barbatruc

Release 0.2.0a,

Team COOP

Sep 08, 2021

Contents

Contents:

CHAPTER 1

Welcome to Barbatruc

1.1 The BARBATRUC project

BARBATRUC is a computationnal fluid dynamics (CFD) package dedicated to training. The target audience is people starting CFD. The (tentative) objectives are:

- **Physics**: Toy with basic CFD configurations (Poiseuille, Lid-driven cavity, Karman-street) with a light-weight solver.
- **Scientific computing**: Tinker several solvers for CFD, a Navier-Stockes finite difference incompressible solver and a Lattice Boltzmann solver.
- **Software development**: See how two solvers show the same interface, by messing around with the post-processing of the data-structure regardless of the solver. Explore the continuous integration, linting, testing and automatic documentation featured by the package.

1.2 Installation

Barbatruc is available on Python Package Index

```
pip install barbatruc
```

You might also want to make a git clone from the barbatruc gitlab.com repository (under creation)

```
git clone git@gitlab.com:cerfacs/barbatruc.git
```

> Note : The gitlab.com repository is read only. It is a mirror of the actual repository hosted privately at Cerfacs Forge

1.3 Usage

Type `barbatruc` and you will gat the CLI menu.

```
Usage: barbatruc [OPTIONS] COMMAND [ARGS]...

  ---------------   BARBATRUC  --------------------

  You are now using the Command line interface of BARBATRUC, a Python3
  training package on CFD , created at CERFACS (https://cerfacs.fr).

  This is a python package currently installed in your python environement.

Options:
  --help  Show this message and exit.

Commands:
  create  Wizard creating a BARBATRUC script NAME.py.
  docu    Open Barbatruc documentation
```

Create a new project with the `create`option.

```
>barbatruc create barbibul cfd_poiseuille
```

This will create a local copy of the Poiseuille example. Now you can edit the newly created script `barbibul.py` and run it with

```
>python barbibul.py
```

> Note : Soon the software will be able to say "barbatruc" when you interact with it. As this feature is missing now, please say "barbatruc" whenever you type a command or start a run.

1.4 An example:

A typical barbatruc script looks like:

```python
import numpy as np
from barbatruc.fluid_domain import DomainRectFluid
from barbatruc.fd_ns_2d import NS_fd_2D_explicit

def cfd_lid_driven(nsave, t_end):
    """Startup computation
    solve a lid_driven_cavity  problem
    """
    dom = DomainRectFluid(dimx=0.82, dimy=0.61, delta_x=0.02)
    vel = 0.0001
    dom.switch_bc_xmin_wall_noslip()
    dom.switch_bc_xmax_wall_noslip()
    dom.switch_bc_ymax_moving_wall(vel_u=vel)
    dom.switch_bc_ymin_wall_noslip()
    time = 0.0
    time_step = t_end/nsave
    solver = NS_fd_2D_explicit(dom, obs_ib_factor=0.9)
    for i in range(nsave):
        time += time_step
```

(continues on next page)

(continued from previous page)

```
        solver.iteration(time, time_step)
        print('  Max u:', np.max(dom.fields["vel_u"]))
        print('  Time :', time)
        print('  Iteration %d' % (i))
    dom.show_fields()
    dom.show_flow()
    print('Normal end of execution.')

if __name__ == "__main__":
    cfd_lid_driven(10, 0.50)
```

1.5 Why this weird name?

Barbapapa is a 1970 childern picture book where the character can morph into anything while saying "Barba-trick", in french "Barbatruc". The Navier Stokes solver stems from Pr. Lorena Barba's 12 steps to Naviers Stokes course. Initially a running gag between co-workers, `barbatruc` eventually became the name of the package and landed on PyPI.

> Note to Pr. Barba, if you are reading this. All apologies for this pun, you are probably totally sick of it. Many thanks for making available to all your 12-steps-to-NS course.

CHAPTER 2

How to interact with barbatruc ?

2.1 LAUNCHING

To have the results :

Go to the folder where your file is :

```
cd folder/
```

To launch the simulation :

```
python3 cfd_cylinder.py
```

and you get :

```
        .  Rectangular Grid.
=======================

     y_max: 0.5m, nodes: 100
     +------------------------------+
     |                              |
     |                              |
     |                              |
     |                              |
     |                              |
     +------------------------------+
   (0,0)                          x_max: 1.0m, nodes: 200


                periodic
     +------------------------------+
     |                              |
   inlet
     |                            outlet
```

(continues on next page)

(continued from previous page)

```
     |                            |
     |                            |
     +----------------------------+
                  periodic

.  Fields stats
=======================
.  min|avg|max
 vel_u : 1.0 | 1.0 | 1.0
 vel_v : 0.1 | 0.10000000000000002 | 0.1
 scal : 0.0 | 0.0 | 0.0
 press : 0.0 | 0.0 | 0.0




================================

  Iteration 1/200, Time :, 0.02s
  Reynolds : 111.92422560738474

.  Fields stats
=======================
.  min|avg|max
 vel_u : 0.12019082043493529 | 0.9940530128736734 | 1.3605910402433394
 vel_v : -0.3009720354811324 | 0.09675753736792093 | 0.4664584367055691
 scal : 0.0 | 0.0 | 0.0
 press : -2.2893733111942414 | 0.0 | 1.7798464788156056



================================

  Iteration 2/200, Time :, 0.04s
  Reynolds : 112.09757332022016

.  Fields stats
=======================
.  min|avg|max
 vel_u : 0.023893464480471707 | 0.9953864667605914 | 1.3804213543256483
 vel_v : -0.329530409060508 | 0.0944418335727916 | 0.5040636022356708
 scal : 0.0 | 0.0 | 0.0
 press : -1.4163675489683598 | -7.105427357601002e-18 | 1.4278708080524152



================================

  Iteration 3/200, Time :, 0.06s
  Reynolds : 111.90076036667641

.  Fields stats
=======================
.  min|avg|max
 vel_u : -0.06308165801559718 | 0.9935347780711958 | 1.3528315628913046
 vel_v : -0.3441370891819653 | 0.0922008027913569 | 0.5008225700261179
 scal : 0.0 | 0.0 | 0.0
```

(continues on next page)

(continued from previous page)

```
 press : -1.276280378390031 | 2.1316282072803004e-18 | 1.2726099477265453



==============================
...
```

To interrupt the simulation :

```
ctrl + c
```

To know how much time the simulation takes :

```
time python3 cfd_cylinder.py


real        1m10.974s
user        1m49.836s
sys         0m14.051s
```

2.1.1 Remarks :

In the python file, a function is defined to compute the results :

```
def cfd_cylinder(nsave):
```

time_step is the observation time and not the one which respect the CFL.

nsave is the number of iteration :

```
time_step = t_end/nsave

t_end = 4.0 * lenght / vel
```

2.2 CHANGING CASE

2.2.1 Trick

Comment what you don't want to use.

To change the solver :

```
 #solver = Lattice(dom, max_vel=2*vel)
 solver = NS_fd_2D_explicit(dom, obs_ib_factor=0.9)
```

obs_ ib _factor can be considered as a porosity factor of the obstacle (cylinder).

Take an example with known results, apply the two solvers, choose the one which is better for this case.

To change the boundary conditions :

```
dom.switch_bc_xmin_inlet(vel_u=vel)
#dom.switch_bc_xmax_outlet()
dom.switch_bc_ymax_wall_noslip()
dom.switch_bc_ymin_wall_noslip()
```

2.2.2 To change the case :

The **default parameters** are in *fluid_domain.py*. If they are **specified** in one of the examples (*cfd_poiseuille.py*, *cfd_lid _driven.py*, *cfd_cylinder.py*) then those values are used.

1. Change the dimensions with delta_x = dimy/number of cells :

```
dom = DomainRectFluid(dimx=0.82, dimy=0.61, delta_x=0.02)
```

2. To have the Reynolds number that you want either change the velocity or the viscosity :

For Re = 20 and default viscosity nu_ = 1.0 :

```
vel = 32.7869
```

Be aware that by changing the velocity, you need to change the time : t_end = dimx/vel.

or with default velocity vel = 0.0001

```
dom = DomainRectFluid(dimx=0.82, dimy=0.61, delta_x=0.02, nu = 0.000003)
```

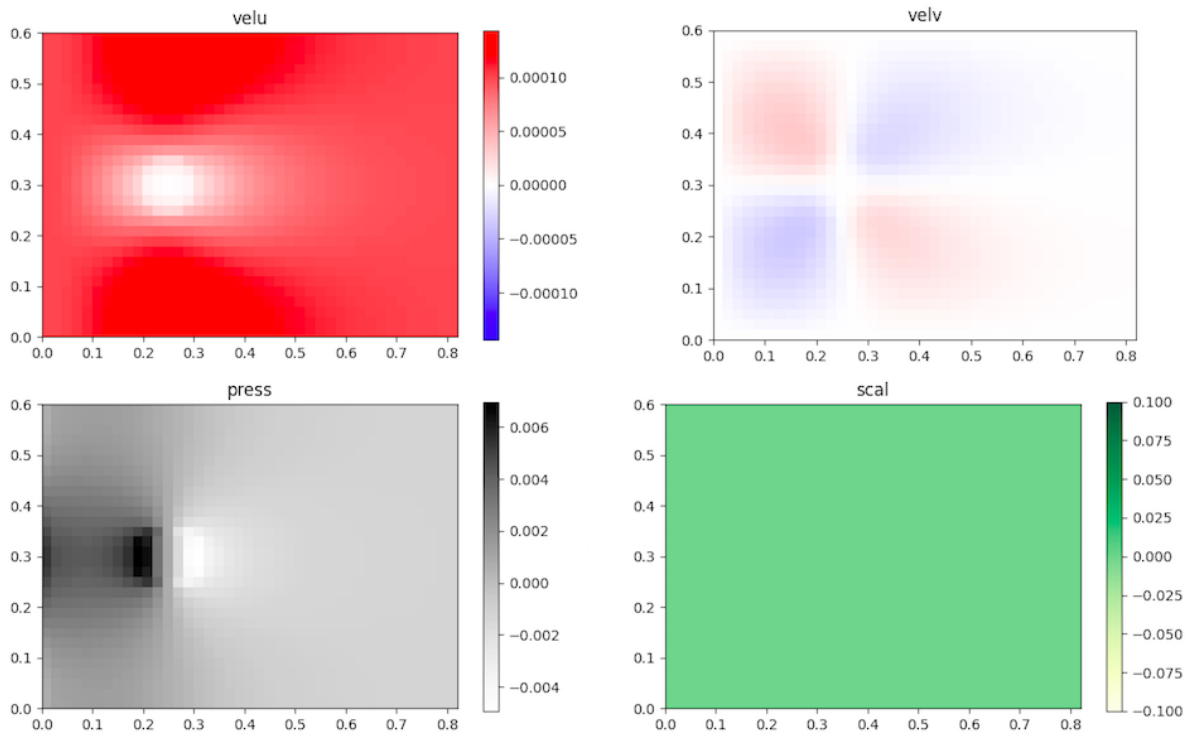
2.3 VIEWING

Results for a flow around a cylinder

2.3.1 Fields at Small Reynolds Number :

```
dom.show_fields()
```

- vel_u : horizontal velocity
- vel_v : vertical velocity
- press : pressure
- scal : passive scalar which can simulate the trajectory of a leak of pollutant or the dissipation of a perfume in a room

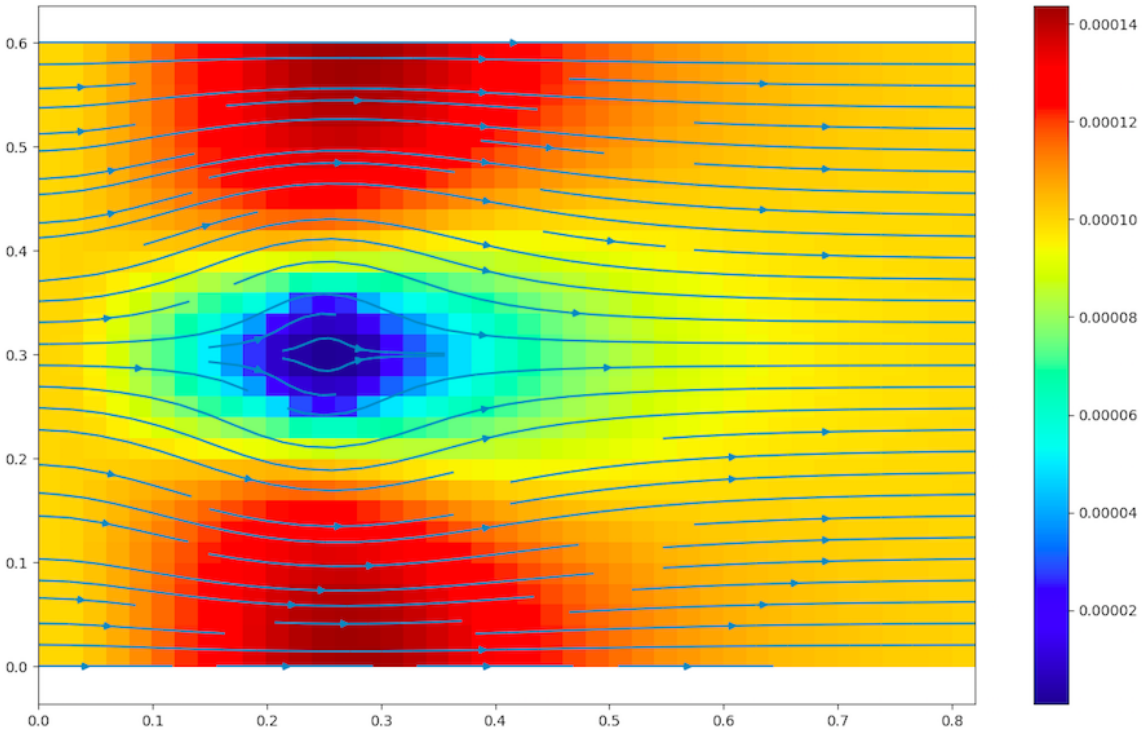


cylinderfields

2.3.2 The flow output at Small Reynolds Number :

```
dom.show_flow()
```

- velocity field with its intensity (color code)
- streamlines (blue vectors)

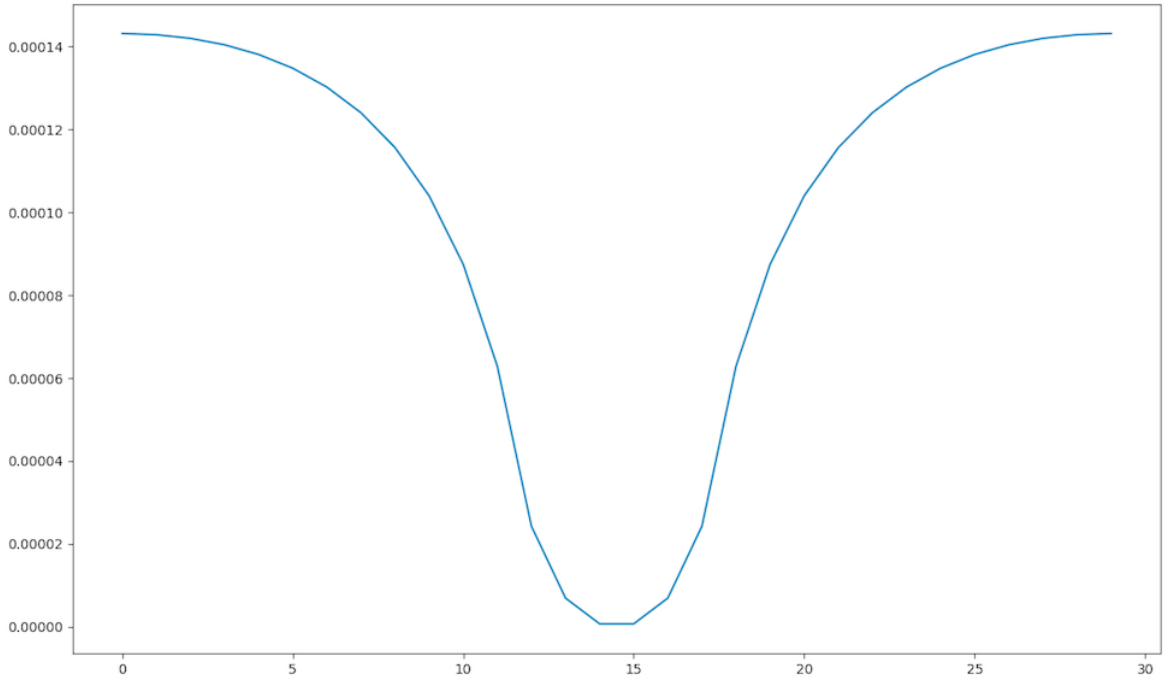


cylinderflow

2.3.3 The velocity profile at Small Reynolds Number :

To get the profile at x=cste : change the value of **xtgt**.

```
dom.show_profile_y(xtgt=0.25, dimx=0.82, delta_x=0.02)
```



cylinderprofile

2.3.4 The monitors at High Reynolds Number :

This kind of monitoring only works with he Navier-Stokes solver; the LBM solver is not ready yet.

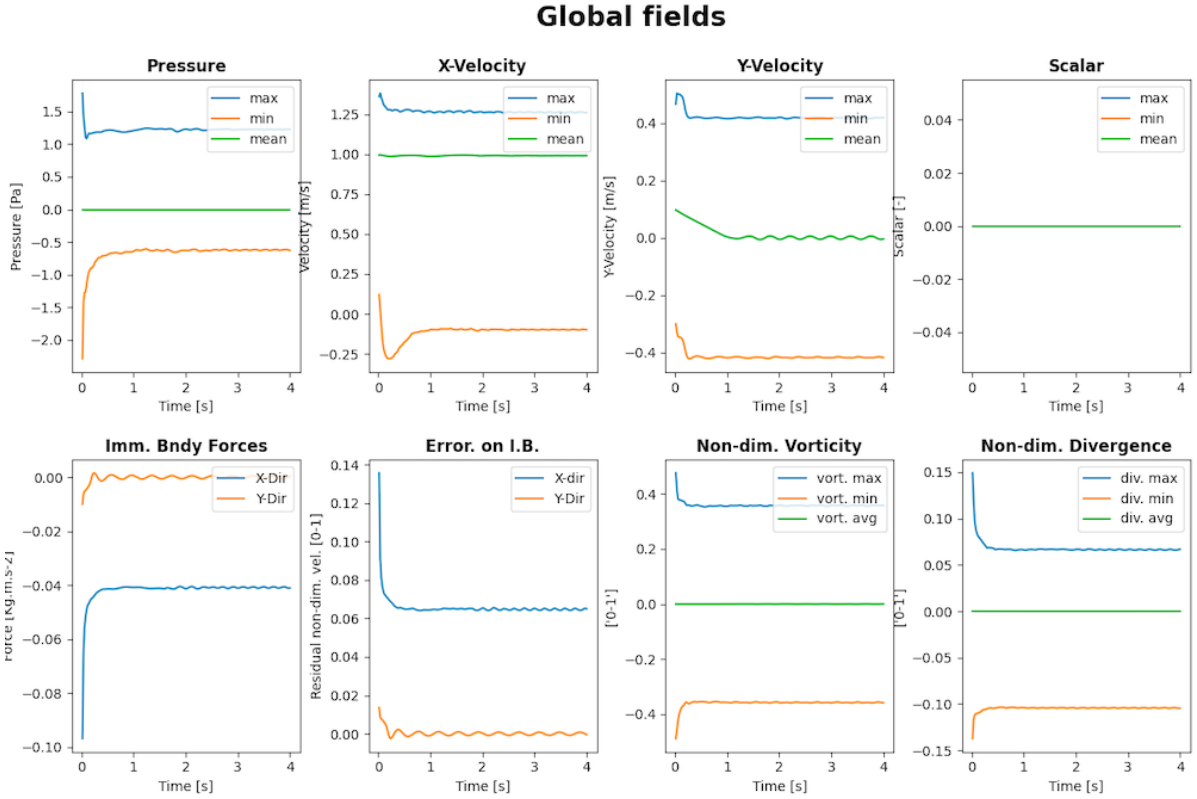
```
barbatruc monitor solver
```

The **solver monitoring** gives the following data in terms of time:

- the velocity for the number of step of Poisson
- the Poisson final residual
- the Maximum velocity

karmanmonitor

```
barbatruc monitor glob
```



karmanmonitor

The **solver monitoring** gives the following data in terms of time:

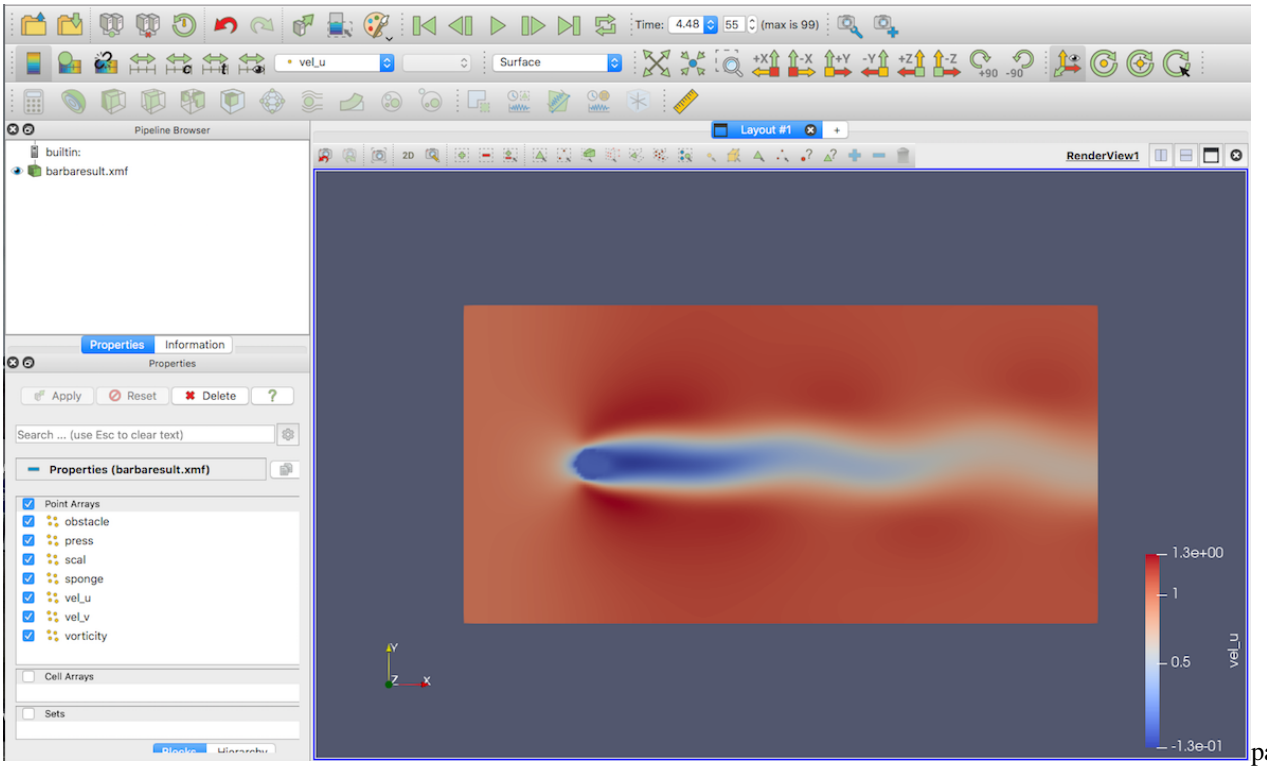
- pressure
- x-velocity
- y-velocity
- scalar
- Immersed boundaries forces, for further information
- Error on Immersed boundary
- Non dimensional Vorticity
- Non dimensional Divergence

2.3.5 Paraview

For the cylinder case, .xmf files are created and can be read with paraview. Those file are created in the time loop thanks to :

```
for i in range(nsave):
        dom.dump_paraview(time=time)
```

Open **barbaresult.xmf** in paraview :



paraviewcylinder

CHAPTER 3

Poiseuille Periodic 2D flow

3.1 1. Domain parameters

- Lx = dimx = 4 m
- Ly = width + d_x = 1.07 m
- delta_x = width/number of cells = 1/15 m
- U_0 = 1 m/s
- **Re = 100, the most important parameter**

To have great results : **dimx must be at least twice dimy.**

3.2 2. Boundary conditions

- Lower wall: no slip
- upper wall: no slip
- input: periodic
- output: periodic

A volumic force will be added to the flow.

Thanks to the Poiseuille equation, the source term is added as follow:

```
    dom.set_source_terms({
        "force_x": 8. * nu_ / width**2. * np.ones(dom.shape) * vel,
        "force_y": np.zeros(dom.shape),
        "scal": np.zeros(dom.shape)
    })
```

3.3 Reference case

3.3.1 the script

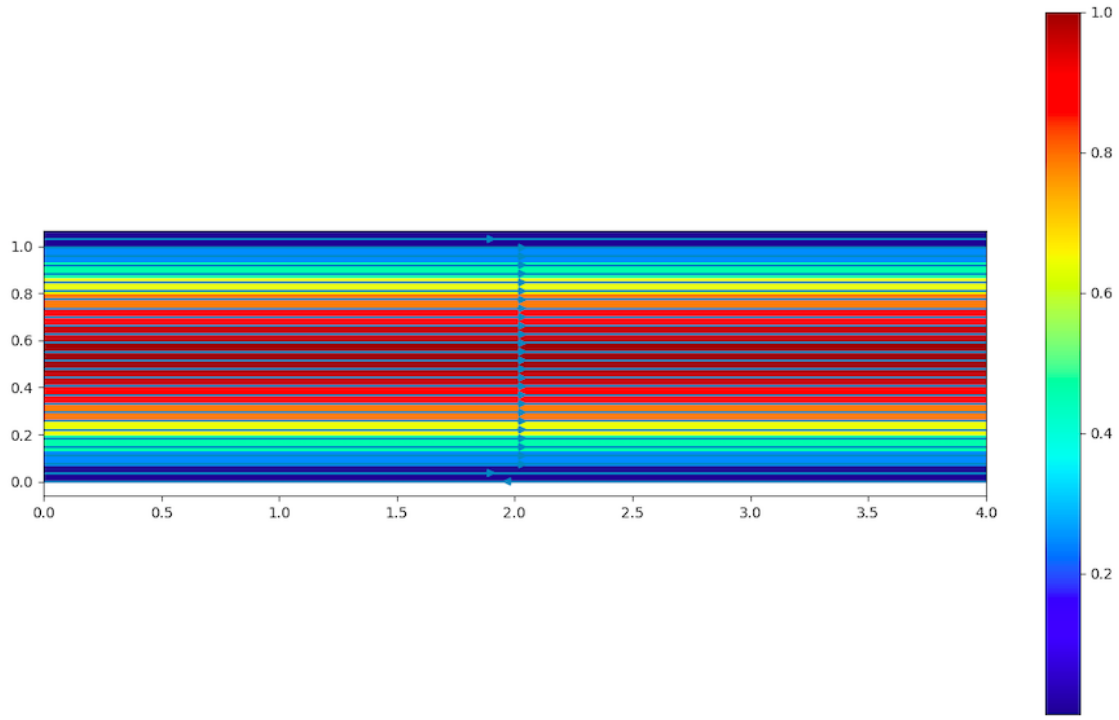
```
        """Example on how to solve a Poiseuille problem  with the navier stokes solver
↪"""

        import numpy as np
        import matplotlib.pyplot as plt
        from barbatruc.fluid_domain import DomainRectFluid
        from barbatruc.fd_ns_2d import NS_fd_2D_explicit
        #from barbatruc.lattice import Lattice

        __all__ = ["cfd_poiseuille"]


        # pylint: disable=duplicate-code
        def cfd_poiseuille(nsave):
            """Startup computation
            solve a poiseuille problem
            """
            width = 1.
            length = width*4
            dimx=length
            vel = 1.0
            t_end = 10.*length/vel
            nu_ = 0.01
            ncell = 15
            d_x = width/ncell
            dimy=width+d_x
            delta_x=width/ncell
            rho=1.12
            dom = DomainRectFluid(
                dimx=dimx,
                dimy=dimy,
                delta_x=delta_x,
                nu_=nu_,
                rho=rho)

            dom.switch_bc_ymax_wall_noslip()
            dom.switch_bc_ymin_wall_noslip()
            dom.fields["vel_u"] += vel
            dom.set_source_terms({
                "force_x": 8. * nu_ / width**2. * np.ones(dom.shape) * vel,
                "force_y": np.zeros(dom.shape),
                "scal": np.zeros(dom.shape)
            })

            print(dom)
            time = 0.0
            time_step = t_end/nsave

            #solver = Lattice(dom, max_vel=3*vel)
            solver = NS_fd_2D_explicit(dom, max_vel=1.8)
            for i in range(nsave):
                time += time_step
```

(continues on next page)

(continued from previous page)

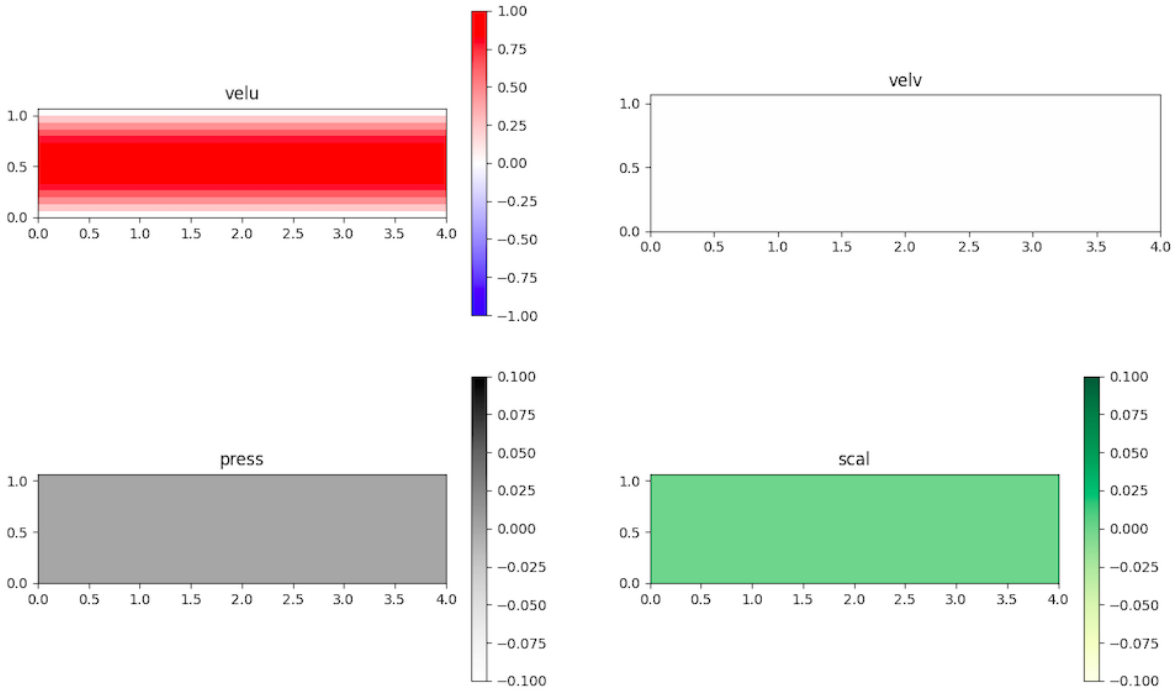
```
                solver.iteration(time, time_step)
                print("\n\n==============================")
                print(f"\n  Iteration {i+1}/{nsave}, Time :, {time}s")
                print(f"  Reynolds : {dom.reynolds(width)}")
                print(dom.fields_stats())
                dom.dump_paraview(time=time)
                dom.dump_global(time=time)

            y = np.linspace(0,1.07,num=1000)
            u=np.zeros(len(y))
            u_bulk=0.66
            Umax=(3/2)*u_bulk

            for i in range(len(y)):
                u[i]=Umax*4*(y[i]/dimy)*(1-(y[i]/dimy))
            plt.plot(y,u,color="red",marker="", label= 'theoretical curve')

            dom.show_profile_y()
            solver.plot_monitor()
            dom.show_fields()
            dom.show_flow()
            dom.show_debit_over_x()

            print('Normal end of execution.')


        if __name__ == "__main__":
            cfd_poiseuille(10)
```

3.3.2 The flow output :



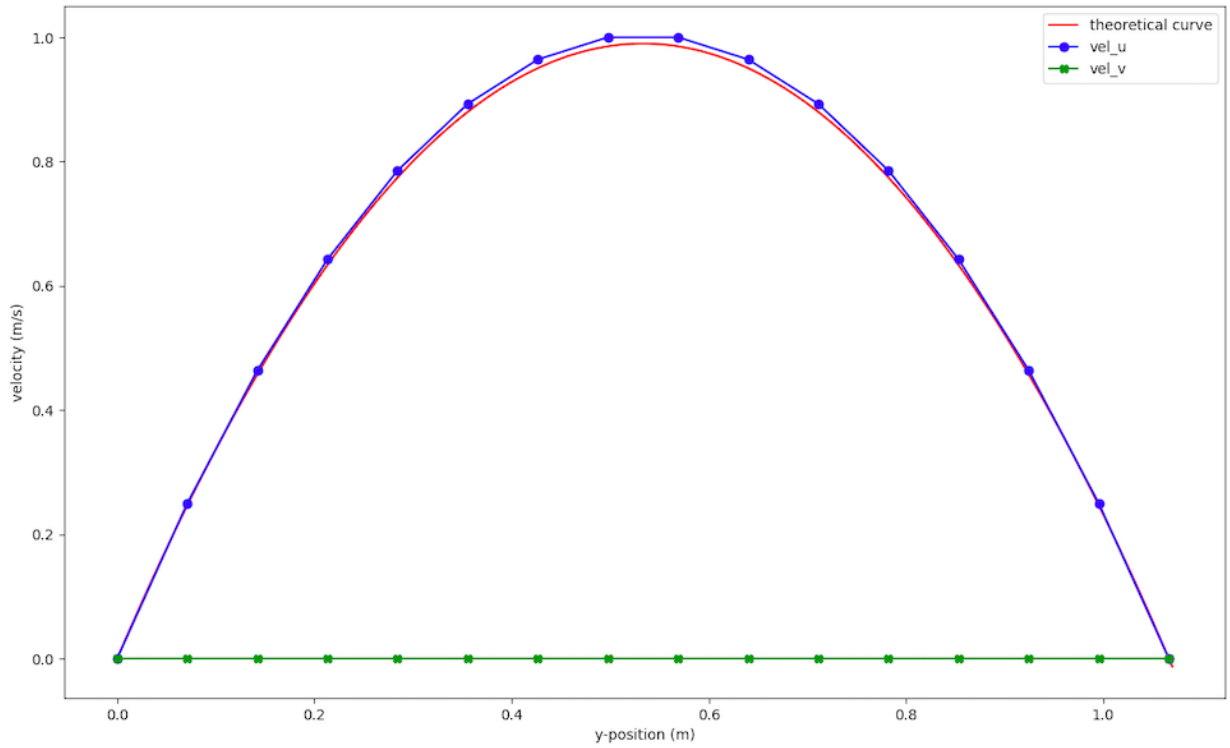
poiseuilleflow

3.3.3 The fields :



poiseuillefields

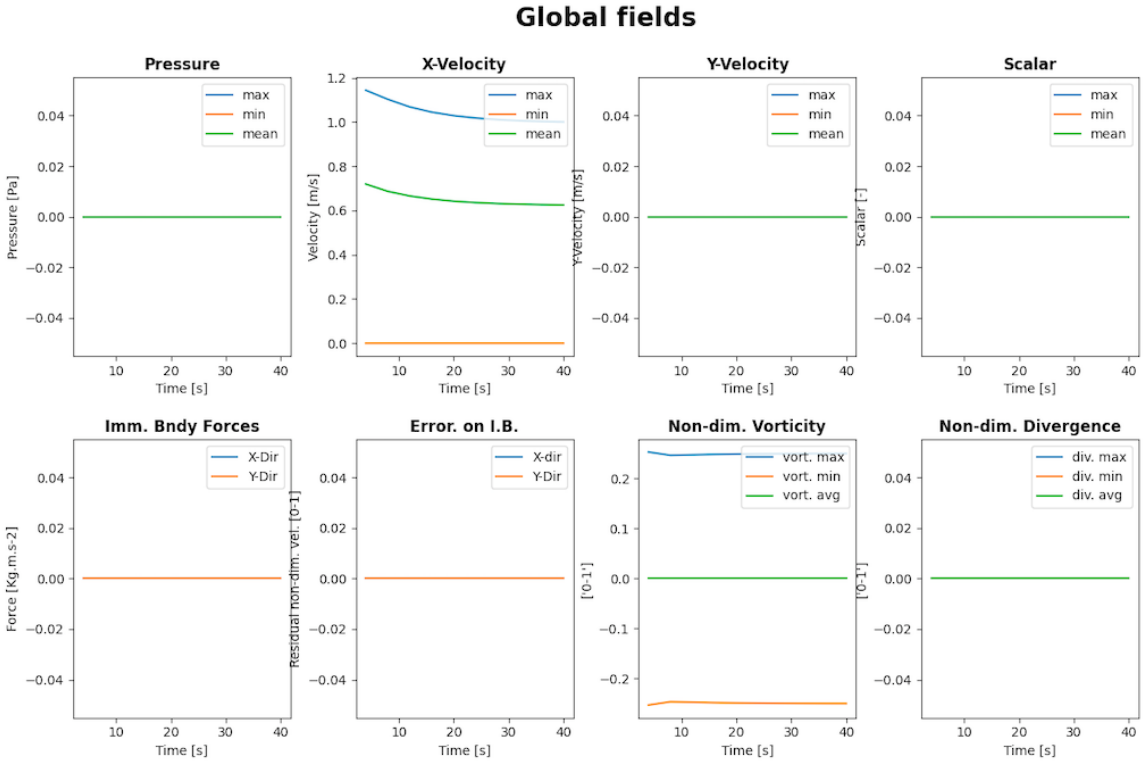
3.3.4 The flow velocity profile :



poiseuilleprofile

‘

3.3.5 Global parameters:



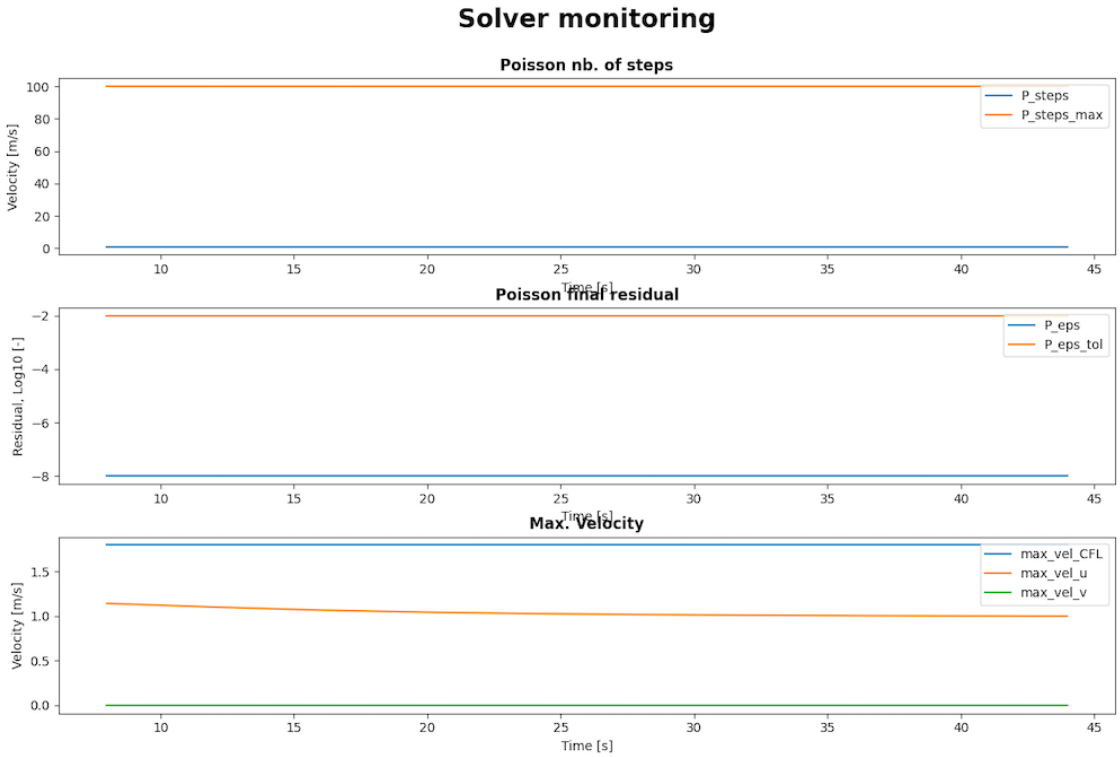
poiseuilleglob

We added a **volumic force** to simulate the **pressure gradient**, therefore, the solution correspond to the analytic solution:

Pmax = Pmin = Pmean = 0

You can obtain the mean velocity from the X-Velocity graph.

3.3.6 Solver parameters:



poiseuillesolver

max_vel is a parameter of the solver, one can study the convergence of the simulation thanks to the residual of Poisson.

For instance, using **max_vel=1.8** instead of **max_vel=2.0** improves the residual:

```
solver = NS_fd_2D_explicit(dom, max_vel=1.8)
```

3.4 To go further: how to modify the parameters to get Re = 19

1. Change the value of the viscosity to obtain the correct value of the Reynolds number by keeping the default parameters

```
dom = DomainRectFluid(nu_=0.0000032105, dimx=0.82, dimy=0.61, delta_x=0.02)

vel = 0.0001
```

2. Put the value of the initial velocity that you want and adapt the viscosity to have the right Reynolds number

```
dom = DomainRectFluid(nu_=0.048158, dimx=0.82, dimy=0.61, delta_x=0.02)

vel = 1.5
```

3. Modify the dimensions of the problem

```
dom = DomainRectFluid(nu_=0.000015789,dimx=0.006, dimy=0.0002,delta_x=0.00001)

vel=1.5
```

Pay attention to the value of the number of convective times : Lx/U_0=t_end

CHAPTER 4

Lid Driven Cavity Flow

4.1 Configuration

4.1.1 1. We want to have a lid driven cavity flow with (according to the QPF) :

Small Reynolds Number

Re = 0.000061

```
dom = DomainRectFluid(nu_=1.0, dimx=0.82, dimy=0.61, delta_x=0.02)
vel = 0.0001

cfd_lid_driven(10, 0.30)
```

4.1.2 2. Works also with another configuration to get another Reynolds number :

High Reynolds Number

Re = 300

```
size = 1.0
dom = DomainRectFluid(dimx=size, dimy=size, delta_x=size / 60, rho=1.0, nu_=0.01)

vel = 3.0
```

Because we are at high Reynolds number :

- we need to have an important t_end
- Moreover to avoid the following error:

  ```
  RuntimeError: too many sub iterations 10101
  ```

  one can put more saving moments to have weaker iterations.

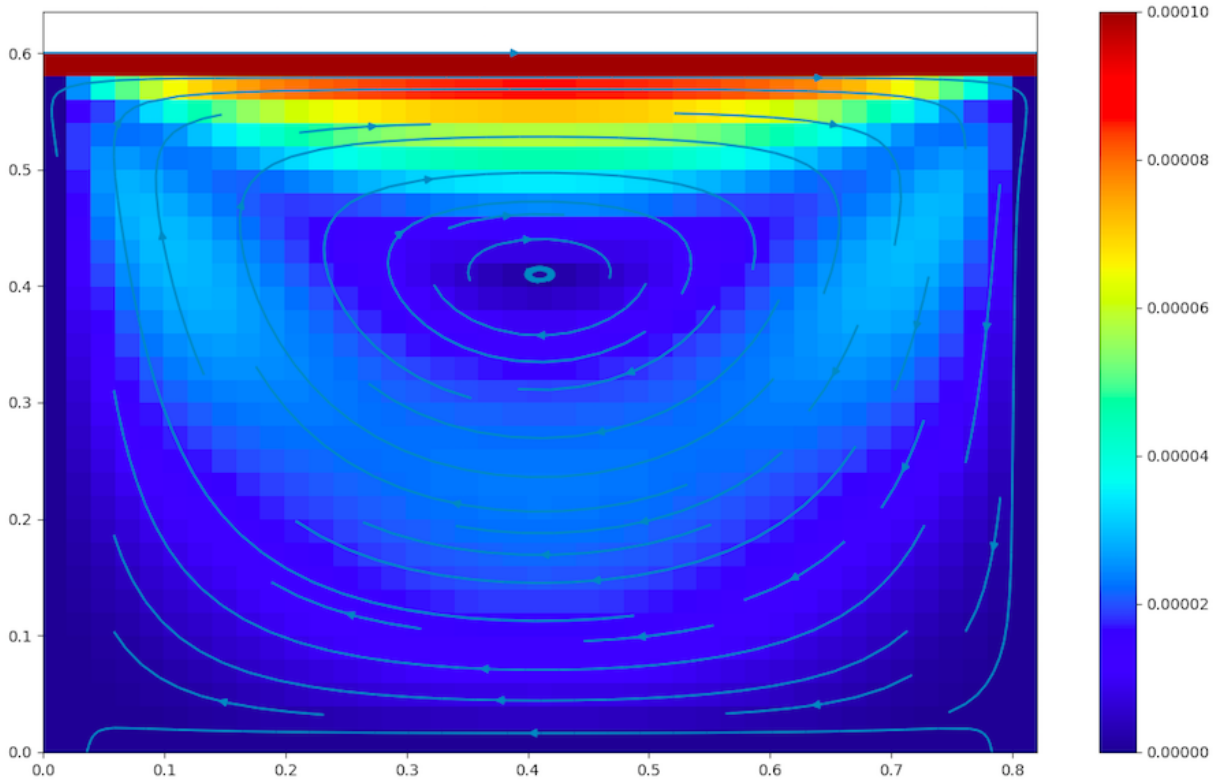
4.2 Reference case

4.2.1 the script

```
        """Example on how to solve a Lid driven cavity problem with the navier stokes␣
↪solver"""

        from barbatruc.fluid_domain import DomainRectFluid
        from barbatruc.fd_ns_2d import NS_fd_2D_explicit

        # from barbatruc.lattice import Lattice

        __all__ = ["cfd_lid_driven"]


        # pylint: disable=duplicate-code
        def cfd_lid_driven(nsave, tchar):
            """Startup computation
            solve a lid_driven_cavity  problem
            """
            size = 1.0
            dom = DomainRectFluid(dimx=size, dimy=size, delta_x=size / 60, rho=1.0,␣
↪nu_=0.01)

            vel = 3.0
            t_end = tchar * size / vel
            dom.switch_bc_xmin_wall_noslip()
            dom.switch_bc_xmax_wall_noslip()
            dom.switch_bc_ymax_moving_wall(vel_u=vel)
            dom.switch_bc_ymin_wall_noslip()

            time = 0.0
            time_step = t_end / nsave

            # solver = Lattice(dom, max_vel=2*vel)
            solver = NS_fd_2D_explicit(
                dom, obs_ib_factor=0.01, press_maxsteps=200, press_tol=5.0e-3, max_
↪vel=4.0
            )
            for i in range(nsave):
                time += time_step
                solver.iteration(time, time_step)
                print("\n\n================================")
                print(f"\n  Iteration {i+1}/{nsave}, Time :, {time}s")
                print(f"  Reynolds : {dom.reynolds(size)}")
                print(dom.fields_stats())
                dom.dump_paraview(time=time)
                dom.dump_global(time=time)
            dom.show_fields()
            dom.show_flow()
            dom.show_profile_y(xtgt=0.41)
            print("Normal end of execution.")


        if __name__ == "__main__":
            cfd_lid_driven(10, 10)
```

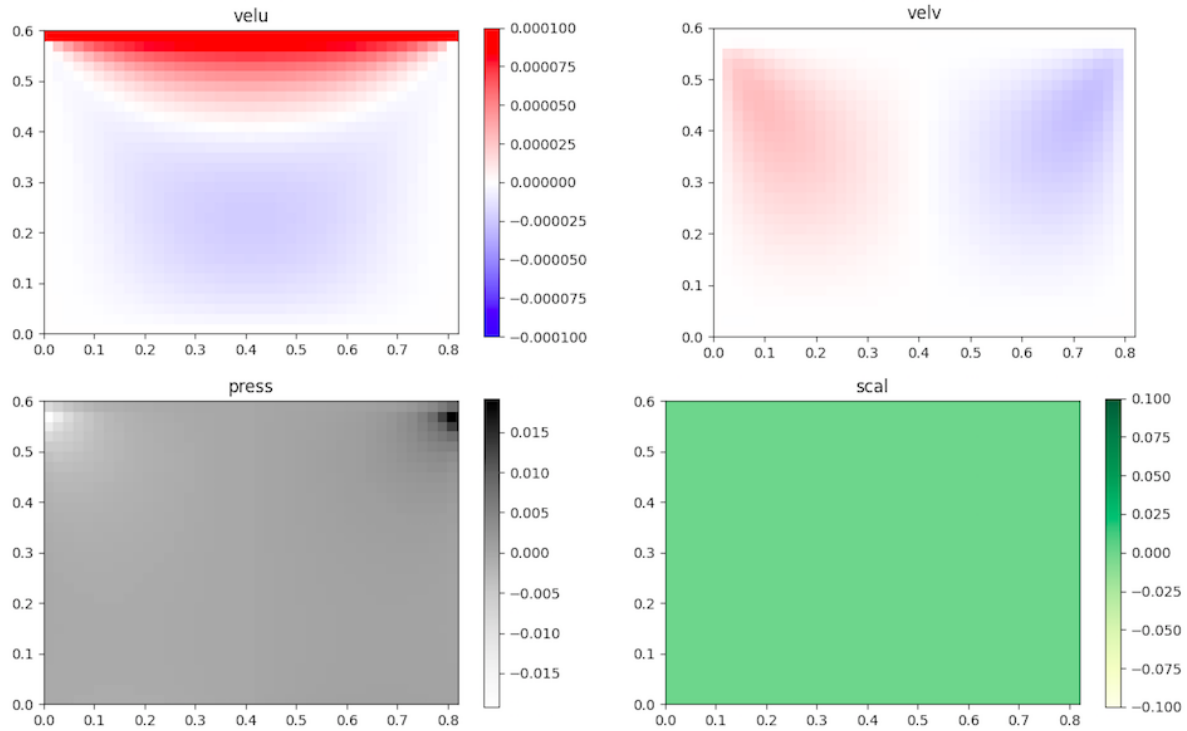
4.3 For Re = 0.000061

4.3.1 The flow output :



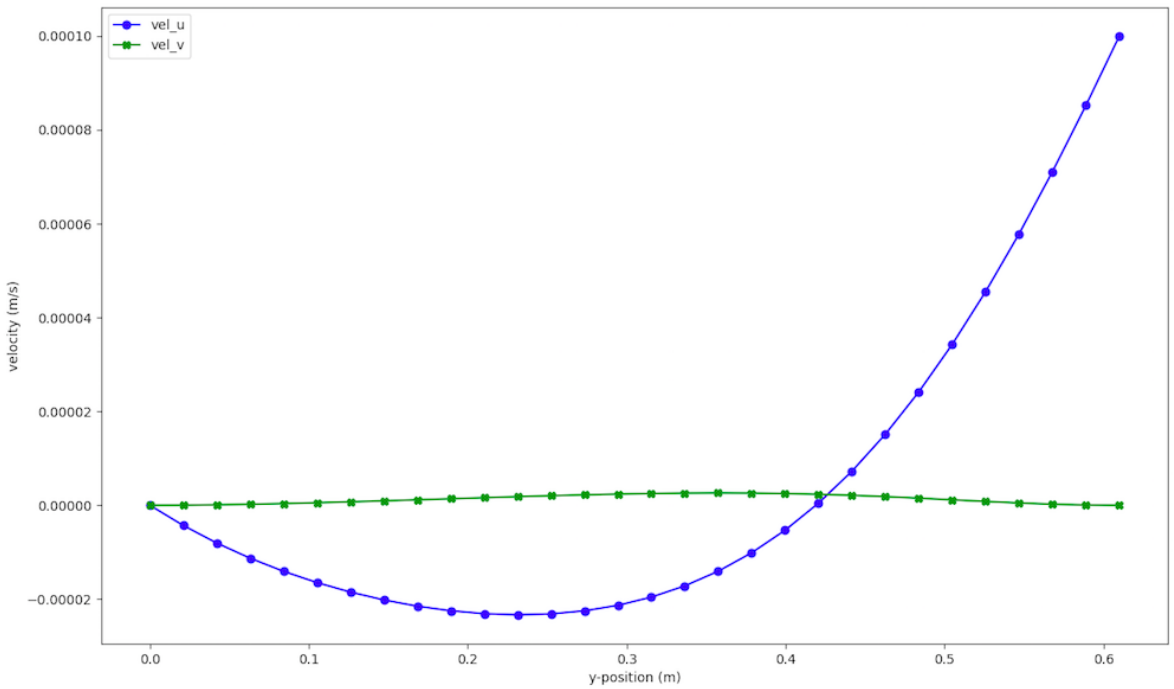
cavityflow

4.3.2 The fields :



cavityfields

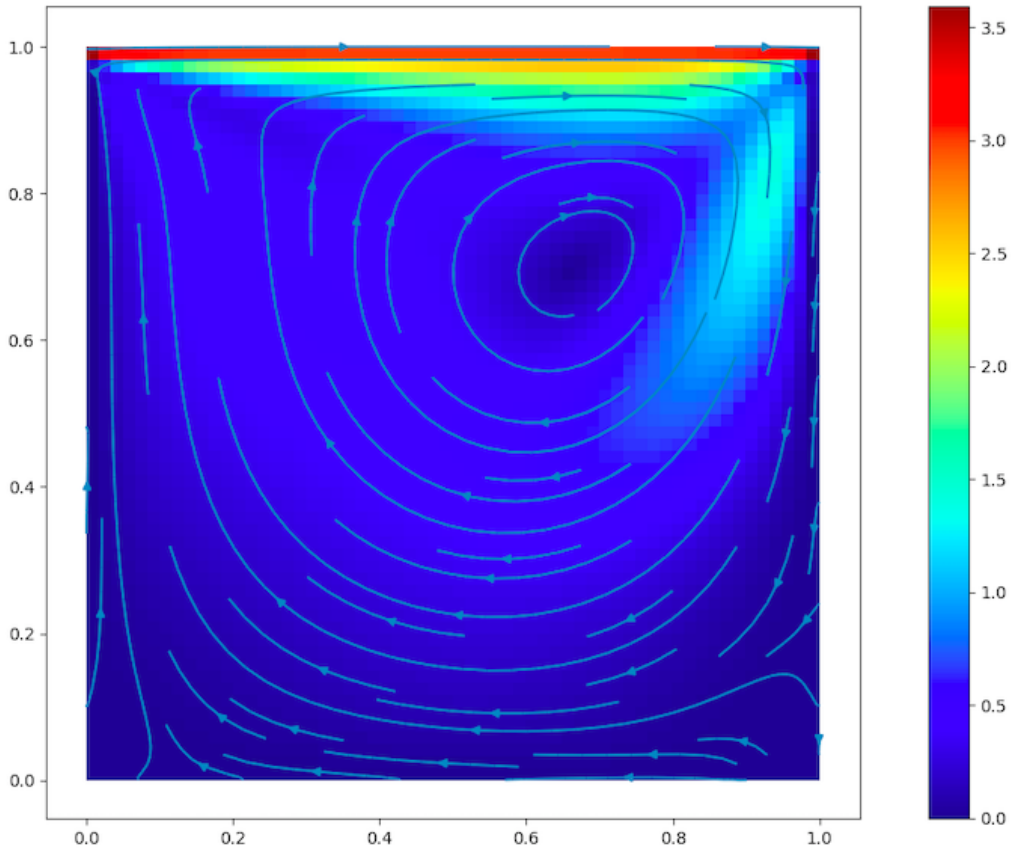
4.3.3 The velocity profile :



cavityprofile

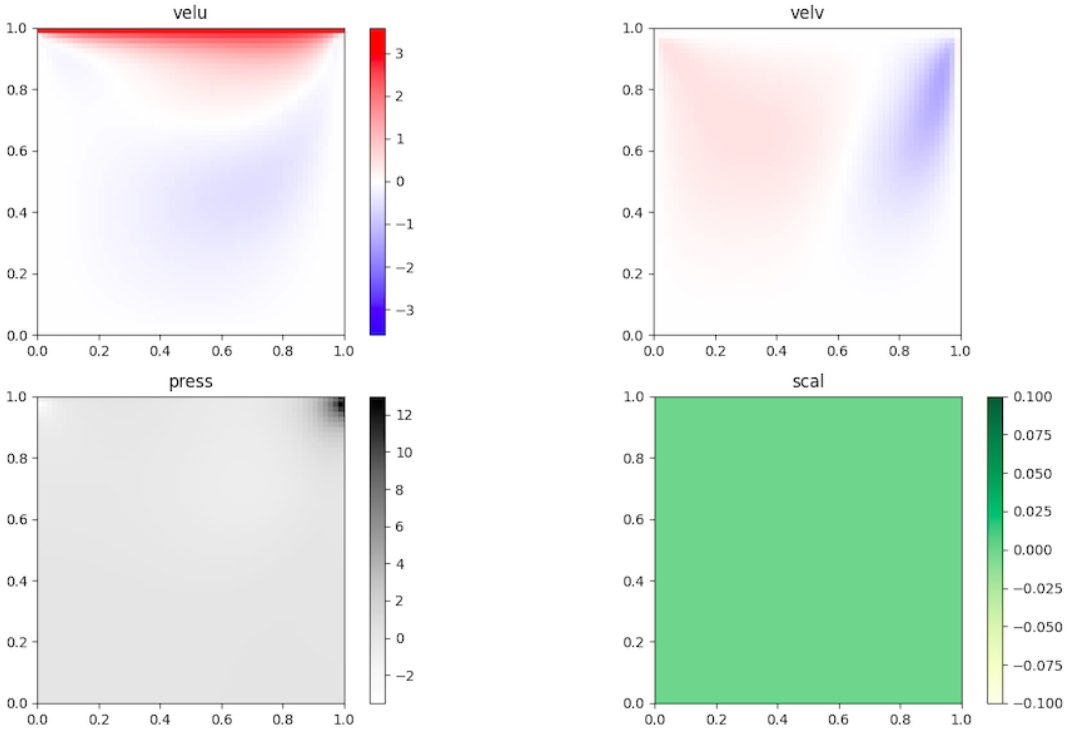
4.4 For Re = 300

4.4.1 The flow output :



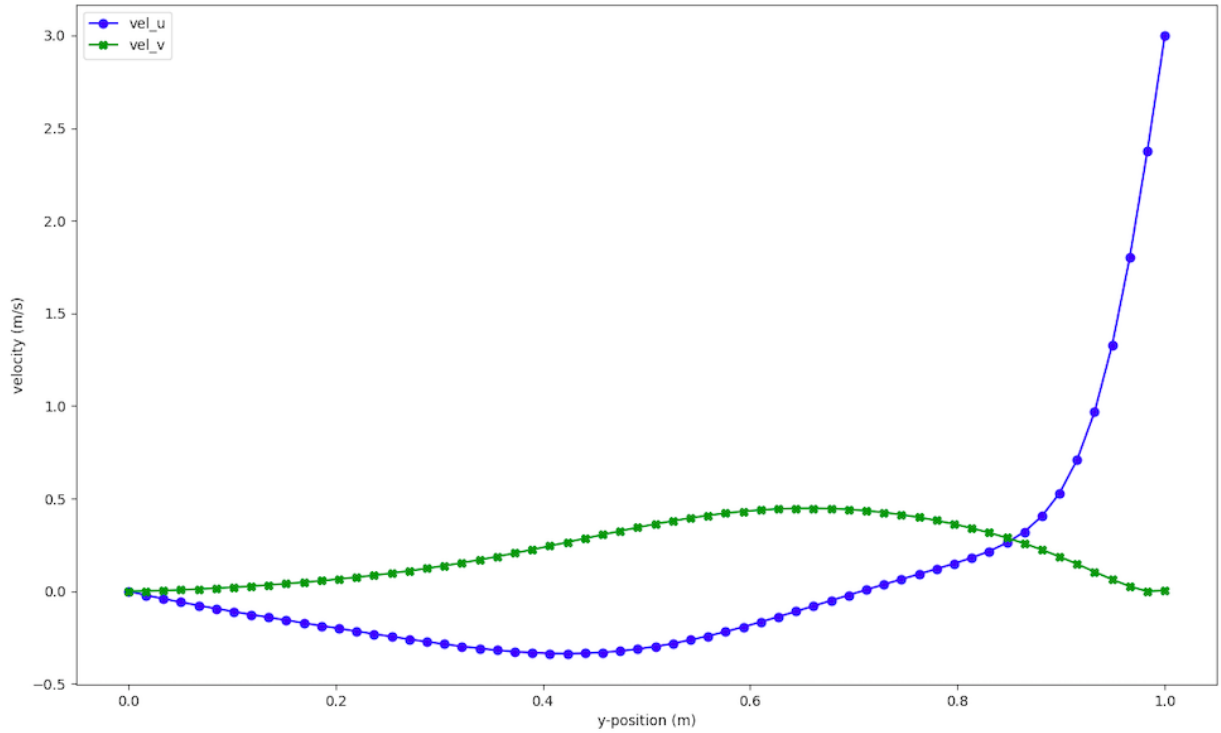
cavity300flow

4.4.2 The fields :



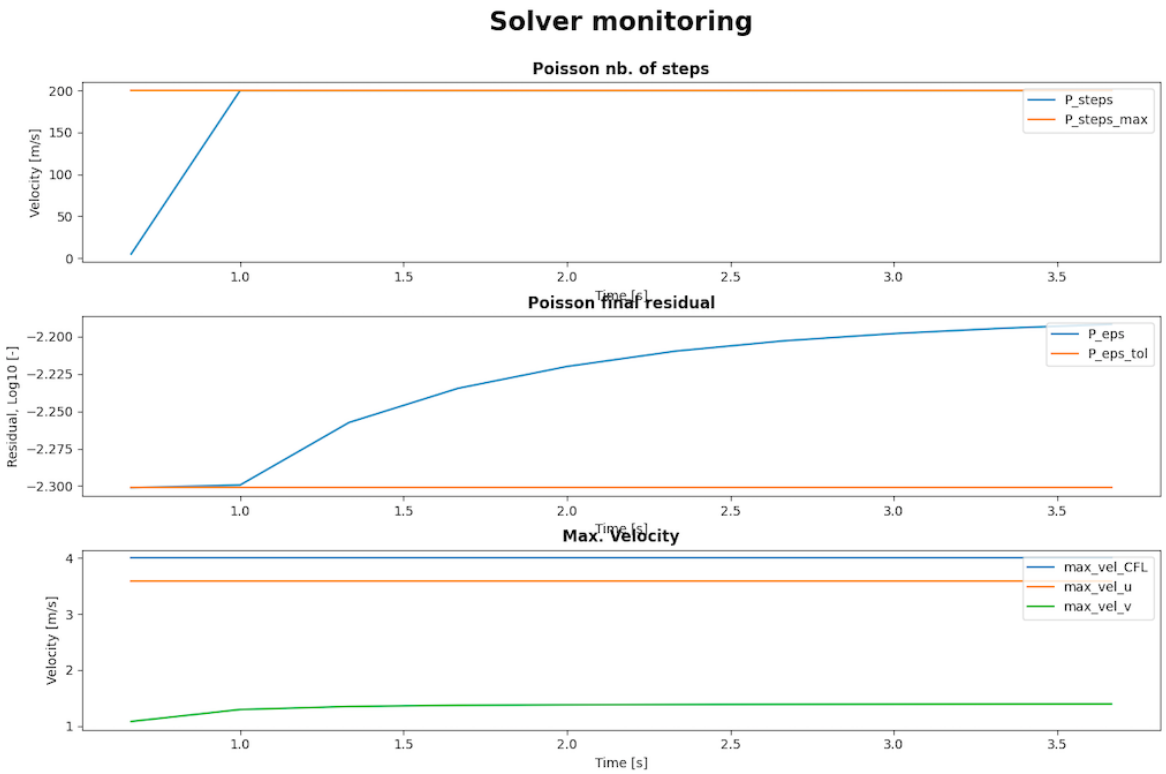
cavityfields300

4.4.3 The velocity profile :



cavityeprofile300

4.4.4 Solver parameters:



cavitysolver

Before modifications:

```
  solver = NS_fd_2D_explicit(dom, obs_ib_factor=0.9, max_vel=2*vel)
```

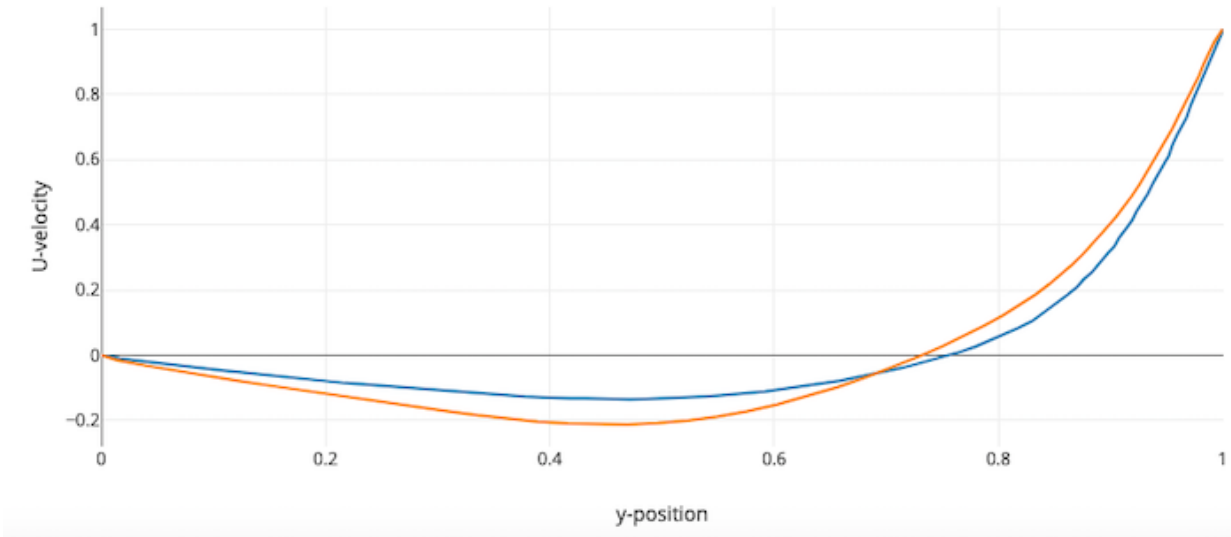
To make the **residual** better we used a small delta_x and we manipulated the different parameters of the solver until having satisfactory results:

```
          solver = NS_fd_2D_explicit(
              dom, obs_ib_factor=0.01, press_maxsteps=200, press_tol=5.0e-3, max_
↪vel=4.0)
```

cavityVORTICITY

There is a high *vorticity* at the top corners, because they are *singularities* and the solver has difficulties to converge.

4.5 Comparison with the documentation for Re = 100



comparison

The curves are not exactly the same because the barbatruc solver is light weighted.

The blue curve is from the simulation and the orange curve is from the figure 5 (with AR=1) of the **CFD SIMULATIONS OF LID DRIVEN CAVITY FLOW AT MODERATE REYNOLDS NUMBER**, *by Reyad Omari,*

Department of Mathematics, Al-Balqa Applied University, European Scientific Journal May 2013 edition vol.9, No.15 ISSN: 1857 – 7881 (Print) e - ISSN 1857- 7431 22

To have a better graph, one can try to:

- refine the mesh
- modify the pressure solver

Indeed, there are two main problems:

- the cavity has singularities
- **there might be an issue of mass conservation, indeed the solver of Poisson did not converge enough and therefore mass conservation is not completely achieved**

CHAPTER 5

Cylinder Flow - Von Karman Street

5.1 Configuration

We want to have a flow around a cylinder with :

Re = 100

```
lenght = 1.0
vel = 1.0
diameter = 0.05
t_end = 4.0 * lenght / vel
dom = DomainRectFluid(dimx=lenght, dimy=0.5 * lenght, delta_x=0.005, nu_=0.0005)
```

5.2 Reference case

5.2.1 the script

```
                        """Example on how to solve a Karmann Street problem with the␣
↪navier stokes solver"""

        from barbatruc.fluid_domain import DomainRectFluid
        from barbatruc.fd_ns_2d import NS_fd_2D_explicit

        # from barbatruc.lattice import Lattice

        __all__ = ["cfd_cylinder"]


        # pylint: disable=duplicate-code
        def cfd_cylinder(nsave):
            """Startup computation
```

(continues on next page)

(continued from previous page)

```
            solve a cylinder obstacle problem
            """
            lenght = 1.0
            vel = 1.0
            diameter = 0.05
            t_end = 4.0 * lenght / vel
            dom = DomainRectFluid(dimx=lenght, dimy=0.5 * lenght, delta_x=0.005, nu_
↪=0.0005)
            dom.add_obstacle_circle(x_c=0.2 * lenght, radius=0.5 * diameter)
            dom.switch_bc_xmin_inlet(vel_u=vel)
            dom.switch_bc_xmax_outlet()
            dom.fields["vel_u"] += vel
            dom.fields["vel_v"] += 0.1 * vel

            print(dom)

            time = 0.0
            time_step = t_end / nsave

            # solver_lbm = Lattice(dom, max_vel=2*vel)
            solver = NS_fd_2D_explicit(
                dom,
                max_vel=2.0 * vel,
                obs_ib_factor=0.4,
                obs_ib_integral=False,
                # obs_ib_factor=0.001,
                press_maxsteps=200,
                damp_outlet=True,
            )

            for i in range(nsave):
                solver.iteration(time, time_step)
                time += time_step
                print("\n\n================================")
                print(f"\n  Iteration {i+1}/{nsave}, Time :, {time}s")
                print(f"  Reynolds : {dom.reynolds(diameter)}")
                print(dom.fields_stats())
                dom.dump_paraview(time=time)
                dom.dump_global(time=time)
            dom.show_fields()
            dom.show_flow()
            dom.show_profile_y(xtgt=0.5)

            print("Normal end of execution.")


        if __name__ == "__main__":
            cfd_cylinder(200)
```

```python

### The flow output :
![karmancylinderflow](cylinder/flow_cyl_100.png)

### The fields :
![karmancylinderfields](cylinder/fields_cyl_100.png)

### The velocity profile :
```

(continues on next page)

(continued from previous page)

```
![karmancylinderprofile](cylinder/profile_cyl_100.png)


### Global parameters:
![global](cylinder/global_cylinder.png)

### Drag force:
![frag](cylinder/f_x.png)

Thanks to this graph, we got the experimental drag coefficient : **C_x = 1.43**. The␣
↪theoretical one is **C_x=1.3**

The drag Force is about **0.04 N/m**.

### Lift force:
![lift](cylinder/f_y.png)

Thanks to this graph, we got the vortex shedding frequency : **f = 2.5773 Hz**, which␣
↪leads to a Strouhal number: **St = 0.128865**, the theoretical one is **St=0.16**

### Solver parameters:
![global](cylinder/solver_cylinder.png)
```

CHAPTER 6

Navier-Stokes solver documentation

Here is a verbatim of the Lorena Barba 12 steps to Naviers Stokes computation, an excellent step-by-step course oriented on scientific computing. The solver is incompressible and deals with the Pressure Poisson equation using a martix-free, pseudo time integration.

6.1 Governing equations:

Here are our Navier-Stokes equations:

```
du      du      du    - 1 dp       ( d2 u    d2 u )
-- + u -- + v -- = --- -- + nu ( ---- + ---- )
dt      dx      dy    rho dx       ( dx2     dy2  )


dv      dv      dv    - 1 dp       ( d2 v    d2 v )
-- + u -- + v -- = --- -- + nu ( ---- + ---- )
dt      dx      dy    rho dy       ( dx2     dy2  )

d2 p    d2 p           ( du du      du dv    dv dv )
---- + ---- = - rho ( -- -- + 2 -- -- + -- -- )
dx2     dy2            ( dx dx      dy dx    dy dy )
```

With patience and care, we write the discretized form of the equations.

The u-momentum equation:

```
u(n+1,i,j) - u(n,i,j)
---------------------
          dt

            u(n,i,j) - u(n,i-1,j)              u(n,i,j) - u(n,i,j-1)
+ u(n,i,j) --------------------- + v(n,i,j) ---------------------
                    dx                                 dy
```

(continues on next page)

(continued from previous page)

```
   -1 p(n,i+1,j) - p(n,i-1,j)
= --- ------------------------
  rho           2 dx

      ( u(n,i+1,j) - 2 u(n,i,j) + u(n,i-1,j)    u(n,i,j+1) - 2 u(n,i,j) + u(n,i,j-1) )
+ nu  ( ------------------------------------ + ------------------------------------ )
      (             dx^2                                   dy^2                      )

+ f(i,j)
```

The v-momentum equation:

```
v(n+1,i,j) - v(n,i,j)
---------------------
          dt

           v(n,i,j) - v(n,i-1,j)             v(n,i,j) - v(n,i,j-1)
+ u(n,i,j) --------------------- + v(n,i,j) ---------------------
                   dx                                 dy

   -1 p(n,i+1,j) - p(n,i-1,j)
= --- ------------------------
  rho           2 dy

      ( v(n,i+1,j) - 2 v(n,i,j) + v(n,i-1,j)    v(n,i,j+1) - 2 v(n,i,j) + v(n,i,j-1) )
+ nu  ( ------------------------------------ + ------------------------------------ )
      (             dx^2                                   dy^2                      )
```

And the pressure equation:

```
p(n,i+1,j) - 2 p(n,i,j) + p(n,i-1,j)    p(n,i,j+1) - 2 p(n,i,j) + p(n,i,j-1)
------------------------------------ + ------------------------------------
                 dx^2                                   dy^2

      (  1 ( u(n,i+1,j) - u(n,i-1,j)    v(n,i,j+1) - v(n,i,j-1) )
= rho ( -- ------------------------- + ------------------------ )
      ( dt            2 dx                        2 dy          )

  u(n,i+1,j) - u(n,i-1,j)  u(n,i+1,j) - u(n,i-1,j)
- ------------------------  ------------------------
         2 dx                       2 dx

    u(n,i,j+1) - u(n,i,j-1)  v(n,i+1,j) - v(n,i-1,j)
- 2 ------------------------  ------------------------
            2 dy                       2 dx

  v(n,i,j+1) - v(n,i,j-1)  v(n,i,j+1) - v(n,i,j-1)
- ------------------------  ------------------------
         2 dy                       2 dy
```

6.2 Resolution of momentum:

As always, we re-arrange these equations to the form we need in the code to make the iterations proceed.

The momentum equation in the u direction:

```
u(n+1,i,j) = u(n,i,j)
  - u(n,i,j) dt/dx ( u(n,i,j) - u(n,i-1,j)
  - v(n,i,j) dt/dy ( u(n,i,j) - u(n,i,j-1)
  - dt / ( 2 rho dx ) ( p(n,i+1,j) - p(n,i-1,j) )
  + nu dt/dx^2 ( u(n,i+1,j) - 2 u(n,i,j) + u(n,i-1,j) )
  + nu dt/dy^2 ( u(n,i,j+1) - 2 u(n,i,j) + u(n,i,j-1) )
  + dt * f(i,j)
```

The momentum equation in the v direction:

```
v(n+1,i,j) = v(n,i,j)
  - u(n,i,j) dt/dx ( v(n,i,j) - v(n,i-1,j)
  - v(n,i,j) dt/dy ( v(n,i,j) - v(n,i,j-1)
  - dt / ( 2 rho dy ) ( p(n,i,j+1) - p(n,i,j-1) )
  + nu dt/dx^2 ( v(n,i+1,j) - 2 v(n,i,j) + v(n,i-1,j) )
  + nu dt/dy^2 ( v(n,i,j+1) - 2 v(n,i,j) + v(n,i,j-1) )
```

6.3 Resolution of Pressure:

And for the pressure equation, we isolate the term p(n,i,j) to iterate in pseudo-time:

```
             ( p(n,i+1,j) + p(n,i-1,j) ) dy^2 + ( p(n,i,j+1) + p(n,i,j-1) ) dx^2
p(n,i,j) = ---------------------------------------------------------------------
                                    2 ( dx^2 + dy^2 )

        rho dx^2 dy^2
  - ------------------
    2 ( dx^2 + dy^2 )

    [  1 ( u(n,i+1,j) - u(n,i-1,j)    v(n,i,j+1) - v(n,i,j-1) )
  * [ -- ( ----------------------- + ----------------------- )
    [ dt (           2 dx                       2 dy          )

    - u(n,i+1,j) - u(n,i-1,j)  u(n,i+1,j) - u(n,i-1,j)
      -----------------------  -----------------------
                2 dx                       2 dx

        u(n,i,j+1) - u(n,i,j-1)  v(n,i+1,j) - v(n,i-1,j)
    - 2 -----------------------  -----------------------
                  2 dy                       2 dx

      v(n,i,j+1) - v(n,i,j-1)  v(n,i,j+1) - v(n,i,j-1) ]
    - -----------------------  ----------------------- ]
                2 dy                       2 dy        ]
```

6.4 Present BARBATRUC implementation

The present implementation have the following differences with the final step of 12-steps-to-NS.

- The spatial resolution `dx` and `dy` are equaland constant, because we wanted to share the same data-structure as a 2DQ9 LBM solver. This simplifies equations a lot.

- We added a scalar transport equation.
- We added a "crude" immersed boundary method.
- There is a convergence criterion in the Pressure solver.

In terms of implementation, the most obvious difference is the separation of derivatives in a dedicated routine. It makes the code more readable, by removing the Finite-Difference clutter from the solver, and allows testing.

CHAPTER 7

Lattice boltzmann solver documentation

The LBM solver of Barbatruc is written following two Books:

- Mohamad, A. A. (2011). Lattice Boltzmann Method (Vol. 70). London: Springer. :
- Sukop, M. (2006). DT Thorne, Jr. Lattice Boltzmann Modeling Lattice Boltzmann Modeling.

7.1 General algorithm

The solver is using the D2Q9 Lattice and follows the classical approach depicted on the Wikipedia introdution on LBM.

7.2 Inlet boundary conditions

The default inlet BC. at x_min is using the *Bounceback* approach (p77 book Mohamad, p198 book Sukop). Two other formulations are also available , with *Non equilibrium extrapolation method* (p79 book Mohamad, p194 book Sukop) and *equilibrium* (p191 book Sukop)

7.3 Outlet boundary conditions

The default outlet BC, at x_max is using the *extrapolation* method (p79 mohamad). There is also the *density_imposed* option (p79 mohamad).

7.4 DIfferences with the NS solver

The LBM solver share the same API as the NS solver. In particular it features a cell-blocking method using the same obstacle.

However, the current LBM implmentation has been **only tested for poiseuille flow**. Use this solver with caution for the moment Take particular care about the conversion between physical quantities and Lattice non-dimensionned quantities. Moreover, it **does not transport the scalar** yet.

> Note to developers: unfortunately the axes are swapped between NS and LBM implementation. NS use X on the "axis 1" (second index of numpy array), while LBM use X on the "axis 0" (first index of numpy array). This is the origin of the following initialization

```
self.init_velocity = np.stack(
            (np.transpose(dm.fields["vel_u"]),
             np.transpose(dm.fields["vel_v"])),
            axis=0)
```

CHAPTER 8

barbatruc package

8.1 Subpackages

8.1.1 barbatruc.examples package

Submodules

barbatruc.examples.cfd_cylinder module

barbatruc.examples.cfd_lid_driven module

barbatruc.examples.cfd_poiseuille module

8.2 Submodules

8.3 barbatruc.barb_orig module

8.4 barbatruc.cli module

8.5 barbatruc.fd_ns_2d module

8.6 barbatruc.fd_operators module

8.7 barbatruc.fluid_domain module

8.8 barbatruc.lattice module

CHAPTER 9

Indices and tables

- genindex
- modindex
- search