Tutorial 4

Graetz problem

Keywords: Non-compliant outputs, successive constraints method

1 Introduction

This Tutorial addresses a so-called *non-compliant* output of interest, geometrical parametrization and the successive constraints method (SCM). In particular, we will solve the Graetz problem, which deals with forced heat convection in a channel divided into two parts (see Figure 1), so that $\Omega_o = \Omega_o^1 \cup \Omega_o^2$. Within the first part Ω_o^1 (the portion depicted in blue) the temperature is kept constant and the flow has a known given convective field.

2 Parametrized formulation

The length of the Ω_o^2, along the axis x, with respect to length of Ω_o^1, is given by the parameter μ_1. The heat transfer between the domains can be taken into account by means of the Péclet number, which will be labeled as the parameter μ_2. The ranges of the two parameters are the following:

$$\mu_1 \in [0.01, 10.0],$$

$$\mu_2 \in [0.01, 10.0],$$

The problem can be stated as follows: for any $\boldsymbol{\mu} = (\mu_1, \mu_2)$, find

$$u_o(\boldsymbol{\mu}) \in \mathbb{V}(\mu_1) = \left\{ v \in H^1(\Omega_o(\mu_1)) : v|_{\Gamma_{o,1,5,6}} = 0, v|_{\Gamma_{o,2,4}} = 1 \right\}$$

such that

$$\mu_2 \int_{\Omega_o(\mu_1)} \nabla u_o(\boldsymbol{\mu}) \cdot \nabla v \, d\boldsymbol{x} + \int_{\Omega_o(\mu_1)} y(1-y) \partial_x u_o(\boldsymbol{\mu}) \, v \, d\boldsymbol{x} = 0 \quad \forall v \in \mathbb{W}(\mu_1) = H^1_{\partial\Omega_o(\mu_1) \setminus \Gamma_{o,3}}(\Omega_o(\mu_1))$$

The output of interest $s\,(u(\boldsymbol{\mu}))$ is the following:

$$s_o\,(u(\boldsymbol{\mu})) = \int_{\Gamma_{o,3}} u(\boldsymbol{\mu}).$$

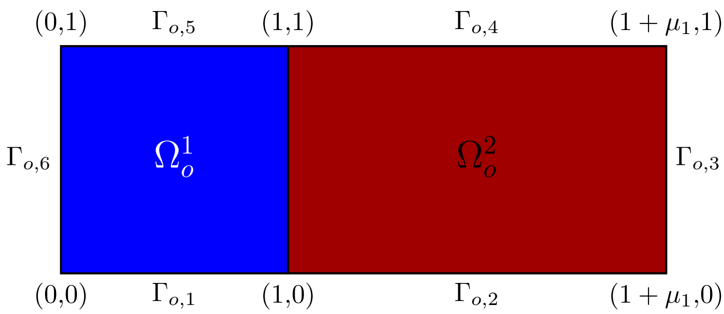

Figure 1: Subdomain division.

3 Affine decomposition

After a change of variable, the problem can be reformulated on the reference domain Ω as follows

$$
\begin{aligned}
&\text{find } u(\boldsymbol{\mu}) \in \mathbb{V}(\mu_1 \equiv 1)\\
&\mu_2 \int_{\Omega^1} \nabla u(\boldsymbol{\mu}) \cdot \nabla v \; d\boldsymbol{x} + \frac{\mu_2}{\mu_1} \int_{\Omega^2} \partial_x u(\boldsymbol{\mu}) \; \partial_x v \; d\boldsymbol{x} + \mu_2 \mu_1 \int_{\Omega^2} \partial_y u(\boldsymbol{\mu}) \; \partial_y v \; d\boldsymbol{x}\\
&\qquad + \int_{\Omega} y(1-y) \partial_x u(\boldsymbol{\mu}) \; v \; d\boldsymbol{x} = 0 \qquad\qquad \forall v \in \mathbb{W}(\mu_1 \equiv 1),
\end{aligned}
$$

being $\Omega^1 = \Omega^1_o(\mu_1 \equiv 1)$, $\Omega^2 = \Omega^2_o(\mu_1 \equiv 1)$ and $\Omega = \Omega^1 \cup \Omega^2$.

Finally, using the (finite element interpolation of the) lifting function

$$
R(x,y) = \begin{cases} 0 & x < 1 \\ 1 & x \geq 1 \end{cases}
$$

the problem is written as

$$
\begin{aligned}
&\text{find } w(\boldsymbol{\mu}) \in \mathbb{W}(1)\\
&\mu_2 \int_{\Omega^1} \nabla w(\boldsymbol{\mu}) \cdot \nabla v \; d\boldsymbol{x} + \frac{\mu_2}{\mu_1} \int_{\Omega^2} \partial_x w(\boldsymbol{\mu}) \; \partial_x v \; d\boldsymbol{x} + \mu_2 \mu_1 \int_{\Omega^2} \partial_y w(\boldsymbol{\mu}) \; \partial_y v \; d\boldsymbol{x} + \int_{\Omega} y(1-y) \; \partial_x w(\boldsymbol{\mu}) v \; d\boldsymbol{x} =\\
&-\mu_2 \int_{\Omega^1} \nabla R \cdot \nabla v \; d\boldsymbol{x} - \frac{\mu_2}{\mu_1} \int_{\Omega^2} \partial_x R \, \partial_x v \; d\boldsymbol{x} - \mu_2 \mu_1 \int_{\Omega^2} \partial_y R \, \partial_y v \; d\boldsymbol{x} - \int_{\Omega} y(1-y) \partial_x R \, v \; d\boldsymbol{x} \quad \forall v \in \mathbb{W}(1)
\end{aligned}
$$

4 Implementation in RBniCS

The implementation of this Tutorial can be found in solve_graetz.py.

4.1 The `Graetz` class – dual approach for non-compliant outputs

In the `Graetz` class we exploit a dual approach for the error estimation of non-compliant outputs. In RBniCS, this is done inheriting from the class `EllipticCoerciveRBNonCompliantBase`.

```
class Graetz(EllipticCoerciveRBNonCompliantBase):
```

Two additional methods needs to be implemented, and are related to the assembly of the non-compliant output.

```
    def compute_theta_s(self):
        return (1.0,)

    def assemble_truth_s(self):
        v = self.v
        dx = self.dx
        s0 = v*dx(2)

        # Assemble and return
        S0 = assemble(s0)
        return (S0,)
```

4.2 The `Graetz` class – SCM for the lower bound of the coercivity constant

An implementation of SCM is also provided in RBniCS. The first step to utilize it is to declare a new SCM object in the constructor of the `Graetz` class.

```python
def __init__(self, V, mesh, subd, bound):
    ...
    self.SCM_obj = SCM(self)
```

Then, for any new instance of the parameter $\boldsymbol{\mu}$, the SCM algorithm can be queried to obtain a lower bound of the coercivity constant of the problem as follows:

```python
def get_alpha_lb(self):
    return self.SCM_obj.get_alpha_LB(self.mu)
```

Few setters need to be modified to propagate the values also to the SCM object.

```python
def setNmax(self, nmax):
    EllipticCoerciveRBNonCompliantBase.setNmax(self, nmax)
    self.SCM_obj.setNmax(nmax)
def settol(self, tol):
    EllipticCoerciveRBNonCompliantBase.settol(self, tol)
    self.SCM_obj.settol(tol)
def setmu_range(self, mu_range):
    EllipticCoerciveRBNonCompliantBase.setmu_range(self, mu_range)
    self.SCM_obj.setmu_range(mu_range)
def setxi_train(self, ntrain, sampling="random"):
    EllipticCoerciveRBNonCompliantBase.setxi_train(self, ntrain,
        ↪ sampling)
    self.SCM_obj.setxi_train(ntrain, sampling)
def setxi_test(self, ntest, sampling="random"):
    EllipticCoerciveRBNonCompliantBase.setxi_test(self, ntest, sampling
        ↪ )
    self.SCM_obj.setxi_test(ntest, sampling)
def setmu(self, mu):
    EllipticCoerciveRBNonCompliantBase.setmu(self, mu)
    self.SCM_obj.setmu(mu)
```

Moreover, the `offline` method is overridden so that is executes the offline stage of the SCM object too.

```python
def offline(self):
    # Perform first the SCM offline phase, ...
    bak_first_mu = tuple(list(self.mu))
    self.SCM_obj.offline()
    # ..., and then call the parent method.
    self.setmu(bak_first_mu)
    EllipticCoerciveRBNonCompliantBase.offline(self)
```

The code solve_graetz.py is executed as described in Tutorial 1.

5 A look under the hood of RBniCS

The class `EllipticCoerciveRBNonCompliantBase`, defined `elliptic_coercive_rb_non_compliant_base.py`, extends the standard reduced basis functionalities to the case of non-compliant elliptic coervice problems. The class `SCM` in `scm.py` provides an implementation of the successive constraints method for the approximation of the coercivity constant. Further details can be found in the doxygen documentation.