Laminar Incompressible Viscous Flow over a Backward Facing Step

May 29, 2018

1 Backward Facing Step Problem using Python

The following code is written in python to solve the laminar incompressible flow over a backward facing step problem

1.1 Method of solution : Pressure Correction Technique SIMPLE

1.2 Author of the code : Ramkumar

1.3 Developed on : saturday, 19/05/2018 @ 12:44 PM

1.4 Ref. for validation : verification and validation guide-Abaqus/CFD 6th unit

The following modules are imported for computation

```
In [1]: import numpy as np
        from copy import copy as cp
```

The numpy module is used for performing numerically intensive tasks like grid development and to deal with arrays. The copy module is used to over-ride the default absolute-referencing of variables by python.

The following lines describle the fluid domain size and number of grid points used in computation

```
In [2]: length = 30.0           # length of fluid domain
        width  = 1.0            # width of domain
        nx = 51                 # number of grids on x direction
        ny = 21                 # number of grids on y direction
```

The following couple of lines generate the mesh grid required for computation

```
In [3]: X,Y = np.meshgrid(np.linspace(0,length,nx),np.linspace(-width/2,width/2,ny))
        dx = length/float(nx-1); dy = width/float(ny-1)
```

dx stands for space step in x-direction and dy stands for the same in y-direction

```
In [4]: Nss = 0                 # step wall start node
        Nse = int(ny/2)         # step wall end node
```

The above 2 lines give the start and end nodes of wall at west boundary.
The fluid parameters such as density and Reynolds Number are defined by the following lines

```
In [5]: rho = 1                 # fluid density
        Re = 800.0              # Reynolds Number based on channel height
        dt = 1e-02              # computational time step
        Nstp = 1000             # number of time steps
        Vavg = 1.0              # average velocity @ inlet (parabolic inlet specified)
```

Here, the variable Re defines the Reynolds number of flow field based on channel height, which was twice the height of step. the variable dt defines the computational time step for simulation and Nstp defines the number of time-steps that the computation has to run.

The computation is performed over a staggered-grid so as to eliminate the checker board pattern in pressure and velocity field that occurs due to improper initial values. The following lines define the number of grid points for pressure, x and y velocities. Here the forward staggered grid approach is made, i.e. some pressure points fall into ghost cells leaving velocity to be present exactly at boundary line.

```
In [6]: npx = nx+1; npy = ny+1  # pressure points
        nux = npx-1; nuy = npy  # x velocity points
        nvx = npx; nvy = npy-1  # y velocity points
```

The molecular viscosity is calculated from the Reynolds number, using average face velocity and width as follows

```
In [7]: mu = rho*Vavg*width/Re
```

Then the coefficient matrices of primitive variables are initialized using numpy arrays, as follows

```
In [8]: u = np.zeros([nuy,nux],dtype = float)
        apu = cp(u); aeu = cp(u); awu = cp(u); anu = cp(u); asu = cp(u); bu = cp(u);
        du  = cp(u);

        v = np.zeros([nvy,nvx], dtype = float)
        apv = cp(v); aev = cp(v); awv = cp(v); anv = cp(v); asv = cp(v); bv = cp(v);
        dv  = cp(v);

        p = np.zeros([npy,npx],dtype = float)
        app = cp(p); aep = cp(p); awp = cp(p); anp = cp(p); asp = cp(p); Bp = cp(p);
```

The coefficients for diffusion and initial conditions are defined by the following lines

```
In [9]: ap0 = rho*dx*dy/dt
        De = mu/dx*dy; Dw = cp(De); Dn = mu/dy*dx; Ds = cp(Dn)
        u[int(nuy/2):nuy,0] = cp(Vavg); us = cp(u); vs = cp(v); pp = cp(p)
```

Entering the main loop of computation, the sections in following program are explained under Explanations with reference number

```
In [ ]: for itr in range(Nstp):
            # x-momentum equation coefficients computation
            for i in range(1,nux-1):
                for j in range(1,nuy-1):
                    Fe = rho*dy*0.5*(u[j,i]+u[j,i+1])                          # REF - 001
                    Fw = rho*dy*0.5*(u[j,i]+u[j,i-1])
                    Fn = rho*dx*0.5*(v[j,i]+v[j,i+1])
                    Fs = rho*dx*0.5*(v[j-1,i]+v[j-1,i+1])

                    Pe = Fe/De; Pw = Fw/Dw; Pn = Fn/Dn; Ps = Fs/Ds              # REF - 002

                    aeu[j,i] = De*np.max([0,(1-0.1*abs(Pe))**5]) + np.max([0,-Fe])# REF - 003
                    awu[j,i] = Dw*np.max([0,(1-0.1*abs(Pw))**5]) + np.max([0, Fw])
                    anu[j,i] = Dn*np.max([0,(1-0.1*abs(Pn))**5]) + np.max([0,-Fn])
                    asu[j,i] = Ds*np.max([0,(1-0.1*abs(Ps))**5]) + np.max([0, Fs])
                    apu[j,i] = aeu[j,i]+awu[j,i]+anu[j,i]+asu[j,i]+ap0
                    bu[j,i] = ap0*u[j,i]; du[j,i] = dy/apu[j,i]

            # y-momentum equation coefficients computation                     # REF - 004
            for i in range(1,nvx-1):
                for j in range(1,nvy-1):
                    Fe = rho*dy*0.5*(u[j,i]+u[j+1,i])
                    Fw = rho*dy*0.5*(u[j,i-1]+u[j+1,i-1])
                    Fn = rho*dx*0.5*(v[j,i]+v[j+1,i])
                    Fs = rho*dx*0.5*(v[j,i]+v[j-1,i])

                    Pe = Fe/De; Pw = Fw/Dw; Pn = Fn/Dn; Ps = Fs/Ds

                    aev[j,i] = De*np.max([0,(1-0.1*abs(Pe))**5]) + np.max([0,-Fe])
                    awv[j,i] = Dw*np.max([0,(1-0.1*abs(Pw))**5]) + np.max([0, Fw])
                    anv[j,i] = Dn*np.max([0,(1-0.1*abs(Pn))**5]) + np.max([0,-Fn])
                    asv[j,i] = Ds*np.max([0,(1-0.1*abs(Ps))**5]) + np.max([0, Fs])
                    apv[j,i] = aev[j,i]+awv[j,i]+anv[j,i]+asv[j,i]+ap0
                    bv[j,i] = ap0*v[j,i]; dv[j,i] = dx/apv[j,i]

            # momentum equations Solution
            uprev = cp(us); vprev = cp(vs)
            for iterate_v in range(100):
                for i in range(1,nux-1):
                    for j in range(1,nuy-1):                                    # REF - 005
                        us[j,i] = 1/apu[j,i]*(aeu[j,i]*us[j,i+1]+awu[j,i]*us[j,i-1]+\
                        anu[j,i]*us[j+1,i]+asu[j,i]*us[j-1,i]+bu[j,i]) + \
                        du[j,i]*(p[j,i]-p[j,i+1])
                for i in range(1,nvx-1):
                    for j in range(1,nvy-1):                                    # REF - 006
                        vs[j,i] = 1/apv[j,i]*(aev[j,i]*vs[j,i+1]+awv[j,i]*vs[j,i-1]+\
                        anv[j,i]*vs[j+1,i]+asv[j,i]*vs[j-1,i]+bv[j,i]) + \
                        dv[j,i]*(p[j,i]-p[j+1,i])
```

```
    us[:,nux-1] = cp(us[:,nux-2])
    vs[:,nvx-1] = cp(vs[:,nvx-2])
    convergence_u = np.max(abs(uprev-us)); uprev = cp(us)
    convergence_v = np.max(abs(vprev-vs)); vprev = cp(vs)

    if convergence_u<1e-7 and convergence_v<1e-7:                    # REF - 007
        break

# pressure correction equation coefficients computation
for i in range(1,npx-1):
    for j in range(1,npy-1):
        aep[j,i] = rho*dy*du[j,i]                                    # REF - 008
        awp[j,i] = rho*dy*du[j,i-1]
        anp[j,i] = rho*dx*dv[j,i]
        asp[j,i] = rho*dx*dv[j-1,i]
        app[j,i] = aep[j,i]+awp[j,i]+anp[j,i]+asp[j,i]
        Bp[j,i] = rho*(dy*(us[j,i-1]-us[j,i])+dx*(vs[j-1,i]-vs[j,i]))

# pressure correction equation Solution
pp = np.zeros([npy,npx], dtype = float); pprev = cp(pp)
for iterate_p in range(100):
    for i in range(1,npx-1):                                         # REF - 009
        for j in range(1,npy-1):
            pp[j,i] = 1/app[j,i]*(aep[j,i]*pp[j,i+1]+awp[j,i]*pp[j,i-1]+\
            anp[j,i]*pp[j+1,i]+asp[j,i]*pp[j-1,i] + Bp[j,i])
    pp[:,0] = cp(pp[:,1])
    pp[:,npx-1] = cp(pp[:,npx-2])
    pp[0,:] = cp(pp[1,:])
    pp[npy-1,:] = cp(pp[npy-2,:])

    convergence_p = np.max(abs(pprev-pp)); pprev = cp(pp)

    if convergence_p < 1e-5:
        break

# pressure and velocity Correction
p = p + 0.1*pp                                                       # REF - 010
for i in range(1,nux-1):
    for j in range(1,nuy-1):
        u[j,i] = us[j,i] + du[j,i]*(pp[j,i]-pp[j,i+1])
for i in range(1,nvx-1):
    for j in range(1,nvy-1):
        v[j,i] = vs[j,i] + dv[j,i]*(pp[j,i]-pp[j+1,i])
u[:,nux-1] = cp(u[:,nux-2])
v[:,nvx-1] = cp(v[:,nvx-2])

print("\n TimeStep : ",itr+1)
```

```
        if np.isnan(p).any():                                    # REF - 011
            print("\n ERROR!! Solution diverged")
            break
```

1.4.1 Explanation for above code

REF - 001 The X-momentum equation coefficients are computed in this snippet, the coefficents of convective fluxes are computed exactly at this reference point. Fe Fw Fn Fs are the convective fluxes in east, west, north and south faces of u-control volume.

REF - 002 The power-law scheme is implemented in the coefficients computation, for that, the Peclet numbers are computed for each fluxes along with each side's diffusion terms.

REF - 003 The coefficients of the X-momentum equation is computed at this snippet along with the source terms

REF - 004 The stuff that occured with x-momentum equations, i.e. coefficients computation, is happening with the Y-momentum equations in same order

REF - 005 The implicit type solution to X-momentum equation occurs in this snippet

REF - 006 The implicit type solution to Y-momentum equation occurs in this snippet

REF - 007 The convergence condition for implicit solution of momentum equations are checked at this snippet of code.

REF - 008 The coefficients for Pressure Correction Equation is computed in this set of lines.

REF - 009 The implicit type pressure correction equation solution is carried out at this snippet of code

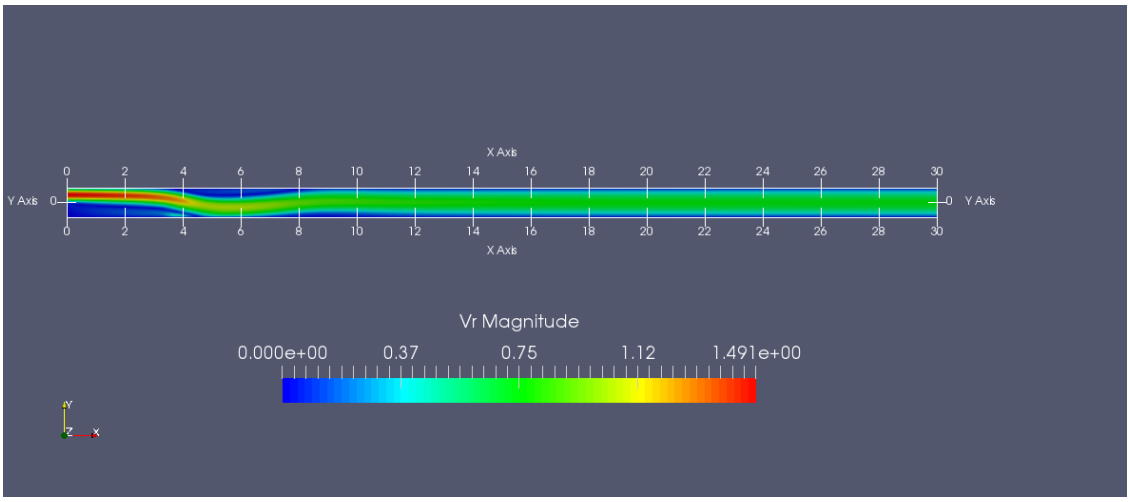
REF - 010 The Correction of Pressure at Velocity occurs in this snippet.

REF - 011 The check for solution divergence by encountering Not-A-Number values in results is determined by these set of lines

1.5 Result of Computation

```
In [14]: from IPython.display import Image
         Image("velocity_magnitude.png")
```

Out[14]:



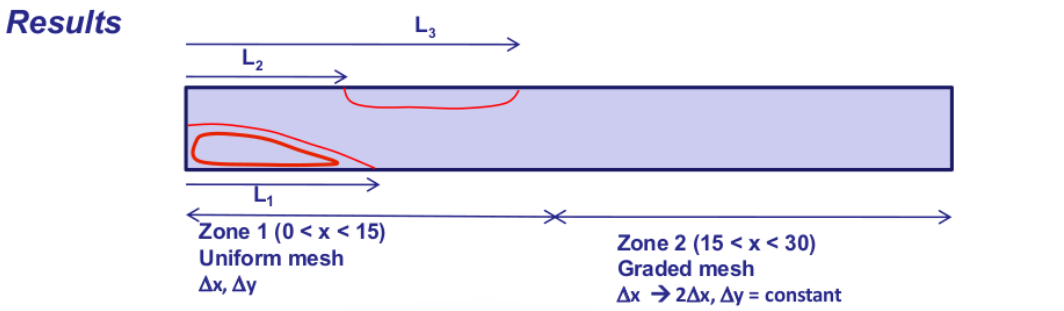
The above contour shows the velocity magnitude across the fluid domain. The output is taken from Paraview through import of CSV file. The CSV file is generated within the Ipython Console.

1.6 Validation

The results from benchmark test case is given in the table below

```
In [18]: Image("ref_result.png")
```

Out[18]:



	Mesh (Across the channel x along the channel length)	L_1 Length from the step face to the lower re-attachment point	L_2 Length from the step face to upper separation point	L_3 Length of the upper separation bubble
Gartling (1990)	40x800	6.1	4.85	10.48
Abaqus/CFD	Fine 80x1200x1 (Zone 1) 80x832x1 (Zone 2)	5.9919	4.9113	10.334
Abaqus/CFD	Medium 40x600x1 (Zone 1) 40x416x1 (Zone 2)	5.7471	4.8379	10.101
Abaqus/CFD	Coarse 20x300x1 (Zone 1) 20x208x1 (Zone 2)	4.5018	3.9659	8.7748

Present Work Results: for a Grid size of 41X1201 : L1 = 6.0; L2 = 3.7; L3 = 11.0

1.7 References for present work

H K Versteeg and W Malalasekera, "An Introduction to Computational Fluid Dynamics, The Finite Volume Method"

Patankar S V, "Numerical Heat Transfer And Fluid Flow"

Abaqus/CFD - Sample Problems, Abaqus 6.10 PDF File