waveEquation

Milad H. Mobarhan & Svenn-Arne Dragly

October 14, 2013

1 Project 2: 2D wave equation

Summary. The aim of this project is to develop a solver for the two-dimensional, standard, linear wave equation, with damping, and verify the solver.

```
In [1]: %cd -q ../../src/
```

1.1 Mathematical problem

The general wave equation in d space dimensions, with variable coefficients, can be written in the compact form

$$\varrho\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = \nabla\cdot(q\nabla u) + f \text{ for } \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\ t\in(0,T],$$

which in 2D becomes

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

To save some writing and space we may use the index notation, where subscript t, x or y means differentiation with respect to that coordinate, i.e.

$$\begin{aligned}
u_t &= \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2}\\
u_x &= \frac{\partial u}{\partial x}, \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}\\
u_y &= \frac{\partial u}{\partial t}, \quad u_{yy} = \frac{\partial^2 u}{\partial y^2}\\
(qu_x)_x &= \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right)\\
(qu_y)_y &= \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right)\,.
\end{aligned}$$

The 2D versions of the two model PDEs, with and without variable coefficients, can with now with the aid of the index notation for differentiation be stated as

$$\varrho u_{tt} + bu_t = (qu_x)_x + (qu_z)_z + (qu_z)_z + f$$

For our project, we will use $\varrho = 1$.

Since this PDE contains a second-order derivative in time, we need two initial conditions; (1) the initial shape of the string, I, and (2) the initial velocity of the string, V;

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

In addition, PDEs need boundary conditions, which are of three principal types:

- Dirchlet boundary condition: u is given for the boundaries ($u = 0$ or a known time variation for an incoming wave).
- Neumann boundary condition: $\partial u/\partial n = \boldsymbol{n} \cdot \nabla u$ (zero for reflecting boundaries).
- Open boundary condition (also called radiation condition): Specified to let waves travel undisturbed out of the domain.

We will in this project focus on the Neumann boundary condition.

1.2 Discretization

In this section we will derive the discrete set of equations to be implemented in a program. We will for simplicity assume constant spacing between the mesh points. Our mesh points are

$$x_i = i\Delta x,\ i = 0, \ldots, N_x, y_j = j\Delta y,\ j = 0, \ldots, N_y, t_i = n\Delta t,\ n = 0, \ldots, N_t\,.$$

Note that this results in a number of $N_x + 1$ mesh points in x-direction, and $N_y + 1$ and $N_n + 1$ for y and t. Our goal is to find the solution $u(x, y, t)$ that fulfills the above stated wave equation for all mesh points. We introduce the mesh function $u^n_{i,j}$, which approximates the exact solution at the mesh point (x_i, y_j, t_n) for $i = 0, \ldots, N_x$, $j = 0, \ldots, N_y$ and $n = 0, \ldots, N_t$.

Replacing derivatives by finite differences

The second-order derivatives can be replaced by central differences. The most widely used difference approximation of the second-order derivative is

$$\frac{\partial^2}{\partial t^2} u(x_i, y_j, t_n) \approx \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2}\,.$$

It is convenient to introduce the finite difference operator notation

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

The first-order derivative in time in the damping term can be approximated by

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

Discretizing the variable coefficient

The principal idea is to first discretize the outer derivative. We define

$$\phi^x = q(x,y)\frac{\partial u}{\partial x},$$

and use a centered derivative around $x = x_i$ for the derivative of ϕ:

$$\left[\frac{\partial \phi^x}{\partial x}\right]_i^n \approx \frac{\phi^x_{i+\frac{1}{2}} - \phi^x_{i-\frac{1}{2}}}{\Delta x} = [D_x \phi^x]_i^n .$$

We then discretize ϕ^x further as

$$\phi^x_{i+\frac{1}{2}} = q_{i+\frac{1}{2}} \left[\frac{\partial u}{\partial x}\right]^n_{i+\frac{1}{2}} \approx q_{i+\frac{1}{2}} \frac{u^n_{i+1} - u^n_i}{\Delta x} = [qD_x u]^n_{i+\frac{1}{2}} .$$

Similarly,

$$\phi^x_{i-\frac{1}{2}} = q_{i-\frac{1}{2}} \left[\frac{\partial u}{\partial x}\right]^n_{i-\frac{1}{2}} \approx q_{i-\frac{1}{2}} \frac{u^n_i - u^n_{i-1}}{\Delta x} = [qD_x u]^n_{i-\frac{1}{2}} .$$

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u^n_{i+1} - u^n_i\right) - q_{i-\frac{1}{2}}\left(u^n_i - u^n_{i-1}\right)\right)$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_x q D_x u]^n_i$$

Similarly for

$$\phi^y = q(x,y)\frac{\partial u}{\partial y}.$$

we have

$$\left[\frac{\partial}{\partial y}\left(q\frac{\partial u}{\partial y}\right)\right]_i^n \approx [D_y q D_y u]^n_i$$

In order to compute $[D_x q D_x u]^n_i$ and $[D_y q D_y u]^n_i$ we need to evaluate $q_{i\pm\frac{1}{2},j}$ and $q_{i,j\pm\frac{1}{2}}$. If q is a known function, we can easily evaluate $q_{i\pm\frac{1}{2},j}$ simply as $q(x_{i\pm\frac{1}{2}}, y_j)$ with $x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$. However, in many cases q, is only known as a discrete function, and evaluating must be done by averaging. The most commonly used averaging technique is the arithmetic mean:

$$q_{i+\frac{1}{2},j} \approx \frac{1}{2}\left(q_{i+1,j} + q_{i,j}\right) = [\overline{q}^x]_{i+\frac{1}{2},j}, q_{i-\frac{1}{2},j} \approx \frac{1}{2}\left(q_{i,j} + q_{i-1,j}\right) = [\overline{q}^x]_{i-\frac{1}{2},j},$$

which we are going to use in this project.

Algebraic version of the initial conditions

The initial conditions are given as

$$u(x, y, t_0) = I(x, y),$$
$$u_t(x, y, t_0) = V(x, y).$$

The first condition can be computed by

$$u_i^0 = I(x_i), \quad i = 0, \ldots, N_x \,.$$

For the second one we use a centered difference of type

$$\frac{\partial}{\partial t} u(x_i, y_j, t_0) \approx \frac{u_{i,j}^1 - u_{i,j}^{-1}}{2\Delta t} = [D_{2t} u]_{i,j}^0 = V(x_i, y_j).$$

Algebraic version of the PDE

Interior spatial mesh points

The PDE with variable coefficients is discretized term by term:

$$[\varrho D_t D_t u + b D_{2t} u = (D_x \overline{q}^x D_x u + D_y \overline{q}^y D_y u) + f]_{i,j}^n \,.$$

When written out and solved for the unknown $u_{i,j}^{n+1}$ one gets the scheme

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

Modified scheme for the first step

A problem with algebraic version of the PDE equation arises when $n = 0$ since the formula for $u_{i,j}^1$ involves $u_{i,j}^{-1}$, which is an undefined quantity outside the time mesh (and the time domain). However, we can use the initial condition to arrive at a special formula for $u_{i,j}^1$. From initial condition we have

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

which may be rewritten to find $u_{i,j}^{-1}$ as

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

Inserting this into the algebraic version of the PDE equation for $n = 0$ gives:

$$u^1_{i,j} = \Bigg\{ 2u^0_{i,j} + \left(u^1_{i,j} - 2\Delta t V_{i,j}\right)\left[\frac{b\Delta t}{2\varrho_{i,j}} - 1\right] + \frac{\Delta t^2}{\varrho_{i,j}} f^0_{i,j}$$
$$+ \frac{1}{\varrho_{i,j}}\frac{\Delta t^2}{\Delta x^2}\left[\frac{1}{2}(q_{i,j}+q_{i+1,j})(u^0_{i+1,j}-u^0_{i,j}) - \frac{1}{2}(q_{i-1,j}+q_{i,j})(u^0_{i,j}-u^0_{i-1,j})\right]$$
$$+ \frac{1}{\varrho_{i,j}}\frac{\Delta t^2}{\Delta y^2}\left[\frac{1}{2}(q_{i,j}+q_{i,j+1})(u^0_{i,j+1}-u^0_{i,j}) - \frac{1}{2}(q_{i,j-1}+q_{i,j})(u^0_{i,j}-u^0_{i,j-1})\right]\Bigg\}\left(\frac{1}{1+\frac{b\Delta t}{2\varrho_{i,j}}}\right).$$

Thus, the special formula for $u^1_{i,j}$ is

$$u^1_{i,j} = \frac{1}{2}\Bigg\{ 2u^0_{i,j} - 2\Delta t V_{i,j}\left[\frac{b\Delta t}{2\varrho_{i,j}} - 1\right] + \frac{\Delta t^2}{\varrho_{i,j}} f^0_{i,j}$$
$$+ \frac{1}{\varrho_{i,j}}\frac{\Delta t^2}{\Delta x^2}\left[\frac{1}{2}(q_{i,j}+q_{i+1,j})(u^0_{i+1,j}-u^0_{i,j}) - \frac{1}{2}(q_{i-1,j}+q_{i,j})(u^0_{i,j}-u^0_{i-1,j})\right]$$
$$+ \frac{1}{\varrho_{i,j}}\frac{\Delta t^2}{\Delta y^2}\left[\frac{1}{2}(q_{i,j}+q_{i,j+1})(u^0_{i,j+1}-u^0_{i,j}) - \frac{1}{2}(q_{i,j-1}+q_{i,j})(u^0_{i,j}-u^0_{i,j-1})\right]\Bigg\}.$$

1.3 Boundary condition

In a rectangular spatial domain $\Omega = [0, L_x] \times [0, L_y]$, the homogeneous Dirichlet condition, is given by

$$u^1_{i,j} = u^{n+1}_{i,j} = 0$$

for $n = 1, 2, \ldots, N_t - 1$, when $i = 0,\ i = N_x,\ j \in [0, L_y]$ and when $j = 0,\ j = N_y,\ i \in [0, L_x]$. Note that $u^0_{i,j}$ is given by I, i.e.

$$u^0_{i,j} = I(x_i, y_j) \quad \text{for } i = 0, \ldots, N_x \quad j = 0, \ldots, Ny$$

The Neumann boundary condition is given by

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial y} = 0, \tag{1}$$

at $x = 0, N_x$ and $y = 0, N_y$. Since we have used central differences in all the other approximations to derivatives in the scheme, it is tempting to implement this condition by the difference

$$[-D_{2x}u = 0]^n_{0,j}, \quad [D_{2x}u = 0]^n_{L_x,j} \Rightarrow \quad u^n_{-1,j} = u^n_{1,j}, \quad u^n_{N_x+1,j} = u^n_{N_x-1,j}$$
$$[-D_{2y}u = 0]^n_{i,0}, \quad [D_{2y}u = 0]^n_{i,L_y} \Rightarrow \quad u^n_{i,-1} = u^n_{i,1}, \quad u^n_{i,N_y+1} = u^n_{i,N_y-1}\,.$$

The problem is that $u^n_{-1,j}$, $u^n_{i,-1}$, $u^n_{N_x+1,j}$ and u^n_{i,N_y+1} are not a u values that are being computed since these points are outside the mesh. However we can use the relations given above and the algebraic version of the PDE equation to arrive at a modified scheme for the boundary points. The implementation of the special formulas for the boundary points can benefit from using the general formula for the interior points also at the boundaries, but replacing $u^n_{i-1,j}$ by $u^n_{i+1,j}$ when computing $u^{n+1}_{i,j}$ for $i = 0$, and $u^n_{i+1,j}$ by $u^n_{i-1,j}$ when computing $u^{n+1}_{i,j}$ for $i = N_x$. A similar modification must be done for $j = 0, N_y$.

Implementation of Neumann conditions via ghost cell

Instead of modifying the scheme at the boundary, we can introduce extra points outside the domain such that the fictitious values $u^n_{-1,j}$, $u^n_{i,-1}$, $u^n_{N_x+1,j}$ and u^n_{i,N_y+1} are defined in the mesh. Thus we add the points $i = -1$, $i = N_x + 1$, $j = -1$ and $j = N_y + 1$, which are known as ghost points, and values at these points, are known as ghost points. The important idea is to ensure that we always have

$$u^n_{-1,j} = u^n_{1,j}, \quad u^n_{N_x+1,j} = u^n_{N_x-1,j}$$
$$u^n_{i,-1} = u^n_{i,1}, \quad u^n_{i,N_y+1} = u^n_{i,N_y-1}$$

because then the application of the standard scheme at a boundary point will be correct and guarantee that the solution is compatible with the boundary condition $u_x = 0$ and $u_y = 0$.

1.4 Truncation error

Truncation error analysis provides a widely applicable framework for analyzing the accuracy of finite difference schemes and is a measure for the extent the exact solution u_e fits the discrete equations. We define

$$R = \mathcal{L}_\Delta(u_\mathrm{e}),$$

where the residual R is known as the truncation error of the finite difference scheme $\mathcal{L}_\Delta(u) = 0$.

The algebraic version of the wave equation with variable coefficients and damping can be written as:

$$[\varrho D_t D_t u + bD_{2t}u = (D_x\overline{q}^x D_x u + D_y\overline{q}^y D_y u) + f]^n_{i,j}\,.$$

Inserting the exact solution $u_e(x, y, t)$ in this equation makes this function fulfill the equation if we add the term R:

$$[\varrho D_t D_t u_e + bD_{2t}u_e = (D_x\overline{q}^x D_x u_e + D_y\overline{q}^y D_y u_e) + f + R]^n_{i,j}\,.$$

Our purpose is to calculate the truncation error R and this can be done by taylor expantion of the terms in the PDE:

$$[D_t D_t u_\mathrm{e}]^n_{i,j} = u_{\mathrm{e},tt}(x_i, y_j, t_n) + \frac{1}{12}u_{\mathrm{e},tttt}(x_i, y_j, t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)$$

$$[D_{2t}u_e]^n_{i,j} = u_{\mathrm{e},t}(x_i, y_j, t_n) + \frac{1}{6}u_{\mathrm{e},ttt}(x_i, y_j, t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)$$

$$[D_x\overline{q}^x D_x u_\mathrm{e}]^n_{i,j} = (qu_{\mathrm{e},x})_x + \mathcal{O}(\Delta x^2)\,.$$

$$[D_y\overline{q}^y D_y u_\mathrm{e}]^n_{i,j} = (qu_{\mathrm{e},y})_y + \mathcal{O}(\Delta y^2)\,.$$

Inserting all these terms into the discrete equation gives us

$$\varrho\left(u_{\mathrm{e},tt}(x_i, y_j, t_n) + \frac{1}{12}u_{\mathrm{e},tttt}(x_i, y_j, t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\right) + b\left(u_{\mathrm{e},t}(x_i, y_j, t_n) + \frac{1}{6}u_{\mathrm{e},ttt}(x_i, y_j, t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\right)$$
$$= (q(x_i, y_j)u_{\mathrm{e},x}(x_i, y_j, t_n))_x + \mathcal{O}(\Delta x^2) + (q(x_i, y_j)u_{\mathrm{e},y}(x_i, y_j, t_n))_y + \mathcal{O}(\Delta y^2) + f(x_i, y_j, t_n) + R^n_{i,j}$$

Because u_e fulfills the partial differential equation (PDE):

$$\varrho u_{tt} + bu_t = (qu_x)_x + (qu_z)_z + (qu_z)_z + f$$

some of the terms cancel out, and we are left with

$$R^n_{i,j} = \mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta y^2) + \varrho \left(\frac{1}{12} u_{\mathrm{e},tttt}(x_i, y_j, t_n)\Delta t^2 \right) + b \left(\frac{1}{6} u_{\mathrm{e},ttt}(x_i, y_j, t_n)\Delta t^2 \right)$$

The scheme is of second-order in time and space. The key ingredients for second order are the centered differences and the arithmetic mean for q: all those building blocks feature second-order accuracy.

1.5 Verification: Constant

This is implemented and works as expected:

In [2]:
```
%run tests/test_wm_constantSolution.py
```

```
------------------test constant solution----------------
vec:  test_constantSolution succeeded!
scalar:  test_constantSolution succeeded!
```

1.6 Verification: Cubic solution

We assume an exact solution

$$u_e(x, y, t) = X(x)Y(y)T(t)$$

where X, Y and T are polynomials of degree three or less. A solution that obeys the boundary condition of a vanishing normal derivative is found by setting

$$X(x) = (2x^3 - 3L_x x^2)$$
$$Y(y) = (2y^3 - 3L_y y^2)$$

and

$$T(t) = t$$

The final choice of $T(t) = t$ simplifies the equation. Inserting this solution into the wave equation with $q(x, y) = q$ and $b = 0$ we find

$$f(x, y, t) = -q(12x - 6L_x)Y(y)T(t) - qX(x)(12y - 6L_y)T(t)$$

which may now be used to verify the cubic solution.

On the boundary, we need to check if we need to compensate for our discrete implementation of the boundary condition $\frac{\partial}{\partial n}u_e$. This was implemented as

$$[D_{2x}u]_i^n = 0$$

for $i = 0$ and $i = N_x$. This does not hold for our cubic solution, because

$$\begin{aligned}[D_{2x}u_e]_i^n &= \frac{u_e(x_i + \Delta x) - u(x_i - \Delta x)}{2\Delta x} \\ &= (6x_i^2 - 6L_x x_i + 2\Delta x^2)\end{aligned}$$

On the boundaries, this reduces to

$$[D_{2x}u_e]_i^n = 2\Delta x^2, \qquad i = \{0, N_x\}$$

We now have that, for the ghost points

$$\begin{aligned}u_{-1}^n &= u_1^n - 2\Delta x^2 \\ u_{N_x+1}^n &= u_{N_x-1}^n + 2\Delta x^2\end{aligned}$$

Which together with the iterative scheme gives us a correction to f, which we will call g, so that on the boundary our source term is $\hat{f} = f + g$. This makes the solution exact for the discrete equation on the boundaries too:

$$\begin{aligned}g_{-1}^n &= -4\Delta x T(t)Y(y)q \\ g_{N_x+1}^n &= 4\Delta x T(t)Y(y)q\end{aligned}$$

This is implemented in the code with a check for vectorized or scalar versions of x and y when calling f, and the correction is then added as $\hat{f} = f + g$ if we are on the boundary.

In [3]:
```
%run tests/test_wm_cubicSolution.py
```

```
------------------test cubic solution-------------------
vec:  test_cubicSolution succeeded!
scalar:  test_cubicSolution succeeded!
```

1.7 Verification: Plug wave

This is implemented by inserting a plug wave and checking that we get back initial wave after one period. This is tested and works:

In [4]:
```
%run tests/test_wm_plugwaveSolution.py
```

```
--------------test plug wave solution---------------
vec-x: test_plugwaveSolution succeeded!
vec-y: test_plugwaveSolution succeeded!
```

```
scalar-x: test_plugwaveSolution succeeded!
scalar-y: test_plugwaveSolution succeeded!
```

1.8 Verification: Standing, undamped waves

The verification of the standing undamped waves is done directly by inserting the exact solution into the solver and letting it run for a limited number of time steps. This needs to be limited because the discrete solution will have a frequency $\tilde{\omega}$ that is different from the real ω. Eventually this will lead the error to be exactly the amplitude of the wave, which is not interesting to measure.

Instead we measure the error after a shorter time and this is found to converge as expected:

In [5]:
```
%run tests/test_wm_standingUndampedSolution.py
```

```
------------------test standing undamped----------------
vec: test_standingUndampedSolution succeeded!
scalar: test_standingUndampedSolution succeeded!
```

1.9 Verification: Standing, damped waves

In [6]:
```
from sympy import *
from IPython.display import display
init_printing(use_latex=True)
A,B,b,w,kx,ky,  q, d = symbols("A,B,b,omega,k_x,k_y,q,d")
x,y,t = symbols("x,y,t")
```

The expression for u and u inserted into the wave equation are, respectively:

In [7]:
```
u = (A*cos(w*t)+B*sin(w*t))*exp(-d*t)*cos(kx*x)*cos(ky*y)
u_t = diff(u,t).simplify()
u_tt = diff(u_t,t).simplify()
u_xx = diff(u,x,x).simplify()
u_yy = diff(u,y,y).simplify()
eq  = u_tt + b*u_t - q*u_xx - q*u_yy
display(u, eq)
```

$$\left(A \cos\left(\omega t\right) + B \sin\left(\omega t\right)\right) e^{-dt} \cos\left(k_x x\right) \cos\left(k_y y\right)$$

$$b\left(-Ad\cos\left(\omega t\right) - A\omega \sin\left(\omega t\right) - Bd\sin\left(\omega t\right) + B\omega \cos\left(\omega t\right)\right) e^{-dt} \cos\left(k_x x\right) \cos\left(k_y y\right) - k_x^2 q \left(-A\cos\left(\omega\right.\right.$$

The initial conditions are used to find B:

In [8]:
```
I = u.subs(t,0)
V = u_t.subs(t,0)
display(I,V)

B_sol = solve(V,B)
eq_B = eq.subs(B,B_sol[0])
display("B = ", B_sol)
```

$$A\cos\left(k_x x\right)\cos\left(k_y y\right)$$

$$\left(-Ad + B\omega\right)\cos\left(k_x x\right)\cos\left(k_y y\right)$$

```
B =
```

$$\left[\frac{Ad}{\omega}\right]$$

PDE at $t = 0$:

In [9]:
```
u_t_t0  = u_t.subs(t, 0)
u_tt_t0 = u_tt.subs(t, 0)
u_xx_t0 = u_xx.subs(t, 0)
u_yy_t0 = u_yy.subs(t, 0)
eq_t0 = eq_B.subs(t,0).simplify()
display(eq_t0)
```

$$A\left(-d^2 + k_x^2 q + k_y^2 q - \omega^2\right)\cos\left(k_x x\right)\cos\left(k_y y\right)$$

We use this to find an expression for ω:

In [10]:
```
w_sol = solve(eq_t0,w)
w_sol
```

Out [10]:
$$\left[-\sqrt{-d^2 + k_x^2 q + k_y^2 q},\quad \sqrt{-d^2 + k_x^2 q + k_y^2 q}\right]$$

PDE at t, with B and ω inserted:

In [11]:
```
eq_B_w = eq_B.subs(w, w_sol[1])
```

Solving for c:

In [12]:
```
d_sol = solve(eq_B_w, d)
d_sol
```

Out [12]:
$$\left[\frac{1}{2}b\right]$$

We insert the above expressions for d, ω and B into our exact solution and insert this into the solver. The error converges as expected.

In [13]:
```
%run tests/test_wm_standingDampedSolution.py
```

```
------------------test standing damped----------------
vec: test_standingDampedSolution succeeded!
scalar: test_standingDampedSolution succeeded!
```

1.10 Verification: Manufactured solution

In [14]:
```
from sympy import *
init_printing(use_latex=True)
A,B,b,w,kx,ky,  q, d = symbols("A,B,b,w,k_x,k_y,q,d")
x,y,t = symbols("x,y,t")
```

In [15]:
```
u = (A*cos(w*t)+B*sin(w*t))*exp(-d*t)*cos(kx*x)*cos(ky*y)
u_t = diff(u,t).simplify()
u_tt = diff(u_t,t).simplify()

ls = (u_tt +b*u_t).simplify().collect(sin(w*t)).collect(cos(w*t))
display(ls)
```

$$\left(\left(-Abd + Ad^2 - Aw^2 + Bbw - 2Bdw\right)\cos\left(tw\right) + \left(-Abw + 2Adw - Bbd + Bd^2 - Bw^2\right)\sin\left(tw\right)\right)$$

Initial conditions:

In [16]:
```
I = u.subs(t,0)
V = u_t.subs(t,0)
display(I,V)
```

$$A\cos\left(k_x x\right)\cos\left(k_y y\right)$$

$$\left(-Ad + Bw\right)\cos\left(k_x x\right)\cos\left(k_y y\right)$$

Choice of q:

In [17]:
```
q = x
q
```

Out [17]:

$$x$$

Finding the derivatives on the right hand side:

In [18]:
```
u_xx= diff(q*diff(u, x), x).simplify()
u_yy= diff(q*diff(u, y), y).simplify()
rs = (u_xx + u_yy).simplify()
display(u_xx,u_yy,rs)
```

$$k_x\left(A\cos\left(tw\right) + B\sin\left(tw\right)\right)\left(-k_x x\cos\left(k_x x\right) - \sin\left(k_x x\right)\right)e^{-dt}\cos\left(k_y y\right)$$

$$k_y^2 x\left(-A\cos\left(tw\right) - B\sin\left(tw\right)\right)e^{-dt}\cos\left(k_x x\right)\cos\left(k_y y\right)$$

$$\left(A\cos\left(tw\right) + B\sin\left(tw\right)\right)\left(-k_x\left(k_x x\cos\left(k_x x\right) + \sin\left(k_x x\right)\right) - k_y^2 x\cos\left(k_x x\right)\right)e^{-dt}\cos\left(k_y y\right)$$

Finding f:

In [19]:
```
f = ls-rs
display(f)
```

$$-\left(A\cos\left(tw\right) + B\sin\left(tw\right)\right)\left(-k_x\left(k_x x\cos\left(k_x x\right) + \sin\left(k_x x\right)\right) - k_y^2 x\cos\left(k_x x\right)\right)e^{-dt}\cos\left(k_y y\right) + \left(\left(-A\right.\right.$$

We insert the exact solution and our expression for f in to the solver and find that the error converges as expected:

In [20]:
```
%run tests/test_wm_manufacturedSolution.py
```

```
---------------test manufactured solution----------------
vec: test_manufacturedSolution succeeded!
scalar:
```

1.11 Physical problem

We investigated a couple of physical problems, with the goal of simulating water waves on different types of subsea geometries. All simulations were performed over a total time of $T = 5$ and with a timestep $\Delta t = 0.005$ on a domain of $L_x = L_y = 1$. A selected few of the results from these simulations are shown below:

In [21]:
```
from IPython.display import YouTubeVideo
from IPython.display import display
from IPython.core.display import Latex
def youTubeVideo(videoID, width=640, height=480):
    display(YouTubeVideo(videoID, width=width, height=height))
    display(Latex("Can't see the video? Click here: \url{http://youtube.co
```

Gaussian sphere wave over gaussian elliptic hill. $\Delta t = 0.005$, $\Delta x = \Delta y = 0.01$:

In [22]:
```
youTubeVideo("SWDLRVmWn1U")
```

```
<IPython.lib.display.YouTubeVideo object at 0x55dc5d0>
```

Can't see the video? Click here: http://youtube.com/watch?v=SWDLRVmWn1U

Wall wave over gaussian spherical hill. $\Delta t = 0.005$, $\Delta x = \Delta y = 0.01$:

In [23]:
```
youTubeVideo("G_Pmdh-uptU")
```

```
<IPython.lib.display.YouTubeVideo object at 0x55dc5d0>
```

Can't see the video? Click here: http://youtube.com/watch?v=G_Pmdh-uptU

Wall wave over gaussian spherical cliff. $\Delta t = 0.005$, $\Delta x = \Delta y = 0.01$:

In [24]:
```
youTubeVideo("IlQwI2azEfU")
```

```
<IPython.lib.display.YouTubeVideo object at 0x55dc5d0>
```

Can't see the video? Click here: http://youtube.com/watch?v=IlQwI2azEfU

Wall wave over double slit wall. $\Delta t = 0.005$, $\Delta x = \Delta y = 0.01$.

In [25]:
```
youTubeVideo("fnTyEUTE2XE")
```

```
<IPython.lib.display.YouTubeVideo object at 0x55dc5d0>
```

Can't see the video? Click here: http://youtube.com/watch?v=fnTyEUTE2XE