Abstract

The two dimensional wave equation can be used to model many interesting problems such as electromagnetic waves and tsunami waves. We develop a numerical solver based on an explicit scheme and discuss algorithm validation, stability, visualization and techniques for complex geometries. MayaVi is used to visualize and we have written our own visualizer using OpenGL.

Theory

The wave equation is quite powerful in the sense that it describes many important systems in physics. We have the general equation with damping coefficient b, wave velocity $q(x, y)$ and source $f(x, y, t)$ given as

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

or more compact

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = \nabla \cdot \big(q(x,y)\nabla u\big) \tag{2}$$

The velocity function $q(x, y)$ describes the wave propagation speed in the point (x, y). For water waves, this can be interpreted as the depth h at the point (x, y) as water wave velocities behave like $c(h) \propto \sqrt{h}$.

Boundary and initial conditions

The equation is second order in time and space, so in order to have a unique solution, we need the initial condition $u(x, y, t = 0) = I(x, y)$ and its derivative $\frac{\partial u}{\partial t}|_{t=0} = V(x, y)$. In addition, when the wave hits a wall, we use the boundary condition

$$\nabla u \cdot \mathbf{n} = 0 = \frac{\partial u}{\partial n} \tag{3}$$

where $\frac{\partial u}{\partial n}$ is the derivative in the direction normal to the boundary.

Standing wave

By assuming a constant wave velocity $q(x, y) = k$, a solution to the equation is the standing wave

$$\begin{aligned} u(x,y,t) &= \exp(-bt)\cos\left(\frac{m_x x\pi}{L_x}\right)\cos\left(\frac{m_y y\pi}{L_y}\right)\cos\omega t \\ &= \exp(-bt)\cos(k_x x)\cos(k_y y)\cos\omega t \end{aligned}$$

for arbitrary integers m_x, m_y and frequency ω. This solution gives a rather messy source term $f(x, y, t)$, but it's trivial to calculate. We rewrite the solution to $u(x, y, t) = T(t) \cdot R(x, y) = T(t) \cdot X(x) \cdot Y(y)$

$$\begin{aligned} b\frac{\mathrm{d}T}{\mathrm{d}t} &= -e^{-bt}(b^2\cos\omega t + b\omega\sin\omega t) \\ \frac{\mathrm{d}^2 T}{\mathrm{d}t^2} &= e^{-bt}\Big[(b^2-\omega^2)\cos\omega t + 2b\omega\sin\omega t\Big] \\ \frac{\mathrm{d}^2 X}{\mathrm{d}x^2} &= -k_x^2 X(x) \end{aligned}$$

The left hand side of the wave equation now reads

$$\begin{aligned} e^{-bt}R(x,y)\Big[(b^2-\omega^2)\cos\omega t + 2b\omega\sin\omega t - b^2\cos\omega t - b\omega\sin\omega t\Big] \\ = e^{-bt}R(x,y)\Big[-\omega^2\cos\omega t + b\omega\sin\omega t\Big] \\ = u(x,y,t)\Big[-\omega^2 + b\omega\tan\omega t\Big] \end{aligned}$$

Which gives

$$u(x,y,t)\Big[-\omega^2 + b\omega\tan\omega t\Big] = -q\big[k_x^2+k_y^2\big]u(x,y,t) + f(x,y,t)$$

We can solve this for the source term

$$f(x,y,t) = u(x,y,t)\Big[k^2 - \omega^2 + b\omega\tan\omega t\Big]$$

where $k^2 = q(k_x^2+k_y^2)$. If we choose this as the source term, we only need to calculate the first time derivative to have a unique solution.

$$\begin{aligned} V(x,y) &= \frac{\partial u(x,y,t)}{\partial t}\Big|_{t=0} \\ &= -e^{-bt}(b\cos\omega t + \omega\sin\omega t)\cos(k_x x)\cos(k_y y)\Big|_{t=0} \\ &= -b\cos(k_x x)\cos(k_y y) \end{aligned}$$

The initial condition is

$$I(x,y) = u(x,y,t=0) = \cos(k_x x)\cos(k_y y)$$

We will use this solution in one of the verifications later.

Numerical implementation

Discretization

If we look at a system of widths L_x and L_y, we can divide the system into a grid with $N_x \times N_y$ points where we have the steplengths $\Delta x = \frac{L_x}{N_x-1}$ and $\Delta y = \frac{L_y}{N_y-1}$. The wave function u is now represented as a matrix $u_{i,j}$ where $u_{i,j} = u(x_i, y_j)$ in each timestep t_n. The numerical wave equation can be discretized on operator form

$$\Big[D_t D_t u + bD_{2t}u = D_x q[D_x u] + D_y q[D_y u]\Big]_{i,j}^n$$

where we have applied the standard approximation for second derivatives in time, the centered difference scheme on the first derivative and the Crank-Nicolson twice in the spatial part. Written out in full index form, this evaluates to

$$\begin{aligned} \text{L.H.S.} &= \frac{u_{i,j}^{n+1} + u_{i,j}^{n-1} - 2u_{i,j}^n}{\Delta t^2} + b\frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} \\ \text{R.H.S.} &= \frac{1}{\Delta x}\left(q_{i+\frac{1}{2},j}\left[\frac{u_{i+1,j}^n - u_{i,j}^n}{\Delta x}\right] - q_{i-\frac{1}{2},j}\left[\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x}\right]\right) \\ &\quad + \frac{1}{\Delta y}\left(q_{i,j+\frac{1}{2}}\left[\frac{u_{i,j+1}^n - u_{i,j}^n}{\Delta y}\right] - q_{i,j-\frac{1}{2}}\left[\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y}\right]\right) \end{aligned}$$

This can be solved for $u_{i,j}^{n+1}$ and we can easily evolve the system in time. We notice that to calculate timestep $n+1$, we need both timesteps n and $n-1$ because of the second derivative in time. This reflects the requirement of both $u(x,y,t=0) = I(x,y)$ and $u(x,y,t+\Delta t)$ as mentioned above. There are two standard ways to calculate the second timestep $u(x,y,0+\Delta t)$. If we have the initial condition $\frac{\partial u}{\partial t}|_{t=0} = 0$, we apply the forward difference scheme

$$\frac{u_{i,j}^{+1} - u_{i,j}^{0}}{\Delta t} = 0$$

which gives

$$u_{i,j}^{+1} = u_{i,j}^{0}$$

for all i, j. If we have the more complicated and general system where $\frac{\partial u}{\partial t}|_{t=0} = V(x,y)$, we can use the centered difference scheme and define

$$\frac{u_{i,j}^{+1} - u_{i,j}^{-1}}{2\Delta t} = V(x_i, y_j)$$

which gives

$$u_{i,j}^{-1} = u_{i,j}^{+1} + V_{i,j}$$

where we evaluate $V_{i,j} = V(x_i, y_j)$. We can again solve the equation for $u_{i,j}^{+1}$ by using only one timestep. The second derivative in the spatial dimensions also require both neighbor points which will give problems at the boundaries where such points might not exist. We solve this by using (3) and apply the centered difference scheme on $\frac{\partial u}{\partial n}$, assuming that the normal vector is in either the x- or y-direction and define

$$\frac{\partial u}{\partial x} = 0 = \frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x}$$

which again gives

$$u_{i-1,j}^{n} = u_{i+1,j}^{n} \tag{4}$$

These points are often called ghost points, or ghost cells, because they don't really exist.

More general boundary conditions

If we have walls at the boundaries $u_{i,j}|i,j \in \{0, N_x - 1\}$, we can easly do an if-test and apply (4). The code will then be rather messy with special cases for each boundary, and the systems geometry is limited. We can instead introduce the wall matrix $W \in \mathbb{B}^{N_x \times N_y}$ where $\mathbb{B}$ is the boolean domain. Each point $W_{i,j}$ contains either a *true* or *false* value defining if that point is a wall or not. To implement this method, we have the matrices $u\_$ and $u\_prev\_$ which contains all the values of u in the two previous timesteps. In addition we have the wall matrix W with booleans. When we access the elements $u_{i\pm1,j}$ or $u_{i,j\pm1}$, we create a function that returns the elements $u_{i\mp1,j}$ or $u_{i,j\mp1}$ if there is a wall at the respective point. The algorithm looks like

```
int idx(int i) {
    return (i+10*Nr)\%Nr;
}

double u(int i,int j, int di, int dj) {
    if(W(idx(i+di),idx(j+dj))) {
        return u_(idx(i-di),idx(j-dj));
    }

    return u_(idx(i+di),idx(j+dj));
}
```

Complex objects and varying $q(x, y)$

By using the method above we can represent more complex objects and geometries by filling the wall matrix W instead of making complicated special cases programmatically. One way to handle both boundary conditions and the wave velocity field $q(x, y)$ is to use BMP-images to represent the geometry of the system. If we create a black-and-white BMP-image of size $N_x \times N_y$ pixels, this can in our program be represented as a matrix $G \in \mathbb{R}^{N_x \times N_y}$ where each value $G_{i,j} \in (0, 1)$ containing the white level value in that point. This image can be used to define both boundaries (walls and objects) and ground (wave velocity) by choosing a threshold value p such that $W_{i,j} = G_{i,j} \geq p$. By shifting the values $G_{i,j} \to G_{i,j} - p$, the value in G can now be interpreted as the depth where values above zero will behave like hard boundaries and values below zero will affect the wave velocity.

```
void calculateWalls() {
    for(int i=0;i<Nx;i++) {
        for(int j=0;j<Ny;j++) {
            walls(i,j) = ground(i,j) >= 0;
        }
    }
}
```

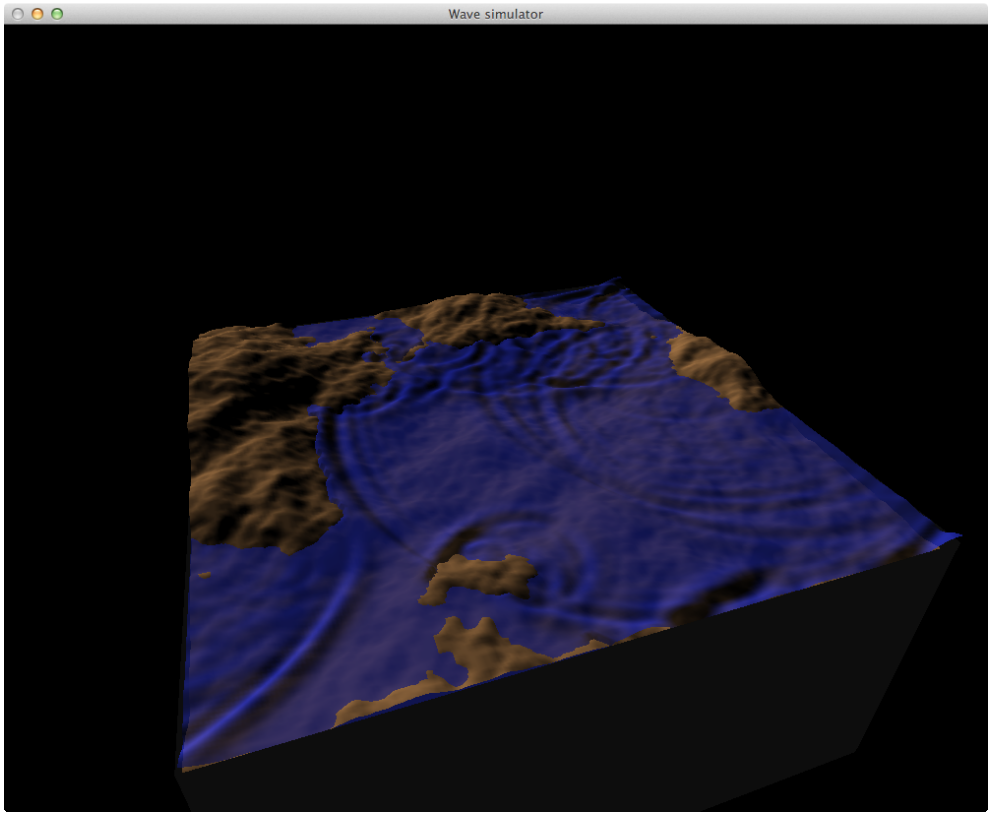

Figure 1: Waves that are reflected by the boundaries, e.g. the mountains.

Source term and rain

One application of source terms can be raindrops falling into the water. We can create raindrops randomly at a height z_0 and let them fall with constant velocity. Once a raindrop hits the surface, $z_{i,j} \leq u_{i,j}$, we increase the source term $f_{i,j}$ by a suitable value, we used 0.005.

```
void moveRainDrops() {
    for(int k=0;k<raindrops.size();k++) {
        raindrops[k].z -= 0.005;

        if(raindrops[k].z <= u_(raindrops[k].i,raindrops[k].j)) {
            source(raindrops[k].i,raindrops[k].j) = 0.01;
            raindrops.erase(raindrops.begin()+k--);
        }
    }
}
```

This creates those beautiful ripples we expect from raindrops hitting the surface. Usually there are a lot of raindrops, so in figure 2, we see that the water surface is quite noisy.

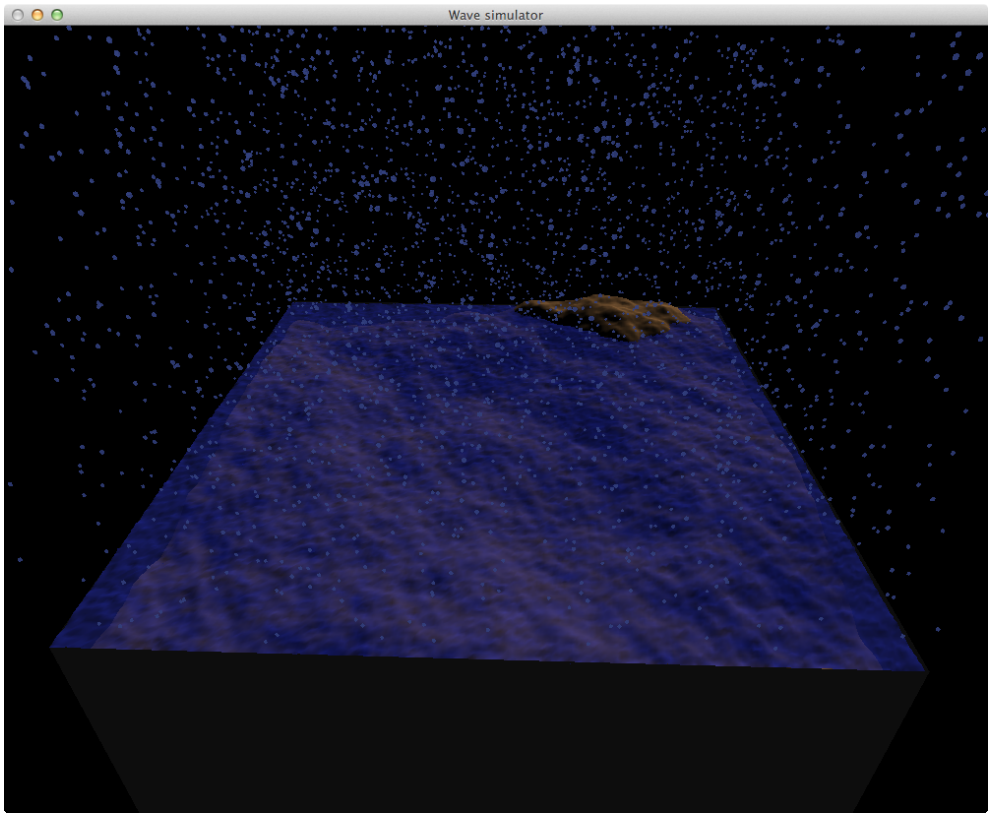

Figure 2: Raindrops creating a lot of ripples that add up to noisy surface.

Verification

In order to make sure the solver is free of bugs, we need to run it through some test solutions. Natural choices that should be reproduced exactly is the constant solution (for instance, $u(x, y, t) = 1.2$) and a plug wave. For python based programs, these tests can be put into a nosetest module, and done in `nose_test_betterwave.py` which can be found in the project folder. We there let each test run over all versions of the code (scalar, vectorized and (ideally) c++). For a constant solution we may create an initial state by

```
def I(x, y):
    return 1.2
```

and test the output of the solver, u, through for instance:

```
u_exact = 1.2*np.ones((Nx+1,Ny+1))
diff = np.abs(u_exact - u).max()
nt.assert_almost_equal(diff, 0, delta=1E-14)
```

For the plug wave, we can do a very similar test, as long as we make sure the wave travels for exactly, and integer number of periods, so that the wave in the exact same state as it was initially. In the nosetest module this was done by trial and error while using a relatively coarse grid. We made the initial state by the function

```
def I(x, y):
        return np.logical_and((np.abs(x) < Lx/2.+1), (np.abs(x) > Lx/2.-1))*1.
```

and test the output of the solver, u, through for instance:

```
x = np.linspace(0, Lx, Nx+1)  # mesh points in x dir
y = np.linspace(0, Ly, Ny+1)  # mesh points in y dir
X,Y = np.meshgrid(x,y)
u_exact = I(X,Y)
T = 1.0
n = int(T/dt - 5)
fname = 'solution_%06d.txt' % n
u = np.loadtxt(fname)
diff = np.abs(u_exact - u[:,0]).max()
nt.assert_almost_equal(diff, 0, delta=1E-14)
```

where the suitable value for n had to be found manually.

Once these tests are passed, we began looking at the convergence rate of the solvers. In order to do this, we note in theory, the error in the solution should to lowest order behave as

$$E = C_1(\Delta x)^2 + C_2(\Delta y)^2 + C_3(\Delta t)^2 \tag{5}$$

So that if we double the amount of mesh points in both time and space, we should reduce the error by a factor of 4. In order to measure the error, we decided to run the solver up to a given time (for simplicity, $T = 1.0$), and compare the solution to the exact solution some manufactured function in each point (x_i, y_j). If we denote this error by $e_{i,j}$, then a reasonable expression for the total error would be

$$E = \left(\frac{1}{N_x N_y} \sum_{i,j} e_{i,j}^2 \right)^{1/2}. \tag{6}$$

And so we check if this error is reduced by roughly a factor 4 if we double the number of grid points. Since this is only an estimate of the error behavior we will need to be a bit gentle on this requirement, and so we should check that the ratio between consecutive errors is, for instance, $1/4$ up to an accuracy of ± 0.05. We decided to test this on the standing wave solution

$$u(x, y, t) = e^{-bt} \cos\left(\frac{m_x \pi x}{L_x}\right) \cos\left(\frac{m_y \pi y}{L_y}\right) \cos(\omega t) \tag{7}$$

with source and initial state as described earlier. A test for convergence rate may look something like

```
mx = 4.; my = 4.; b = 0.01; omega=np.pi;
def u_exact(x,y,t):
    return ...
def f(x,y,t):
    return ...
def I(x, y):
    return ...
def V(x,y):
    return ...
Nx = 40; Ny = 40; T = 1.0
dt = 0.01
for version in ['vectorized']:
```

```
    errorlist=[0,0,0]
    for i in range(3):
        ... # construct the X-Y-grid, load the last step from file
        u_ex = u_exact(X,Y,T2)
        error = np.sqrt(np.sum((u_ex-u)**2)*dx*dy)
        errorlist[i] = error
        Nx = Nx*2;
        Ny = Ny*2;
        dt = dt/2
ratio1 = errorlist[1]/errorlist[0]
ratio2 = errorlist[2]/errorlist[1]
nt.assert_almost_equal(ratio1, 0.25, delta=0.05)
nt.assert_almost_equal(ratio2, 0.25, delta=0.05)
```

After some debugging, the program was able to successfully pass all tests for both the scalar and vectorized version.

c++ implementation

One of our extensions is to solve the given 2d wave problem by using C++ and its (almost) built-in libraries std and boost. This is as simple as coding the problem in python, as the syntax of the numerics is identical. One difference is that we have to use scalar programming as a vectorized form puts requirements on all called functions (e.g. double sin(double x) must be rewritten/overloaded to take a matrix as argument and also return a matrix). This is not a viable approach, but scalar programming done correctly should also be quite readable as the numerics become very close to the mathematics.

We wrote most of the code directly in main(). This could easily be converted into a general class solver, but this requires a lot of assignment code and was not done for simplicity. Also, most of the parameters are hardcoded and can be changed at the top of main(). The various input functions (i.e. I(x,y), V(x,y), q(x,y,t) and f(x,y,t)) can also be set at this point by assigning the general function pointers to point to the relevant functions.

We implemented the solver by using the boost::numeric::ublas::matrix template class for the matrices. This removes the need for explicit double pointer 'arithmetics' in addition to manual memeory allocation and deallocation. The syntax is easy

```
matrix<double> u(Nx, Ny); // create an Nx by Ny matrix
u(0,6) = 5; // set element (0,6) equal to 5.
```

The contrast from double pointers is obvious u(i,j) $\equiv$ u[i][j]. The matrix class also overloades the arithmetic operators (+,-,*,/), thus allowing vectorized syntax

```
matrix<double> u(Nx, Ny);
matrix<double> v(Nx, Ny);
...
u = u + v - 2*v;
```

This is not used in our code.

C++ parallelization

An additional extension is to parallelize our implementation. We did this by using OpenMP, which is an API that supports shared memory multiprocessing in C, C++ and Fortran.

Our implementation is to some extent easily parallelizable. From the numerics, we note that all calculations in a given timestep are idependent of each other. That is, for a given n, all calculations are only dependent on the previous timestep, n-1

$$u_n = u_n(i, j, u_{n-1}).$$

This implies that the nested spatial loop is embarrassingly parallel. In OpenMP, the parallelization is done by a pragma directive

```
// time loop
for (int n=1; n<Nt; n++) {
    // space loops
    #pragma omp parallel for
    for (int i=0; i<Nx; i++) // <-- this is parallelized
    for (int j=0; j<Ny; j++) // < -- all threads run this loop
        unext(i,j) = ... something only dep. on n-1 ...
}
```

This code forks the outer spatial loop into N_{cores} threads, where N_{cores} is the number of CPU cores. The threads eavenly (if possible) divide the i interval [0,Nx) into N_{cores} intervals and runs the i-loop for the given values of i. At the end of the i-loop, the slave threads terminate and only the master thread continues execution. Since all threads share the same memory, the variables (declared before the loop) used by all threads are the same. We could try to parallelize the innermost loop (i.e. the j-loop) instead, but this would probably decrease the efficiency since the threads would then be created/killed for each iteration of i (compared to each n), thus creating more overhead.

Speedtests

To check the efficiency of the parallelized code, we tried a few runs with and without parallelization. We use the following spatial domains: 50×50, 100×100, 400×400. The time used (in seconds) by each run is given in table 1. If the entire algorithm was completely parallelizable the ratio has an obvious theoretical

# cores	50x50	100x100	400x400
1	0.89	3.54	57.1
4	0.5	0.97	15.3
Ratio	1.8	3.65	3.73

Table 1: Execution time as a function of number of cores and spatial domain.

maximum ratio of N_{cores}. We can see that the parallel code is about 3.7 times faster, which is lower than the theoretical maximum. This must be so, since the algorithm has not been fully parallelized. Also, we see that the ratio falls with decreasing spatial domain size, which is probably due to increased overhead from thread creation/destruction compared to the gain from parallelization. Larger spatial domain should increase the ratio even further, as this overhead becomes more and more negligible.

Error convergance

We are given an exact solution to the 2d wave equation (1) in the project text. In this section, we study the numerical approximation of this solution. The derivations of $I(x,y)$, $V(x,y)$ and $f(x,y,t)$ are given in the section above.

There are various ways to calculate the error of the approximation. We wanted to check the dependence of the error in spatial resolution, Δx and Δt. We take $\Delta x = \Delta y \equiv h$ ($F_x = F_y = 1$) and $\Delta t = 0.1h$ ($F_t = 0.1$) to ensure stability. To calculate the error, we use equation 6 in the following form

$$E = h\sqrt{\sum_{i,j}(u_{i,j} - u_{i,j}^{\text{exact}})^2}. \tag{8}$$

We expect E/h^2 to be constant

$$\begin{aligned} E &= C_t\Delta t^2 + C_x\Delta x^2 + C_y\Delta_y^2 \\ &= C_t F_t^2 h^2 + C_x F_x^2 h^2 + C_y F_y^2 h^2 \\ &= (C_t F_t^2 + C_x F_x^2 + C_y F_y^2)h^2 \\ &\equiv Ch^2. \end{aligned}$$

Tests

The error can readily be calculated in the code, using 8, at a given (nonzero) time, e.g. $T = 1$. The results for various values of h are listed in table 2. We see that the factor E/h^2 is approximatly constant

h	E	E/h^2
0.08	0.0179	2.80
0.04	0.00437	2.73
0.02	0.00107	2.68
0.01	0.000266	2.66
0.005	6.62×10^{-5}	2.65
0.004	4.23×10^{-5}	2.65

Table 2: Error for given h at $T = 1$.

(≈ 2.7), as expect from error analysis. If we decrease h by a factor of 2 the error decreases by a factor of $2^2 = 4$. The nosetest for this is described above. The fact that the error behaves as expected is evidence that our implementation is correct.

Visualization

MayaVi

For a 3-dimensional animation, the standard matplotlib-library in Python shows a poor performance, with unacceptible speeds. In order to get sufficiently fast real-time animation, as well as a large improvement in speed for saving animations, we decided to animate our solutions with the MayaVi library. Using this we were able to plot solutions at a useful speed. Saving animations was done by saving the individual frames plotted by mayavi, and turning the set of pictures into an animation using the program Mencoder. Saving animations was still painfully slow, as MayaVi is not very efficient at saving plots. It turned out that, using built-in MayaVi function, one could store the picture as a set of pixel values. This set could then be plotted and saved in Pylab for an overall noticable speedup in movie creation. The details of this was stored in the function `mcrtmv(frames,dt,Lx,Ly,Nx,Ny,savemovie=False,mvname='test')` which can be found in the source code in the module of the same name. An screenshot of a plotted and stored solution can be found in figure 3 and several stored animations can be found among the source files.

OpenGL

OpenGL (Open Graphics Library, OGL) is an API that can be used to communicate with the graphics card. In a basic OGL-program, we define a window that has a width and height (measured in pixels). We are offered a set of objects or *primitives* (points, triangles, quads, polygons etc) that can be rendered in the window by OGL.

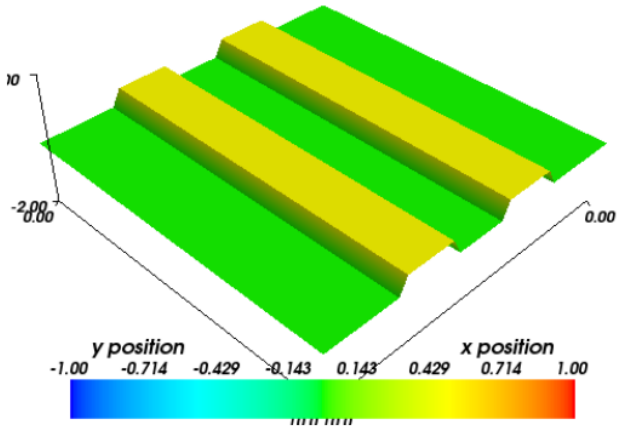

Figure 3: The figure shows a screenshot of a stored animation of the plug wave solution

Rendering points and triangles

We can render a set of points at positions $r_i = (x_i, y_i, z_i)$ saved in a matrix with the following code

```
glBegin(GL_VERTEX);
for (int i=0; i<10; i++) {
   glVertex(points[n].x,points[n].y,points[n].z);
}
glEnd();
```

Usually we want to render surfaces rather than points, so we use one of the other primitives such as *GL_TRIANGLES*. If we call *glBegin(GL_TRIANGLES);* instead, OGL expects three points defining a triangle, a *face*. We can then render our whole grid by dividing it into small triangles. Rendering code might then look like

```
glBegin(GL_TRIANGLE);
for (int i=0; i<n_faces; i++) {
   for(int j=0;j<3;j++) {
      glVertex(faces[i].points[j].x,faces[i].points[j].y,faces[i].points[j].z);
   }
}
glEnd();
```

The grid will be triangulated like we see in figure 4.

By increasing the grid size, the triangles get smaller and smaller and we won't even notice that there isn't a smooth surface.

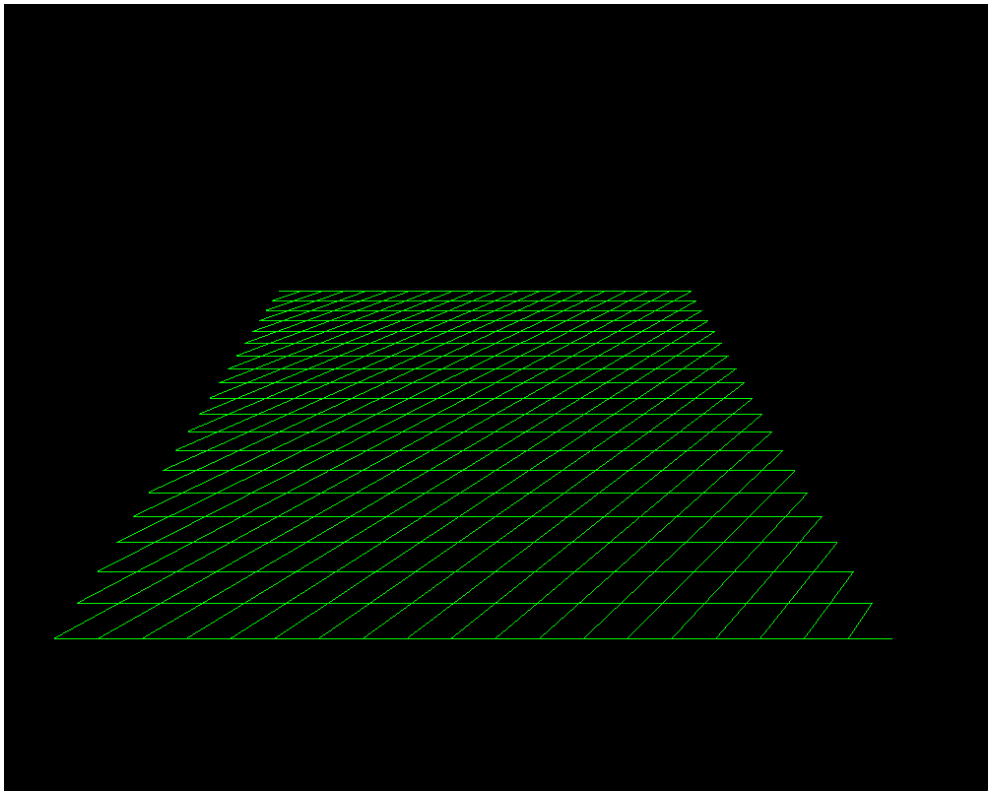

Figure 4: Triangulation of a 20x20 grid.

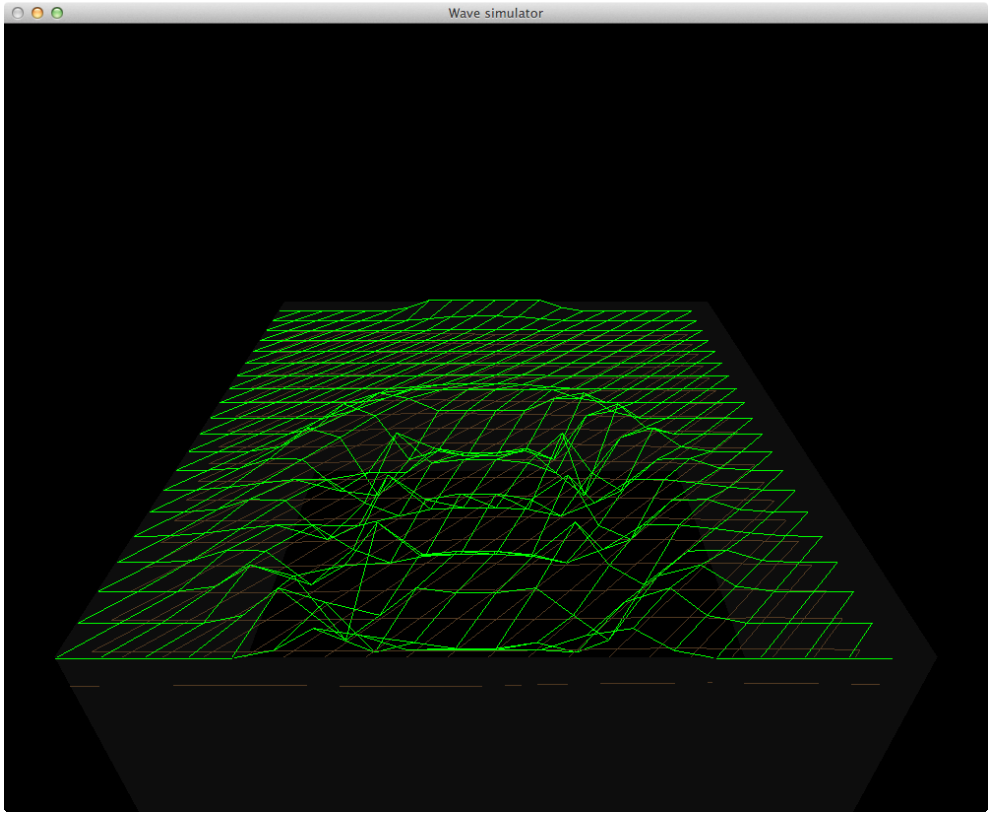

Figure 5: Triangulation of a 20x20 grid with a wave.

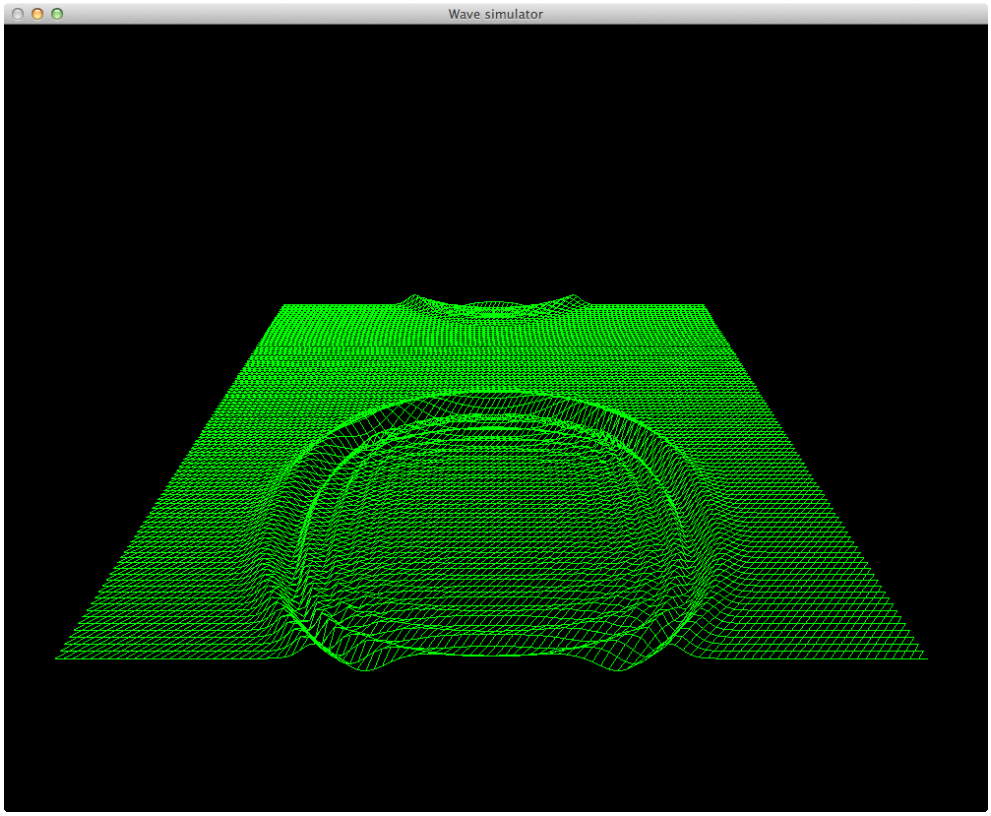

Figure 6: Triangulation of a 100x100 grid with the same wave as in figure 5.

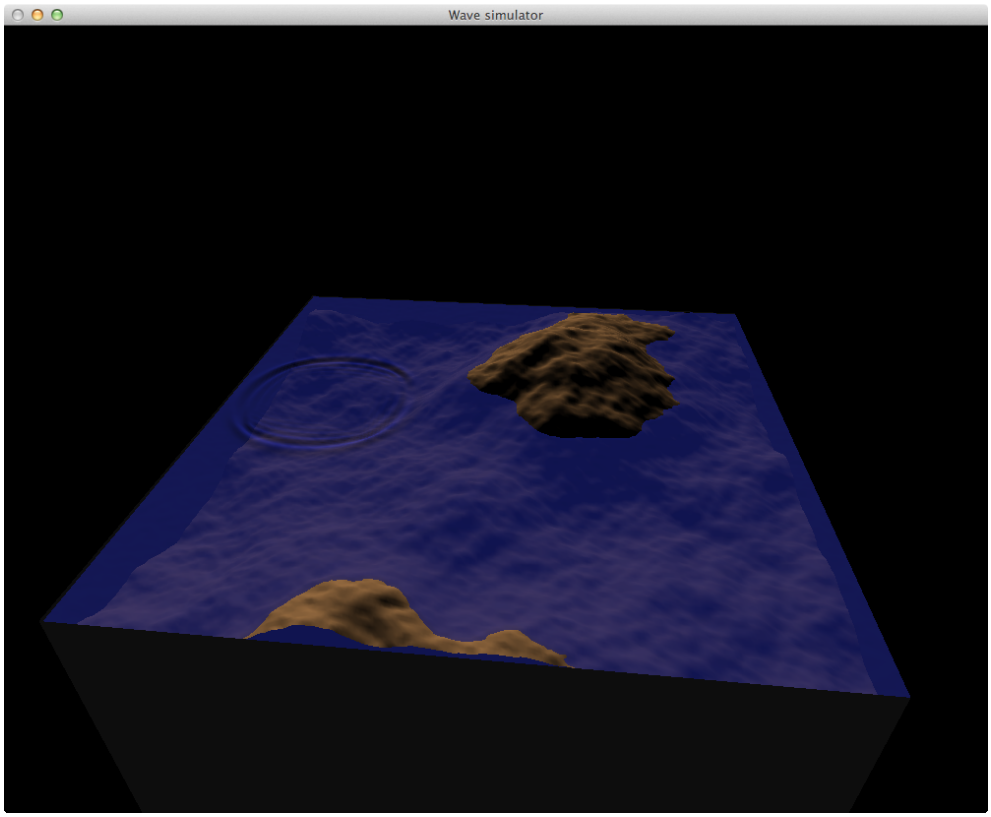

Figure 7: Beautiful wave in a realistic wave simulator

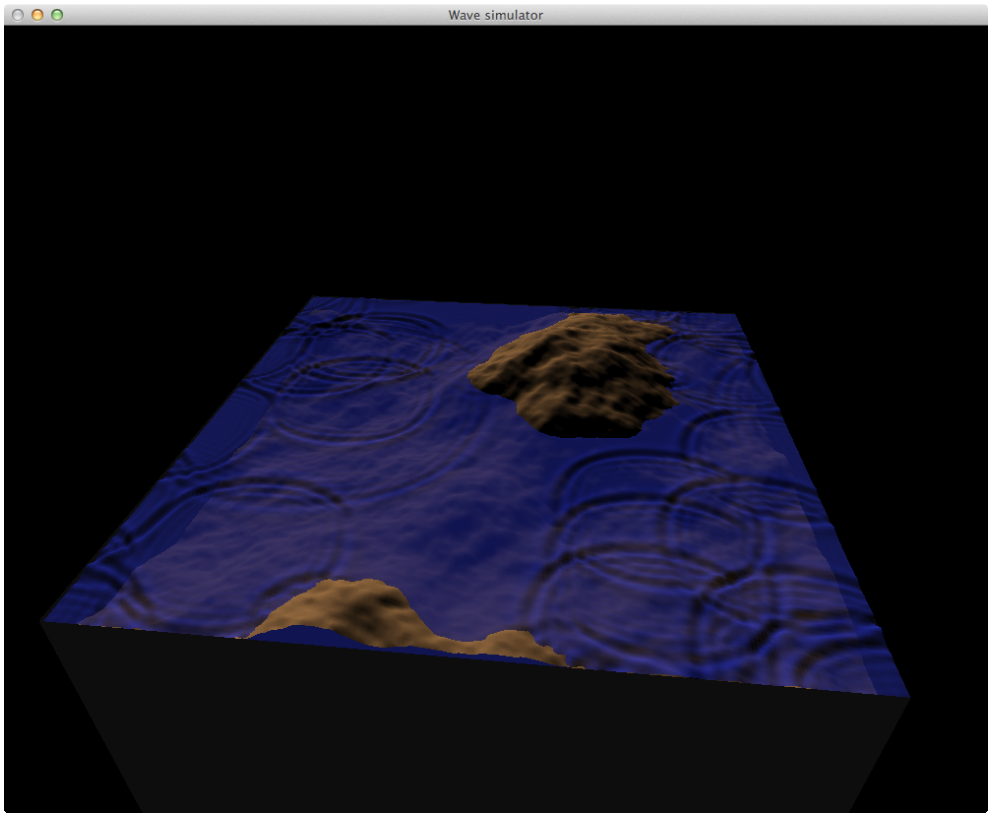

Figure 8: Beautiful waves in a realistic wave simulator