Project: 2D wave equation
INF5620

Andreas Hafver

October 17, 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 can 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 coordinates $x = (i + 1/2)dx$ 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

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

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

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

2.3 Reflective boundary conditions

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

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

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

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 this report the Python version is described. It was inspired by a sample code from http://www.scipy.org/PerformancePython and adapted to the current problem.

The implementation is based on two classes, 'Grid' and 'Wavesolver'. The 'Grid' class contains information about the computational domain as well as function values at the mesh points. The 'WaveSolver' class takes a grid as input and solves the given problem.

The implementation may be divided into four steps:

1. Setting up the grid:

```
g = wavesolver2d.Grid(Nx, Ny, Lx, Ly)
```

2. Setting up the problem, i.e. calculating and storing functionns values in grid. Decide how to evaluate $q(x, y)$ at midpoints (see below):

```
g.Initialize(I, V, q, q_type = 'arithmetic')
```

3. Setting up the solver:

```
s = wavesolver2d.WaveSolver(g,f,b,T,dt,dt_safety_factor,user_action,version)
```

4. Solve:

```
s.solve()
```

(Further documentation is included in the code 'wavesolver2d.py' and the tests in 'test_wavesolver2d.py' serve as examples.)

3.1 Versions

The following versions were included in the script 'wavesolver2d.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 and using Blitz arrays;
4. 'wave_fastinline': scalar code was converted to C and called using the weave package and array pointers;
5. 'weave_fastinline_omp': The C code of the latter version was parallelized using OpenMP.

3.1.1 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 throughout the simulation. In the current version the data is written to file by each thread separately and data analysis/plotting is done afterwards. The Python implementation advances one time step at the time with communication and possible user interactions in between.

3.2 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. In the code there are three options available in the initialisation of the Grid class (parameter 'q_type'):

- 'eval': Evaluate a known $q(x, y)$ function at the midpoints;

- 'arithmetic': Use an arithmetic mean to interpolate $q(x, y)$ to the midpoints;
- 'harmonic': Use an harmonic mean to interpolate $q(x, y)$ to the midpoints;

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:

```
Testing constant solution:
scalar :  Passed test
vectorized :  Passed test
blitz :  Passed test
weave_inline :  Passed test
weave_fastinline :  Passed test
weave_fastinline_omp :  Passed 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.

```
Testing plug pulse:
scalar :  Passed test
vectorized :  Passed test
blitz :  Passed test
weave_inline :  Passed test
weave_fastinline :  Passed test
weave_fastinline_omp :  Passed test
```

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  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
vectorized :
Nx  dx   Err   Err/dx^2
8  0.1250  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
blitz :
Nx  dx   Err   Err/dx^2
8  0.1250  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
weave_inline :
```

```
Nx  dx   Err   Err/dx^2
8  0.1250  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
weave_fastinline :
Nx  dx   Err   Err/dx^2
8  0.1250  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
weave_fastinline_omp :
Nx  dx   Err   Err/dx^2
8  0.1250  0.0151  0.9660
16  0.0625  0.0042  1.0844
32  0.0312  0.0011  1.1150
64  0.0156  0.0003  1.1228
128  0.0078  0.0001  1.1247
```

We see that the error$/(dx)^2$ is more or less constant and equal for all versions , indicating quadratic convergence of the error, and we therefore 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  0.0462  0.2993
16  0.1963  0.0223  0.5790
32  0.0982  0.0059  0.6134
64  0.0491  0.0015  0.6224
128  0.0245  0.0004  0.6580
vectorized :
Nx  dx   Err   Err/dx^2
8  0.3927  0.0460  0.2986
16  0.1963  0.0222  0.5770
32  0.0982  0.0059  0.6112
64  0.0491  0.0015  0.6202
128  0.0245  0.0004  0.6556
blitz :
Nx  dx   Err   Err/dx^2
8  0.3927  0.0460  0.2986
16  0.1963  0.0222  0.5770
32  0.0982  0.0059  0.6112
64  0.0491  0.0015  0.6202
128  0.0245  0.0004  0.6556
weave_inline :
Nx  dx   Err   Err/dx^2
8  0.3927  0.0460  0.2986
16  0.1963  0.0222  0.5770
32  0.0982  0.0059  0.6112
64  0.0491  0.0015  0.6202
128  0.0245  0.0004  0.6556
weave_fastinline :
Nx  dx   Err   Err/dx^2
8  0.3927  0.0460  0.2986
16  0.1963  0.0225  0.5848
32  0.0982  0.0060  0.6225
64  0.0491  0.0015  0.6332
128  0.0245  0.0004  0.6709
weave_fastinline_omp :
Nx  dx   Err   Err/dx^2
8  0.3927  0.0460  0.2986
16  0.1963  0.0225  0.5848
32  0.0982  0.0060  0.6225
```

```
64  0.0491  0.0015  0.6332
128  0.0245  0.0004  0.6709
```

We see that the error/$(dx)^2$ is more or less constant for all versions, and based on convergence the test was therefore passed. However the $Error/dx^2$ gave slightly different values on the $64x64$ and $128x128$ grid for the 'fast_inline' methods. The difference from the previous test is the inclusion of a variable $q(x, y)$ and a source term $f(x, y)$, and there could be an undiscovered bug here. Alternatively the difference could be due to differences in the way arrays are stored and different round-off errors in the various methods.

5 Speed comparison

Nx x Ny	scalar	vectorized	blitz	weave inline	weave fastinline	weave fastinline omp
25x25	3.3	0.0	0.0	0.0	0.0	0.1
	330.0	2.0	1.0	1.0	1.0	6.0
50x50	7.6	0.0	0.0	0.0	0.0	0.0
	760.0	3.0	1.0	1.0	1.0	3.0
100x100	15.2	0.1	0.0	0.0	0.0	0.1
	761.5	2.5	1.5	1.0	1.0	2.5
150x150	34.4	0.1	0.0	0.1	0.0	0.1
	1146.3	3.0	1.7	2.0	1.0	3.3
200x200	60.9	0.2	0.1	0.1	0.1	0.2
	1015.5	2.7	1.3	1.7	1.0	3.2
300x300	138.2	0.3	0.2	0.2	0.1	0.4
	1063.4	2.5	1.5	1.5	1.0	2.8
400x400	244.9	0.6	0.3	0.4	0.2	0.6
	1065.0	2.5	1.5	1.6	1.0	2.7
500x500	384.3	0.9	0.5	0.6	0.4	1.0
	1038.6	2.5	1.4	1.5	1.0	2.7

The simulation output shows the computation time in seconds and computation time normalised to the fastest method for the various versions.

Vectorization gives a significant speed up compared to Python for loops, by a factor 400. The Blitz version and weave_inline versions perform similarly and give a further 50% performance boost compared to the vectorized version. The 'weave_fastinline' version is twice as fast as the 'weave_inline' version.

The OpenMP version is slower than the normal 'fast_inline' code, but it becomes comparatively better for large systems. Possibly it could give a benefit for larger systems.

6 Conclusions and remarks

- Vectorization gives a significant performance advantage compared to plain Python loops.
- We demonstrated that it is possible to gain even more speed by exporting loops to C. It is also fairly easy to do, and the benefit is that that you can take advantage of the best features of both Python and C.
- An obvious criticism is that is that the source term $f(x, y)$ was evaluated using NumPy vectorization in all cases. A further speed up may be achieved if we define and call the function f as a C function.
- We were not able to see a benefit using OpenMP for paralellization. Possibly the there is an imbalance between communication and computation that could be resolved by setting appropriate OpenMP directives. It could also be that our systems were two small to see any benefit of parallelisation.
- We were not able to compare the speed of the Python implementation with the C++/MPI version because we were not able to run both codes on the same system.