Project in Continuum Mechanics: Simulating Fluid Flow in Complex Geometries using FEniCS

Gaute Linga* and Asger Bolet
Biocomplexity, Niels Bohr Institute, University of Copenhagen
*gaute.linga@nbi.ku.dk

Abstract

This document presents the project given in the course Continuum Mechanics, 2016, concerning simulation of steady viscous flow in various 2D geometries. We briefly present some background and instructions on the flow equations, the finite element method, and the FEniCS framework, which shall be used to solve the flow equations. A guide on how to set up FEniCS on your own computer is presented. Three exercises are given, with increasing difficulty: (1) Laminar pipe flow, (2) force calculation from flow around objects, and (3) lid-driven cavity flow.

Introduction

Stokes flow

Incompressible flow is in general described by the Navier–Stokes equations,

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\boldsymbol{\nabla})\mathbf{u} - \nu\boldsymbol{\nabla}^2\mathbf{u} = \mathbf{f} - \boldsymbol{\nabla}p, \tag{1a}$$

$$\boldsymbol{\nabla}\cdot\mathbf{u} = 0, \tag{1b}$$

where $\mathbf{u}(\mathbf{x}, t)$ is the velocity field, $\mathbf{f}(\mathbf{x}, t)$ a body force, ν the kinematic viscosity, and $p(\mathbf{x}, t)$ the pressure field.

In many settings the inertial forces are small compared to viscous forces, i.e. the Reynolds number $\mathrm{Re} = |\mathbf{u}|\ell/\nu \ll 1$, where ℓ is a characteristic length scale of the flow. In this case, eq. (1a) reduces to the time-independent equation[1]

$$\nu\boldsymbol{\nabla}^2\mathbf{u} = \boldsymbol{\nabla}p - \mathbf{f}. \tag{2}$$

If we further neglect body forces and rescale $p \to \nu p$, we get the equation system

$$\boldsymbol{\nabla}^2\mathbf{u} = \boldsymbol{\nabla}p, \tag{3a}$$

$$\boldsymbol{\nabla}^2 p = 0, \tag{3b}$$

where eq. (3b) is found by taking the gradient of eq. (3a) and using the incompressibility condition eq. (1b).

The system of eqs. (3a) and (3b), to be solved for $\mathbf{x} \in \Omega$ where Ω is our domain, is the starting point of the analysis you will be doing. For simplicity, we will assume prescribed velocity boundary conditions,

$$\mathbf{u}(\mathbf{x}) = \mathbf{u}_0(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \partial\Omega_{\mathbf{u}}, \tag{4}$$

and prescribed pressure boundary conditions (normal stress),

$$p(\mathbf{x}) = p_0(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \partial\Omega_p, \tag{5}$$

where $\partial\Omega = \partial\Omega_{\mathbf{u}} \cup \partial\Omega_p$ is the boundary of Ω.

Finite Element Method

The finite element method (FEM) is an efficient method for solving partial differential equations (PDEs) in complex geometries. Basically, it consists in approximating the *solution* to a PDE rather than approximating the *equation itself* (as would finite difference methods and finite volume methods). This is obtained by employing calculus of variations. We will not present the FEM theory

[1] On very short time scales, however, the time-derivative term may be important.

in detail here, just a direct explanation of how our program will work. For a thorough introduction, see e.g. [1, 5].

The quick and dirty recipe of FEM is the following: The domain is discretized by dividing it into many subdomains, called elements. In our case, the elements are triangles. On each element, we approximate the solution by linear combinations of basis functions defined on each element. The element-wise equations defining the coefficients in these linear combinations are assembled for the entire system, yielding a (large, and often sparse) set of algebraic equations to be solved.

In order to use FEM to solve our equations, we must formulate the problem in a weak (variational) form. The basis functions belong to the function spaces V (velocity space) and P (pressure space)

By multiplying our equations (unknowns = trial functions $(\mathbf{u}, p)$) by the test functions $(\mathbf{v}, q)$ and successively integrating by parts, we may obtain the following (weak) problem formulation:

Weak formulation: *Find* $(\mathbf{u}, p) \in W$ *such that*

$$a((\mathbf{u}, p), (\mathbf{v}, q)) = L((\mathbf{v}, q)) \quad \text{for all} \quad (\mathbf{v}, q) \in W, \tag{6a}$$

$$\text{and} \quad \mathbf{u} = u_0 \quad \text{on} \quad \partial\Omega, \tag{6b}$$

where $W = V \times P$ *is a mixed function space, and*

$$a((\mathbf{u}, p), (\mathbf{v}, q)) = \int_\Omega (\boldsymbol{\nabla}\mathbf{u} : \boldsymbol{\nabla}\mathbf{v} - p\boldsymbol{\nabla} \cdot \mathbf{v} + q\boldsymbol{\nabla} \cdot \mathbf{u})\, \mathrm{d}V, \tag{6c}$$

$$L((\mathbf{v}, p)) = -\int_{\partial\Omega_p} p_0 \mathbf{n} \cdot \mathbf{v} \mathrm{d}S. \tag{6d}$$

This problem formulation is all we need. Now, to avoid most of the technicalities of FEM (actually constructing the basis functions, solving the integral equations on the element level, assembling the system matrix, and enforcing the boundary conditions, . . .) and get straight to the physics, we use the FEniCS framework.

FEniCS/DOLFIN

FEniCS is a collection of software for automated solution of PDEs using FEM [7]. The component of FEniCS we will use to communicate with the FEniCS engine via Python is called DOLFIN [8]. Some of the functionality we will use is

- Making a simple mesh
- Setting up subspaces and basis functions
- Applying boundary conditions
- Solving the system
- Visualizing the solution

The bottom line is, all we need to do is to supply the variational form and boundary conditions and the order of the elements (i.e. the polynomial degree of the basis functions)—FEniCS does the rest (and it's quite fast!).

Tip: If you want to get really dirty with FEniCS, you should consider following their excellent tuturial [6].

Setting up FEniCS

In this section, you will learn how to set up FEniCS on your own computer.

Getting FEniCS

Go to http://fenicsproject.org/download/, choose your operative system, and follow the instructions thereunder. Make sure you get the latest stable version (1.6.0 at the time of writing this). If you have the choice, I strongly recommend you to use Ubuntu.

Ubuntu: To get the latest version, make sure to add the FEniCS PPA before installing. Execute the following in a terminal:

```
sudo add-apt-repository ppa:fenics-packages/fenics
sudo apt-get update
sudo apt-get install fenics
sudo apt-get dist-upgrade
```

Mac OS X or Windows:

- *Alternative 1: (Advanced)* For people who want to continue with scientific programming[2], I recommend you to switch to Ubuntu—e.g. make a dual boot or a Ubuntu Live USB. This project is a unique possibility to start using it.

- *Alternative 2:* The recommended way (by the FEniCS team) of installing FEniCS on Mac and Windows is to use their prebuilt Docker images. To do this; first install Docker (full version) for your operating system. Instructions for installing Docker are found at `https://docs.docker.com/engine/getstarted/step_one/`. The instructions for installing FEniCS in Docker are found at `http://fenics-containers.readthedocs.io/en/latest/introduction.html`.

Additionally, we will need a working Python installation, with Numpy and Matplotlib. If you want to visualize your results, I suggest installing ParaView. You can write your code in any text editor; Emacs is a good candidate.

Minimal working example

To check if the installation was successful, open the terminal (Ctrl+Alt+T) and open Python in the terminal:

```
python
```

and perform the commands

```
import dolfin as df
df.__version__
```

If the output says '`2016.2.0`, you have probably succeeded.

If you want to further test your installation, you can run some of the scripts found at the FEniCS home page, for example `https://fenics.readthedocs.io/projects/dolfin/en/latest/demos.html` These should work out of the box.

How to structure your program

In general, I suggest that you break your program down in small units (functions) which you can test individually. If the components themselves work, for any input, it is likely that the whole thing will work.

The way the example program is structured in the following is not necessarily good coding conduct; it is structured this way for pedagogical purposes.

In the following, you will be asked to run simulations for different input parameters. If you are an advanced Python programmer, you may want to make automated scripts, and systematically store the input and output values to a data file. However, this is not necessary—you can also copy/paste the output to your data analysis software of choice.

When you are asked to analyze or plot the output of FEniCS simulations, use the software you are most familiar and comfortable with: Matlab, Python with Matplotlib (be sure to make a separate script), Gnuplot, R, or Excel. I won't judge you.

[2]In this context, astrophysics, astrology and astronomy is not regarded as 'scientific'.

Exercises

It is time to get to the exercises. We will restrict our scope to flow in two dimensions.

Pipe flow

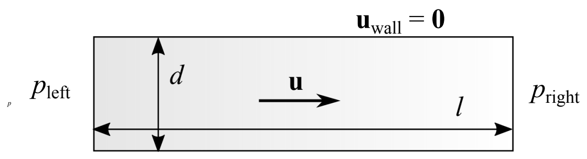

Figure 1: Pipe flow setup.

The first case concerns simulation of pipe flow. In this case we will go step-by-step through an example program. Use your favourite text editor to make a file named, e.g., `pipeflow.py`. You are encouraged to run your program several times as we go along. To do this, use the command:

```
python pipeflow.py
```

We shall consider a pipe of diameter $d = 4$ and length $l = 10$, as shown in fig. 1. On the left side, we prescribe the pressure $p(0, y) = 1$ and on the right $p(l, y) = 0$. Along the pipe walls we use the no-slip condition $\mathbf{u}(x, 0) = \mathbf{u}(x, D) = \mathbf{0}$.

Load DOLFIN: First, we need to load all the capabilities of FEniCS using the Python module DOLFIN, and the numerical tools package Numpy. This can be done using

```
from dolfin import *
import numpy as np
```

where we have for convenience adopted the entire `dolfin` namespace.

Domain and mesh: We now have to define and discretize our domain. To create the domain, we can use the function `RectangleMesh(p0, p1, nx, ny, diagonal="right")` which draws a `nx`×`ny` rectangle between the points `p0` and `p1`.

This can e.g. be done as

```
# Define domain and discretization
delta = 0.5
length = 10.0
diameter = 4.0

# Create mesh
p0 = Point(np.array([0.0, 0.0]))
p1 = Point(np.array([length, diameter]))
nx = int(length/delta)
ny = int(diameter/delta)

mesh = RectangleMesh(p0, p1, nx, ny)
```

You can interactively visualize the mesh using

```
plot(mesh, title="Mesh")
interactive()
```

Mark the boundary: In order to apply boundary conditions, we must mark the different parts of the boundary correctly. We may first define some labels,

```
# Making a mark dictionary
# Note: the values should be UNIQUE identifiers.
mark = {"generic": 0,
        "wall": 1,
        "left": 2,
        "right": 3 }
```

Now we define a function marking the subdomains of the mesh. We first mark the entire domain as “generic”.

```
subdomains = MeshFunction("size_t", mesh, 1)
subdomains.set_all(mark["generic"])
```

The subdomain function should be marked with the correct labels on the correct parts of the mesh. Now, we define the `Left` part of the boundary, which inherits the `SubDomain` class:

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

Here, `on_boundary` is a flag indicating if a node is on the boundary, and `near(a, b)` is a function returning true if `a` and `b` are approximately equal (difference below some threshold). Note that `x[0]` refers to the first coordinate of `x`.

♠ Write the classes `Right` and `Wall` for the remaining boundaries.

Now we apply the mark defined previously for the boundaries:

```
left = Left()
left.mark(subdomains, mark["left"])
```

Now the `left` boundary has been marked.

♠ Mark the boundaries `right` and `wall` as well.

Finally, you can visualize the `subdomains` mesh function using the command

```
plot(subdomains, title="Subdomains")
interactive()
```

♠ Do the boundaries have the correct value according to the `mark` labels defined above?

Define function spaces: We now move away from the boundary for a little while. We define our function spaces, effectively the set of basis functions on our elements. To avoid stability issues which can be a problem for mixed spaces (look up the Babuska–Brezzi condition), we use Taylor–Hood elements to resolve the problem. This means that we use second-order continuous-Galerkin (CG) basis functions for the velocity and first-order CG for the pressure.

This can be implemented as follows, by first defining the elements and constructing a function space from this:

```
# Define function spaces
V = VectorElement("CG", triangle, 2)
P = FiniteElement("CG", triangle, 1)
W = FunctionSpace(mesh, V*P)
```

Here, `W` is the mixed space.

Define variational problem: We now define trial and test functions living in the space `W`:

```
# Define variational problem
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
```

We also define measures for our integration (matrix assembly). We supply `subdomains` to divide the domain according to which `mark` it has (see below).

```
dx = Measure("dx", domain=mesh, subdomain_data=subdomains)   # Volume integration
ds = Measure("ds", domain=mesh, subdomain_data=subdomains)   # Surface integration
```

To assemble the surface integral in the variational form, we need the surface normal, the inlet/outlet pressures, and a (dummy) body force:

```
# Surface normal
n = FacetNormal(mesh)

# Pressures. First define the numbers (for later use):
p_left = 1.0
p_right = 0.0

# ...and then the DOLFIN constants:
pressure_left = Constant(p_left)
pressure_right = Constant(p_right)

# Body force:
force = Constant((0.0, 0.0))
```

Finally, we are ready to feed DOLFIN the variational form, cf. eqs. (6a) to (6d):

```
a = inner(grad(u), grad(v))*dx - p*div(v)*dx + q*div(u)*dx
L = inner(force, v) * dx \
    - pressure_left * inner(n, v) * ds(mark["left"]) \
    - pressure_right * inner(n, v) * ds(mark["right"])
```

Apply (Dirichlet) boundary conditions: We now return to the boundary. The no-slip (zero velocity) boundary condition can be defined as:

```
noslip = Constant((0.0, 0.0))
bc_wall = DirichletBC(W.sub(0), noslip, subdomains, mark["wall"])
```

Here, `W.sub(0)` refers to the first subspace of `W`, namely `V`. Similarly, `W.sub(1)` refers to `P`.

♠ Implement also the *pressure* boundary conditions `bc_left` and `bc_right`, using, respectively, `pressure_left` and `pressure_right`. Remember that this must be applied to the pressure subspace using `W.sub(1)`.

Finally, we collect the boundary conditions into one vector:

```
bcs = [bc_wall, bc_left, bc_right]
```

Solve variational problem: The problem is now fully specified, and all that remains is to solve it. We define the temporary solution vector `w` (corresponding to the mixed space `W`), and solve the variational problem with the supplied Dirichlet BCs `bcs` using the function `solve(problem, w, bcs)`. If `problem` is of the form `a == L`, as below, DOLFIN automatically deduces that it is a linear problem.

```
# Compute solution
w = Function(W)
solve(a == L, w, bcs)
```

Finally, we split the solution vector `w` into its velocity and pressure parts:

```
# Split using deep copy
(u, p) = w.split(True)
```

This should give you the solution for velocity and pressure, respectively through `u` and `p`.

Visualize the solution: The straightforward plotting tool is easy to use:

```
# Plot solution
plot(u, title="Velocity")
plot(p, title="Pressure")
interactive()
```

You might want to visualize the speed $|\mathbf{u}|$. It is therefore useful to implement the function `magnitude`:

```
# Magnitude function
def magnitude(vec):
    return sqrt(vec**2)
```

Then you can plot the speed by:

```
plot(magnitude(u), "Speed")
```

Tip: For more sophisticated interactive plotting, you can use ParaView[4]. Export the solution using the following commands:

```
# Save solution in VTK format
ufile = File("velocity.pvd")
ufile << u
pfile = File("pressure.pvd")
pfile << p
```

Now you can open the `.pvd`-files using ParaView.

Assess the solution: You are now asked to inspect the solution.

- ♠ How does the solution for the velocity field $\mathbf{u}$ look? What about the pressure p?
- ♠ Vary and interchange the values of `p_left` and `p_right`. How does the solution change?
- ♠ How does the numerical solution, for $\mathbf{u}$ and p, compare to the analytical solution?

One way of doing this quantitatively is the following: Construct first the analytical solution in the domain. This is done by constructing an `Expression` (which evaluates C code in the domain; therefore we have to pass variables in a special way—see code below), and interpolating it onto the `V` subspace. For both `p` and `u`:

```
# Define coefficient
coeff = (p_left-p_right)/(2*length)
# Define expressions
u_analytic = Expression(("C*x[1]*(2.0*rad - x[1])", "0.0"),
                        C=coeff, rad=0.5*diameter, degree=2)
p_analytic = Expression("p_l - 2*C*x[0]", p_l=p_left, C=coeff,
                        degree=1)
# Project onto domains
u_analytic = interpolate(u_analytic, W.sub(0).collapse())
p_analytic = interpolate(p_analytic, W.sub(1).collapse())
```

Subtract the analytical solution from the original one and take the magnitude:

```
u_error_local = magnitude(u - u_analytic)
```

Integrate (assemble) over the whole domain to get the global error:

```
u_error = assemble(u_error_local*dx)
print "u_error =", u_error
```

♠ How do the errors in $\mathbf{u}$ and p change as you refine the grid (i.e. change `delta`)?

Normally, we would expect that the error decreases with refined grid according to the order of the elements.

♠ Why may this not happen here? **Hint:** Look at the polynomial order of the function spaces `V` and `P` and of the analytical solution for $\mathbf{u}$ and p.

The divergence of the velocity field is found by

```
div_u = div(u)
div_u = project(div_u, W.sub(1).collapse()) # Project on the pressure subspace since it is scalar order 1 (V is 2)
```

and it should, of course, be zero for infinite numerical precision.

♠ How does the error in $\nabla \cdot \mathbf{u}$ change as you refine the grid?

Flow around objects

In this exercise, you will learn to calculate the drag force F_{drag} and lift force F_{lift} on arbitrarily shaped objects in an otherwise uniform creeping flow, as sketched in fig. 2.

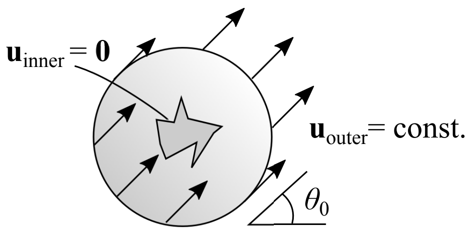
$\mathbf{u}_{\text{inner}} = \mathbf{0}$
$\mathbf{u}_{\text{outer}}$= const.
θ_0

Figure 2: Setup for flow around objects.

We will use a constant velocity condition on the outer boundary,

$$\mathbf{u}(\mathbf{x}) = U_0 \cos\theta_0 \hat{\mathbf{e}}_x + U_0 \sin\theta_0 \hat{\mathbf{e}}_y, \tag{7}$$

where θ_0 is a fixed angle. The final aim of this exercise is to calculate F_{drag} and F_{lift} as a function of rotation angle θ_0, and thereby, among other things, find the most "aerodynamic" angle in low Reynolds number flow.

♠ You can now open your favourite text editor and create a file named, e.g., `flowaround.py`.

It is a good idea to base the program on what you learned from the previous exercise, and reuse some functions where possible. For example, the function spaces, test functions and trial functions will be the same as before. In the variational problem, the only change is that `L` becomes

```
L = inner(force, v) * dx
```

since we no longer have pressure boundary conditions.

Meshes: In this case, we will supply you with meshes (or you can make your own, if you prefer and are able to).

The following meshes are provided: An obstacle-free mesh (`free_2d.xml.gz`), a mesh with circular obstacle (`circle_2d.xml.gz`), and a mesh with a NACA airfoil [2] as obstacle (`naca_2d.xml.gz`). You can load a pre-generated mesh by the following command:

```
mesh = Mesh("free_2d.xml.gz")
```

Mark the boundary: In this case, we have two boundaries: outer and inner boundary.

♠ Mark the boundary in a similar manner as in the previous exercise, but now with the boundary mark labels `"inner_boun"` and `"outer_boun"`.

Hint: The command

```
rad2 = x[0]**2 + x[1]**2
```

gives the squared distance from the center, and the flag `on_boundary` yields (as before) `True` if a point is on the boundary or `False` if not. The meshes we provide you with are centered at the origo. **Note:** You should not use the name `inner` on a variable, as this name is reserved for the inner product function in DOLFIN.

♠ Visualize the `subdomains` function to ensure that the boundary is correctly marked.

Apply (Dirichlet) boundary conditions: Similarly as in the previous exercise, we will apply Dirichlet boundary conditions on the velocity field. For prescribed, constant boundary velocity, this can be achieved by the commands:

```
u_outer_boun = Constant((U_0*cos(theta), U_0*sin(theta)))
bc_outer_boun = DirichletBC(W.sub(0), u_outer_boun, subdomains, mark["outer_boun"])
```

♠ Implement and apply the (no-slip) inner boundary condition as well.

To have a well-posed problem you need to fix the pressure at one node to some reference value. Take e.g. the node to the left:

```
bc_p = DirichletBC(W.sub(1), 0, "x[0] < -1.0 + DOLFIN_EPS", "pointwise")
```

Analyse the solution: You now have all the information needed to calculate the flow in any of these domains.

First, to verify that the solver is working correctly, you can run a simulation using the mesh `free_2d.xml.gz`, where the domain is free of obstacles.

♠ Does the solution match the (trivial) analytical solution?

Now, we switch to more complex geometries. The following tasks should be done for the mesh `circle_2d.xml.gz` (flow around a circle) and `naca_2d.xml.gz` (NACA airfoil).

♠ Visualize the solution for an angle of choice θ_0. Where is the pressure and velocity highest and lowest? Why?

The *second deviatoric stress invariant*, J_2, which measures how sheared the liquid is, is given by

$$J_2 = \frac{1}{2}\mathrm{tr}(\boldsymbol{\sigma}_{\mathrm{visc}}^2). \tag{8}$$

Calculation of the viscous stress tensor, total stress tensor and J_2 can be implemented as follows:

```
# Viscous stress
# sym returns the symmetric part of a matrix
stress_visc = 2*sym(grad(u))

# Total stress
stress = -p*Identity(2) + stress_visc

# Second deviatoric (viscous) stress invariant
J_2 = 0.5*tr(stress_visc*stress_visc)
```

- ♠ Calculate the viscous and total stress in the fluid, and visualize J_2.

Tip: If you want to visualize it in ParaView, you can export the tensor field $\boldsymbol{\sigma}$ by the following commands:

```
T = TensorFunctionSpace(mesh, "CG", 1) # Order 1, as it is a deriv. of V (order 2)
stress = project(stress, T)
stress_file = File("stress.pvd")
stress_file << stress
```

We will now calculate the traction on the inner boundary, and decompose it into a drag force and a lift force by using the direction of the imposed flow, `n_flow`. A way to do this is the following, assuming you have already calculated the boundary normal `n` and the surface measure `ds` (see previous exercise if you haven't):

```
# Imposed flow direction:
n_flow = Constant((cos(theta), sin(theta)))
n_flow = project(n_flow, W.sub(0).collapse())

# Perpendicular to n_flow:
t_flow = Constant((sin(theta), -cos(theta)))
t_flow = project(t_flow, W.sub(0).collapse())

# Compute traction vector
traction = dot(stress, n)

# Integrate decomposed traction to get total F_drag and F_lift
F_drag = assemble(dot(traction, n_flow) * ds(mark["inner"]))
F_lift = assemble(dot(traction, t_flow) * ds(mark["inner"]))
```

Now you should be able to calculate the lift force F_{lift} and drag force F_{drag} for any imposed flow angle θ_0.

- ♠ Systematically vary the imposed flow angle $\theta_0 \in [0, 2\pi]$ and plot F_{lift} and F_{drag} as functions of θ_0. Locate the extremal points.

- ♠ Do you recognize the symmetry of the geometry (and the equations) in the plotted graphs?

Extra task for the most ambitious: As you might be aware of, "unbounded" Stokes flow can not occur in 2D (see Stokes' paradox, e.g. [3]). This makes our numerical experiments only applicable to confined flows, and our calculated drag coefficients would not converge to a finite value as we increased the domain and kept the obstacle size fixed. To add some realism, you are therefore invited to include the inertia term in your calculations. The necessary information for doing so is given in the next exercise.

- ♠ Plot the drag force F_{drag} as a function of Re for different angles. How does it change with increasing Re? It has been proposed that $F_{\text{drag}} \sim \text{Re}^{-1}$ for small Re. Can you see this from your simulations?

Lid-driven cavity

You are quickly becoming an expert in FEniCS. We shall now consider one of the classic test cases in computational fluid dynamcs (CFD), namely that of lid-driven cavity flow. We consider a square geometry (cavity) with side length $l = 1$. The left, right and bottom boundaries have no-slip conditions and the top (lid) is driven with a velocity $U_0 = 1$ in the x direction. This results in a circulation in the cavity. The setup is sketched in fig. 3.

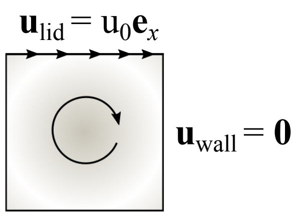
$\mathbf{u}_{\text{lid}} = \mathbf{u}_0\mathbf{e}_x$
$\mathbf{u}_{\text{wall}} = \mathbf{0}$

Figure 3: Setup for lid-driven cavity flow.

- ♠ Open your favourite text editor and create a file named, e.g., `cavity.py`.
- ♠ Create a mesh for the cavity, in the area $[0, l] \times [0, l]$. You can start with discretization resolution $N = 32$ (=`nx`=`ny`) points along each axis.
- ♠ Create the classes `Wall` and `Lid` to mark the boundary of the cavity.

In this case, we shall explore the effect of including the advection term in the solver, i.e. investigate $\mathrm{Re} > 0$. We will still restrict our scope to seeking steady-state solutions. The equations describing the problem are then, disregarding the body force,

$$\nu\boldsymbol{\nabla}^2\mathbf{u} - (\mathbf{u}\cdot\boldsymbol{\nabla})\mathbf{u} = \boldsymbol{\nabla}p, \tag{9a}$$

$$\boldsymbol{\nabla}\cdot\mathbf{u} = 0 \tag{9b}$$

so that the weak form of the problem becomes: Find $(\mathbf{u}, p) \in W$ such that

$$\int_\Omega (\nu\boldsymbol{\nabla}\mathbf{u} : \boldsymbol{\nabla}\mathbf{v} + (\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u})\cdot\mathbf{v} - p\boldsymbol{\nabla}\cdot\mathbf{v} + q\boldsymbol{\nabla}\cdot\mathbf{u})\,\mathrm{d}V = 0 \quad \text{for all} \quad (\mathbf{v}, q) \in W. \tag{10}$$

The non-linear functional `F` describing the problem can be impemented as

```
F = inner(grad(u)*u, v)*dx \
    + nu*inner(grad(u), grad(v))*dx \
    - p*div(v)*dx \
    - q*div(u)*dx
```

Here, some variables (`w`, `u`, `p`, `nu`) have to be defined beforehand, by

```
Reynolds = 1.0  # or any other value
nu = 1./Reynolds

# Solution vectors
w = Function(W)
u, p = (as_vector((w[0], w[1])), w[2])
```

The inertia term is non-linear, so a standard linear solver will not do in solving this problem. To optimally use a non-linear solver, we supply the Jacobian with respect to the (mixed) field `w` that we are solving for. This is implemented as the following:

```
J = derivative(F, w)  # Jacobian
```

Then the non-linear solver can be invoked:

```
solve(F == 0, w, bcs, J=J)
```

Here, the `solve()` function automatically recognizes the form `F == 0` as a non-linear problem. Newton's method is the default non-linear solver, and works well for relatively small meshes. You can set the reference pressure BC at the bottom-left-most node by

```
bc_p = DirichletBC(W.sub(1), 0, "x[0] < DOLFIN_EPS && x[1] < DOLFIN_EPS", "pointwise")
```

Remember that you have to include `bc_p` in the `bcs` vector.

♠ Implement the above, along with the described boundary conditions, to solve the steady-state Navier–Stokes equations for lid-driven cavity flow. Visualize the resulting fields.

The most straightforward way to get the streamlines of the flow field we can place "particles" at random in the domain and passively advect them, i.e. integrate up their velocity. The script `trace.py` we provide you with does this. First, you have to export the mesh and velocity field to `.xml.gz` format.

```
u_file = File("u_Re" + str(Reynolds) + "_N" + str(N) + ".xml.gz")
mesh_file = File("mesh.xml.gz")
u_file << u
mesh_file << mesh
```

Then you can open a new terminal and run the script by:

```
python trace.py mesh.xml.gz u_Re10_N100.xml.gz trace_Re10_N100.dat -ds 0.005 -S 5 -n 100
```

This will create a data file `trace_Re10_N100.dat` with the trajectories of 100 tracers, where the integration step length is 0.005 and total path length is 5. The latter file is structured as blocks (one block per tracer) of space-separated data in the following order: tracer ID, time t, x, y, u_x, u_y, i.e. you need to plot column 3 and 4. In Gnuplot, this is straightforward:

```
pl 'cavity_data/trace_Re10_N100.dat' u 3:4 w l
```

♠ Plot streamlines of the flow for some values of Re < 1000 (changing Re amounts to changing the viscosity ν). How does the flow change?

Tip: You may have to increase the grid resolution N to achieve convergence.

References

[1] Wikipedia: Finite element method. https://en.wikipedia.org/wiki/Finite_element_method, 2016. Online; accessed 13 March 2016.

[2] Wikipedia: NACA airfoil. https://en.wikipedia.org/wiki/NACA_airfoil, 2016. Online; accessed 13 March 2016.

[3] Wikipedia: Stokes' paradox. https://en.wikipedia.org/wiki/Stokes%27_paradox, 2016. Online; accessed 13 March 2016.

[4] Utkarsh Ayachit. The ParaView guide: A parallel visualization application. *Kitware, Inc.*, 2015.

[5] Young W. Kwon and Hyochoong Bang. *The finite element method using MATLAB*. CRC press, 2000.

[6] Hans Petter Langtangen and Anders Logg. Solving PDEs in Minutes - The FEniCS Tutorial Volume I. https://fenicsproject.org/pub/tutorial/html/ftut1.html, 2016. Online; accessed 13 March 2016.

[7] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[8] Anders Logg, Garth N Wells, and Johan Hake. DOLFIN: A C++/Python finite element library. In *Automated Solution of Differential Equations by the Finite Element Method*, pages 173–225. Springer, 2012.