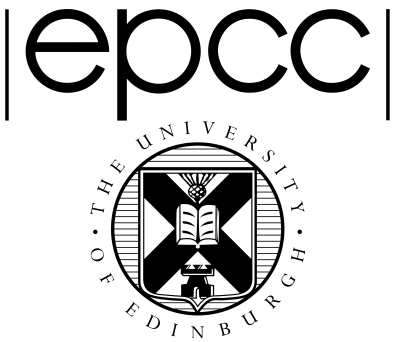
Scientific Python: Computational Fluid Dynamics

17 July 2014



Introduction and Aims

This exercise takes an example from one of the most common applications of HPC resources: Fluid Dynamics. We will look at how a simple fluid dynamics problem can be run using Python and NumPy; and how Fortran and C code can be called from within Python. The exercise will compare the performance of the different approaches.

We will also use this exercise to demonstrate the use of matplotlib to plot a visulisation of the simulation results.

This exercise aims to introduce:

- Python lists and functions
- Basic NumPy array manipulation
- Plotting using matplotlib
- Calling Fortran/C from Python
- Benchmarking Python performance

Fluid Dynamics

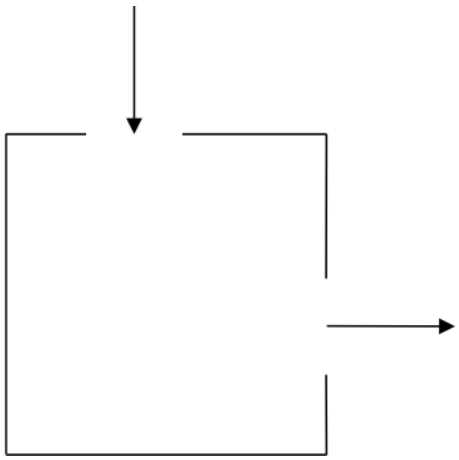
Fluid Dynamics is the study of the mechanics of fluid flow, liquids and gases in motion. This can encompass aero- and hydro-dynamics. It has wide ranging applications from vessel and structure design to weather and traffic modelling. Simulating and solving fluid dynamic problems requires large computational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

Fluid dynamics is an example of 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 exercise 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 process continues 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.

Using Python

This calculation is useful to look at in Python for a number of reasons:

- The problem can be scaled to an arbitrary size
- It requires the use of 2-dimensional lists/arrays
- The algorithm can easily be implemented in Python, NumPy,
- Fortran and C
- Visualising the results demonstrates the use of matplotlib

You are given a basic code that uses Python lists to run the simulation. Look at the structure of the code. In particular, note:

- How the external "jacobi" function is included
- How the lists are declared and initialised to zero
- How the timing works

Exercises

Get the Code Bundle

Use `wget` to copy the file `cfd-python.tar.gz` from the ARCHER web pages at the URL provided by the trainers and unpack the tarball to a local directory. The tarball should contain the following subdirectories:

- `python`: Contains the basic Python version of the code and the plotting utility
- `verfiy`: Contains various outputs to verify your results against

First Run and Verification

Firstly, you should verify that your copy of the code is producing the correct results.

Move into the `python` subdirectory and run the program with:

```
prompt:~/python> ./cfd.py 1 1000
```

This runs the CFD simulation with a *scalefactor* of 1 and 1000 Jacobi *iteration steps*. The *scalefactor* determines the size of the simulation (1 corresponds to a 32x32 gris, 2 to a 64x64 grid, etc.); *interation steps* are the number of iterations performed in the Jacobi algorithm – you will need more iteration steps to converge larger grids. As the program is running you should see output that looks something like:

```
2D CFD Simulation
=================
Scale Factor = 1
  Iterations = 1000

Initialisation took 0.00007s

Grid size = 32 x 32

Starting main Jacobi loop...
...finished

Calculation took 0.53424s
```

The program will produce an text output file called `flow.dat` with the computed velocities at each grid point. A simple verification is to use `diff` to compare your output with one of the verification datasets. For example:

```
prompt:~/python> diff flow.dat ../verify/flow_1_1000.dat
```

`diff` will only produce any output if it finds any differences between the two files. If you see any differences at this point, please ask a tutor.

Initial Benchmarking

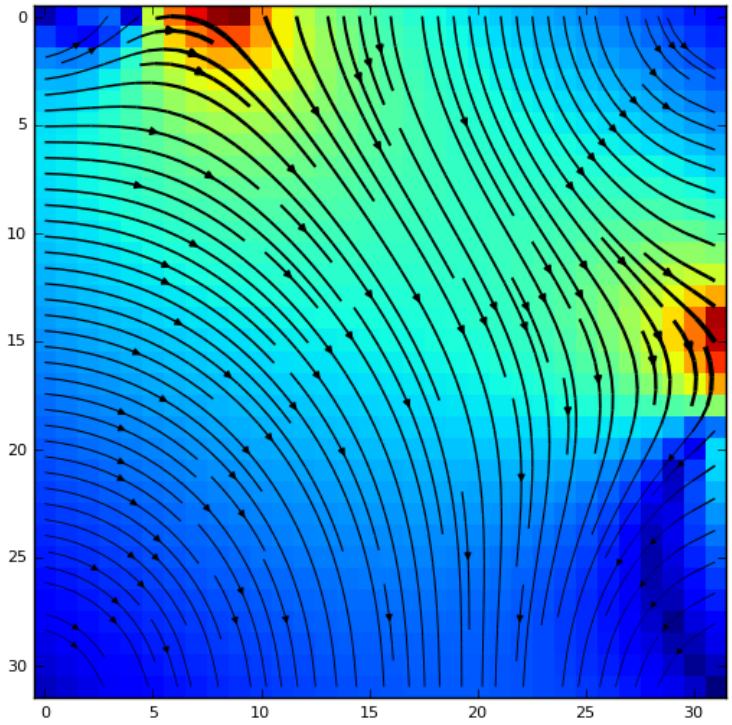
You should now produce some baseline figures with which to compare our future versions of the code. You should pick a set of representative problem sizes (defined by scale and number of iterations) that run in a sensible time on your machine but do not complete instantaneously. (A good place to start is with scale factor 2 and 5000 iterations. You will also need some smaller and larger examples.)

Record the benchmarking calculation times for future reference.

The directory includes a utility called `plot_flow.py` that produces a graphical representation of the final state of the simulation. You can use this to produce a PNG image as follows:

```
prompt:~/python> ./plot_flow.py flow.dat flow.png
```

now `flow.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.

Using numpy Arrays

We will now re-factor the CFD code to use numpy arrays rather than Python lists. This has a number of advantages:

- numpy is closely integrated with matplotlib and using numpy arrays will allow us to produce the visualisation directly from our simulation rather than using a separate utility.
- numpy arrays should allow us to access better performance using more concise code.
- numpy arrays are directly compatible with native code produced by Fortran and C compilers. This will allow us to re-code the key part of our algorithm and achieve better performance while still having the benefits of coding in Python.

Replace the `psi` and `tmp` lists in the code with numpy arrays. (Remember to take a copy of the code in a new directory before you start work so you do not lose the original version.) You will need the statement:

```
import numpy as np
```

at the top of all your source files to ensure you can access the numpy functionality. The arrays will need to be implemented in the main function and all of the other functions where they are used.

Declaring and zeroing numpy arrays can be done in a single statement such as:

```
psi = np.zeroes((m+2, n+2))
```

Once you think you have the correct code, run your new program and compare the output with that from the original code. If you are convinced that the output is the same then move on and benchmark your new code.

What do you find? Has using numpy arrays increased the performance of the code? Can you think of an explanation of why the performance has altered in the way that it has?

Can you change the implementation to produce a better performing version of the CFD code? Hint, you should use array index syntax if you have not already done so to specify blocks of arrays to operate on.

Incorporating matplotlib

matplotlib and numpy have a very close relationship: matplotlib can use the typing and structure of numpy arrays to simplify the plotting of data.

We will use matplotlib to add a function to out program that produces an image of the final state of the flow.

Define the plotting function

Define a function in your main `cfd.py` file called `plot_flow`. This function should take two arguments: the first is `psi` the numpy array containing the stream function values and the second is the name of the file to save the image to. For example:

```
def plot_flow(psi, outfile):
```

This function should use the stream function values to compute the x and y components of the velocities and the magnitude of the velocity and store them in appropriate numpy arrays. Remember, you will need to extract the velocities of the internal part of the matrix and exclude the fixed boundary conditions.

Once you have these arrays you need to initialise the matplotlib module to plot to an image file rather than the console with the following lines:

```
import matplotlib
# Plot to image file without need for X server
matplotlib.use("Agg")
```

Next, you should import the required matplotlib functionality with:

```
from matplotlib import pyplot as plt
from matplotlib import cm
fig = plt.figure()
plt.subplot(1, 1, 1)
```

The first import is the core plotting fucntionality, the second line is the colour-mapping functionality, the third line makes the fig object available for saving our figure to an image file later, and the final line specifies the subplot we are working on.

The simplest plot to produce is a heatmap of the magnitude of the velocity.
You can use the `imshow` function to do this in a single line. Assuming that the velocity magnitudes are in a numpy array called `vmag`:

```
plt.imshow(vmag)
```

To produce the image file we need to add one further line:

```
fig.savefig(outfile)
```

The final matplotlib code to produce a simple heatmap of the velocity magnitude all pulled together looks like:

```
import matplotlib
# Plot to image file without need for X server
matplotlib.use("Agg")
from matplotlib import pyplot as plt
from matplotlib import cm
fig = plt.figure()
plt.subplot(1, 1, 1)
plt.imshow(vmag)
fig.savefig(outfile)
```

You can now start to add more features to make the plot more useful (all of these additions should go before the line that saves the image).

Add a colour bar to quantify the heatmap:

```
plt.colorbar()
```

Add streamlines indicating the direction of the flow velocity: this is more complex as you first need to set up regularly spaced values to describe your grid. Set up the x and y ranges with:

```
(i, j) = vmag.shape
x = np.linspace(0, i-1, i)
y = np.linspace(0, j-1, j)
```

(Look at the online `linspace` documentation to understand what is happening here.) Once you have these values you can use the `streamplot` function to add the streamlines (assuming the x and y velocity components are saved in `xvel` and `yval` respectively):

```
plt.streamplot(x, y, xvel, yvel, color='k', density=1.5)
```

`color='k'` sets the streamline colour to black and `density=1.5` sets the number of streamlines plotted. You can also vary the width of the streamlines using the magnitude of the velocity:

```
lw = 3 * vmag/vmag.max()
```

```
plt.streamplot(x, y, xvel, yvel, color='k', density=1.5,
linewidth=lw)
```

(The factor, 3, just scales the effect to a nice size.)

Using Scipy

You should have found in the numpy implementation that simply iterating over the indices using loops does not improve performance over using Python lists. You may have managed to overcome this performance hit using indexing. The indexing solution can produce very fast code but it is at the expense of readability- it is quite hard to see what the code is doing just by inspection.

The Scipy function, `convolve`, allows us to keep most of our performance improvement and also improve the readability of the code.

You should write a version of the jacobi.py routine that uses the convolve function and benchmark its performance against your previous versions.

Note, you will need to define the mask to use for the convolution. This is essentially a stencil that you place over the current element that describes how to combine the surrounding elements to produce the required function. In order to write this version of the function you will need to design your stencil and express it in the code as a 2D numpy array.

Use the online reference documentation for `convolve` to work out how to use it.

Calling External Code from Python

We are going to continue using our numpy implementation of the CFD code to illustrate calling Fortran code from Python. Calling any external code (written in any language) from Python requires that

- the data we are passing from Python to the external code (and vice-versa) is of the correct size and layout;
- the external routines are packages in a way that allows them to be called from Python.

You must have the required compilers installed on the machine you are using to be able to compile external code. All Linux machines should have gcc (a C compiler) installed by default and it should be trivial to add the Fortran compiler: gfortran. For Mac users you can install Xcode to get gcc and then get gfortran from the web. Windows users can download and install MinGW to get access to gcc and gfortran.

Using numpy ensures that the data is of the correct size and layout to be passed to external Fortran and C code. The Python package f2py provides a simple way to package external code in a way that can be imported and called from Python programs.

Calling Fortran from Python

It is easiest to start with Fortran as this is what f2py was designed for and the packaging process is simpler than for C code.

All of the files and commands should be present/be issued in the directory containing the main cfd.py code. You should take a separate copy of the code before you start this process to ensure that you do not overwrite any previous work.

We will replace our numpy `jacobi` function with a Fortran subroutine that does the same calculation. If you know Fortran then you can produce your own code, if not you can download a version we have prepared, this code (`jacobi.f90`) is also included below:

```
!
! Jacobi routine for CFD calculation
!
subroutine jacobi(m, n, niter, psi)
    implicit none

    integer, intent(in) :: m, n, niter
    real*8, dimension(0:m+1, 0:n+1), intent(inout) :: psi
    !f2py intent(in) m
    !f2py intent(in) n
    !f2py intent(in) niter
```

```
    !f2py intent(inplace) psi

    integer :: iter, i, j

    ! Define the temporary array
    real*8, dimension(0:m+1, 0:n+1) :: tmp

    ! Zero the temporary array
    tmp = 0.0

    ! Number of iterations
    do iter = 1, niter
       ! Compute the stream function values
       do i = 1, m
          do j = 1, n
             tmp(j, i) = 0.25 * (psi(j+1,i) + psi(j-1,i) + psi(j,i+1) +
psi(j,i-1))
          end do
       end do
       ! Copy new version back into psi (without boundaries)
       psi(1:m,1:n) = tmp(1:m,1:n)
    end do

end subroutine jacobi
```

The lines with `!f2py` are used to define the attributes of the argument variables for the f2py tool. `intent(in)` specifies that the variables are input variables only, `intent(inplace)` specifies that the variable is being passed by reference (so any changes to the variable in this routine will be reflected in the calling code).

We now need to use the f2py tool to produce a *signature* of this Fortran subroutine that specifies how to construct the interface between Python and Fortran. You do this with the following command:

```
f2py jacobi.f90 -m jacobi -h jacobi.pyf
```

This will produce a text file called `jacobi.pyf` that contains the signature for the Fortran subroutine "jacobi". We now need to combine this signature with the actual Fortran code to produce a *dynamic library* that can be imported by Python. We use the f2py command to do this (note that this requires a Fortran compiler to be available):

```
f2py –c jacobi.pyf jacobi.f90
```

If this works it should produce a file called "jacobi.so", this is the dynamic library that can be imported by Python. We can test the import works using an interactive IPython shell with the following commands:

```
prompt:~/python> ipython
Python 2.7.6 |Anaconda 1.9.2 (64-bit)| (default, Jan 17 2014, 10:13:17)
```

```
Type "copyright", "credits" or "license" for more information.

IPython 1.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: from jacobi import jacobi

In [2]: print jacobi.__doc__
jacobi(niter,psi,[m,n])

Wrapper for ``jacobi``.

Parameters
----------
niter : input int
psi :  rank-2 array('d') with bounds (m + 2,n + 2)

Other Parameters
----------------
m : input int, optional
    Default: (shape(psi,0)-2)
n : input int, optional
    Default: (shape(psi,1)-2)
```

As you can see, the calling sequence from Python differs from the argument sequence specified in the Fortran code. The two arguments `m` and `n` are not required when calling from Python. f2py has realized that these variables are related to the dimensions of the psi array. The interface wrapper knows how to extract these values from the numpy array in Python so you can just call the function with:

```
from Jacobi import Jacobi
jacobi(niter, psi)
```

This means that this version should be directly compatible with the numpy version (as the calling sequence is the same) of `cfd.py` – all you need to do is replace the file `jacobi.py` with `jacobi.so`.

If you have a Fortran compiler you should test the performance of the Fortran version against the pure Python lists and numpy versions of the code.

Calling C from Python

Calling C from Python is slightly more complicated. Perhaps surprisingly, the simplest way to set this up is to use f2py to generate the dynamic library as we did for Fortran above. The additional complications are:

1. You will need to generate the Jacobi.pyf signature file by hand as f2py cannot analyse the C code to extract the argument types and data dependencies. (This is much simpler in this example as we can use the signature generated for the Fortran version above as a starting point.)
2. The interface does not support multidimensional C arrays so all arrays will arrive in the C code as 1D arrays. You will need to compute the corrct indexing to get the correct array element.

The C version of the Jacobi function is shown below and can be download from the course material web page. Note that psi is a 1D array in the routine and we calculate the indices required from knowledge of the ordering of C array elements.

```
/*
 * Jacobi iteration
 */
void jacobi(int m, int n, int niter, double *psi)
{

    // Local variables
    int i, j, iter;
    int idx, jdx, im1, ip1;
    double tmp[(m+2)*(n+2)];

    // Zero the tmp array
    for ( i=0; i<n+2; i++)  {
        for ( j=0; j<m+2; j++)  {
            // Compute correct index
            jdx  = i*(n+2) + j;
            tmp[jdx] = 0.0;
        }
    }
    // Jacobi iterations
    for (iter=0; iter<niter; iter++) {
        for ( i=1; i<n+1; i++)  {
            for ( j=1; j<m+1; j++)  {
                // Compute correct indices
                im1 = (i-1)*(n+2) + j;
                ip1 = (i+1)*(n+2) + j;
                jdx = i*(n+2) + j;
                tmp[jdx] = 0.25 * (psi[jdx-1]+psi[jdx+1]+psi[im1]+psi[ip1]);
            }
        }
        // Copy inner part of tmp to psi
        for ( i=1; i<n+1; i++)  {
            for ( j=1; j<m+1; j++)  {
                // Compute correct index
                jdx = i*(n+2) + j;
                psi[jdx] = tmp[jdx];
            }
        }
    }
}
```

As for the Fortran example, the first step is to produce the signature file: jacobi.pyf. The signature is shown below.

```
!    -*- c -*-
! Note: the context of this file is case sensitive.

python module jacobi ! in
    interface  ! in :jacobi
        subroutine jacobi(m,n,niter,psi) ! in :jacobi:jacobi.c
            intent(c) jacobi
            intent(c)
            integer, optional,intent(in),check((shape(psi,0)-
2)==m),depend(psi) :: m=(shape(psi,0)-2)
            integer, optional,intent(in),check((shape(psi,1)-
2)==n),depend(psi) :: n=(shape(psi,1)-2)
            integer intent(in) :: niter
            real*8 dimension(m + 2,n + 2),intent(inplace) :: psi
        end subroutine jacobi
    end interface
end python module jacobi
```

This is identical to the version for the Fortran code except for the addition of two lines:

```
            intent(c) jacobi
            intent(c)
```

The first line tells f2py that the routine being compiled is C code (rather than Fortran) and the second adds the `intent(c)` attribute to all the argument descriptions to tell f2py that they are C variables.

Once you have the signature file you can produce the dynamic library in the same way as you did for Fortran:

```
f2py –c jacobi.pyf jacobi.c
```

As before, this will produce a file called “jacobi.so” that can be imported as above.

If you have a C compiler you should test the performance of the C version against the pure Python lists and numpy versions of the code.

Summary

You should have a number of different versions of the code to compare the performance.

- What does the benchmarking reveal?
- Can you explain any of the performance differences you see?
- How does the performance vary with system size?
- Which version of the code would you prefer to work on and why?

Further Exercises

Other things to try could include:

- Adding extra plotting complexity – plot the magnitude of the velocity as a 3D surface.
- Measuring the impact of optimization parameters in on the Fortran/C code.
- Parallelise the code using mpi4py.
- Re-implement the program in an object-oriented approach.