Tutorial 3

Geometrical parametrization

Keywords: Geometrical parametrization, mesh motion for display

1 Introduction

This Tutorial introduces problems featuring a geometrical parametrization, by solving a thermal conduction problem on a parametrized computational domain whose geometry is sketched in Figure 1.

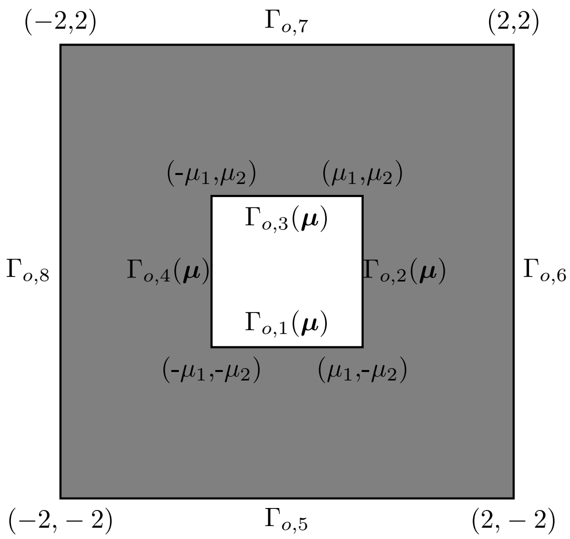

Figure 1: Parametrized domain.

2 Parametrized formulation

The parameters μ_1 and μ_2 are related to the shape of the central hole and their range is as follows:

$$\mu_1 \in [0.5, 1.5] \quad \text{and} \quad \mu_2 \in [0.5, 1.5].$$

The parameter μ_3 is the Biot number, which allows for heat exchange with a surrounding fluid (e.g., air). μ_3 can vary within the following range:

$$\mu_3 \in [0.01, 1].$$

The bilinear form associated to the left-hand side of the problem is given by:

$$a_o(w, v; \boldsymbol{\mu}) = \int_{\Omega_o(\boldsymbol{\mu})} \nabla w \cdot \nabla v \; d\boldsymbol{x} + \mu_3 \left(\int_{\Gamma_{o,5}} w\, v \; ds + \int_{\Gamma_{o,6}} w\, v \; ds + \int_{\Gamma_{o,7}} w\, v \; ds + \int_{\Gamma_{o,8}} w\, v \; ds \right).$$

A constant heat flux is imposed on the interior boundary as follows:

$$f_o(v; \boldsymbol{\mu}) = \int_{\Gamma_{o,1}(\boldsymbol{\mu})} v \ ds + \int_{\Gamma_{o,2}(\boldsymbol{\mu})} v \ ds + \int_{\Gamma_{o,3}(\boldsymbol{\mu})} v \ ds + \int_{\Gamma_{o,4}(\boldsymbol{\mu})} v \ ds.$$

3 Affine decomposition

The problem is cast on a fixed reference domain $\Omega = \Omega_o(\mu_1 \equiv \mu_2 \equiv 1)$. It is straightforward to obtain a piecewise affine map $\boldsymbol{T} : \Omega \to \Omega_o(\boldsymbol{\mu})$ thanks to a partition of the reference domain in several triangular subdomains. A possible set of subdomains is depicted in Figure 2. Then, the equivalent bilinear and linear forms on the reference domain after a change of variable are as follows:

$$a(w, v; \boldsymbol{\mu}) = \sum_{p=1}^{13} \Theta^a_p(\boldsymbol{\mu}) \ a_p(w, v) \qquad\qquad f(v; \boldsymbol{\mu}) = \sum_{q=1}^{4} \Theta^f_q(\boldsymbol{\mu}) \ f_q(v)$$

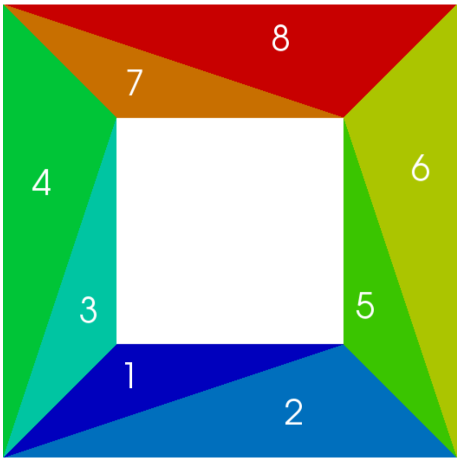

Figure 2: RB triangulation.

For instance, the first three terms of the affine expansion of $a(w, v; \boldsymbol{\mu})$ are related to subdomains 1 and 7 as follows:

$$\begin{aligned}
\Theta^a_1(\boldsymbol{\mu}) &= -\frac{\mu_2 - 2}{\mu_1} - 4\frac{(\mu_1 - 1) \ (\mu_1 - 1)}{\mu_1 \ (\mu_2 - 2)} \\
\Theta^a_2(\boldsymbol{\mu}) &= -\frac{\mu_1}{\mu_2 - 2} \\
\Theta^a_3(\boldsymbol{\mu}) &= -2\frac{\mu_1 - 1}{\mu_2 - 2} \\
a_1(w, v) &= \int_{\Omega_1} w_{,0} \ v_{,0} \ d\boldsymbol{x} + \int_{\Omega_7} w_{,0} \ v_{,0} \ d\boldsymbol{x} \\
a_2(w, v) &= \int_{\Omega_1} w_{,1} \ v_{,1} \ d\boldsymbol{x} + \int_{\Omega_7} w_{,1} \ v_{,1} \ d\boldsymbol{x} \\
a_3(w, v) &= \int_{\Omega_1} (w_{,0} \ v_{,1} + w_{,1} \ v_{,0}) \ d\boldsymbol{x} - \int_{\Omega_7} (w_{,0} \ v_{,1} + w_{,1} \ v_{,0}) \ d\boldsymbol{x},
\end{aligned}$$

being the transformation $\boldsymbol{T}$ restricted to subdomain 1 defined as

$$\begin{cases} x_{o,1} = 2 - 2\mu_1 + \mu_1 \ x_1 + (2 - 2\mu_1) \ x_2 \\ x_{o,2} = 2 - 2\mu_2 - \mu_2 \ x_2 \end{cases} \qquad \forall \boldsymbol{x} = (x_1, x_2) \in \Omega_1.$$

A MATLAB script `map.m` is provided in the `data` folder to ease the computation of the transformation $\boldsymbol{T}$ on the remaining domains.

4 Implementation in RBniCS

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

4.1 The Hole class

As in the previous Tutorial, we are solving a coercive elliptic problem employing a POD–Galerkin reduced order model. Thus, the Hole class inherits from EllipticCoercivePODBase:

```
class Hole(EllipticCoercivePODBase):
```

For mesh motion during the visualization, the constructor needs to store a copy of the nodes in the undeformed configuration (read from file) and a vector FE space to interpolate the deformation.

```
    def __init__(self, V, mesh, subd, bound):
        # Call the standard initialization
        EllipticCoercivePODBase.__init__(self, V, None)
        # ... and also store FEniCS data structures for assembly ...
        self.dx = Measure("dx")[subd]
        self.ds = Measure("ds")[bound]
        # ... and, finally, FEniCS data structure related to the
            ↪ geometrical parametrization
        self.mesh = mesh
        self.subd = subd
        self.xref = mesh.coordinates()[:,0].copy()
        self.yref = mesh.coordinates()[:,1].copy()
        self.deformation_V = VectorFunctionSpace(self.mesh, "Lagrange", 1)
        self.subdomain_id_to_deformation_dofs = ()
        for subdomain_id in np.unique(self.subd.array()):
            self.subdomain_id_to_deformation_dofs += ([],)
        for cell in cells(mesh):
            subdomain_id = int(self.subd.array()[cell.index()] - 1) # tuple
                ↪  start from 0, while subdomains from 1
            dofs = self.deformation_V.dofmap().cell_dofs(cell.index())
            for dof in dofs:
                self.subdomain_id_to_deformation_dofs[subdomain_id].append(
                    ↪ dof)
```

The methods compute_theta_a, compute_theta_f, assemble_truth_a and assemble_truth_f are implemented as in the previous Tutorials to store the affine expansion.

Moreover, the Hole class defines three additional internal methods related to mesh motion. These methods are never used in the assembly process, but are employed for visualization purposes. The first method is compute_displacement, which interpolates the displacement of each node $\mathbf{x}$ of the FE mesh for given values of $\boldsymbol{\mu}$.

```
    def compute_displacement(self):
        expression_displacement_subdomains = (
            Expression(("2.0 - 2.0*mu_1 + mu_1*x[0] - x[0] +(2.0-2.0*mu_1)*
                ↪ x[1]", "2.0 -2.0*mu_2 + (1.0-mu_2)*x[1]"), mu_1 = self.
                ↪ mu[0], mu_2 = self.mu[1]), # subdomain 1
            Expression(("2.0*mu_1-2.0 +(mu_1-1.0)*x[1]", "2.0 -2.0*mu_2 +
                ↪ (1.0-mu_2)*x[1]"), mu_1 = self.mu[0], mu_2 = self.mu[1])
                ↪ , # subdomain 2
            Expression(("2.0 - 2.0*mu_1 + (1.0-mu_1)*x[0]", "2.0 -2.0*mu_2
                ↪ + (2.0-2.0*mu_2)*x[0] + (mu_2 - 1.0)*x[1]"), mu_1 = self
                ↪ .mu[0], mu_2 = self.mu[1]), # subdomain 3
            Expression(("2.0 - 2.0*mu_1 + (1.0-mu_1)*x[0]", "2.0*mu_2 -2.0
                ↪ + (mu_2-1.0)*x[0]"), mu_1 = self.mu[0], mu_2 = self.mu
                ↪ [1]), # subdomain 4
            Expression(("2.0*mu_1 -2.0 + (1.0-mu_1)*x[0]", "2.0 -2.0*mu_2 +
                ↪  (2.0*mu_2-2.0)*x[0] + (mu_2 - 1.0)*x[1]"), mu_1 = self.
                ↪ mu[0], mu_2 = self.mu[1]), # subdomain 5
```

```
            Expression(("2.0*mu_1 -2.0 + (1.0-mu_1)*x[0]", "2.0*mu_2 -2.0 +
                ↪  (1.0 - mu_2)*x[0]"), mu_1 = self.mu[0], mu_2 = self.mu
                ↪ [1]), # subdomain 6
            Expression(("2.0 -2.0*mu_1 + (mu_1-1.0)*x[0] + (2.0*mu_1-2.0)*x
                ↪ [1]", "2.0*mu_2 -2.0 + (1.0 - mu_2)*x[1]"), mu_1 = self.
                ↪ mu[0], mu_2 = self.mu[1]), # subdomain 7
            Expression(("2.0*mu_1 -2.0 + (1.0-mu_1)*x[1]", "2.0*mu_2 -2.0 +
                ↪  (1.0 - mu_2)*x[1]"), mu_1 = self.mu[0], mu_2 = self.mu
                ↪ [1]) # subdomain 8
        )
        displacement_subdomains = ()
        for i in range(len(expression_displacement_subdomains)):
            displacement_subdomains += (interpolate(
                ↪ expression_displacement_subdomains[i], self.
                ↪ deformation_V),)
        displacement = Function(self.deformation_V)
        for i in range(len(displacement_subdomains)):
            subdomain_dofs = self.subdomain_id_to_deformation_dofs[i]
            displacement.vector()[subdomain_dofs] = displacement_subdomains
                ↪ [i].vector()[subdomain_dofs]
        return displacement
```

The other two internal methods `move_mesh` (`reset_reference`) actually carry out (undo, resp.) the mesh motion:

```
    def move_mesh(self):
        print "moving mesh (it may take a while)"
        displacement = self.compute_displacement()
        self.mesh.move(displacement)

    def reset_reference(self):
        print "back to the reference mesh"
        new_coor = np.array([self.xref, self.yref]).transpose()
        self.mesh.coordinates()[:] = new_coor
```

Finally, I/O methods defined in the base class are overridden to properly deform the mesh before providing the user any output of the solution.

```
    def online_solve(self, N=None, with_plot=True):
        # Call the parent method, disabling plot ...
        EllipticCoercivePODBase.online_solve(self, N, False)
        # ... and then deform the mesh and perform the plot
        if with_plot == True:
            self.move_mesh()
            plot(self.red, title = "Reduced solution. mu = " + str(self.mu)
                ↪ , interactive = True)
            self.reset_reference()

    def export_solution(self, solution, filename):
        self.move_mesh()
        file = File(filename + ".pvd", "compressed")
        file << solution
        self.reset_reference()
```

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