pyMOR School 2024

Reduced Basis Methods for parametric problems

Exercise Problems

Problem 1 (2D diffusion problem with output)

In this exercise we will solve a two-dimensional parametric diffusion problem, add an output functional and solve a time-dependent version of the problem.

(a) Solve on $\Omega := (0,1)^2$ the steady-state diffusion problem

$$\begin{aligned} \nabla \cdot (-\sigma(x,y;\mu)\nabla u(x,y;\mu)) &= f(x,y) && (x,y) \in \Omega, \\ u(x,y;\mu) &= 0 && (x,y) \in \partial\Omega,\ y = 0, \\ \sigma(x,y;\mu)\nabla u(x,y;\mu) \cdot n(x,y) &= 0 && (x,y) \in \partial\Omega,\ y \neq 0, \end{aligned}$$

where for $\mu \in \mathbb{R}^{>0}$

$$\sigma(x,y;\mu) := \begin{cases} \mu & x \in (0.45, 0.55), y \in (0.5, 1) \\ 1 & \text{otherwise,} \end{cases}$$

and

$$f(x,y) := \begin{cases} 100 & (x-0.25)^2 + (y-0.75)^2 < 0.01 \\ 0 & \text{otherwise.} \end{cases}$$

Use pyMOR's builtin discretization toolkit to discretize the problem. Solve the resulting discrete model for several parameter values and visualize the solution.

Hints:
- *All relevant classes and functions can be found in the* `pymor.basic` *module.*
- *Create a* `StationaryProblem` *to feed into* `discretize_stationary_cg`*.*
- *Use* `RectDomain` *to specify* Ω *and the Dirichlet/Neumann parts of* $\partial\Omega$*.*
- *Use* `ExpressionFunction` *to define* σ *and* f*. Start with a non-parametric version of the diffusivity* σ*. To refer to the first coordinate use* `x[0]` *in the definition of your* `ExpressionFunction`*. Use* `x[1]` *to refer to the second coordinate. Use* `{'bar': 1}` *to specify that the* `ExpressionFunction` σ *should depend on a single parameter* `bar`*, which is a vector of dimension 1.*

(b) Add an output functional s to the model, given by the integral

$$s(u) := \int_{\Omega_{out}} u(x,y)\,\mathrm{d}x\mathrm{d}y,$$

where $\Omega_{out} := \{x, y \in \mathbb{R} \,|\, (x-0.75)^2 + (y-0.75)^2 < 0.01\}$. Plot the parameter-to-output map.

Hints:
- *To specify an output functional, use the* **`outputs`** *parameter of* ***`StationaryProblem`***.
- *Use the* **`output`** *method of* ***`StationaryModel`*** *to compute the output.*

(c) Solve the time-dependent version of the problem given by

$$\begin{aligned} \partial_t u(x,y,t;\mu)\nabla\cdot(-\sigma(x,y;\mu)\nabla u(x,y,t;\mu)) &= f(x,y) && (x,y)\in\Omega, t\in(0,10),\\ u(x,y,t;\mu) &= 0 && (x,y)\in\partial\Omega, y=0, t\in(0,10),\\ \sigma(x,y;\mu)\nabla u(x,y,t;\mu)\cdot n(x,y) &= 0 && (x,y)\in\partial\Omega, y\neq 0, t\in(0,10),\\ u(x,y,0;\mu) &= 0 && (x,y)\in\Omega. \end{aligned}$$

Visualize the solution for several parameters and plot the time-to-output map.

Hint: Construct an ***`InstationaryProblem`*** *from your given* ***`StationaryProblem`*** *and feed it into* ***`discretize_instationary_cg`***.

Problem 2 (1D diffusion problem)

Apart from 2D models, pyMOR's builtin discretization toolkit also supports 1D problems. Discretize the boundary value problem

$$\begin{aligned} (-\sigma(x;\mu)\cdot u'(x;\mu))' &= f(x) && x\in(-1,1),\\ u(-1;\mu) &= 0,\\ u(1;\mu) &= 0, \end{aligned}$$

where the source term $f(x)$ and the diffusivity $\sigma(\sigma;\mu)$ are given by

$$f(x)=\begin{cases}1 & x<0\\ 0 & x>0\end{cases} \qquad \text{and} \qquad \sigma(x;\mu)=\begin{cases}1 & x<0\\ e^{\mu} & x>0.\end{cases}$$

Solve the resulting model for a few parameter values and visualize the solution.

Hint: Use **`LineDomain`** *to specify a one-dimensional domain.*

Problem 3 (Solving advection-diffusion equations)

So far we have only considered pure diffusion equations. In this exercise we will add an advection term.

(a) Discretize and solve the following boundary value problem for different values of μ:

$$\begin{aligned} -\Delta u(x,y;\mu)+\mu\cdot\nabla\cdot\left(\begin{bmatrix}-y\\ x\end{bmatrix}\cdot u(x,y;\mu)\right) &= f(x,y) && (x,y)\in\Omega:=(-1,1)\times(-1,1),\\ u(x,y;\mu) &= 0 && (x,y)\in\partial\Omega. \end{aligned}$$

The source term $f(x, y)$ is given as

$$f(x,y) = \begin{cases} 1 & (x-0.5)^2 + y^2 < 0.01 \\ 0 & \text{otherwise.} \end{cases}$$

Hint: *Use the* `advection` *parameter of* `StationaryProblem` *to specify the flux field* $[-y, x]^T$.

(b) Also solve the time-dependent version of this problem.

Problem 4 (Unstructured meshes and Robin boundary conditions)

pyMOR's discretization toolkit also supports unstructured triangle meshes created with Gmsh. These can be read using `pymor.discretizers.builtin.grids.gmsh.load_gmsh`. In this exercise, we will use pyMOR `domaindescriptions` that are automatically transformed into a Gmsh geometry definition for meshing.

(a) Solve the Poisson equation

$$\begin{aligned} -\Delta u(x) &= f(x) && x \in \Omega, \\ u(x) &= 0 && x \in \partial\Omega, \end{aligned}$$

where the domain Ω is the circular sector defined by

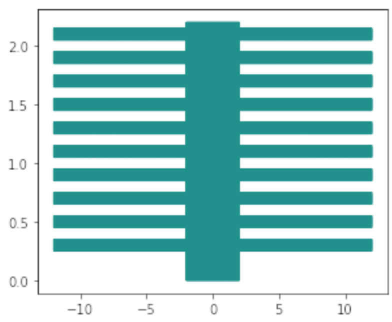
$$\Omega := \left\{ \begin{bmatrix} r \cdot \cos(\phi) \\ r \cdot \sin(\phi) \end{bmatrix} \middle| \; 0 \leq r < 1, \; 0 \leq \phi < 1.9 \cdot \pi \right\}.$$

Hint: *Use* `CircularSectorDomain` *to define* Ω.

(b) Solve

$$\begin{aligned} -\Delta u(x) &= f(x) && x \in \Omega, \\ \nabla u(x) \cdot n &= 1 && x \in \partial\Omega \cap \mathbb{R} \times \{0\}, \\ u(x) &= 0 && x \in \partial\Omega \setminus \mathbb{R} \times \{0\}, \end{aligned}$$

on the domain Ω given by the following heat-sink geometry:



Hint: *Use `PolygonalDomain` to define* Ω*.*

(c) Let's solve a physically somewhat more realistic model by imposing Robin boundary conditions on the fins, of the heat sink, i.e., solve

$$\begin{aligned} -\sigma \cdot \Delta u(x) &= f(x) && x \in \Omega, \\ \sigma \cdot \nabla u(x) \cdot n &= 80 && x \in \partial\Omega \cap \mathbb{R} \times \{0\}, \\ -\sigma \cdot \nabla u(x) \cdot n &= 1 \cdot (u(x) - 24) && x \in \partial\Omega \setminus \mathbb{R} \times \{0\}, \end{aligned}$$

with $\sigma = 10^3$ for the same heat-sink domain Ω as before.

Hint: *Pass the (constant) Robin data functions 1 and 24 as a tuple to `StationaryProblem.__init__` via the `robin_data` parameter.*

Problem 5 (Parameter Separation)

The models defined in problems 1, 2 and 3 are parameter separable. Reformulate the definitions of the corresponding `StationaryProblems` such that the resulting discrete `Models` reflect that structure.

Hints:

- *Use `LincombFunction` to define the relevant data functions as a linear combination of non-parametric `Functions` with appropriate constants or `ParameterFunctionals` as coefficients.*
- *To specify a $\theta_q(\mu)$ of the form $\theta_q(\mu) = \mu_i$, use `ProjectionParameterFunctional`. For arbitrary expressions in μ use `ExpressionParameterFunctional`.*
- *`discretze_stationary_cg` and `discretize_instationary_cg` automatically detect `LincombFunctions` and assemble corresponding matrices $\mathbb{A}^{(q)}$.*
- *If the parameter separation was successful, subsequent calls to `solve` should not require the assembly of any finite-element matrix. Check pyMOR's log output to verify that this is the case.*

Problem 6 (Multiple Parameters)

We add an additional parameter to the model in problem 1 and solve for $\mu \in (\mathbb{R}^{>0})^2$ the PDE

$$\begin{aligned} \nabla \cdot (-\sigma(x, y; \mu) \nabla u(x, y; \mu)) &= f(x, y; \mu) && (x, y) \in \Omega, \\ u(x, y; \mu) &= 0 && (x, y) \in \partial\Omega,\ y = 0, \\ \sigma(x, y; \mu) \nabla u(x, y; \mu) \cdot n(x, y) &= 0 && (x, y) \in \partial\Omega,\ y \neq 0, \end{aligned}$$

where

$$\sigma(x, y; \mu) := \begin{cases} \mu_1 & x \in (0.45, 0.55), y \in (0.5, 1) \\ 1 & \text{otherwise,} \end{cases}$$

and

$$f(x, y; \mu) := \begin{cases} 100 & (x - 0.25)^2 + (y - 0.75)^2 < 0.01 \\ \mu_2 & \text{otherwise.} \end{cases}$$

Extend your problem definition to include the additional parameter. Ensure parameter separation.

Problem 7 (Orthogonal Projection onto Reduced Space)

In this exercise we will construct a reduced space from some random snapshot data and compute the best-approximation error w.r.t. this space. In the following you can choose any of the parametric discrete models you have created in the previous exercises.

(a) Build a `ParameterSpace` for your problem by either specifying `parameter_ranges` when constructing the `StationaryProblem` or by directly constructing a `ParameterSpace`. Use the `sample_randomly` method to create a number of `Mu` instances holding random parameter values from this space.

(b) Collect the corresponding solution snapshots in a `VectorArray` `U`. We will use these snapshot vectors as a basis for our reduced space V_N.

(c) The best-approximation $u_N^*(\mu) \in V_N$ of $u_h(\mu)$ in V_N satisfying

$$\|u_h(\mu) - u_N^*(\mu)\| = \inf_{v_N \in V_N} \|u_h(\mu) - v_N(\mu)\|$$

is given by the orthogonal projection of $u_h(\mu)$ onto V_N. Hence, $u_N^*(\mu)$ satisfies:

$$(u_N^*(\mu), v_N) = (u_h(\mu), v_N) \qquad \forall v_N \in V_N. \tag{1}$$

Representing $u_N^*(\mu)$ as $u_N^*(\mu) = \sum_{i=1}^N \underline{u}_{N,i}^*(\mu) u_i$ where u_i denote the vectors in `U`, find a linear system corresponding to (1) which determines $\underline{u}_N^*(\mu)$ for given μ.

(d) Assemble the linear system using pyMOR and determine the solution $\underline{u}_N^*(\mu)$. In (1) use both the Euclidean and the H^1-inner product to compute a best approximation w.r.t. these norms. Reconstruct $u_N^*(\mu)$ from $\underline{u}_N^*(\mu)$. Visualize $u_N^*(\mu)$ alongside $u_h(\mu)$.

(e) Compute the maximum/average error $\|u_h(\mu) - u_N^*(\mu)\|$ in the Euclidean and H^1-norm for a validation set of random parameters μ. Verify that the error is zero for the μ used to build `U`.

Hints:

- *To create an empty* `VectorArray` *of suitable type, use the* `empty` *method of the* `solution_space` *of your model. Use the* `append` *method of the array to append the solution snapshots to it.*
- *To assemble* (1) *use the* `inner` *and* `gramian` *methods of* `VectorArray`*. Use* `lincomb` *to reconstruct* $u_N^*(\mu)$*. Norms are computed using the* `norm` *method.* `discrete_stationary_cg` *automatically assembles several inner product* `Operators`*, which are available as attributes of the resulting discrete* `Model`*.*

Problem 8 (Manual Reduced Basis Projection)

In the last problem we have constructed reduced spaces for parametrized problems using random snapshot data, and we have computed the best-approximation error w.r.t. to these spaces. We will now compute the Galerkin projection into these spaces and compare it with the best-approximation.

(a) Using the basis `VectorArray U`, compute the reduced system matrix $\mathbb{A}^{(N)}(\mu)$ and right-hand side vector $\mathbb{F}^{(N)}$. Solve the resulting linear equation system to determine $\underline{u}_N(\mu)$. Reconstruct $u_N(\mu)$.

(b) Compute the MOR error $\|u_h(\mu) - u_N(\mu)\|_1$ and compare it with the best-approximation error $\|u_h(\mu) - u_N^*(\mu)\|_1$. Compute the maximum/average errors over a validation set of random parameters. Plot these errors in dependence on the basis size. Can you avoid re-assembling the corresponding linear systems for smaller basis sizes?

(c) Measure the times required for assembling $\mathbb{A}^{(N)}(\mu)$, solving for $\underline{u}_N(\mu)$ and reconstructing $u_N(\mu)$. Plot theses timings in dependence on the basis size.

(d) Exploit the parameter separability and pre-assemble $\mathbb{A}^{(N,q)}$. Also measure the time needed to assemble $\mathbb{A}^{(N)}$ using these matrices. Verify that you obtain the same result.

Hints:

- *A* `StationaryModel` *stores the bilinear form* a *in the* `operator` *attribute,* ℓ *is given by the* `rhs` *attribute.*
- *To interpret the* `Operator fom.operator` *as a bilinear form and evaluate it, use the* `apply2` *method.*
- ℓ *is encoded as a linear* `Operator` *mapping real numbers* x *to the coefficient vector* $x \cdot \mathbb{F}$*. To obtain a* `VectorArray` *containing* $\mathbb{F}$ *use the* `as_vector` *method.*
- *Parameter separation in pyMOR* `Models` *is encoded using* `LincombOperators`*. These hold the summands* $\mathbb{A}^{(q)}$ *in the* `operators` *attribute. The corresponding* `ParameterFunctionals` *are stored in the* `coefficients` *attribute.*

Problem 9 (Automatic Operator Projection)

In pyMOR, the (Petrov)-Galerkin projection of `Operators` is handled by the `project` method.

Update your code to use `project` and construct a reduced `StationaryModel` from the resulting reduced `Operators`. What happens if you project a `LincombOperator`? What happens if your parametric `Operator` is not decomposed as a `LincombOperator`?

Problem 10 (Error-vs-Parameter Plot)

Choose a full-order model with one- or two-dimensional parameter domain, build a reduced order model from random parameter samples, and plot the model order reduction error over the parameter domain. Use a logarithmic scale for the error.

Problem 11 (Reductors)

Instead of manually projecting each `Operator` of a `Model` and constructing a reduced `Model` from the projected `Operators`, we can use a `Reductor` to facilitate the process.

(a) Modify your existing code to use the `reduce` method of `StationaryRBReductor` to build the ROM. Use the `reconstruct` method to reconstruct finite-element vectors from the reduced solutions.

(b) Use `CoerciveRBReductor` instead of `StationaryRBReductor` to additionally assemble an error estimator for the ROM. Plot the actual and estimated errors over the parameter domain.

(c) Use `estimator.reduce(N)` to quickly obtain a ROM for V_N when the ROM for $V_{N'}$, $N < N'$, has already been computed. Plot the maximum MOR error in dependence on the basis size.

Problem 12 (Greedy algorithm with pyMOR)

Greedy algorithms for constructing reduced approximation spaces can be found in pyMOR's `algorithms.greedy` and `algorithms.adaptivegreedy` modules.

(a) Use `rb_greedy` to build a reduced basis with the estimated MOR error as a surrogate for the best-approximation error. Plot the maximum MOR error on the training set and on a validation set in dependence on the basis size. Compare the result with reduced spaces obtained from random parameter selection. Also plot the MOR error over the parameter domain.

(b) Set `use_error_estimator` to `False` to study the effect of the error estimator.

(c) Specify a `WorkerPool` to parallelize the greedy search.

(d) Try `rb_adaptive_greedy` as a replacement for `rb_greedy`.

(e) Write a `strong_greedy` method which produces a strong greedy sequence for a given `VectorArray` of snapshot vectors to approximate. Compare the quality of the resulting ROM with the weak greedy ROM.

Problem 13 (POD-Greedy)

In this exercise we will reduce a parametric, time-dependent diffusion-advection-reaction equation using the POD-greedy algorithm, which employs a greedy search in the parameter domain and uses POD to extract low-rank spaces in the time domain.

(a) Create a discrete model for the diffusion-advection-reaction equation

$$\partial_t u(x,y,t,\mu) - \mu_d \cdot \Delta u(x,y,t,\mu) + \partial_x u(x,y,t,\mu) + \mu_r \cdot u(x,y,t,\mu) = f(x,y)$$

with $(x,y) \in \Omega := (-1,1) \times (-1,1),\ t \in (0,0.5)$ and boundary/inital conditions

$$\begin{aligned} u(x,y,t,\mu) &= 0 && (x,y) \in \partial\Omega,\ t \in (0,0.5) \\ u(x,y,0,\mu) &= u_0(x,y) && (x,y) \in \Omega. \end{aligned}$$

Here, the initial data u_0 is given by

$$u_0(x,y) = \begin{cases} 1 & x^2 + y^2 < 0.04 \\ 0 & \text{otherwise} \end{cases}$$

and the source term f is given by

$$f(x,y) = \begin{cases} 1 & (x+0.5)^2 + (y+0.5)^2 < 0.04 \\ 0 & \text{otherwise.} \end{cases}$$

Specify `parameter_ranges` of $[0.01, 1]$ for μ_d and of $[0, 100]$ for μ_r. Use continuous finite elements with a `diameter` of $1/100$ and 10 time steps. Visualize the solution for some combinations of parameter values.

(b) To compute a basis using POD-greedy, we first need a reductor that assembles an online-efficient error estimator for the ROM. Since the problem is of parabolic type, we can use `ParabolicRBReductor` for that, which will provide an estimator that bounds the error measure

$$\left[C_a^{-1}(\mu) \|e_N(\mu)\|^2 + \sum_{n=1}^{N} \Delta t \|e_n(\mu)\|_e^2 \right]^{1/2},$$

where $\|\cdot\|$ denotes the L^2-norm, $\|\cdot\|_e$ an energy norm w.r.t. which the bilinear form of the spatial differential operator is coercive, $C_a(\mu)$ is a lower bound for the coercivity constant, Δt is the time-step size, N the number of time steps, and $e_n(\mu)$ is the error at time step n for parameter values μ. So, in particular, this quantity is an upper bound for discrete version of the space-time energy error

$$\left[\int_0^T \|e(t,\mu)\|^2 \right]^{1/2}.$$

As the energy norm use the norm induced by the `fom.h1_0_semi_product`, for which the coercivity constant is simply given by μ_d.

In the POD-greedy algorithm we do the following in each iteration:

(i) Determine μ^* for which the estimated space-time error is maximal.

(ii) Compute the FOM solution $u(\mu^*)$ and the ROM solution $u_{red}(\mu^*)$.

(iii) Compute the orthogonal-projection $u_{proj}(\mu^*)$ of $u_{red}(\mu^*)$ onto the reduced basis at each time instance.

(iv) Compute a POD of the projection defects $u_\perp(t, \mu^*) := u_{red}(t, \mu^*) - u_{proj}(t, \mu^*)$.

(v) Extend the basis with a certain number of POD modes of $u_\perp$.

This will all happen automatically for you in `rb_greedy` when the algorithm detects that `solve` returns `VectorArrays` with more than one solution vector. By default, one POD mode per iteration will be added to the basis.

Compute a POD-greedy basis for a training set of 20×20 uniformly sampled parameter values. Specify an absolute error tolerance of 10^{-2}.

(c) Compute the model order reduction error and the estimated error for a test set of 30 randomly sampled parameter values. Determine the maximum and minimum ratio between error and estimated error. Visualize the FOM and ROM solutions as well as their difference for the parameter values maximizing the error. Also compute the ROM speedup.

(d) Since `rb_greedy` only adds one POD mode per iteration, it can happen that the same parameter values are selected multiple times during basis generation. Hence, we can save some offline time by caching the FOM solutions. To do so with pyMOR, we have to activate caching by calling `fom.enable_caching('disk')`. Compute the POD-greedy basis again with caching enabled. Compare the timings.

(e) Disk-based caching is only possible when the solution `VectorArrays` can be serialized using the `pickle` protocol. When using an external solver, this might not be the case. As an alternative, we can use memory-based caching using `fom.enable_caching('memory')`. Another approach to accelerate the offline phase is to add more than one POD mode to the basis per iteration, possibly at the expense of a slightly bigger final reduced basis. To do this, we need to pass appropriate `pod_modes` in the `extension_params` dict to `rb_greedy`.

Disable caching again by calling `fom.disable_caching`. Build a new POD-greedy basis by adding 3 POD modes per iteration. Compare the offline times as well as the quality of the ROM on the test set from part c).

Hints:

- *Use the* `reaction` *parameter of* `StationaryProblem.__init__` *to specify a constant reaction coefficient of value 1.*
- *You need to provide a (lower bound)* `coercivity_estimator` *for* C_a *to* `ParabolicRBReductor.__init__`*. In this case you can use* `ProjectionParameterFunctional('diffusion', 1)`*.*

Problem 14 (Reducing a model with output)

So far we have only looked at state-space approximations of the FOM solutions. In this exercise we will build a ROM with an output functional that can be efficiently evaluated without depending on any full-order calculations.

(a) In this exercise we will work again with the heat-sink model from Exercise 4. Add the output functional $\ell(u)$ to the model which is given by the average temperature at the base of the heat sink:

$$\ell(u) := \frac{1}{|\Gamma_b|} \int_{\Gamma_b} u(s)\,\mathrm{ds}, \qquad \Gamma_b := \partial\Omega \cap \mathbb{R} \times \{0\}.$$

Let the constant diffusion coefficient d be a parameter of the model. Plot of the base temperature in dependence of the diffusion coefficient for $d \in [1, 10^5]$.

(b) Use `scipy.optimize.bisect` to determine the diffusion coefficient $d \in [1, 10^5]$ for which the base temperature is 45. How many solutions of the FOM are required?

(c) Create a reduced basis from 5 logarithmically spaced solution snapshots of the FOM. Manually build a ROM by using the `project` method. Again determine d for base temperature 45, this time using the ROM in the `bisect` call.

(d) pyMOR's reductors automatically project the output functional for you. Rebuild the ROM for the same basis using `StationaryRBReductor`. Plot the diffusion coefficient in dependence of the base temperature using repeated `bisect` calls.

Note: Of course, this exercise has to be taken with a grain of salt. `bisect` is not a very efficient root-finding algorithm and the default tolerance for convergence is much smaller than the model order reduction error. The main purpose of this exercise is to show that the ROM output can be used as a more efficient drop-in replacement for the FOM output. If you want to learn more about using ROMs in the context of optimization, see the 'Model order reduction for PDE-constrained optimization problems' tutorial in pyMOR's documentation.

Problem 15 (Working with affine spaces)

By now we have only considered PDEs with homogeneous Dirichlet boundary conditions, causing the ansatz space for the weak formulation to be a linear space. For problems with non-zero Dirichlet boundaries, the ansatz space will be an affine space. In this exercise we will treat the reduction of problems with affine solution spaces.

(a) Create a discrete model for the 2×2 `thermal_block_problem`. However, change the boundary condition to be given by:

$$u(x, y) = \begin{cases} 1 & y = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{for} \qquad x \in \partial\Omega.$$

(b) Use `to_numpy()` and `np.where` on a solution of the model to determine the degrees of freedom associated with the non-zero Dirichlet boundary. Solve for different parameter values to check that, indeed, all solutions are exactly one at these DOFs.

(c) Let's just ignore that the solutions lie in an affine space. Build a ROM from 5 random solution snapshots. Solve FOM and ROM for new parameter values and compute the error. Is the boundary condition fulfilled?

(d) We see that, although the boundary condition is not fulfilled exactly, it is fulfilled up to a quite small approximation error. In many cases, it is actually quite feasible to just ignore the fact that the ansatz space is an affine subspace of the discrete function space and work with this entire space. In fact, for many discretization methods, like finite volume methods, Dirichlet boundary conditions are only weakly enforced. However, there are many reasons to work with affine spaces and to enforce the ROM solution to exactly lie in this affine space. In particular:

- There are many different ways to realize the handling of Dirichlet boundary conditions in PDE solvers, and it is often unclear/unknown how these affect the ROM. This not only affects the ansatz space but also the test space which would normally consist only of functions which are zero at the Dirichlet boundaries. In particular, most implementations retain the degrees of freedom related to the Dirichlet boundaries and modify the system matrix to include an equation forcing the associated degrees of freedom to have the right value. (This is also what is done in pyMOR's builtin discretization toolkit.) In the ROM, these equations will in effect put a penalty on the violation of the boundary conditions. The weighting of this penalty is usually unknown, however. So there is no way of controlling how well the boundary conditions will be fulfilled.
- Often system matrices are also used to define energy norms. Due to boundary treatment, these matrices often are no longer symmetric even though their corresponding bilinear form is. Using those matrices to orthogonalize functions with non-zero Dirichlet boundaries will cause errors.
- Sometimes, in particular for non-linear problems, it might be a requirement to choose ansatz functions from an appropriate affine space, since an application of the 'Operator' to functions outside this space might cause internal errors inside the PDE solver.
- For some problems, e.g., elliptic problems with pure Neumann boundary conditions, the ROM equations may become ill-posed when an inappropriate ansatz space is chosen.
- Sometimes it may be a user requirement that the Dirichlet constraints are exactly fulfilled.

A standard way to mitigate issues with affine ansatz spaces is to reformulate the problem to have solutions in a linear space. In particular, consider a standard linear `StationaryModel` of the form

$$A(\mu) \cdot u(\mu) = f.$$

Assume that u_{aff} is any function from the affine space, e.g. a function with the correct boundary values, then we can decompose $u(\mu)$ as

$$u(\mu) = u_{aff} + u_0(\mu)$$

and solve

$$A(\mu) \cdot [u_{aff} + u_0(\mu)] = f$$

for $u_0(\mu)$. If we now collect snapshots $u_0(\mu_i)$ and construct a linear subspace V_N from the linear span of these snapshots, then we can build the ROM using ansatz functions from

$$u_{aff} + V_N,$$

which will be an affine subspace of the affine solution space. In the case of Dirichlet boundary conditions, V_N will only consist of functions with zero boundary values.

We still need to build an online-efficient ROM from our ansatz. In the linear case this is quite straightforward, as we can rewrite the equation system as

$$A(\mu) \cdot u_0(\mu) = f - A(\mu) \cdot u_{aff}.$$

This means that we solve an equation system with the same system matrix, but with a modified right-hand side.

Build a `StationaryModel` for this equation system and check that its solutions have zero boundary values. Reduce the modified ROM using snapshot solutions from the original FOM. Reconstruct a full-order function for the ROM solution and compare the error with your earlier ROM solution. Verify that the Dirichlet condition is exactly fulfilled.

(e) Use `StationaryModel.deaffinize` to let pyMOR construct the modified FOM for the given shift vector u_{aff}.

Hints:

- *To create the FOM, first call `thermal_block_problem` to obtain a standard thermal-block problem with homogeneous Dirichlet boundary conditions. Use `with_` to exchange `dirichlet_data` with an appropriate `ExpressionFunction`.*
- *Use a solution of the FOM for the definition of u_{aff}.*
- *To build the shifted FOM, first wrap u_{aff} as an Operator using `VectorOperator`. Use `@` to concatenate `fom.operator` with your `VectorOperator` and subtract it from `fom.rhs`. Then, use `fom.with_` to get a new FOM with replaced right-hand side.*
- *pyMOR should automatically handle the correct offline/online decomposition of the new right-hand side when projecting (check that).*