Finite difference simulation of 2D waves

In this project we are going to address the two-dimensional, standard, linear wave equation with damping,

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

in a rectangular spatial domain $\Omega = [0, L_x] \times [0, L_y]$. The boundary condition is the homogenous Neumann condition,

$$\frac{\partial u}{\partial n} = \hat{n} \cdot \nabla u = 0,$$

where $\hat{n}$ is the normal vector at the boundary. The initial conditions are

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

where $u_t = \frac{\partial u}{\partial t}$.

Python scripts

All of the codes used in this assignment can be found here, along with the gifs generated in part 4 of the project.

Discretization

To solve the above wave equation the different terms have to approximated and discretized. We'll approximate the derivatives, and hence derive their discretization, through Taylor expansion. The following notation for the functions like $u(x,y,t)$ will be used,

$$u(x_i, y_j, t_n) \equiv u^n_{i,j} \quad \text{where } x_i = i\Delta x,\ y_j = j\Delta y,\ t_n = n\Delta t.$$

where $i = 0, \dots, N_x + 1$, $j = 0, \dots, N_y + 1$ and $n = 0, \dots, N_t$. The lenght of the three arrays storing the different variable values are of different length, this is due to the fact that we will be using ghost cells to store values that lies outside our spatial domain at the boundaries. Meaning that in our calculation $\Omega = [-\Delta x, L_x + \Delta x] \times [-\Delta y, L_y + \Delta y]$. The points $i, j = 0$, $i = N_x + 1$ and $j = N_y + 1$ are the ghost points.

The first order partial derivative with respect to one of the variables will be denoted as $u_k = \frac{\partial u}{\partial k}$, while the second order partial derivative is $u_{kk} = \frac{\partial^2 u}{\partial k^2}$, where $k = \{x, y, t\}$.

Second order partial time derivative

The term u_{tt} is obtained through Taylor expansion both forward and backwards in time around the point (x, y, t).

$$\begin{aligned} u^{n+1}_{i,j} &= u^n_{i,j} + u_t\Delta t + u_{tt}\frac{\Delta t^2}{2} + u_{ttt}\frac{\Delta t^3}{6} + \mathcal{O}(\Delta t^4) \\ u^{n-1}_{i,j} &= u^n_{i,j} - u_t\Delta t + u_{tt}\frac{\Delta t^2}{2} - u_{ttt}\frac{\Delta t^3}{6} + \mathcal{O}(\Delta t^4) \end{aligned}$$

By adding the two equations, and divide both sides with Δt^2 we obtain the following approximation for the second order partial derivative with respect to time:

$$u_{tt} = \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2} + \mathcal{O}(\Delta t^2) \approx \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2}.$$

First order partial time derivative

The approximation to the first order partial derivative with respect to time can be obtained by truncating the equations for $u^{n\pm1}_{i,j}$ at the second term, and subtracting instead of adding them togehter. We then obtain the following

$$b \cdot u_t = b\left[\frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2\Delta t} + \mathcal{O}(\Delta t)\right] \approx b\left[\frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2\Delta t}\right].$$

The variable coefficients

The discretization and approxiamtion of the terms

$$\frac{\partial}{\partial x}(q(x,y)u_x) \quad \text{and} \quad \frac{\partial}{\partial y}\left(q(x,y)u_y\right),$$

are found with the exact same routine but with respect to different variables, so the derivation will only be done for the variable x.

The first step is to discretize the outer derivative. We start by defining

$$\phi(x,y) = q(x,y)u_x.$$

Again we Taylor expand forward and backward in space with respect to x around the point $(x \pm \Delta x/2, y)$ using a stepsize of $\Delta x/2$.

$$\begin{aligned} \phi_{i+\frac{1}{2},j} &= \phi_{i,j} + \frac{\Delta x}{2}\phi_x + \mathcal{O}(\Delta x^2), \\ \phi_{i-\frac{1}{2},j} &= \phi_{i,j} - \frac{\Delta x}{2}\phi_x + \mathcal{O}(\Delta x^2), \end{aligned}$$

Subtracting the two equations yield

$$\phi_x = \frac{\phi_{i+\frac{1}{2},j} - \phi_{i-1\frac{1}{2},j}}{\Delta x}.$$

The next step is to discretize $\phi_{i\pm\frac{1}{2},j}$. Again, the procedure for deriving the two expressions are almost identical, so this will only be done for $\phi_{i+\frac{1}{2},j}$.

From the definition above,

$$\phi_{i+\frac{1}{2},j} = q(x')\frac{\partial u(x', y, t)}{\partial x}, \quad \text{where } x' = x + \frac{\Delta x}{2}.$$

Again we will Taylor expand forward and backwards around the point $(x \pm \Delta x/2, y, t)$ using a stepsize of $\Delta x/2$, for obtaining an approximation for $u_x(x', y, t)$.

$$\begin{aligned} u^n_{i'+\frac{1}{2},j} &= u^n_{i',j} + \frac{\Delta x}{2}u_x(x', y, t) + \mathcal{O}(\Delta x^2), \\ u^n_{i'-\frac{1}{2},j} &= u^n_{i',j} - \frac{\Delta x}{2}u_x(x', y, t) + \mathcal{O}(\Delta x^2). \end{aligned}$$

In the two equations $i' = (x + \frac{\Delta x}{2})$. Subtracting the two yields

$$u(x', y, t)_x = \frac{u^n_{i+1,j} - u^n_{i,j}}{\Delta x} + \mathcal{O}(\Delta x) \approx \frac{u^n_{i+1,j} - u^n_{i,j}}{\Delta x}.$$

The final result for $\frac{\partial}{\partial x}(q(x,y)u_x)$, and hence also $\frac{\partial}{\partial y}\left(q(x,y)u_y\right)$ then reads:

$$\begin{aligned} \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) &= \frac{\phi_{i+\frac{1}{2},j} - \phi_{i-1\frac{1}{2},j}}{\Delta x} = \frac{1}{\Delta x^2}\left[q_{i+\frac{1}{2},j}\left(u^n_{i+1,j} - u^n_{i,j}\right) - q_{i-\frac{1}{2},j}\left(u^n_{i,j} - u^n_{i-1,j}\right)\right], \\ \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) &= \frac{\phi_{i,j+\frac{1}{2}} - \phi_{i,j-\frac{1}{2}}}{\Delta y} = \frac{1}{\Delta y^2}\left[q_{i,j+\frac{1}{2}}\left(u^n_{i,j+1} - u^n_{i,j}\right) - q_{i,j-\frac{1}{2}}\left(u^n_{i,j} - u^n_{i,j-1}\right)\right]. \end{aligned}$$

We further assume that $q(x,y)$ is a discrete function, so we approximate $q_{k\pm\frac{1}{2}}$ to be

$$q_{k\pm\frac{1}{2}} = \frac{1}{2}(q_{k\pm1} + q_k).$$

The final expressions for the variable coefficients are thus

$$\begin{aligned} \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) &= \frac{1}{\Delta x^2}\left[\frac{1}{2}\left(q_{i+1,j} + q_{i,j}\right)\left(u^n_{i+1,j} - u^n_{i,j}\right) - \frac{1}{2}\left(q_{i,j} + q_{i-1,j}\right)\left(u^n_{i,j} - u^n_{i-1,j}\right)\right], \\ \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) &= \frac{1}{\Delta y^2}\left[\frac{1}{2}\left(q_{i,j+1} + q_{i,j}\right)\left(u^n_{i,j+1} - u^n_{i,j}\right) - \frac{1}{2}\left(q_{i,j} + q_{i,j-1}\right)\left(u^n_{i,j} - u^n_{i,j-1}\right)\right]. \end{aligned}$$

The driving force

The last term to discretize is the driving force. We simply denote its contribution as

$$f(x,y,t) = f^n_{i,j}.$$

General scheme for $u^{n+1}_{i,j}$

Now that all the terms in the wave equation are approximated and discretized, we can manipulate the expressions to obtain the general scheme for computing $u^{n+1}_{i,j}$ at the interior spatial mesh points.

We define $A \equiv 1/(2 + b\Delta t)$, so to evolve the solution $u(x,y,t)$ one step in the time domain we compute

$$\begin{aligned} u^{n+1}_{i,j} = {} & A(b\Delta t - 2)u^{n-1}_{i,j} + 4Au^n_{i,j} + \left(\frac{\Delta t}{\Delta x}\right)^2 A\left[\left(q_{i+1,j} + q_{i,j}\right)\left(u^n_{i+1,j} - u^n_{i,j}\right) - \left(q_{i,j} + q_{i-1,j}\right)\left(u^n_{i,j} - u^n_{i-1,j}\right)\right] \\ & + \left(\frac{\Delta t}{\Delta y}\right)^2 A\left[\left(q_{i,j+1} + q_{i,j}\right)\left(u^n_{i,j+1} - u^n_{i,j}\right) - \left(q_{i,j} + q_{i,j-1}\right)\left(u^n_{i,j} - u^n_{i,j-1}\right)\right] + 2A\Delta t^2 f^n_{i,j}. \end{aligned}$$

Modified scheme for the first step

When $n = 0$ we will run into a problem regarding the term $u^{-1}_{i,j}$. To find a solution to this problem we look at the intial condtion $u_t = V(x,y)$. The discretized version of the condtion reads

$$u(x,y,0)_t = \frac{u^1_{i,j} - u^{-1}_{i,j}}{2\Delta t} = V_{i,j} \quad \rightarrow \quad u^{-1}_{i,j} = u^1_{i,j} - 2\Delta t V_{i,j}.$$

Inserting this into the general scheme with $n = 0$, we obtain the following first step,

$$\begin{aligned} u^1_{i,j} = \frac{1}{4}\Bigg[& 2(2 - b\Delta t)\Delta t V_{i,j} + 4u^0_{i,j} + \left(\frac{\Delta t}{\Delta x}\right)^2\left[\left(q_{i+1,j} + q_{i,j}\right)\left(u^0_{i+1,j} - u^0_{i,j}\right) - \left(q_{i,j} + q_{i-1,j}\right)\left(u^0_{i,j} - u^0_{i-1,j}\right)\right] \\ & + \left(\frac{\Delta t}{\Delta y}\right)^2\left[\left(q_{i,j+1} + q_{i,j}\right)\left(u^0_{i,j+1} - u^0_{i,j}\right) - \left(q_{i,j} + q_{i,j-1}\right)\left(u^0_{i,j} - u^0_{i,j-1}\right)\right] + 2\Delta t^2 f^0_{i,j}\Bigg]. \end{aligned}$$

Modified scheme at boundary points

By using ghost cells the indices that corresponds with the boundary points are $i, j = 1$, $i = N_x$ and $j = N_y$. This means that our interior mesh points are $i = \{2, \dots, N_x - 1\}$ and $j = \{2, \dots, N_y - 1\}$.

The Neumann boundary condtion can be discretized with a centered derivative, invoking the condtion at the boundaries $x = 0, L_x$ and $y = 0, L_y$ generates four important relations for our ghost points.

1. $x = 0$: $i = 1$ and $\hat{n} = -\hat{\imath}$,

$$-\hat{\imath} \cdot \nabla u = -u_x(0, y, t) = -\left(\frac{u^n_{2,j} - u^n_{0,j}}{2\Delta x}\right) = 0$$

$$\rightarrow \quad u^n_{0,j} = u^n_{2,j}$$

1. $x = L_x$: $i = N_x$ and $\hat{n} = \hat{\imath}$

$$\hat{\imath} \cdot \nabla u = u_x(L_x, y, t) = \left(\frac{u^n_{N_x+1,j} - u^n_{N_x-1,j}}{2\Delta x}\right) = 0$$

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

1. $y = 0$: $j = 1$ and $\hat{n} = -\hat{\jmath}$,

$$-\hat{\jmath} \cdot \nabla u = -u_y(x, 0, t) = -\left(\frac{u^n_{i,2} - u^n_{i,0}}{2\Delta y}\right) = 0$$

$$\rightarrow \quad u^n_{i,0} = u^n_{i,2}$$

1. $x = L_y$: $j = N_j$ and $\hat{n} = \hat{\jmath}$

$$\hat{\jmath} \cdot \nabla u = u_y(x, L_y, t) = \left(\frac{u^n_{i,N_y+1} - u^n_{i,N_y-1}}{2\Delta y}\right) = 0$$

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

The ghost values takes the same values as the first interior mesh points. Meaning that in our calculations we can iterate over the index sets $i = \{1, \dots, N_x\}$ and $j = \{1, \dots, N_y\}$.

From the general scheme for $u^{n+1}_{i,j}$, it is evident that the terms containing $q_{k\pm1}$ can cause problems at the boundaries. To remedy this we again do a Taylor expansion both forward and backwards in space. The routine is identical for both spatial directions, so this will only be done for x.

$$\begin{aligned} q_{i+1,j} &= q_{i,} + \Delta x q_x + \mathcal{O}(\Delta x^2) \\ q_{i-1,j} &= q_{i,j} - \Delta x q_x + \mathcal{O}(\Delta x^2) \end{aligned}$$

Adding the two equations,

$$q_{i+1,j} + q_{i-1,j} = 2q_{i,j} + \mathcal{O}(\Delta x^2).$$

From the above equation we can approximate the ghost values for the wave velocity as

$$\begin{aligned} q_{0,j} &\approx 2q_{1,j} - q_{2,j} & \qquad q_{N_x+1,j} &\approx 2q_{N_x,j} - q_{N_x-1,j} \\ q_{i,0} &\approx 2q_{i,1} - q_{i,2} & \qquad q_{i,N_y+1} &\approx 2q_{i,N_y} - q_{i,N_y-1} \end{aligned}$$

Verification with constant solution

To verify our implementation (see WaveSolver.py) we construct a constant solution,

$$u(x,y,t) = U.$$

Since the solution is constant through time the initial condtion $u(x,y,0)$ has to be equal to the constant U, given the constant solution there cannot exists damping, i. e. $b = 0$. The initial conditions is thus reduced to

$$\begin{aligned} u(x,y,0) &= I(x,y) = U, \\ u_t(x,y,0) &= V(x,y) = 0. \end{aligned}$$

All the derivatives in our original PDE will be equal to zero for a constant solution, the same is true for the driving force $f(x,y,t)$.

Inserting what we now know into the general scheme for $u^{n+1}_{i,j}$ yields the following relation,

$$\begin{aligned} U = -U + 2U &+ \left(\frac{\Delta t}{\Delta x}\right)^2\frac{1}{2}\left[(1+1)(U-U) - (1+1)(U-U)\right] \\ &+ \left(\frac{\Delta t}{\Delta y}\right)^2\frac{1}{2}\left[(1+1)(U-U) - (1+1)(U-U)\right], \\ U &= U. \end{aligned}$$

As we can see, the constant solution solves the original PDE.

Performing the calculations with these parameters should produce a solution that is equal to our choice of U in all the spatial mesh points. Our test function will verify that all the spatial elements in the final time step $u^{N_t}_{i,j}$ are in fact identical and equal to I.

```
In [2]: import numpy as np
        from WaveSolver2D import WaveSolver2D


        b = 0
        Nx = 10
        Ny = 10
        Lx = 1
        Ly = 1
        T = 1

        def I(x,y):
            return 1

        def V(x,y):
            return 0

        def q(x,y):
            return 1

        def f(x,y,t):
            return 0

        solver = WaveSolver2D(b, Nx, Ny, Lx, Ly, T)
        solver.Initialize(I, V, q, f)
        solver.FirstTimeStep()
        solver.AdvanceTime()

        def test_constant_solution(u, I):
            u_vec = u[1:-1,1:-1]
            constant = True
            for i in range(len(u_vec)):
                if np.any(u_vec != I):
                    print("Solution vector contains elements that are not equal to the inital condition")
                    constant = False
                    break
            return print(constant)

        test_constant_solution(solver.u, 1)
```

```
True
```

Standing, undamped wave

I have defined Δt from Δx and Δy, where $\Delta x = \Delta y$. The expression for Δt is

$$\Delta t = \frac{\beta}{\sqrt{\max(q(x,y))(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2})}},$$

where β is a factor to ensure that

$$\Delta t \leq \frac{1}{\sqrt{(\max(q(x,y))\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2})}}.$$

From the above expression it is evident that $\Delta t \propto \Delta x$, the same is true for Δy. So in our implementation Δx will play the role as the discretized parameter h.

Given the analytical solution

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

our initial condtions becomes

$$\begin{aligned} u(x,y,0) &= I(x,y) = A\cos(k_x x)\cos(k_y y), \\ u_t(x,y,0) &= V(x,y) = 0. \end{aligned}$$

To determine ω we take a look at the PDE for a standing, undamped wave

$$\frac{\partial^2 u}{\partial t^2} = q\nabla^2 u.$$

Computing the different terms with $u = u_e$ and inserting them back into the PDE yields

$$\begin{aligned} \frac{\partial^2 u_e}{\partial t^2} &= -\omega^2 u_e, \\ \frac{\partial^2 u_e}{\partial x^2} &= -k_x^2 u_e, \\ \frac{\partial^2 u_e}{\partial y^2} &= -k_y^2 u_e, \\ \Rightarrow \omega^2 &= (k_x^2 + k_y^2)q. \end{aligned}$$

So in our calculations this will be our choice for ω. All of the other constants are put equal to 1.

The error norm computed is the L^∞-norm for the last time step, the error E is thus

$$E = \|e^{N_t}_{i,j}\|_{L^\infty} = \|u^{N_t}_{i,j} - u_e(x_i, y_j, T)\|_{L^\infty} = \max_i \max_j |u^{N_t}_{i,j} - u_e(x_i, y_j, T)|.$$

To determine the convergence rate we run the calculations for stepsizes $h = 1/2^i$ for $i = 1, \dots, 8$, and compute

$$r = \frac{\log(E_{j+1}/E_j)}{\log(h_{j+1}/h_j)},$$

here the j's describes consecutive experiments.

```
In [7]: import numpy as np
        from WaveSolver2D import WaveSolver2D

        b = 0
        Lx = 1
        Ly = 1
        T = 1
        mx = 1; my = 1;
        kx = mx*np.pi/Lx
        ky = my*np.pi/Ly

        def I(x,y):
            A = 1
            kx = np.pi
            ky = np.pi
            return A*np.cos(kx*x)*np.cos(ky*y)

        def V(x,y):
            return 0

        def q(x,y):
            return 1

        def f(x,y,t):
            return 0

        def analytical(x, y, t):
            A = 1
            kx = np.pi
            ky = np.pi
            w = np.sqrt(2*np.pi**2)
            return A*np.cos(kx*x)*np.cos(ky*y)*np.cos(t*w)

        N = [2**i for i in range(1,8)]
        h = [1/(i-1) for i in N]
        error = []

        for i in range(len(N)):
            solver = WaveSolver2D(b, N[i], N[i], Lx, Ly, T)
            solver.Initialize(I, V, q, f)
            solver.FirstTimeStep()
            solver.AdvanceTime()

            error.append(solver.ComputeError(analytical))

        for i in range(len(N)-1):
            r = np.log10(error[i+1]/error[i])/np.log10(h[i+1]/h[i])
            print(r)
```

```
2.8642942003317198
2.200213492936624
1.9669160761073963
2.0367989520074636
2.0186919282906763
1.9934508412750134
```

Waves with damping and variable wave velocity

We choose the following non-constant $q(x,y) = y$, thus making the term $dq/dx = 0$, along with the following constants

$$A = 1, \quad B = 0, \quad d = 1, \quad b = 2d = 2, \quad \omega = d = 1.$$

This yields the source term

$$f(x,y,t) = \cos(k_x x)\cos(k_y y)\cos(t)e^{-t}\left(yk_x^2 + yk_y^2 - 2 + k_y\tan(k_y y)\right).$$

The analytical solution then reads

$$u_e = \cos(k_x x)\cos(k_y y)\cos(t)e^{-t},$$

with initial condtions

$$\begin{aligned} I(x,y,0) &= \cos(k_x x)\cos(k_y y), \\ V(x,y,0) &= -\cos(k_x x)\cos(k_y y). \end{aligned}$$

Below follows simulation of the new wave equation with damping and variable wave velocity, where the convergence rate r is the ouput.

```
In [1]: import numpy as np
        from WaveSolver2D import WaveSolver2D

        b = 2
        Lx = 1
        Ly = 1
        T = 1
        mx = 1; my = 1;
        kx = mx*np.pi/Lx
        ky = my*np.pi/Ly

        def I(x,y, A=1):
            kx = np.pi
            ky = np.pi
            return A*np.cos(kx*x)*np.cos(ky*y)

        def V(x,y, A=1, d=1):
            kx = np.pi
            ky = np.pi
            return A*np.cos(kx*x)*np.cos(ky*y)*(-d)

        def q(x,y):
            return y

        def f(x,y,t, A=1, d=1, w=1):
            kx = np.pi
            ky = np.pi
            return A*np.cos(kx*x)*np.cos(ky*y)*np.cos(w*t)*np.exp(-d*t)*(y*kx*kx + y*ky*ky - 2*d*d + ky*np.tan(
        ky*y))


        def analytical(x, y, t, A=1, d=1, w=1):
            kx = np.pi
            ky = np.pi
            return A*np.cos(kx*x)*np.cos(ky*y)*np.cos(t*w)*np.exp(-d*t)


        N = [2**i for i in range(1,7)]
        h = [1/(i-1) for i in N]
        error = []

        for i in range(len(N)):
            solver = WaveSolver2D(b, N[i], N[i], Lx, Ly, T)
            solver.Initialize(I, V, q, f)
            solver.FirstTimeStep()
            solver.AdvanceTime()

            error.append(solver.ComputeError(analytical))


        for i in range(len(N)-1):
            r = np.log10(error[i+1]/error[i])/np.log10(h[i+1]/h[i])
            print(r)
```

```
2.533705795064227
1.4087291014090986
1.967827379300808
1.9802748322436066
1.986922922018016
```

Investigate a physical problem

Again, visit this page to see the output gifs.

The following code was run to generate the gif for the Gaussian Hill surface.

```
In [4]: import numpy as np
        from WaveSolver2D import WaveSolver2D
        import os

        b = 0
        Nx = 50
        Ny = 50
        Lx = 1
        Ly = 1
        T = 10.


        def I(x,y, I0=1, Ia=1, Im=0.5, Is=0.1):
            return I0 + Ia*np.exp(-((x-Im)/Is)**2)

        def V(x,y):
            return 0

        def q(x,y, H0 = 10):
            g = 9.81
            return g*(H0 - B(x,y))

        def B(x, y, B0=0, Ba=1, Bmx=0.5, Bmy=0.5, Bs=0.1, b=1):
            return B0 + Ba*np.exp(-((x-Bmx)/Bs)**2-((y-Bmy)/(b*Bs))**2)

        def f(x,y,t):
            return 0

        solver = WaveSolver2D(b, Nx, Ny, Lx, Ly, T)
        solver.Initialize(I, V, q, f)
        solver.FirstTimeStep()
        solver.AdvanceTime()

        #os.system("python3 animate.py")
```

For the cosine hat function the only thing changed from the code segment above is the B function.

```
In [5]: def B(x, y, B0=0, Ba=1, Bmx=0.5, Bmy=0.5, Bs=2, b=1):
            return B0 + Ba*np.cos(np.pi*(x-Bmx/(2*Bs)))*np.cos(np.pi*(y-Bmy/(2*Bs)))
```