Assignment 5

Håkon V. Treider
`https://github.com/UiO-INF3331/INF3331-HaakonVikor/`

November 1, 2015

Introduction

In this project I've implemented a finite difference scheme for the heat equation, and simulated a 2D-surface under influence by a constant heat source. To do this I have used different types of solvers, like NumPy and C and tested how fast they are compared to a pure pythonic implementation. We use Dirichlet boundary conditions on the edges, (and keep the temperature here at zero for all times.)

A small note, when running the individual solver-files, animation is on by default. As I expect the reader to be familiar with the assignment text, run
`>>> heat_equation_ui.py`
in stead and give whatever arguments you want changed on the commando line.

$$\frac{\partial u}{\partial t} - \nu \Delta u = f \tag{1}$$

..where $u(x, y, t)$ is the temperature distribution, ν is the thermal diffusivity and $f(x, y)$ is a constant heat source. By using the finite differences for the spatial and temporal derivatives we get a formula for u at the next time step:

$$\begin{aligned} u_{i,j}^{t+\mathrm{dt}} &= u_{i,j}^{t} \\ &\quad + \mathrm{dt}\,\nu \left(u_{i-1,j}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t} + u_{i,j+1}^{t} + u_{i+1,j}^{t}\right) \\ &\quad + \mathrm{dt}\left(f_{i,j}\right) \end{aligned} \tag{2}$$

5.1 Pure Python Solver

In the file `heat_equation.py` I have used nested python-lists to store the temperature distribution at the current timestep u^t and a separate for the next timestep $u^{t+\mathrm{dt}}$. The elements, which correspond to the temperature at a certain place on the mesh-grid [i][j], are stored as floating point numbers.

Let's take a look at the code, in partiular the the loop over all timesteps and mesh points:

```python
while t < t_end:
    for i in range(1,n-1): # Not including first and last element
        for j in range(1,m-1):
           u_new[i][j] = u[i][j] \
                         + dt*nu*(u[i-1][j] + u[i][j-1] -  4*u[i][j] \
                         +        u[i][j+1] + u[i+1][j]) +nu*f[i][j]

    t += dt                           # Jump to next timestep
    u = [vec[:] for vec in u_new]     # Update u for next iteration
```

The main idea is to loop over all time steps until the specified end time $t1$ =t_end is reached. For each iteration we have to go through all the inner mesh points and update them according to 2. Then, we need to copy the updated $u^{t+\mathrm{dt}}$ by value (and NOT by reference) to u^t so that we can use these values in the next iteration. This is done with a list comprehension where we loop through all the 'vectors' (lists) and copy them one by one with the slice [:]. This is way faster than i.e. `copy.deepcopy()`.

Results, Python

When run with settings $n = 50, m = 100, t0 = 0, t1 = 1000, \mathrm{dt} = 0.1, \nu = 1$ and $f = 1$ (from now referred to as "basic settings"):

```
>>> python heat_equation.py
```

and the last u is plotted, we get a temperature distribution as can be seen in figure 1

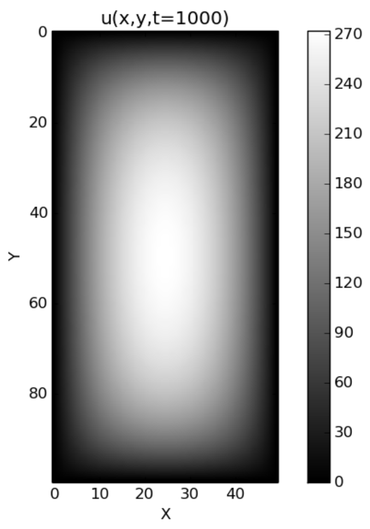

Figure 1: Temperature distribution [Pure Python-solver]

5.2 Numpy and C implementations

Numpy

In stead of computing the terms one-by-one as done in 5.1, we can compute the whole 2D-matrix (called an array) at once and thus get rid of the loop over x and y-coordinate. This is done by vectorization of the code. One important thing to keep in mind, is that we want to exclude the boundary points. This means we have to pick out only the inner points and is done with the slice [1:-1,1:-1]. Note that we can use "comma" between the indices. Here we choose all x- and y-values starting from the second element and up to, but not included, the last one (index "-1"). When our update formula (2), says ± 1 on an index, we have to move the entire slice the same. This can be seen in the code below:

```python
while t < t_end:
    u[1:-1,1:-1] = u[1:-1,1:-1] \
                   + dt*nu*(u[:-2,1:-1] + u[1:-1,:-2] - 4*u[1:-1,1:-1] \
                   +        u[1:-1,2:]  + u[2:,1:-1]) +nu*f[1:-1,1:-1]
    t += dt # Jump to next timestep
```

Since the right hand side is computed and then assigned to the left hand side in one big step, we do not need a copy of the solution at the next time step, like we did before. Combined, these two changes saves computation time (drastically!) and halves the memory usage. Nice!

Results, NumPy

The results is element wise exactly equal to the Python implementation, so no new figure is really needed (see fig.1). However, the time spent computing is drastically lower, as we'll see later.

C, via Weave

A third way to implement the time loop, is to use another language than Python, like C, and then do the heavy calculations here. Then we need some way to bridge these two languages, and my decision ended up on using Weave from the SciPy-package. We can then write C-code as a string, and call it with the `weave.inline()`-function. The time loop (and mesh grid loops) now look a lot like they did in the pure python-implementation, as can be seen below:

```c
int t,i,j;
for (t=0; t*dt<t_end+dt; t++){
    for (i=1; i<Nu[0]-1; i++) {
       for (j=1; j<Nu[1]-1; j++) {
           UN2(i,j) = U2(i,j) \
                    + dt*(U2(i-1,j) + U2(i,j-1) - 4*U2(i,j) \
                    + U2(i,j+1) + U2(i+1,j)) + nu*F2(i,j);
       }
    }
    for (i=1; i<Nu[0]-1; i++) {
       for (j=1; j<Nu[1]-1; j++) {
           U2(i,j) = UN2(i,j);
       }
    }
}
```

The C-code above is stored as a string called `code` and is then executed with the following Python-code, where we have to tell what should be interpreted as constants, arrays etc, in a list of string-names. Weave then automatically makes some new variables we can use, like the dimensions n and m, is Nu[0] and Nu[1] (has nothing to do with ν / `nu`) and a 2D-array `u` is called `U2`. Indices is separated by a comma as in NumPy, but the brackets are curly. After computing `u_new`, I copy these values to `u` through a double for-loop, to be used in the next iteration. This might seem slow, but as we'll see later, "for loops" in C are *very* fast, so this is just fine!

```
code = "int t,i,j; (.....)"
weave.inline(code, ['t_end','dt', 'u', 'un','f', 'nu'])
```

Results, C via Weave

As before, we get the exact same values computed (for all n-times-m-elements) and therefore I'd still like to refer the reader to figure 1. The advantage is of course even better performance in terms of speed, as we are going to examine in the next sections.we are going to examine in the next sections.

5.3 Testing the solvers

Accuracy

From the animation and plots, we can see that the solvers behave reasonably (in a physical sense). While keeping the edges cooled and constant, the temperature rises gradually with time until a steady state is reached. But do we know that we actually have a mathematically correct solver? To check this, we use the analytical solution when we let $t \to \infty$ and have a manufactured source term f:

$$f(x,y) = \nu\left(\left(\frac{2\pi}{(m-1)}\right)^2 + \left(\frac{2\pi}{(n-1)}\right)^2\right)\sin\left(\frac{2\pi x}{(n-1)}\right)\sin\left(\frac{2\pi y}{(m-1)}\right) \quad (3)$$

$$u_{exact}(x,y) = \sin\left(\frac{2\pi x}{n-1}\right)\sin\left(\frac{2\pi y}{m-1}\right) \quad (4)$$

This is implemented in the file `tests.py` and can be run both directly or with py.test. If run directly, it will test what happens with the error in the computation, when the mesh grid is increased.

Since all of the solvers give the exact same level of accuracy (or discrepancy between the analytic and numericaly computed solution), I'll list four runs where I have started with a 10-by-20 grid and then doubled the size three times. This can be seen in table 1. Due to the pure python-version being way too slow, it is not used to generate this data.

What we can see from the tabular data is that the error in the computation seems to decrease linearly with increasing mesh points. Since we quadruple the total number of points each time, the error should reduce by a factor 4. This is also the case.

n	m	Max error	Numpy [sec]	Weave [sec]
10	20	3.5E-2	1.150	0.041
20	40	7.8E-3	1.426	0.190
40	80	1.8E-3	2.303	0.814
80	160	4.5E-4	6.336	3.478

Table 1: Error and speed of solvers. dt = 0.1, t1 = 2000

Speed

Let's take a look at the speed-up we get from using NumPy and C. I have started with a grid size of 5-by-10, and then doubled the points three times, but this time also including the pure python-version. The results are put into table 1.

n	m	Python only	Numpy	Speed-up	Weave	Speed-up
5	10	0.307 sec	0.513 sec	0.6 X	0.006 sec	55 X
10	20	1.775 sec	0.567 sec	3,2 X	0.002 sec	93 X
20	40	8.400 sec	0.656 sec	13 X	0.090 sec	93 X
40	80	36.54 sec	1.123 sec	33 X	0.402	91 X

Table 2: Run time comparison with pure-python-version. dt = 0.1, t1 = 1000

From this, we immediately see that the larger the grid size, the better speed-up we can gain from NumPy. Meanwhile C (through Weave) gives an almost constant 90 X-improvement, which is just impressive!

5.4 User interface

When using the file `heat_equation_ui.py`, the user can specify almost any setting from the command line. If any (or few or none) arguments are given, the default values are used, which are the "basic settings" we declared above. In addition to the grid size, time step, and interval settings, the user can also use "-v" or "-verbose" to get more information throughout the computation. Also, for time measurements "-time A B" can be given and this will make the program run the timeit-module A tests with each, B iterations. The total number of runs will be A times B.

The perhaps most important setting, is the choice of solver. Since only one might be used at the time, the user is only allowed to specify either "-python", "-numpy" or "-weave". If no argument is given, weave is used for its consistent speed. To save the last solution use "-s" or "-save". To output the last solution (as an array of numbers) use "-o FILENAME". If you wish to use this as a starting point for a new calculations, specify "-i FILENAME". A test will make sure that the dimensions agree. The user can also run:
```
>>> python heat_equation_ui.py -h
```
to get a full documentation of possibilities.

5.6