FEniCS hands-on

Release 2017.2.0.dev0

Jan Blechta, Roland Herzog, Jaroslav Hron, Gerd Wachsmuth

Sep 21, 2018

Preliminaries

1 License

2 Prolog

Motto

“My theory by A. Elk. Brackets Miss, brackets. This theory goes as follows and begins now. All brontosauruses are thin at one end, much much thicker in the middle and then thin again at the far end. That is my theory, it is mine, and belongs to me and I own it, and what it is too.”

– Anne Elk (Miss)

This document served primarily as task sheets for FEniCS hands-on lectures held on Chemnitz University of Technology in September 2018. Nevertheless it is not excluded that these sheets could not be used separately or for any other occassion.

2.1 Target audience

This tutorial gives lectures on usage of FEniCS version 2017.2.0 through its Python 3 user interface. It is specifically intended for newcomers to FEniCS and as such does not assume any knowledge in Python programming. Rather than taking a Python tutorial first, the intent is to learn-by-doing. As a consequence first steps consist of modificating existing FEniCS demos while gradually taking bigger and bigger tasks in writing original code.

2.2 FEniCS installation

Obviously we will need a working installation of FEniCS. FEniCS can be installed in different ways which all of them have some pros and cons. On TU Chemnitz this taken care of by organizers of the hands-on and participants do not have to worry about this.

Nevertheless participants might want to install FEnicsS to their laptops, workstation, home computers to practice or use FEniCS outside of the tutorial classes. The easiest option for new FEniCS users on Ubuntu is to install using APT from FEniCS PPA.

Note: This lecture material including the reference solutions is verified to be compatible with:

- FEniCS 2017.2.0,
- FEniCS 2018.1.0.

Ubuntu packages

Installing FEniCS (including mshr) from PPA:

```
sudo apt-get install --no-install-recommends software-properties-common
sudo add-apt-repository ppa:fenics-packages/fenics
sudo apt-get update
sudo apt-get install --no-install-recommends fenics
```

will install the following versions:

Ubuntu	FEniCS
Xenial 16.04	2017.2.0
Bionic 18.04	2018.1.0

On the other hand FEniCS 2017.2.0 can be installed on Bionic by

```
# Remove PPA if previously added
sudo apt-get install --no-install-recommends software-properties-common
sudo add-apt-repository --remove ppa:fenics-packages/fenics

# Install DOLFIN from official Bionic package
sudo apt-get update
sudo apt-get install --no-install-recommends python3-dolfin

# Optionally install mshr from source
sudo apt-get install libgmp-dev libmpfr-dev
wget https://bitbucket.org/fenics-project/mshr/downloads/mshr-2017.2.0.tar.gz
tar -xzf mshr-2017.2.0.tar.gz
```

(continues on next page)

(continued from previous page)

```
cd mshr-2017.2.0
mkdir build
cd build
cmake -DPYTHON_EXECUTABLE=/usr/bin/python3 ..
make
sudo make install
sudo ldconfig
```

Docker images

On the other hand FEniCS images for Docker provide the most portable solution, with arbitrary FEniCS version choice, for systems where Docker CE can be installed and run; see https://fenicsproject.org/download/.

2.3 Resources

- DOLFIN docs
- DOLFIN Python API docs
- UFL manual and API docs
- mshr API docs
- Python docs
- FEniCS AllAnswered

Periodic Table of Finite Elements

3 Poisson in a hundred ways

3.1 First touch

Login by SSH to `tyche` and type: **How to login**

Open a terminal window by hitting `CTRL+ALT+T`. Use `ssh` to connect to a remote system:

```
ssh -X -C tyche
```

Note:

- `-X` enable `X11` forwarding (allows processes on the remote machine opening windows of graphical applications on the local machine)
- `-C` enables compression which is mainly beneficial for access from a remote network
- `tyche` stands here for machine `tyche.mathematik.tu-chemnitz.de`; username on the local machine is used by default to login to the remote machine; the machine is not accessible from outside the univerity, so one would login through a jump host

```
luigivercotti@local_machine:~$ ssh -X -C user@login.tu-chemnitz.de
user@login:~$ ssh -X -C tyche
user@tyche:~$
```

Alternatively one can use VPN.

For experts: Kerberos + public key + jump host

The most comfortable solution for password-less logins outside of the university:

- add

```
Host login
   Hostname login.tu-chemnitz.de
   User <user>
   ForwardAgent yes
   ForwardX11 yes
   ForwardX11Trusted yes
   GSSAPIAuthentication yes
   GSSAPIDelegateCredentials yes

Host tyche
   Hostname tyche.mathematik.tu-chemnitz.de
   User <user>
   ForwardAgent yes
   ForwardX11 yes
   ForwardX11Trusted yes
   ProxyCommand ssh login -W %h:%p
```

  to `~/.ssh/config`,

- upload public key to `login` and `tyche` by

```
ssh-copy-id login
ssh-copy-id tyche
```

- install Kerberos

which allows to get Kerberos ticket by

```
kinit <user>@TU-CHEMNITZ.DE
```

and during its validity password-less login from anywhere

```
ssh tyche
```

```
source /LOCAL/opt/fenics-2017.2.0/fenics.conf
```

to prepare environment for using FEniCS. Now fire up interactive Python 3 interpreter:

```
python3
```

You should see something like:

```
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now type:

```
>>> from dolfin import *
>>> import matplotlib.pyplot as plt
>>>
>>> mesh = UnitSquareMesh(13, 8)
>>> plot(mesh)
[<matplotlib.lines.Line2D object at 0x7fe0003d65c0>, <matplotlib.lines.Line2D␣
↪object at 0x7fe0003d6748>]
>>> plt.show()
```

Hint: Click on >>> in the right top corner of the code snippet to make the code copyable.

A graphical plot of the mesh should appear. If any of the steps above failed, you're not correctly set up to use FEniCS. If everything went fine, close the plot window and hit `^D` to quit the interpreter.

3.2 Run and modify Poisson demo

Task 1

Get the Poisson demo from FEniCS install dir and run it:

```
mkdir -p work/fenics/poisson
cd work/fenics/poisson
cp /LOCAL/opt/fenics-2017.2.0/share/dolfin/demo/documented/poisson/python/demo_
↪poisson.py .
python3 demo_poisson.py
```

You should see some console output and a plot of the solution.

Now login to `tyche` from another terminal window and open the demo file using your favourite editor (if you don't have any you can use `gedit`, `nano`, . . .):

```
cd work/fenics/poisson
<editor> demo_poisson.py
```

Task 2

Now add keyword argument `warp='mode'` to the `plot` function call by applying the following diff:

```
 # Plot solution
 import matplotlib.pyplot as plt
-plot(u)
+plot(u, mode='warp')
 plt.show()
```

and run the demo again by `python3 demo_poisson.py`.

Hint

`Constant`, `Expression`, and similar are clickable links leading to their documentation.

Open Poisson demo documentation on the FEniCS website. Notice that the doc page is generated from the demo file. Go quickly through the docpage while paying attention to

- definition of weak formulation through forms `a` and `L`,
- usage of `Constant` and `Expression` classes.

Task 3

Modify the code to solve the following problem instead:

$$\begin{aligned} -\Delta u + cu &= f && \text{in } \Omega, \\ u &= u_\mathrm{D} && \text{on } \Gamma_\mathrm{D}, \\ \tfrac{\partial u}{\partial \mathbf{n}} &= g && \text{on } \Gamma_\mathrm{N} \end{aligned}$$

with

$$\Omega = (0,1)^2, \qquad \Gamma_\mathrm{D} = \{(x,y), x = 1, 0 < y < 1\}, \qquad \Gamma_\mathrm{N} = \partial\Omega \setminus \Gamma_\mathrm{D}, \tag{1}$$

$$c = 6, \qquad f(x,y) = x, \qquad u_\mathrm{D}(x,y) = y, \qquad g(x,y) = \sin(5x)\exp(y). \tag{2}$$

3.3 Semilinear Poisson equation

Task 4

Derive weak formulation for the following semilinear Poisson problem:

$$\begin{aligned} -\Delta u + u^3 + u &= f \quad \text{in } \Omega, \\ \tfrac{\partial u}{\partial \mathbf{n}} &= g \quad \text{on } \partial\Omega \end{aligned} \tag{3}$$

with

$$\Omega = (0,1)^2, \qquad f(x,y) = x, \qquad g(x,y) = \sin(5x)\exp(y). \tag{4}$$

Notice that the weak formulation has the form

Find $u \in H^1(\Omega)$ such that

$$F(u;v) = 0 \qquad \text{for all } v \in H^1(\Omega)$$

with certain F depending on u in nonlinear fashion but being linear in test functions v. One can find the solution iteratively by the Newton method:

1. Choose $u_0 \in H^1(\Omega)$,
2. For $k = 1, 2, \ldots$ do
   (a) Find $\delta u \in H^1(\Omega)$ such that

$$\frac{\partial F}{\partial u}(u_k; v, \delta u) = -F(u_k; v) \qquad \text{for all } v \in H^1(\Omega), \tag{5}$$

   (b) Set $u_{k+1} = u_k + \delta u$.
   (c) Check certain convergence criterion and eventually stop iterating.

Here Jacobian $\frac{\partial F}{\partial u}(u; v, \delta u)$ is Gâteaux derivative of F. It is generally nonlinear in u, but linear in v and δu. Hence with fixed $u_k \in H^1(\Omega)$ the left-hand side and the right-hand side of (5) are a bilinear and linear form respectively and (5) is just ordinary linear problem.

Task 5

Modify the previous code to adapt it to problem (3), (4). Define F by filing the gaps in the following code:

```
u = Function(V)
v = TestFunction(V)
f = Expression(...)
g = Expression(...)

F = ...
```

If in doubts, peek into Nonlinear Poisson demo documentation.

Look into documentation of `solve` function, read section *Solving nonlinear variational problems*. Now you should be able to call the `solve` function to obtain the solution.

3.4 Nonlinear Dirichlet problem

Task 6

Modify the code to solve the following Dirichlet problem:

$$\begin{aligned} -\operatorname{div}(c\nabla u) + 10u^3 + u &= f && \text{in } \Omega, \\ u &= u_{\mathrm{D}} && \text{on } \partial\Omega \end{aligned}$$

with

$$\Omega = (0,1)^2, \qquad f(x,y) = 100x, \qquad u_{\mathrm{D}}(x,y) = y, \qquad c(x,y) = \tfrac{1}{10} + \tfrac{1}{2}(x^2+y^2).$$

Hint: Supply instance of `SubDomain` class to `DirichletBC`. How do you tell `SubDomain` to define $\partial\Omega$? What do you fill in?

```python
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return ...
```

`on_boundary` argument evaluates to `True` on boundary facets, `False` otherwise.

3.5 Variational formulation

For $u \in H^1(\Omega)$ consider functional

$$E(u) = \int_\Omega \left(\tfrac{1}{2}|\nabla u|^2 + \tfrac{1}{4}u^4 + \tfrac{1}{2}u^2 - fu\right) \mathrm{d}x - \int_{\partial\Omega} gu \, \mathrm{d}s.$$

Convince yourself that minimization of F over $H^1(\Omega)$ is equivalent to problem (3).

Task 7

By filling the following code:

```
u = Function(V)
f = Expression(...)
g = Expression(...)

E = ...
```

define $E(u)$ for data (4). Remember that functionals (zero-forms) do not have any test and trial functions.

Obtain $F(u;v) := \frac{\partial E}{\partial u}(u;v)$ using `derivative`:

```
F = derivative(E, u)
```

and run the solver like in *Task 5*. Check you get the same solution.

3.6 Yet another nonlinearity

Consider quasilinear equation in divergence form

$$\begin{aligned} -\operatorname{div}(\mathcal{A}\nabla u) + u &= f && \text{in } \Omega, \\ \frac{\partial u}{\partial \mathcal{A}^\top \mathbf{n}} &= 0 && \text{on } \partial\Omega, \\ \mathcal{A} &= \left[\tfrac{1}{10} + u^2 \quad 00 \quad 1+u^2\right] && \text{in } \Omega \end{aligned} \tag{6}$$

with data

$$\Omega = (0,1)^2, \qquad f(x,y) = \tfrac{1}{2}(x+y). \tag{7}$$

Task 8

Derive weak formulation for the problem (6).

Solve the problem (6), (7) using FEniCS. Employ `as_matrix` function to define $\mathcal{A}$:

```
u = Function(V)
v = TestFunction(V)

A = as_matrix((
    (..., ...),
    (..., ...),
))
F = inner(A*grad(u), grad(v))*dx + ...
```

3.7 Reference solution

Show/Hide Code

Download Code

```
from dolfin import *
import matplotlib.pyplot as plt


def solve_task3(V):
    """Return solution of Task 3 on space V"""

    # Define Dirichlet boundary (x = 0 or x = 1)
    def boundary(x, on_boundary):
        return on_boundary and x[0] > 1.0 - DOLFIN_EPS

    # Define boundary condition
    uD = Expression("x[1]", degree=1)
    bc = DirichletBC(V, uD, boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    f = Expression("x[0]", degree=1)
    g = Expression("sin(5*x[0])*exp(x[1])", degree=3)
    a = inner(grad(u), grad(v))*dx + 6*u*v*dx
    L = f*v*dx + g*v*ds

    # Compute solution
    u = Function(V)
    solve(a == L, u, bc)

    return u


def solve_task5(V):
    """Return solution of Task 5 on space V"""

    # Define variational problem
```

(continues on next page)

(continued from previous page)

```
    u = Function(V)
    v = TestFunction(V)
    f = Expression("x[0]", degree=1)
    g = Expression("sin(5*x[0])*exp(x[1])", degree=3)
    F = inner(grad(u), grad(v))*dx + (u**3 + u)*v*dx - f*v*dx - g*v*ds

    # Compute solution
    solve(F == 0, u)

    return u


def solve_task6(V):
    """Return solution of Task 6 on space V"""

    # Define Dirichlet boundary
    class Boundary(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary

    # Define boundary condition
    boundary = Boundary()
    uD = Expression("x[1]", degree=1)
    bc = DirichletBC(V, uD, boundary)

    # Define variational problem
    u = Function(V)
    v = TestFunction(V)
    c = Expression("0.1 + 0.5*(x[0]*x[0] + x[1]*x[1])", degree=2)
    f = Expression("100*x[0]", degree=1)
    F = c*inner(grad(u), grad(v))*dx + (10*u**3 + u)*v*dx - f*v*dx

    # Compute solution
    solve(F == 0, u, bc)

    return u


def solve_task7(V):
    """Return solution of Task 7 on space V"""

    # Define variational problem
    u = Function(V)
    f = Expression("x[0]", degree=1)
    g = Expression("sin(5*x[0])*exp(x[1])", degree=3)
    E = ( grad(u)**2/2 + u**4/4 + u**2/2 - f*u )*dx - g*u*ds
    F = derivative(E, u)

    # Compute solution
    solve(F == 0, u)

    return u


def solve_task8(V):
    """Return solution of Task 8 on space V"""

    # Define variational problem
    u = Function(V)
    v = TestFunction(V)
    f = Expression("0.5*(x[0] + x[1])", degree=1)
```

(continues on next page)

(continued from previous page)

```
    A = as_matrix((
        (0.1 + u**2,          0),
        (          0, 1 + u**2),
    ))
    F = inner(A*grad(u), grad(v))*dx + u*v*dx - f*v*dx

    # Compute solution
    solve(F == 0, u)

    return u


if __name__ == '__main__':

    # Create mesh and define function space
    mesh = UnitSquareMesh(32, 32)
    V = FunctionSpace(mesh, "Lagrange", 1)

    # Solve all problems
    u3 = solve_task3(V)
    u5 = solve_task5(V)
    u6 = solve_task6(V)
    u7 = solve_task7(V)
    u8 = solve_task8(V)

    # Compare solution which should be same
    err = ( grad(u7 - u5)**2 + (u7 - u5)**2 )*dx
    err = assemble(err)
    print("||u7 - u5||_H1 =", err)

    # Plot all solutions into separate figures
    plt.figure()
    plot(u3, title='u3', mode="warp")

    plt.figure()
    plot(u5, title='u5', mode="warp")

    plt.figure()
    plot(u6, title='u6', mode="warp")

    plt.figure()
    plot(u7, title='u7', mode="warp")

    plt.figure()
    plot(u8, title='u8', mode="warp")

    # Display all plots
    plt.show()
```

4 Heat equation

Goals

Learn how to deal with time-dependent problems. Solve heat equation by θ-scheme. Solve wave equation with central differences. Plot some nice figures.

We will be interested in solving heat equation:

$$\begin{aligned} u_t - \Delta u &= f && \text{in } \Omega \times (0,T), \\ \tfrac{\partial u}{\partial \mathbf{n}} &= g && \text{on } \partial\Omega \times (0,T), \\ u &= u_0 && \text{on } \Omega \times \{0\} \end{aligned}$$

using θ-scheme discretization in time and arbitrary FE discretization in space with given data f, g, u_0. θ-scheme time-discrete heat equation reads:

$$\begin{aligned} \frac{1}{\Delta t}\Big(u^{n+1} - u^n\Big) - \theta\Delta u^{n+1} - (1-\theta)\Delta u^n &= \theta f(t_{n+1}) + (1-\theta) f(t_n) && \text{in } \Omega,\ n = 0,1,2,\dots \\ \tfrac{\partial u^n}{\partial \mathbf{n}} &= g(t_n) && \text{on } \partial\Omega,\ n = 0,1,2,\dots \\ u^0 &= u_0 && \text{in } \Omega \end{aligned} \tag{8}$$

for a certain sequence $0 = t_0 < t_1 < t_2 < ... \leq T$. Special cases are:

$\theta = 0$	explicit Euler scheme,
$\theta = \frac{1}{2}$	Crank-Nicolson scheme,
$\theta = 1$	implicit Euler scheme.

Task 1

Test (8) by functions from $H^1(\Omega)$ and derive a weak formulation of θ-scheme for heat equation.

4.1 First steps

Consider data

$$\begin{aligned} \Omega &= (0,1)^2, \\ T &= 2, \\ f &= 0, \\ g &= 0, \\ u_0(x,y) &= x. \end{aligned} \tag{9}$$

Task 2

Write FEniCS code implementing problem (8), (9), assuming general θ, and arbitrary but fixed Δt. In particular assume:

```
from dolfin import *

mesh = UnitSquareMesh(32, 32)
V = FunctionSpace(mesh, "Lagrange", 1)

theta = Constant(0.5)
dt = Constant(0.1)
```

Proceed step-by-step.

1. **Define all relevant data from** (9). Use `Constant` or `Expression` classes to define f, g, u_0.
2. Define a finite element function for holding solution at a particular time step:

   ```
   u_n = Function(V)
   ```

   and arguments of linear and bilinear forms:

```
u, v = TrialFunction(V), TestFunction(V)
```

3. **Define bilinear and linear forms describing Galerkin descretization of the weak formulation derived in** *Task 1* **on the space** `V`.

   You can conveniently mix bilinear and linear terms into a single expression:

   ```
   F = 1/dt*(u - u_n)*v*dx + ...
   ```

   and separate bilinear and linear part using `lhs`, `rhs`:

   ```
   a, L = lhs(F), rhs(F)
   ```

   Tip: It is good to execute your code every once in a while, even when it is not doing anything useful so far, e.g., does not have time-stepping yet. You will catch the bugs early and fix them easily.

4. **Prepare for the beggining of time-stepping.** Assume `u0` is an `Expression` or `Constant`. You can use `Function.interpolate()` or `interpolate()`:

   ```
   u_n.interpolate(u0)
   # or
   u_n = interpolate(u0, V)
   ```

5. **Implement time-stepping.** Write a control flow statement (for example a `while` loop) which executes the solver for problem `a == L` repeatedly while updating what needed.

   Hint: Note that a single `Function` object is needed to implement the time-stepping. The function can be used to hold the value of u_n and then be updated by calling `solve(...)`.

6. **Run with different values of $\theta = 1, \frac{1}{2}, 0$.**

   As a first indicator of correctness of the implementation you can drop into the loop lines like:

   ```
   energy = assemble(u_n*dx)
   print("Energy =", energy)
   ```

   Are you observing expected value?

4.2 Data IO, plotting

There are several possibilities for visualization of data.

XDMF output and Paraview

One possibility is to use IO facilities of FEniCS and visualize using external software, for example Paraview.

Note: This approach allows to separate

- actual computation, which can happen in headless HPC environment, for example big parallel clusters of thousands of CPU cores,
- and visualization, which many times needs human interaction.

One can used `XDMFFile` to store data:

```
# Open file for XDMF IO
f = XDMFFile('solution.xdmf')

while t < T:

    # Compute time step
    perform_timestep(u_n, t, dt)
    t += dt

    # Save the result to file at time t
    f.write(u_n, t)
```

Then you can open Paraview by shell command

```
paraview &
```

and visualize the file `solution.xdmf`.

Matplotlib – native plotting in Python

Another possibility is to use Python plotting library Matplotlib.

Note: Matplotlib is Python native plotting library, which is programmable and supports

- interactive use from Python interpreters, including popular shells like Jupyter,
- high-quality vector output suitable for scientific publishing.

FEniCS `plot(obj, **kwargs)` function implements plotting using Matplotlib for several different types of `obj`, for instance `Function`, `Expression`, `Mesh`, `MeshFunction`. As Matplotlib is highly programmable and customizable, FEniCS `plot()` is typically accompanied by some native matplotlib commands. Mimimal example of interaction of FEniCS and matplotlib:

```
from dolfin import *
import matplotlib.pyplot as plt

mesh = UnitSquareMesh(64, 64)
plot(mesh)
plt.savefig('mesh_64_64.pdf')  # Render to PDF
plt.show()  # Render into interactive window
```

Add something along the lines of:

```
import matplotlib.pyplot as plt

# Open a plot window
fig = plt.figure()
fig.show()

while t < T:

    # Compute time step
    perform_timestep(u_n, t, dt)
    t += dt

    # Update plot to current time step
    fig.clear()
    p = plot(u_n, mode="warp")
    fig.colorbar(p)
    fig.gca().set_zlim((0, 2))
    fig.canvas.draw()
```

Warning: Matplotlib's interactive capabalities aparently depend on used Matplotlib backend. In particular updating the contents of the plot window seems to work fine with `TkAgg` backend. Issue shell command

```
export MPLBACKEND=tkagg
```

to choose `TkAgg` in the current shell session.

Task 3

Implement at least one of the aforementioned ways to plot your solutions in time. Check that your solution of *Task 2* looks reasonable.

4.3 Nonhomogeneous Neumann BC

Consider (8), (9) but now with nonhomogeneous Neumann data

$$\begin{aligned} g &= 1 \text{ on } \{x = 0\}, \\ g &= 0 \text{ elsewhere.} \end{aligned} \tag{10}$$

Task 3

1. Derive weak formulation describing (8), (9), (10).
2. Define surface measure supported on the left boundary of the unit square mesh by following steps:
   (a) subclass `SubDomain`,
   (b) define `MeshFunction`,
   (c) mark the mesh function using `SubDomain.mark` method,
   (d) define integration `Measure`.

Hint:

Show/Hide Code

```
# Define instance of SubDomain class
class Left(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 0)
left = Left()

# Define and mark mesh function on facets
facets = MeshFunction('size_t', mesh, mesh.topology().dim()-1)
left.mark(facets, 1)

# Define exterior facet measure where facets==1
ds_left = Measure("ds", mesh, subdomain_data=facets, subdomain_id=1)
```

---

3. Using the surface measure, modify the implementation from *Task 2* to incorporate boundary condition (10).
4. Run the code with $\theta = 1$ and check that the results look as expected.

4.4 Time-dependent BC

Consider time-dependent data

$$\begin{aligned} f(x,t) &= 2-t, \\ g(x,t) &= \begin{cases} t & x = 0,0 \\ \text{otherwise.} \end{cases} \end{aligned} \tag{11}$$

Task 4

Modify solution of the previous task to use data (11).

Hint: You can use `Constant.assign()` or `Expression.<param> = <value>` to change existing `Constant` or `Expression`. Look for *User defined parameters* in `Expression` documentation.

Now consider different time-dependent data

$$\begin{aligned} f(x,t) &= 0, \\ g(x,t) &= \begin{cases} \max\big(0, \frac{1-t}{2}\big) & x = 0,0 \\ \text{otherwise.} \end{cases} \end{aligned} \tag{12}$$

Task 5

Modify solution of the previous task to use data (12).

4.5 Adaptive time-stepping

Consider solution of *low* precision generated by timestep Δt:

$$\frac{1}{\Delta t}\Big(u_{\text{low}}^{n+1} - u^n\Big) - \theta\Delta u_{\text{low}}^{n+1} - (1-\theta)\Delta u^n = \theta f(t_{n+1}) + (1-\theta) f(t_n) \tag{13}$$

and solution of *high* precision computed by two timesteps of a half size:

$$\begin{aligned} \frac{1}{\Delta t/2}\Big(u_{\text{high}}^{n+1/2} - u^n\Big) - \theta\Delta u_{\text{high}}^{n+1/2} - (1-\theta)\Delta u^n &= \theta f(t_{n+1/2}) + (1-\theta) f(t_n), \\ \frac{1}{\Delta t/2}\Big(u_{\text{high}}^{n+1} - u_{\text{high}}^{n+1/2}\Big) - \theta\Delta u_{\text{high}}^{n+1} - (1-\theta)\Delta u_{\text{high}}^{n+1/2} &= \theta f(t_{n+1}) + (1-\theta) f(t_{n+1/2}). \end{aligned} \tag{14}$$

By Richardson extrapolation one can estimate the error of discretization (in time) by quantity:

$$\eta := \frac{\|u_{\text{high}}^{n+1} - u_{\text{low}}^{n+1}\|_{L^2(\Omega)}}{2^p - 1} \tag{15}$$

where

$$p = \begin{cases} 2 \\ \theta = \frac{1}{2},\ 1 \\ \text{otherwise} \end{cases} \tag{16}$$

is a theoretical order of accuracy of the θ-scheme. Given a tolerance Tol set the new timestep to

$$\Delta t^* := \left(\frac{\rho\,\mathrm{Tol}}{\eta}\right)^{\frac{1}{p}} \Delta t. \tag{17}$$

Here $0 < \rho \le 1$ is a chosen safety factor. That asymptotically ensures that the error (or at least the estimator) committed with the new time step is ρ-multiple of the tolerance.

Now consider an algorithm:

1. compute u^{n+1}_{low} and u^{n+1}_{high}
2. compute η
3. compute Δt^*
4. if $\eta \leq \text{Tol}$:
   $u^{n+1} := u^{n+1}_{\text{high}}$
   $n \mathrel{+}= 1$
5. update timestep $\Delta t := \Delta t^*$

Task 6

Solve (8), $(9)_{1,2,5}$, (12) using the adaptive strategy described above.

Hint: You will need more than one `Function` and perform assignments between them. Having `Functions` `f`, `g` on the same space you can perform assignment $f := g$ by

```
f.vector()[:] = g.vector()
```

4.6 Wave equation

Now consider problem

$$\begin{aligned} u_{tt} - \Delta u &= f && \text{in } \Omega \times (0,T), \\ u &= 0 && \text{on } \partial\Omega \times (0,T), \\ u(\cdot,t) &= u_0 && \text{in } \Omega, \\ u_t(\cdot,t) &= v_0 && \text{in } \Omega. \end{aligned} \tag{18}$$

This problem can be discretized in time as

$$\frac{1}{(\Delta t)^2}\Big(u^{n+1} - 2u^n + u^{n-1}\Big) - \tfrac{1}{2}\Delta(u^{n+1} + u^{n-1}) = f(t^n). \tag{19}$$

The iteration can be bootstrapped by

$$u^1 := u^0 + \Delta t v^0.$$

Task 7

Implement solver for problem (18) by discretizing in time with (19). Solve the problem with data

$$\begin{aligned} f &= 0, \\ u^0(x,y) &= \max(0, 1 - 4r(x,y)) \qquad \text{where } r(x,y) = \operatorname{dist}((x,y), (\tfrac{1}{2}, \tfrac{3}{10})), \\ v^0(x,y) &= 0, \\ T &= 5, \\ \Omega &= (0,1)^2. \end{aligned}$$

Visualize the result.

4.7 Reference solution

Note: The reference solution follows the DRY principle. Hands-on participants are not expected to write such a structured code during the session.

Attention: For on-the-fly plotting, `TkAgg` Matplotlib backend has been tested. You can enforce its selection by a shell command

```
export MPLBACKEND=tkagg
```

Note that the the plotting is the bottleneck of the code. The code runs much faster without plots which can be ensured by

```
DOLFIN_NOPLOT=1 MPLBACKEND=template python3 heat.py
```

We leave as an exercise to add XDMF output for plotting in Paraview.

Show/Hide Code

Download Code

```python
from dolfin import *
import matplotlib.pyplot as plt


def create_timestep_solver(get_data, dsN, theta, u_old, u_new):
    """Prepare timestep solver by theta-scheme for given
    function get_data(t) returning data (f(t), g(t)), given
    solution u_old at time t and unknown u_new at time t + dt.
    Return a solve function taking (t, dt).
    """

    # Initialize coefficients
    f_n, g_n = get_data(0)
    f_np1, g_np1 = get_data(0)
    idt = Constant(0)

    # Extract function space
    V = u_new.function_space()

    # Prepare weak formulation
    u, v = TrialFunction(V), TestFunction(V)
    theta = Constant(theta)
    F = ( idt*(u - u_old)*v*dx
        + inner(grad(theta*u + (1-theta)*u_old), grad(v))*dx
        - (theta*f_np1 + (1-theta)*f_n)*v*dx
        - (theta*g_np1 + (1-theta)*g_n)*v*dsN
    )
    a, L = lhs(F), rhs(F)

    def solve_(t, dt):
        """Update problem data to interval (t, t+dt) and
        run the solver"""

        # Update coefficients to current t, dt
        get_data(t, (f_n, g_n))
        get_data(t+dt, (f_np1, g_np1))
        idt.assign(1/dt)

        # Push log level
```

(continues on next page)

(continued from previous page)

```
        old_level = get_log_level()
        warning = LogLevel.WARNING if cpp.__version__ > '2017.2.0' else WARNING
        set_log_level(warning)

        # Run the solver
        solve(a == L, u_new)

        # Pop log level
        set_log_level(old_level)

    return solve_


def timestepping(V, dsN, theta, T, dt, u_0, get_data):
    """Perform timestepping using theta-scheme with
    final time T, timestep dt, initial datum u_0 and
    function get_data(t) returning (f(t), g(t))"""

    # Initialize solution function
    u = Function(V)

    # Prepare solver for computing time step
    solver = create_timestep_solver(get_data, dsN, theta, u, u)

    # Set initial condition
    u.interpolate(u_0)

    # Open plot window
    fig = init_plot()

    # Print table header
    print("{:10s} | {:10s} | {:10s}".format("t", "dt", "energy"))

    # Perform timestepping
    t = 0
    while t < T:

        # Report some numbers
        energy = assemble(u*dx)
        print("{:10.4f} | {:10.4f} | {:#10.4g}".format(t, dt, energy))

        # Perform time step
        solver(t, dt)
        t += dt

        # Update plot
        update_plot(fig, u)


def timestepping_adaptive(V, dsN, theta, T, tol, u_0, get_data):
    """Perform adaptive timestepping using theta-scheme with
    final time T, tolerance tol, initial datum u_0 and
    function get_data(t) returning (f(t), g(t))"""

    # Initialize needed functions
    u_n = Function(V)
    u_np1_low = Function(V)
    u_np1_high = Function(V)

    # Prepare solvers for computing tentative time steps
    solver_low = create_timestep_solver(get_data, dsN, theta, u_n, u_np1_low)
```

(continues on next page)

(continued from previous page)

```
    solver_high_1 = create_timestep_solver(get_data, dsN, theta, u_n, u_np1_high)
    solver_high_2 = create_timestep_solver(get_data, dsN, theta, u_np1_high, u_np1_
↪high)

    # Initial time step; the value does not really matter
    dt = T/2

    # Set initial conditions
    u_n.interpolate(u_0)

    # Open plot window
    fig = init_plot()

    # Print table header
    print("{:10s} | {:10s} | {:10s}".format("t", "dt", "energy"))

    # Perform timestepping
    t = 0
    while t < T:

        # Report some numbers
        energy = assemble(u_n*dx)
        print("{:10.4f} | {:10.4f} | {:#10.4g}".format(t, dt, energy))

        # Compute tentative time steps
        solver_low(t, dt)
        solver_high_1(t, dt/2)
        solver_high_2(t+dt, dt/2)

        # Compute error estimate and new timestep
        est = compute_est(theta, u_np1_low, u_np1_high)
        dt_new = compute_new_dt(theta, est, tol, dt)

        if est > tol:
            # Tolerance not met; repeat the step with new timestep
            dt = dt_new
            continue

        # Move to next time step
        u_n.vector()[:] = u_np1_high.vector()
        t += dt
        dt = dt_new

        # Update plot
        update_plot(fig, u_n)


def compute_est(theta, u_L, u_H):
    """Return error estimate by Richardson extrapolation"""
    p = 2 if theta == 0.5 else 1
    est = sqrt(assemble((u_L - u_H)**2*dx)) / (2**p - 1)
    return est


def compute_new_dt(theta, est, tol, dt):
    """Return new time step"""
    p = 2 if theta == 0.5 else 1
    rho = 0.9
    dt_new = dt * ( rho * tol / est )**(1/p)
    return dt_new

```

(continues on next page)

(continued from previous page)

```python
def init_plot():
    """Open plot window and return its figure object"""
    fig = plt.figure()
    fig.show()
    return fig


def update_plot(fig, u, zlims=(0, 2)):
    """Plot u in 3D warp mode with colorbar into figure fig;
    use zlims as limits on z-axis"""
    fig.clear()
    p = plot(u, mode="warp")
    if p is None:
        return
    fig.colorbar(p)
    fig.gca().set_zlim(zlims)
    fig.canvas.draw()


def create_function_space():
    """Return (arbitrary) H^1 conforming function space on
    unit square domain"""
    mesh = UnitSquareMesh(32, 32)
    V = FunctionSpace(mesh, "P", 1)
    return V


def create_surface_measure_left(mesh):
    """Return surface measure on the left boundary of unit
    square"""
    class Left(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary and near(x[0], 0)
    facets = MeshFunction('size_t', mesh, mesh.topology().dim()-1)
    Left().mark(facets, 1)
    ds_left = Measure("ds", mesh, subdomain_data=facets, subdomain_id=1)
    return ds_left


def get_data_2(t, result=None):
    """Create or update data for Task 2"""
    f, g = result or (Constant(0), Constant(0))
    f.assign(0)
    g.assign(0)
    return f, g


def get_data_3(t, result=None):
    """Create or update data for Task 3"""
    f, g = result or (Constant(0), Constant(0))
    f.assign(1)
    g.assign(0)
    return f, g


def get_data_4(t, result=None):
    """Create or update data for Task 4"""
    f, g = result or (Constant(0), Constant(0))
    f.assign(2-t)
    g.assign(t)
```

(continues on next page)

(continued from previous page)

```python
    return f, g


def get_data_5(t, result=None):
    """Create or update data for Task 5 and Task 6"""
    f, g = result or (Constant(0), Constant(0))
    f.assign(0)
    g.assign(max(0, 1-t)/2)
    return f, g


if __name__ == '__main__':
    # Common data
    V = create_function_space()
    ds_left = create_surface_measure_left(V.mesh())
    T = 2
    u_0 = Expression("x[0]", degree=1)

    # Run all problems
    theta = 1
    print("Task 2, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_2)

    theta = 1/2
    print("Task 2, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_2)

    theta = 0
    print("Task 2, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_2)

    theta = 1
    print("Task 3, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_3)

    theta = 1
    print("Task 4, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_4)

    theta = 1
    print("Task 5, theta =", theta)
    dt = 0.1
    timestepping(V, ds_left, theta, T, dt, u_0, get_data_5)

    theta = 1
    print("Task 6, theta =", theta)
    tol = 1e-3
    timestepping_adaptive(V, ds_left, theta, T, tol, u_0, get_data_5)

    theta = 1/2
    print("Task 6, theta =", theta)
    tol = 1e-3
    timestepping_adaptive(V, ds_left, theta, T, tol, u_0, get_data_5)

    # Hold plots before quitting
    plt.show()
```

5 Navier-Stokes equations

Goals

Learn how to deal with mixed finite elements. Remember how fragile can numerical solutions be. Reproduce some cool physics – Kármán vortex street.

5.1 Stokes flow around cylinder

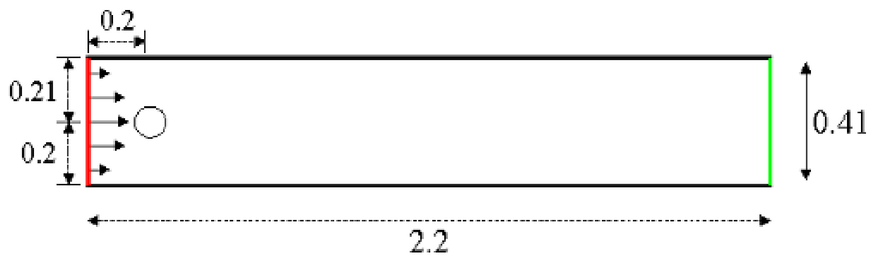
Solve the following linear system of PDEs

$$
\begin{aligned}
-\nu\Delta\mathbf{u} + \nabla p &= \mathbf{0} && \text{in } \Omega,\\
\operatorname{div}\mathbf{u} &= 0 && \text{in } \Omega,\\
\mathbf{u} &= 0 && \text{on } \Gamma_\mathrm{D},\\
\mathbf{u} &= \mathbf{u}_\mathrm{IN} && \text{on } \Gamma_\mathrm{IN},\\
\nu\frac{\partial\mathbf{u}}{\partial\mathbf{n}} - p\mathbf{n} &= 0 && \text{on } \Gamma_\mathrm{N}
\end{aligned}
\tag{20}
$$

using FE discretization with data

$$
\begin{aligned}
\Omega &= (0,2.2)\times(0,0.41) - B_{0.05}\left((0.2,0.2)\right),\\
\Gamma_\mathrm{N} &= \{x=2.2\} = \text{(green)},\\
\Gamma_\mathrm{IN} &= \{x=0.0\} = \text{(red)},\\
\Gamma_\mathrm{D} &= \partial\Omega\setminus(\Gamma_\mathrm{N}\cup\Gamma_\mathrm{IN}) = \text{(black)},\\
u_\mathrm{IN} &= \left(\frac{4Uy(0.41-y)}{0.41^2}, 0\right),\\
\nu &= 0.001, \qquad U = 0.3
\end{aligned}
\tag{21}
$$

where $B_R(\mathbf{z})$ is a disc of radius R and center $\mathbf{z}$



Task 1

Write the weak formulation of the problem and a spatial discretization by a mixed finite element method.

Task 2

Build a mesh, prepare a mesh function marking Γ_IN, Γ_N and Γ_D and plot it to check its correctness.

Hint: Use the FEniCS meshing tool `mshr`, see mshr documentation.

```
from dolfin import *
import mshr

# Discretization parameters
```

(continues on next page)

(continued from previous page)

```
N_circle = 16
N_bulk = 64

# Define domain
center = Point(0.2, 0.2)
radius = 0.05
L = 2.2
W = 0.41
geometry =  mshr.Rectangle(Point(0.0, 0.0), Point(L, W)) \
           -mshr.Circle(center, radius, N_circle)

# Build mesh
mesh = mshr.generate_mesh(geometry, N_bulk)
```

Hint: Try yet another way to mark the boundaries by direct access to the mesh entities by `vertices(mesh)`, `facets(mesh)`, `cells(mesh)` mesh-entity iterators:

```
# Construct facet markers
bndry = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
for f in facets(mesh):
    mp = f.midpoint()
    if near(mp[0], 0.0):  # inflow
        bndry[f] = 1
    elif near(mp[0], L):  # outflow
        bndry[f] = 2
    elif near(mp[1], 0.0) or near(mp[1], W):  # walls
        bndry[f] = 3
    elif mp.distance(center) <= radius:  # cylinder
        bndry[f] = 5

# Dump facet markers to file to plot in Paraview
with XDMFFile('facets.xdmf') as f:
    f.write(bndry)
```

Task 3

Construct the mixed finite element space and the bilinear and linear forms together with appropriate `DirichletBC` object.

Hint: Use for example the stable Taylor-Hood finite elements:

```
# Build function spaces (Taylor-Hood)
P2 = VectorElement("P", mesh.ufl_cell(), 2)
P1 = FiniteElement("P", mesh.ufl_cell(), 1)
TH = MixedElement([P2, P1])
W = FunctionSpace(mesh, TH)
```

Hint: To define Dirichlet BC on subspace use the `W.sub()` method:

```
bc_walls = DirichletBC(W.sub(0), (0, 0), bndry, 3)
```

Hint: To build the forms use:

```
# Define trial and test functions
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
```

Then you can define forms on mixed space using `u`, `p`, `v`, `q` as usual.

5.2 Steady Navier-Stokes flow

Task 4

Modify the problem into the Navier-Stokes equations given by

$$-\nu\Delta\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u} + \nabla p = 0 \quad \text{in } \Omega \tag{22}$$

together with (20)$_2$–(20)$_5$. Compute the DFG-flow around cylinder benchmark 2D-1, laminar case, Re=20 given by (22), (20)$_2$–(20)$_5$, (21).

Hint: As usual get rid of `TrialFunctions` in favour of nonlinear dependence on `Function`. You can split a `Function` on a mixed space into components:

```
w = Function(W)
u, p = split(w)

F = nu*inner(grad(u), grad(v))*dx + ...
```

Task 5

Add computation of lift and drag coefficients C_D, C_L and pressure difference p_diff as defined on the DFG 2D-1 website.

Hint: Use `assemble` function to evaluate the lift and drag functionals.

Use either `Function.split()` or `Function.sub()` to extract pressure `p` from solution `w` for evaluation. Evaluate the pressure `p` at point `a = Point(234, 567)` by calling `p(a)`.

Task 6

Check computed pressure difference and lift/drag coefficents against the reference. Investigate if/how the lift coefficent is sensitive to changes in the discretization parameters – conduct a convergence study.

5.3 Kármán vortex street

Task 7

Consider evolutionary Navier-Stokes equations

$$u_t - \nu\Delta\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u} + \nabla p = 0. \tag{23}$$

Prepare temporal discretization using *the Crank-Nicolson scheme* to compute a solution of (23), $(20)_2$–$(20)_5$, (21) on time interval $(0, 8)$ but use

$$U = 1$$

instead of $(21)_{6b}$. Plot the transient solution.

5.4 Reference solution

Note: You can run FEniCS codes in parallel (using MPI) by

```
mpirun -n <np> python3 <yourscript>.py
```

where for <np> substitute number of processors to use.

To benefit from parallism you can run the unsteady Navier-Stokes part of the code below on, say, eight cores:

```
mpirun -n 8 python3 -c"import dfg; dfg.task_7()"
```

Show/Hide Code

Download Code

```
from dolfin import *
import mshr
import matplotlib.pyplot as plt


def build_space(N_circle, N_bulk, u_in):
    """Prepare data for DGF benchmark. Return function
    space, list of boundary conditions and surface measure
    on the cylinder."""

    # Define domain
    center = Point(0.2, 0.2)
    radius = 0.05
    L = 2.2
    W = 0.41
    geometry = mshr.Rectangle(Point(0.0, 0.0), Point(L, W)) \
             - mshr.Circle(center, radius, N_circle)

    # Build mesh
    mesh = mshr.generate_mesh(geometry, N_bulk)

    # Construct facet markers
    bndry = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
    for f in facets(mesh):
        mp = f.midpoint()
        if near(mp[0], 0.0):  # inflow
            bndry[f] = 1
        elif near(mp[0], L):  # outflow
            bndry[f] = 2
        elif near(mp[1], 0.0) or near(mp[1], W):  # walls
            bndry[f] = 3
        elif mp.distance(center) <= radius:  # cylinder
            bndry[f] = 5

    # Build function spaces (Taylor-Hood)
```

(continues on next page)

(continued from previous page)

```
    P2 = VectorElement("P", mesh.ufl_cell(), 2)
    P1 = FiniteElement("P", mesh.ufl_cell(), 1)
    TH = MixedElement([P2, P1])
    W = FunctionSpace(mesh, TH)

    # Prepare Dirichlet boundary conditions
    bc_walls = DirichletBC(W.sub(0), (0, 0), bndry, 3)
    bc_cylinder = DirichletBC(W.sub(0), (0, 0), bndry, 5)
    bc_in = DirichletBC(W.sub(0), u_in, bndry, 1)
    bcs = [bc_cylinder, bc_walls, bc_in]

    # Prepare surface measure on cylinder
    ds_circle = Measure("ds", subdomain_data=bndry, subdomain_id=5)

    return W, bcs, ds_circle


def solve_stokes(W, nu, bcs):
    """Solve steady Stokes and return the solution"""

    # Define variational forms
    u, p = TrialFunctions(W)
    v, q = TestFunctions(W)
    a = nu*inner(grad(u), grad(v))*dx - p*div(v)*dx - q*div(u)*dx
    L = inner(Constant((0, 0)), v)*dx

    # Solve the problem
    w = Function(W)
    solve(a == L, w, bcs)

    return w


def solve_navier_stokes(W, nu, bcs):
    """Solve steady Navier-Stokes and return the solution"""

    # Define variational forms
    v, q = TestFunctions(W)
    w = Function(W)
    u, p = split(w)
    F = nu*inner(grad(u), grad(v))*dx + dot(dot(grad(u), u), v)*dx \
        - p*div(v)*dx - q*div(u)*dx

    # Solve the problem
    solve(F == 0, w, bcs)

    return w


def solve_unsteady_navier_stokes(W, nu, bcs, T, dt, theta):
    """Solver unsteady Navier-Stokes and write results
    to file"""

    # Current and old solution
    w = Function(W)
    u, p = split(w)

    w_old = Function(W)
    u_old, p_old = split(w_old)

    # Define variational forms
```

(continues on next page)

(continued from previous page)

```
    v, q = TestFunctions(W)
    F = ( Constant(1/dt)*dot(u - u_old, v)
          + Constant(theta)*nu*inner(grad(u), grad(v))
          + Constant(theta)*dot(dot(grad(u), u), v)
          + Constant(1-theta)*nu*inner(grad(u), grad(v))
          + Constant(1-theta)*dot(dot(grad(u_old), u_old), v)
          - p*div(v)
          - q*div(u)
        )*dx
    J = derivative(F, w)

    # Create solver
    problem = NonlinearVariationalProblem(F, w, bcs, J)
    solver = NonlinearVariationalSolver(problem)
    solver.parameters['newton_solver']['linear_solver'] = 'mumps'

    f = XDMFFile('velocity_unteady_navier_stokes.xdmf')
    u, p = w.split()

    # Perform time-stepping
    t = 0
    while t < T:
        w_old.vector()[:] = w.vector()
        solver.solve()
        t += dt
        f.write(u, t)


def save_and_plot(w, name):
    """Saves and plots provided solution using the given
    name"""

    u, p = w.split()

    # Store to file
    with XDMFFile("results_{}/u.xdmf".format(name)) as f:
        f.write(u)
    with XDMFFile("results_{}/p.xdmf".format(name)) as f:
        f.write(p)

    # Plot
    plt.figure()
    pl = plot(u, title='velocity {}'.format(name))
    plt.colorbar(pl)
    plt.figure()
    pl = plot(p, mode='warp', title='pressure {}'.format(name))
    plt.colorbar(pl)


def postprocess(w, nu, ds_circle):
    """Return lift, drag and the pressure difference"""

    u, p = w.split()

    # Report drag and lift
    n = FacetNormal(w.function_space().mesh())
    force = -p*n + nu*dot(grad(u), n)
    F_D = assemble(-force[0]*ds_circle)
    F_L = assemble(-force[1]*ds_circle)

    U_mean = 0.2
```

(continues on next page)

(continued from previous page)

```
    L = 0.1
    C_D = 2/(U_mean**2*L)*F_D
    C_L = 2/(U_mean**2*L)*F_L

    # Report pressure difference
    a_1 = Point(0.15, 0.2)
    a_2 = Point(0.25, 0.2)
    try:
        p_diff = p(a_1) - p(a_2)
    except RuntimeError:
        p_diff = 0

    return C_D, C_L, p_diff


def tasks_1_2_3_4():
    """Solve and plot alongside Stokes and Navier-Stokes"""

    # Problem data
    u_in = Expression(("4.0*U*x[1]*(0.41 - x[1])/(0.41*0.41)", "0.0"),
                      degree=2, U=0.3)
    nu = Constant(0.001)

    # Discretization parameters
    N_circle = 16
    N_bulk = 64

    # Prepare function space, BCs and measure on circle
    W, bcs, ds_circle = build_space(N_circle, N_bulk, u_in)

    # Solve Stokes
    w = solve_stokes(W, nu, bcs)
    save_and_plot(w, 'stokes')

    # Solve Navier-Stokes
    w = solve_navier_stokes(W, nu, bcs)
    save_and_plot(w, 'navier-stokes')

    # Open and hold plot windows
    plt.show()


def tasks_5_6():
    """Run convergence analysis of drag and lift"""

    # Problem data
    u_in = Expression(("4.0*U*x[1]*(0.41 - x[1])/(0.41*0.41)", "0.0"),
                      degree=2, U=0.3)
    nu = Constant(0.001)

    # Push log levelo to silence DOLFIN
    old_level = get_log_level()
    warning = LogLevel.WARNING if cpp.__version__ > '2017.2.0' else WARNING
    set_log_level(warning)

    fmt_header = "{:10s} | {:10s} | {:10s} | {:10s} | {:10s} | {:10s}"
    fmt_row = "{:10d} | {:10d} | {:10d} | {:10.4f} | {:10.4f} | {:10.6f}"

    # Print table header
    print(fmt_header.format("N_bulk", "N_circle", "#dofs", "C_D", "C_L", "p_diff"))
```

(continues on next page)

(continued from previous page)

```
    # Solve on series of meshes
    for N_bulk in [32, 64, 128]:
        for N_circle in [N_bulk, 2*N_bulk, 4*N_bulk]:

            # Prepare function space, BCs and measure on circle
            W, bcs, ds_circle = build_space(N_circle, N_bulk, u_in)

            # Solve Navier-Stokes
            w = solve_navier_stokes(W, nu, bcs)

            # Compute drag, lift
            C_D, C_L, p_diff = postprocess(w, nu, ds_circle)
            print(fmt_row.format(N_bulk, N_circle, W.dim(), C_D, C_L, p_diff))

    # Pop log level
    set_log_level(old_level)


def task_7():
    """Solve unsteady Navier-Stokes to resolve
    Karman vortex street and save to file"""

    # Problem data
    u_in = Expression(("4.0*U*x[1]*(0.41 - x[1])/(0.41*0.41)", "0.0"),
                      degree=2, U=1)
    nu = Constant(0.001)
    T = 8

    # Discretization parameters
    N_circle = 16
    N_bulk = 64
    theta = 1/2
    dt = 0.2

    # Prepare function space, BCs and measure on circle
    W, bcs, ds_circle = build_space(N_circle, N_bulk, u_in)

    # Solve unsteady Navier-Stokes
    solve_unsteady_navier_stokes(W, nu, bcs, T, dt, theta)


if __name__ == "__main__":

    tasks_1_2_3_4()
    tasks_5_6()
    task_7()
```

6 Hyperelasticity

Find approximate solution to following non-linear system of PDEs

$$
\begin{aligned}
\mathbf{u}_t &= \mathbf{v} && \text{in } \Omega \times (0,T),\\
\mathbf{v}_t &= \operatorname{div}(J\mathbb{T}\mathbb{F}^{-\top}) && \text{in } \Omega \times (0,T),\\
J^2 - 1 &= \begin{cases} 0 & \text{incompressible case} - p/\lambda \\ \text{compressible case} \end{cases} && \text{in } \Omega \times (0,T),\\
\mathbf{u} = \mathbf{v} &= 0 && \text{on } \Gamma_\mathrm{D} \times (0,T),\\
J\mathbb{T}\mathbb{F}^{-\top}\mathbf{n} &= \mathbf{g} && \text{on } \Gamma_\mathrm{N} \times (0,T),\\
J\mathbb{T}\mathbb{F}^{-\top}\mathbf{n} &= 0 && \text{on } \partial\Omega \backslash (\Gamma_\mathrm{D} \cup \Gamma_\mathrm{N}) \times (0,T),\\
\mathbf{u} = \mathbf{v} &= 0 && \text{on } \Omega \times \{0\}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathbb{F} &= \mathbb{I} + \nabla\mathbf{u},\\
J &= \det\mathbb{F},\\
\mathbb{B} &= \mathbb{F}\mathbb{F}^\top,\\
\mathbb{T} &= -p\mathbb{I} + \mu(\mathbb{B} - \mathbb{I})
\end{aligned}
$$

using θ-scheme discretization in time and arbitrary discretization in space with data

$$
\begin{aligned}
\Omega &= \begin{cases} (0,20) \times (0,1) & \text{in 2Dlego brick } 10 \times 2 \times 1H \\ \text{in 3D} \end{cases}\\
\Gamma_\mathrm{D} &= \begin{cases} \{x = 0\} & \text{in 2D } \{x = \inf_{\mathbf{x}\in\Omega} x\} \\ \text{in 3D} \end{cases}\\
\Gamma_\mathrm{N} &= \begin{cases} \{x = 20\} & \text{in 2D } \{x = \sup_{\mathbf{x}\in\Omega} x\} \\ \text{in 3D} \end{cases}\\
T &= 5,\\
\mathbf{g} &= \begin{cases} J\mathbb{F}^{-\top}\left[0100t\right] & \text{in 2D} J\mathbb{F}^{-\top}\left[00100t\right] \\ \text{in 3D} \end{cases}\\
\mu &= \frac{E}{2(1+\nu)},\\
\lambda &= \begin{cases} \infty & \text{incompressible case} \frac{E\nu}{(1+\nu)(1-2\nu)} \\ \text{compressible case} \end{cases}\\
E &= 10^5,\\
\nu &= \begin{cases} 1/2 & \text{incompressible case} 0.3 \\ \text{compressible case} \end{cases}
\end{aligned}
$$

Mesh file of lego brick `lego_beam.xml`. Within shell download by

Task 1

Discretize the equation in time using the Crank-Nicolson scheme and derive a variational formulation of the problem. Consider discretization using P1/P1/P1 mixed element.

Task 2

Build 2D mesh:

```
mesh = RectangleMesh(Point(x0, y0), Point(x1, y1), 100, 5, 'crossed')
```

Prepare facet function marking Γ_N and Γ_D and plot it to check its correctness.

Hint: You can get coordinates of Γ_D by something like `x0 = mesh.coordinates()[:, 0].min()` for lego mesh. Analogically for Γ_N.

Task 3

Define Cauchy stress and variational formulation of the problem.

Hint: Get geometric dimension by `gdim = mesh.geometry().dim()` to be able to write the code independently of the dimension.

Task 4

Prepare a solver and write simple time-stepping loop. Use time step $\Delta t = \frac{1}{4}$.

Prepare a solver by:

```
problem = NonlinearVariationalProblem(F, w, bcs=bcs, J=J)
solver = NonlinearVariationalSolver(problem)
solver.parameters['newton_solver']['relative_tolerance'] = 1e-6
solver.parameters['newton_solver']['linear_solver'] = 'mumps'
```

to increase the tolerance reasonably and employ powerful sparse direct solver MUMPS.

Prepare nice plotting of displacement by:

```
plot(u, mode="displacement")
```

Manipulate the plot how shown in *the Matplotlib note*.

Task 4

Solve the compressible 2D problem.

Solve the incompressible 2D problem.

Task 5

Solve the 3D compressible problem. Use time step $\Delta t = \frac{1}{2}$.

Load mesh by:

```
mesh = Mesh('lego_beam.xml')
```

Use the following optimization:

```
# Limit quadrature degree
dx = dx(degree=4)
ds = ds(degree=4)
```

You can also try to run the 3D problem in parallel:

```
# Disable plotting
export MPLBACKEND=template
export DOLFIN_NOPLOT=1

# Run the code on <np> processors
mpirun -n <np> python <script>.py
```

Task 6

Plot computed displacement u in Paraview using `Warp by vector` filter.

6.1 Reference solution

Show/Hide Code

Download Code

```
from dolfin import *
import matplotlib.pyplot as plt
import os


def solve_elasticity(facet_function, E, nu, dt, T_end, output_dir):
    """Solves elasticity problem with Young modulus E, Poisson ration nu,
    timestep dt, until T_end and with output data going to output_dir.
    Geometry is defined by facet_function which also defines rest boundary
    by marker 1 and traction boundary by marker 2."""

    # Get mesh and prepare boundary measure
    mesh = facet_function.mesh()
    gdim = mesh.geometry().dim()
    dx = Measure("dx")
    ds = Measure("ds", subdomain_data=facet_function, subdomain_id=2)

    # Limit quadrature degree
    dx = dx(degree=4)
    ds = ds(degree=4)

    # Build function space
    element_v = VectorElement("P", mesh.ufl_cell(), 1)
    element_s = FiniteElement("P", mesh.ufl_cell(), 1)
    mixed_element = MixedElement([element_v, element_v, element_s])
    W = FunctionSpace(mesh, mixed_element)
    info("Num DOFs {}".format(W.dim()))

    # Prepare BCs
    bc0 = DirichletBC(W.sub(0), gdim*(0,), facet_function, 1)
    bc1 = DirichletBC(W.sub(1), gdim*(0,), facet_function, 1)
    bcs = [bc0, bc1]

    # Define constitutive law
    def stress(u, p):
        """Returns 1st Piola-Kirchhoff stress and (local) mass balance
        for given u, p."""
        mu = Constant(E/(2.0*(1.0 + nu)))
        F = I + grad(u)
        J = det(F)
        B = F * F.T
```

(continues on next page)

(continued from previous page)

```
        T = -p*I + mu*(B-I) # Cauchy stress
        S = J*T*inv(F).T # 1st Piola-Kirchhoff stress
        if nu == 0.5:
            # Incompressible
            pp = J-1.0
        else:
            # Compressible
            lmbd = Constant(E*nu/((1.0 + nu)*(1.0 - 2.0*nu)))
            pp = 1.0/lmbd*p + (J*J-1.0)
        return S, pp

    # Timestepping theta-method parameters
    q = Constant(0.5)
    dt = Constant(dt)

    # Unknowns, values at previous step and test functions
    w = Function(W)
    u, v, p = split(w)
    w0 = Function(W)
    u0, v0, p0 = split(w0)
    _u, _v, _p = TestFunctions(W)

    I = Identity(W.mesh().geometry().dim())

    # Balance of momentum
    S, pp = stress(u, p)
    S0, pp0 = stress(u0, p0)
    F1 = (1.0/dt)*inner(u-u0, _u)*dx \
       - ( q*inner(v, _u)*dx + (1.0-q)*inner(v0, _u)*dx )
    F2a = inner(S, grad(_v))*dx + pp*_p*dx
    F2b = inner(S0, grad(_v))*dx + pp0*_p*dx
    F2 = (1.0/dt)*inner(v-v0, _v)*dx + q*F2a + (1.0-q)*F2b

    # Traction at boundary
    F = I + grad(u)
    bF_magnitude = Constant(0.0)
    bF_direction = {2: Constant((0.0, 1.0)), 3: Constant((0.0, 0.0, 1.0))}[gdim]
    bF = det(F)*dot(inv(F).T, bF_magnitude*bF_direction)
    FF = inner(bF, _v)*ds

    # Whole system and its Jacobian
    F = F1 + F2 + FF
    J = derivative(F, w)

    # Initialize solver
    problem = NonlinearVariationalProblem(F, w, bcs=bcs, J=J)
    solver = NonlinearVariationalSolver(problem)
    solver.parameters['newton_solver']['relative_tolerance'] = 1e-6
    solver.parameters['newton_solver']['linear_solver'] = 'mumps'

    # Extract solution components
    u, v, p = w.split()
    u.rename("u", "displacement")
    v.rename("v", "velocity")
    p.rename("p", "pressure")

    # Create files for storing solution
    vfile = XDMFFile(os.path.join(output_dir, "velo.xdmf"))
    ufile = XDMFFile(os.path.join(output_dir, "disp.xdmf"))
    pfile = XDMFFile(os.path.join(output_dir, "pres.xdmf"))
```

(continues on next page)

(continued from previous page)

```
    # Prepare plot window
    fig = plt.figure()
    fig.show()

    # Time-stepping loop
    t = 0
    while t <= T_end:
        t += float(dt)
        info("Time: {}".format(t))

        # Increase traction
        bF_magnitude.assign(100.0*t)

        # Prepare to solve and solve
        w0.assign(w)
        solver.solve()

        # Store solution to files and plot
        ufile.write(u, t)
        vfile.write(v, t)
        pfile.write(p, t)
        fig.clear()
        plot(u, mode="displacement")
        fig.canvas.draw()

    # Close files
    vfile.close()
    ufile.close()
    pfile.close()


def geometry_2d(length):
    """Prepares 2D geometry. Returns facet function with 1, 2 on parts of
    the boundary."""
    n = 5
    x0 = 0.0
    x1 = x0 + length
    y0 = 0.0
    y1 = 1.0
    mesh = RectangleMesh(Point(x0, y0), Point(x1, y1), int((x1-x0)*n), int((y1-
↪y0)*n), 'crossed')
    boundary_parts = MeshFunction('size_t', mesh, mesh.topology().dim()-1)
    left  = AutoSubDomain(lambda x: near(x[0], x0))
    right = AutoSubDomain(lambda x: near(x[0], x1))
    left .mark(boundary_parts, 1)
    right.mark(boundary_parts, 2)
    return boundary_parts


def geometry_3d():
    """Prepares 3D geometry. Returns facet function with 1, 2 on parts of
    the boundary."""
    mesh = Mesh('lego_beam.xml')
    gdim = mesh.geometry().dim()
    x0 = mesh.coordinates()[:, 0].min()
    x1 = mesh.coordinates()[:, 0].max()
    boundary_parts = MeshFunction('size_t', mesh, mesh.topology().dim()-1)
    left  = AutoSubDomain(lambda x: near(x[0], x0))
    right = AutoSubDomain(lambda x: near(x[0], x1))
    left .mark(boundary_parts, 1)
    right.mark(boundary_parts, 2)
```

(continues on next page)

(continued from previous page)

```
    return boundary_parts


if __name__ == '__main__':
    parameters['std_out_all_processes'] = False

    solve_elasticity(geometry_2d(20.0), 1e5, 0.3, 0.25, 5.0, 'results_2d_comp')
    solve_elasticity(geometry_2d(20.0), 1e5, 0.5, 0.25, 5.0, 'results_2d_incomp')
    solve_elasticity(geometry_2d(80.0), 1e5, 0.3, 0.25, 5.0, 'results_2d_long_comp
↪')
    solve_elasticity(geometry_3d(),      1e5, 0.3, 0.50, 5.0, 'results_3d_comp')
```

7 Eigenfunctions of Laplacian and Helmholtz equation

7.1 Wave equation with time-harmonic forcing

Let's have wave equation with special right-hand side

$$\begin{aligned} w_{tt} - \Delta w &= f\, e^{i\omega t} && \text{in } \Omega\times(0,T), \\ w &= 0 && \text{on } \partial\Omega\times(0,T) \end{aligned} \tag{24}$$

with $f \in L^2(\Omega)$. Assuming ansatz

$$w(t,x) = u(x)e^{i\omega t}$$

we observe that u has to fulfill

$$\begin{aligned} -\Delta u - \omega^2 u &= f && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{25}$$

Task 1

Try solving (25) in FEniCS with data

$$\begin{aligned} \Omega &= (0,1)\times(0,1), \\ \omega &= \sqrt{5}\pi, \\ f &= x+y \end{aligned} \tag{26}$$

on series of refined meshes. Observe behavior of solution energy $\|\nabla u\|_2$ with refinement. Is there a convergence or not?

Define eigenspace of Laplacian (with zero BC) corresponding to ω^2 as

$$E_{\omega^2} := \left\{ u \in H^1_0(\Omega) : -\Delta u = \omega^2 u \right\}.$$

$E_{\omega^2} \neq \{0\}$ if and only if ω^2 is an eigenvalue. Note that E_{ω^2} is finite-dimensional. Now define P_{ω^2} as L^2-orthogonal projection onto E_{ω^2}. It is not difficult to check that the function

$$w(t,x) = \frac{t e^{i\omega t}}{2i\omega}(P_{\omega^2} f)(x) + e^{i\omega t}u(x) \tag{27}$$

solves (24) provided u fulfills

$$\begin{aligned} -\Delta u - \omega^2 u &= (1-P_{\omega^2})f && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{28}$$

Note that problem (28) has a solution which is uniquely determined up to arbitrary function from E_{ω^2}.

Task 2

Construct basis of E_{ω^2} by numerically solving the corresponding eigenproblem with data (26).

Hint: Having forms `a`, `m` and boundary condition `bc` representing eigenvalue problem

$$\begin{aligned} -\Delta u = \lambda u \qquad & \text{in } \Omega, \\ u = 0 \qquad & \text{on } \partial\Omega. \end{aligned}$$

assemble matrices `A`, `B` using function `assemble_system`

```
A = assemble_system(a, zero_form, bc)
B = assemble(m)
```

Then the eigenvectors solving

$$Ax = \lambda Bx$$

with λ close to target `lambd` can be found by:

```
eigensolver = SLEPcEigenSolver(as_backend_type(A), as_backend_type(B))
eigensolver.parameters['problem_type'] = 'gen_hermitian'
eigensolver.parameters['spectrum'] = 'target real'
eigensolver.parameters['spectral_shift'] = lambd
eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
eigensolver.parameters['tolerance'] = 1e-6
#eigensolver.parameters['verbose'] = True  # for debugging
eigensolver.solve(number_of_requested_eigenpairs)

eig = Function(V)
eig_vec = eig.vector()
space = []
for j in range(eigensolver.get_number_converged()):
    r, c, rx, cx = eigensolver.get_eigenpair(j)
    eig_vec[:] = rx
    plot(eig, title='Eigenvector to eigenvalue %g'%r)
    plt.show()
```

Task 4

Implement projection P_{ω^2}. Use it to solve problem (28) with data (26).

Task 5

Construct the solution $w(t, x)$ of the wave equations (24) using formula (27). Plot temporal evolution of its real and imaginary part.

7.2 Mesh generation by Gmsh

Task 6

Modify a Gmsh demo to mesh a half ball

$$\{(x,y,z), x^2+y^2+z^2<1, y>0\}$$

using the following code:

```
wget https://gitlab.onelab.info/gmsh/gmsh/blob/
↪ad0ab3d5c310e7048ffa6e032ccd4e8f0108aa12/demos/api/boolean.py
source /LOCAL/opt/gmsh-4.0.0/gmsh.conf
python3 boolean.py
meshio-convert -p -o xdmf-xml boolean.msh boolean.xdmf
paraview boolean.xdmf &

<edit> boolean.py

python3 boolean.py
meshio-convert -p -o xdmf-xml boolean.msh boolean.xdmf
```

If in a need peek into

```
>>> import gmsh
>>> help(gmsh.model.occ.addSphere)
```

Task 7

Find E_{ω^2} with $\omega^2 \approx 70$ on the half ball. Plot the eigenfunctions in Paraview.

Hint: Use `Glyph` filter, `Sphere` glyph type, decrease the scale factor to ca. 0.025.

Use `Clip` filter. Drag the clip surface by mouse, hit `Alt+A` to refresh.

7.3 Reference solution

Show/Hide Code

Download Code

```
from dolfin import *
import matplotlib.pyplot as plt


def solve_helmholtz(V, lambd, f):
    """Solve Helmholtz problem

        -\Delta u - lambd u = f   in \Omega
                          u = 0   on \partial\Omega

    and return u.
    """

    bc = DirichletBC(V, 0, lambda x, on_boundary: on_boundary)
    u, v = TrialFunction(V), TestFunction(V)
    a = inner(grad(u), grad(v))*dx - Constant(lambd)*u*v*dx
    L = f*v*dx
    u = Function(V)
    solve(a == L, u, bc)
    return u
```

(continues on next page)

(continued from previous page)

```python
def build_laplacian_eigenspace(V, lambd, maxdim, tol):
    """For given space V finds eigenspace of Laplacian
    (with zero Dirichlet BC) corresponding to eigenvalues
    close to lambd by given tolerance tol. Return list
    with basis functions of the space.
    """

    # Assemble Laplacian A and mass matrix B
    bc = DirichletBC(V, 0, lambda x, on_boundary: on_boundary)
    u, v = TrialFunction(V), TestFunction(V)
    a = inner(grad(u), grad(v))*dx
    L_dummy = Constant(0)*v*dx
    m = u*v*dx
    A, _ = assemble_system(a, L_dummy, bc)
    B = assemble(m)

    # Prepare eigensolver for
    #
    #    A x = lambda B x
    eigensolver = SLEPcEigenSolver(as_backend_type(A), as_backend_type(B))
    eigensolver.parameters['problem_type'] = 'gen_hermitian'
    eigensolver.parameters['spectrum'] = 'target real'
    eigensolver.parameters['spectral_shift'] = float(lambd)
    eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
    eigensolver.parameters['tolerance'] = 1e-6
    #eigensolver.parameters['verbose'] = True  # for debugging

    # Solve for given number of eigenpairs
    eigensolver.solve(maxdim)

    # Iterate over converged eigenpairs
    space = []
    for j in range(eigensolver.get_number_converged()):

        # Get eigenpair
        r, c, rx, cx = eigensolver.get_eigenpair(j)

        # Check that eigenvalue is real
        assert near(c/r, 0, 1e-6)

        # Consider found eigenvalues close to the target eigenvalue
        if near(r, lambd, tol*lambd):
            print('Found eigenfunction with eigenvalue {} close to target {} '
                  'within tolerance {}'.format(r, lambd, tol))

            # Store the eigenfunction
            eig = Function(V)
            eig.vector()[:] = rx
            space.append(eig)

    # Check that we got whole eigenspace, i.e., last eigenvalue is different one
    assert not near(r, lambd, tol), "Possibly don't have whole eigenspace!"

    # Report
    print('Eigenspace for {} has dimension {}'.format(lambd, len(space)))

    return space
```

(continues on next page)

(continued from previous page)

```python
def orthogonalize(A):
    """L^2-orthogonalize a list of Functions living on the same
    function space. Modify the functions in-place.
    Use classical Gramm-Schmidt algorithm for brevity.
    For numerical stability modified Gramm-Schmidt would be better.
    """

    # Set of single function is orthogonal
    if len(A) <= 1:
        return

    # Orthogonalize overything but the last function
    orthogonalize(A[:-1])

    # Orthogonalize the last function to the previous ones
    f = A[-1]
    for v in A[:-1]:
        r = assemble(inner(f, v)*dx) / assemble(inner(v, v)*dx)
        assert f.function_space() == v.function_space()
        f.vector().axpy(-r, v.vector())


def task_1():

    # Problem data
    f = Expression('x[0] + x[1]', degree=1)
    omega2 = 5*pi**2

    # Iterate over refined meshes
    ndofs, energies = [], []
    for n in (2**i for i in range(2, 7)):

        mesh = UnitSquareMesh(n, n)
        V = FunctionSpace(mesh, "Lagrange", 1)
        u = solve_helmholtz(V, omega2, f)

        # Store energy to check convergence
        ndofs.append(u.function_space().dim())
        energies.append(norm(u, norm_type='H10'))

    # Plot energies against number dofs
    plt.plot(ndofs, energies, 'o-')
    plt.xlabel('dimension')
    plt.ylabel('energy')
    plt.show()


def tasks_2_3_4():

    # Problem data
    f = Expression('x[0] + x[1]', degree=1)
    omega2 = 5*pi**2

    # Iterate over refined meshes
    ndofs, energies = [], []
    for n in (2**i for i in range(2, 7)):

        mesh = UnitSquareMesh(n, n)
        V = FunctionSpace(mesh, "Lagrange", 1)

        # Build eingenspace of omega2
```

(continues on next page)

(continued from previous page)

```
        eigenspace = build_laplacian_eigenspace(V, omega2, 10, 0.1)

        # Orthogonalize f to the eigenspace
        f_perp = project(f, V)
        orthogonalize(eigenspace+[f_perp])

        # Find particular solution with orthogonalized rhs
        u = solve_helmholtz(V, omega2, f_perp)

        # Store energy to check convergence
        ndofs.append(u.function_space().dim())
        energies.append(norm(u, norm_type='H10'))

    # Plot energies against number dofs
    plt.plot(ndofs, energies, 'o-')
    plt.xlabel('dimension')
    plt.ylabel('energy')
    plt.show()

    # Create and save w(t, x) for plotting in Paraview
    omega = omega2**0.5
    Pf = project(f - f_perp, V)
    T_per = 2*pi/omega
    create_and_save_w(omega, Pf, u, 20*T_per, 0.1*T_per)


def create_and_save_w(omega, Pf, u, T, dt):
    """Create and save w(t, x) on (0, T) with time
    resolution dt
    """

    # Extract common function space
    V = u.function_space()
    assert V == Pf.function_space()

    w = Function(V)

    Pf_vec = Pf.vector()
    u_vec = u.vector()
    w_vec = w.vector()

    f = XDMFFile('w.xdmf')
    f.parameters['rewrite_function_mesh'] = False
    f.parameters['functions_share_mesh'] = True

    def c1(t):
        return t*sin(omega*t)/(2*omega), -t*cos(omega*t)/(2*omega)

    def c2(t):
        return cos(omega*t), sin(omega*t)

    t = 0
    while t < T:
        c1_real, c1_imag = c1(t)
        c2_real, c2_imag = c2(t)

        # Store real part
        w.vector().zero()
        w.vector().axpy(c1_real, Pf_vec)
        w.vector().axpy(c2_real, u_vec)
        w.rename('w_real', 'w_real')
```

(continues on next page)

(continued from previous page)

```
        f.write(w, t)

        # Store imaginary part
        w.vector().zero()
        w.vector().axpy(c1_imag, Pf_vec)
        w.vector().axpy(c2_imag, u_vec)
        w.rename('w_imag', 'w_imag')
        f.write(w, t)

        t += dt

    f.close()


if __name__ == '__main__':

    task_1()
    tasks_2_3_4()
```

8 Heat equation in moving media

Find approximate solution to following linear PDE

$$\begin{aligned} u_t + \mathbf{b}\cdot\nabla u - \operatorname{div}(K\nabla u) &= f && \text{in } \Omega\times(0,T),\\ u &= u_{\mathrm{D}} && \text{in } \Omega_{\mathrm{D}}\times(0,T),\\ \tfrac{\partial u}{\partial \mathbf{n}} &= g && \text{on } \Gamma_{\mathrm{N}}\times(0,T),\\ u &= u_0 && \text{on } \Omega\times\{0\} \end{aligned}$$

using θ-scheme discretization in time and arbitrary FE discretization in space with data

- $\Omega = (0,1)^2$
- $T = 10$
- $\Gamma_{\mathrm{N}} = \{x=0\}$
- $\Gamma_{\mathrm{D}} = \{x=1\}\cup\{y=0\}$
- $g = 0.1$
- $K = 0.01$
- $\mathbf{b} = \left(-(y-\frac{1}{2}), x-\frac{1}{2}\right)$
- $f = \chi_{B_{1/5}([\frac{3}{4},\frac{3}{4}])}$
- $u_0(\mathbf{x}) = \left(1 - 25\,\mathrm{dist}\left(\mathbf{x}, \left[\frac{1}{4},\frac{1}{4}\right]\right)\right)\chi_{B_{1/5}([\frac{1}{4},\frac{1}{4}])}$
- $u_{\mathrm{D}} = 0$

where χ_X is a characteristic function of set X, $B_R(\mathbf{z})$ is a ball of radius R and center $\mathbf{z}$ and $\mathrm{dist}(\mathbf{p},\mathbf{q})$ is Euclidian distance between points $\mathbf{p}$, $\mathbf{q}$.

Task 1

Discretize the equation in time and write variational formulation of the problem.

Task 2

Build mesh, prepare facet function marking Γ_{N} and Γ_{D} and plot it to check its correctness.

Hint: You can follow the procedure from subdomains-poisson demo. (Follow a construction of `boundaries` object therein.)

```
mesh = UnitSquareMesh(10, 10, 'crossed')

# Create boundary markers
tdim = mesh.topology().dim()
boundary_parts = MeshFunction('size_t', mesh, tdim-1)
left   = AutoSubDomain(lambda x: near(x[0], 0.0))
right  = AutoSubDomain(lambda x: near(x[0], 1.0))
bottom = AutoSubDomain(lambda x: near(x[1], 0.0))
left  .mark(boundary_parts, 1)
right .mark(boundary_parts, 2)
bottom.mark(boundary_parts, 2)
```

Task 3

Define expressions $\mathbf{b}$, f, u_0 and plot them.

Hint: According to your personal taste, get hint at Expression class documentation or any documented demo. Use any kind of expression you wish (subclassing Python `Expression`, oneline C++, subclassing C++ `Expression`).

```
# python Expression subclass
class B(Expression):
    def eval(self, value, x):
        vx = x[0] - 0.5
        vy = x[1] - 0.5
        value[0] = -vy
        value[1] =  vx
    def value_shape(self):
        return (2,)

b = B()
```

```
# oneline C++
b = Expression(("-(x[1] - 0.5)", "x[0] - 0.5"))
```

Task 4

Use facet markers from Task 2 to define `DirichletBC` object and `Measure` for integration along Γ_N.

Hint: See UFL class Measure

```
dsN = Measure("ds", subdomain_id=1, subdomain_data=boundary_parts)
```

Task 5

Now proceed to variational formulation and time-stepping loop. Write bilinear and linear form representing PDE. How is solution at previous time-step represented therein?

Hint: Use `LinearVariationalProblem` and `LinearVariationalSolver` classes so that `solve` method of an instance of the latter is called every time-step while nothing else is touched excepted updating value of solution from previous time-step figuring in variational form. You can use for instance `Function.assign` method to do that.

Task 5

Add solution output for external visualisation, like Paraview.

Hint:

```
# Create file for storing results
f = XDMFFile("results/u.xdmf")

u.rename("u", "temperature")
f.write(u, t)
```

8.1 Reference solution

Show/Hide Code

Download Code

```
from dolfin import *
import matplotlib.pyplot as plt

# Create mesh and build function space
mesh = UnitSquareMesh(40, 40, 'crossed')
V = FunctionSpace(mesh, "Lagrange", 1)

# Create boundary markers
tdim = mesh.topology().dim()
boundary_parts = MeshFunction('size_t', mesh, tdim-1)
left   = AutoSubDomain(lambda x: near(x[0], 0.0))
right  = AutoSubDomain(lambda x: near(x[0], 1.0))
bottom = AutoSubDomain(lambda x: near(x[1], 0.0))
left  .mark(boundary_parts, 1)
right .mark(boundary_parts, 2)
bottom.mark(boundary_parts, 2)

# Initial condition and right-hand side
ic = Expression("""pow(x[0] - 0.25, 2) + pow(x[1] - 0.25, 2) < 0.2*0.2
                   ? -25.0 * ((pow(x[0] - 0.25, 2) + pow(x[1] - 0.25, 2)) - 0.2*0.
↪2)
                   : 0.0""", degree=1)
f = Expression("""pow(x[0] - 0.75, 2) + pow(x[1] - 0.75, 2) < 0.2*0.2
                  ? 1.0
                  : 0.0""", degree=1)

# Equation coefficients
K = Constant(1e-2) # thermal conductivity
g = Constant(0.01) # Neumann heat flux
b = Expression(("-(x[1] - 0.5)", "x[0] - 0.5"), degree=1) # convecting velocity
```

(continues on next page)

(continued from previous page)

```
# Define boundary measure on Neumann part of boundary
dsN = Measure("ds", subdomain_id=1, subdomain_data=boundary_parts)

# Define steady part of the equation
def operator(u, v):
    return ( K*inner(grad(u), grad(v)) - f*v + dot(b, grad(u))*v )*dx - K*g*v*dsN

# Define trial and test function and solution at previous time-step
u = TrialFunction(V)
v = TestFunction(V)
u0 = Function(V)

# Time-stepping parameters
t_end = 10
dt = 0.1
theta = Constant(0.5) # Crank-Nicolson scheme

# Define time discretized equation
F = (1.0/dt)*inner(u-u0, v)*dx + theta*operator(u, v) + (1.0-theta)*operator(u0, v)

# Define boundary condition
bc = DirichletBC(V, Constant(0.0), boundary_parts, 2)

# Prepare solution function and solver
u = Function(V)
problem = LinearVariationalProblem(lhs(F), rhs(F), u, bc)
solver  = LinearVariationalSolver(problem)

# Prepare initial condition
u0.interpolate(ic)

# Create file for storing results
f = XDMFFile("results/u.xdmf")

# Time-stepping
t = 0.0
u.rename("u", "temperature")
u.interpolate(ic)

# Save initial solution
f.write(u, t)

# Open figure for plots
fig = plt.figure()
plt.show(block=False)

while t < t_end:

    # Solve the problem
    solver.solve()

    # Store solution to file and plot
    f.write(u, t)
    p = plot(u, title='Solution at t = %g' % t)
    if p is not None:
        plt.colorbar(p)
    fig.canvas.draw()
    plt.clf()

    # Move to next time step
    u0.assign(u)
```

(continues on next page)

(continued from previous page)

```
    t += dt

    # Report flux
    n = FacetNormal(mesh)
    flux = assemble(K*dot(grad(u), n)*dsN)
    info('t = %g, flux = %g' % (t, flux))
```

9 p-Laplace equation

9.1 Potential for Laplace equation

Task 1

Formulate Laplace equation

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega, \end{aligned}$$

as a variational problem (i.e., find potential for the equation) and solve it using FEniCS with data

- $\Omega = [0,1] \times [0,1]$,
- $f = 1 + \cos(2\pi x)\sin(2\pi y)$.

Use UFL function derivative to automatically obtain Galerkin formulation from the potential. Don't assume linearity of the PDE - solve it as nonliner (Newton will converge in 1 step).

9.2 Potential for p-Laplace equation

Task 2

Replace every occurrence of number 2 in potential for Laplace equation by p. This is called p-Laplacian for $1 < p < +\infty$.

Convince yourself that resulting PDE is non-linear whenever $p \neq 2$.

Task 3

Run the algorithm from Task 1 with $p = 1.1$ and $p = 11$.

Hint: Use DOLFIN class `Constant` to avoid form recompilation by FFC for distinct values of p.

Do you know where is the problem? If not, compute second Gateaux derivative of the potential (which serves as Jacobian for the Newton-Rhapson algorithm) and look at its value for $u = 0$.

Task 4

Add some regularization to the potential to make it non-singular/non-degenerate. (Prefer regularization without square roots.) Find a solution of the original p-Laplace problem with $p = 1.1$ and $p = 11$ using careful approximation by regularized problem. Report $max_\Omega u_h$ for u_h being the approximate solution.

Hint: For `u_h` Lagrange degree 1 `Function` the maximum matches maximal nodal value so it is `u_h.vector().max()` because nodal basis is used. *Warning.* This does not generally hold for higher polynomial degrees!

9.3 Reference solution

Reference solution consists of three files – one module file:

Show/Hide File `p_laplace.py`

`Download Code`

```
from dolfin import *
import matplotlib.pyplot as plt

mesh = UnitSquareMesh(40, 40)
V = FunctionSpace(mesh, 'Lagrange', 1)
f = Expression("1.+cos(2.*pi*x[0])*sin(2.*pi*x[1])", degree=2)

def p_laplace(p, eps, u0=None):
    """Solves regularized p-Laplacian with mesh, space and right-hand side
    defined above. Returns solution, regularized energy and energy."""
    p = Constant(p)
    eps = Constant(eps)

    # Initial approximation for Newton
    u = u0.copy(deepcopy=True) if u0 else Function(V)

    # Energy functional
    E = ( 1./p*(eps + dot(grad(u), grad(u)))**(0.5*p) - f*u ) * dx

    # First Gateaux derivative
    F = derivative(E, u)

    # Solve nonlinear problem; Newton is used by default
    bc = DirichletBC(V, 0.0, lambda x,onb: onb)
    solver_parameters = {'newton_solver': {'maximum_iterations': 1000}}
    solve(F == 0, u, bc, solver_parameters=solver_parameters)

    plt.gcf().show()
    plt.clf()
    plot(u, mode="warp", title='p-Laplace, p=%g, eps=%g'%(float(p),␣
↪float(eps)))
    plt.gcf().canvas.draw()

    # Compute energies
    energy_regularized = assemble(E)
    eps.assign(0.0)
    energy = assemble(E)

    return u, energy_regularized, energy
```

and two executable scripts:

Show/Hide File `p_small.py`

`Download Code`

```
from p_laplace import p_laplace
import numpy as np
import matplotlib.pyplot as plt

epsilons = [10.0**i for i in range(-10, -22, -2)]
energies = []
maximums = []

# Converge with regularization
for eps in epsilons:
    result = p_laplace(1.1, eps)
    u = result[0]
    energies.append(result[1:])
    # Maxmimal nodal value (correct maximum only for P1 function)
    maximums.append(u.vector().max())
energies = np.array(energies)

# Report
print(' epsilon  |  energy_reg  |  energy | u max \n',
    np.concatenate( ( np.array([epsilons,]).T,
                      energies,
                      np.array([maximums,]).T,
                    ), axis=1))

# Plot energies
plt.figure()
plt.plot(epsilons, energies[:,0], 'o-', label='energy regularized')
plt.plot(epsilons, energies[:,1], 'o-', label='energy')
plt.xscale('log')
plt.legend(loc='upper left')
plt.show()
```

Show/Hide File `p_large.py`

Download Code

```
from p_laplace import p_laplace
import numpy as np
import matplotlib.pyplot as plt

epsilons = [10.0**i for i in np.arange(1.0, -6.0, -0.5)]
energies = []
maximums = []

# Converge with regularization,
# use previous solution to start next Newton!
u = None
for eps in epsilons:
    result = p_laplace(11.0, eps, u)
    u = result[0]
    energies.append(result[1:])
    # Maxmimal nodal value (correct maximum only for P1 function)
    maximums.append(u.vector().max())
energies = np.array(energies)

# Report
print(' epsilon  |  energy_reg  |  energy | u max \n',
    np.concatenate( ( np.array([epsilons,]).T,
                      energies,
                      np.array([maximums,]).T,
                    ), axis=1))
```

(continues on next page)

(continued from previous page)

```python
# Plot energies
plt.figure()
plt.plot(epsilons, energies[:,0], 'o-', label='energy regularized')
plt.plot(epsilons, energies[:,1], 'o-', label='energy')
plt.xscale('log')
plt.legend(loc='upper left')
plt.show()
```