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 $O(10^2)$ speed-ups with $O(10^{-3})$ to $O(10^{-4})$ 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, neuro-transmission 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)
§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 compute computational complexity of forming the space–time ROM operators in terms of the blocks. Then, we compared Galerkin and least-squares Petrov–Galerkin space–time ROM operators derived in [1] in Section 5.1. In Section 5.2, we derive least-squares Petrov–Galerkin space–time ROM operators 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).

\[
\frac{\partial u(t; \mu)}{\partial t} = A(\mu)u(t; \mu) + B(\mu)f(t; \mu), \quad u(0; \mu) = u^0(\mu),
\]

(2.1)

where \( \mu \in \Omega_\mu \subset \mathbb{R}^n \) denotes a parameter vector, \( u : [0, T] \times \mathbb{R}^s \to \mathbb{R}^N_s \) denotes a time dependent state variable function, \( u^0 : \mathbb{R}^s \to \mathbb{R}^N_s \) denotes an initial state, and \( f : [0, T] \times \mathbb{R}^s \to \mathbb{R}^N_i \) denotes a time dependent input variable function. The operators \( A : \mathbb{R}^s \to \mathbb{R}^{N_s \times N_s}, B : \mathbb{R}^s \to \mathbb{R}^{N_s \times N_i} \), 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):

\[
\begin{pmatrix}
I_{N_s} - \Delta t^{(k)} A^{(k)} \\
\end{pmatrix}
\begin{pmatrix}
u^{(k)} \\
\end{pmatrix} = \begin{pmatrix}
u^{(k-1)} + \Delta t^{(k)} B^{(k)} f^{(k)}(\mu) \\
\end{pmatrix},
\]

(2.2)

where \( I_{N_s} \in \mathbb{R}^{N_s \times N_s} \) is the identity matrix, \( \Delta t^{(k)} \) is the 4th time step size with \( T = \sum_{k=1}^{n} \Delta t^{(k)} \) and \( \Delta t^{(k)} = \Delta t \sum_{k=1}^{n} \Delta t^{(k)} \), and \( u^{(k)}(\mu) := u(I_{k}; \mu) \) and \( f^{(k)}(\mu) := f(I_{k}; \mu) \) are the state and input vectors at 4th time step where \( k \in \mathbb{N}(\mathbb{N}) \). 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.

\[
A^{(s)}(\mu)u^{(s)}(\mu) = f^{(s)}(\mu) + u^{(0)}(\mu),
\]

(2.3)

where

\[
A^{(s)}(\mu) = \begin{bmatrix}
I_{N_s} - \Delta t^{(1)} A(\mu) & -I_{N_s} & \ldots & -I_{N_s} \\
I_{N_s} - \Delta t^{(2)} A(\mu) & \ldots & \ldots & \ldots \\
\vdots & \vdots & \ddots & \vdots \\
I_{N_s} - \Delta t^{(n)} A(\mu) & \ldots & \ldots & I_{N_s} - \Delta t^{(n)} A(\mu)
\end{bmatrix},
\]

(2.4)

\[
u^{(s)}(\mu) = \begin{bmatrix}
u^{(1)}(\mu) \\
u^{(2)}(\mu) \\
\vdots \\
u^{(n)}(\mu)
\end{bmatrix},
\]

(2.5)

\[
f^{(s)}(\mu) = \begin{bmatrix}
\Delta t^{(1)} B(\mu) f^{(1)}(\mu) \\
\Delta t^{(2)} B(\mu) f^{(2)}(\mu) \\
\vdots \\
\Delta t^{(n)} B(\mu) f^{(n)}(\mu)
\end{bmatrix},
\]

(2.6)

\[
u^{(0)}(\mu) = \begin{bmatrix}
u^{(1)}(\mu) \\
0 \\
\vdots \\
0
\end{bmatrix}
\]

(2.7)
The space–time system matrix $A^s$ has dimensions $\mathbb{R}^{n_s} \to \mathbb{R}^{N_s \times N_t}$, the space–time state vector $u^s$ has dimensions $\mathbb{R}^{n_s} \to \mathbb{R}^{N_s}$, the space–time input vector $f^s$ has dimensions $\mathbb{R}^{n_s} \to \mathbb{R}^{N_t}$, and the space–time initial vector $u_0^s$ has dimensions $\mathbb{R}^{n_s} \to \mathbb{R}^{N_s}$. 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:

$$u^s(\mu) \approx \hat{u}^s(\mu) \equiv \Phi_{n} \hat{u}^s(\mu),$$

(3.1)

where $\hat{u}^s(\mu) : \mathbb{R}^{n_s} \to \mathbb{R}^{n_s}$ with $n_s \ll N_s$ and $n_t \ll N_t$. The space–time basis, $\Phi_{n} \in \mathbb{R}^{N_s \times n_n}$ is defined as

$$\Phi_{n} := \begin{bmatrix} \phi_{i(n_s + j-1)} & \cdots & \phi_{i(n_s)} \end{bmatrix},$$

(3.2)

where $i \in \mathbb{N}(n_0)$, $j \in \mathbb{N}(n_t)$. Substituting Equation (3.1) into the space–time formulation in Equation (2.3) gives an over-determined system of equations:

$$A^s(\mu) \Phi_{n} \hat{u}^s(\mu) = f^s(\mu) + u_0^s(\mu)$$

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 $\Phi_{n}^T$.

$$\Phi_{n}^T A^s(\mu) \Phi_{n} \hat{u}^s(\mu) = \Phi_{n}^T f^s(\mu) + \Phi_{n}^T u_0^s(\mu).$$

(3.4)

For notational simplicity, let us define the reduced space–time system matrix as $\hat{A}^{s,\beta}(\mu) := \Phi_{n}^T A^s(\mu) \Phi_{n}$, reduced space–time input vector as $\hat{f}^{s,\beta}(\mu) := \Phi_{n}^T f^s(\mu)$, and reduced space–time initial state vector as $\hat{u}_{0}^{s,\beta}(\mu) := \Phi_{n}^T u_0^s(\mu)$.

3.3 Least-squares Petrov–Galerkin projection

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

$$\tilde{f}^s(\hat{u}^s; \mu) = f^s(\mu) + u_0^s(\mu) - A^s(\mu) \Phi_{n} \hat{u}^s(\mu),$$

(3.5)

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

$$\hat{u}^s = \arg\min_{u \in \mathbb{R}^{n_s}} \frac{1}{2} \| \tilde{f}^s(\hat{u}^s, \mu) \|^2_2.$$

(3.6)

The solution to Equation (3.6) satisfies

$$(\hat{A}^{s,\beta}(\mu) \Phi_{n})^T \tilde{f}^s(\hat{u}^s, \mu) = 0$$

(3.7)

leading to

$$\Phi_{n}^T A^{s,\beta}(\mu)^T \hat{A}^{s,\beta}(\mu) \Phi_{n} \hat{u}^s(\mu) = \Phi_{n}^T f^{s,\beta}(\mu) + \Phi_{n}^T u_0^{s,\beta}(\mu).$$

(3.8)

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

$$\hat{A}^{s,\beta}(\mu) := \Phi_{n}^T A^{s,\beta}(\mu)^T \hat{A}^{s,\beta}(\mu) \Phi_{n},$$

reduced space–time input vector as $\hat{f}^{s,\beta}(\mu) := \Phi_{n}^T A^{s,\beta}(\mu)^T f^{s,\beta}(\mu)$, and reduced space–time initial state vector as $\hat{u}_{0}^{s,\beta}(\mu) := \Phi_{n}^T A^{s,\beta}(\mu)^T u_0^{s,\beta}(\mu)$.

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 |
|---------|----------------|
| $A^{s,\beta}(\mu) := \Phi_{n}^T A^{s}(\mu) \Phi_{n}$ | $\hat{A}^{s,\beta}(\mu) := \Phi_{n}^T A^{s}(\mu)^T \hat{A}^{s}(\mu) \Phi_{n}$ |
| $\hat{f}^{s,\beta}(\mu) := \Phi_{n}^T f^{s}(\mu)$ | $\hat{f}^{s,\beta}(\mu) := \Phi_{n}^T f^{s}(\mu)$ |
| $\hat{u}_{0}^{s,\beta}(\mu) := \Phi_{n}^T u_0^{s}(\mu)$ | $\hat{u}_{0}^{s,\beta}(\mu) := \Phi_{n}^T u_0^{s}(\mu)$ |
4 Space-time Basis Generation

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

We follow the method of snapshots described by Sirovich [71]. First, let \( \{\mathbf{\mu}_1, \ldots, \mathbf{\mu}_{n_s}\} \) be a set of parameter samples, where we run full order model simulations. Let \( U_p \equiv [\mathbf{u}^{(1)}(\mathbf{\mu}_p) \ldots \mathbf{u}^{(N_s)}(\mathbf{\mu}_p)] \in \mathbb{R}^{N \times N_s}, \ p \in \mathbb{N}(n_p), \) be a full order model solution matrix for a sample parameter, \( \mathbf{\mu}_p \in \mathbb{S}_{p} \). Then concatenating all the solution matrices defines a snapshot matrix, \( U \in \mathbb{R}^{N \times n_s N_s}, \) i.e.,

\[
U \equiv [U_1 \ldots U_{n_s}] .
\]

We use Proper Orthogonal Decomposition (POD) to construct the spatial basis, \( \Phi \), the temporal basis, \( \Psi \), and 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.

**Figure 1:** Illustration of spatial and temporal bases construction, using SVD with \( n_s = 3 \). The right singular vector, \( \mathbf{v}_i \), describes three different temporal behaviors of a left singular basis vector \( \mathbf{w}_i \), i.e., three different temporal behaviors of a spatial mode. Each temporal behavior is denoted as \( \mathbf{v}_{i1}^T, \mathbf{v}_{i2}^T, \) and \( \mathbf{v}_{i3}^T \).

We set \( \mathbf{U}_i \equiv \mathbf{w}_i \mathbf{v}_i^T \) to be the leading \( n_s \times n_s \) block matrices, which are associated with the \( n_s \)-dimensional subspace that optimally represents the solution snapshot, \( \mathbf{w}_i \). The equivalent sum-of-squares form is written in (4.3), where \( \mathbf{v}_i \) is the \( i \)th singular value, \( \mathbf{w}_i \) and \( \mathbf{v}_i \) are \( i \)th left and right singular vectors, respectively. Note that \( \mathbf{v}_i \) describes \( n_s \) different temporal behaviors of \( \mathbf{w}_i \). For example, Figure 1 illustrates the case of \( n_s = 3 \), where \( \mathbf{v}_{i1}^T, \mathbf{v}_{i2}^T, \) and \( \mathbf{v}_{i3}^T \) describe three different temporal behaviors of a specific spatial basis vector, i.e., \( \mathbf{w}_i \). For general \( n_s \), we note that \( \mathbf{v}_i \) describes \( n_s \) different temporal behaviors of \( i \)th spatial basis vector, i.e., \( \mathbf{v}_i \). We set \( \mathbf{\Psi}_i = \left[ \mathbf{w}_{i1} \mathbf{w}_{i2} \cdots \mathbf{w}_{iN_s} \right]^T \) to be \( i \)th temporal snapshot matrix, where \( \mathbf{w}_i^T \equiv [\mathbf{w}_i(1 + (k - 1) N_s), \mathbf{w}_i(2 + (k - 1) N_s), \ldots, \mathbf{w}_i(k N_s)]^T \in \mathbb{R}^{N_s} \) for \( k \in \mathbb{N}(n_s) \), where \( \mathbf{w}_i(j), j \in \mathbb{N}(n_s) \), is the \( j \)th component of the vector. The SVD of \( \mathbf{\Psi}_i \) is

\[
\mathbf{\Psi}_i = \Lambda_i \mathbf{\Sigma}_i \Psi_i^T .
\]

Then, choosing the leading \( n_s \) vectors of \( \Lambda_i \), yields the temporal basis, \( \Psi_i \), for \( i \)th spatial basis vector. Finally, we can construct a space–time basis vector, \( \phi_{i+\mu_{n_s}(j-1)}^{\mu_{n_s}(j-1)} \in \mathbb{R}^{N \times N_s} \), in Equation (2.4) as

\[
\phi_{i+\mu_{n_s}(j-1)}^{\mu_{n_s}(j-1)} = \phi_{ij}^T \otimes \phi_i^T ,
\]

where \( \otimes \) denotes Kronecker product, \( \phi_i^T \in \mathbb{R}^{N_s} \) is \( i \)th vector of the spatial basis, \( \Phi \), and \( \phi_{ij}^T \in \mathbb{R}^{N_s} \) is \( j \)th vector of the temporal basis, \( \Psi_i \) that describes a temporal behavior of \( \phi_i^T \).

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 Petrov–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:

\[ \Phi_{st} = \begin{pmatrix} \Phi_1 & \cdots & \Phi_{N_t} \\ \Phi_2 & \cdots & \Phi_{N_t} \\ \vdots & \ddots & \vdots \\ \Phi_{N_t} & \cdots & \Phi_{N_t} \end{pmatrix} \in \mathbb{R}^{N_t \times N_s \times n_s} \] (5.1)

where the \( k \)th time step of the temporal basis matrix is a diagonal matrix defined as

\[ D_k = \begin{pmatrix} \phi_{1,k} & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \phi_{N_t,k} \end{pmatrix} \in \mathbb{R}^{N_s \times N_s} \] (5.2)

where \( \phi_{i,k} \in \mathbb{R} \) is the \( k \)th element of \( \phi_i \in \mathbb{R}^{N_t} \).

5.2 Block structures of Galerkin projection

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

\[ \hat{A}^{st-g} (\mu) = \Phi_{st}^T \hat{A} (\mu) \Phi_{st} \] (5.3)

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

\[ \hat{A}^{st-g} (\mu) = \begin{pmatrix} \hat{A}^{st-g}_{1,1} (\mu) & \cdots & \hat{A}^{st-g}_{1,n_g} (\mu) \\ \cdots & \ddots & \cdots \\ \hat{A}^{st-g}_{n_s,1} (\mu) & \cdots & \hat{A}^{st-g}_{n_s,n_g} (\mu) \end{pmatrix} \] (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 \( \hat{A}^{st-g}_{j,j} (\mu) \in \mathbb{R}^{N_s \times n_g} \) where the \((j', j)\)th block matrix is:

\[ \hat{A}^{st-g}_{j', j} (\mu) = \sum_{k=1}^{N_t} (D_k^T D_k - \Delta^{(k)} D_k^T \Phi_{st}^T A (\mu) \Phi_{st} D_k) - \sum_{k=1}^{N_t-1} D_{k+1}^T D_k \] (5.5)

The reduced space–time Galerkin input vector \( \hat{f}^{st-g} (\mu) \in \mathbb{R}^{n_s \times n_g} \) is

\[ \hat{f}^{st-g} (\mu) = \Phi_{st}^T \hat{f} (\mu). \] (5.6)

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

\[ \hat{f}^{st-g}(\mu)_{(j)} = \sum_{k=1}^{N_t} \Delta^{(k)} D_k^T \Phi_{st}^T B (\mu) f^{(k)} (\mu). \] (5.7)

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

\[ \hat{u}_0^{st-g} (\mu) = \Phi_{st}^T u_0 (\mu). \] (5.8)

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

\[ \hat{u}_0^{st-g}(\mu)_{(j)} = D_j^T \Phi_{st}^T u_0 (\mu). \] (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, \( \hat{A}^{st-pg} (\mu) \) is:

\[ \hat{A}^{st-pg} (\mu) = \Phi_{st}^T \hat{A}^{st} (\mu)^T \hat{A}^{st} (\mu) \Phi_{st} \] (5.10)

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

\[ \hat{A}^{st-pg} (\mu) = \begin{pmatrix} \hat{A}^{st-pg}_{1,1} (\mu) & \cdots & \hat{A}^{st-pg}_{1,n_g} (\mu) \\ \cdots & \ddots & \cdots \\ \hat{A}^{st-pg}_{n_s,1} (\mu) & \cdots & \hat{A}^{st-pg}_{n_s,n_g} (\mu) \end{pmatrix} \] (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 $A_{i',j'}^{st,pg}(\mu) \in \mathbb{R}^{n_s \times n_s}$ where the $(j',j)$th block matrix is:

$$A_{i',j'}^{st,pg}(\mu) = \sum_{k=0}^{N_s} \left[ D_k^T \Phi_i^T \left( I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \left( I_{N_s} - \Delta t^{(k)} A(\mu) \right) \Phi_j \right]$$

(5.12)

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

$$\hat{f}^{st,pg}(\mu) = \Phi_i^T A^*(\mu)^T \Phi_j u(\mu).$$

(5.13)

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.

| Galerkin | Petrov–Galerkin |
|----------|----------------|
| $A^{st}(\mu)_{i',j'} = \sum_{k=0}^{N_s} \left[ D_k^T \Phi_i^T \left( I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \left( I_{N_s} - \Delta t^{(k)} A(\mu) \right) \Phi_j \right]$ | $A^{st,pg}(\mu)_{i',j'} = \sum_{k=0}^{N_s} \left[ D_k^T \Phi_i^T \left( I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \left( I_{N_s} - \Delta t^{(k)} A(\mu) \right) \Phi_j \right]$ |
| $f^{st}(\mu)_{i',j'} = \sum_{k=0}^{N_s} \left[ \Phi_i^T \left( I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \Delta t^{(k)} B(\mu) \right] \Phi_j$ | $f^{st,pg}(\mu)_{i',j'} = \sum_{k=0}^{N_s} \left[ \Phi_i^T \left( I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \Delta t^{(k)} B(\mu) \right] \Phi_j$ |
| $u_{0}^{st}(\mu)_{i',j'} = D_k^T \Phi_i^T u(\mu)$ | $u_{0}^{st,pg}(\mu)_{i',j'} = D_k^T \Phi_i^T u(\mu)$ |

5.5 Computational complexity of forming space–time ROM operators

To compute the 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 $A(\mu) \in \mathbb{R}^{N_s \times N_s}$ is a band matrix with the bandwidth, $b$ and $B(\mu)$ is a identity matrix, $I_{N_s}$. The band structure of $A(\mu)$ is often seen in mathematical models because of local approximations to derivative terms. Then, the bandwidth of $A^{st}(\mu) \in \mathbb{R}^{N_s \times N_s}$ formed with backward Euler scheme is $N_s$. We also assume that the spatial basis vectors $\phi_i^T \in \mathbb{R}^{N_s}, i \in \mathbb{N}(n_s)$ and temporal basis vectors $\phi_i^T \in \mathbb{R}^{N_s}, i \in \mathbb{N}(n_t)$ and $j \in \mathbb{N}(n_s)$ are given.

Let us start to compute the computational cost without use of block structures. Constructing space–time basis costs $\Theta(N_s, N_s, n_s)$. For Galerkin projections, computing the reduced space–time system matrix, input vectors, and initial state vector costs $\Theta(2N_s^2 N_s, n_s) + \Theta(N_s N_s, n_s^2)$, $\Theta(N_s, N_s, n_s)$, and $\Theta(N_s, n_s)$, respectively. Then, keeping the dominant terms and taking off coefficient 2 lead to $\Theta(2N_s^2 N_s, n_s) + \Theta(N_s, n_s)^2$. With the assumption of $n_t, n_s \ll N_s$, we have $\Theta(N_s, n_s)^2$.

For Petrov–Galerkin projections, we first compute $\Phi_i^T A(\mu)$ resulting in $\Theta(N_s, n_s)$. Then, computing the reduced space–time system matrix, input vectors, and initial state vector costs $\Theta(N_s, n_s)^2$, $\Theta(N_s, n_s, n_s)$, and $\Theta(N_s, n_s)$, respectively. Then, keeping the dominant terms and taking off coefficient 2 lead to $\Theta(N_s, n_s)^2$. With the assumption of $n_t, n_s \ll N_s$, we have $\Theta(N_s, n_s)^2$.

Now, let us compute the computational complexity with use of block structures. For Galerkin projection, we first compute $\Phi_i^T A(\mu) \Phi_j$, which will be reused, resulting in $\Theta(2bN_s, n_s)$. Then, we compute $n_s^2$ blocks for the reduced space–time system matrix. It takes $\Theta(N_s(2n_s^2 + 2n_s))$ to compute each block. Thus, it costs $\Theta(N_s(2n_s^2 + 2n_s))$ to compute the reduced space–time system matrix. For the reduced space–time input vector, $n_s$ blocks are needed and each block costs $\Theta(N_s, n_s + n_t)$, resulting in $\Theta(6N_s(n_n + n_t))$. The computing the reduced space–time initial vector costs $\Theta(N_s + n_t)$. Thus, keeping the dominant terms and taking off coefficient 2 lead to $\Theta(bN_s, n_s) + \Theta(N_s n_s + 2) + \Theta(N_s, n_s^2)$. With the assumptions of $b \ll N_s, n_s \ll N_s$, and $n_t, n_s \ll N_s$, we have $\Theta(N_s, n_s^2)$. For Petrov–Galerkin projection, we compute $(I_{N_s} - \Delta t A(\mu)) \Phi_j$, resulting in $\Theta(bN_s, n_s)$. Then, we compute $\Phi_i^T (I_{N_s} - \Delta t A(\mu)^T) \Phi_j$, and $\Phi_i^T (I_{N_s} - \Delta t A(\mu)) \Phi_j$, for re-use. Each of them costs $\Theta(N_s^2 n_s)$. Now, we compute $n_s^2$ blocks for the reduced space–time system matrix. It takes $\Theta(N_s(n_s^2 + n_s))$ to compute each.
block. Thus, it costs $O(Nn^2(0n_s+n_n))$ to compute the reduced space–time system matrix. For the reduced space–time input vector, $n_t$ blocks are needed and each block costs $O(N(Nn_s+n_n))$, resulting in $O(Nn_t(Nn_s+n_n))$. The computing the reduced space–time initial vector costs $O(Nn_t+n_n)$. Thus, keeping the dominant terms and taking off coefficients 2 and 6 lead to $O(bNn_s+n_s+O(Nn_s+n_n))$ and $O(N(Nn_s+n_n))$. With the assumptions of $b<n_t$, $n_s<n_t$, and $n_n<n_t$, we have $O(N(Nn_s+n_n))$.

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 | $O(N_s^2N_t n_n)$ | $O(N_s^2N_t n_n)$ |
| Using block structures   | $O(N_s^2N_t n_n)$ | $O(N_s^2N_t n_n)$ |

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 $e(k) = u(k) - \hat{u}(k) \in \mathbb{R}^{N_s}$, where $u(k) \in \mathbb{R}^{N_s}$ denotes FOM solution, $\hat{u}(k) \in \mathbb{R}^{N_s}$ denotes approximate solution, and $k \in \mathbb{N}(N_t)$. Let $A^s \in \mathbb{R}^{N_s \times N_s}$ be the space–time system matrix, $r(k) \in \mathbb{R}^{N_s}$ be the residual computed using FOM solution at kth time step, and $\hat{r}(k) \in \mathbb{R}^{N_s}$ be the residual computed using approximate solution at kth time step. For example, $r(k)$ and $\hat{r}(k)$ after applying the backward Euler scheme with uniform time step become

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

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

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

$$\max_{k \in \mathbb{N}(N_t)} \|e(k)\|_2 \leq \eta \max_{k \in \mathbb{N}(N_t)} \|\hat{r}(k)\|_2 \quad (6.3)$$

where $\eta \equiv \sqrt{N_s}||A^s||^{-1}$ denotes the stability constant.

**Proof.** Let us define the space–time residual as

$$r^s : v \rightarrow f^s + u_0^s - A^sv \quad (6.4)$$

with $r^s : \mathbb{R}^{N_s} \rightarrow \mathbb{R}^{N_s}$.

Then, we have

$$r^s(u^s) = f^s + u_0^s - A^su^s = 0 \quad (6.5)$$

$$r^s(\hat{u}^s) = f^s + u_0^s - A^s\hat{u}^s \quad (6.6)$$

where $u^s \in \mathbb{R}^{N_s}$ is the space–time FOM solution and $\hat{u}^s \in \mathbb{R}^{N_s}$ is the approximate space–time solution.

Subtracting Equation (6.6) from Equation (6.5) gives

$$r^s(\hat{u}^s) = A^se^s \quad (6.7)$$

where $e^s \equiv u^s - \hat{u}^s \in \mathbb{R}^{N_s}$. Inverting $A^s$ yields

$$e^s = (A^s)^{-1}r^s(\hat{u}^s). \quad (6.8)$$

Taking $l_2$ norm and Holders’ inequality gives

$$\|e^s\|_2 \leq ||(A^s)^{-1}||_2\|r^s(\hat{u}^s)\|_2 \quad (6.9)$$

We can re-write this in the following form

$$\sum_{k=1}^{N_t} \|e(k)\|_2^2 \leq \|((A^s)^{-1})\|_2\sum_{k=1}^{N_t} \|\hat{r}(k)\|_2^2 \quad (6.10)$$

Using the relations

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

and

$$\sum_{k=1}^{N_t} \|\hat{r}(k)\|_2^2 \leq N_t \max_{k \in \mathbb{N}(N_t)} \|\hat{r}(k)\|_2^2 \quad (6.12)$$

we have

$$\max_{k \in \mathbb{N}(N_t)} \|e(k)\|_2 \leq \sqrt{N_s}||A^s||^{-1}\|r^s(\hat{u}^s)\|_2 \quad (6.13)$$

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

A numerical demonstration with space–time system matrices, $A^s$ that have the same structure as the ones used in Section 7.1 and Section 7.2.1 shows the magnitude of $\|((A^s)^{-1})\|$ 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 $\sqrt{N_s}$, the stability constant $\eta$ grows exponentially as 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].
Stability constant, is assessed from its relative error by:

\[
\text{relative error} = \frac{\| \tilde{u}^n(\mu) - u^n(\mu) \|}{\| u^n(\mu) \|}
\]

and the \( \ell_2 \) norm of space–time residual:

\[
\| r^n(\tilde{u}^n(\mu)) \|_{\ell_2}.
\]

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 performed on an Intel(R) Core(TM) i9-10900T CPU @ 1.90GHz and DDR4 Memory @ 2933MHz.

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 \( \tilde{u}^n(\mu) \) is assessed from its relative error by:

\[
\text{relative error} = \frac{\| \tilde{u}^n(\mu) - u^n(\mu) \|}{\| u^n(\mu) \|}
\] (7.1)

The forward Euler time stepping scheme with uniform time step size, \( \Delta t = 10^{-2} \) is used.

7.1 2D linear diffusion equation

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

\[
\n(x, y) \in [0, 1] \times [0, 1], \quad t \in [0, 2], \quad (\mu_1, \mu_2) \in [-1.7, -0.2] \times [-1.7, -0.2].
\]

The boundary condition is

\[
\begin{align*}
\frac{\partial u}{\partial t} &= \left[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] - \frac{1}{\sin 2\pi} \frac{u}{\sqrt{(x - \mu_1)^2 + (y - \mu_2)^2}} \\
&\quad + \frac{u}{\sqrt{(x - \mu_1)^2 + (y - \mu_2)^2}}
\end{align*}
\]

(7.3)

where \( (x, y) \in [0, 1] \times [0, 1], \quad t \in [0, 2], \quad (\mu_1, \mu_2) \in [-1.7, -0.2] \times [-1.7, -0.2]. \)

The boundary condition is

\[
\begin{align*}
u(x, y, t) &= 0, \\
u(x = 0, y, t) &= 0, \\
u(x = 1, y, t) &= 0, \\
u(x, y = 0, t) &= 0, \\
u(x, y = 1, t) &= 0.
\end{align*}
\]

(7.4)

and the initial condition is

\[
\begin{align*}
u(x, y, t = 0) &= 0.
\end{align*}
\]

(7.5)

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 \( n_t \).

The Galerkin and Petrov–Galerkin space–time 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 relative errors of Petrov–Galerkin projection is larger than Petrov–Galerkin projection. This is because Petrov–Galerkin space–time ROM solution minimizes the space–time residual.

The backward Euler time stepping scheme with uniform time step size, \( \Delta t = 10^{-2} \) is employed, where we set \( N_s = 50. \) 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 238,650 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 error at the FOM 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 \( n_t \).

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 \( (N_s N_t)/(n_s n_t) \) of 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 \times 10^{-1} \) and 1.7646 \times 10^{-5} \) 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 6.0899 \times 10^{-1} \) and 1.6717 \times 10^{-5} \) seconds, respectively, resulting in speed-up of 376.04. For accuracy, the Galerkin method results in 1.210 \times 10^{-2} \% relative error and 1.249 \times 10^{-2} \space–\text{time residual norm while the Petrov–Galerkin results in 2.626 \times 10^{-2} \% relative error and 1.029 \times 10^{-2} \space–\text{time residual norm.} \)
Figure 3: 2D linear diffusion equation. Relative errors vs reduced dimensions.

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

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

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, \((\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = -1.7 + 1.5/14, i = 0, 1, \cdots, 14\} \times \{\mu_2 | \mu_2 = -1.7 + 1.5/14, j = 0, 1, \cdots, 14\}\). 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.

7.2 2D linear convection diffusion equation

7.2.1 Without source term

We consider a parameterized 2D linear convection diffusion equation

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

(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

\[
\begin{align*}
    u(x = 0, y, t) &= 0 \\
    u(x = 1, y, t) &= 0 \\
    u(x, y = 0, t) &= 0 \\
    u(x, y = 1, t) &= 0
\end{align*}
\]

(7.7)

The initial condition is given by

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

(7.8)

and shown in Fig. 8.
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.

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

\[
(0.03, 0.33), (0.03, 0.35), (0.05, 0.33), (0.05, 0.35)
\]

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 \( \times \) 10\(^{-1}\) and 1,3645 \( \times \) 10\(^{-2}\) 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 \( \times \) 10\(^{-1}\) and 1,5541 \( \times \) 10\(^{-3}\) seconds, respectively, resulting in speed-up of 370.74. For accuracy, the Galerkin method results in 4,898 \( \times \) 10\(^{-2}\) \% relative error and 1,503 space–time residual norm while the Petrov–Galerkin results in 5,878 \( \times \) 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.
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.
and the initial condition is given by
\[ u(x, y, t = 0) = 0. \]  
(7.12)

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 238 grid points, excluding boundary grid points. As a result, there are 248,650 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 \( (\mu_1, \mu_2) \). 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.

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 \( 50 \) free dimensions.

For the Galerkin method, the FOM and space–time ROM simulation with \( n_s = 19 \) and \( n_t = 3 \) takes an average time of 6.1290 × 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 3.8780 × 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 × 10⁻¹ space–time residual norm while the Petrov–Galerkin results in 2.652 × 10⁻¹ % relative error and 1.550 × 10⁻¹ 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, \( (\mu_1, \mu_2) \) 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, \( (\mu_1, \mu_2) \in \{0.195, 0.016 + 0.008, 11, i = 0, 1, \ldots, 11\} \times \{0.195, 0.205\} \). 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
The space–time ROM

(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.

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.

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 $O(10^{-3})$ to $O(10^{-4})$ relative errors with $O(10^2)$ 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-
The space–time ROM

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). Younghyu 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 Petrov–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

```python
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(); no_test=no_para*nparam1*nparam2

# 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

# construct Phi_s_e
Phi_e=PHIstT
print(Phi_e.T)
```

A.1.1 Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term

```python
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(); no_test=no_para*nparam1*nparam2

# 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_e
Phi_e=PHIstT
print(Phi_e.T)
```
The space–time ROM

62 # set model
63 A=A2D-sp.sparse.diags(1/np.sqrt((x1-param1)**2+(x2-param2)**2),format='csc')
64 f=(1/np.sqrt((x1-param1)**2+(x2-param2)**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
65
66 # construct Ast and fst using block structure (ust0 is zero)
67 As=PHIsT@A@PHIs; fs=PHIsT@f
68
69 Ast=np.zeros((ns*nt,ns*nt))
70 for i in range(nt):
71     for j in range(nt):
72         Astij=np.zeros((ns,ns))
73         for kk in range(Nt):
74             Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
75             Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
76             Astij+=Dik@Djk-k*Dik@As@Djk
77             if kk != Nt-1:
78                 Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
79                 Astij-=Dik_next@Djk
80             Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
81
82 fst=np.zeros(ns*nt)
83     for j in range(nt):
84         for kk in range(Nt):
85             Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
86             fst[ns*j:ns*(j+1)]+=k*Djk@fs[:,kk+1]
87
88 # Space–Time ROM (online phase)
89 UstROM=PHIst@np.linalg.solve(Ast,fst)
90
91 # FOM
92 u=np.zeros((Ns,Nt+1))
93     for n in range(Nt):
94         u[:,n+1]=spsolve(I-k*A,u[:,n]+k*f[:,n+1])
95     UstFOM=u[:,1:].flatten(order='F')
96
97 AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
98
99 rstROM=np.zeros(Ns*Nt) # ST residual
100     for n in range(1,Nt+1):
101         if n==1:
102             rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
103         else:
104             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])
105     STResROM_g2[test_count]=np.linalg.norm(rstROM)
106
107 # store solutions for each param
108 urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
109     urom[0]=np.zeros((N+1,N+1)); ufom[0]=np.zeros((N+1,N+1))
110     for n in range(1,Nt+1):
111         urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
112         ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
113
A.2 Petrov–Galerkin Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term

1 1 6 2
1 4 3
1 3 5
1 5 4
1 7 7
1 7 7
1 8 8
1 8 8
1 9 9
1 9 9
1 10 10
1 10 10
1 11 11
1 11 11
1 12 12
1 12 12
1 13 13
1 13 13
1 14 14
1 14 14
1 15 15
1 15 15
1 16 16
1 16 16
1 17 17
1 17 17
1 18 18
1 18 18
1 19 19
1 19 19
1 20 20
1 20 20
1 21 21
1 21 21
1 22 22
1 22 22
1 23 23
1 23 23
1 24 24
1 24 24
1 25 25
1 25 25
1 26 26
1 26 26
1 27 27
1 27 27
1 28 28
1 28 28
1 29 29
1 29 29
1 30 30
1 30 30
1 31 31
1 31 31
1 32 32
1 32 32
1 33 33
1 33 33
1 34 34
1 34 34
1 35 35
1 35 35
1 36 36
1 36 36
1 37 37
1 37 37
1 38 38
1 38 38
1 39 39
1 39 39
1 40 40
1 40 40
1 41 41
1 41 41
1 42 42
1 42 42
1 43 43
1 43 43
1 44 44
1 44 44
1 45 45
1 45 45
1 46 46
1 46 46
1 47 47
1 47 47
1 48 48
1 48 48
1 49 49
1 49 49
1 50 50
1 50 50
1 51 51
1 51 51
1 52 52
1 52 52
1 53 53
1 53 53
1 54 54
1 54 54
1 55 55
1 55 55
1 56 56
1 56 56
1 57 57
1 57 57
1 58 58
1 58 58
1 59 59
1 59 59
1 60 60
1 60 60
1 61 61
1 61 61
1 62 62
1 62 62
1 63 63
1 63 63
1 64 64
1 64 64
1 65 65
1 65 65
1 66 66
1 66 66
1 67 67
1 67 67
1 68 68
1 68 68
1 69 69
1 69 69
1 70 70
1 70 70
1 71 71
1 71 71
1 72 72
1 72 72
1 73 73
1 73 73
1 74 74
1 74 74
1 75 75
1 75 75
1 76 76
1 76 76
1 77 77
1 77 77
1 78 78
1 78 78
1 79 79
1 79 79
1 80 80
1 80 80
1 81 81
1 81 81
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')
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')

# 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])
    for n in range(1,Nt+1):
        urom[p,n-1]=u[:,n].flatten()

# 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 Djk
```python
# 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)
```

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

1. Import numpy as np, import scipy as sp
2. From scipy.sparse.linalg import spsolve
3. # train parameter space
4. mu1, mu2 = meshgrid(np.linspace(0.03, 0.05, 2), np.linspace(0.33, 0.35, 2), indexing='ij')
5. mu1 = mu1.flatten(); mu2 = mu2.flatten(); no_para = np.size(mu1, axis=0)
6. # space and time domain
7. N = 70; Ns = (N-1)**2; Nt = 50; h = 1./N; Tfinal = 1.; k = Tfinal/Nt
8. x1, x2 = meshgrid(np.linspace(0, 1, N+1)[1:-1], np.linspace(0, 1, N+1)[1:-1], indexing='ij')
9. # IC
10. 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()
11. # basic operators
12. e = np.ones(N-1); I = sp.eye(Ns, format='csc')
13. A1D_diff = 1/h**2 * sp.sparse.spdiags(np.vstack((e, -2*e, e)), [-1, 0, 1], N-1, N-1, format='csc')
14. A2D_diff = sp.sparse.kron(A1D_diff, sp.eye(N-1, format='csc'), format='csc') + sp.sparse.kron(sp.eye(N-1, format='csc'), A1D_diff, format='csc')
15. A1D_conv = 1/(h) * sp.sparse.spdiags(np.vstack((-1*e, 1*e, 0*e)), [-1, 0, 1], N-1, N-1, format='csc')
16. A2D_conv = sp.sparse.kron(A1D_conv, sp.eye(N-1, format='csc'), format='csc') + sp.sparse.kron(sp.eye(N-1, format='csc'), A1D_conv, format='csc')
17. # snapshot, free DoFs, No IC included
18. U = np.zeros((Ns, no_para*Nt))
19. for p in range(no_para):
20.    A = -mu1[p]*A2D_conv + mu2[p]*A2D_diff
21.    u = np.zeros((Ns, Nt+1)); u[:, 0] = u0
22.    for n in range(Nt):
23.        u[:, n+1] = spsolve(I - k*A, u[:, n])
24.    U[:, np.arange(p*Nt, (p+1)*Nt)] = u[:, 1:]
25. # POD of solution snapshot
26. W, S, VT = linalg.svd(U)
27. # test parameter space
28. nparam1 = 12; nparam2 = 12; ParamD1, ParamD2 = meshgrid(np.linspace(0.01, 0.07, nparam1), np.linspace(0.31, 0.37, nparam2))
29. ParamD1 = ParamD1.flatten(); ParamD2 = ParamD2.flatten(); num_test = np.prod(ParamD1.shape)
30. # variables to store (also store ParamD1 and ParamD2)
# A.5 Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term

```python
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])

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]=U[:,n-1].reshape((N-1,N-1))
    UFOM[n,1:-1,1:-1]=U[:,n].reshape((N-1,N-1))
```

The space–time ROM

A.5 Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term
[code]
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')) +
          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))

def writeModel():
    sp.savez('model.npz', A1D_diff, A1D_conv, A2D_diff, A2D_conv, f)

for mu in range(no_para):
    print('mu:', mu)
    writeModel()

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')) +
          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))
    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
    Astij += Dik@ATIfs[:,kk+1]
    if kk != Nt-1:
      fs_next = fs[:,kk+1+1]
      Astij += Dik@fs_next
    Astij[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
  ftemp=zero(nz)
  for j in range(nt):
    Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
    ftemp[j*temp.size]=Djk@ATIfs[:,kk]
  for kk in range(Nt):
    Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
    ftemp[ns*i:ns*(i+1)]=Dik@ATIs@Djk
  Astij[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
  UstROM=PHIst@np.linalg.solve(Ast,fst)
  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 ocean 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] Zhaojun Bai. Krylov subspace techniques for reduced-order modeling of large-scale dynamical systems. Applied numerical mathematics, 43(1-2):9–44, 2002.

[11] Serkan Gugercin, Athanasios C Antoulas, and Christopher Beattie. H2 model reduction for large-scale linear dynamical systems. SIAM journal on matrix analysis and applications, 30(2):609–638, 2008.

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

[13] Eli Chizhov and Michael Nakha. Generalized moment-matching methods for transient analysis of interconnected networks. In [1995] Proceedings 29th ACM/IEEE Design Automation Conference, pages 201–206. IEEE, 1992.

[14] 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.

[15] Amine Ammar, Béchir Mokdad, Francisco Chinesta, and Roland Kernting. 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, 144(2-3):98–121, 2007.

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

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

[18] 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.

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

[20] Eugene Giner, Brice Bognet, Juan J Ródenas, Adrien Leygue, F Javier Fuenmayor, and Francisco Chinesta. 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.

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

[22] 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.

[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 applications to aerelasticity. AIAA journal, 46(7):1803–1813, 2008.

[25] Jeffrey P Thomas, Earl H Dowell, and Kenneth C Hall. Three-dimensional transonic aerelasticity 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 Elías Cueto. A short review on model order reduction based on proper generalized decomposition for linear and nonlinear applications. SIAM Journal on Scientific Computing, 30(2):609–638, 2008.

[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.
[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.