Efficient space–time reduced order model for linear dynamical systems in Python using less than 120 lines of code

Youngkyu Kim[*‡] Karen May Wang[†‡] Youngsoo Choi[§]

Abstract

A classical reduced order model (ROM) for dynamical problems typically involves only the spatial reduction of a given problem. Recently, a novel space–time ROM for linear dynamical problems has been developed [1], which further reduces the problem size by introducing a temporal reduction in addition to a spatial reduction without much loss in accuracy. The authors show an order of a thousand speed-up with a relative error of less than 10^{-5} for a large-scale Boltzmann transport problem. In this work, we present for the first time the derivation of the space–time Petrov–Galerkin projection for linear dynamical systems and its corresponding block structures. Utilizing these block structures, we demonstrate the ease of construction of the space–time ROM method with two model problems: 2D diffusion and 2D convection diffusion, with and without a linear source term. For each problem, we demonstrate the entire process of generating the full order model (FOM) data, constructing the space–time ROM, and predicting the reduced-order solutions, all in less than 120 lines of Python code. We compare our Petrov–Galerkin method with the traditional Galerkin method and show that the space–time ROMs can achieve $\mathcal{O}(10^2)$ speed-ups with $\mathcal{O}(10^{-3})$ to $\mathcal{O}(10^{-4})$ relative errors for these problems. Finally, we present an error analysis for the space–time Petrov–Galerkin projection and derive an error bound, which shows an improvement compared to traditional spatial Galerkin ROM methods.

Keywords— space–time reduced order model, Python codes, proper orthogonal decomposition, linear dynamical systems, least-squares Petrov–Galerkin projection, error bound

1 Introduction

Many computational models for physical simulations are formulated as linear dynamical systems. Examples of linear dynamical systems include, but are not limited to, the Schrödinger equation that arises in quantum mechanics, the computational model for the signal propagation and interference in electric circuits, storm surge prediction models before an advancing hurricane, vibration analysis in large structures, thermal analysis in various media, neurotransmission models in the nervous system, various computational models for micro-electro-mechanical systems, and various particle transport simulations. These linear dynamical systems can quickly become large scale and computationally expensive, which prevents fast generation of solutions. Thus, areas in design optimization, uncertainty quantification, and controls where large parameter sweeps need to be done can become intractable, and this motivates the need for developing a Reduced Order Model (ROM) that can accelerate the solution process without loss in accuracy.

Many ROM approaches for linear dynamical systems have been developed, and they can be broadly categorized as data-driven or non data-driven approaches. We give a brief background of some of the methods here. For the non data-driven approaches, there are several methods, including: balanced truncation methods [2–10], moment-matching methods [11–15], and Proper Generalized Decomposition (PGD) [16] and its extensions [17–26]. The balanced truncation method is by far the most popular method, but it requires the solution of two Lyapunov equations to construct bases, which is a formidable task in large-scale problems. Moment matching methods were originally developed as non data-driven, although later papers extended the method to include it. They provide a computationally efficient framework using Krylov subspace techniques in an iterative fashion where only matrix-vector multiplications are required. The optimal H_2 tangential interpolation for nonparametric systems [12] is also available. Proper Generalized Decomposition was first developed as a numerical method for solving boundary value problems. It utilizes techniques to separate space and time for an efficient solution procedure and is considered a model reduction technique. For the detailed description of PGD, we refer to a short review paper [27]. Many data driven ROM approaches have been developed as well. When datasets are available either from experiments or high-fidelity simulations, these datasets can contain rich information about the system of interest and utilizing this in the construction of a ROM can produce an optimal basis. Although there are some data-driven moment matching works available [28, 29], two popular methods are Dynamic Mode Decomposition (DMD) and Proper Orthogonal decomposition (POD). DMD generates reduced modes that embed an intrinsic temporal behavior and was first developed by Peter Schmid [30]. The method has been actively developed and extended to many applications [31–38]. For a more detailed description about DMD, we refer to this preprint [39] and book [40]. POD utilizes the method of snapshots to obtain an optimal basis of a system and typically applies only to spatial projections, although temporal projection techniques have been developed as well [41–53].

In our paper, we focus on building a space–time ROM where both spatial and temporal projections are applied to achieve an optimal reduction. This method has been developed by previous authors [54–57], and a space–time ROM for large-scale linear dynamical systems has been recently introduced [1]. The authors show a speed-up of $> 8,000$ with good accuracy for a large-scale transport problem. In our work, we present several new contributions on the space–time ROM development:

- We derive the block structures of least–squares Petrov–Galerkin space–time ROM operators and compare them with the Galerkin space–time ROM operators and show that the computational cost saving due to the block structure is a factor of the FOM spatial degrees of freedom.

[*]Mechanical Engineering, University of California, Berkeley, CA 94720 (youngkyu_kim@berkeley.edu)
[†]Design physicist, Lawrence Livermore National Laboratory, Livermore, CA, USA (wang79@llnl.gov)
[‡]These authors contributed equally to this work.
[§]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550 (choi15@llnl.gov)

- We present an error analysis of both Galerkin and least–squares Petrov–Galerkin space–time ROMs and demonstrate the growth rate of the stability constant with the actual space–time operators used in our numerical results.
- Utilizing the block structures derived, we demonstrate the ease of implementing both Galerkin and least–squares Petrov–Galerkin space–time ROM implementations and provide the source code for three canonical problems. For each problem, we cover the entire space–time ROM process in less than 120 lines of Python code, which includes sweeping a wide parameter space and generating data from the full order model, constructing the space–time ROM, and generating the ROM prediction in the online phase.
- Finally, we present our results for the two model problems and compare the speed up and relative error between the Galerkin and Petrov–Galerkin methods and show that they give similar results.

We hope that by providing full access to the Python source codes, researchers can easily apply space–time ROMs to their linear dynamical problem of interest. Furthermore, we have curated the source codes to be simple and short so that it may be easily extended in various multi-query problem settings, such as design optimization [58–64], uncertainty quantification [65–67], and optimal control problems [68–70].

1.1 Organization of the paper

The paper is organized in the following way: Section 2 describes a parametric linear dynamical systems and space–time formulation. Section 3 introduces linear subspace solution representation in Section 3.1 and space–time ROM formulation using Galerkin projection in Section 3.2 and least–squares Petrov–Galerkin projection in Section 3.3. Then, both space–time ROMs are compared in Section 3.4. Section 4 describes how to generate space–time basis. We investigate block structures of space–time ROM basis in Section 5.1. We introduce the block structures of Galerkin space–time ROM operators derived in [1] in Section 5.2. In Section 5.3, we derive least-squares Petrov–Galerkin space–time ROM operators in terms of the blocks. Then, we compared Galerkin and least–squares Petrov–Galerkin block structures in Section 5.4. We compute computational complexity of forming the space–time ROM operators in Section 5.5. The error analysis is presented in Section 6. We demonstrate the performance of both Galerkin and least-squares Petrov–Galerkin space–time ROMs in two numerical experiments in Section 7. Finally, the paper is concluded with summary and future works in Section 8. Note that we use "least–squares Petrov–Galerkin" and "Petrov–Galerkin" interchangeably throughout the paper. Appendix A presents six Python codes with less than 120 lines that are used to generate our numerical results.

2 Linear dynamical systems

We consider the parameterized linear dynamical system shown in Equation (2.1).

$$\frac{\partial \boldsymbol{u}(t;\boldsymbol{\mu})}{\partial t} = \boldsymbol{A}(\boldsymbol{\mu})\boldsymbol{u}(t;\boldsymbol{\mu}) + \boldsymbol{B}(\boldsymbol{\mu})\boldsymbol{f}(t;\boldsymbol{\mu}), \quad \boldsymbol{u}(0;\boldsymbol{\mu}) = \boldsymbol{u}^0(\boldsymbol{\mu}), \tag{2.1}$$

where $\boldsymbol{\mu} \in \Omega_\mu \subset \mathbb{R}^{n_\mu}$ denotes a parameter vector, $\boldsymbol{u} : [0,T] \times \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s}$ denotes a time dependent state variable function, $\boldsymbol{u}^0 : \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s}$ denotes an initial state, and $\boldsymbol{f} : [0,T] \times \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_i}$ denotes a time dependent input variable function. The operators $\boldsymbol{A} : \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s \times N_s}$, $\boldsymbol{B} : \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s \times N_i}$, and are real valued matrices that are independent of state variables.

Although any time integrator can be used, for the demonstration purpose, we choose to apply a backward Euler time integration scheme shown in Equation (2.2):

$$\left(\boldsymbol{I}_{N_s} - \Delta t^{(k)}\boldsymbol{A}(\boldsymbol{\mu})\right)\boldsymbol{u}^{(k)} = \boldsymbol{u}^{(k-1)} + \Delta t^{(k)}\boldsymbol{B}(\boldsymbol{\mu})\boldsymbol{f}^{(k)}(\boldsymbol{\mu}), \tag{2.2}$$

where $\boldsymbol{I}_{N_s} \in \mathbb{R}^{N_s \times N_s}$ is the identity matrix, $\Delta t^{(k)}$ is the kth time step size with $T = \sum_{k=1}^{N_t} \Delta t^{(k)}$ and $t_{(k)} = \sum_{j=1}^{k} \Delta t^{(j)}$, and $\boldsymbol{u}^{(k)}(\boldsymbol{\mu}) := \boldsymbol{u}(t_{(k)};\boldsymbol{\mu})$ and $\boldsymbol{f}^{(k)}(\boldsymbol{\mu}) := \boldsymbol{f}(t_{(k)};\boldsymbol{\mu})$ are the state and input vectors at kth time step where $k \in \mathbb{N}(N_t)$. The Full Order Model (FOM) solves Equation (2.2) for every time step, where its spatial dimension is N_s and the temporal dimension is N_t. Each time step of the FOM can be written out and put in another matrix system shown in Equation (2.3). This is known as the space–time formulation.

$$\boldsymbol{A}^{\mathrm{st}}(\boldsymbol{\mu})\boldsymbol{u}^{\mathrm{st}}(\boldsymbol{\mu}) = \boldsymbol{f}^{\mathrm{st}}(\boldsymbol{\mu}) + \boldsymbol{u}_0^{\mathrm{st}}(\boldsymbol{\mu}), \tag{2.3}$$

where

$$\boldsymbol{A}^{\mathrm{st}}(\boldsymbol{\mu}) = \begin{bmatrix} \boldsymbol{I}_{N_s} - \Delta t^{(1)}\boldsymbol{A}(\boldsymbol{\mu}) & & & \\ -\boldsymbol{I}_{N_s} & \boldsymbol{I}_{N_s} - \Delta t^{(2)}\boldsymbol{A}(\boldsymbol{\mu}) & & \\ & \ddots & \ddots & \\ & & -\boldsymbol{I}_{N_s} & \boldsymbol{I}_{N_s} - \Delta t^{(N_t)}\boldsymbol{A}(\boldsymbol{\mu}) \end{bmatrix}, \tag{2.4}$$

$$\boldsymbol{u}^{\mathrm{st}}(\boldsymbol{\mu}) = \begin{bmatrix} \boldsymbol{u}^{(1)}(\boldsymbol{\mu}) \\ \boldsymbol{u}^{(2)}(\boldsymbol{\mu}) \\ \vdots \\ \boldsymbol{u}^{(N_t)}(\boldsymbol{\mu}) \end{bmatrix}, \tag{2.5}$$

$$\boldsymbol{f}^{\mathrm{st}}(\boldsymbol{\mu}) = \begin{bmatrix} \Delta t^{(1)}\boldsymbol{B}(\boldsymbol{\mu})\boldsymbol{f}^{(1)}(\boldsymbol{\mu}) \\ \Delta t^{(2)}\boldsymbol{B}(\boldsymbol{\mu})\boldsymbol{f}^{(2)}(\boldsymbol{\mu}) \\ \vdots \\ \Delta t^{(N_t)}\boldsymbol{B}(\boldsymbol{\mu})\boldsymbol{f}^{(N_t)}(\boldsymbol{\mu}) \end{bmatrix}, \tag{2.6}$$

$$\boldsymbol{u}_0^{\mathrm{st}}(\boldsymbol{\mu}) = \begin{bmatrix} \boldsymbol{u}^0(\boldsymbol{\mu}) \\ \boldsymbol{0} \\ \vdots \\ \boldsymbol{0} \end{bmatrix}. \tag{2.7}$$

The space–time system matrix $\boldsymbol{A}^{\text{st}}$ has dimensions $\mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s N_t \times N_s N_t}$, the space–time state vector $\boldsymbol{u}^{\text{st}}$ has dimensions $\mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s N_t}$, the space–time input vector $\boldsymbol{f}^{\text{st}}$ has dimensions $\mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s N_t}$, and the space–time initial vector $\boldsymbol{u}_0^{\text{st}}$ has dimensions $\mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s N_t}$. Although it seems that the solution can be found in a single solve, in practice there is no computational saving gained from doing so since the block structure of the space–time system will solve the system in a time-marching fashion anyways. However, we formulate the problem in this way since our reduced order model (ROM) formulation can reduce and solve the space–time system efficiently. In the following sections, we describe the parametric Galerkin and least–squares Petrov–Galerkin ROM formulations.

3 Space–time reduced order models

We investigate two projection-based space–time ROM formulations: the Galerkin and least–squares Petrov–Galerkin formulations. Here, we use "least-squares Petrov–Galerkin" and "Petrov–Galerkin" interchangeably throughout the paper.

3.1 Linear subspace solution representation

Both the Galerkin and Petrov–Galerkin methods reduce the number of space–time degrees of freedom by approximating the space–time state variables as a smaller linear combination of space–time basis vectors:

$$\boldsymbol{u}^{\text{st}}(\boldsymbol{\mu}) \approx \tilde{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}) \equiv \boldsymbol{\Phi}_{\text{st}} \hat{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}), \tag{3.1}$$

where $\hat{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}) : \mathbb{R}^{n_\mu} \to \mathbb{R}^{n_s n_t}$ with $n_s \ll N_s$ and $n_t \ll N_t$. The space–time basis, $\boldsymbol{\Phi}_{\text{st}} \in \mathbb{R}^{N_s N_t \times n_s n_t}$ is defined as

$$\boldsymbol{\Phi}_{\text{st}} \equiv \begin{bmatrix} \boldsymbol{\phi}_1^{\text{st}} & \cdots & \boldsymbol{\phi}_{i+n_s(j-1)}^{\text{st}} & \cdots \boldsymbol{\phi}_{n_s n_t}^{\text{st}} \end{bmatrix}, \tag{3.2}$$

where $i \in \mathbb{N}(n_s)$, $j \in \mathbb{N}(n_t)$. Substituting Equation (3.1) into the space–time formulation in Equation (2.3) gives an over-determined system of equations:

$$\boldsymbol{A}^{\text{st}}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}} \hat{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}) = \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu}) + \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu}) \tag{3.3}$$

This over-determined system of equations can be closed by either the Galerkin or Petrov–Galerkin projections.

3.2 Galerkin projection

In the Galerkin formulation, Equation 3.3 is closed by the Galerkin projection, where both sides of the equation is multiplied by $\boldsymbol{\Phi}_{\text{st}}^T$. Thus, we solve following reduced system for the unknown generalized coordinates, $\boldsymbol{u}^{\text{st}}$:

$$\boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}} \hat{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu}) + \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu}). \tag{3.4}$$

For notational simplicity, let us define the reduced space–time system matrix as $\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}}$, reduced space–time input vector as $\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu})$, and reduced space–time initial state vector as $\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu})$.

3.3 Least-squares Petrov–Galerkin projection

In the least-squares Petrov–Galerkin formulation, we first define the space–time residual as

$$\tilde{\boldsymbol{r}}^{\text{st}}(\hat{\boldsymbol{u}}^{\text{st}}; \boldsymbol{\mu}) := \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu}) + \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu}) - \boldsymbol{A}^{\text{st}}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}} \hat{\boldsymbol{u}}^{\text{st}} \tag{3.5}$$

where $\tilde{\boldsymbol{r}}^{\text{st}} : \mathbb{R}^{n_s n_t} \times \mathbb{R}^{n_\mu} \to \mathbb{R}^{N_s N_t}$. Note Equation (3.5) is an over-determined system. To close the system and solve for the unknown generalized coordinates, $\hat{\boldsymbol{u}}^{\text{st}}$, the least-squares Petrov–Galerkin method takes the squared norm of the residual vector function and minimize it:

$$\hat{\boldsymbol{u}}^{\text{st}} = \underset{\hat{\boldsymbol{v}} \in \mathbb{R}^{n_s n_t}}{\text{argmin}} \quad \frac{1}{2} \left\| \tilde{\boldsymbol{r}}^{\text{st}}(\hat{\boldsymbol{v}}; \boldsymbol{\mu}) \right\|_2^2. \tag{3.6}$$

The solution to Equation (3.6) satisfies

$$(\boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}})^T \tilde{\boldsymbol{r}}^{\text{st}}(\hat{\boldsymbol{u}}^{\text{st}}; \boldsymbol{\mu}) = \boldsymbol{0} \tag{3.7}$$

leading to

$$\boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}} \hat{\boldsymbol{u}}^{\text{st}}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu}) + \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu}). \tag{3.8}$$

For notational simplicity, let us define the reduced space–time system matrix as

$$\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}},$$

reduced space–time input vector as $\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu})$, and reduced space–time initial state vector as $\hat{\boldsymbol{u}_0}^{st,pg}(\boldsymbol{\mu}) := \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu})$.

3.4 Comparison of Galerkin and Petrov–Galerkin projections

The reduced space–time system matrices, reduced space–time input vectors, and reduced space–time initial state vectors for Galerkin and Petrov–Galerkin projections are presented in Table 1.

Table 1: Comparison of Galerkin and Petrov–Galerkin projections

Galerkin	**Petrov–Galerkin**
$\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}}$	$\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}}$
$\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu})$	$\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu})$
$\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu})$	$\hat{\boldsymbol{u}_0}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu})$

4 Space-time Basis Generation

In this section, we repeat Section 4.1 in [1] to be self-contained.

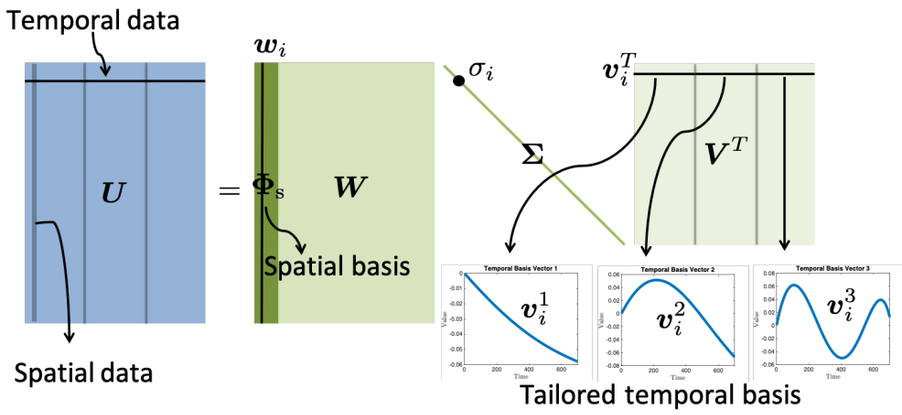


Figure 1: Illustration of spatial and temporal bases construction, using SVD with $n_\mu = 3$. The right singular vector, $\boldsymbol{v}_i$, describes three different temporal behaviors of a left singular basis vector $\boldsymbol{w}_i$, i.e., three different temporal behaviors of a spatial mode. Each temporal behavior is denoted as $\boldsymbol{v}_i^1$, $\boldsymbol{v}_i^2$, and $\boldsymbol{v}_i^3$.

We follow the method of snapshots described by Sirovich [71]. First, let $\{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_{n_\mu}\}$ be a set of parameter samples, where we run full order model simulations. Let $\boldsymbol{U}_p \equiv \begin{bmatrix}\boldsymbol{u}^{(1)}(\boldsymbol{\mu}_p) & \cdots & \boldsymbol{u}^{(N_t)}(\boldsymbol{\mu}_p)\end{bmatrix} \in \mathbb{R}^{N_s \times N_t}$, $p \in \mathbb{N}(n_\mu)$, be a full order model solution matrix for a sample parameter, $\boldsymbol{\mu}_p \in \Omega_\mu$. Then concatenating all the solution matrix defines a snapshot matrix, $\boldsymbol{U} \in \mathbb{R}^{N_s \times n_\mu N_t}$, i.e.,

$$\boldsymbol{U} \equiv \begin{bmatrix}\boldsymbol{U}_1 & \cdots & \boldsymbol{U}_{n_\mu}\end{bmatrix}. \tag{4.1}$$

We use Proper Orthogonal Decomposition (POD) to construct the spatial basis, $\boldsymbol{\Phi}_\mathrm{s}$. POD [41] obtains $\boldsymbol{\Phi}_\mathrm{s}$ by choosing the leading n_s columns of the left singular matrix, $\boldsymbol{W}$, of the following Singular Value Decomposition (SVD) with $\ell \equiv \min(N_s, n_\mu N_t)$ and $n_s < n_\mu N_t$:

$$\boldsymbol{U} = \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{4.2}$$

$$= \sum_{i=1}^{\ell} \sigma_i \boldsymbol{w}_i \boldsymbol{v}_i^T, \tag{4.3}$$

where $\boldsymbol{W} \in \mathbb{R}^{N_s \times \ell}$ and $\boldsymbol{V} \in \mathbb{R}^{n_\mu N_t \times \ell}$ are orthogonal matrices and $\boldsymbol{\Sigma} \in \mathbb{R}^{\ell \times \ell}$ is a diagonal matrix with singular values on its diagonal. The spatial POD basis, $\boldsymbol{\Phi}_\mathrm{s}$ minimizes

$$\left\| \boldsymbol{U} - \boldsymbol{\Phi}_\mathrm{s}\boldsymbol{\Phi}_\mathrm{s}^T\boldsymbol{U} \right\|_F^2 \tag{4.4}$$

over all $\boldsymbol{\Phi}_\mathrm{s} \in \mathbb{R}^{N_s \times n_s}$ with orthonormal columns, where $\|\cdot\|_F$ denotes the Frobenius norm. The POD procedure seeks the n_s-dimensional subspace that optimally represents the solution snapshot, $\boldsymbol{U}$. The equivalent summation form is written in (4.3), where $\sigma_i \in \mathbb{R}$ is ith singular value, $\boldsymbol{w}_i$ and $\boldsymbol{v}_i$ are ith left and right singular vectors, respectively. Note that $\boldsymbol{v}_i$ describes n_μ different temporal behavior of $\boldsymbol{w}_i$. For example, Figure 1 illustrates the case of $n_\mu = 3$, where $\boldsymbol{v}_i^1$, $\boldsymbol{v}_i^2$, and $\boldsymbol{v}_i^3$ describe three different temporal behavior of a specific spatial basis vector, i.e., $\boldsymbol{w}_i$. For general n_μ, we note that $\boldsymbol{v}_i$ describes n_μ different temporal behavior of ith spatial basis vector, i.e., $\boldsymbol{\phi}_i^\mathrm{s} = \boldsymbol{w}_i$. We set $\boldsymbol{\Upsilon}_i = \begin{bmatrix}\boldsymbol{v}_i^1 & \cdots & \boldsymbol{v}_i^{n_\mu}\end{bmatrix}$ to be ith temporal snapshot matrix, where $\boldsymbol{v}_i^k \equiv [\boldsymbol{v}_i(1 + (k-1)N_t), \boldsymbol{v}_i(2 + (k-1)N_t), \cdots, \boldsymbol{v}_i(kN_t)]^T \in \mathbb{R}^{N_t}$ for $k \in \mathbb{N}(n_\mu)$, where $\boldsymbol{v}_i(j), j \in \mathbb{N}(n_\mu N_t)$ is the jth component of the vector. The SVD of $\boldsymbol{\Upsilon}_i$ is

$$\boldsymbol{\Upsilon}_i = \boldsymbol{\Lambda}_i \boldsymbol{\Sigma}_i \boldsymbol{\Psi}_i^T. \tag{4.5}$$

Then, choosing the leading n_t vectors of $\boldsymbol{\Lambda}_i$ yields the temporal basis, $\boldsymbol{\Phi}_\mathrm{t}^i$ for ith spatial basis vector. Finally, we can construct a space–time basis vector, $\boldsymbol{\phi}_{i+n_s(j-1)}^\mathrm{st} \in \mathbb{R}^{N_s N_t}$, in Equation (3.2) as

$$\boldsymbol{\phi}_{i+n_s(j-1)}^\mathrm{st} = \boldsymbol{\phi}_{ij}^\mathrm{t} \otimes \boldsymbol{\phi}_i^\mathrm{s}, \tag{4.6}$$

where $\otimes$ denotes Kronecker product, $\boldsymbol{\phi}_i^\mathrm{s} \in \mathbb{R}^{N_s}$ is ith vector of the spatial basis, $\boldsymbol{\Phi}_\mathrm{s}$, and $\boldsymbol{\phi}_{ij}^\mathrm{t} \in \mathbb{R}^{N_t}$ is jth vector of the temporal basis, $\boldsymbol{\Phi}_\mathrm{t}^i$ that describes a temporal behavior of $\boldsymbol{\phi}_i^\mathrm{s}$.

5 Space-time reduced order models in block structure

We avoid building the space–time basis vector defined in Equation (4.6) because it requires much memory for storage. Thus, we can exploit the block structure of the matrices to save computational cost and storage of the matrices in memory. Section 5.2 introduces such block structures for the space–time Galerkin projection, while Section 5.3 shows block structures for the space–time Petrov–Galerkin projection. First, we introduce common block structures that appear both the Galerkin and Petrov–Galerkin projections in Section 5.1.

5.1 Block structures of space–time basis

Following [1]'s notation, we define the block structure of the space–time basis to be:

$$\boldsymbol{\Phi}_{st} = \begin{pmatrix} \boldsymbol{\Phi}_s \boldsymbol{D}_1^1 & \cdots & \cdots & \cdots & \boldsymbol{\Phi}_s \boldsymbol{D}_1^{n_t} \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ \vdots & \cdots & \boldsymbol{\Phi}_s \boldsymbol{D}_k^j & \cdots & \vdots \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ \boldsymbol{\Phi}_s \boldsymbol{D}_{N_t}^1 & \cdots & \cdots & \cdots & \boldsymbol{\Phi}_s \boldsymbol{D}_{N_t}^{n_t} \end{pmatrix} \in \mathbb{R}^{N_s N_t \times n_s n_t} \tag{5.1}$$

where the kth time step of the temporal basis matrix is a diagonal matrix defined as

$$\boldsymbol{D}_k^j = \begin{pmatrix} \phi_{1j,k}^t & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & 0 & \cdots & \vdots \\ \vdots & 0 & \phi_{ij,k}^t & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \phi_{n_s j,k}^t \end{pmatrix} \in \mathbb{R}^{n_s \times n_s} \tag{5.2}$$

where $\phi_{ij,k}^t \in \mathbb{R}$ is the kth element of $\boldsymbol{\phi}_{ij}^t \in \mathbb{R}^{N_t}$.

5.2 Block structures of Galerkin projection

As shown in Table 1, the reduced space–time Galerkin system matrix, $\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu})$ is:

$$\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{\text{st}} \tag{5.3}$$

Now, We define the block structure of this matrix as:

$$\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu}) = \begin{pmatrix} \hat{\boldsymbol{A}}_{(1,1)}^{st,g}(\boldsymbol{\mu}) & \cdots & \cdots & \cdots & \hat{\boldsymbol{A}}_{(1,n_t)}^{st,g}(\boldsymbol{\mu}) \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ \vdots & \cdots & \hat{\boldsymbol{A}}_{(j',j)}^{st,pg}(\boldsymbol{\mu}) & \cdots & \vdots \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ \hat{\boldsymbol{A}}_{(n_t,1)}^{st,g}(\boldsymbol{\mu}) & \cdots & \cdots & \cdots & \hat{\boldsymbol{A}}_{(n_t,n_t)}^{st,g}(\boldsymbol{\mu}) \end{pmatrix} \tag{5.4}$$

so that we can exploit the block structure of these matrices such that we do not need to form the entire matrix. We derive that $\hat{\boldsymbol{A}}_{(j',j)}^{st,g}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s \times n_s}$ where the (j', j)th block matrix is:

$$\hat{\boldsymbol{A}}_{(j',j)}^{st,g}(\boldsymbol{\mu}) = \sum_{k=1}^{N_t} \left(\boldsymbol{D}_k^{j'} \boldsymbol{D}_k^j - \Delta t^{(k)} \boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \boldsymbol{A}(\boldsymbol{\mu}) \boldsymbol{\Phi}_s \boldsymbol{D}_k^j \right) - \sum_{k=1}^{N_t - 1} \boldsymbol{D}_{k+1}^{j'} \boldsymbol{D}_k^j \tag{5.5}$$

The reduced space–time Galerkin input vector $\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s n_t}$ is

$$\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{f}^{\text{st}}(\boldsymbol{\mu}). \tag{5.6}$$

Again, utilizing the block structure of matrices, we compute jth block vector $\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu})_{(j)} \in \mathbb{R}^{n_t}$ to be:

$$\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu})_{(j)} = \sum_{k=1}^{N_t} \Delta t^{(k)} \boldsymbol{D}_k^j \boldsymbol{\Phi}_s^T \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k)}(\boldsymbol{\mu}). \tag{5.7}$$

Finally, the space–time Galerkin initial vector, $\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s n_t}$, can be computed as:

$$\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{\text{st}}^T \boldsymbol{u}_0^{\text{st}}(\boldsymbol{\mu}) \tag{5.8}$$

where the jth block vector, $\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu})_{(j)} \in \mathbb{R}^{n_s}$, is:

$$\hat{\boldsymbol{u}_0}^{st,g}(\boldsymbol{\mu})_{(j)} = \boldsymbol{D}_1^j \boldsymbol{\Phi}_s^T \boldsymbol{u}_0(\boldsymbol{\mu}). \tag{5.9}$$

5.3 Block structures of least-squares Petrov–Galerkin projection

As shown in Table 1, the reduced space–time Petrov–Galerkin system matrix, $\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu})$ is:

$$\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{st}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{A}^{st}(\boldsymbol{\mu}) \boldsymbol{\Phi}_{st} \tag{5.10}$$

Now, We define the block structure of this matrix as:

$$\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu}) = \begin{pmatrix} \hat{\boldsymbol{A}}_{(1,1)}^{st,pg}(\boldsymbol{\mu}) & \cdots & \cdots & \cdots & \hat{\boldsymbol{A}}_{(1,n_t)}^{st,pg}(\boldsymbol{\mu}) \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ \vdots & \cdots & \hat{\boldsymbol{A}}_{(j',j)}^{st,pg}(\boldsymbol{\mu}) & \cdots & \vdots \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ \hat{\boldsymbol{A}}_{(n_t,1)}^{st,pg}(\boldsymbol{\mu}) & \cdots & \cdots & \cdots & \hat{\boldsymbol{A}}_{(n_t,n_t)}^{st,pg}(\boldsymbol{\mu}) \end{pmatrix} \tag{5.11}$$

so that we can exploit the block structure of these matrices such that we do not need to form the entire matrix. We derive that $\hat{\boldsymbol{A}}^{st,pg}_{(j',j)}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s \times n_s}$ where the (j',j)th block matrix is:

$$\begin{aligned}\hat{\boldsymbol{A}}^{st,pg}_{(j',j)}(\boldsymbol{\mu}) =& \sum_{k=1}^{N_t} \Big[\boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_k^j\Big] \\ &+ \sum_{k=1}^{N_t-1} \Big[\boldsymbol{D}_k^{j'} \boldsymbol{D}_k^j - \boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k+1)} \boldsymbol{A}(\boldsymbol{\mu})\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_{k+1}^j \\ &- \boldsymbol{D}_{k+1}^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k+1)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_k^j\Big]\end{aligned} \tag{5.12}$$

The reduced space–time Petrov–Galerkin input vector $\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s n_t}$ is

$$\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{st}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{f}^{\mathrm{st}}(\boldsymbol{\mu}). \tag{5.13}$$

Again, utilizing the block structure of matrices, we compute jth block vector $\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu})_{(j)} \in \mathbb{R}^{n_t}$ to be:

$$\begin{aligned}\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu})_{(j)} =& \sum_{k=1}^{N_t} \Big[\boldsymbol{D}_k^j \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\Delta t^{(k)} \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k)}(\boldsymbol{\mu})\Big] \\ &- \sum_{k=1}^{N_t-1} \Big[\boldsymbol{D}_k^j \boldsymbol{\Phi}_s^T \Delta t^{(k+1)} \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k+1)}(\boldsymbol{\mu})\Big].\end{aligned} \tag{5.14}$$

Finally, the space–time Petrov–Galerkin initial vector, $\hat{\boldsymbol{u_0}}^{st,pg}(\boldsymbol{\mu}) \in \mathbb{R}^{n_s n_t}$, can be computed as:

$$\hat{\boldsymbol{u_0}}^{st,pg}(\boldsymbol{\mu}) = \boldsymbol{\Phi}_{st}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})^T \boldsymbol{u}_0^{\mathrm{st}}(\boldsymbol{\mu}) \tag{5.15}$$

where the jth block vector, $\hat{\boldsymbol{u_0}}^{st,pg}(\boldsymbol{\mu})_{(j)} \in \mathbb{R}^{n_s}$, is:

$$\hat{\boldsymbol{u_0}}^{st,pg}(\boldsymbol{\mu})_{(j)} = \boldsymbol{D}_1^j \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(1)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\boldsymbol{u}_0(\boldsymbol{\mu}). \tag{5.16}$$

5.4 Comparison of Galerkin and Petrov–Galerkin block structures

The block structures of space–time reduced order model operators are summarized in Table 2.

Table 2: Comparison of Galerkin and Petrov–Galerkin block structures

Galerkin	**Petrov–Galerkin**
$\hat{\boldsymbol{A}}^{st,g}(\boldsymbol{\mu})_{(j',j)} = \sum_{k=1}^{N_t} \Big(\boldsymbol{D}_k^{j'} \boldsymbol{D}_k^j - \Delta t^{(k)} \boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \boldsymbol{A}(\boldsymbol{\mu}) \boldsymbol{\Phi}_s \boldsymbol{D}_k^j\Big) - \sum_{k=1}^{N_t-1} \boldsymbol{D}_{k+1}^{j'} \boldsymbol{D}_k^j$	$\hat{\boldsymbol{A}}^{st,pg}(\boldsymbol{\mu})_{(j',j)} = \sum_{k=1}^{N_t} \Big[\boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_k^j\Big] + \sum_{k=1}^{N_t-1} \Big[\boldsymbol{D}_k^{j'} \boldsymbol{D}_k^j - \boldsymbol{D}_k^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k+1)} \boldsymbol{A}(\boldsymbol{\mu})\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_{k+1}^j - \boldsymbol{D}_{k+1}^{j'} \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k+1)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\boldsymbol{\Phi}_s \boldsymbol{D}_k^j\Big]$
$\hat{\boldsymbol{f}}^{st,g}(\boldsymbol{\mu})_{(j)} = \sum_{k=1}^{N_t} \boldsymbol{D}_k^j \Delta t^{(k)} \boldsymbol{\Phi}_s^T \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k)}(\boldsymbol{\mu})$	$\hat{\boldsymbol{f}}^{st,pg}(\boldsymbol{\mu})_{(j)} = \sum_{k=1}^{N_t} \Big[\boldsymbol{D}_k^j \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(k)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\Delta t^{(k)} \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k)}(\boldsymbol{\mu})\Big] - \sum_{k=1}^{N_t-1} \Big[\boldsymbol{D}_k^j \boldsymbol{\Phi}_s^T \Delta t^{(k+1)} \boldsymbol{B}(\boldsymbol{\mu}) \boldsymbol{f}^{(k+1)}(\boldsymbol{\mu})\Big]$
$\hat{\boldsymbol{u_0}}^{st,g}(\boldsymbol{\mu})_{(j)} = \boldsymbol{D}_1^j \boldsymbol{\Phi}_s^T \boldsymbol{u}_0(\boldsymbol{\mu})$	$\hat{\boldsymbol{u_0}}^{st,pg}(\boldsymbol{\mu})_{(j)} = \boldsymbol{D}_1^j \boldsymbol{\Phi}_s^T \Big(\boldsymbol{I}_{N_s} - \Delta t^{(1)} \boldsymbol{A}(\boldsymbol{\mu})^T\Big)\boldsymbol{u}_0(\boldsymbol{\mu})$

5.5 Computational complexity of forming space–time ROM operators

To compute computational complexity of forming the reduced space–time system matrices, input vectors, and initial state vectors for Galerkin and Petrov–Galerkin projections, we assume that $\boldsymbol{A}(\boldsymbol{\mu}) \in \mathbb{R}^{N_s \times N_s}$ is a band matrix with the bandwidth, b and $\boldsymbol{B}(\boldsymbol{\mu})$ is a identity matrix, $\boldsymbol{I}_{N_s}$. The band structure of $\boldsymbol{A}(\boldsymbol{\mu})$ is often seen in mathematical models because of local approximations to derivative terms. Then, the bandwidth of $\boldsymbol{A}^{st}(\boldsymbol{\mu}) \in \mathbb{R}^{N_s N_t \times N_s N_t}$ formed with backward Euler scheme is N_s. We also assume that the spatial basis vectors $\boldsymbol{\phi}_i^{\mathrm{s}} \in \mathbb{R}^{N_s}, i \in \mathbb{N}(n_s)$ and temporal basis vectors $\boldsymbol{\phi}_{ij}^{\mathrm{t}} \in \mathbb{R}^{N_t}, i \in \mathbb{N}(n_s)$ and $j \in \mathbb{N}(n_t)$ are given.

Let us start to compute the computational cost without use of block structures. Constructing space–time basis costs $\mathcal{O}(N_s N_t n_s n_t)$. For Galerkin projections, computing the reduced space–time system matrix, input vectors, and initial state vector costs $\mathcal{O}(2N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$, $\mathcal{O}(N_s N_t n_s n_t)$, and $\mathcal{O}(N_s n_s n_t)$, respectively. Thus, keeping the dominant terms and taking off coefficient 2 lead to $\mathcal{O}(N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$. With the assumption of $n_s n_t \ll N_s$, we have $\mathcal{O}(N_s^2 N_t n_s n_t)$. For Petrov–Galerkin projections, we first compute $\boldsymbol{\Phi}_{\mathrm{st}}^T \boldsymbol{A}^{st}(\boldsymbol{\mu})$, resulting in $\mathcal{O}(2N_s^2 N_t n_s n_t)$. Then, computing the reduced space–time system matrix, input vectors, and initial state vector costs $\mathcal{O}(N_s N_t (n_s n_t)^2)$, $\mathcal{O}(N_s N_t n_s n_t)$, and $\mathcal{O}(N_s n_s n_t)$, respectively. Thus, keeping the dominant terms and taking off coefficient 2 lead to $\mathcal{O}(N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$. With the assumption of $n_s n_t \ll N_s$, we have $\mathcal{O}(N_s^2 N_t n_s n_t)$.

Now, let us compute the computational complexity with use of block structures. For Galerkin projection, we first compute $\boldsymbol{\Phi}_s^T \boldsymbol{A}(\boldsymbol{\mu}) \boldsymbol{\Phi}_s$ which will be reused, resulting in $\mathcal{O}(2bN_s n_s) + \mathcal{O}(N_s n_s^2)$. Then, we compute n_t^2 blocks for the reduced space–time system matrix. It takes $\mathcal{O}(N_t(2n_s^2 + 2n_s))$ to compute each block. Thus, it costs $\mathcal{O}(N_t n_t^2 (2n_s^2 + 2n_s))$ to compute the reduced space–time system matrix. For the reduced space–time input vector, n_t blocks are needed and each block costs $\mathcal{O}(N_t(N_s n_s + n_s))$, resulting in $\mathcal{O}(N_t n_t (N_s n_s + n_s))$. The computing the reduced space–time initial vector costs $\mathcal{O}(N_s n_s + n_s)$. Thus, keeping the dominant terms and taking off coefficient 2 lead to $\mathcal{O}(bN_s n_s) + \mathcal{O}(N_s n_s^2) + \mathcal{O}(N_t (n_s n_t)^2) + \mathcal{O}(N_s N_t n_s n_t)$. With the assumptions of $b \ll N_t n_t$, $n_s \ll N_t n_t$, and $n_s n_t \ll N_s$, we have $\mathcal{O}(N_s N_t n_s n_t)$. For Petrov–Galerkin projection, we compute $(\boldsymbol{I}_{N_s} - \Delta t \boldsymbol{A}(\boldsymbol{\mu}))\boldsymbol{\Phi}_s$, resulting in $\mathcal{O}(2bN_s n_s)$. Then, we compute $\boldsymbol{\Phi}_s^T (\boldsymbol{I}_{N_s} - \Delta t \boldsymbol{A}(\boldsymbol{\mu})^T)(\boldsymbol{I}_{N_s} - \Delta t \boldsymbol{A}(\boldsymbol{\mu}))\boldsymbol{\Phi}_s$ and $\boldsymbol{\Phi}_s^T (\boldsymbol{I}_{N_s} - \Delta t \boldsymbol{A}(\boldsymbol{\mu}))\boldsymbol{\Phi}_s$ for re-use. Each of them costs $\mathcal{O}(N_s n_s^2)$. Now, we compute n_t^2 blocks for the reduced space–time system matrix. It takes $\mathcal{O}(N_t(6n_s^2 + n_s))$ to compute each

block. Thus, it costs $\mathcal{O}(N_t n_t^2(6n_s^2 + n_s))$ to compute the reduced space–time system matrix. For the reduced space–time input vector, n_t blocks are needed and each block costs $\mathcal{O}(N_t(N_s n_s + n_s))$, resulting in $\mathcal{O}(N_t n_t(N_s n_s + n_s))$. The computing the reduced space–time initial vector costs $\mathcal{O}(N_s n_s + n_s)$. Thus, keeping the dominant terms and taking off coefficients 2 and 6 lead to $\mathcal{O}(bN_s n_s) + \mathcal{O}(N_s n_s^2) + \mathcal{O}(N_t(n_s n_t)^2) + \mathcal{O}(N_s N_t n_s n_t)$. With the assumptions of $b \ll N_t n_t$, $n_s \ll N_t n_t$, and $n_s n_t \ll N_s$, we have $\mathcal{O}(N_s N_t n_s n_t)$.

In summary, the computational complexities of forming space–time ROM operators in training phase for Galerkin and Petrov–Galerkin projections are presented in Table 3. We observe that a lot of computational costs are reduced by making use of block structures for forming space–time reduced order models.

Table 3: Comparison of Galerkin and Petrov–Galerkin computational complexities

	Galerkin	**Petrov–Galerkin**
Not using block structures	$\mathcal{O}(N_s^2 N_t n_s n_t)$	$\mathcal{O}(N_s^2 N_t n_s n_t)$
Using block structures	$\mathcal{O}(N_s N_t n_s n_t)$	$\mathcal{O}(N_s N_t n_s n_t)$

6 Error analysis

We present error analysis of the space–time ROM method. The error analysis is based on [1]. *A posteriori* error bound is derived in this section. Here, we drop the parameter dependence for notational simplicity.

Theorem 6.1. *We define the error at kth time step as* $\boldsymbol{e}^{(k)} \equiv \boldsymbol{u}^{(k)} - \tilde{\boldsymbol{u}}^{(k)} \in \mathbb{R}^{N_s}$ *where* $\boldsymbol{u}^{(k)} \in \mathbb{R}^{N_s}$ *denotes FOM solution,* $\tilde{\boldsymbol{u}}^{(k)} \in \mathbb{R}^{N_s}$ *denotes approximate solution, and* $k \in \mathbb{N}(N_t)$*. Let* $\boldsymbol{A}^{st} \in \mathbb{R}^{N_s N_t \times N_s N_t}$ *be the space–time system matrix,* $\boldsymbol{r}^{(k)} \in \mathbb{R}^{N_s}$ *be the residual computed using FOM solution at kth time step, and* $\tilde{\boldsymbol{r}}^{(k)} \in \mathbb{R}^{N_s}$ *be the residual computed using approximate solution at kth time step. For example,* $\boldsymbol{r}^{(k)}$ *and* $\tilde{\boldsymbol{r}}^{(k)}$ *after applying the backward Euler scheme with the uniform time step become*

$$\boldsymbol{r}^{(k)}(\boldsymbol{u}^{(k)}, \boldsymbol{u}^{(k-1)}) = \Delta t \boldsymbol{f}^{(k)} + \boldsymbol{u}^{(k-1)} - (\boldsymbol{I} - \Delta t \boldsymbol{A})\boldsymbol{u}^{(k)} = \boldsymbol{0} \tag{6.1}$$

$$\tilde{\boldsymbol{r}}^{(k)}(\tilde{\boldsymbol{u}}^{(k)}, \tilde{\boldsymbol{u}}^{(k-1)}) = \Delta t \boldsymbol{f}^{(k)} + \tilde{\boldsymbol{u}}^{(k-1)} - (\boldsymbol{I} - \Delta t \boldsymbol{A})\tilde{\boldsymbol{u}}^{(k)} \tag{6.2}$$

with $\tilde{\boldsymbol{u}}^{(0)} = \boldsymbol{u}^0$*. Then, the error bound is given by*

$$\max_{k \in \mathbb{N}(N_t)} \|\boldsymbol{e}^{(k)}\|_2 \leq \eta \max_{k \in \mathbb{N}(N_t)} \|\tilde{\boldsymbol{r}}^{(k)}\|_2 \tag{6.3}$$

where $\eta \equiv \sqrt{N_t}\|(\boldsymbol{A}^{st})^{-1}\|_2$ *denotes the stability constant.*

Proof. Let us define the space–time residual as

$$\boldsymbol{r}^{\text{st}} : \boldsymbol{v} \mapsto \boldsymbol{f}^{\text{st}} + \boldsymbol{u}_0^{\text{st}} - \boldsymbol{A}^{\text{st}}\boldsymbol{v} \tag{6.4}$$

with $\boldsymbol{r}^{\text{st}} : \mathbb{R}^{N_s N_t} \to \mathbb{R}^{N_s N_t}$. Then, we have

$$\boldsymbol{r}^{\text{st}}(\boldsymbol{u}^{\text{st}}) = \boldsymbol{f}^{\text{st}} + \boldsymbol{u}_0^{\text{st}} - \boldsymbol{A}^{\text{st}}\boldsymbol{u}^{\text{st}} = \boldsymbol{0} \tag{6.5}$$

$$\boldsymbol{r}^{\text{st}}(\tilde{\boldsymbol{u}}^{\text{st}}) = \boldsymbol{f}^{\text{st}} + \boldsymbol{u}_0^{\text{st}} - \boldsymbol{A}^{\text{st}}\tilde{\boldsymbol{u}}^{\text{st}} \tag{6.6}$$

where $\boldsymbol{u}^{\text{st}} \in \mathbb{R}^{N_s N_t}$ is the space–time FOM solution and $\tilde{\boldsymbol{u}}^{\text{st}} \in \mathbb{R}^{N_s N_t}$ is the approximate space–time solution. Subtracting Equation (6.6) from Equation (6.5) gives

$$\boldsymbol{r}^{\text{st}}(\tilde{\boldsymbol{u}}^{\text{st}}) = \boldsymbol{A}^{\text{st}}\boldsymbol{e}^{st} \tag{6.7}$$

where $\boldsymbol{e}^{st} \equiv \boldsymbol{u}^{\text{st}} - \tilde{\boldsymbol{u}}^{\text{st}} \in \mathbb{R}^{N_s N_t}$. Inverting $\boldsymbol{A}^{\text{st}}$ yields

$$\boldsymbol{e}^{st} = (\boldsymbol{A}^{\text{st}})^{-1}\boldsymbol{r}^{\text{st}}(\tilde{\boldsymbol{u}}^{\text{st}}). \tag{6.8}$$

Taking ℓ_2 norm and Hölders' inequality gives

$$\|\boldsymbol{e}^{st}\|_2 \leq \|(\boldsymbol{A}^{\text{st}})^{-1}\|_2 \|\boldsymbol{r}^{\text{st}}(\tilde{\boldsymbol{u}}^{\text{st}})\|_2. \tag{6.9}$$

We can re-write this in the following form

$$\sqrt{\sum_{k=1}^{N_t} \|\boldsymbol{e}^{(k)}\|_2^2} \leq \|(\boldsymbol{A}^{\text{st}})^{-1}\|_2 \sqrt{\sum_{k=1}^{N_t} \|\tilde{\boldsymbol{r}}^{(k)}\|_2^2}. \tag{6.10}$$

Using the relations

$$\max_{k \in \mathbb{N}(N_t)} \|\boldsymbol{e}^{(k)}\|_2^2 \leq \sum_{k=1}^{N_t} \|\boldsymbol{e}^{(k)}\|_2^2 \tag{6.11}$$

and

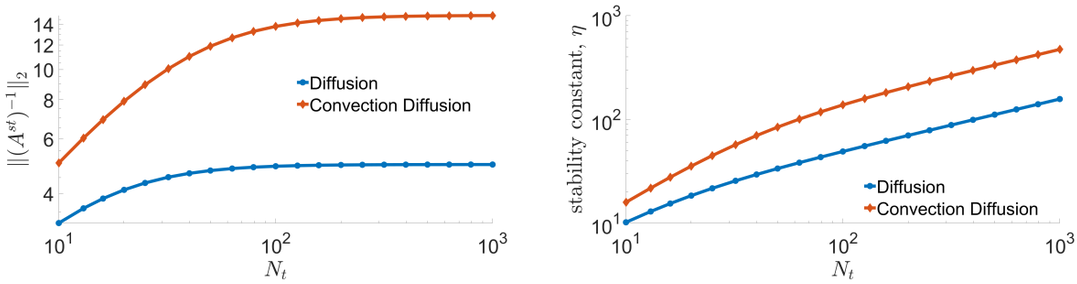
$$\sum_{k=1}^{N_t} \|\tilde{\boldsymbol{r}}^{(k)}\|_2^2 \leq N_t \max_{k \in \mathbb{N}(N_t)} \|\tilde{\boldsymbol{r}}^{(k)}\|_2^2, \tag{6.12}$$

we have

$$\max_{k \in \mathbb{N}(N_t)} \|\boldsymbol{e}^{(k)}\|_2 \leq \sqrt{N_t}\|(\boldsymbol{A}^{\text{st}})^{-1}\|_2 \max_{k \in \mathbb{N}(N_t)} \|\tilde{\boldsymbol{r}}^{(k)}\|_2, \tag{6.13}$$

which is equivalent to the error bound in (6.3). □

A numerical demonstration with space–time system matrices, $\boldsymbol{A}^{\text{st}}$ that have the same structure as the ones used in Section 7.1 and Section 7.2.1 shows the magnitude of $\|(\boldsymbol{A}^{\text{st}})^{-1}\|_2$ increases linearly for small N_t, while it becomes eventually flattened for large N_t as shown in Fig. 2(a) for the backward Euler time integrator with uniform time step size. Combined with $\sqrt{N_t}$, the stability constant η growth rate is shown in Fig. 2(b). These error bound shows much improvement against the ones for the spatial Galerkin and Petrov–Galerkin ROMs, which grows exponentially in time [1].



(a) $\|(\boldsymbol{A}^{\mathrm{st}})^{-1}\|_2$ in inequality (6.3)

(b) Stability constant, η in inequality (6.3)

Figure 2: Growth rate of stability constant in Theorem 6.1. Backward Euler time stepping scheme with uniform time step size, $\Delta t = 10^{-2}$ is used.

7 Numerical results

In this section, we apply both the space–time Galerkin and Petrov–Galerkin ROMs to two model problems: (i) a 2D linear diffusion equation in Section 7.1 and (ii) a 2D linear convection-diffusion equation in Section 7.2. We demonstrate their accuracy and speed-up. The space–time ROMs are trained with solution snapshots associated with parameters in a chosen domain and used to predict the solution of a parameter that is not included in the trained parameter domain. We refer to this as the predictive case. The accuracy of space–time ROM solution $\tilde{\boldsymbol{u}}^{\mathrm{st}}(\boldsymbol{\mu})$ is assessed from its relative error by:

$$\text{relative error} = \frac{\left\|\tilde{\boldsymbol{u}}^{\mathrm{st}}(\boldsymbol{\mu}) - \boldsymbol{u}^{\mathrm{st}}(\boldsymbol{\mu})\right\|_2}{\left\|\boldsymbol{u}^{\mathrm{st}}(\boldsymbol{\mu})\right\|_2} \tag{7.1}$$

and the ℓ_2 norm of space–time residual:

$$\left\|\boldsymbol{r}^{\mathrm{st}}(\tilde{\boldsymbol{u}}^{\mathrm{st}}(\boldsymbol{\mu}))\right\|_2. \tag{7.2}$$

The computational cost is measured in terms of CPU wall-clock time. The online speed-up is evaluated by dividing the wall-clock time of the FOM by the online phase of the ROM. For the multi-query problems, total speed-up is evaluated by dividing the time of all FOMs by the time of all ROMs including training time. All calculations are performed on an Intel(R) Core(TM) i9-10900T CPU @ 1.90GHz and DDR4 Memory @ 2933MHz.

7.1 2D linear diffusion equation

We consider a parameterized 2D linear diffusion equation with a source term

$$\begin{aligned}\frac{\partial u}{\partial t} = \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right] - \left[\frac{1}{\sqrt{(x-\mu_1)^2 + (y-\mu_2)^2}}\right] u \\ + \frac{\sin 2\pi t}{\sqrt{(x-\mu_1)^2 + (y-\mu_2)^2}}\end{aligned} \tag{7.3}$$

where $(x, y) \in [0, 1] \times [0, 1]$, $t \in [0, 2]$ and $(\mu_1, \mu_2) \in [-1.7, -0.2] \times [-1.7, -0.2]$. The boundary condition is

$$\begin{aligned} u(x = 0, y, t) &= 0 \\ u(x = 1, y, t) &= 0 \\ u(x, y = 0, t) &= 0 \\ u(x, y = 1, t) &= 0 \end{aligned} \tag{7.4}$$

and the initial condition is

$$u(x, y, t = 0) = 0. \tag{7.5}$$

The backward Euler with uniform time step size $\frac{2}{N_t}$ is employed, where we set $N_t = 50$. For spatial differentiation, the second order central difference scheme is implemented for the diffusion terms. Discretizing the space domain into $N_x = 70$ and $N_y = 70$ uniform meshes in x and y directions, respectively, gives $N_s = (N_x - 1) \times (N_y - 1) = 4,761$ grid points, excluding boundary grid points. As a result, there are $238,050$ free degrees of freedom in space–time.

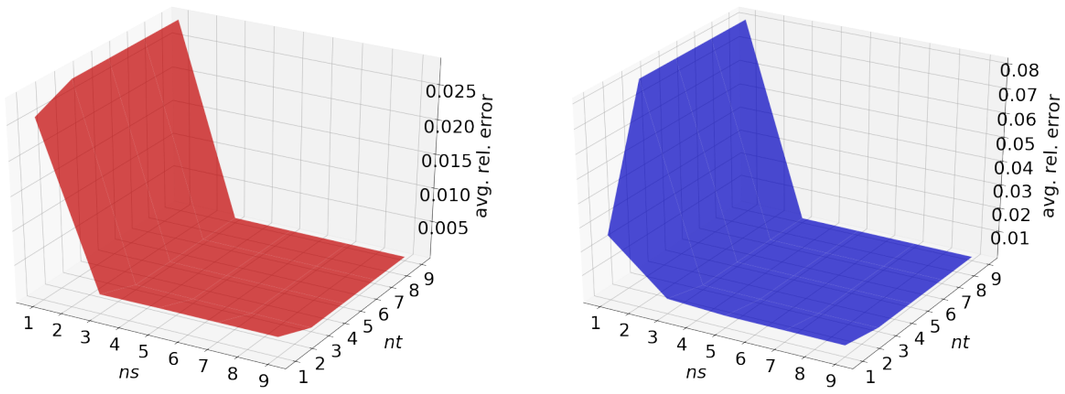
For training phase, we collect solution snapshots associated with the following parameters:

$$(\mu_1, \mu_2) \in \{(-0.9, -0.9), (-0.9, -0.5), (-0.5, -0.9), (-0.5, -0.5)\}$$

at which the FOM is solved.

The Galerkin and Petrov–Galerkin space–time ROMs solve the Equation (7.3) with the target parameter $(\mu_1, \mu_2) = (-0.7, -0.7)$). Fig. 3, 4, and 5 show the relative errors, the space–time residuals, and the online speed-ups as a function of the reduced dimension n_s and n_t. We observe that both Galerkin and Petrov–Galerkin ROMs with $n_s = 5$ and $n_t = 3$ achieve a good accuracy (i.e., relative errors of 0.012% and 0.026%, respectively) and speed-up (i.e., 350.31 and 376.04, respectively). We also observe that the relative errors of Galerkin projection is smaller but the space–time residual is larger than Petrov–Galerkin projection. This is because Petrov–Galerkin space–time ROM solution minimizes the space–time residual.

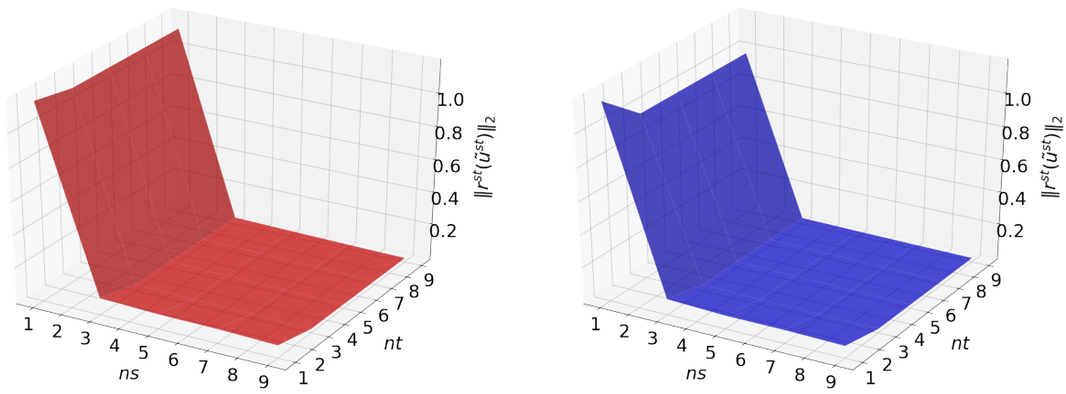
The final time snapshots of FOM, Galerkin space–time ROM, and Petrov–Galerkin space–time ROM are seen in Fig. 6. Both ROMs have a basis size of $n_s = 5$ and $n_t = 3$, resulting in a reduction factor of $(N_s N_t)/(n_s n_t) = 15,870$. For the Galerkin method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ takes an average time of 6.1816×10^{-1} and 1.7646×10^{-3} seconds, respectively, resulting in speed-up of 350.31. For the Petrov–Galerkin method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ takes an average time of 6.0809×10^{-1} and 1.6171×10^{-3} seconds, respectively, resulting in speed-up of 376.04. For accuracy, the Galerkin method results in 1.210×10^{-2} % relative error and 1.249×10^{-2} space–time residual norm while the Petrov–Galerkin results in 2.626×10^{-2} % relative error and 1.029×10^{-2} space–time residual norm.



(a) Relative errors vs reduced dimensions for Galerkin projection

(b) Relative errors vs reduced dimensions for Petrov–Galerkin projection

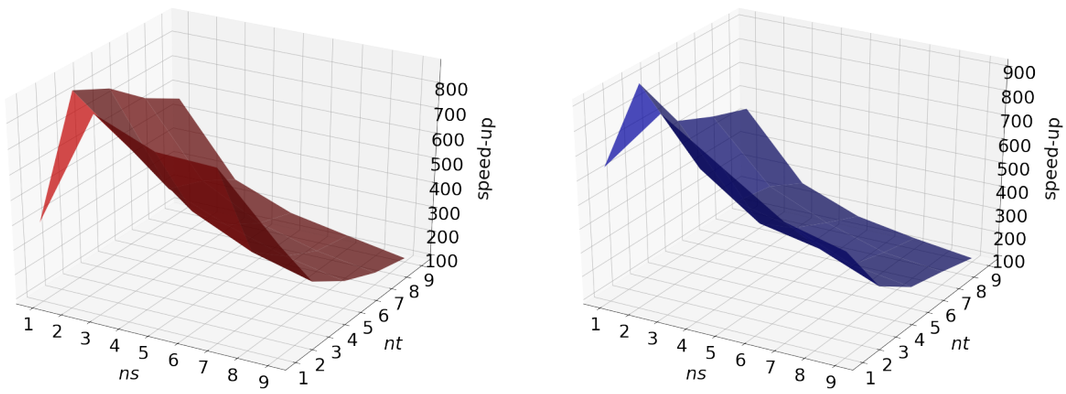
Figure 3: 2D linear diffusion equation. Relative errors vs reduced dimensions.



(a) Space-time residuals vs reduced dimensions for Galerkin projection

(b) Space-time residuals vs reduced dimensions for Petrov–Galerkin projection

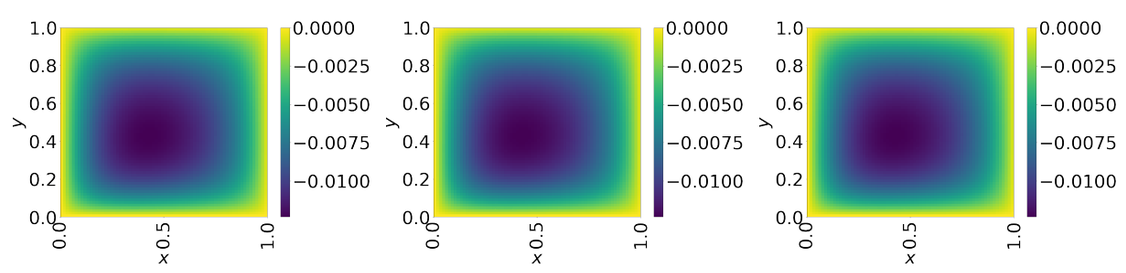
Figure 4: 2D linear diffusion equation. Space-time residuals vs reduced dimensions.



(a) Speedups vs reduced dimensions for Galerkin projection

(b) Speedups vs reduced dimensions for Petrov–Galerkin projection

Figure 5: 2D linear diffusion equation. Speedups vs reduced dimensions.



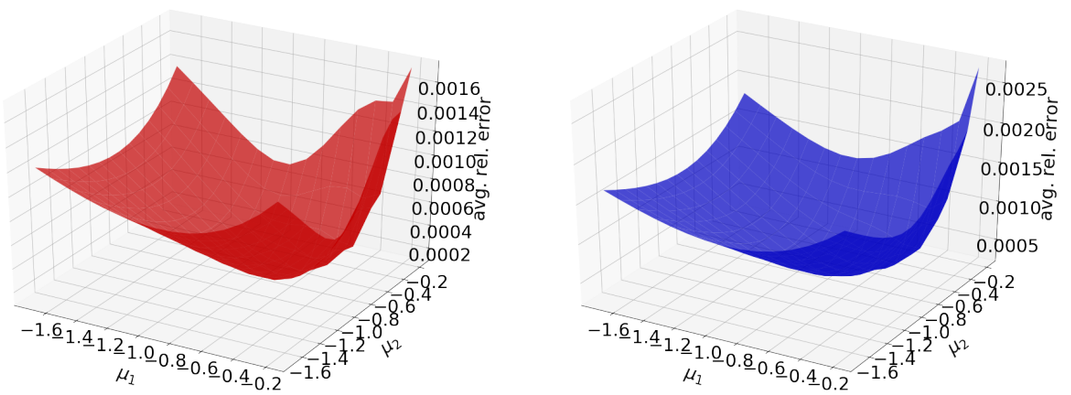
(a) FOM

(b) Galerkin ROM

(c) Petrov–Galerkin ROM

Figure 6: Solution snapshots of FOM, Galerkin ROM, and Petrov–Galerkin ROM at $t = 2$.

We investigate the numerical tests to see the generalization capability of both Galerkin and Petrov–Galerkin ROMs. The train parameter set, $(\mu_1, \mu_2) \in \{(-0.9, -0.9), (-0.9, -0.5), (-0.5, -0.9), (-0.5, -0.5)\}$ is used to train a space–time ROMs with a basis of $n_s = 5$ and $n_t = 3$. Then trained ROMs solve predictive cases with the test parameter set, $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = -1.7 + 1.5/14i, i = 0, 1, \cdots, 14\} \times \{\mu_2 | \mu_2 = -1.7 + 1.5/14j, j = 0, 1, \cdots, 14\}$. Fig. 7 shows the relative errors over the test parameter set. The Galerkin and Petrov–Galerkin ROMs are the most accurate within the range of the train parameter points, i.e., $[-0.9, -0.5] \times [-0.9, -0.5]$. As the parameter points go beyond the train parameter domain, the accuracy of the Galerkin and Petrov–Galerkin ROMs start to deteriorate gradually. This implies that the Galerkin and Petrov–Galerkin ROMs have a trust region. Its trust region should be determined by the application space. For Galerkin ROM, online speed-up is about 389 in average and total time for ROM and FOM are 107.14 and 132.66 seconds, respectively, resulting in total speed-up of 1.24. For Petrov–Galerkin ROM, online speed-up is about 386 in average and total time for ROM and FOM are 117.96 and 132.42 seconds, respectively, resulting in total speed-up of 1.12. Since the training time doesn't depend on the number of test cases, we expect more speed-up for a larger number of test cases.



(a) Glaerkin (b) Petrov–Galerkin

Figure 7: The comparison of the Galerkin and Petrov–Galerkin ROMs for predictive cases

7.2 2D linear convection diffusion equation

7.2.1 Without source term

We consider a parameterized 2D linear convection diffusion equation

$$\frac{\partial u}{\partial t} = -\mu_1 \left[\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right] + \mu_2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right] \tag{7.6}$$

where $(x, y) \in [0, 1] \times [0, 1]$, $t \in [0, 1]$ and $(\mu_1, \mu_2) \in [0.01, 0.07] \times [0.31, 0.37]$. The boundary condition is given by

$$\begin{aligned} u(x = 0, y, t) &= 0 \\ u(x = 1, y, t) &= 0 \\ u(x, y = 0, t) &= 0 \\ u(x, y = 1, t) &= 0. \end{aligned} \tag{7.7}$$

The initial condition is given by

$$u(x, y, t = 0) = \begin{cases} 100 \sin(2\pi x)^3 \cdot \sin(2\pi y)^3 & \text{if } (x, y) \in [0, 0.5] \times [0, 0.5] \\ 0 & \text{otherwise} \end{cases} \tag{7.8}$$

and shown in Fig. 8.

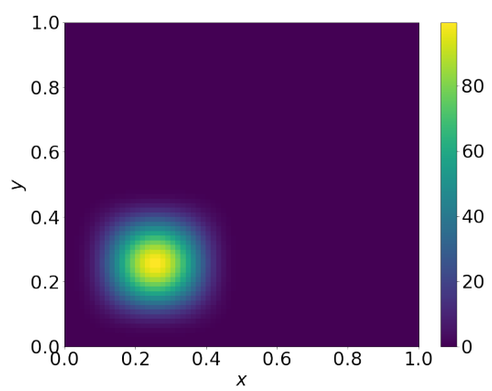


Figure 8: Plot of Equation (7.8)

The backward Euler with uniform time step size $\frac{1}{N_t}$ is employed where we set $N_t = 50$. For spatial differentiation, a second order central difference scheme for the diffusion terms and a first order backward difference scheme for the

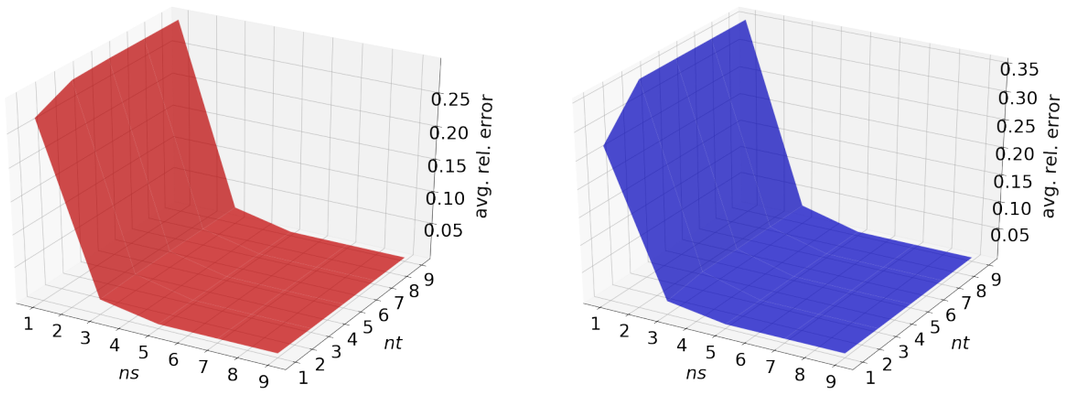
convection terms are implemented. Discretizing the space domain into $N_x = 70$ and $N_y = 70$ uniform meshes in x and y directions, respectively, gives $N_s = (N_x - 1) \times (N_y - 1) = 4,761$ grid points, excluding boundary grid points. As a result, there are $238,050$ free degrees of freedom in space–time.

For training phase, we collect solution snapshots associated with the following parameters:

$$(\mu_1, \mu_2) \in \{(0.03, 0.33), (0.03, 0.35), (0.05, 0.33), (0.05, 0.35)\},$$

at which the FOM is solved.

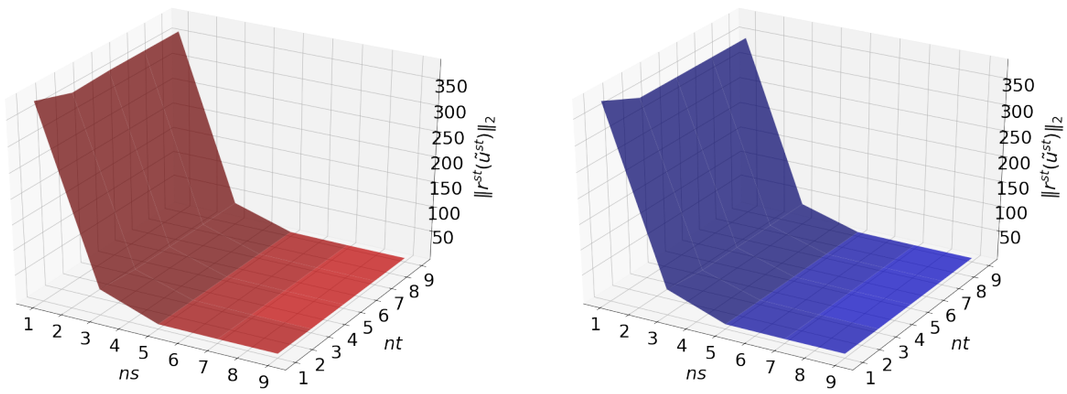
The Galerkin and Petrov–Galerkin space–time ROMs solve the Equation (7.6) with the target parameter $(\mu_1, \mu_2) = (0.04, 0.34)$. Fig. 9, 10, and 11 show the relative errors, the space–time residuals, and the online speed-ups as a function of the reduced dimension n_s and n_t. We observe that both Galerkin and Petrov–Galerkin ROMs with $n_s = 5$ and $n_t = 3$ achieve a good accuracy (i.e., relative errors of 0.049% and 0.059%, respectively) and speed-up (i.e., 451.17 and 370.74, respectively). We also observe that the relative errors of Galerkin projection is smaller but the space–time residual is larger than Petrov–Galerkin projection. This is because Petrov–Galerkin space–time ROM solution minimizes the space–time residual.



(a) Relative errors vs reduced dimensions for Galerkin projection

(b) Relative errors vs reduced dimensions for Petrov–Galerkin projection

Figure 9: 2D linear convection diffusion equation. Relative errors vs reduced dimensions.



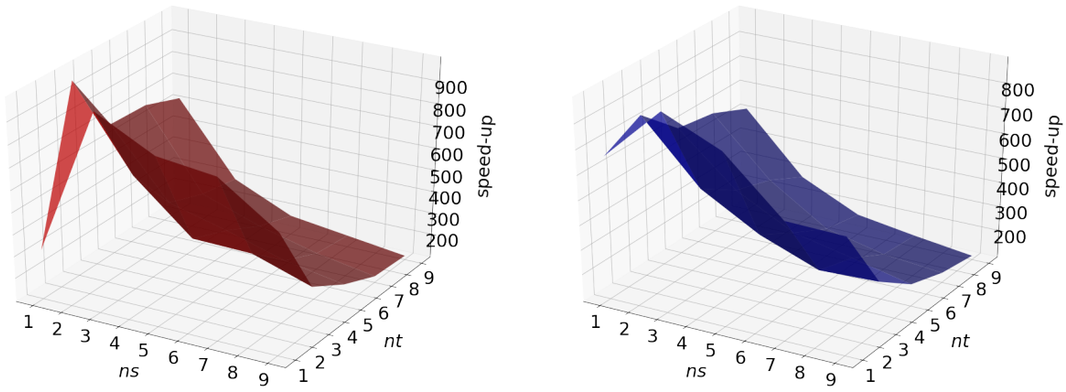
(a) Space-time residuals vs reduced dimensions for Galerkin projection

(b) Space-time residuals vs reduced dimensions for Petrov–Galerkin projection

Figure 10: 2D linear convection diffusion equation. Space-time residuals vs reduced dimensions.

The final time snapshots of FOM, Galerkin space–time ROM, and Petrov–Galerkin space–time ROM are seen in Fig. 12. Both ROMs have a basis size of $n_s = 5$ and $n_t = 3$, resulting in a reduction factor of $(N_s N_t)/(n_s n_t) = 15,870$. For the Galerkin method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ takes an average time of 6.1562×10^{-1} and 1.3645×10^{-3} seconds, respectively, resulting in speed-up of 451.17. For the Petrov–Galerkin method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ takes an average time of 5.7617×10^{-1} and 1.5541×10^{-3} seconds, respectively, resulting in speed-up of 370.74. For accuracy, the Galerkin method results in 4.898×10^{-2} % relative error and 1.503 space–time residual norm while the Petrov–Galerkin results in 5.878×10^{-2} % relative error and 1.459 space–time residual norm.

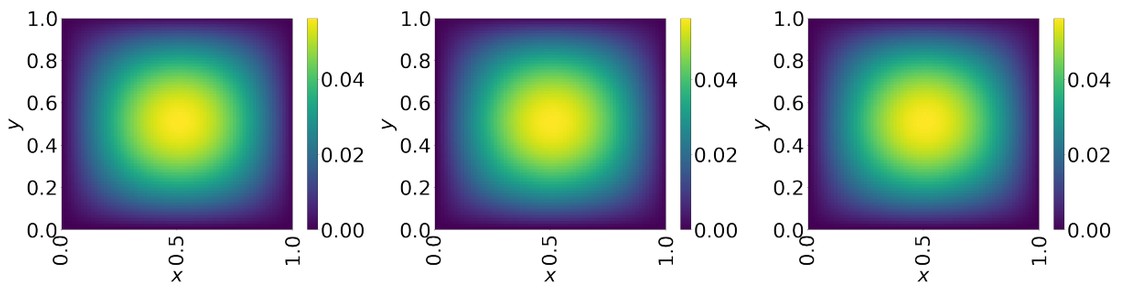
We investigate the numerical tests to see the generalization capability of both Galerkin and Petrov–Galerkin ROMs. The train parameter set, $(\mu_1, \mu_2) \in \{(0.03, 0.33), (0.03, 0.35), (0.05, 0.33), (0.05, 0.35)\}$ is used to train a space–time ROMs with a basis of $n_s = 5$ and $n_t = 3$. Then trained ROMs solve predictive cases with the test parameter set, $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = 0.01 + 0.06/11i, i = 0, 1, \cdots, 11\} \times \{\mu_2 | \mu_2 = 0.31 + 0.06/11j, j = 0, 1, \cdots, 11\}$. Fig. 13 shows the relative errors over the test parameter set. The Galerkin and Petrov–Galerkin ROMs are the most accurate within the range of the train parameter points, i.e., $[0.03, 0.33] \times [0.05, 0.35]$. As the parameter points go beyond the train parameter domain, the accuracy of the Galerkin and Petrov–Galerkin ROMs start to deteriorate gradually. This implies that the Galerkin and Petrov–Galerkin ROMs have a trust region. Its trust region should be determined by an application. For Galerkin ROM, online speed-up is about 387 in average and total time for ROM and FOM are 65.03 and 83.89 seconds, respectively, resulting in total speed-up of 1.29. For Petrov–Galerkin ROM, online speed-up is about 385 in average and total time for ROM and FOM are 70.55 and 83.34 seconds, respectively,



(a) Speedups vs reduced dimensions for Galerkin projection

(b) Speedups vs reduced dimensions for Petrov–Galerkin projection

Figure 11: 2D linear convection diffusion equation. Speedups vs reduced dimensions.



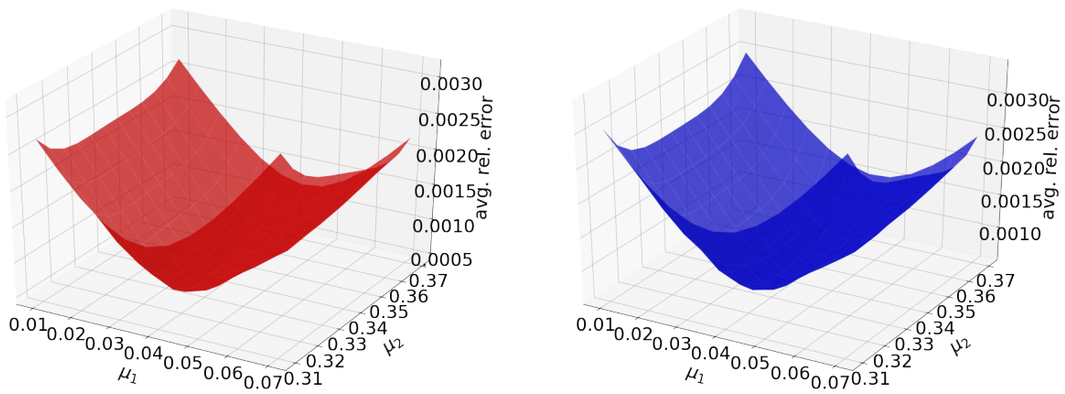
(a) FOM

(b) Galerkin ROM

(c) Petrov–Galerkin ROM

Figure 12: Solution snapshots of FOM, Galerkin ROM, and Petrov–Galerkin ROM at $t = 1$.

resulting in total speed-up of 1.18. Since the training time doesn't depend on the number of test cases, we expect more speed-up for the larger number of test cases.



(a) Glaerkin

(b) Petrov–Galerkin

Figure 13: The comparison of the Galerkin and Petrov–Galerkin ROMs for predictive cases

7.2.2 With source term

We consider a parameterized 2D linear convection diffusion equation

$$\frac{\partial u}{\partial t} = -\mu_1 \left[0.1\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right] + \mu_2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right] + f(x, y, t) \tag{7.9}$$

with the source term $f(x, y, t)$ which is given by

$$f(x, y, t) = 10^5 \exp(-((\frac{x - 0.5 + 0.2\sin(2\pi t)}{0.1})^2 + (\frac{y}{0.05})^2)) \tag{7.10}$$

where $(x, y) \in [0, 1] \times [0, 1]$, $t \in [0, 2]$ and $(\mu_1, \mu_2) \in [0.195, 0.205] \times [0.018, 0.022]$. The boundary condition is given by

$$\begin{aligned} u(x = 0, y, t) &= 0 \\ u(x = 1, y, t) &= 0 \\ u(x, y = 0, t) &= 0 \\ u(x, y = 1, t) &= 0 \end{aligned} \tag{7.11}$$

and the initial condition is given by

$$u(x, y, t = 0) = 0. \tag{7.12}$$

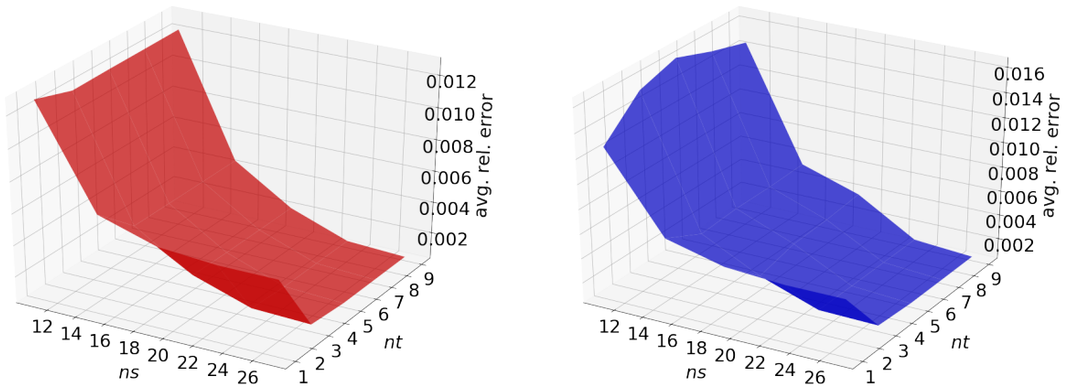
The backward Euler with uniform time step size $\frac{2}{N_t}$ is employed where we set $N_t = 50$. For spatial differentiation, a second order central difference scheme for the diffusion terms and a first order backward difference scheme for the convection terms are implemented. Discretizing the space domain into $N_x = 70$ and $N_y = 70$ uniform meshes in x and y directions, respectively, gives $N_s = (N_x - 1) \times (N_y - 1) = 4,761$ grid points, excluding boundary grid points. As a result, there are $238,050$ free degrees of freedom in space–time.

For training phase, we collect solution snapshots associated with the following parameters:

$$(\mu_1, \mu_2) \in \{(0.195, 0.018), (0.195, 0.022), (0.205, 0.018), (0.205, 0.022)\},$$

at which the FOM is solved.

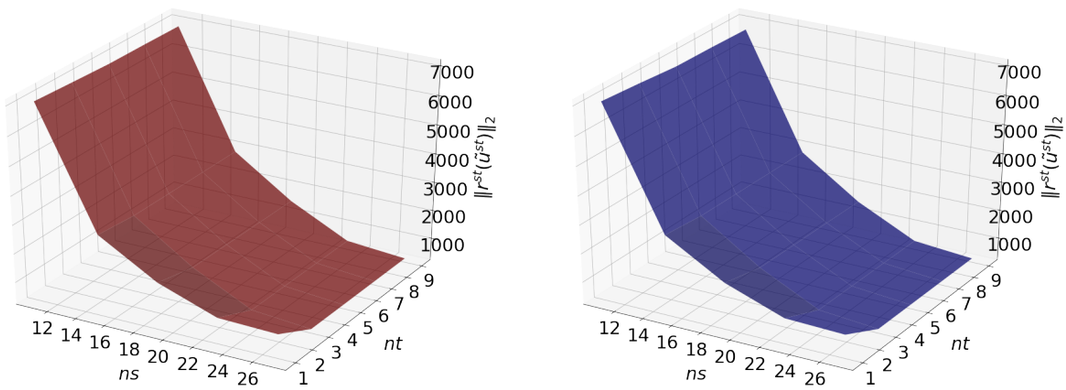
The Galerkin and Petrov–Galerkin space–time ROMs solve the Equation (7.9) with the target parameter $(\mu_1, \mu_2) = (0.2, 0.02))$. Fig. 14, 15, and 16 show the relative errors, the space–time residuals, and the online speed-ups as a function of the reduced dimension n_s and n_t. We observe that both Galerkin and Petrov–Galerkin ROMs with $n_s = 19$ and $n_t = 3$ achieve a good accuracy (i.e., relative errors of 0.217% and 0.265%, respectively) and speed-up (i.e., 153.87 and 139.88, respectively). We also observe that the relative errors of Galerkin projection is smaller but the space–time residual is larger than Petrov–Galerkin projection. This is because Petrov–Galerkin space–time ROM solution minimizes the space–time residual.



(a) Relative errors vs reduced dimensions for Galerkin projection

(b) Relative errors vs reduced dimensions for Petrov–Galerkin projection

Figure 14: 2D linear convection diffusion equation with source term. Relative errors vs reduced dimensions.



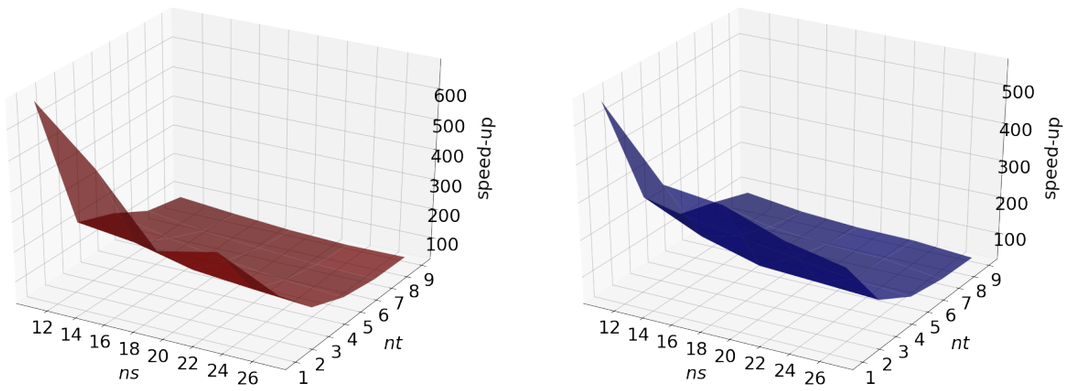
(a) Space-time residuals vs reduced dimensions for Galerkin projection

(b) Space-time residuals vs reduced dimensions for Petrov–Galerkin projection

Figure 15: 2D linear convection diffusion equation with source term. Space-time residuals vs reduced dimensions.

The final time snapshots of FOM, Galerkin space–time ROM, and Petrov–Galerkin space–time ROM are seen in Fig. 17. Both ROMs have a basis size of $n_s = 19$ and $n_t = 3$, resulting in a reduction factor of $(N_s N_t)/(n_s n_t) = 4,176$. For the Galerkin method, the FOM and space–time ROM simulation with $n_s = 19$ and $n_t = 3$ takes an average time of 6.1209×10^{-1} and 3.9780×10^{-3} seconds, respectively, resulting in speed-up of 153.87. For the Petrov–Galerkin method, the FOM and space–time ROM simulation with $n_s = 19$ and $n_t = 3$ takes an average time of 5.8780×10^{-1} and 4.2020×10^{-3} seconds, respectively, resulting in speed-up of 139.89. For accuracy, the Galerkin method results in 2.174×10^{-1} % relative error and 1.564×10^{3} space–time residual norm while the Petrov–Galerkin results in 2.652×10^{-1} % relative error and 1.550×10^{3} space–time residual norm.

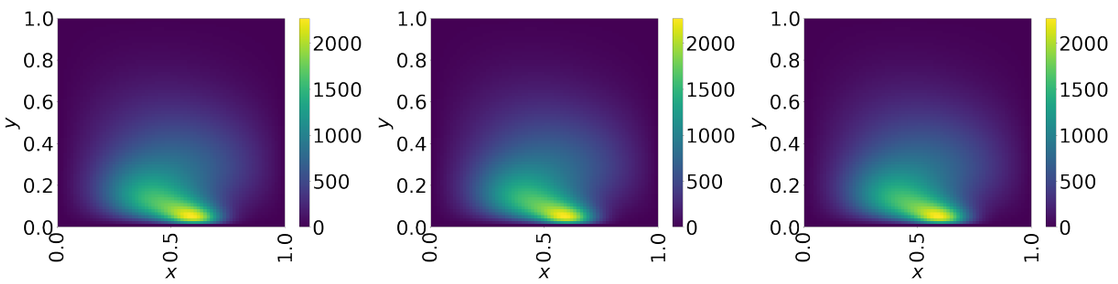
We investigate the numerical tests to see the generalization capability of both Galerkin and Petrov–Galerkin ROMs. The train parameter set, $(\mu_1, \mu_2) \in \{(0.195, 0.018), (0.195, 0.022), (0.205, 0.018), (0.205, 0.022)\}$ is used to train a space–time ROMs with a basis of $n_s = 19$ and $n_t = 3$. Then trained ROMs solve predictive cases with the test parameter set, $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = 0.160 + 0.08/11i, i = 0, 1, \cdots, 11\} \times \{\mu_2 | \mu_2 = 0.016 + 0.008/11j, j = 0, 1, \cdots, 11\}$. Fig. 18 shows the relative errors over the test parameter set. The Galerkin and Petrov–Galerkin ROMs are the most accurate within the range of the train parameter points, i.e., $[0.195, 0.205] \times [0.018, 0.022]$. As the parameter points



(a) Speedups vs reduced dimensions for Galerkin projection

(b) Speedups vs reduced dimensions for Petrov–Galerkin projection

Figure 16: 2D linear convection diffusion equation with source term. Speedups vs reduced dimensions.



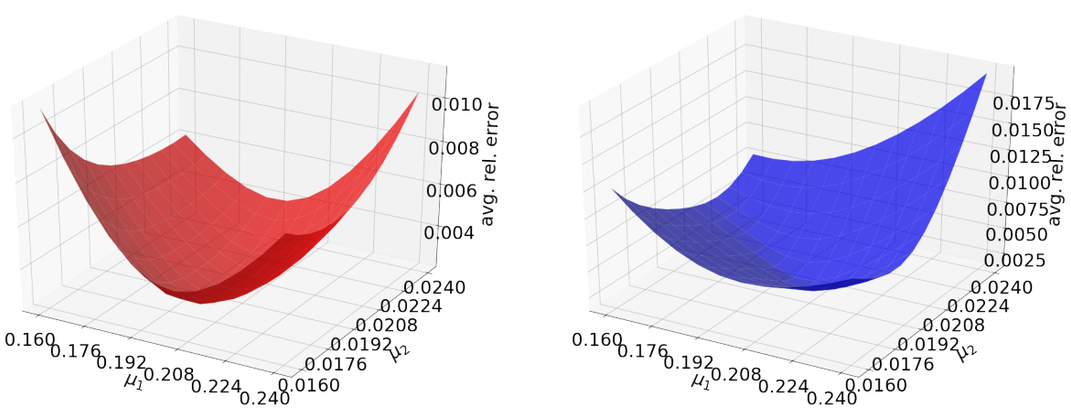
(a) FOM

(b) Galerkin ROM

(c) Petrov–Galerkin ROM

Figure 17: Solution snapshots of FOM, Galerkin ROM, and Petrov–Galerkin ROM at $t = 2$.

go beyond the train parameter domain, the accuracy of the Galerkin and Petrov–Galerkin ROMs start to deteriorate gradually. This implies that the Galerkin and Petrov–Galerkin ROMs have a trust region. Its trust region should be determined by an application. For Galerkin ROM, online speed-up is about 133 in average and total time for ROM and FOM are 68.49 and 82.35 seconds, respectively, resulting in total speed-up of 1.20. For Petrov–Galerkin ROM, online speed-up is about 138 in average and total time for ROM and FOM are 75.75 and 86.04 seconds, respectively, resulting in total speed-up of 1.14. Since the training time doesn't depend on the number of test cases, we expect more speed-up for the larger number of test cases.



(a) Glaerkin

(b) Petrov–Galerkin

Figure 18: The comparison of the Galerkin and Petrov–Galerkin ROMs for predictive cases

8 Conclusion

In this work, we have formulated Galerkin and Petrov–Galerkin space–time ROMs using block structures which enable us to implement the space–time ROM operators efficiently. We also presented an *a posteriori* error bound for both Galerkin and Petrov–Galerkin space–time ROMs. We demonstrated that both Galerkin and Petrov–Galerkin space–time ROMs solves 2D linear diffusion problems and 2D linear convection diffusion problems accurately and efficiently. Both space–time reduced order models were able to achieve $\mathcal{O}(10^{-3})$ to $\mathcal{O}(10^{-4})$ relative errors with $\mathcal{O}(10^2)$ speed-ups. We also presented our Python codes used for the numerical examples in Appendix A so that readers can easily reproduce our numerical results. Furthermore, each Python code is less than 120 lines, demonstrating the ease of implementing our space–time ROMs.

We used a linear subspace based ROM which is suitable for accelerating physical simulations whose solution space has a small Kolmogorov n-width. However, the linear subspace based ROM is not able to represent advection-

dominated or sharp gradient solutions with a small number of bases. To address this challenge, a nonlinear manifold based ROM can be used, and recently, a nonlinear manifold based ROM has been developed for spatial ROMs [72,73]. In future work, we aim to develop a nonlinear manifold based space–time ROM.

Acknowledgments

This work was performed at Lawrence Livermore National Laboratory and was supported by LDRD program (project 20-FS-007). Youngkyu was also supported for this work through generous funding from DTRA. Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344 and LLNL-JRNL-816093.

A Python codes in less than 120 lines of code for all numerical models described in Section 7

The Python code used for the numerical examples described in this paper are included in the following pages of the appendix are listed below. The total number of lines in each of the files are denoted in the parentheses. Note that we removed print statements of the results.

1. All input code for the Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term (111 lines)
2. All input code for the Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term (117 lines)
3. All input code for the Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation (114 lines)
4. All input code for the Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation (116 lines)
5. All input code for the Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term (115 lines)
6. All input code for the Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term (119 lines)

A.1 Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(-0.9,-0.5,2),np.linspace(-0.9,-0.5,2),indexing='ij')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1/N; k=2/Nt
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
x1=x1.flatten(); x2=x2.flatten(); t=np.linspace(0,2,Nt+1)

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D=sp.sparse.kron(A1D,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D,format='csc')

# snapshot, free DoFs, No IC included
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=A2D-sp.sparse.diags(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2),format='csc')
    f=(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
    u=np.zeros((Ns,Nt+1)) # IC is zero
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

W,S,VT=np.linalg.svd(U) # POD of solution snapshot

# test parameter space
nparam1=15; nparam2=15; ParamD1,ParamD2=np.meshgrid(np.linspace(-1.7,-0.2,nparam1),np.linspace(-1.7,-0.2,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
UROM_g2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))

# Construct Phi_s
ns=5; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
nt=3; D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T

for test_count in range(num_test): # predictive cases
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter

```

```
    # set model
    A=A2D-sp.sparse.diags(1/np.sqrt((x1-param1)**2+(x2-param2)**2),format='csc')
    f=(1/np.sqrt((x1-param1)**2+(x2-param2)**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))

    # construct Ast and fst using block structure (ust0 is zero)
    As=PHIsT@A@PHIs; fs=PHIsT@f
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij+=Dik@Djk-k*Dik@As@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Astij-=Dik_next@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    fst=np.zeros(ns*nt)
    for j in range(nt):
        for kk in range(Nt):
            Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
            fst[ns*j:ns*(j+1)]+=k*Djk@fs[:,kk+1]

    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,fst)

    # FOM
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]+UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_g2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    urom[0]=np.zeros((N+1,N+1)); ufom[0]=np.zeros((N+1,N+1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_g2[test_count]=urom; UFOM_2[test_count]=ufom
```

A.2 Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(-0.9,-0.5,2),np.linspace(-0.9,-0.5,2),indexing='ij')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1/N; k=2/Nt
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
x1=x1.flatten(); x2=x2.flatten(); t=np.linspace(0,2,Nt+1)

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D=sp.sparse.kron(A1D,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D,format='csc')

# snapshot, free DoFs, No IC included
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=A2D-sp.sparse.diags(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2),format='csc')
    f=(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
    u=np.zeros((Ns,Nt+1)) # IC is zero
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

W,S,VT=np.linalg.svd(U) # POD of solution snapshot

# test parameter space
nparam1=15; nparam2=15; ParamD1,ParamD2=np.meshgrid(np.linspace(-1.7,-0.2,nparam1),np.linspace(-1.7,-0.2,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))

# Construct Phi_s
ns=5; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
nt=3;  D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T
```

```

# predictive cases
for test_count in range(num_test):
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter

    # set model
    A=A2D-sp.sparse.diags(1/np.sqrt((x1-param1)**2+(x2-param2)**2),format='csc')
    f=(1/np.sqrt((x1-param1)**2+(x2-param2)**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))

    # construct Ast and fst using block structure (ust0 is zero)
    As=PHIsT@A@PHIs; fs=PHIsT@f
    ATAs=PHIsT@(I-k*A.T)@(I-k*A)@PHIs; AIs=PHIsT@(I-k*A)@PHIs; ATIs=PHIsT@(I-k*A.T)@PHIs; ATIfs=PHIsT@(I-k*A.T)@f
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij += Dik@ATAs@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Djk_next=sp.sparse.diags(D[:,j*Nt+kk+1],format='csc')
                    Astij += Dik@Djk - Dik@AIs@Djk_next - Dik_next@ATIs@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    fst=np.zeros(ns*nt)
    for j in range(nt):
        for kk in range(Nt):
            Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
            fst[ns*j:ns*(j+1)]+= k*Djk@ATIfs[:,kk+1]
            if kk != Nt-1:
                fs_next = fs[:,kk+1+1]
                fst[ns*j:ns*(j+1)]-=k*Djk@fs_next

    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,fst)

    # FOM
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]+UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_pg2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    urom[0]=np.zeros((N+1,N+1)); ufom[0]=np.zeros((N+1,N+1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_pg2[test_count]=urom; UFOM_2[test_count]=ufom
```

A.3 Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(0.03,0.05,2),np.linspace(0.33,0.35,2),indexing='ij')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')

# IC
u0=100*np.sin(2*np.pi*x1)**3*np.sin(2*np.pi*x2)**3; u0[np.nonzero(x1>0.5)]=0.0; u0[np.nonzero(x2>0.5)]=0.0; u0=u0.flatten()

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D_diff=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D_diff=sp.sparse.kron(A1D_diff,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_diff,format='csc')
A1D_conv=1/(h)*sp.sparse.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
A2D_conv=sp.sparse.kron(A1D_conv,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_conv,format='csc')

# snapshot, free DoFs, No IC included
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff
    u=np.zeros((Ns,Nt+1)); u[:,0]=u0
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n])
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

# POD of solution snapshot
W,S,VT=np.linalg.svd(U)

# test parameter space
nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.01,0.07,nparam1),np.linspace(0.31,0.37,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
UROM_g2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))

# Construct Phi_s
ns=5; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
```

```
nt=3; D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T

# predictive cases
for test_count in range(num_test):
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu

    A = -param1*A2D_conv + param2*A2D_diff # set model

    # construct Ast, ust (fst is zero)
    As=PHIsT@A@PHIs; us0=PHIsT@u0
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij+=Dik@Djk-k*Dik@As@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Astij-=Dik_next@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    ust=np.zeros(ns*nt)
    for j in range(nt):
        Djk=sp.sparse.diags(D[:,j*Nt],format='csc')
        ust[ns*j:ns*(j+1)] += Djk@us0

    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,ust)

    # FOM
    u=np.zeros((Ns,Nt+1)); u[:,0] = u0
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=u0-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_g2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    urom[0,1:-1,1:-1]=u0.reshape((N-1,N-1)); ufom[0,1:-1,1:-1]=u0.reshape((N-1,N-1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_g2[test_count]=urom; UFOM_2[test_count]=ufom
```

A.4 Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(0.03,0.05,2),np.linspace(0.33,0.35,2),indexing='ij')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')

# IC
u0=100*np.sin(2*np.pi*x1)**3*np.sin(2*np.pi*x2)**3; u0[np.nonzero(x1>0.5)]=0.0; u0[np.nonzero(x2>0.5)]=0.0; u0=u0.flatten()

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D_diff=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D_diff=sp.sparse.kron(A1D_diff,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_diff,format='csc')
A1D_conv=1/(h)*sp.sparse.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
A2D_conv=sp.sparse.kron(A1D_conv,sp.sparse.eye(N-1,format='csc'),format='csc')\
    +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_conv,format='csc')

# snapshot, free DoFs, No IC included
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff
    u=np.zeros((Ns,Nt+1)); u[:,0]=u0
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n])
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

# POD of solution snapshot
W,S,VT=np.linalg.svd(U)

# test parameter space
nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.01,0.07,nparam1),np.linspace(0.31,0.37,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
```

```
UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))

# Construct Phi_s
ns=5; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
nt=3; D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T

# predictive cases
for test_count in range(num_test):
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu

    A= -param1*A2D_conv + param2*A2D_diff # set model

    # construct Ast, ust (fst is zero)
    As=PHIsT@A@PHIs; us0=PHIsT@u0
    ATAs=PHIsT@(I-k*A.T)@(I-k*A)@PHIs; AIs=PHIsT@(I-k*A)@PHIs; ATIs=PHIsT@(I-k*A.T)@PHIs; ATIu0=PHIsT@(I-k*A.T)@u0
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij += Dik@ATAs@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Djk_next=sp.sparse.diags(D[:,j*Nt+kk+1],format='csc')
                    Astij += Dik@Djk - Dik@AIs@Djk_next - Dik_next@ATIs@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    ust=np.zeros(ns*nt)
    for j in range(nt):
        Djk=sp.sparse.diags(D[:,j*Nt],format='csc')
        ust[ns*j:ns*(j+1)] += Djk@ATIu0

    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,ust)

    # FOM
    u=np.zeros((Ns,Nt+1)); u[:,0] = u0
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM)  # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=u0-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_pg2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    urom[0,1:-1,1:-1]=u0.reshape((N-1,N-1)); ufom[0,1:-1,1:-1]=u0.reshape((N-1,N-1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_pg2[test_count]=urom; UFOM_2[test_count]=ufom
```

A.5 Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(0.195,0.205,2),np.linspace(0.018,0.022,2),indexing='xy')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 2.; k=Tfinal/Nt; t=np.linspace(0,Tfinal,Nt+1)
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='xy')
x1=x1.flatten(); x2=x2.flatten()

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D_diff=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D_diff=sp.sparse.kron(A1D_diff,sp.sparse.eye(N-1,format='csc'),format='csc')+\
    sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_diff,format='csc')
A1D_conv=1/(h)*sp.sparse.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
A2D_conv=sp.sparse.kron(A1D_conv,sp.sparse.eye(N-1,format='csc'),format='csc') + \
         0.1*sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_conv,format='csc')

# snapshot, free DoFs, No IC included
f=np.zeros((Ns,Nt+1))
for i in range(Nt+1):
    f[:,i] = 1e5*np.exp(-(((x1-0.5+(0.2*np.sin(2*np.pi*t[i])))/0.1)**2 + ((x2-0)/0.05)**2))
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
```

```
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

# POD of solution snapshot
W,S,VT=np.linalg.svd(U)

# test parameter space
nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.160,0.240,nparam1),np.linspace(0.016,0.024,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
UROM_g2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))

# Construct Phi_s
ns=19; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
nt=3; D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T

# predictive cases
for test_count in range(num_test):
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu

    A= -param1*A2D_conv + param2*A2D_diff # set model

    # construct Ast, fst (ust is zero)
    As=PHIsT@A@PHIs; fs=PHIsT@f
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij+=Dik@Djk-k*Dik@As@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Astij-=Dik_next@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    fst=np.zeros(ns*nt)
    for j in range(nt):
        for kk in range(Nt):
            Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
            fst[ns*j:ns*(j+1)]+=k*Djk@fs[:,kk+1]

    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,fst)

    # FOM
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]+k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_g2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_g2[test_count]=urom; UFOM_2[test_count]=ufom
```

A.6 Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term

```
import numpy as np; import scipy as sp
from scipy.sparse.linalg import spsolve

# train parameter space
mu1,mu2=np.meshgrid(np.linspace(0.195,0.205,2),np.linspace(0.018,0.022,2),indexing='xy')
mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)

# space and time domain
N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt; t=np.linspace(0,Tfinal,Nt+1)
x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='xy')
x1=x1.flatten(); x2=x2.flatten()

# basic operators
e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
A1D_diff=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
A2D_diff=sp.sparse.kron(A1D_diff,sp.sparse.eye(N-1,format='csc'),format='csc')+\
    sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_diff,format='csc')
A1D_conv=1/(h)*sp.sparse.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
A2D_conv=sp.sparse.kron(A1D_conv,sp.sparse.eye(N-1,format='csc'),format='csc') + \
         0.1*sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D_conv,format='csc')

# snapshot, free DoFs, No IC included
f=np.zeros((Ns,Nt+1))
```

```
for i in range(Nt+1):
    f[:,i] = 1e5*np.exp(-(((x1-0.5+(0.2*np.sin(2*np.pi*t[i])))/0.1)**2 + ((x2-0)/0.05)**2))
U=np.zeros((Ns,no_para*Nt))
for p in range(no_para):
    A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]

# POD of solution snapshot
W,S,VT=np.linalg.svd(U)

# test parameter space
nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.160,0.240,nparam1),np.linspace(0.016,0.024,nparam2))
ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)

# variables to store (also store ParamD1 and ParamD2)
UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_pg2=np.zeros((num_test,Nt+1,N+1,N+1))
AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))

# Construct Phi_s
ns=19; PHIs=W[:,:ns]; PHIsT=PHIs.T

# construct Djk
nt=3; D=np.zeros((ns,nt*Nt))
for i in range(ns):
    Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
    Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,:nt]
    for j in range(nt):
        D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]

# construct PHIst for post processing, i.e., reconstruction
PHIst=np.zeros((Ns*Nt,ns*nt))
for i in range(Nt):
    for j in range(nt):
        PHIstij=np.zeros((Ns,ns))
        Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
        PHIst[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
PHIstT=PHIst.T

# predictive cases
for test_count in range(num_test):
    param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu

    A= -param1*A2D_conv + param2*A2D_diff # set model

    # construct Ast, ust (fst is zero)
    As=PHIsT@A@PHIs; fs=PHIsT@f
    ATAs=PHIsT@(I-k*A.T)@(I-k*A)@PHIs; AIs=PHIsT@(I-k*A)@PHIs; ATIs=PHIsT@(I-k*A.T)@PHIs; ATIfs = PHIsT@(I-k*A.T)@f
    Ast=np.zeros((ns*nt,ns*nt))
    for i in range(nt):
        for j in range(nt):
            Astij=np.zeros((ns,ns))
            for kk in range(Nt):
                Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
                Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
                Astij += Dik@ATAs@Djk
                if kk != Nt-1:
                    Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
                    Djk_next=sp.sparse.diags(D[:,j*Nt+kk+1],format='csc')
                    Astij += Dik@Djk - Dik@AIs@Djk_next - Dik_next@ATIs@Djk
            Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
    fst=np.zeros(ns*nt)
    for j in range(nt):
        for kk in range(Nt):
            Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
            fst[ns*j:ns*(j+1)]+= k*Djk@ATIfs[:,kk+1]
            if kk != Nt-1:
                fs_next = fs[:,kk+1+1]
                fst[ns*j:ns*(j+1)]-=k*Djk@fs_next
    # Space-Time ROM (online phase)
    UstROM=PHIst@np.linalg.solve(Ast,fst)

    # FOM
    u=np.zeros((Ns,Nt+1))
    for n in range(Nt):
        u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
    UstFOM=u[:,1:].flatten(order='F')

    AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error

    rstROM=np.zeros(Ns*Nt) # ST residual
    for n in range(1,Nt+1):
        if n==1:
            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
        else:
            rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]+k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
    STResROM_pg2[test_count]=np.linalg.norm(rstROM)

    # store solutions for each param
    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
    for n in range(1,Nt+1):
        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
    UROM_pg2[test_count]=urom; UFOM_pg2[test_count]=ufom
```

References

[1] Youngsoo Choi, Peter Brown, William Arrighi, Robert Anderson, and Kevin Huynh. Space–time reduced order model for large-scale linear dynamical systems with application to boltzmann transport problems. *Journal of Computational Physics*, page 109845, 2020.

[2] C Mullis and RA Roberts. Synthesis of minimum roundoff noise fixed point digital filters. *IEEE Transactions on Circuits and Systems*, 23(9):551–562, 1976.

[3] Bruce Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE transactions on automatic control*, 26(1):17–32, 1981.

[4] Karen Willcox and Jaime Peraire. Balanced model reduction via the proper orthogonal decomposition. *AIAA journal*, 40(11):2323–2330, 2002.

[5] Karen Willcox and Alexandre Megretski. Fourier series for accurate, stable, reduced-order models in large-scale linear applications. *SIAM Journal on Scientific Computing*, 26(3):944–962, 2005.

[6] Matthias Heinkenschloss, Danny C Sorensen, and Kai Sun. Balanced truncation model reduction for a class of descriptor systems with application to the oseen equations. *SIAM Journal on Scientific Computing*, 30(2):1038–1063, 2008.

[7] Henrik Sandberg and Anders Rantzer. Balanced truncation of linear time-varying systems. *IEEE Transactions on automatic control*, 49(2):217–229, 2004.

[8] Carsten Hartmann, Valentina-Mira Vulcanov, and Christof Schütte. Balanced truncation of linear second-order systems: a hamiltonian approach. *Multiscale Modeling & Simulation*, 8(4):1348–1367, 2010.

[9] Mihály Petreczky, Rafael Wisniewski, and John Leth. Balanced truncation for linear switched systems. *Nonlinear Analysis: Hybrid Systems*, 10:4–20, 2013.

[10] Zhanhua Ma, Clarence W Rowley, and Gilead Tadmor. Snapshot-based balanced truncation for linear time-periodic systems. *IEEE Transactions on Automatic Control*, 55(2):469–473, 2010.

[11] Zhaojun Bai. Krylov subspace techniques for reduced-order modeling of large-scale dynamical systems. *Applied numerical mathematics*, 43(1-2):9–44, 2002.

[12] Serkan Gugercin, Athanasios C Antoulas, and Christopher Beattie. H_2 model reduction for large-scale linear dynamical systems. *SIAM journal on matrix analysis and applications*, 30(2):609–638, 2008.

[13] Alessandro Astolfi. Model reduction by moment matching for linear and nonlinear systems. *IEEE Transactions on Automatic Control*, 55(10):2321–2336, 2010.

[14] Eli Chiprout and Michael Nakhla. Generalized moment-matching methods for transient analysis of interconnect networks. In *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, pages 201–206. IEEE, 1992.

[15] Marco Pratesi, Fortunato Santucci, and Fabio Graziosi. Generalized moment matching for the linear combination of lognormal rvs: application to outage analysis in wireless systems. *IEEE Transactions on Wireless Communications*, 5(5):1122–1132, 2006.

[16] Amine Ammar, Béchir Mokdad, Francisco Chinesta, and Roland Keunings. A new family of solvers for some classes of multidimensional partial differential equations encountered in kinetic theory modeling of complex fluids. *Journal of Non-Newtonian Fluid Mechanics*, 139(3):153–176, 2006.

[17] Amine Ammar, Béchir Mokdad, Francisco Chinesta, and Roland Keunings. A new family of solvers for some classes of multidimensional partial differential equations encountered in kinetic theory modelling of complex fluids: Part ii: Transient simulation using space-time separated representations. *Journal of Non-Newtonian Fluid Mechanics*, 144(2-3):98–121, 2007.

[18] Francisco Chinesta, Amine Ammar, and Elías Cueto. Proper generalized decomposition of multiscale models. *International Journal for Numerical Methods in Engineering*, 83(8-9):1114–1132, 2010.

[19] Etienne Pruliere, Francisco Chinesta, and Amine Ammar. On the deterministic solution of multidimensional parametric models using the proper generalized decomposition. *Mathematics and Computers in Simulation*, 81(4):791–810, 2010.

[20] Francisco Chinesta, Amine Ammar, Adrien Leygue, and Roland Keunings. An overview of the proper generalized decomposition with applications in computational rheology. *Journal of Non-Newtonian Fluid Mechanics*, 166(11):578–592, 2011.

[21] Eugenio Giner, Brice Bognet, Juan J Ródenas, Adrien Leygue, F Javier Fuenmayor, and Francisco Chinesta. The proper generalized decomposition (pgd) as a numerical procedure to solve 3d cracked plates in linear elastic fracture mechanics. *International Journal of Solids and Structures*, 50(10):1710–1720, 2013.

[22] Andrea Barbarulo, Pierre Ladevèze, Hervé Riou, and Louis Kovalevsky. Proper generalized decomposition applied to linear acoustic: a new tool for broad band calculation. *Journal of Sound and Vibration*, 333(11):2422–2431, 2014.

[23] David Amsallem and Charbel Farhat. Stabilization of projection-based reduced-order models. *International Journal for Numerical Methods in Engineering*, 91(4):358–377, 2012.

[24] David Amsallem and Charbel Farhat. Interpolation method for adapting reduced-order models and application to aeroelasticity. *AIAA journal*, 46(7):1803–1813, 2008.

[25] Jeffrey P Thomas, Earl H Dowell, and Kenneth C Hall. Three-dimensional transonic aeroelasticity using proper orthogonal decomposition-based reduced-order models. *Journal of Aircraft*, 40(3):544–551, 2003.

[26] Kenneth C Hall, Jeffrey P Thomas, and Earl H Dowell. Proper orthogonal decomposition technique for transonic unsteady aerodynamic flows. *AIAA journal*, 38(10):1853–1862, 2000.

[27] Francisco Chinesta, Pierre Ladeveze, and Elias Cueto. A short review on model order reduction based on proper generalized decomposition. *Archives of Computational Methods in Engineering*, 18(4):395, 2011.

[28] AJ Mayo and AC Antoulas. A framework for the solution of the generalized realization problem. *Linear algebra and its applications*, 425(2-3):634–662, 2007.

[29] Giordano Scarciotti and Alessandro Astolfi. Data-driven model reduction by moment matching for linear and nonlinear systems. *Automatica*, 79:340–351, 2017.

[30] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

[31] Kevin K Chen, Jonathan H Tu, and Clarence W Rowley. Variants of dynamic mode decomposition: boundary condition, koopman, and fourier analyses. *Journal of nonlinear science*, 22(6):887–915, 2012.

[32] Matthew O Williams, Ioannis G Kevrekidis, and Clarence W Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, 2015.

[33] Naoya Takeishi, Yoshinobu Kawahara, and Takehisa Yairi. Learning koopman invariant subspaces for dynamic mode decomposition. In *Advances in Neural Information Processing Systems*, pages 1130–1140, 2017.

[34] Travis Askham and J Nathan Kutz. Variable projection methods for an optimized dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 17(1):380–416, 2018.

[35] Peter J Schmid, Larry Li, Matthew P Juniper, and O Pust. Applications of the dynamic mode decomposition. *Theoretical and Computational Fluid Dynamics*, 25(1-4):249–259, 2011.

[36] J Nathan Kutz, Xing Fu, and Steven L Brunton. Multiresolution dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 15(2):713–735, 2016.

[37] Qianxiao Li, Felix Dietrich, Erik M Bollt, and Ioannis G Kevrekidis. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10):103111, 2017.

[38] Joshua L Proctor, Steven L Brunton, and J Nathan Kutz. Dynamic mode decomposition with control. *SIAM Journal on Applied Dynamical Systems*, 15(1):142–161, 2016.

[39] Jonathan H Tu, Clarence W Rowley, Dirk M Luchtenburg, Steven L Brunton, and J Nathan Kutz. On dynamic mode decomposition: theory and applications. *arXiv preprint arXiv:1312.0041*, 2013.

[40] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016.

[41] Gal Berkooz, Philip Holmes, and John L Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. *Annual review of fluid mechanics*, 25(1):539–575, 1993.

[42] Martin Gubisch and Stefan Volkwein. Proper orthogonal decomposition for linear-quadratic optimal control. *Model reduction and approximation: theory and algorithms*, 5:66, 2017.

[43] Karl Kunisch and Stefan Volkwein. Galerkin proper orthogonal decomposition methods for parabolic problems. *Numerische mathematik*, 90(1):117–148, 2001.

[44] Michael Hinze and Stefan Volkwein. Error estimates for abstract linear–quadratic optimal control problems using proper orthogonal decomposition. *Computational Optimization and Applications*, 39(3):319–345, 2008.

[45] Gaetan Kerschen, Jean-claude Golinval, Alexander F Vakakis, and Lawrence A Bergman. The method of proper orthogonal decomposition for dynamical characterization and order reduction of mechanical systems: an overview. *Nonlinear dynamics*, 41(1-3):147–169, 2005.

[46] Franz Bamer and Christian Bucher. Application of the proper orthogonal decomposition for linear and nonlinear structures under transient excitations. *Acta Mechanica*, 223(12):2549–2563, 2012.

[47] Jeanne A Atwell and Belinda B King. Proper orthogonal decomposition for reduced basis feedback controllers for parabolic equations. *Mathematical and computer modelling*, 33(1-3):1–19, 2001.

[48] Muruhan Rathinam and Linda R Petzold. A new look at proper orthogonal decomposition. *SIAM Journal on Numerical Analysis*, 41(5):1893–1925, 2003.

[49] Martin Kahlbacher and Stefan Volkwein. Galerkin proper orthogonal decomposition methods for parameter dependent elliptic systems. *Discussiones Mathematicae, Differential Inclusions, Control and Optimization*, 27(1):95–117, 2007.

[50] Jean P Bonnet, Daniel R Cole, Joël Delville, Mark N Glauser, and Lawrence S Ukeiley. Stochastic estimation and proper orthogonal decomposition: complementary techniques for identifying structure. *Experiments in fluids*, 17(5):307–314, 1994.

[51] Antoine Placzek, D-M Tran, and Roger Ohayon. Hybrid proper orthogonal decomposition formulation for linear structural dynamics. *Journal of Sound and Vibration*, 318(4-5):943–964, 2008.

[52] Patrick LeGresley and Juan Alonso. Airfoil design optimization using reduced order models based on proper orthogonal decomposition. In *Fluids 2000 conference and exhibit*, page 2545, 2000.

[53] Mehmet Onder Efe and Hitay Ozbay. Proper orthogonal decomposition for reduced order modeling: 2d heat flow. In *Proceedings of 2003 IEEE Conference on Control Applications, 2003. CCA 2003.*, volume 2, pages 1273–1277. IEEE, 2003.

[54] Youngsoo Choi and Kevin Carlberg. Space–time least-squares petrov–galerkin projection for nonlinear model reduction. *SIAM Journal on Scientific Computing*, 41(1):A26–A58, 2019.

[55] Karsten Urban and Anthony Patera. An improved error bound for reduced basis approximation of linear parabolic problems. *Mathematics of Computation*, 83(288):1599–1615, 2014.

[56] Masayuki Yano, Anthony T Patera, and Karsten Urban. A space-time hp-interpolation-based certified reduced basis method for burgers' equation. *Mathematical Models and Methods in Applied Sciences*, 24(09):1903–1935, 2014.

[57] Masayuki Yano. A space-time petrov–galerkin certified reduced basis method: Application to the boussinesq equations. *SIAM Journal on Scientific Computing*, 36(1):A232–A266, 2014.

[58] Gil Ho Yoon. Structural topology optimization for frequency response problem using model reduction schemes. *Computer Methods in Applied Mechanics and Engineering*, 199(25-28):1744–1763, 2010.

[59] Oded Amir, Mathias Stolpe, and Ole Sigmund. Efficient use of iterative solvers in nested topology optimization. *Structural and Multidisciplinary Optimization*, 42(1):55–72, 2010.

[60] David Amsallem, Matthew Zahr, Youngsoo Choi, and Charbel Farhat. Design optimization using hyper-reduced-order modelsvd. *Structural and Multidisciplinary Optimization*, 51(4):919–940, 2015.

[61] Christian Gogu. Improving the efficiency of large scale topology optimization through on-the-fly reduced order model construction. *International Journal for Numerical Methods in Engineering*, 101(4):281–304, 2015.

[62] Youngsoo Choi, Gabriele Boncoraglio, Spenser Anderson, David Amsallem, and Charbel Farhat. Gradient-based constrained optimization using a database of linear reduced-order models. *Journal of Computational Physics*, page 109787, 2020.

[63] Youngsoo Choi, Geoffrey Oxberry, Daniel White, and Trenton Kirchdoerfer. Accelerating design optimization using reduced order models. *arXiv preprint arXiv:1909.11320*, 2019.

[64] Daniel A White, Youngsoo Choi, and Jun Kudo. A dual mesh method with adaptivity for stress-constrained topology optimization. *Structural and Multidisciplinary Optimization*, 61(2):749–762, 2020.

[65] Habib N Najm. Uncertainty quantification and polynomial chaos techniques in computational fluid dynamics. *Annual review of fluid mechanics*, 41:35–52, 2009.

[66] Robert W Walters and Luc Huyse. Uncertainty analysis for fluid mechanics with applications. Technical report, NATIONAL AERONAUTICS AND SPACE ADMINISTRATION HAMPTON VA LANGLEY RESEARCH CENTER, 2002.

[67] Thomas A Zang. *Needs and opportunities for uncertainty-based multidisciplinary design methods for aerospace vehicles*. National Aeronautics and Space Administration, Langley Research Center, 2002.

[68] N Anders Petersson, Fortino M Garcia, Austin E Copeland, Ylva L Rydin, and Jonathan L DuBois. Discrete adjoints for accurate numerical optimization with application to quantum control. *arXiv preprint arXiv:2001.01013*, 2020.

[69] Youngsoo Choi, Charbel Farhat, Walter Murray, and Michael Saunders. A practical factorization of a schur complement for pde-constrained distributed optimal control. *Journal of Scientific Computing*, 65(2):576–597, 2015.

[70] Youngsoo Choi. *Simultaneous analysis and design in PDE-constrained optimization*. PhD thesis, Stanford University, 2012.

[71] Lawrence Sirovich. Turbulence and the dynamics of coherent structures. i. coherent structures. *Quarterly of applied mathematics*, 45(3):561–571, 1987.

[72] Kookjin Lee and Kevin T Carlberg. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders. *Journal of Computational Physics*, 404:108973, 2020.

[73] Youngkyu Kim, Youngsoo Choi, David Widemann, and Tarek Zohdi. A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder, 2020.