Final project: Course NMMO403, Computer Solutions of Continuum Physics Problems, Charles University in Prague

Dominik Vach

September 26, 2017

Visualisation of viscous flows of liquid helium due to an oscillating rectangular object

This project serves as a visualisation of processes investigated by D. Duda et al. (2015) in their article Visualization of viscous and quantum flows of liquid ^{4}He due to an oscillating cylinder of rectangular cross section. In the paper, the authors compare experiments with He I and He II with a vertically moving rectangular obstacle. They found out that large-scale millimeter-sized vortices are generated in the surrounding fluid. Their experiment has shown that magnitudes describing the strength of the vortical structures is very similar for values of $Re > 10^4$ for both types of helium if the kinematic viscosity is suitably defined for He II. On the other hand, for $Re < 10^4$, the strength of the large scale vortices is smaller in He II than in He I. They suggest this difference might be an effect caused by both viscous and quantum features of the flows of He II driven by the mechanical force.

Problem setting

In this simulation the obstacle cross section is 3 mm high and 10 mm wide and oscillates with an amplitude of 5 mm which coincides exactly with the parameters used in the article. We simulate the movement with slightly higher oscillating frequency of 2 Hz. Considered Reynolds number is $Re = 10^5$ and we set viscosity as $\nu = 1/Re$. We solve the following set of equations

$$\mathbb{T}(p, v) = -p\mathbb{I} + \nu(\nabla\vec{v} + (\nabla\vec{v})^T)$$

$$\nabla \cdot \vec{v} = 0$$

$$\frac{\partial\vec{v}}{\partial t} + \vec{v} \cdot \nabla\vec{v} = div\mathbb{T}$$

on a 40mm×40mm rectangular domain with free boundary in the top and the bottom and noslip condition on the sides of the domain. The space discretization is done by the method of finite elements with twice refined mesh in the location of the rectangular obstacle. The time discretization is done by implementing theta scheme which approximates the equation

$$\frac{\partial v}{\partial t} + F(t) = 0$$

as

$$\frac{v - v0}{\Delta t} + (1 - \theta)F(t_0) + \theta F(t)$$

where t_0 denotes previous time step and t the current one. The time step is chosen as $0.005s$ and the simulation proceeds from $0.000s$ to $1.000s$ which means two complete periods of rectangular object movement were simulated. There are two stabilizations used in the numerical implementation of the solution. The first one stabilizes boundary velocity magnitude and the second one gives interior penalty in order to stabilize the solution.

Implementation

In this problem, there was a particular problem in considering moving object which was part of the boundary. In FEniCS, the platform we used for the implementation of the problem, there is a possibility to create finite elements mesh. However, in every time step the mesh differed from the previous one. Therefore, we decided to take a constant mesh but refine it enough so that small movements of the rectangular object are large in comparison with the mesh density. To ensure this, one has to change Dirichlet boundary condition on the rectangle object in every time step. In the default implementation of FEniCS class NonlinearProblem, the class takes only initial Dirichlet boundary condition and keeps it during the existence of the NonlinearProblem instance. Therefore we had to create our own class called UpgradedNonlinearProblem practically inherited from NonlinearProblem class with the only difference of adding a method which updates the Dirichlet boundary condition. The sample of the implementation code is following:

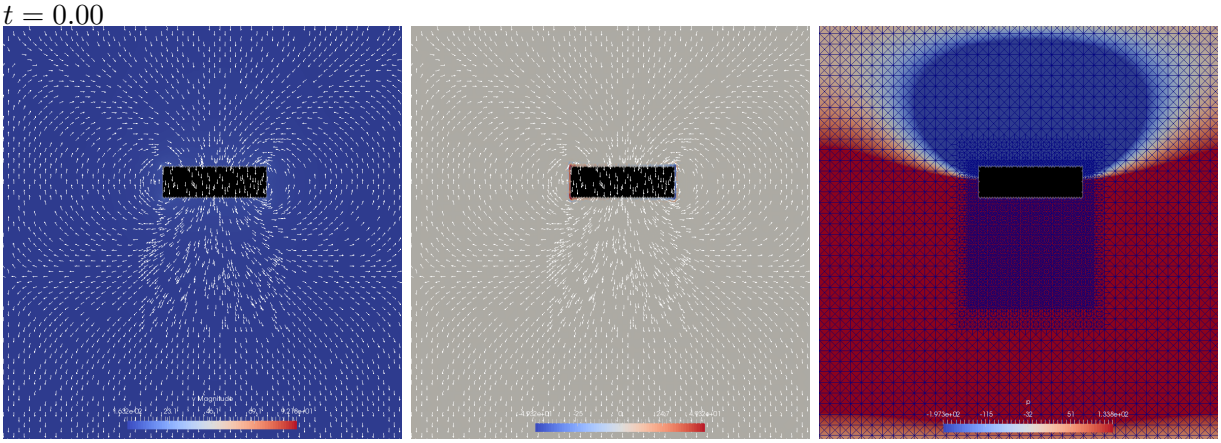
```
# create our own nonlinear problem with possibility of updated DirichletBC
class UpgradedNonlinearProblem(NonlinearProblem):
    def __init__(self, F, J, bcs):
        NonlinearProblem.__init__(self)
        self.PF = F
        self.PJ = J
        self.bcs = bcs
        self.fb = []

    def UpdateObjectDirichletBC(self, W, velocity, mask):
        self.fb = [
            DirichletBC(W.sub(0), velocity, mask, method='pointwise')
            ]

    def F(self, b, x):
        assemble(self.PF, tensor=b)
        for bc in self.bcs : bc.apply(b, x)
        for bc in self.fb : bc.apply(b, x)

    def J(self, A, x):
        assemble(self.PJ, tensor=A)
        for bc in self.bcs : bc.apply(A)
        for bc in self.fb : bc.apply(A)
```

The further implementation issue was to handle low dimension of Lagrange finite elements with adding volume bubble functions for triangular elements in order to avoid volume locking and to damp stress oscillations. This was implemented using the mixed element consisting of Lagrange elements for both velocity and pressure of order one and bubble triangle element of order three.

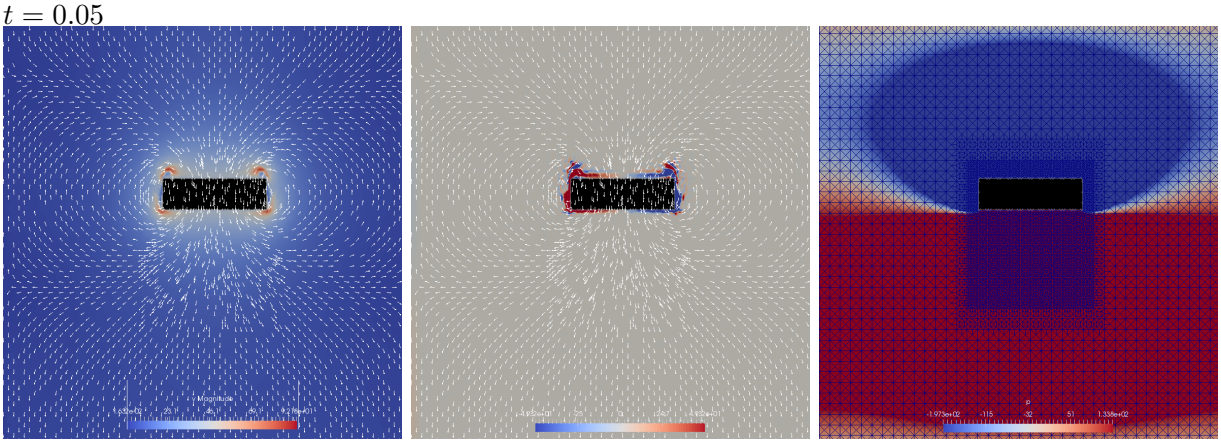
Results

The results of the simulation were visualized by Paraview program. We were particularly interested in the rotation of the velocity field, therefore we calculated also this value. In the following figures there are on each row velocity field, rotation of velocity field and pressure field, respectively. In the pressure field figures, there is also depicted the mesh discretization. I attach to this project also an animation with time step $\Delta t = 0.005$. The simulation agrees with the article on the generation of large-sized vortices in the liquid surrounding the moving rectangular object. Vortices in simulation have similar shape and magnitude as in the referred article.

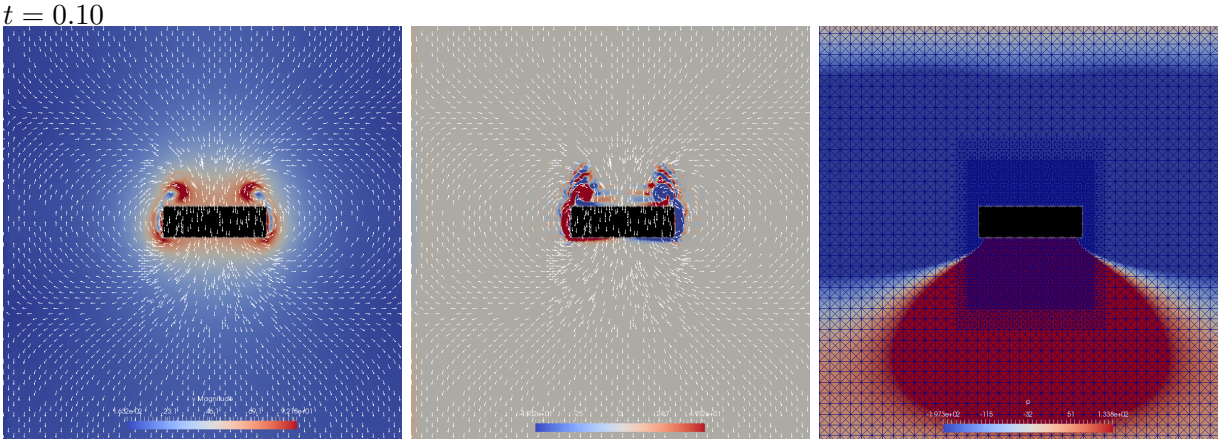
$t = 0.00$



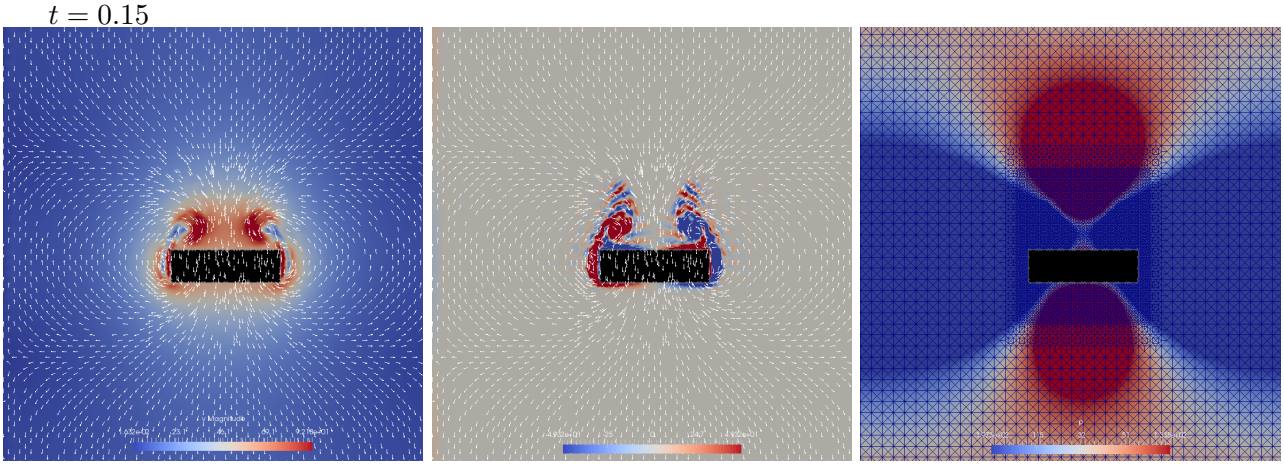
$t = 0.05$



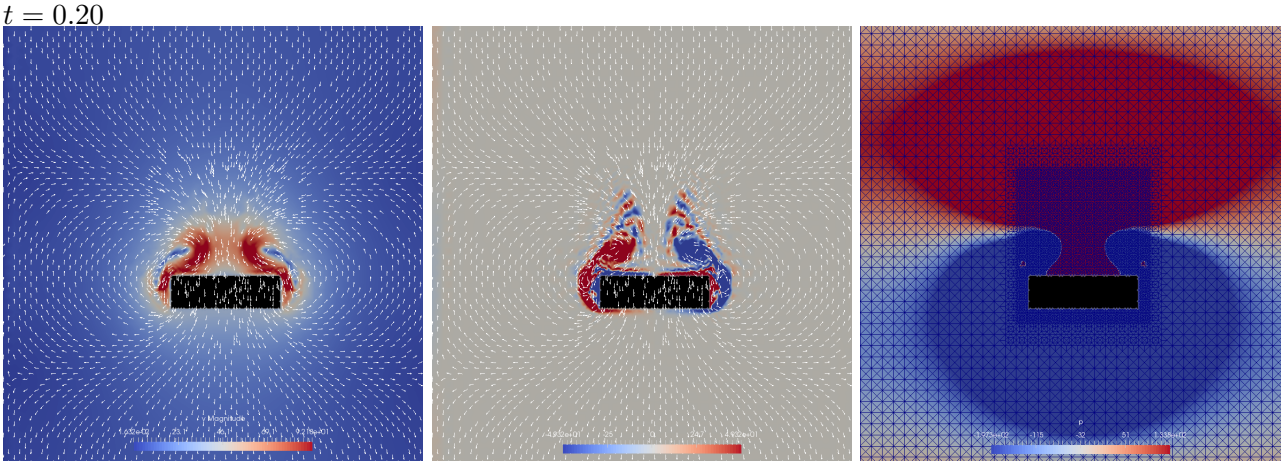
$t = 0.10$



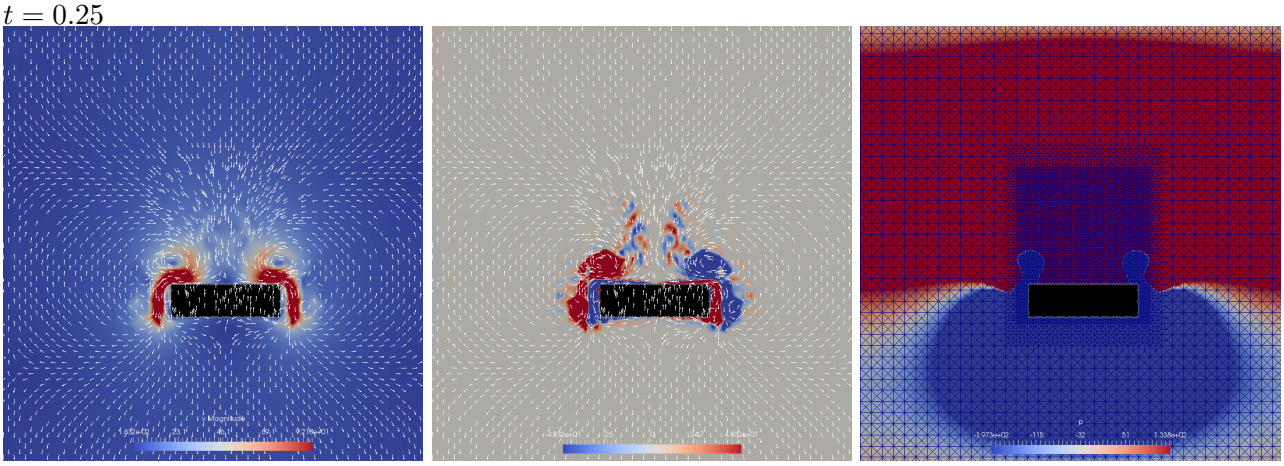
$t = 0.15$



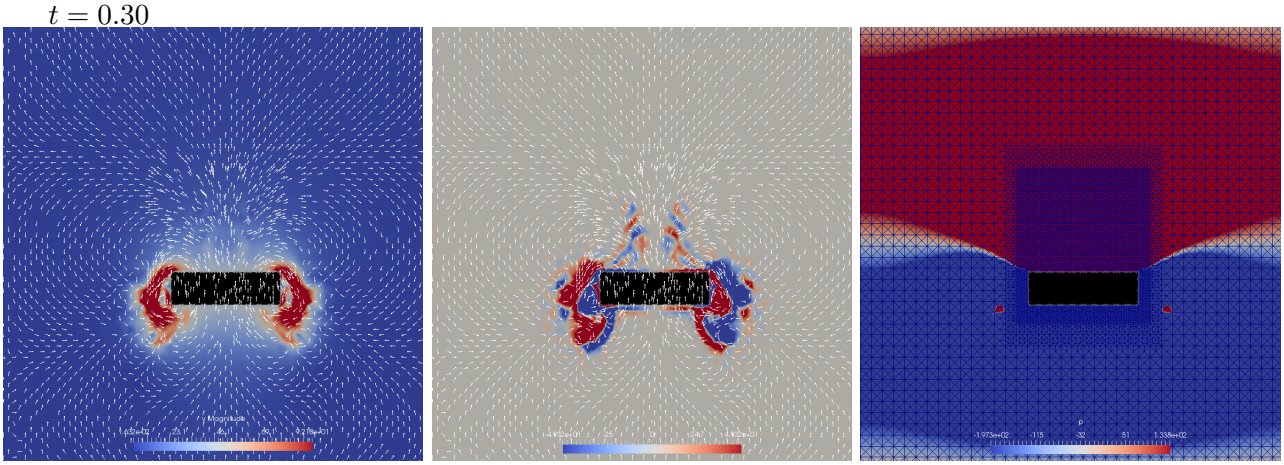
$t = 0.20$



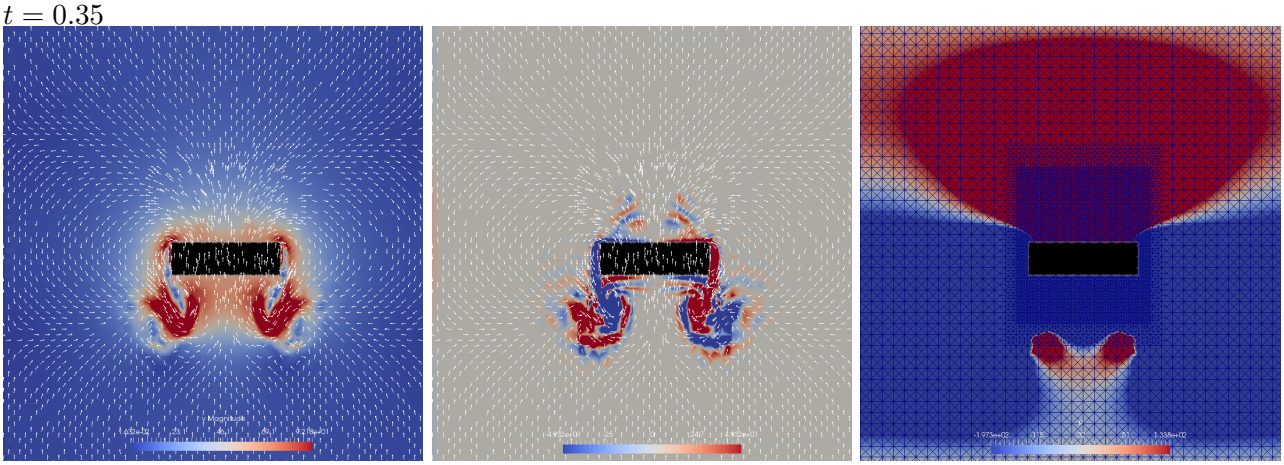
$t = 0.25$



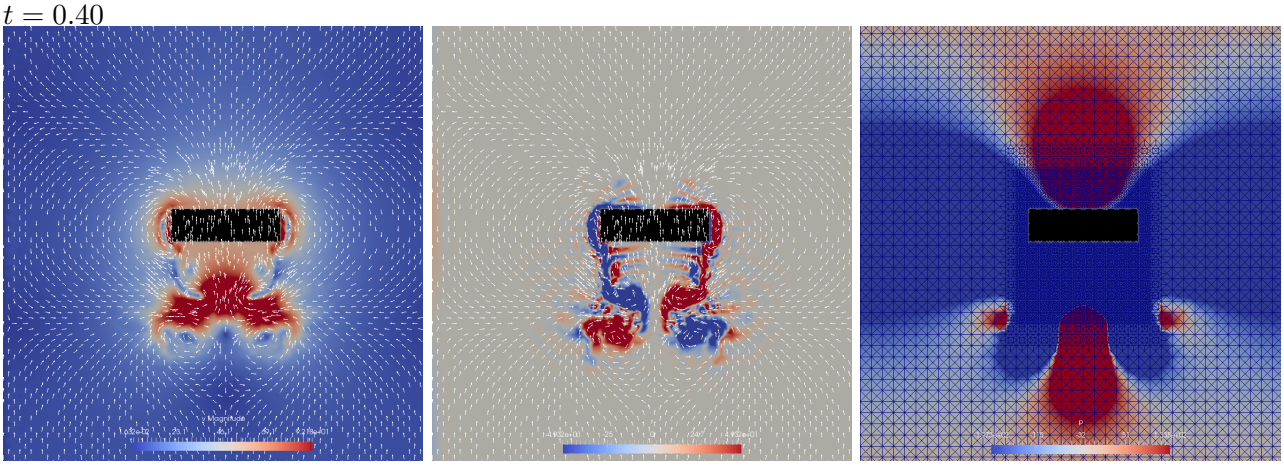
$t = 0.30$



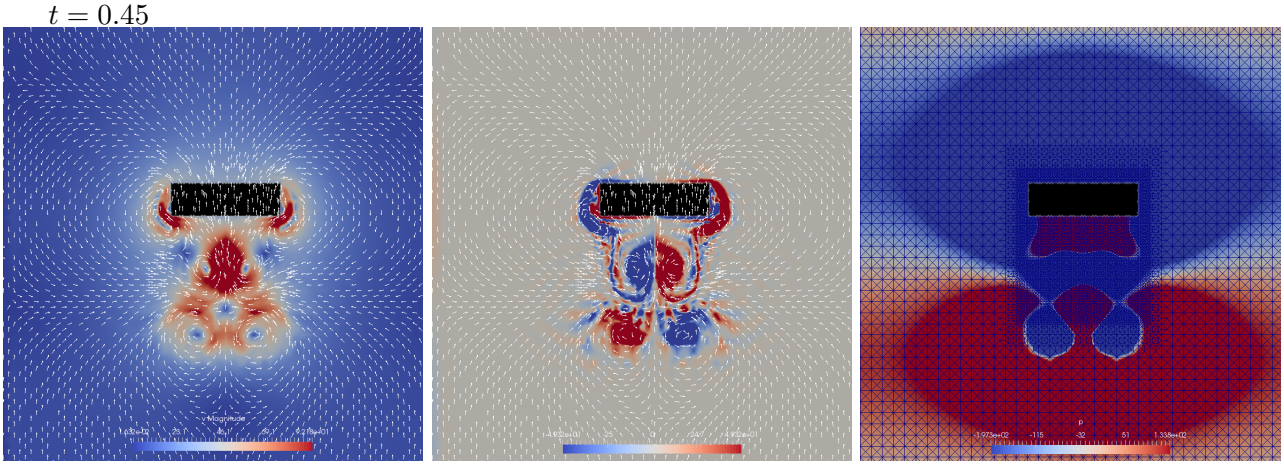
$t = 0.35$



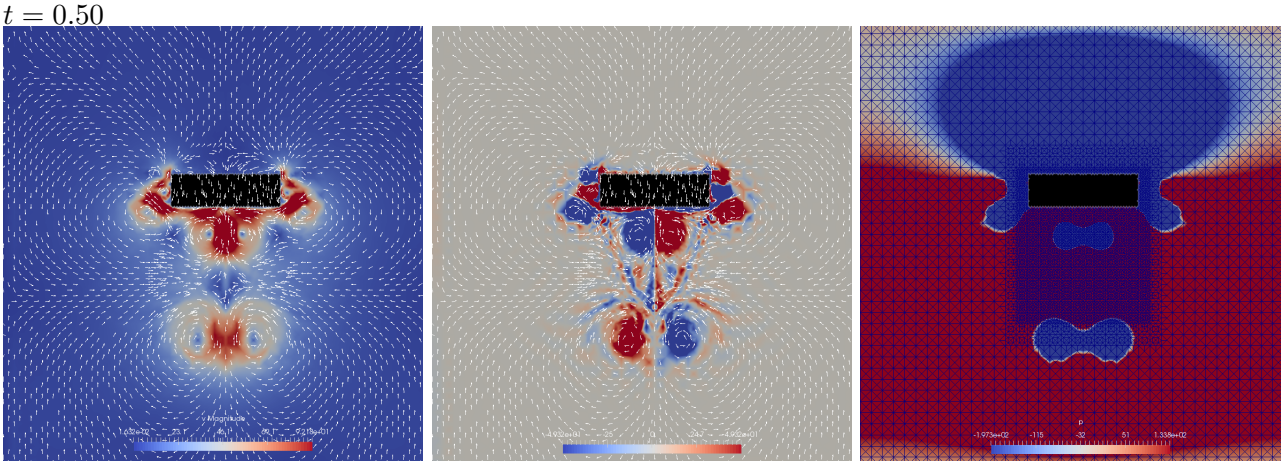
$t = 0.40$



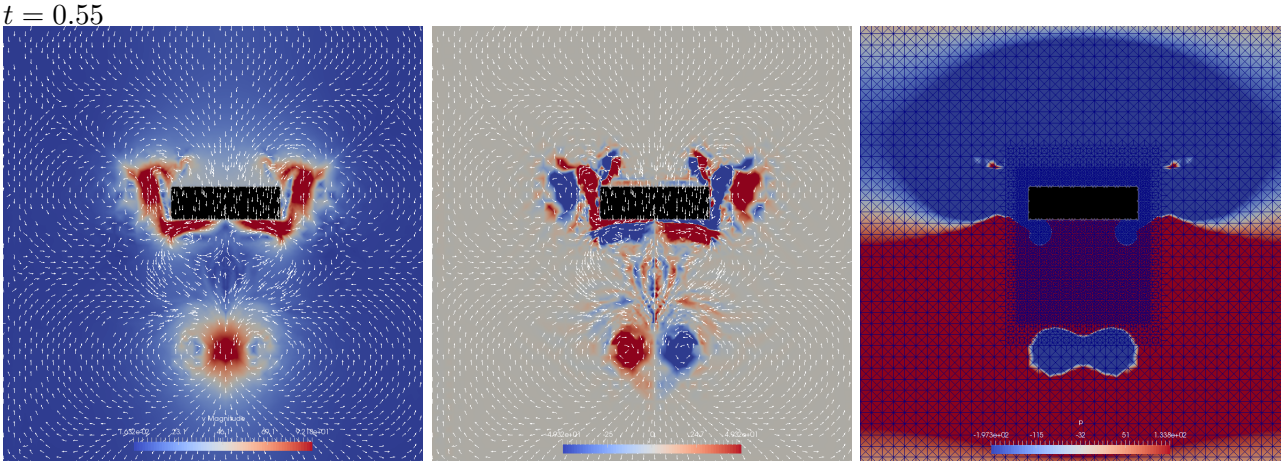
$t = 0.45$



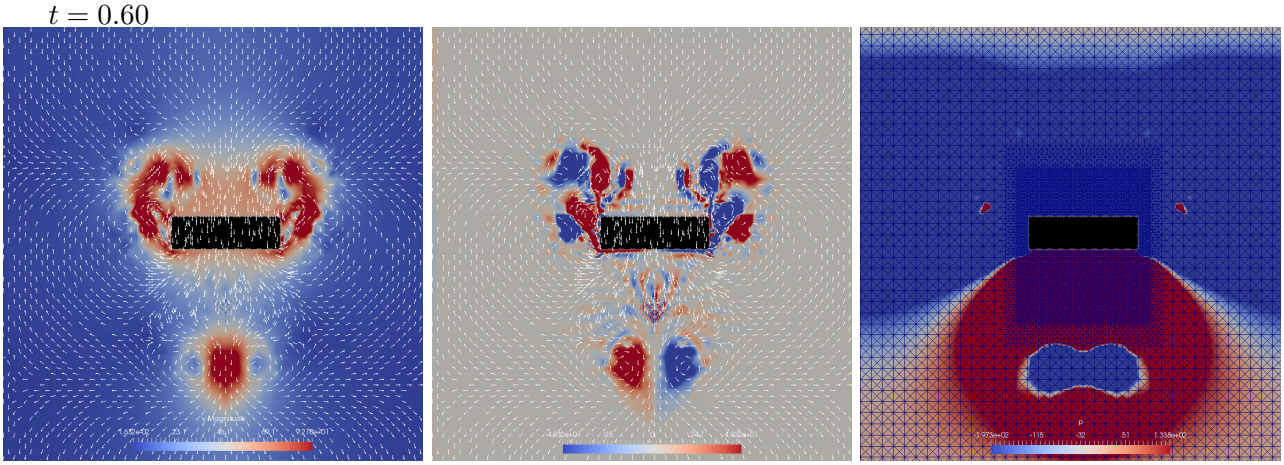
$t = 0.50$



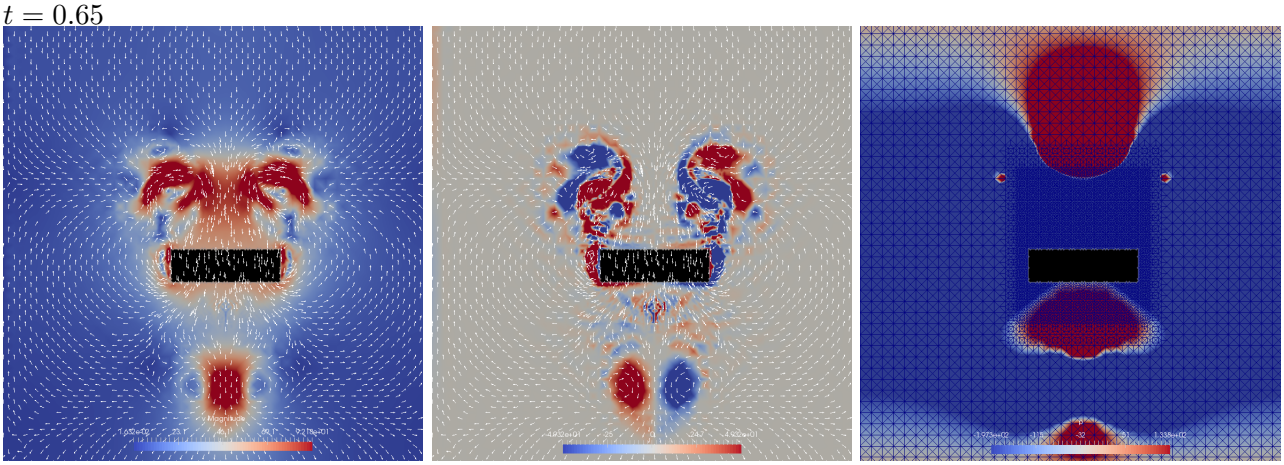
$t = 0.55$



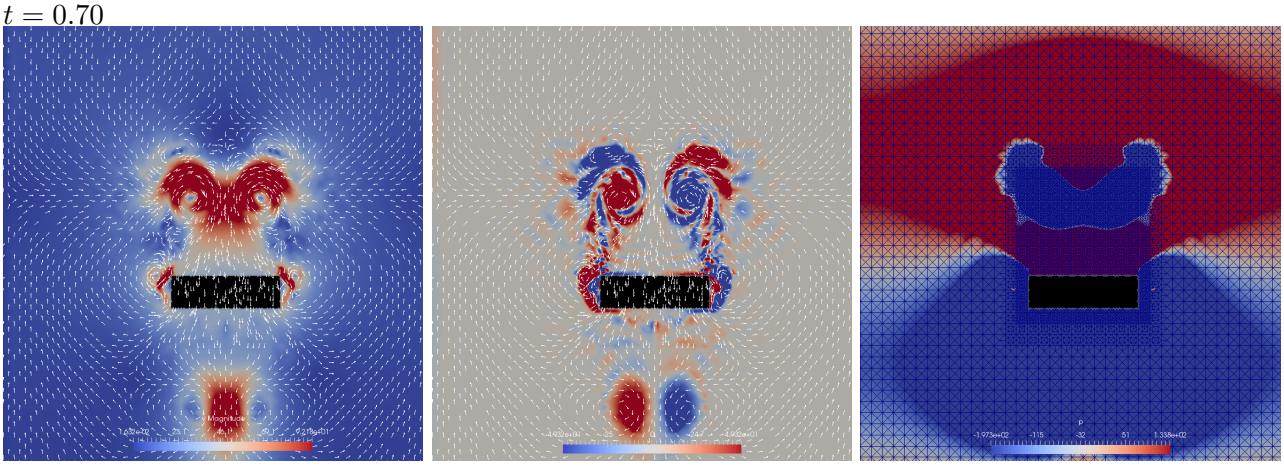
$t = 0.60$



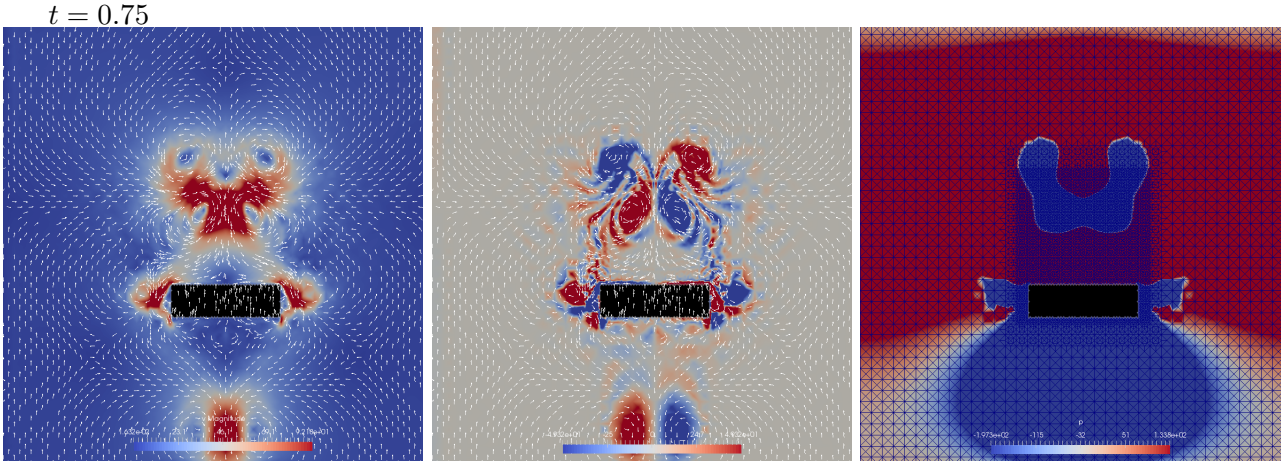
$t = 0.65$



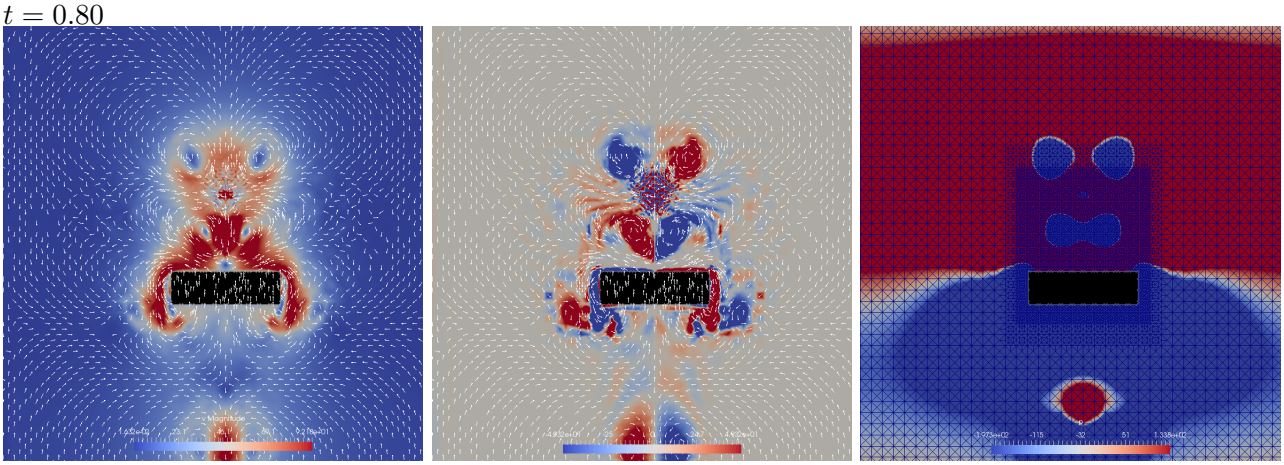
$t = 0.70$



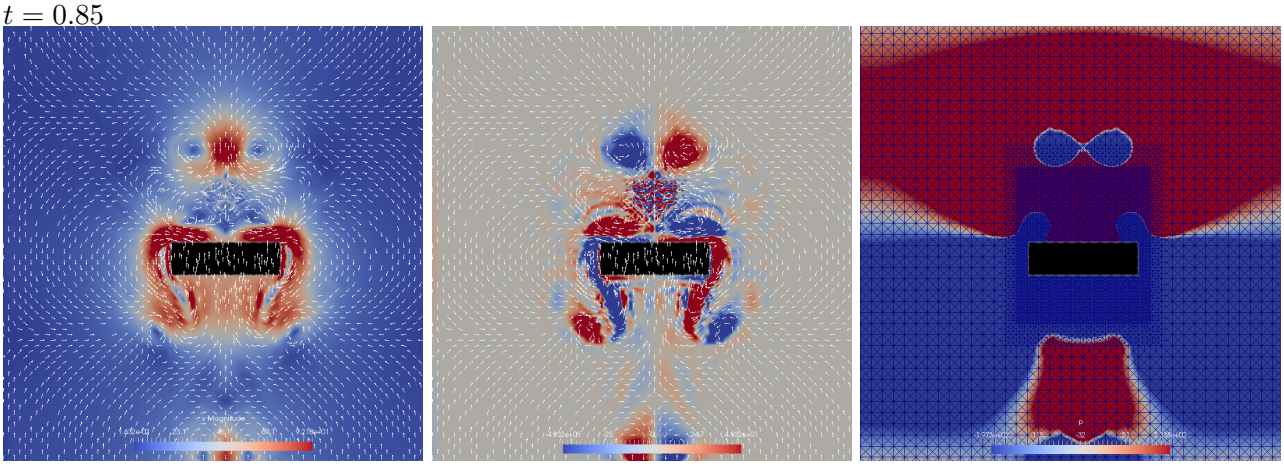
$t = 0.75$



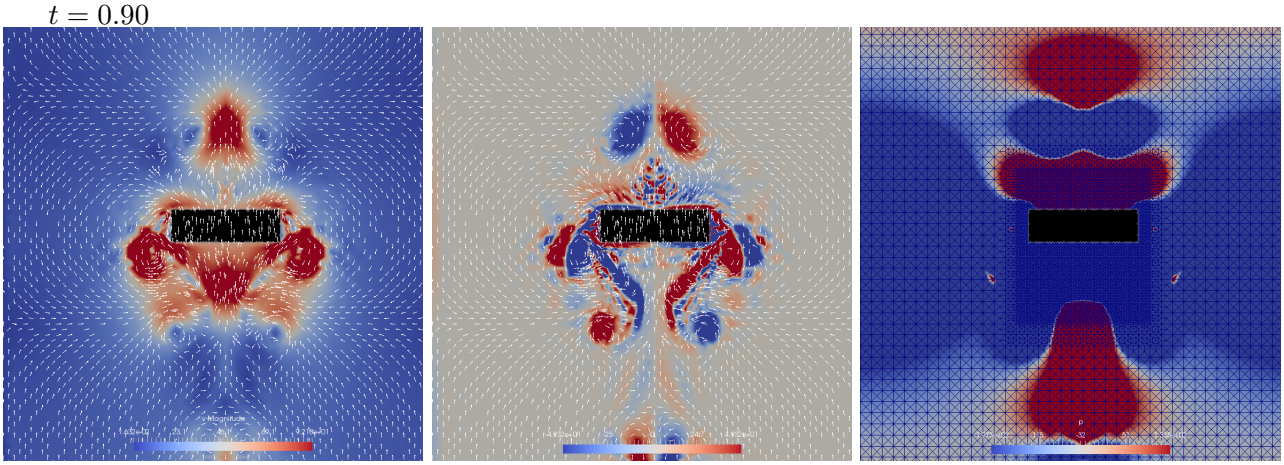
$t = 0.80$



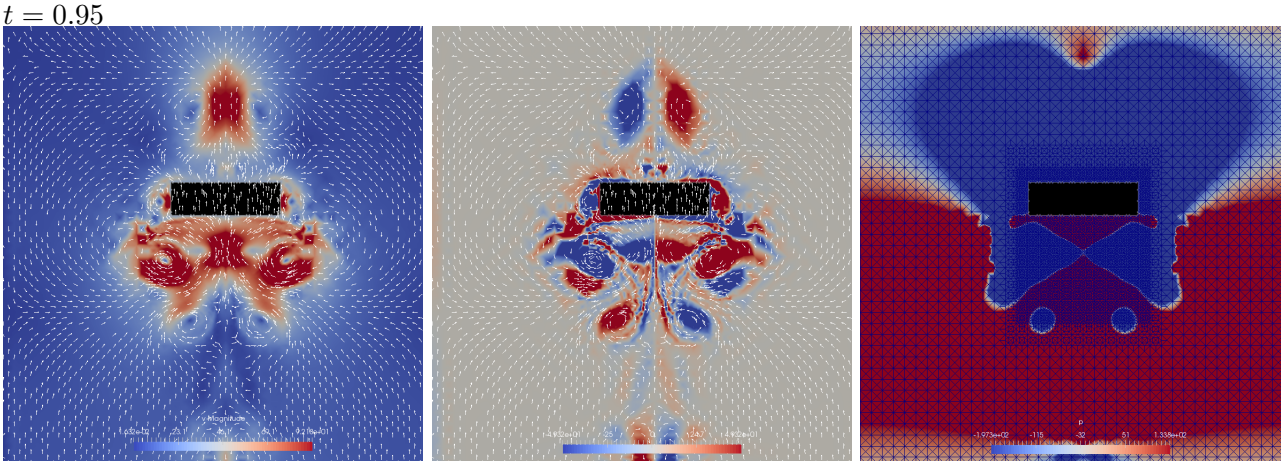
$t = 0.85$



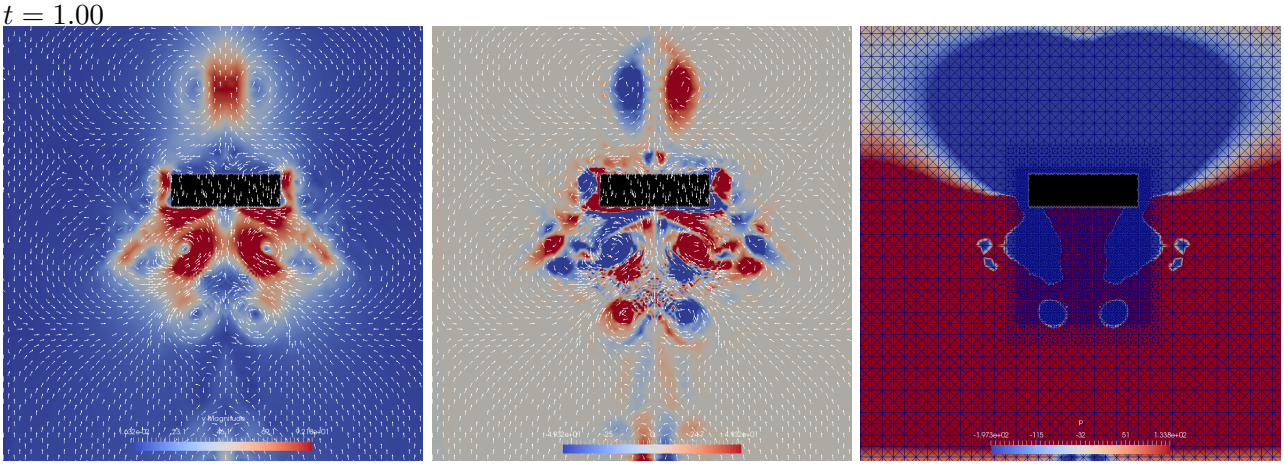
$t = 0.90$



$t = 0.95$



$t = 1.00$



Complete python code

```
from dolfin import *
import mshr
from sympy.utilities.codegen import ccode
```

```
import sympy

comm = mpi_comm_world()
rank = MPI.rank(comm)
set_log_level(INFO if rank ==0 else INFO+1)
parameters["std_out_all_processes"] = False
parameters["ghost_mode"] = "shared_vertex"
parameters['form_compiler']['cpp_optimize'] = True
parameters['form_compiler']['optimize'] = True

#Domain and mesh
W = 40.0 #Width
H = 40.0 #Height
resolution = 1.0
mesh = RectangleMesh(Point(-W/2.0,-H/2.0),Point(W/2.0,H/2.0),int(W/resolution), int(H

#Object translation in time
x,y,t = sympy.symbols('x[0], x[1], t')
PI = 3.141592
c0 = 0.0
c1 = 0.0+5.0*sympy.cos(2.0*PI*(t/0.5))

#Object size
width = 10.0
height = 3.0
objectBool = (x>c0-width/2.0) & (x<c0+width/2.0) & (y>c1-height/2.0) & (y<c1+height/2
object = Expression(ccode(objectBool), t=0, domain=mesh, degree=1)
objectSubDomain = CompiledSubDomain(ccode(objectBool),t=0)
objectVelocity = Expression((ccode(sympy.diff(c0,t)),ccode(sympy.diff(c1,t))), t=0, d

#Mesh refinement near the object
def MarkRefinedCells(mesh,x,xstep,y,ystep,i):
        markers = MeshFunction("bool", mesh, mesh.topology().dim())
        markers.set_all(False)
        for c in cells(mesh):
                bool1= between(c.midpoint()[0],(-x-xstep*i,-width/2.0+xstep))
                bool2= between(c.midpoint()[0],(width/2.0-xstep,x+xstep*i))
                bool3= between(c.midpoint()[1],(-y-ystep*i,-height/2.0+ystep))
                bool4= between(c.midpoint()[1],(height/2.0-ystep,y+ystep*i))
                bool5= between(c.midpoint()[0],(-x-xstep*i,x+xstep*i))
                bool6= between(c.midpoint()[1],(-y-ystep*i,y+ystep*i))
                if ((bool1 or bool2) and bool6) or ((bool3 or bool4) and bool5):
                        markers[c] = True
        return markers

refinements = 2
```

```
for i in range(0,refinements):
        markers = MarkRefinedCells(mesh,6,1,7,2,i)
        mesh = refine(mesh,markers,redistribute = True)

#Facet markers
bndry = FacetFunction("size_t", mesh)
for f in facets(mesh):
    mp = f.midpoint()
    if near(mp[1], -H/2.0):
        bndry[f] = 1
    elif near(mp[1], H/2.0):
        bndry[f] = 2
    elif near(mp[0], -W/2.0) or near(mp[0], W/2.0): # walls
        bndry[f] = 3

#Define finite elements
Ep = FiniteElement("CG",mesh.ufl_cell(),1)
Ev = VectorElement("CG",mesh.ufl_cell(),1)
Ebubble = VectorElement("Bubble", 'triangle', 3)
Evp=MixedElement([Ev,Ebubble,Ep])

#Function space MINI-element
V = FunctionSpace(mesh,Ev)
P = FunctionSpace(mesh,Ep)
W = FunctionSpace(mesh,Evp)

#Function space for object position and velocity
CP = FunctionSpace(mesh,FiniteElement("CG",mesh.ufl_cell(),1))
CV = FunctionSpace(mesh,VectorElement("CG",mesh.ufl_cell(),1))

#Matrix size
info("dimension: {}".format(W.dim()))

# No-slip BC for velocity on walls - boundary id 3
noslip = Constant((0, 0))
bcv_walls = DirichletBC(W.sub(0), noslip, bndry, 3)
bcs = [bcv_walls]

#Auxiliary functions - visualisation
cp=interpolate(object,CP)
cv=interpolate(objectVelocity,CV)

#Facet normal, identity tensor and boundary measure, theta scheme
n = FacetNormal(mesh)
I = Identity(mesh.geometry().dim())
ds = Measure("ds", subdomain_data=bndry)
```

```
Re=10000.0
nu = Constant(1.0/Re)
dt = 0.005
t_end = 1
theta=0.5
k=Constant(1/dt)


#Define unknown and test function(s), b_ is bubble function
(v_, b_, p_) = TestFunctions(W)
v_=v_+b_

#Current unknown time step
w = Function(W)
(v, b, p) = split(w)
v=v+b

#Previous known time step
w0 = Function(W)
(v0, b0, p0) = split(w0)
v0=v0+b0

def T(p,v): return ( -p*I + 2*nu*sym(grad(v)) )

def a(p,v,q,u) :
    return ( inner(grad(v)*v, u) + inner( T(p,v) , grad(u)))*dx

def b(q,v) :
    return inner(div(v),q)*dx

# Define variational forms without time derivative in previous time
F0_eq1 = a(p,v0,p_,v_)
F0_eq2 = b(p_,v)
F0 = F0_eq1 + F0_eq2

# variational form without time derivative in current time
F1_eq1 = a(p,v,p_,v_)
F1_eq2 = b(p_,v)
F1 = F1_eq1 + F1_eq2

# STABILIZATION - interior penalty
h = CellSize(mesh)
alpha = Constant(0.1)
r = avg(alpha)*avg(h) * inner(jump(grad(v),n), jump(grad(v_),n))*dS

# STABILIZATION - directional do-nothing boundary stabilization
```

```
vn=inner(v,n)
sbc= ( inner(grad(v).T*n,v_) )*ds(1)  - 0.5*inner(conditional(gt(vn,0.0),Constant(0.0

#combine variational forms with time derivative
#
#  dw/dt + F(t) = 0 is approximated as
#  (w-w0)/dt + (1-theta)*F(t0) + theta*F(t) = 0
#
F = k*inner((v-v0),v_)*dx + Constant(1.0-theta)*F0 + Constant(theta)*F1 + r + sbc

# Create files for storing solution
name="results_final"
vfile = XDMFFile(mpi_comm_world(),"results_%s/v.xdmf" % name)
cfile = XDMFFile(mpi_comm_world(),"results_%s/c.xdmf" % name)
pfile = XDMFFile(mpi_comm_world(),"results_%s/p.xdmf" % name)
rfile = XDMFFile(mpi_comm_world(),"results_%s/r.xdmf" % name)

for i in [vfile, cfile, pfile, rfile]:
    i.parameters["flush_output"] = True
    i.parameters["rewrite_function_mesh"] = False



# create our own nonlinear problem with possibility of updated DirichletBC
class UpgradedNonlinearProblem(NonlinearProblem):
    def __init__(self, F, J, bcs):
        NonlinearProblem.__init__(self)
        self.PF = F
        self.PJ = J
        self.bcs = bcs
        self.fb = []

    def UpdateObjectDirichletBC(self, W, velocity, mask):
        self.fb = [
            DirichletBC(W.sub(0), velocity, mask, method='pointwise')
            ]

    def F(self, b, x):
        assemble(self.PF, tensor=b)
        for bc in self.bcs: bc.apply(b, x)
        for bc in self.fb: bc.apply(b, x)

    def J(self, A, x):
        assemble(self.PJ, tensor=A)
        for bc in self.bcs: bc.apply(A)
        for bc in self.fb: bc.apply(A)
```

```
J = derivative(F, w)
problem=UpgradedNonlinearProblem(F,J,bcs)
solver=NewtonSolver()

solver.parameters['linear_solver'] = 'mumps'
solver.parameters['absolute_tolerance'] = 1E-8
solver.parameters['relative_tolerance'] = 1e-10
solver.parameters['maximum_iterations'] = 10
solver.parameters['report'] = True
solver.parameters['error_on_nonconvergence'] = False




# Time-stepping
t = dt
while t < t_end:

    info("t = {}".format(t))

    #Update object position and velocity
    object.t=t
    objectSubDomain.t=t
    objectVelocity.t=t

    #Output object position - visualization
    cp.assign(interpolate(object,CP))
    cv.assign(interpolate(objectVelocity,CV))
    cp.rename("c", "mask")
    cfile.write(cp,t)

    # Compute the problem
    begin("Solving ...")
    problem.UpdateObjectDirichletBC(w.function_space(),objectVelocity,objectSubDomain
    solver.solve(problem, w.vector())
    end()

    # Extract solutions:
    (v, b, p) = w.split()

    # output for visualization - v, p, r=rot(v)
    r=project(rot(v),P)
    v=project(v+b,V)
```

```
v.rename("v", "velocity")
p.rename("p", "pressure")
r.rename("r", "vorticity")
# Save to file
vfile.write(v,t)
pfile.write(p,t)
rfile.write(r,t)

# Move to next time step
w0.assign(w)
t += dt
```