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:

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

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:

```
1  while t < t_end:
2      for i in range(1,n-1): # Not including first and last element
3          for j in range(1,m-1):
4              u_new[i][j] = u[i][j] \
5                          + dt*nu*(u[i-1][j] + u[i][j-1] -  4*u[i][j] \
6                          +        u[i][j+1] + u[i+1][j]) +nu*f[i][j]
7
8      t += dt                            # Jump to next timestep
9      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`

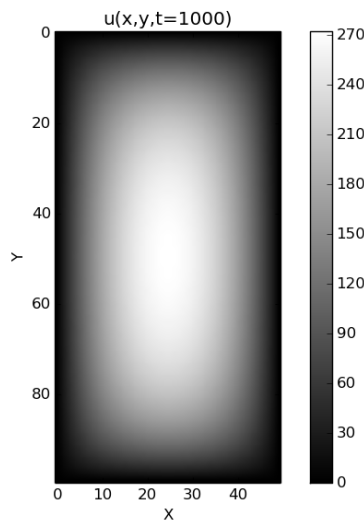and the last u is plotted, we get a temperature distribution as can be seen in figure 1

Figure 1: Temperature distribution [Pure Python-solver]

2

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:

```
1  while t < t_end:
2      u[1:-1,1:-1] = u[1:-1,1:-1] \
3                     + dt*nu*(u[:-2,1:-1] + u[1:-1,:-2] - 4*u[1:-1,1:-1] \
4                     +        u[1:-1,2:]  + u[2:,1:-1]) +nu*f[1:-1,1:-1]
5      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:

```
1  int t,i,j;
2  for (t=0; t*dt<t_end+dt; t++){
3      for (i=1; i<Nu[0]-1; i++) {
4          for (j=1; j<Nu[1]-1; j++) {
5              UN2(i,j) = U2(i,j) \
6                         + dt*(U2(i-1,j) + U2(i,j-1) - 4*U2(i,j) \
7                         + U2(i,j+1) + U2(i+1,j)) + nu*F2(i,j);
8          }
9      }
10     for (i=1; i<Nu[0]-1; i++) {
11         for (j=1; j<Nu[1]-1; j++) {
12             U2(i,j) = UN2(i,j);
13         }
14     }
15 }
```

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!

```
1  code = "int t,i,j; (.....)"
2  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.