An Example of Computational Fluid Dynamics in Python

03 September 2014

Introduction

The Python code in this directory is an example from one of the most common applications of HPC resources: fluid dynamics. It shows how a simple fluid dynamics problem can be run using Python.

Fluid Dynamics

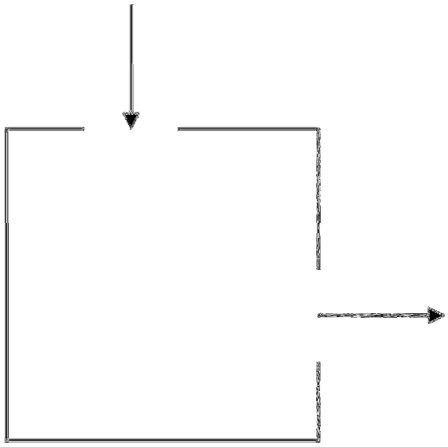
Fluid dynamics is the study of the mechanics of fluid flow, liquids and gases in motion. This can encompass aero- and hydro-dynamics. It has wide ranging applications from vessel and structure design to weather and traffic modelling. Simulating and solving fluid dynamic problems requires large computational resources.

Fluid dynamics is an example of a **continuous system** which can be described by **partial differential equations**. For a computer to simulate these systems, the equations must be discretised onto a grid. If this grid is regular, then a finite difference approach can be used. Using this method means that the value at any point in the grid is updated using some combination of the neighbouring points.

Discretisation is the process of approximating a continuous (i.e. infinite-dimensional) problem by a finite-dimensional problem suitable for a computer. This is often accomplished by putting the calculations into a grid or similar construct.

The Problem

In this example the finite difference approach is used to determine the flow pattern of a fluid in a cavity. For simplicity, the liquid is assumed to have zero viscosity which implies that there can be no vortices (i.e. no whirlpools) in the flow. The cavity is a square box with an inlet on one side and an outlet on another as shown below.



A bit of maths

In two dimensions it is easiest to work with the *stream function* Ψ (see below for how this relates to the fluid velocity). For zero viscosity, Ψ satisfies the following equation:

$$\nabla^2\Psi = \frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2} = 0$$

The finite difference version of this equation is:

$$\Psi_{i-1,j} + \Psi_{i+1,j} + \Psi_{i,j-1} + \Psi_{i,j+1} - 4\Psi_{i,j} = 0$$

With the boundary values fixed, the stream function can be calculated for each point in the grid by averaging the value at that point with its four nearest neighbours (the points above, below, left and right). We can repeatedly update these averages until the algorithm converges on a solution that stays unchanged by the averaging process. This simple approach to solving a PDE is called the **Jacobi Algorithm**.

In order to obtain the flow pattern of the fluid in the cavity we want to compute the velocity field $\tilde{u}$. The x and y components of $\tilde{u}$ are related to the stream function by

$$u_x = \frac{\partial\Psi}{\partial y} = \frac{1}{2}\left(\Psi_{i,j+1} - \Psi_{i,j-1}\right)$$
$$u_y = \frac{\partial\Psi}{\partial x} = \frac{1}{2}\left(\Psi_{i-1,j} - \Psi_{i+1,j}\right)$$

This means that the velocity of the fluid at each grid point can also be calculated from the surrounding grid points.

An Algorithm

The outline of the algorithm for calculating the velocities is as follows:

```
Set the boundary values for stream function
WHILE (convergence == FALSE) DO
  FOR EACH interior grid point DO
    Update value of stream function by averaging with its 4
    nearest neighbours
  END DO
  Check for convergence
END DO
FOR EACH interior grid point DO
  Calculate x component of velocity
  Calculate y component of velocity
END DO
```

For simplicity, here we simply run the calculation for a fixed number of iterations; a real simulation would continue until some chosen accuracy was achieved.

Running the code

configs/ contains sample configuration files. These define:

- The number of iterations of the Jacobi algorithm to run – you will need more iteration steps to converge larger grids.
- The size of each edge of the grid.
- The position and width of the inlet on the top edge of the grid.
- The position and height of the outlet on the right edge of the grid.

For example:

```
$ cat configs/32x32.cfg
[Simulation]
iterations=1000
[Grid]
edge=32
[Inlet]
x=5
width=5
[Outlet]
y=15
height=5
```

The implementation of the Jacobi algorithm can be run as follows, providing a configuration file and the name of a file for the output:

```
$ python cfd.py configs/32x32.cfg 32x32.dat
```

As the program is running you should see output that looks something like:

```
2D CFD Simulation
=================
   Iterations = 1000
         Edge = 32
      Inlet X = 5
  Inlet width = 5
     Outlet Y = 15
Outlet height = 5
Grid size     = 32 x 32

Initialisation took 0.00000s

Starting main Jacobi loop...
...finished

Calculation took 0.55800s
```

The program will produce a text output file called `32x32.dat`. The output file contains the width and height of grid followed by width*height rows specifying:

- Row
- Column
- X component of velocity
- Y component of velocity
- Scaled magnitude of the velocity:

$$\left(\left(u_x + u_y\right)^2\right)^{0.3}$$

For example:

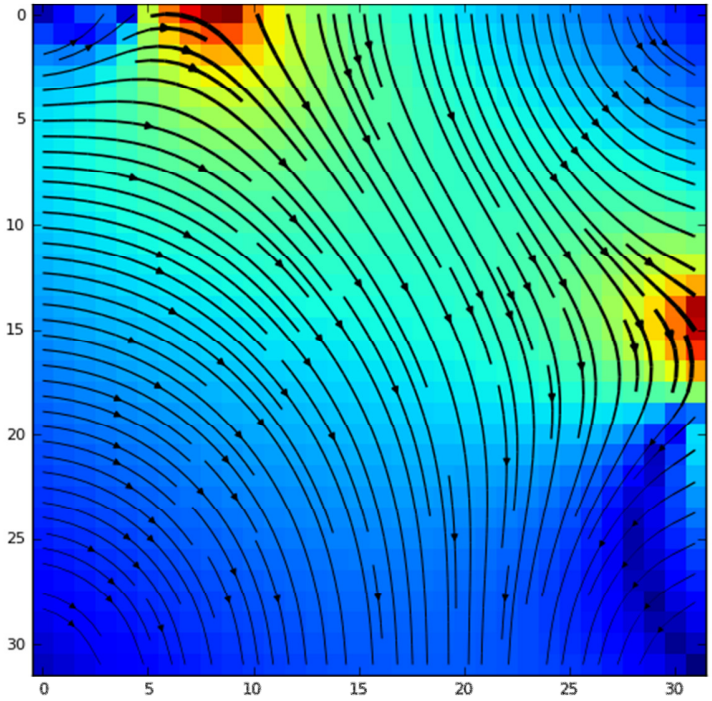
```
$ head 32x32.dat
32 32
    0     0    0.06593   -0.05693    0.05924
    0     1    0.08183   -0.12046    0.14195
    0     2    0.11904   -0.19930    0.22014
    0     3    0.20769   -0.30709    0.25028
    0     4    0.44520   -0.46584    0.09746
    0     5    0.67803   -0.20215    0.64047
```

```
0     6    0.75028    0.01750    0.85338
0     7    0.74941    0.22417    0.98406
0     8    0.67537    0.44551    1.07086
```

The data can be plotted as follows:

```
$ python plot_flow.py 32x32.dat 32x32.png
```

This produces a graphical representation of the final state of the simulation. For example, 32x32.png should contain a picture similar to the image below:



If the fluid is flowing along the top then down the right-hand edge, rather than through the middle of the cavity, then this is an indication that the Jacobi algorithm has not yet converged. Convergence requires more iterations on larger problem sizes.

Numpy and scipy

The implementation is very simple. A more efficient and elegant implementation would use the capabilities of the Python numpy and scipy libraries.