q_minus_half` and the other way around does not cause a non-constant solution.

- Failing to set the value of the ghost cells for each time step will give a non-constant solution.
- adding a factor to $f$ or changing the sign in front of it in the scheme will not give a non-constant solution.

## B. Plug Wave

Creating a plug wave by letting the intial condtion $I(x,y)$ be a pulse that is 1 for a given x or y-coordinate in the center, but 0 elsewhere, gives rise to two identical waves, traveling in opposite directions. Choosing a wave velocity $c$ and setting $q(x,y) = c^2$, we simulate the wave using $\Delta t = \frac{\Delta x}{c}$ for a wave in x-direction and $\Delta t = \frac{\Delta y}{c}$ in y-direction. If $c = 0.5$ we expect the wave to be back and equal to the initial pulse after one period which is then $T = 4$. The `test_plug_wave()` function in the test file checks that this is indeed the case for both x- and y- direction, which it is using the wave2D class. In the folder **figs** there are movies showing a simulation of both plug waves, which can also be produced by running the `plot_wave.py` file.

## C. Standing, Undamped Waves and Convergence

The next thing we want to test is how the error of the solution converges for a given case. A standing, undamped wave has the exact solution

$$\begin{aligned} u_e(x,y,t) =& A\cos(k_x x)\cos(k_y y)\cos(\omega t) \\ k_* =& \frac{m_* \pi}{L_*} \end{aligned} \tag{5}$$

Which means that the intial conditions become

$$\begin{aligned} I(x,y) &= u_e(x,y,0) = A\cos(k_x x)\cos(k_y y) \\ V(x,y) &= \frac{\partial u_e(x,y,t=0)}{\partial t} = 0 \end{aligned}$$

We use $f(x,y,t) = 0$, $b = 0$, $q(x,y) = c^2$ and are free to set the other constants as we choose. Setting $\omega = 2\pi$ is convenient as one period then is $T = 1$. For a visualization of the standing wave solution, see the movie in the **figs** folder, or run the script `plot_wave.py`.

In order to compute the convergence of the error, we introduce the discretization parameter $h$ that is proportional to both $\Delta t, \Delta x$ and $\Delta y$. I choose to simply use $h = \Delta x \Delta y \Delta t$. Defining the error as the difference between the exact solution and the discrete solution

$$e^n_{i,j} = u_e(x,y,t) - u^n_{i,j}$$

we use the error norm

$$E = ||e^n_{i,j}||_{\text{inf}}$$

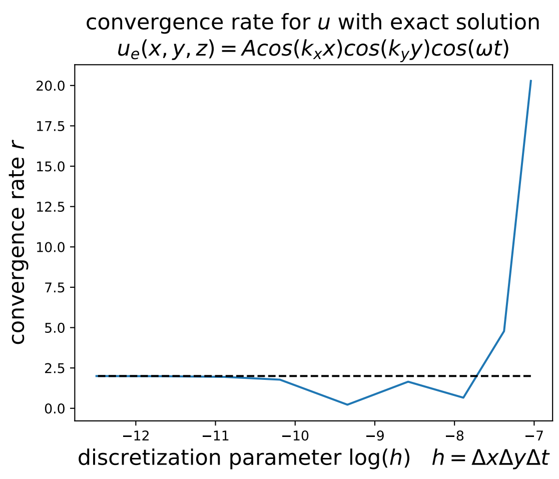
Be the maximum error at any grid point at any time for the given solution. We then expect the error to behave as

$$E = Ch^r$$

Where $h$ is the discretization parameter, $C$ is a constant and $r$ is the convergence rate. We expect the convergence rate to be $r = 2$. To compute what our $r$ is we use (by some logarithms and algebra):

$$r = \frac{\log\left(\frac{E_1}{E_2}\right)}{\log\left(\frac{h_1}{h_2}\right)} \tag{6}$$

Where $h_1$ and $h_2$ are a result of different choices of step sizes, and $E_1$ and $E_2$ are their corresponding error norms. By choosing a set of different values by $h$ by varying the step sizes of the grid or diffrent $\Delta t$ we can compute a series of error norms and corresponding convergence rates. This is done in `convergence.py`, with the exact solution given by (5) and the discrete solution found with wave2D. The results are the following:



```
convergence rates:

[20.283731631740096,
4.780395669166134,
0.6580368669323325,
1.6491890724515759,
0.2240277019640846,
1.7755267247397404,
1.9459428676739572,
1.9886536112659243,
1.9978620702744785]
```

As we clearly see, $r$ moves towards 2 as $h$ decreases, and a converge rate of 2 is what we expect, so this means that the class behaves as it should, and is good news.

## D. Manufactured Solution of Standing, Damped Waves

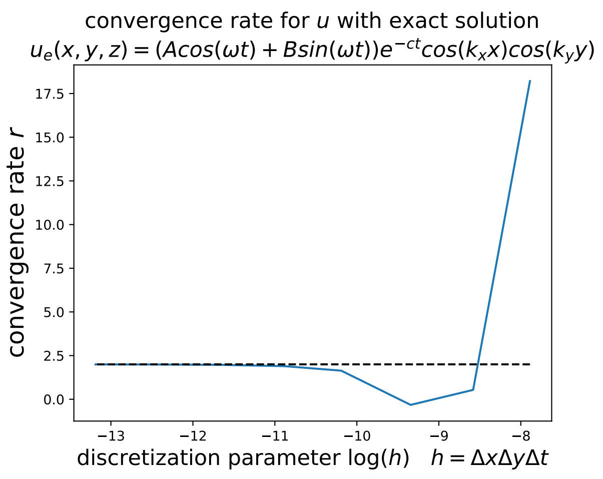
This time looking at standing, damped waves with exact solution:

$$u_e(x,y,t) = (A\cos(\omega t) + B\sin(\omega t))e^{-ct}\cos(k_x x)\cos(k_y y) \tag{7}$$

We wish to make a manufactured solution that will make (1) fulfill this $u_e$. To do this we choose $q(x,y) = 1$ and $b = 0.5$, and I made the script source_term.py that uses sympy to do the numerical work and compute the corresponding source term $f(x,y,z)$. Doing this, using $\omega = 1$ we get the source term and initial functions from the output when running the script

```
-----------------------
f(x,y,t) = (-0.5*A*sin(t) - A*cos(t)
- B*sin(t) + 0.5*B*cos(t) + c**2*(A*cos(t)
+ B*sin(t)) + 2*c*(A*sin(t) - B*cos(t))
- 0.5*c*(A*cos(t) + B*sin(t)) + kx**2*(A*cos(t)
+ B*sin(t)) + ky**2*(A*cos(t)
+ B*sin(t)))*exp(-c*t)*cos(kx*x)*cos(ky*y)
-----------------------
I(x,y) = A*cos(kx*x)*cos(ky*y)
V(x,y) = (-A*c + B*w)*cos(kx*x)*cos(ky*y)
-----------------------
```

Using this $f$, $q$, $\omega$ and $b$ we are free to choose the other constants. The we use the same method as for undamped waves, comparing the exact solution to the discrete equation, and computing the convergence rate. This is also done in the convergence.py file. The results are the following:



```
convergence rates:

[18.212589404576526,
0.537333534690724,
-0.32099363324969976,
1.6385238225264025,
1.896197292167576,
1.9705612152594651,
1.9908815444580714,
1.9961621626844892]
```

Like with the undamped case, we see that the convergence rate moves towards $r = 2$, making this our ecpected converge rate. We use this to implement another test for our class, asserting that for small enough steps, checking two different values of $h$ and their corresponding $E$, should give a converge rate close to $r \approx 2$, using the same source term and constant values. This is done in the `test_convergence_manufactured_solution` function in the test script, and it does indeed pass.

This concludes the verification of the model, although several other tests can probably be made. All of the implemented tests pass, which is a good indication that our class does indeed work.

## V. Investigation of a Physical Problem

We are now going to have a look at a physical problem with variable wave velocity, in which a tsunami goes over a subsea hill. The wave velocity is given by the gravity $g$ times the hight between the sea surface $I_0$ when the water is still and the seabottom described by a function $B(x,y)$, resulting in the wave velocity being

$$q(x,y) = gH(x,y) = g(I_0 - B(x,y))$$

For details on the initial condition formula and the equations for the various shapes, see the project instructions which is also in the git repository.

Using the wave2D class, we simulate the different bottom shapes and the consequences it has for the waves. This is done in the `tsuami.py` file, where one can se the different functions used to model the bottom shapes.

In this case I believe the movies, also found in the figs folder, speak well for themselves. I noticed that for a more steep hill, a lot smaller steps were needed. Also, for the gaussian hill, when $b = 5$, so the wave was met by an even hill, the solution was a lot more stable when $b = 1$, when the hill was rounded.