Report for assignment 5 in INF3331

INF3331-Oystein, oystekap@student.matnat.uio.no

Sunday, 1. November 2015

Assignment 5.1: Python implementation of the heat equation

The update formula was given in the assignment text, so basically all that remained was to put the update formula inside a triple loop (over time values (t), row indexes (i) and column indexes (j)). The new temperature distribution list (new_u) is copied at the end of each time step so that the old u (u) is up to date for the next one. The rest of the code is just checks, conversions and optional printouts as explained in the comments.

Assignment 5.2: NumPy and C implementations

The numpy is similar to the python solution, except that I have used vectorisation. That is, instead of looping over the values for i and j, the code updates the array in a single step (for each time value). To illustrate the difference, imagine the the simpler case of adding all elements of one list to the corresponding elements of another. In pure python syntax, this can be achieved like this

```python
for i in range(len(lst1)):
    for j in range(len(lst1[i])):
        lst3[i][j]=lst1[i][j]+lst2[i][j]
```

whereas with numpy arrays this can be achieved by simply adding the two arrays

```python
lst3=lst1+lst2
```

Here all the values for lst1[i][j] and lst2[i][j] are added in a single step without the need for a loop over i and j. The numpy solution in the assignment is just a somewhat more complicated case where it is specified which coordinates should be added together.

In the weave implementation a string of c code is passed to the weave.inline function, which executes the code.

Assignment 5.3: Testing

The script consists of two functions: make_u_f, which initializes the analytic_u and f arrays, and test_heat_equation, which executes the tests as specified in the assignment text. The line py.test.cmdline.main(["-v", 'heat_equation_test.py']) enables the script to be run like an ordinary python script with printouts.

Assignment 5.4: Develop a user interface

I have used the argparse module to create a common user interface, as the assignment text specifies. The add.argument function is used to add arguments to the implementation. The arguments can be passed from the command line like this:
python heat_equation_ui.py -i numpy -v # run numpy implementation with verbose mode activated
python heat_equation_ui.py -i weave -dt 0.2 -plot_file plot.png # run weave implementation with dt set to 0.2 and save plot as plot.png
etc.

The user can specify the dimensions of the arrays by passing e.g. -dim 50 40 (creates an initial temperature array and heat source of dimensions 50x40 and all values set to zero and one, respectively) or passing customized arrays (e.g. -u 'zeros((30, 40))' -f 'ones((30, 40))'. Note that the arguments must be enclosed in quotation marks (" or '); they will be converted to arrays inside the program. If the user passes arguments for both dimensions and customized arrays, the former will override the latter. Also note that some of the arguments can be given in either short or long forms, e.g. -v or –verbose.

Runtime comparison

The run times for (three runs of) the different loops are as follows:
Pure python:
82.376 sec
83.010 sec

80.991 sec
Average: 82.126 sec

Numpy:
0.987 sec
1.013 sec
1.028 sec
Average: 1.009 sec

Weave:
0.347 sec
0.331 sec
0.352 sec
Average: 0.343 sec

Python is a high-level, interpreted language, whereas C is a low-level language, which basically means that every time you run a python script the computer has to translate it into something more computerish, whereas C programs are compiled into something that is much more similar to the way computers operates, so that after the compilation is done, a C program can run very fast. Weave, of course, gives you the best of both worlds by letting the user write all but the most time-consuming parts of the code in python. Similarly, numpy is faster than pure python, as numpy array loops are compiled in a way that enables the computer to execute many operation at the same time.