CSCI 5636 Final Project

A finite-element approach simulating flow over an airfoil using FEniCS

Luke Meszar

December 15, 2018

Introduction

In this project, I set up a simulation for flow over an airfoil. I used the open source finite-element software FEniCS [ABH+15] to set up the simulations. The goal was to simulate low Reynolds number flow over an airfoil and calculate the lift and drag with various angles of attack.

Methodology

The NACA 5012 airfoil was used for this simulation. The NACA airfoil series [Mar09] is a simple way to describe the geometry of an airfoil. In the four-digit series, the first two numbers describe the maximum camber as a percentage of the maximum chord length and the last two numbers describe the maximum thickness as a percentage of the chord length. This airfoil can be described by a series of Cartesian coordinates. Using the *dat2gmsh.py* script found here. I was able to convert to a format that was readable by GMSH. GMSH [GR09] is an open source geometry and meshing tool. Once I created a mesh fig. 1 in GMSH I was able to export it to a format readable by FEniCS using their tool *dolfin-convert.*

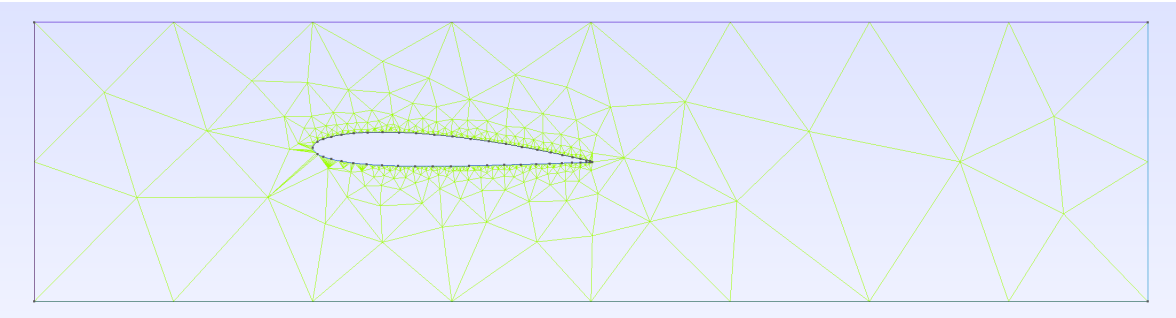

Figure 1: Mesh of the NACA 5012 Airfoil

Now that I had an xml file of the mesh, I was able to write a script that generated the subdomains of the mesh. By assigning different numbers to each region, I was able to partition the mesh. I created a different domain for the interior, the boundary of the airfoil, the left wall, the right wall, and the top and bottom walls. The code I used for this is in *generate_subdomains.py* which can be found in the appendix.

Now that the mesh has been generated and the necessary subdomains marked, a numerical solution can be computed. For this problem, I solved the following incompressible form of the Navier-Stokes equation:

$$\rho\left(\frac{\partial u}{\partial t} + u \cdot \nabla u\right) = \nabla \cdot \sigma(u,p) + f \tag{1.1}$$

$$\nabla \cdot u = 0 \tag{1.2}$$

for the velocity u and the pressure p. Here ρ is the density of liquid and f is the external body forces. Finally, $\sigma(u,p)$ is the stress tensor given by

$$\sigma(u,p) = 2\mu\varepsilon(u) - pI \tag{1.3}$$

where μ is the dynamic viscosity and $\varepsilon(u)$ is the strain-rate tensor

$$\varepsilon(u) = \frac{1}{2}\left(\nabla u + (\nabla u)^T\right). \tag{1.4}$$

To solve this system of equations using a finite-element method, it will have to be converted to a weak form first. Due to the inherent structure of these equations, a simple finite-difference method at each timestep will not work. Instead the method known as IPCS [God79] (Incremental Pressure Correction System) will be used.

At each timestep, we will have to eventually update the values for the velocity and pressures. The IPCS method starts by estimating a tentative velocity u^* using a midpoint finite-difference scheme as

$$\langle \rho(u^* - u^n)/\Delta t, v\rangle + \langle \rho u^n \cdot \nabla u^n, v\rangle + \langle \sigma(u^{n+\frac{1}{2}}, p^n), \varepsilon(v)\rangle + \langle p^n n, v\rangle_{\partial\Omega} - \langle \mu \nabla u^{n+\frac{1}{2}} \cdot n, v\rangle_{\partial\Omega} = \langle f^{n+1}, v\rangle \tag{1.5}$$

with the notation

$$\langle v, w\rangle = \int_\Omega vw\, dx$$
$$\langle v, w\rangle_{\partial\Omega} = \int_{\partial\Omega} vw\, ds.$$

Now, we move on to compute the new pressure using the tentative velocity u^* as

$$\langle \nabla p^{n+1}, \nabla q\rangle = \langle \nabla p^n, \nabla q\rangle - \Delta t^{-1}\langle \nabla \cdot u^*, q\rangle. \tag{1.6}$$

Finally, to compute the new velocity we use

$$\langle u^{n+1}, v\rangle = \langle u^*, v\rangle - \Delta t\langle \nabla(p^{n+1}p^n), v\rangle. \tag{1.7}$$

Now, to show how to implement this method into FEniCS.

```
V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)
```

We create a finite-element space for the velocity and pressure of dimension two and one respectively. This defines a domain consisting of Taylor-Hood elements needed for stability for these equations.

Next, we create the boundary conditions. We choose to have the left, top, and bottom walls have flow defined by the vector $U_0\cos(\theta)\hat{\mathbf{i}} + U_0\sin(\theta)\hat{\mathbf{j}}$. By varying θ, different angles of attack of the wing can be simulated. The boundary of the airfoil is given a noslip boundary condition. The outflow is given a constant value of 0 for the pressure.

```
theta = float(sys.argv[1])
U_0 = 1.0
u_outer_boun = Constant((U_0*cos(theta), U_0*sin(theta)))
noslip = Constant((0, 0))

#airfoil
bcu_airfoil = DirichletBC(V, noslip, sub_domains, 0)
#top and bottom walls
bcu_top_bot = DirichletBC(V, u_outer_boun, sub_domains, 3)
#left wall
bcu_inflow = DirichletBC(V, u_outer_boun, sub_domains, 1)
#right wall
bcp_outflow = DirichletBC(Q, Constant(0), sub_domains, 2)
```

Then the necessary trial and test functions and other parameters necessary to set up the weak form equations are defined. In the code $u\_$ takes the place of u^{n+1} and u_n takes the place of u^n from above.

```
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)

u_n = Function(V)
u_  = Function(V)
p_n = Function(Q)
p_  = Function(Q)



U  = 0.5*(u_n + u)
n  = FacetNormal(mesh)
f  = Constant((0, 0))
k  = Constant(dt)
mu = Constant(mu)
rho = Constant(rho)

def epsilon(u):
return 0.5*(nabla_grad(u) + nabla_grad(u).T)

def sigma(u, p):
return 2*mu*epsilon(u) - p*Identity(len(u))

dx = Measure("dx", domain=mesh, subdomain_data=sub_domains)
ds = Measure("ds", domain=mesh, subdomain_data=sub_domains)
```

Now, we can definition the variation equations for the weak form:

```
F1 = rho*dot((u - u_n) / k, v)*dx \
+ rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
```

```
+ inner(sigma(U, p_n), epsilon(v))*dx \
+ dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
- dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

a2 = dot(nabla_grad(p), nabla_grad(q))*dx
L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx -
(1/k)*div(u_)*q*dx

a3 = dot(u, v)*dx
L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx
```

Here $a1$ and $L1$ correspond to eq. (1.5), $a2$ and $L2$ correspond to eq. (1.6) and $a3$ and $L3$ correspond to eq. (1.7).

Next, we define the functional forms for drag and lift as

```
D = p*n[0]*ds(0)
L = p*n[1]*ds(0)
```

For lift we want to find the contribution of the pressure in the upwards direction and for drag we want to find it in the left to right direction.

Finally, we come to the timestep loop.

```
t = 0
for n in range(num_steps):
        # Update current time
        t += dt

        # Step 1: Tentative velocity step
        b1 = assemble(L1)
        [bc.apply(b1) for bc in bcu]
        solve(A1, u_.vector(), b1, 'bicgstab', 'ilu')

        # Step 2: Pressure correction step
        b2 = assemble(L2)
        [bc.apply(b2) for bc in bcp]
        solve(A2, p_.vector(), b2, 'cg', 'hypre_amg')

        # Step 3: Velocity correction step
        b3 = assemble(L3)
        solve(A3, u_.vector(), b3, 'bicgstab', 'ilu')

        drag_list.append(drag.inner(p_n.vector()))
        lift_list.append(lift.inner(p_n.vector()))
        time_list.append(t)

        # Update previous solution
        u_n.assign(u_)
        p_n.assign(p_)
```

For each solve, a Krylov method was used with a different iteration methods and preconditioners. At each time step, the lift and drag where computed by evaluating the necessary surface integrals.

The full code can be found in *flow_over_airfoil.py* and in the appendix. This code was modeled off the flow around a cylinder example found in this tutorial by Langtangen and Logg [LL16].

Results

The simulation was run with values of θ in $\{0.1, 0.2, \ldots, 1.4\}$. At $\theta = 1.5$, the solution becomes unstable and fails to converge. The simulation was run for 1 second with 250 times steps. I attempted running longer simulations with more times steps, but the results weren't significantly different. I would hypothesis this is because I was working with low Reynolds number flow. This should keep the amount of turbulence low so there wouldn't be any interesting long term behavior.

At each value of θ, a plot was made of the lift and drag over time. For references, the plots for $\theta = 0.0$ and $\theta = 1.4$ are included.

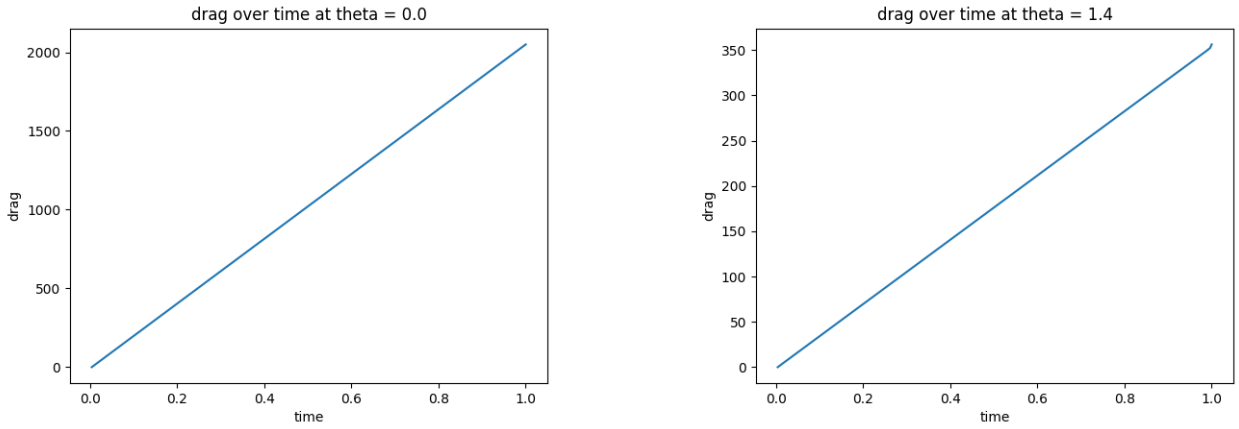

Figure 2: Values for Drag vs. Time

They both display a linear relationship of drag over time. However, at $\theta = 1.4$, the maximum drag is significantly less than for $\theta = 0.0$.

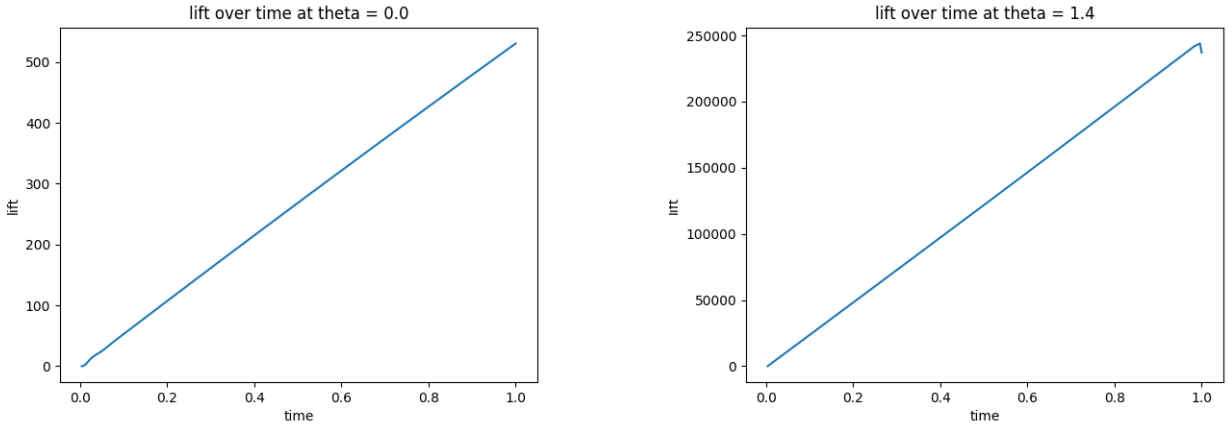

Figure 3: Values for Lift vs. Time

Again, there is almost linear relationship between lift and time and there is an increase in maximum lift at higher values of θ. It is interesting for $\theta = 1.4$, the lift drops off right around one second.

The next two plots show how the maximum lift and drag change with θ.

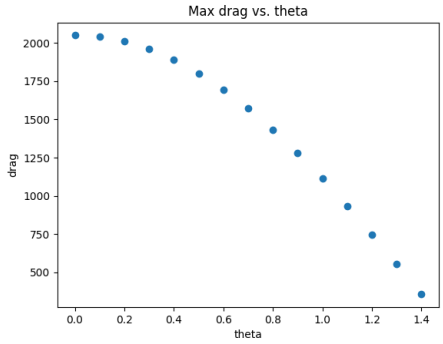

Figure 4

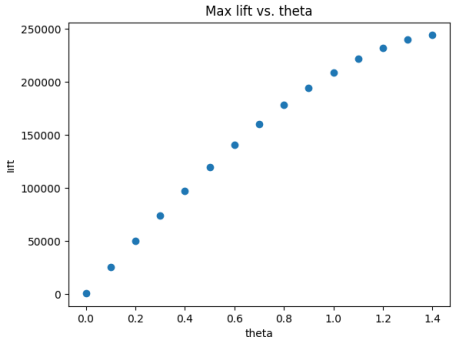

Figure 5

Here, we see that lift and drag have the opposite change as θ increases and in both cases, it makes sense. Note that $\theta = 0.0$ can be though of as being air flowing head on to the leading edge and $\theta = 1.4 \approx \pi/2$ as air flowing from below. Then, it makes sense there would be more drag along against the leading edge if the air is coming head on at $\theta = 0.0$ than when it is coming from below. Conversely, it makes sense that there is more lift if the air is coming from below at $\theta = 1.4$ then from head on.

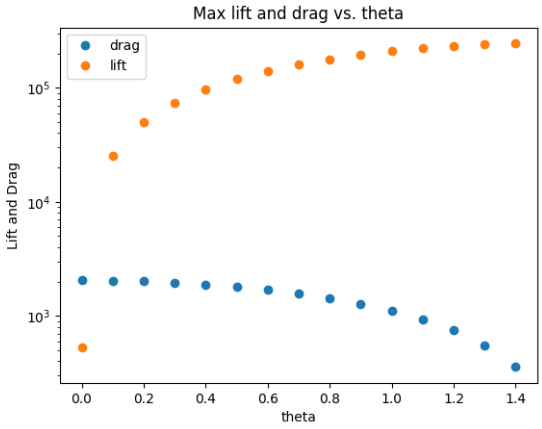

Figure 6: Lift vs. θ on a log-linear scale and drag vs. θ on a linear-linear scale

In fig. 6 we see lift and drag vs. θ together. However, since lift is plotted on a log-linear scale, it is clear that the affects on lift are much greater as θ changes than for drag.

There are a number of .xdmf files in the */navier_stokes_airfoil* folder that provide animations for the velocity and pressure over time for various θ values. However, I was unable to export them and embed them in this writeup due to issue with Paraview.

Future Work

In the future, the first thing I would do is to develop a better understanding of the physics behind fluid dynamics. At the moment, I am not able to accurately verify that my results seems reasonable since I do not understand the underlying dynamics well enough.

Another avenue to explore is how changing the boundary condition on the right wall would affect the end result. I know that what type of condition imposed here can affect the solution upstream so it would be interesting to see what type of changes occur.

Another avenue of future work would be to compare different airfoils. Do they have different maximum lift and drags over different values of θ? Would the equivalent of fig. 6 look different for different airfoils?

References

[ABH+15] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells, *The fenics project version 1.5*, Archive of Numerical Software **3** (2015), no. 100.

[God79] Katuhiko Goda, *A multistep technique with implicit difference schemes for calculating two-or three-dimensional cavity flows*, Journal of Computational Physics **30** (1979), no. 1, 76–95.

[GR09] Christophe Geuzaine and Jean-François Remacle, *Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities*, International journal for numerical methods in engineering **79** (2009), no. 11, 1309–1331.

[LL16] Hans Petter Langtangen and Anders Logg, *Solving pdes in python the fenics tutorial volume i*, Berlin, Springer, 2016.

[Mar09] Pier Marzocca, *The naca airfoil series*, Clarkson University, Retrieved (2009), 07–03.

Appendix

```
from dolfin import *
import sys

set_log_level(1)
# Read mesh
ifile = sys.argv[1]
ofile = sys.argv[2]
if len(sys.argv) > 3:
    ofile2 = sys.argv[3]
else:
    ofile2 = None
mesh = Mesh(ifile)
coords = mesh.coordinates()
x = [a[0] for a in coords]
y = [a[1] for a in coords]
min_x = min(x)
max_x = max(x)
min_y = min(y)
max_y = max(y)
print(min_x,max_x)

# Sub domain for no-slip (mark whole boundary,
#inflow and outflow will overwrite)
class Noslip(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

# Sub domain for inflow (right)
class TopWall(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1], max_y) and on_boundary

class BotWall(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1], min_y) and on_boundary

class Inflow(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0], min_x) and on_boundary

# Sub domain for outflow (left)
class Outflow(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0], max_x) and on_boundary
```

```
# Create mesh functions over the cell facets
sub_domains = MeshFunction("size_t", mesh, mesh.topology().dim() - 1)

# Mark all facets as sub domain 3
sub_domains.set_all(4)

# Mark no-slip facets as sub domain 0
noslip = Noslip()
noslip.mark(sub_domains, 0)

# Mark inflow as sub domain 1
inflow = Inflow()
inflow.mark(sub_domains, 1)

# Mark outflow as sub domain 2
outflow = Outflow()
outflow.mark(sub_domains, 2)

topwall = TopWall()
topwall.mark(sub_domains,3)

botwall = BotWall()
botwall.mark(sub_domains,3)

# Save sub domains to file
file = File(ofile)
file << sub_domains

# Save sub domains to VTK files
if ofile2 != None:
    file = File(ofile2)
    file << sub_domains

from fenics import *
from mshr import *
import numpy as np
import matplotlib.pyplot as plt
import sys

T = 1.0         # final time
num_steps = 250    # number of time steps
dt = T / num_steps # time step size
mu = 0.001         # dynamic viscosity
rho = 1            # density

PROGRESS = 16
TRACE = 13
```

```
mesh = Mesh("airfoil_data/naca5012/naca5012_finer.xml")
sub_domains = MeshFunction("size_t", mesh,\
 "airfoil_data/naca5012/naca5012_finer_subdomains.xml")

V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)

theta = float(sys.argv[1])
U_0 = 1.0
u_outer_boun = Constant((U_0*cos(theta), U_0*sin(theta)))
inflow_profile = ('1', '0')
noslip = Constant((0, 0))


bcu_airfoil = DirichletBC(V, noslip, sub_domains, 0) #airfoil
bcu_top_bot = DirichletBC(V, u_outer_boun, sub_domains, 3) #top and bottom walls
bcu_inflow = DirichletBC(V, u_outer_boun, sub_domains, 1) #left wall
# bcu_top_bot = DirichletBC(V, noslip, sub_domains, 3) #top and bottom walls
# bcu_inflow = DirichletBC(V, Expression(inflow_profile,degree=2), sub_domains,
bcp_outflow = DirichletBC(Q, Constant(0), sub_domains, 2) #right wall
bcu = [bcu_airfoil, bcu_inflow, bcu_top_bot]
bcp = [bcp_outflow]



u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)

u_n = Function(V)
u_  = Function(V)
p_n = Function(Q)
p_  = Function(Q)



U  = 0.5*(u_n + u)
n  = FacetNormal(mesh)
f  = Constant((0, 0))
k  = Constant(dt)
mu = Constant(mu)
rho = Constant(rho)

def epsilon(u):
    return 0.5*(nabla_grad(u) + nabla_grad(u).T)
```

```
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))


dx = Measure("dx", domain=mesh, subdomain_data=sub_domains)
ds = Measure("ds", domain=mesh, subdomain_data=sub_domains)


F1 = rho*dot((u - u_n) / k, v)*dx \
   + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

a2 = dot(nabla_grad(p), nabla_grad(q))*dx
L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx

a3 = dot(u, v)*dx
L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx

A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)

D = p*n[0]*ds(0)
L = p*n[1]*ds(0)



[bc.apply(A1) for bc in bcu]
[bc.apply(A2) for bc in bcp]

xdmffile_u = XDMFFile('navier_stokes_airfoil/velocity_naca5012' + str(theta) +
xdmffile_p = XDMFFile('navier_stokes_airfoil/pressure_naca5012' + str(theta) +

#timeseries_u = TimeSeries('navier_stokes_airfoil/velocity_series_naca5012')
#timeseries_p = TimeSeries('navier_stokes_airfoil/pressure_series_naca5012')

File('navier_stokes_airfoil/naca5012.xml.gz') << mesh

progress = Progress('Time-stepping')
set_log_level(PROGRESS)

drag_list = []
```

```
lift_list = []
time_list = []

drag = assemble(D)
lift = assemble(L)

t = 0
counter = 0
for n in range(num_steps):
    counter += 1
    print("counter", counter)
    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1, 'bicgstab', 'ilu')

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p_.vector(), b2, 'cg', 'hypre_amg')

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u_.vector(), b3, 'bicgstab', 'ilu')

    drag_list.append(drag.inner(p_n.vector()))
    lift_list.append(lift.inner(p_n.vector()))
    time_list.append(t)

    # Plot solution

    # Save solution to file (XDMF/HDF5)
    xdmffile_u.write(u_, t)
    xdmffile_p.write(p_, t)

    #timeseries_u.store(u_.vector(), t)
    #timeseries_p.store(p_.vector(), t)

    # Update previous solution
    u_n.assign(u_)
    p_n.assign(p_)

    # Update progress bar
    print('u_max:', u_.vector().get_local().max())
```

```
def plot_functionals(times, forces, type, theta):

    plt.plot(times, forces)
    plt.xlabel('time')
    plt.ylabel(type)
    plt.title(type + " over time at theta = " + str(theta))
    plt.savefig("lift_and_drag/" + type + "_" + str(theta) + ".png")
    plt.clf()

plot_functionals(time_list, drag_list, "drag", theta)
plot_functionals(time_list, lift_list, "lift", theta)
print(max(drag_list), max(lift_list))
```