Project: 2D wave equation
INF5620

Andreas Hafver

October 15, 2012

1 Problem statement

We consider the two-dimensional wave equation with damping, position-dependent velocity and a source term:

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

with reflective boundary conditions,

$$\frac{\partial u}{\partial n} = 0, \tag{2}$$

on a rectangular domain $[0, L_x] \times [0, L_y]$ and a time domain $[0, T]$.

$u = u(x, y, t)$ is the displacement field we wish to solve for, b is the damping coefficient, $f(x, y)$ is a source term and $q(x, y)$ is related to the variable wave velocity, $c(x, y)$, by $q(x, y) = c(x, y)^2$.

2 Discretization

The above problem camn be discretised using a finite difference approach. In operator notation we have

$$[D_t D_t u + bD_{2t} u = D_x q D_x u + D_y q D_y u + f]^n_{i,j}\,, \tag{3}$$

where $(i, j) \in \{0, N_x\} \times \{0, N_y\}$ are the spatial mesh points and $n \in \{0, N\}$ are the discrete time points. We have chosen to discretize the domain such that the tuple (i, j, n) corresponds to the point with coordinate $x = (i + 1/2)dx$ and $y = (j + 1/2)dy$ at time $t = ndt$, with $dx = L_x/N_x$, $dy = L_y/N_y$ and $dt = T/N$.

We may write out the terms of (3) explicitly.
The first term on the left becomes

$$[D_t D_t u]^n_{i,j} = \frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{(dt)^2}, \tag{4}$$

and the second term becomes

$$[bD_{2t}u]^n_{i,j} = b\frac{u^{n+1}_{i,j} - u^{n-1}_{i,j}}{2dt}. \tag{5}$$

The terms on the right becomes

$$[D_x q D_x u]^n_{i,j} = \frac{[qD_x u]^n_{i+\frac{1}{2},j} - [qD_x u]^n_{i+\frac{1}{2},j}}{dx} = \frac{q^n_{i-\frac{1}{2},j}(u^n_{i+1,j} - u^n_{i,j}) - q^n_{i-\frac{1}{2},j}(u^n_{i,j} - u^n_{i-1,j})}{(dx)^2}, \tag{6}$$

$$[D_y q D_y u]^n_{i,j} = \frac{[qD_y u]^n_{i,j+\frac{1}{2}} - [qD_y u]^n_{i,j+\frac{1}{2}}}{dy} = \frac{q^n_{i,j-\frac{1}{2}}(u^n_{i,j+1} - u^n_{i,j}) - q^n_{i,j-\frac{1}{2}}(u^n_{i,j} - u^n_{i,j-1})}{(dy)^2}, \tag{7}$$

and the last term is simply

$$[f]^n_{i,j} = f^n_{i,j}. \tag{8}$$

2.1 The formula for $u^{n+1}_{i,j}$

From (3 - 8) we note that we have an explicit expression for the unknowns $u^{n+1}_{i,j}$. More specifically we may update u using the formula

$$\begin{aligned} u^{n+1}_{i,j} &= B_1 \Bigg(2u^n_{i,j} - B_2 u^{n-1}_{i,j} + \frac{C_x^2}{2}\left[q_{i+\frac{1}{2},j}(u^n_{i+1,j} - u^n_{i,j}) - q_{i-\frac{1}{2},j}(u^n_{i,j} - u^n_{i-1,j})\right] \\ &\quad + \frac{C_y^2}{2}\left[q_{i,j+\frac{1}{2}}(u^n_{i,j+1} - u^n_{i,j}) - q_{i,j-\frac{1}{2}}(u^n_{i,j} - u^n_{i,j-1})\right] + (dt)^2 f^n_{i,j} \Bigg), \end{aligned} \tag{9}$$

where we have introduced the auxiliary variables $B_1 = (1+\frac{bdt}{2})^{-1}$, $B_2 = 1-\frac{bdt}{2}$, $C_x^2 = (\frac{dt}{dx})^2$ and $C_y^2 = (\frac{dt}{dy})^2$.

2.2 A special formula for the first time step

Before we can proceed with the iterations according to the latter formula, we need to specify the initial condition. One boundary condition is the initial values of the displacement field,

$u^0_{i,j}$. However, in order to calculate $u^1_{i,j}$ we also need $u^{-1}_{i,j}$. We can get $u^{-1}_{i,j}$ by imposing a second initial condition

$$\left[\frac{\partial u}{\partial t}\right]^0_{i,j} = V(x,y) \approx [D_{2t}u]^0_{ij} = \frac{u^1_{i,j} - u^{-1}_{i,j}}{2dt}. \tag{10}$$

Combining (9) and (12) we get a special formula for the first time step, namely

$$\begin{aligned} u^{n+1}_{i,j} &= u^n_{i,j} - dtB_2V_{i,j} + \frac{C_x^2}{4}\left[q_{i+\frac{1}{2},j}(u^n_{i+1,j} - u^n_{i,j}) - q_{i-\frac{1}{2},j}(u^n_{i,j} - u^n_{i-1,j})\right] \\ &+ \frac{C_y^2}{4}\left[q_{i,j+\frac{1}{2}}(u^n_{i,j+1} - u^n_{i,j}) - q_{i,j-\frac{1}{2}}(u^n_{i,j} - u^n_{i,j-1})\right] + \frac{(dt)^2}{2}f^n_{i,j}, \end{aligned} \tag{11}$$

2.3 Reflective boundary conditions

To implant reflective boundary conditions we discretize (2) on the boundaries. For the boundary at $x = 0$ we have

$$0 = \left[\frac{\partial u}{\partial n}\right]^n_{0,j} = -\left[\frac{\partial u}{\partial x}\right]^n_{0,j} \approx -[D_{2x}u]^n_{0,j} = \frac{u^n_{-1,j} - u^n_{1,j}}{2dx}. \tag{12}$$

We may therefore enforce the reflective boundary condition by introducing a set of 'ghost cells' on the outside of our domain to that mirror the boundary points, i.e. $u^n_{-1,j} = u^n_{1,j}$. We do this for all the four boudaries in our python code.

2.4 Calcuating $q(x, y)$ at midpoints

The midpoint terms $q_{i\pm\frac{1}{2},j}$ and $q_{i,j\pm\frac{1}{2}}$ in (6) and (7) can be calculated in different ways. If the function q(x,y) is known one could evaluate it at the midpoints, otherwise one may interpolate between midpoints using an average. Both the arithmetic mean and harmonic mean were implemented in the python code.In the test section we show that the harmonic mean works best for a case with variable $q(x, y)$.

3 Implementation

Our group implemented the code in separately in both python and C++ and also in python with loops exported to C. The C++ version was parallelised using MPI and is described by Christofer Stausland in his report. In the python imementation the foliowing versions were included in the script 'wave2D_dn_vc.py'.:

1. 'scalar': All calculations done in Python loops;
2. 'vectorized': Vector operations done on NumPy arrays;

3. 'wave_inline': scalar code was converted to C and called using the weave package.

4. 'weave_inline_omp' : The C code of the latter version was parallelized using OpenMP.

(Documentation of the code can be found in the code.)

4 Verification

A set of unit tests were implemented to verify the code. These are fund in the script script 'test_wave2D_dn_vc.py'.

4.1 Constant solution

The nose package was used to check if the various implementations can reproduce a constant solution. In the function 'test_constant_solution()' the value at every point and every time step is compared to the initial constant value. All methods passed this test.

4.2 Plug wave

In the second test we check that the program is able to propagate a plug wave exactly in the x (or y) direction when $c\frac{dt}{dx} = 1$ (or $c\frac{dt}{dy} = 1$). For this test we set $q(x, y) = c^2 = constant$ and initialise u to $I(x, y) = 1$ if $x < Lx/2$ and $I(x, y) = 0$ otherwise. Nose was used to check that the solution returns to its initial configuration after one period.

Note: Initially this test did not work. The problem was that we used $x = idx$ and $y = jdy$ as our mesh points, and in that case the pulse returned to its original state after one period + $2dt$. This is because the fictious reflective walls with such a discretisation are located at $x = -dx/2$, $x = L_x + dx/2$, $y = -dy/2$ and $y = L_y + dy/2$. This means that the mesh has dimension $(L_x + dx) \times (L_y + dy)$, and hence the wave must travel longer than L_x (or L_y). To fix this problem we introduced a new discretizition such that the mesh points are located at $x = (i + 1/2)dx$ and $y = (j + 1/2)dy$ and the reflective walls are at $x = 0, L_x$ and at $y = 0, L_y$. With this alteration the test worked for all versions.

4.3 Standing wave

As a third test we considered a standing wave,

$$u(x, y, t) = e^{-bt} \cos(\omega t) \cos(m_x x\pi/L_x) \cos(m_y y\pi/L_y). \tag{13}$$

where m_x and m_y are arbitrary integers. It was verified using Mathematica that the latter is a solution of (1) if $\omega = \pi\sqrt{m_x^2 + m_y^2})$ and $b = 0$. Using $L_x = L_y = 1$ this solution also satisfy the boundary condition (2). In the test we refined the mesh 5 times, keeping the ratios of dx, dy, and dt constant, such that the error can be expressed in terms of one of

them, e.g dx. The following is an example of a test output for $m_x = 2$ and $m_y = 1$ (with $N_x = N_y$):

```
Testing standing wave:
scalar:
Nx  dx   Err   Err/dx^2
8  0.1250  0.0222  1.4228
16  0.0625  0.0065  1.6666
32  0.0312  0.0017  1.7315
64  0.0156  0.0004  1.7479
128  0.0078  0.0001  1.7521
vectorized:
Nx  dx   Err   Err/dx^2
8  0.1250  0.0222  1.4228
16  0.0625  0.0065  1.6666
32  0.0312  0.0017  1.7315
64  0.0156  0.0004  1.7479
128  0.0078  0.0001  1.7521
weave_inline:
Nx  dx   Err   Err/dx^2
8  0.1250  0.0222  1.4228
16  0.0625  0.0065  1.6666
32  0.0312  0.0017  1.7315
64  0.0156  0.0004  1.7479
128  0.0078  0.0001  1.7521
weave_inline_omp:
Nx  dx   Err   Err/dx^2
8  0.1250  0.0222  1.4228
16  0.0625  0.0065  1.6666
32  0.0312  0.0017  1.7315
64  0.0156  0.0004  1.7479
128  0.0078  0.0001  1.7521
```

We see that the error$/(dx)^2$ is more or less constant and equal for all versions , and conclude that the code passed the test.

4.4 Manufactured solution

The program was also tested for a manufactured solution in the presence of a source term $f(x, y)$, with damping (i.e. $b > 0$) and with a variable $q(x, y)$. The chosen solution was

$$u(x, y, t) = e^{-bt} \cos(x) \cos(y), \tag{14}$$

on the domain $[0, \pi] \times [0, \pi]$ with

$$q(x, y) = \sin(x) \sin(y). \tag{15}$$

It was verified by hand and using Mathematica that (14) is a solution of (1) if we include a source term

$$f(x, y, t) = e^{-bt} \sin(2x) \sin(2y). \tag{16}$$

When using arithmetic mean to determine calculate q at the midpoints the code did not achieve second order convergence. This was fixed when changing to harmonic mean. The following is the test output using harmonic mean for q.

```
Testing manufactured solution:
scalar:
Nx  dx   Err   Err/dx^2
8  0.3927  0.004427  0.0287
16  0.1963  0.001149  0.0298
32  0.0982  0.000295  0.0307
64  0.0491  0.000074  0.0306
128  0.0245  0.000018  0.0306
vectorized:
Nx  dx   Err   Err/dx^2
8  0.3927  0.0044  0.0287
16  0.1963  0.0011  0.0298
32  0.0982  0.0003  0.0307
64  0.0491  0.0001  0.0306
128  0.0245  0.0000  0.0306
weave_inline:
Nx  dx   Err   Err/dx^2
8  0.3927  0.0044  0.0287
16  0.1963  0.0011  0.0298
32  0.0982  0.0003  0.0307
64  0.0491  0.0001  0.0306
128  0.0245  0.0000  0.0306
weave_inline_omp:
Nx  dx   Err   Err/dx^2
8  0.3927  0.0044  0.0287
16  0.1963  0.0011  0.0298
32  0.0982  0.0003  0.0307
64  0.0491  0.0001  0.0306
128  0.0245  0.0000  0.0306
```

We see that the error$/(dx)^2$ is more or less constant and equal for all versions and conclude that the code passed the test.

5 Compiled loops

The weave package provided in the Python Scipy module was used to include compiled C code in the Python implementation. This package provides a function 'inline' which lets you pass arguments between Python and C functions. With this method we were able to achieve better performance tha using the vectored NumPy approach. (See section on speed comparison).

6 Parallel implementation

Two approaches were used for paralellization:

6.1 Using OpenMP to paralellize the compiled C loops called from Python

This was done by simply including a one-line directive to OpenMp in the existing C loop. Unfortunately it was not possible to test this implementation properly on multiple cores, as the ava

6.2 Using MPI in a stand-alone C++ version

This approach is more involved and is described in detail by Christoffer Stausland in his report. We wanted to combine this code with the python script, but this proved difficult for two reasons:

1. The MPI code must be compiled with the mpicc compiler and run with a special command mpirun. We did not find a way to do this with the weave package.

2. The main reason for the efficiency of the C++/MPI code is that the threads keeps their data separately.The Python code is centred around user actions at each time step that involves all off the data and therefore communication is neccessary. In the C++ code the data is written to file by each thread separately and data analysis/plotting is done afterwards.

7 Speed comparison

	scalar	vectorized	weave_inline	weave_inline_omp
15x15	22.0	1.8	1.0	1.4
30x30	27.6	1.3	1.0	1.0

45x45	29.3	1.2	1.0	1.0
60x60	29.4	1.1	1.0	1.0

Vectorization gives a significant speed up compared to Python for loops. Compiled C loops does give a small benefit over the vectored version, but surprisingly advantage decreases as the system gets larger. I suspect tat weave sends the arrays by value and not by pointer, so that data transfer becomes a limiting factor. Paralellisation with OpenMp does not give any benefit.