Tutorial 5

Non-affine problem

Keywords: empirical interpolation method

1 Introduction

In this tutorial we tackle a non-affine problem by means of the Empirical Interpolation Method. In particular, we will solve the Laplace equation (on a unit square domain) where the right-hand side is given by a parametrized Gaussian function.

2 Parametrized formulation

The weak formulation of the problem is the following:

$$\begin{aligned}
&\text{For any } \boldsymbol{\mu} = (\mu_1, \mu_2) \in \Omega = [-1,1]^2\,, \\
&\text{find } u(\boldsymbol{\mu}) \in V = H^1_0(\Omega)\,, \\
&\int_\Omega \nabla u(\boldsymbol{\mu}) \cdot \nabla v \; d\boldsymbol{x} = \int_\Omega g(\boldsymbol{\mu}) v \; d\boldsymbol{x} && \forall v \in V\,, \\
&g(\boldsymbol{x}; \boldsymbol{\mu}) = \exp\{-2(x-\mu_1)^2 - 2(y-\mu_2)^2\} && \forall \boldsymbol{x} \in \Omega\,.
\end{aligned}$$

The two parameters can vary within the following range:

$$\mu_1, \mu_2 \in [-1, 1].$$

3 Implementation in RBniCS

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

3.1 The `Gaussian` class

In order to obtain an approximate affine expansion, we declare an object of the `EIM` class, and initialize the parametrized function for which the interpolation is sought.

```
class Gaussian(EllipticCoerciveRBBase):
    def __init__(self, V, subd, bound):
        ...
        # Finally, initialize an EIM object for the interpolation of the
          ↪ forcing term
        self.EIM_obj = EIM(self)
        self.EIM_obj.parametrized_function = "exp( - 2*pow(x[0]-mu_1, 2) -
          ↪ 2*pow(x[1]-mu_2, 2) )"
```

As in the case of SCM in the previous tutorial, few setters need to be modified to propagate the values also to the EIM object.

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

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

```
    def offline(self):
        # Perform first the EIM offline phase, ...
        self.EIM_obj.offline()
        # ..., and then call the parent method.
        EllipticCoerciveRBBase.offline(self)
```

Then, the affine expansion of the right-hand side can obtained querying the EIM object:

```
    def compute_theta_f(self):
        self.EIM_obj.setmu(self.mu)
        return self.EIM_obj.compute_interpolated_theta()

    def assemble_truth_f(self):
        v = self.v
        dx = self.dx
        # Call EIM
        self.EIM_obj.setmu(self.mu)
        interpolated_gaussian = self.EIM_obj.
            ↪ assemble_mu_independent_interpolated_function()
        # Assemble
        all_F = ()
        for q in range(len(interpolated_gaussian)):
            f_q = interpolated_gaussian[q]*v*dx
            all_F += (assemble(f_q),)
        # Return
        return all_F
```

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

4 A look under the hood of RBniCS

The class `EIM`, defined `eim.py`, contains the implementation of the empirical interpolation method for the interpolation of parametrized functions. Further details can be found in the doxygen documentation.