Article

Efficient Space–Time Reduced Order Model for Linear Dynamical Systems in Python Using Less than 120 Lines of Code

Youngkyu Kim ^{1,*}, Karen Wang ^{2,†} and Youngsoo Choi ³ ¹ Mechanical Engineering, University of California, Berkeley, CA 94720, USA² Lawrence Livermore National Laboratory, Livermore, CA 94550, USA; wang79@llnl.gov³ Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA; choi15@llnl.gov

* Correspondence: youngkyu_kim@berkeley.edu

† These authors contributed equally to this work.

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 [Choi et al., Space–time reduced order model for large-scale linear dynamical systems with application to Boltzmann transport problems, *Journal of Computational Physics*, 2020], 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 least-squares Petrov–Galerkin (LSPG) 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 LSPG method with the traditional Galerkin method and show that the space–time ROMs can achieve $\mathcal{O}(10^{-3})$ to $\mathcal{O}(10^{-4})$ relative errors for these problems. Depending on parameter–separability, online speed-ups may or may not be achieved. For the FOMs with parameter–separability, the space–time ROMs can achieve $\mathcal{O}(10)$ online speed-ups. Finally, we present an error analysis for the space–time LSPG projection and derive an error bound, which shows an improvement compared to traditional spatial Galerkin ROM methods.

Citation: Kim, Y.; Wang, K.; Choi, Y. Efficient Space–Time Reduced Order Model for Linear Dynamical Systems in Python Using Less than 120 Lines of Code. *Mathematics* **2021**, *9*, 1690. <https://doi.org/10.3390/math9141690>

Received: 17 May 2021

Accepted: 9 July 2021

Published: 19 July 2021

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

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

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 control 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 [1–9], moment-matching methods [10–14], and Proper Generalized Decomposition (PGD) [15] and its extensions [16–25]. 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 computationally expensive task in large-scale problems using dense solvers. However, the fast and memory-efficient low rank solvers presented in [26,27] can be applied to the large-scale Lyapunov equations. On the other hand, this expensive task can be bypassed by various data-driven empirical and non-intrusive approaches [3,28–31]. Moment matching methods were originally developed as non data-driven, although later studies developed ideas to benefit from the data. 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 [11] 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 [32]. 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 [33,34], 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 [35]. The method has been actively developed and extended to many applications [36–43]. For a more detailed description about DMD, we refer to this preprint [44] and book [45]. POD utilizes the method of snapshots to obtain an optimal basis of a system and typically applies only to spatial projections [46–61].

Although the data-driven ROMs have brought many benefits, such as speed-ups and accuracy, they have been mainly the reduction of the complexity on spatial domains. However, reducing the temporal domain complexity brings even more speed-ups for time dependent problems. Therefore, we focus on building a space–time ROM, where both spatial and temporal projections are applied to achieve an optimal reduction. Several space–time ROMs have been developed, including, but not limited to, the ones derived from the corresponding space–time full order models [62–65], the spectral POD and resolvent modes-based ones [66–68], and the ones that take advantage of the right singular vectors [69–71]. In particular, a space–time ROM for large-scale linear dynamical systems has been recently introduced [72]. The authors show a speed-up of greater than 8000 with good accuracy for a large-scale transport problem. However, the space–time ROM in [72] was limited to Galerkin projection, which can cause stability issues [22,73,74] for nonlinear problems. It is well-studied that the LSPG ROM formulation [75] brings various benefits, such as better stability, over Galerkin projection with some known issues, such as structure preservation [76]. Furthermore, it is not clear if the space–time least-squares Petrov–Galerkin (LSPG) brings any advantage over the space–time Galerkin approach for a linear dynamical system. The current manuscript addresses this aspect.

In this work, we present several contributions on the space–time ROM development for linear dynamical systems:

- We derive the block structures of least-squares Petrov–Galerkin (LSPG) space–time ROM operators for linear dynamical systems for the first time and compare them with the Galerkin space–time ROM operators.
- We present an error analysis for LSPG space–time ROMs for the first time and demonstrate the growth rate of the stability constant with the actual space–time operators used in our numerical results.

- We compare the performance between space–time Galerkin and space–time LSPG reduced order models on several linear dynamical systems.
- For each numerical 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.

We note that the first two bullet points above are minor but useful contributions by making the block structure and error analysis readily available. Additionally, 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 [77–83], uncertainty quantification [84–86], and optimal control problems [87–89].

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 LSPG 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 [72] in Section 5.2. In Section 5.3, we derive LSPG space–time ROM operators in terms of the blocks. Then, we compared Galerkin and LSPG block structures in Section 5.4. We show 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 LSPG space–time ROMs in two numerical experiments in Section 7. Finally, the paper is concluded with summary and future works in Section 8. 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 (1):

$$\frac{\partial \mathbf{u}(t; \boldsymbol{\mu})}{\partial t} = \mathbf{A}(\boldsymbol{\mu})\mathbf{u}(t; \boldsymbol{\mu}) + \mathbf{B}(\boldsymbol{\mu})\mathbf{f}(t; \boldsymbol{\mu}), \quad \mathbf{u}(0; \boldsymbol{\mu}) = \mathbf{u}^0(\boldsymbol{\mu}), \quad (1)$$

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

$$\left(\mathbf{I}_{N_s} - \Delta t^{(k)} \mathbf{A}(\boldsymbol{\mu}) \right) \mathbf{u}^{(k)} = \mathbf{u}^{(k-1)} + \Delta t^{(k)} \mathbf{B}(\boldsymbol{\mu}) \mathbf{f}^{(k)}(\boldsymbol{\mu}), \quad (2)$$

where $\mathbf{I}_{N_s} \in \mathbb{R}^{N_s \times N_s}$ is the identity matrix, $\Delta t^{(k)}$ is the k th time step size with $T = \sum_{k=1}^{N_t} \Delta t^{(k)}$ and $t_{(k)} = \sum_{j=1}^k \Delta t^{(j)}$, and $\mathbf{u}^{(k)}(\boldsymbol{\mu}) := \mathbf{u}(t_{(k)}; \boldsymbol{\mu})$ and $\mathbf{f}^{(k)}(\boldsymbol{\mu}) := \mathbf{f}(t_{(k)}; \boldsymbol{\mu})$ are the state and input vectors at the k th time step where $k \in \mathbb{N}(N_t)$. The Full Order Model (FOM) solves Equation (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 (3). This is known as the space–time formulation.

$$\mathbf{A}^{\text{st}}(\boldsymbol{\mu}) \mathbf{u}^{\text{st}}(\boldsymbol{\mu}) = \mathbf{f}^{\text{st}}(\boldsymbol{\mu}) + \mathbf{u}_0^{\text{st}}(\boldsymbol{\mu}), \quad (3)$$

where

$$A^{\text{st}}(\mu) = \begin{bmatrix} I_{N_s} - \Delta t^{(1)} A(\mu) & & & & \\ & -I_{N_s} & I_{N_s} - \Delta t^{(2)} A(\mu) & & \\ & & \ddots & \ddots & \\ & & & -I_{N_s} & I_{N_s} - \Delta t^{(N_t)} A(\mu) \end{bmatrix}, \quad (4)$$

$$u^{\text{st}}(\mu) = \begin{bmatrix} u^{(1)}(\mu) \\ u^{(2)}(\mu) \\ \vdots \\ u^{(N_t)}(\mu) \end{bmatrix}, \quad (5)$$

$$f^{\text{st}}(\mu) = \begin{bmatrix} \Delta t^{(1)} B(\mu) f^{(1)}(\mu) \\ \Delta t^{(2)} B(\mu) f^{(2)}(\mu) \\ \vdots \\ \Delta t^{(N_t)} B(\mu) f^{(N_t)}(\mu) \end{bmatrix}, \quad (6)$$

$$u_0^{\text{st}}(\mu) = \begin{bmatrix} u^0(\mu) \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}. \quad (7)$$

The space–time system matrix A^{st} has dimensions $\mathbb{R}^{N_\mu} \rightarrow \mathbb{R}^{N_s N_t \times N_s N_t}$, the space–time state vector u^{st} has dimensions $\mathbb{R}^{N_\mu} \rightarrow \mathbb{R}^{N_s N_t}$, the space–time input vector f^{st} has dimensions $\mathbb{R}^{N_\mu} \rightarrow \mathbb{R}^{N_s N_t}$, and the space–time initial vector u_0^{st} has dimensions $\mathbb{R}^{N_\mu} \rightarrow \mathbb{R}^{N_s N_t}$. 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 LSPG ROM formulations.

3. Space–Time Reduced Order Models

We investigate two projection-based space–time ROM formulations: the Galerkin and LSPG formulations.

3.1. Linear Subspace Solution Representation

Both the Galerkin and LSPG 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:

$$u^{\text{st}}(\mu) \approx \hat{u}^{\text{st}}(\mu) \equiv \Phi_{\text{st}} \hat{u}^{\text{st}}(\mu), \quad (8)$$

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

$$\Phi_{\text{st}} \equiv \begin{bmatrix} \phi_1^{\text{st}} & \cdots & \phi_{i+n_s(j-1)}^{\text{st}} & \cdots & \phi_{n_s n_t}^{\text{st}} \end{bmatrix}, \quad (9)$$

where $i \in \mathbb{N}(n_s), j \in \mathbb{N}(n_t)$. Substituting Equation (8) into the space–time formulation in Equation (3) gives an over-determined system of equations:

$$A^{\text{st}}(\mu) \Phi_{\text{st}} \hat{u}^{\text{st}}(\mu) = f^{\text{st}}(\mu) + u_0^{\text{st}}(\mu) \quad (10)$$

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

3.2. Galerkin Projection

In the Galerkin formulation, Equation (10) is closed by the Galerkin projection, where both sides of the equation of multiplied by Φ_{st}^T . Thus, we solve following reduced system for the unknown generalized coordinates, $\hat{u}^{st}$:

$$\Phi_{st}^T A^{st}(\mu) \Phi_{st} \hat{u}^{st}(\mu) = \Phi_{st}^T f^{st}(\mu) + \Phi_{st}^T u_0^{st}(\mu). \quad (11)$$

For notational simplicity, let us define the reduced space–time system matrix as $\hat{A}^{st,g}(\mu) := \Phi_{st}^T A^{st}(\mu) \Phi_{st}$, reduced space–time input vector as $\hat{f}^{st,g}(\mu) := \Phi_{st}^T f^{st}(\mu)$, and reduced space–time initial state vector as $\hat{u}_0^{st,g}(\mu) := \Phi_{st}^T u_0^{st}(\mu)$.

3.3. Least-Squares Petrov–Galerkin (LSPG) Projection

In the LSPG formulation, we first define the space–time residual as

$$\tilde{r}^{st}(\hat{u}^{st}; \mu) := f^{st}(\mu) + u_0^{st}(\mu) - A^{st}(\mu) \Phi_{st} \hat{u}^{st} \quad (12)$$

where $\tilde{r}^{st} : \mathbb{R}^{n_s n_t} \times \mathbb{R}^{N_\mu} \rightarrow \mathbb{R}^{N_s N_t}$. Note Equation (12) is an over-determined system. To close the system and solve for the unknown generalized coordinates, $\hat{u}^{st}$, the LSPG method takes the squared norm of the residual vector function and minimize it:

$$\hat{u}^{st} = \underset{\hat{v} \in \mathbb{R}^{n_s n_t}}{\operatorname{argmin}} \quad \frac{1}{2} \|\tilde{r}^{st}(\hat{v}; \mu)\|_2^2. \quad (13)$$

The solution to Equation (13) satisfies

$$(A^{st}(\mu) \Phi_{st})^T \tilde{r}^{st}(\hat{u}^{st}; \mu) = 0 \quad (14)$$

leading to

$$\Phi_{st}^T A^{st}(\mu)^T A^{st}(\mu) \Phi_{st} \hat{u}^{st}(\mu) = \Phi_{st}^T A^{st}(\mu)^T f^{st}(\mu) + \Phi_{st}^T A^{st}(\mu)^T u_0^{st}(\mu). \quad (15)$$

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

$$\hat{A}^{st,pg}(\mu) := \Phi_{st}^T A^{st}(\mu)^T A^{st}(\mu) \Phi_{st},$$

reduced space–time input vector as $\hat{f}^{st,pg}(\mu) := \Phi_{st}^T A^{st}(\mu)^T f^{st}(\mu)$, and reduced space–time initial state vector as $\hat{u}_0^{st,pg}(\mu) := \Phi_{st}^T A^{st}(\mu)^T u_0^{st}(\mu)$.

3.4. Comparison of Galerkin and LSPG Projections

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

Table 1. Comparison of Galerkin and LSPG projections.

Galerkin	LSPG
$\hat{A}^{st,g}(\mu) = \Phi_{st}^T A^{st}(\mu) \Phi_{st}$	$\hat{A}^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T A^{st}(\mu) \Phi_{st}$
$\hat{f}^{st,g}(\mu) = \Phi_{st}^T f^{st}(\mu)$	$\hat{f}^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T f^{st}(\mu)$
$\hat{u}_0^{st,g}(\mu) = \Phi_{st}^T u_0^{st}(\mu)$	$\hat{u}_0^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T u_0^{st}(\mu)$

4. Space-Time Basis Generation

In this section, we repeat Section 4.1 in [72] to be self-contained.

A data set for space–time basis generation is chosen by the method of snapshots introduced by Sirovich [90]. This data set is called a snapshot matrix. By solving full order

model problems at each of parameters in a parameter set $\{\mu_1, \dots, \mu_{n_\mu}\}$, we obtain a full order model solution matrix $\mathbf{U}_p$ at a certain parameter $\mu_p, p \in \mathbb{N}(n_\mu)$, i.e.,

$$\mathbf{U}_p \equiv [\mathbf{u}^{(1)}(\mu_p) \quad \dots \quad \mathbf{u}^{(k)}(\mu_p) \quad \dots \quad \mathbf{u}^{(N_t)}(\mu_p)], \quad (16)$$

where $\mathbf{u}^{(k)}(\cdot) \in \mathbb{R}^{N_s}$ is a solution at a time step $k \in \mathbb{N}(N_t)$. Concatenating all the solution matrices for the every parameter yields the snapshot matrix, which is given by

$$\mathbf{U} \equiv [\mathbf{U}_1 \quad \dots \quad \mathbf{U}_{n_\mu}] \in \mathbb{R}^{N_s \times n_\mu N_t}. \quad (17)$$

To generate the spatial basis Φ_s , Proper Orthogonal Decomposition (POD) is used [49]. The POD procedure seeks the reduced dimensional subspace that optimally represents the solution snapshot, $\mathbf{U}$. The spatial POD basis Φ_s is obtained by solving the following minimization problem:

$$\Phi_s = \underset{\Phi \in \mathbb{R}^{N_s \times n_s}}{\operatorname{argmin}} \left\| \mathbf{U} - \Phi \Phi^T \mathbf{U} \right\|_F^2 \quad \text{s.t.} \quad \Phi^T \Phi = \mathbf{I}_{n_s}, \quad (18)$$

where $\|\cdot\|_F$ denotes the Frobenius norm and $\mathbf{I}_{n_s} \in \mathbb{R}^{n_s \times n_s}$ is the identity matrix with $n_s < n_\mu N_t$. Then, the solution to the minimization problem is obtained by choosing the leading n_s columns of the left singular matrix of Singular Value Decomposition (SVD) of the snapshot matrix $\mathbf{U}$:

$$\mathbf{U} = \mathbf{W} \Sigma \mathbf{V}^T \quad (19)$$

where $\mathbf{W} \in \mathbb{R}^{N_s \times \ell}$ and $\mathbf{V} \in \mathbb{R}^{n_\mu N_t \times \ell}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{\ell \times \ell}$ is a diagonal matrix with singular values on its diagonal with $\ell \equiv \min(N_s, n_\mu N_t)$.

Equation (19) can be written in the summation form

$$\mathbf{U} = \sum_{i=1}^{\ell} \sigma_i \mathbf{w}_i \mathbf{v}_i^T, \quad (20)$$

where $\sigma_i \in \mathbb{R}$ is the i th singular value, $\mathbf{w}_i$ and $\mathbf{v}_i$ are the i th left and right singular vectors, respectively. Here, n_μ different temporal behavior of $\mathbf{w}_i$ are represented by $\mathbf{v}_i$, e.g., $\mathbf{v}_i^1, \mathbf{v}_i^2$, and $\mathbf{v}_i^3$ describe three different temporal behavior of a certain basis vector $\mathbf{w}_i$ in Figure 1. Now, let us set $\mathbf{Y}_i = [\mathbf{v}_i^1 \quad \dots \quad \mathbf{v}_i^{n_\mu}]$ to be a temporal snapshot matrix associated with the i th spatial basis vector $\mathbf{w}_i$, where $\mathbf{v}_i^k \equiv [\mathbf{v}_i(1 + (k-1)N_t), \mathbf{v}_i(2 + (k-1)N_t), \dots, \mathbf{v}_i(kN_t)]^T \in \mathbb{R}^{N_t}$ for $k \in \mathbb{N}(n_\mu)$, where $\mathbf{v}_i(j), j \in \mathbb{N}(n_\mu N_t)$ is the j th component of the vector $\mathbf{v}_i$. Then, applying SVD on $\mathbf{Y}_i$ gives us

$$\mathbf{Y}_i = \Lambda_i \Sigma_i \Psi_i^T. \quad (21)$$

Leading n_t vectors of the left singular matrix Λ_i become the temporal basis Φ_t^i which describes a temporal behavior of the i th column of spatial basis Φ_s . Finally, we construct a space-time basis vector $\phi_{i+n_s(j-1)}^{\text{st}} \in \mathbb{R}^{N_s N_t}$ given in Equation (9) as

$$\phi_{i+n_s(j-1)}^{\text{st}} = \phi_{ij}^t \otimes \phi_i^s, \quad (22)$$

where $\otimes$ denotes Kronecker product, $\phi_i^s \in \mathbb{R}^{N_s}$ is the i th column of the spatial basis Φ_s and $\phi_{ij}^t \in \mathbb{R}^{N_t}$ is the j th column of the temporal basis Φ_t^i .

Figure 1. Illustration of spatial and temporal bases construction, using SVD with $n_\mu = 3$. The right singular vector, v_i , describes three different temporal behaviors of a left singular basis vector w_i , i.e., three different temporal behaviors of a spatial mode due to three different parameters that are denoted as μ_1, μ_2 , and μ_3 . Each temporal behavior is denoted as v_i^1, v_i^2 , and v_i^3 .

5. Space-Time Reduced Order Models in Block Structure

We avoid building the space–time basis vector defined in Equation (22) because it requires a lot of memory for storage. Instead, 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 LSPG projection. First of all, we introduce common block structures that appear both in the Galerkin and LSPG projections.

5.1. Block Structures of Space–Time Basis

Following [72]’s notation, we define the block structure of the space–time basis to be:

$$\Phi_{st} = \begin{pmatrix} \Phi_s D_1^1 & \cdots & \cdots & \cdots & \Phi_s D_1^{n_t} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \cdots & \Phi_s D_k^j & \cdots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \Phi_s D_{N_t}^1 & \cdots & \cdots & \cdots & \Phi_s D_{N_t}^{n_t} \end{pmatrix} \in \mathbb{R}^{N_s N_t \times n_s n_t} \quad (23)$$

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

$$D_k^j = \begin{pmatrix} \phi_{1j,k}^t & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & 0 & \cdots & \vdots \\ \vdots & 0 & \phi_{ij,k}^t & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \phi_{n_s j,k}^t \end{pmatrix} \in \mathbb{R}^{n_s \times n_s} \quad (24)$$

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

5.2. Block Structures of Galerkin Projection

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

$$\hat{A}^{st,g}(\mu) = \Phi_{st}^T A^{st}(\mu) \Phi_{st} \quad (25)$$

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

$$\hat{A}^{st,g}(\mu) = \begin{pmatrix} \hat{A}_{(1,1)}^{st,g}(\mu) & \cdots & \cdots & \cdots & \hat{A}_{(1,n_t)}^{st,g}(\mu) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \cdots & \hat{A}_{(j',j)}^{st,g}(\mu) & \cdots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{A}_{(n_t,1)}^{st,g}(\mu) & \cdots & \cdots & \cdots & \hat{A}_{(n_t,n_t)}^{st,g}(\mu) \end{pmatrix} \quad (26)$$

so that we can exploit the block structure of these matrices such that we do not need to form the entire matrix. By inserting Equations (4) and (23) to Equation (25), we derive that the (j', j) th block matrix $\hat{A}_{(j',j)}^{st,g}(\mu) \in \mathbb{R}^{n_s \times n_s}$ is:

$$\hat{A}_{(j',j)}^{st,g}(\mu) = \sum_{k=1}^{N_t} \left(D_k^{j'} D_k^j - \Delta t^{(k)} D_k^{j'} \Phi_s^T A(\mu) \Phi_s D_k^j \right) - \sum_{k=1}^{N_t-1} D_{k+1}^{j'} D_k^j. \quad (27)$$

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

$$\hat{f}^{st,g}(\mu) = \Phi_{st}^T f^{st}(\mu). \quad (28)$$

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

$$\hat{f}^{st,g}(\mu)_{(j)} = \sum_{k=1}^{N_t} \Delta t^{(k)} D_k^j \Phi_s^T B(\mu) f^{(k)}(\mu). \quad (29)$$

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

$$\hat{u}_0^{st,g}(\mu) = \Phi_{st}^T u_0^{st}(\mu) \quad (30)$$

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

$$\hat{u}_0^{st,g}(\mu)_{(j)} = D_1^j \Phi_s^T u_0(\mu). \quad (31)$$

5.3. Block Structures of LSPG Projection

In Section 3.3, the reduced space–time LSPG system matrix, $\hat{A}^{st,pg}(\mu)$ is defined as

$$\hat{A}^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T A^{st}(\mu) \Phi_{st}. \quad (32)$$

Then, the block structure of this matrix can be defined as

$$\hat{A}^{st,pg}(\mu) = \begin{pmatrix} \hat{A}_{(1,1)}^{st,pg}(\mu) & \cdots & \cdots & \cdots & \hat{A}_{(1,n_t)}^{st,pg}(\mu) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \cdots & \hat{A}_{(j',j)}^{st,pg}(\mu) & \cdots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{A}_{(n_t,1)}^{st,pg}(\mu) & \cdots & \cdots & \cdots & \hat{A}_{(n_t,n_t)}^{st,pg}(\mu) \end{pmatrix}, \quad (33)$$

where the size of each block matrix is $n_s \times n_s$. Instead of forming the $\hat{A}^{st,pg}(\mu)$ directly from Equation (32), each block matrix $\hat{A}_{(j',j)}^{st,pg}(\mu) \in \mathbb{R}^{n_s \times n_s}$ is computed and put into its location.

Inserting Equations (4) and (23) to Equation (32) gives us $\hat{A}^{st,pg}(\mu)$ and the (j', j) th block of this matrix is derived as

$$\begin{aligned}\hat{A}_{(j',j)}^{st,pg}(\mu) = & \sum_{k=1}^{N_t} \left[D_k^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \left(I_{N_s} - \Delta t^{(k)} A(\mu) \right) \Phi_s D_k^j \right] \\ & + \sum_{k=1}^{N_t-1} \left[D_k^{j'} D_k^j - D_k^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k+1)} A(\mu) \right) \Phi_s D_{k+1}^j \right. \\ & \left. - D_{k+1}^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k+1)} A(\mu)^T \right) \Phi_s D_k^j \right].\end{aligned}\quad (34)$$

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

$$\hat{f}^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T f^{st}(\mu), \quad (35)$$

where the j th block vector $\hat{f}^{st,pg}(\mu)_{(j)} \in \mathbb{R}^{n_t}$ is computed as

$$\begin{aligned}\hat{f}^{st,pg}(\mu)_{(j)} = & \sum_{k=1}^{N_t} \left[D_k^j \Phi_s^T \left(I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \Delta t^{(k)} B(\mu) f^{(k)}(\mu) \right] \\ & - \sum_{k=1}^{N_t-1} \left[D_k^j \Phi_s^T \Delta t^{(k+1)} B(\mu) f^{(k+1)}(\mu) \right].\end{aligned}\quad (36)$$

Lastly, the space–time LSPG initial vector $\hat{u}_0^{st,pg}(\mu) \in \mathbb{R}^{n_s n_t}$ is

$$\hat{u}_0^{st,pg}(\mu) = \Phi_{st}^T A^{st}(\mu)^T u_0^{st}(\mu). \quad (37)$$

By exploiting the block structure of the vector, we compute the j th block vector $\hat{u}_0^{st,pg}(\mu)_{(j)} \in \mathbb{R}^{n_s}$ to be:

$$\hat{u}_0^{st,pg}(\mu)_{(j)} = D_1^j \Phi_s^T \left(I_{N_s} - \Delta t^{(1)} A(\mu)^T \right) u_0(\mu). \quad (38)$$

5.4. Comparison of Galerkin and LSPG Block Structures

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

Table 2. Comparison of Galerkin and LSPG block structures.

Galerkin	LSPG
$\hat{A}^{st,g}(\mu)_{(j',j)} = \sum_{k=1}^{N_t} \left(D_k^{j'} D_k^j - \Delta t^{(k)} D_k^{j'} \Phi_s^T A(\mu) \Phi_s D_k^j \right) - \sum_{k=1}^{N_t-1} D_{k+1}^{j'} D_k^j$	$\hat{A}^{st,pg}(\mu)_{(j',j)} = \sum_{k=1}^{N_t} \left[D_k^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \left(I_{N_s} - \Delta t^{(k)} A(\mu) \right) \Phi_s D_k^j \right] + \sum_{k=1}^{N_t-1} \left[D_k^{j'} D_k^j - D_k^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k+1)} A(\mu) \right) \Phi_s D_{k+1}^j - D_{k+1}^{j'} \Phi_s^T \left(I_{N_s} - \Delta t^{(k+1)} A(\mu)^T \right) \Phi_s D_k^j \right]$
$\hat{f}^{st,g}(\mu)_{(j)} = \sum_{k=1}^{N_t} D_k^j \Delta t^{(k)} \Phi_s^T B(\mu) f^{(k)}(\mu)$	$\hat{f}^{st,pg}(\mu)_{(j)} = \sum_{k=1}^{N_t} \left[D_k^j \Phi_s^T \left(I_{N_s} - \Delta t^{(k)} A(\mu)^T \right) \Delta t^{(k)} B(\mu) f^{(k)}(\mu) \right] - \sum_{k=1}^{N_t-1} \left[D_k^j \Phi_s^T \Delta t^{(k+1)} B(\mu) f^{(k+1)}(\mu) \right]$
$\hat{u}_0^{st,g}(\mu)_{(j)} = D_1^j \Phi_s^T u_0(\mu)$	$\hat{u}_0^{st,pg}(\mu)_{(j)} = D_1^j \Phi_s^T \left(I_{N_s} - \Delta t^{(1)} A(\mu)^T \right) u_0(\mu)$

5.5. Computational Complexity of Forming Space–Time ROM Operators

To show computational complexity of forming the reduced space–time system matrices $\hat{A}^{st,g}(\mu)$ and $\hat{A}^{st,pg}(\mu)$, input vectors $\hat{f}^{st,g}(\mu)$ and $\hat{f}^{st,pg}(\mu)$, and initial state vectors $\hat{u}_0^{st,g}(\mu)$ and $\hat{u}_0^{st,pg}(\mu)$ for Galerkin and LSPG 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 an identity matrix of size N_s in Equation (1). The band structure of $A(\mu)$ is often seen in mathematical models because of local approximations to derivative terms. The structure of $A^{st}(\mu) \in \mathbb{R}^{N_s N_t \times N_s N_t}$ that is formed with backward Euler scheme is lower triangular matrix with bandwidth N_s . We also assume that the spatial basis vectors $\phi_i^s \in \mathbb{R}^{N_s}$, $i \in \mathbb{N}(n_s)$ and temporal basis vectors $\phi_{ij}^t \in \mathbb{R}^{N_t}$, $i \in \mathbb{N}(n_s)$ and $j \in \mathbb{N}(n_t)$ are given.

Let us start to show the computational complexity without use of block structures as presented in Table 1. Space–time basis Φ_{st} is constructed by Equation (22), and it involves $n_s \times n_t$ times Kronecker product of vector of size N_s and vector of size N_t . Thus, it costs $\mathcal{O}(N_s N_t n_s n_t)$. For Galerkin projection, the reduced space–time system matrix $\hat{A}^{st,g}(\mu)$ involves matrix multiplication twice. The first operation is matrix multiplication of $n_s n_t \times N_s N_t$ matrix and $N_s N_t \times N_s N_t$ lower triangular matrix with bandwidth N_s . Then, the second one is matrix multiplication of $n_s n_t \times N_s N_t$ matrix and $N_s N_t \times n_s n_t$ matrix. Thus, the cost of computing the reduce space–time system matrix, $\hat{A}^{st,g}(\mu)$ is $\mathcal{O}(N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$. The input vector $\hat{f}^{st,g}(\mu)$ requires $n_s n_t \times N_s N_t$ matrix and $N_s N_t \times 1$ vector multiplication, resulting in $\mathcal{O}(N_s N_t n_s n_t)$. The initial state vector $\hat{u}_0^{st,g}(\mu)$ requires matrix Φ_{st}^T and vector u_0^{st} multiplication. Considering that the u_0^{st} contains zero values except the first N_s components, the computational cost is $\mathcal{O}(N_s n_s n_t)$. Keeping the dominant terms leads to $\mathcal{O}(N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$. For LSPG projections, we first compute $\Phi_{st}^T A^{st}(\mu)$ which is going to be re-used, resulting in $\mathcal{O}(N_s^2 N_t n_s n_t)$. Then, computing the reduced space–time system matrix $\hat{A}^{st,pg}(\mu)$ requires multiplication of $n_s n_t \times N_s N_t$ precomputed matrix and its transpose matrix, resulting in $\mathcal{O}(N_s N_t (n_s n_t)^2)$. The input vector $\hat{f}^{st,pg}(\mu)$ is computed by multiplication between $n_s n_t \times N_s N_t$ precomputed matrix and $N_s N_t \times 1$ vector with $\mathcal{O}(N_s N_t n_s n_t)$ operation counts. Constructing initial state vector $\hat{u}_0^{st,pg}(\mu)$ costs $\mathcal{O}(N_s n_s n_t)$ because of $n_s n_t \times N_s N_t$ precomputed matrix and $N_s N_t \times 1$ vector whose components except first N_s components are zero. Summing operation counts and keeping the dominant terms gives us $\mathcal{O}(N_s^2 N_t n_s n_t) + \mathcal{O}(N_s N_t (n_s n_t)^2)$. With the assumption of $n_s n_t \ll N_s$, the computational cost without use of block structures for both Galerkin and LSPG projections is $\mathcal{O}(N_s^2 N_t n_s n_t)$.

Now, let us show the computational complexity with the use of block structures. For Galerkin projection, we first compute $\Phi_s^T A(\mu) \Phi_s$ 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. Each block requires $2N_t$ times multiplication between diagonal matrices of size n_s and N_t times multiplication by diagonal matrices of size n_s on the left and right of the precomputed $n_s \times n_s$ matrix, which takes $\mathcal{O}(N_t(2n_s^2 + 2n_s))$. Thus, it costs $\mathcal{O}(N_t n_t^2(2n_s^2 + 2n_s)) + \mathcal{O}(2bN_s n_s) + \mathcal{O}(N_s n_s^2)$ to compute the reduced space–time system matrix $\hat{A}^{st,g}(\mu)$. For the reduced space–time input vector $\hat{f}^{st,g}(\mu)$, n_t blocks are needed. Each block is computed by $n_s \times N_s$ matrix and $N_s \times 1$ vector multiplication followed by the diagonal matrix of size n_s and $n_s \times 1$ vector multiplication, resulting in $\mathcal{O}(N_t(N_s n_s + n_s))$. Thus, the cost of computing the reduced space–time input vector $\hat{f}^{st,g}(\mu)$ is $\mathcal{O}(N_t n_t(N_s n_s + n_s))$. Computing the reduced space–time initial vector $\hat{u}_0^{st,g}(\mu)$ requires $n_s \times N_s$ matrix and $N_s \times 1$ vector multiplication followed by diagonal matrix of size n_s and $n_s \times 1$ vector multiplication. Its cost is $\mathcal{O}(N_s n_s + n_s)$. Summing each computational cost for Galerkin projection gives us $\mathcal{O}(N_t n_t^2(2n_s^2 + 2n_s)) + \mathcal{O}(2bN_s n_s) + \mathcal{O}(N_s n_s^2) + \mathcal{O}(N_t n_t(N_s n_s + n_s)) + \mathcal{O}(N_s n_s + n_s)$. Keeping the dominant terms and taking off coefficient 2 lead to $\mathcal{O}(bN_s n_s) + \mathcal{O}(N_s n_s^2) + \mathcal{O}(N_t(n_s n_t)^2) + \mathcal{O}(N_s N_t n_s n_t)$. For LSPG projection, we compute $(I_{N_s} - \Delta t A(\mu)) \Phi_s$ which will be re-used, resulting in $\mathcal{O}(2bN_s n_s)$. Then, we compute $\Phi_s^T (I_{N_s} - \Delta t A(\mu))^T (I_{N_s} - \Delta t A(\mu)) \Phi_s$, using precomputed $n_s \times N_s$ matrix and its transpose matrix. Its computational cost is

$\mathcal{O}(N_s n_s^2)$. Similarly, when we compute $\Phi_s^T (I_{N_s} - \Delta t A(\mu)) \Phi_s$, the precomputed $n_s \times N_s$ matrix is multiplied by $N_s \times n_s$ matrix, resulting in computational complexity of $\mathcal{O}(N_s n_s^2)$. Now, we compute n_t^2 blocks for the reduced space–time system matrix $\hat{A}^{st,pg}(\mu)$. Each block involves $3N_t$ times multiplication by diagonal matrices of size n_s on the left and right side of $n_s \times n_s$ precomputed matrices and multiplication between diagonal matrices of size n_s , which gives us computational cost of $\mathcal{O}(N_t(6n_s^2 + n_s))$. Thus, it costs $\mathcal{O}(N_t n_t^2(6n_s^2 + n_s))$ to compute the reduced space–time system matrix $\hat{A}^{st,pg}(\mu)$. For the reduced space–time input $\hat{f}^{st,pg}(\mu)$ vector, n_t blocks are needed and each block requires $2N_t$ times $n_s \times N_s$ matrix and $N_s \times 1$ vector multiplication followed by diagonal matrix of size n_s and $n_s \times 1$ vector multiplication. The computational cost of computing each block is $\mathcal{O}(2N_t(N_s n_s + n_s))$. Thus, it takes $\mathcal{O}(2N_t n_t(N_s n_s + n_s))$ to compute the reduced space–time input $\hat{f}^{st,pg}(\mu)$. Computing the reduced space–time initial vector $\hat{u}_0^{st,pg}(\mu)$ costs $\mathcal{O}(N_s n_s + n_s)$ because we multiply the precomputed $n_s \times N_s$ matrix by a vector of size N_s ; then, a diagonal matrix of size n_s is multiplied on the left. Summing computational cost of $\hat{A}^{st,pg}(\mu)$, $\hat{f}^{st,pg}(\mu)$, and $\hat{u}_0^{st,pg}(\mu)$ yields $\mathcal{O}(2bN_s n_s) + \mathcal{O}(2N_s n_s^2) + \mathcal{O}(N_t n_t^2(6n_s^2 + n_s)) + \mathcal{O}(2N_t n_t(N_s n_s + n_s)) + \mathcal{O}(N_s n_s + n_s)$. 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 a computational cost of $\mathcal{O}(N_s N_t n_s n_t)$ for both Galerkin and LSPG projections with use of block structures.

In summary, the computational complexities of forming space–time ROM operators in the training phase for Galerkin and LSPG 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 LSPG computational complexities.

	Galerkin	LSPG
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 [72]. An a posteriori error bound is derived in this section. Here, we drop the parameter dependence for notational simplicity.

Theorem 1. We define the error at the k th time step as $e^{(k)} \equiv u^{(k)} - \tilde{u}^{(k)} \in \mathbb{R}^{N_s}$ where $u^{(k)} \in \mathbb{R}^{N_s}$ denotes FOM solution, $\tilde{u}^{(k)} \in \mathbb{R}^{N_s}$ denotes approximate solution, and $k \in \mathbb{N}(N_t)$. Let $A^{st} \in \mathbb{R}^{N_s N_t \times N_s N_t}$ be the space–time system matrix, $r^{(k)} \in \mathbb{R}^{N_s}$ be the residual computed using FOM solution at the k th time step, and $\tilde{r}^{(k)} \in \mathbb{R}^{N_s}$ be the residual computed using approximate solution at the k th time step. For example, $r^{(k)}$ and $\tilde{r}^{(k)}$ after applying the backward Euler scheme with the uniform time step become

$$r^{(k)}(u^{(k)}, u^{(k-1)}) = \Delta t f^{(k)} + u^{(k-1)} - (I - \Delta t A)u^{(k)} = 0 \quad (39)$$

$$\tilde{r}^{(k)}(\tilde{u}^{(k)}, \tilde{u}^{(k-1)}) = \Delta t f^{(k)} + \tilde{u}^{(k-1)} - (I - \Delta t A)\tilde{u}^{(k)} \quad (40)$$

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

$$\max_{k \in \mathbb{N}(N_t)} \|e^{(k)}\|_2 \leq \eta \max_{k \in \mathbb{N}(N_t)} \|\tilde{r}^{(k)}\|_2 \quad (41)$$

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

Proof. Let us define the space–time residual as

$$\mathbf{r}^{\text{st}} : \mathbf{v} \mapsto \mathbf{f}^{\text{st}} + \mathbf{u}_0^{\text{st}} - \mathbf{A}^{\text{st}} \mathbf{v} \quad (42)$$

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

$$\mathbf{r}^{\text{st}}(\mathbf{u}^{\text{st}}) = \mathbf{f}^{\text{st}} + \mathbf{u}_0^{\text{st}} - \mathbf{A}^{\text{st}} \mathbf{u}^{\text{st}} = \mathbf{0} \quad (43)$$

$$\mathbf{r}^{\text{st}}(\tilde{\mathbf{u}}^{\text{st}}) = \mathbf{f}^{\text{st}} + \mathbf{u}_0^{\text{st}} - \mathbf{A}^{\text{st}} \tilde{\mathbf{u}}^{\text{st}} \quad (44)$$

where $\mathbf{u}^{\text{st}} \in \mathbb{R}^{N_s N_t}$ is the space–time FOM solution, and $\tilde{\mathbf{u}}^{\text{st}} \in \mathbb{R}^{N_s N_t}$ is the approximate space–time solution. Subtracting Equation (44) from Equation (43) gives

$$\mathbf{r}^{\text{st}}(\tilde{\mathbf{u}}^{\text{st}}) = \mathbf{A}^{\text{st}} \mathbf{e}^{\text{st}} \quad (45)$$

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

$$\mathbf{e}^{\text{st}} = (\mathbf{A}^{\text{st}})^{-1} \mathbf{r}^{\text{st}}(\tilde{\mathbf{u}}^{\text{st}}). \quad (46)$$

Taking ℓ_2 norm and Hölders' inequality gives

$$\|\mathbf{e}^{\text{st}}\|_2 \leq \|(\mathbf{A}^{\text{st}})^{-1}\|_2 \|\mathbf{r}^{\text{st}}(\tilde{\mathbf{u}}^{\text{st}})\|_2. \quad (47)$$

We can rewrite this in the following form:

$$\sqrt{\sum_{k=1}^{N_t} \|\mathbf{e}^{(k)}\|_2^2} \leq \|(\mathbf{A}^{\text{st}})^{-1}\|_2 \sqrt{\sum_{k=1}^{N_t} \|\tilde{\mathbf{r}}^{(k)}\|_2^2}. \quad (48)$$

Using the relations

$$\max_{k \in \mathbb{N}(N_t)} \|\mathbf{e}^{(k)}\|_2^2 \leq \sum_{k=1}^{N_t} \|\mathbf{e}^{(k)}\|_2^2 \quad (49)$$

and

$$\sum_{k=1}^{N_t} \|\tilde{\mathbf{r}}^{(k)}\|_2^2 \leq N_t \max_{k \in \mathbb{N}(N_t)} \|\tilde{\mathbf{r}}^{(k)}\|_2^2, \quad (50)$$

we have

$$\max_{k \in \mathbb{N}(N_t)} \|\mathbf{e}^{(k)}\|_2 \leq \sqrt{N_t} \|(\mathbf{A}^{\text{st}})^{-1}\|_2 \max_{k \in \mathbb{N}(N_t)} \|\tilde{\mathbf{r}}^{(k)}\|_2, \quad (51)$$

which is equivalent to the error bound in (41). $\square$

A numerical demonstration with space–time system matrices, $\mathbf{A}^{\text{st}}$, that have the same structure as the ones used in Sections 7.1 and 7.2.1 shows the magnitude of $\|(\mathbf{A}^{\text{st}})^{-1}\|_2$ increases linearly for small N_t , while it becomes eventually flattened for large N_t as shown in Figure 2a 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 Figure 2b. This error bound shows much improvement against the ones for the spatial Galerkin and LSPG ROMs, which grows exponentially in time [72].

Figure 2. Growth rate of stability constant in Theorem 1. Backward Euler time stepping scheme with uniform time step size, $\Delta t = 10^{-2}$ is used. (a): $\|(A^{st})^{-1}\|_2$ in inequality (41), (b): Stability constant, η in inequality (41).

7. Numerical Results

In this section, we apply both the space–time Galerkin and LSPG 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 train parameters in a chosen domain. Then, the space–time ROMs are 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 $\tilde{u}^{st}(\mu)$ is assessed from its relative error by:

$$\text{relative error} = \frac{\|\tilde{u}^{st}(\mu) - u^{st}(\mu)\|_2}{\|u^{st}(\mu)\|_2} \quad (52)$$

and the ℓ_2 norm of space–time residual:

$$\|r^{st}(\tilde{u}^{st}(\mu))\|_2. \quad (53)$$

The computational cost is measured in terms of wall-clock time. The online phase includes all computations required to solve a problem with a new parameter. The online phase of FOM includes assembly of the FOM operators and time integration (i.e., FOM solving), and the online phase of the ROM consists of assembly of the ROM operators and time integration (i.e., ROM solving). The offline phase includes all computations that can be computed once and re-used for a new parameter. The offline phase of the FOM consists of computations that are independent of parameters. The offline phase of the ROM includes FOM snapshot generation, ROM basis generation, and parameter-independent ROM operator generation. Note that the parameter-independent ROM operators can be pre-computed during the offline phase only for parameter-separable cases. Then, the online speed-up is evaluated by dividing the wall-clock time of the FOM runtime by the online phase of the ROM. For the multi-query problems, total speed-up is evaluated by dividing the sum of runtime of all FOMs by the sum of elapsed time of the online and offline phase of all ROMs. All calculations are performed on an Intel(R) Core(TM) i9-10900T CPU @ 1.90 GHz and DDR4 Memory @ 2933 MHz.

In the following subsections, the backward Euler with uniform time step size is employed as a time integrator. For spatial differentiation, the second order central difference scheme for the diffusion terms and the first order backward difference scheme for the convective terms are implemented with uniform mesh size.

We conduct two kinds of tests for each model problem. In the first case, we compare relative errors, space–time residuals, and speed-ups of Galerkin and LSPG projections when the number of spatial basis vectors and temporal basis vectors are varied with the target parameter being fixed. We show the singular value decay of solution snapshot and temporal snapshot matrix for each problem.

We also show the accuracy (i.e., relative error) of the reduced order models versus the number of reduced basis. This clearly shows that the number of reduced basis affects

the accuracy of the reduced order models, depending on the singular value decay. We observe that the relative errors of Galerkin projection are slightly smaller, but the space–time residual is slightly larger than LSPG projection. This is because LSPG space–time ROM solution minimizes the space–time residual as formulated in Section 3.3. However, the Galerkin and LSPG methods give very similar performance both in terms of accuracy and speed-up.

Secondly, we investigate the generalization capability of both Galerkin and LSPG ROMs by solving several predictive cases. We see that, as the parameter points go beyond the train parameter domain, the accuracy of the Galerkin and LSPG ROMs start to deteriorate gradually. This implies that the Galerkin and LSPG ROMs have a trust region. Its trust region should be determined by the application space. We also see that the Galerkin and LSPG space–time ROMs achieve total speed-up in Section 7.2 by taking advantage of parameter–separability but both space–time ROMs are not able to achieve total speed-up in Section 7.1 because we cannot take advantage of parameter–separability, which plays an important role in efficient parametric reduced order models.

7.1. 2D Linear Diffusion Equation

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

$$\frac{\partial u}{\partial t} = \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] - \left[\frac{1}{\sqrt{(x - \mu_1)^2 + (y - \mu_2)^2}} \right] u + \frac{\sin 2\pi t}{\sqrt{(x - \mu_1)^2 + (y - \mu_2)^2}} \quad (54)$$

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

$$\begin{aligned} u(x = 0, y, t) &= 0 \\ u(x = 1, y, t) &= 0 \\ u(x, y = 0, t) &= 0 \\ u(x, y = 1, t) &= 0 \end{aligned} \quad (55)$$

and the initial condition is

$$u(x, y, t = 0) = 0. \quad (56)$$

In Equation (54), the parameters are nonlinearly used in terms of operators, i.e., fractional, so we cannot take advantage of parameter–separability for efficient formation of the reduced order model.

The uniform time step size is given by $\frac{2}{N_t}$ where we set $N_t = 50$. The space domain is discretized into $N_x = 70$ and $N_y = 70$ uniform meshes in x and y directions, respectively. Excluding boundary grid points, we have $N_s = (N_x - 1) \times (N_y - 1) = 4761$ spatial degrees of freedom. Multiplying N_t and N_s gives us 238,050 free degrees of freedom in space–time.

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

$$(\mu_1, \mu_2) \in \{(-0.9, -0.9), (-0.9, -0.5), (-0.5, -0.9), (-0.5, -0.5)\}, \quad (57)$$

at which the FOM is solved. The singular value decays of the solution snapshot and the temporal snapshot are shown in Figure 3.

Figure 3. 2D linear diffusion equation. Graph of singular value decay. (a): Singular value decay of solution snapshot, (b): Singular value decay of temporal snapshot for the first spatial basis vector.

The Galerkin and LSPG space–time ROMs solve the Equation (54) with the target parameter $(\mu_1, \mu_2) = (-0.7, -0.7)$. Figures 4–6 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 LSPG ROMs with $n_s = 5$ and $n_t = 3$ achieve a good accuracy (i.e., relative errors of 0.012% and 0.026%, respectively). However, they are not able to achieve an online speed-up (i.e., 0.082 and 0.073, respectively) because parameter–separability cannot be made use of. For such problems, local ROM operators can be constructed; then, we can use an interpolation to obtain ROM operator at a certain parameter point. For a detailed description of the local ROM interpolation method, please see [81]. By doing so, online speed-up can be achieved. Excluding assembly time of the FOM and ROM operators from the online phase yields $\mathcal{O}(10)$ solving speed-ups as shown in Figure 7. The Galerkin space–time ROM gives slightly lower relative error than the LSPG space–time ROM (see Figure 4), while the LSPG method gives a slightly lower space–time residual norm than the Galerkin method (see Figure 5), as we repeatedly stated in Section 7.

Figure 4. 2D linear diffusion equation. Relative errors vs. reduced dimensions. Note that the scales of the z-axis, i.e., the average relative error, are the same both for Galerkin and LSPG. Although the Galerkin achieves slightly lower minimum average relative error values than the LSPG, both Galerkin and LSPG show comparable results. (a): Relative errors vs. reduced dimensions for Galerkin projection, (b): Relative errors vs. reduced dimensions for LSPG projection.

Figure 5. 2D linear diffusion equation. Space-time residuals vs. reduced dimensions. Note that the scales of the z-axis, i.e., the residual norm, are the same both for Galerkin and LSPG. Although the LSPG achieves slightly lower minimum residual norm values than the Galerkin, both Galerkin and LSPG show comparable results. (a): Space-time residuals vs. reduced dimensions for Galerkin projection, (b): Space-time residuals vs. reduced dimensions for LSPG projection.

Figure 6. 2D linear diffusion equation. Online speed-ups vs. reduced dimensions. Both Galerkin and LSPG show similar speed-ups. (a): Online speed-ups vs. reduced dimensions for Galerkin projection, (b): Online speed-ups vs. reduced dimensions for LSPG projection.

Figure 7. 2D linear diffusion equation. Solving speed-ups vs. reduced dimensions. (a): Solving speed-ups vs. reduced dimensions for Galerkin projection, (b): Solving speed-ups vs. reduced dimensions for LSPG projection.

The final time snapshots of FOM, Galerkin space–time ROM, and LSPG space–time ROM are seen in Figure 8. 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 $n_t = 3$ take an average time of 3.9434×10^{-2} and 5.0507×10^{-1} s, respectively, resulting in online speed-up of 0.082. For the LSPG method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ take an average time of 3.9434×10^{-2} and 5.0507×10^{-1} s, respectively, resulting in speed-up of 0.078. For accuracy, the Galerkin method results in $1.210 \times 10^{-2}\%$ relative error and 1.249×10^{-2} space–time residual norm, while the LSPG results in $2.626 \times 10^{-2}\%$ relative error and 1.029×10^{-2} space–time residual norm.

Figure 8. 2D linear diffusion equation. Solution snapshots of FOM, Galerkin ROM, and LSPG ROM at $t = 2$. (a): FOM, (b): Galerkin ROM, (c): LSPG ROM.

Next, we construct space–time ROMs with a basis of $n_s = 5$ and $n_t = 3$ using the same train parameter set given in Equation (57) for solving the predictive cases. The test parameter set is $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = -1.7 + 1.5/14i, i = 0, 1, \dots, 14\} \times \{\mu_2 | \mu_2 = -1.7 + 1.5/14j, j = 0, 1, \dots, 14\}$. Figure 9 shows the relative errors over the test parameter set. The relative errors of both projection methods are the lowest within the rectangular domain defined by the train parameter points, i.e., $[-0.9, -0.5] \times [-0.9, -0.5]$. For Galerkin ROM, online speed-up is about 0.088 in average and total time for ROM and FOM are 105.89 and 9.31 s, respectively, resulting in total speed-up of 0.088. The total time for the ROM consists of 0.13% of FOM snapshot generation, 0.36% of ROM basis generation, 99.19% of ROM operator assembly, and 0.32% of time integration. For LSPG ROM, online speed-up is about 0.080 in average and total time for ROM and FOM are 114.87 and 9.17 s, respectively, resulting in total speed-up of 0.080. The breakdown of the total time for the ROM is as follows: 0.12% of FOM snapshot generation, 0.33% of ROM basis generation, 99.27% of ROM operator assembly, and 0.29% of time integration.

Figure 9. 2D linear diffusion equation. The comparison of the Galerkin and LSPG ROMs for predictive cases. The block dots indicate the train parameters. (a): Galerkin, (b): LSPG.

7.2. 2D Linear Convection Diffusion Equation

7.2.1. Without Source Term

We consider a parameterized 2D linear convection diffusion equation

$$\frac{\partial u}{\partial t} = -\mu_1 \left[\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} \right] + \mu_2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] \quad (58)$$

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

$$\begin{aligned} u(x=0, y, t) &= 0 \\ u(x=1, y, t) &= 0 \\ u(x, y=0, t) &= 0 \\ u(x, y=1, t) &= 0. \end{aligned} \quad (59)$$

The initial condition is given by

$$u(x, y, t=0) = \begin{cases} 100 \sin(2\pi x)^3 \cdot \sin(2\pi y)^3 & \text{if } (x, y) \in [0, 0.5] \times [0, 0.5] \\ 0 & \text{otherwise} \end{cases} \quad (60)$$

and shown in Figure 10.

Figure 10. Plot of Equation (60).

In this example, the parameters are used linearly in terms of operators, thus we can take advantage of parameter–separability for efficient projection of the full order model.

The time domain is discretized with the uniform time step size $\frac{1}{N_t}$ with $N_t = 50$. Discretizing the space domain into $N_x = 70$ and $N_y = 70$ uniform meshes in x and y directions, respectively and excluding boundary grid points give us $N_s = (N_x - 1) \times (N_y - 1) = 4761$ grid points. As a result, there are 238,050 free degrees of freedom in space–time.

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

$$(\mu_1, \mu_2) \in \{(0.03, 0.33), (0.03, 0.35), (0.05, 0.33), (0.05, 0.35)\}, \quad (61)$$

at which the FOM is solved. Figure 11 shows the singular value decay of the solution snapshot and the temporal snapshot.

Figure 11. 2D linear convection diffusion equation. Graph of singular value decay. (a): Singular value decay of solution snapshot, (b): Singular value decay of temporal snapshot for the first spatial basis vector.

The Galerkin and LSPG space–time ROMs solve the Equation (58) with the target parameter $(\mu_1, \mu_2) = (0.04, 0.34)$. Figures 12–14 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 LSPG 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., 26.52 and 28.59, respectively). Although the relative errors and space–time residuals are similar, the relative error of Galerkin space–time ROM is slightly lower than the LSPG space–time ROM as shown in Figure 12. On the other hand, Figure 13 shows that the space–time residual of the LSPG method is lower than the Galerkin method.

Figure 12. 2D linear convection diffusion equation. Relative errors vs. reduced dimensions. Note that the scales of the z-axis, i.e., the average relative error, are the same both for Galerkin and LSPG. Although the Galerkin achieves slightly lower minimum average relative error values than the LSPG, both Galerkin and LSPG show comparable results. (a): Relative errors vs. reduced dimensions for Galerkin projection, (b): Relative errors vs. reduced dimensions for LSPG projection.

Figure 13. 2D linear convection diffusion equation. Space-time residuals vs. reduced dimensions. Note that the scales of the z-axis, i.e., the residual norm, are the same both for Galerkin and LSPG. Although the LSPG achieves slightly lower minimum residual norm values than the Galerkin, both Galerkin and LSPG show comparable results. (a): Space-time residuals vs. reduced dimensions for Galerkin projection, (b): Space-time residuals vs. reduced dimensions for LSPG projection.

Figure 14. 2D linear convection diffusion equation. Online speed-ups vs. reduced dimensions. Both Galerkin and LSPG show similar speed-ups. (a): Online speed-ups vs. reduced dimensions for Galerkin projection, (b): Online speed-ups vs. reduced dimensions for LSPG projection.

The final time snapshots of FOM, Galerkin space–time ROM, and LSPG space–time ROM are seen in Figure 15. 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 3.4619×10^{-2} and 1.3052×10^{-3} s, respectively, resulting in speed-up of 26.52. For the LSPG method, the FOM and space–time ROM simulation with $n_s = 5$ and $n_t = 3$ takes an average time of 3.4447×10^{-2} and 1.2049×10^{-3} s, respectively, resulting in speed-up of 28.59. For accuracy, the Galerkin method results in $4.898 \times 10^{-2}\%$ relative error and 1.503 space–time residual norm while the LSPG results in $5.878 \times 10^{-2}\%$ relative error and 1.459 space–time residual norm.

Figure 15. 2D linear convection diffusion equation. Solution snapshots of FOM, Galerkin ROM, and LSPG ROM at $t = 1$. (a): FOM, (b): Galerkin ROM, (c): LSPG ROM.

Now, the space–time reduced order models with $n_s = 5$ and $n_t = 3$ are generated using the same train parameter set given in Equation (61) to solve the predictive cases with the test parameter set, $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = 0.01 + 0.06/11i, i = 0, 1, \dots, 11\} \times \{\mu_2 | \mu_2 = 0.31 + 0.06/11j, j = 0, 1, \dots, 11\}$. Figure 16 shows the relative errors over the test parameter set. The Galerkin and LSPG ROMs are the most accurate within the range of the train parameter points, i.e., $[0.03, 0.33] \times [0.05, 0.35]$. For Galerkin ROM, online speed-up is about 24.98 in average and total time for ROM and FOM are 1.23 and 5.62 s, respectively, resulting in total speed-up of 4.59. The total time for the ROM is composed of 10.97% of FOM snapshot generation, 32.12% of ROM basis generation, 38.38% of parameter-independent ROM operator generation, 0.35% of ROM operator assembly, and 18.18% of time integration. For LSPG ROM, online speed-up is about 24.79 in average and total time for ROM and FOM are 1.69 and 5.57 s, respectively, resulting in total speed-up of 3.30. The total time for the ROM breaks down into 7.84% of FOM snapshot generation, 22.99% of ROM basis generation, 55.81% of parameter-independent ROM operator generation, 0.38% of ROM operator assembly, and 12.97% time integration.

Figure 16. 2D linear convection diffusion equation. The comparison of the Galerkin and LSPG ROMs for predictive cases. The block dots indicate the train parameters. (a): Galerkin, (b): LSPG.

7.2.2. With Source Term

We consider a parameterized 2D linear convection diffusion equation

$$\frac{\partial u}{\partial t} = -\mu_1 \left[0.1 \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} \right] + \mu_2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] + f(x, y, t) \quad (62)$$

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

$$f(x, y, t) = 10^5 \exp\left(-\left(\left(\frac{x - 0.5 + 0.2 \sin(2\pi t)}{0.1}\right)^2 + \left(\frac{y}{0.05}\right)^2\right)\right) \quad (63)$$

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

$$\begin{aligned} u(x = 0, y, t) &= 0 \\ u(x = 1, y, t) &= 0 \\ u(x, y = 0, t) &= 0 \\ u(x, y = 1, t) &= 0 \end{aligned} \quad (64)$$

and the initial condition is given by

$$u(x, y, t = 0) = 0. \quad (65)$$

Note that the parameters can be factored out when forming reduced order model operators in this example. Thus, we can avoid a lot of re-computation for a parametric case.

For time domain, we set the uniform time step size $\frac{2}{N_t}$ with $N_t = 50$. For spatial domain, the number of uniform meshes in x and y directions are set $N_x = 70$ and $N_y = 70$, respectively. Considering free spatial degrees of freedom (i.e., excluding boundary grid points), we have $N_s = (N_x - 1) \times (N_y - 1) = 4761$ grid points. Then, the number of space–time free degrees of freedom becomes 238,050.

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

$$(\mu_1, \mu_2) \in \{(0.195, 0.018), (0.195, 0.022), (0.205, 0.018), (0.205, 0.022)\}, \quad (66)$$

at which the FOM is solved. In Figure 17, we can see how the singular values of the solution snapshot and the temporal snapshot decay.

Figure 17. 2D linear convection diffusion equation with source term. Graph of singular value decay. (a): Singular value decay of solution snapshot, (b): Singular value decay of temporal snapshot for the first spatial basis vector.

The Galerkin and LSPG space–time ROMs solve the Equation (62) with the target parameter $(\mu_1, \mu_2) = (0.2, 0.02)$. Figures 18–20 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 LSPG 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., 9.18

and 8.54, respectively). From Figures 18 and 19, we find that there is the same observation that the LSPG method gives lower space–time residuals but higher relative errors as in Sections 7.1 and 7.2.1.

Figure 18. 2D linear convection diffusion equation with source term. Relative errors vs. reduced dimensions. Note that the scales of the z-axis, i.e., the average relative error, are the same both for Galerkin and LSPG. Although the Galerkin achieves slightly lower minimum average relative error values than the LSPG, both Galerkin and LSPG show comparable results. (a): Relative errors vs. reduced dimensions for Galerkin projection, (b): Relative errors vs. reduced dimensions for LSPG projection.

Figure 19. 2D linear convection diffusion equation with source term. Space-time residuals vs. reduced dimensions. Note that the scales of the z-axis, i.e., the residual norm, are the same both for Galerkin and LSPG. Although the LSPG achieves slightly lower minimum residual norm values than the Galerkin, both Galerkin and LSPG show comparable results. (a): Space-time residuals vs. reduced dimensions for Galerkin projection, (b): Space-time residuals vs. reduced dimensions for LSPG projection.

Figure 20. 2D linear convection diffusion equation with source term. Online speed-ups vs. reduced dimensions. Both Galerkin and LSPG show similar speed-ups. (a): Online speed-ups vs. reduced dimensions for Galerkin projection, (b): Online speed-ups vs. reduced dimensions for LSPG projection.

The final time snapshots of FOM, Galerkin space–time ROM, and LSPG space–time ROM are seen in Figure 21. Both ROMs have a basis size of $n_s = 19$ and $n_t = 3$, resulting in

a reduction factor of $(N_s N_t)/(n_s n_t) = 4176$. For the Galerkin method, the FOM and space-time ROM simulation with $n_s = 19$ and $n_t = 3$ takes an average time of 3.9159×10^{-2} and 4.2661×10^{-3} s, respectively, resulting in speed-up of 9.18. For the LSPG method, the FOM and space-time ROM simulation with $n_s = 19$ and $n_t = 3$ takes an average time of 3.6594×10^{-2} and 4.2827×10^{-3} s, respectively, resulting in speed-up of 8.54. For accuracy, the Galerkin method results in $2.174 \times 10^{-1}\%$ relative error and 1.564×10^3 space-time residual norm while the LSPG results in $2.652 \times 10^{-1}\%$ relative error and 1.550×10^3 space-time residual norm.

Figure 21. 2D linear convection diffusion equation with source term. Solution snapshots of FOM, Galerkin ROM, and LSPG ROM at $t = 2$. (a): FOM, (b): Galerkin ROM, (c): LSPG ROM.

Next, we set the test parameter set to be $(\mu_1, \mu_2) \in \{\mu_1 | \mu_1 = 0.160 + 0.08/11i, i = 0, 1, \dots, 11\} \times \{\mu_2 | \mu_2 = 0.016 + 0.008/11j, j = 0, 1, \dots, 11\}$. Then, predictive cases with the test parameter set are solved using the space-time ROMs which are generated using the train parameter set given by Equation (66) with $n_s = 19$ and $n_t = 3$. The relative errors over the test parameter set are shown in Figure 22. We see that the Galerkin and LSPG ROMs are the most accurate inside region of the rectangle defined by the train parameter points, i.e., $[0.195, 0.205] \times [0.018, 0.022]$. For Galerkin ROM, online speed-up is about 8.94 in average and total time for ROM and FOM are 1.82 and 5.99 s, respectively, resulting in total speed-up of 3.29. The total time for the ROM breaks down into 7.82% of FOM snapshot generation, 27.33% of ROM basis generation, 28.04% of parameter-independent ROM operator generation, 1.06% of ROM operator assembly, and 35.75% of time integration. For LSPG ROM, online speed-up is about 8.94 in average and total times for ROM and FOM are 2.35 and 6.11 s, respectively, resulting in total speed-up of 2.60. The total time of the ROM consists of 6.59% of FOM snapshot generation, 22.70% of ROM basis generation, 41.52% of parameter-independent ROM operator generation, 1.09% of ROM operator assembly, and 28.10% of time integration.

Figure 22. 2D linear convection diffusion equation with source term. The comparison of the Galerkin and LSPG ROMs for predictive cases. The block dots indicate the train parameters. (a): Galerkin, (b): LSPG.

8. Conclusions

In this work, we have formulated Galerkin and LSPG space–time ROMs for linear dynamical systems 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 LSPG space–time ROMs. We demonstrated that both Galerkin and LSPG space–time ROMs solve 2D linear dynamical systems with parameter–separability 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 $\mathcal{O}(10)$ online speed-ups, and their differences were negligible. 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 [91–93]. In future work, we aim to develop a nonlinear manifold based space–time ROM. Another interesting future direction is to build a component-wise space–time ROM as an extension to the work in [94], which is attractive for extreme-scale problems, whose simulation data are too big to store in a given memory.

Author Contributions: Conceptualization and methodology, Y.C.; coding and validation, Y.K. and K.W.; formal analysis and investigation, Y.K. and K.W.; writing—original draft preparation, Y.K., K.W. and Y.C.; writing—review and editing, Y.K., K.W. and Y.C.; visualization, Y.K.; supervision, Y.C.; project administration, Y.C.; funding acquisition, Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was performed at Lawrence Livermore National Laboratory and was supported by the LDRD program (project 20-FS-007).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data generated during the current study are available from the corresponding author on reasonable request.

Acknowledgments: We thank four anonymous reviewers whose comments/suggestions helped improve and clarify this manuscript. Youngkyu was 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.

Conflicts of Interest: The authors declare no conflict of interest. The funder had a role in the decision to publish the results. The funder had no role in the design of the study, in the collection, analyses, or interpretation of data, or in the writing of the manuscript.

Appendix 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 and they 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 LSPG 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 (119 lines)
4. All input code for the LSPG Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation (119 lines)
5. All input code for the Galerkin Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term (114 lines)
6. All input code for the LSPG Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term (119 lines)

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

```

1  import numpy as np; import scipy as sp
2  from scipy.sparse.linalg import spsolve, splu
3
4  # train parameter space
5  mu1,mu2=np.meshgrid(np.linspace(-0.9,-0.5,2),np.linspace(-0.9,-0.5,2),indexing='ij')
6  mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)
7
8  # space and time domain
9  N=70; Ns=(N-1)**2; Nt=50; h=1/N; k=2/Nt
10 x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
11 x1=x1.flatten(); x2=x2.flatten(); t=np.linspace(0,2,Nt+1)
12
13 # basic operators
14 e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')
15 A1D=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
16 A2D=sp.sparse.kron(A1D,sp.sparse.eye(N-1,format='csc'),format='csc')\
17     +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D,format='csc')
18
19 # snapshot, free DoFs, No IC included
20 U=np.zeros((Ns,no_para*Nt))
21 for p in range(no_para):
22     A=A2D-sp.sparse.diags(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2),format='csc'); invA=splu(I-k*A)
23     f=(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
24     u=np.zeros((Ns,Nt+1)) # IC is zero
25     for n in range(Nt):
26         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
27     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
28
29 W,S,VT=np.linalg.svd(U) # POD of solution snapshot
30
31 # test parameter space
32 nparam1=15; nparam2=15; ParamD1,ParamD2=np.meshgrid(np.linspace(-1.7,-0.2,nparam1),np.linspace(-1.7,-0.2,nparam2))
33 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
34
35 # variables to store (also store ParamD1 and ParamD2)
36 UROM_g2=np.zeros((num_test,Nt+1,N+1)); UFOM_g2=np.zeros((num_test,Nt+1,N+1))
37 AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))
38
39 # Construct Phi_s
40 ns=5; PHIs=W[:,ns]; PHIsT=PHIs.T
41
42 # construct Djk
43 nt=3; D=np.zeros((ns,nt*Nt))
44 for i in range(ns):
45     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
46     Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,nt]
47     for j in range(nt):
48         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
49

```

```

50 # construct PHist for post processing, i.e., reconstruction
51 PHist=np.zeros((Ns*Nt,ns*nt))
52 for i in range(Nt):
53     for j in range(nt):
54         PHistij=np.zeros((Ns,ns))
55         Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
56         PHist[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHis@Dij
57 PHistT=PHist.T
58
59 for test_count in range(num_test): # predictive cases
60     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter
61
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=PHistT@A@PHis; fs=PHistT*f
68     Ast=np.zeros((ns*nt,ns*nt))
69     for i in range(nt):
70         for j in range(nt):
71             Astij=np.zeros((ns,ns))
72             for kk in range(Nt):
73                 Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
74                 Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
75                 Astij+=Dik@Djk-k*Dik@As@Djk
76                 if kk != Nt-1:
77                     Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
78                     Astij-=Dik_next@Djk
79             Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
80     fst=np.zeros((ns*nt))
81     for j in range(nt):
82         for kk in range(Nt):
83             Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
84             fst[ns*j:ns*(j+1)]=k*Djk@fs[:,kk+1]
85
86     # Space-Time ROM (online phase)
87     UstROM=PHist@np.linalg.solve(Ast,fst)
88
89     # FOM
90     u=np.zeros((Ns,Nt+1)); invA=splu(I-k*A)
91     for n in range(Nt):
92         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
93     UstFOM=u[:,1:].flatten(order='F')
94
95     AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
96
97     rstROM=np.zeros((Ns*Nt)) # ST residual
98     for n in range(1,Nt+1):
99         if n==1:
100             rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
101         else:
102             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])
103     STResROM_g2[test_count]=np.linalg.norm(rstROM)
104
105     # store solutions for each param
106     urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
107     urom[0]=np.zeros((N+1,N+1)); ufom[0]=np.zeros((N+1,N+1))
108     for n in range(1,Nt+1):
109         urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
110         ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
111     UROM_g2[test_count]=urom; UFOM_g2[test_count]=ufom

```

Appendix A.2. LSPG Reduced Order Model for 2D Implicit Linear Diffusion Equation with Source Term

```

1 import numpy as np; import scipy as sp
2 from scipy.sparse.linalg import spsolve, splu
3
4 # train parameter space
5 mu1,mu2=np.meshgrid(np.linspace(-0.9,-0.5,2),np.linspace(-0.9,-0.5,2),indexing='ij')
6 mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)
7
8 # space and time domain
9 N=70; Ns=(N-1)**2; Nt=50; h=1/N; k=2/Nt
10 x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
11 x1=x1.flatten(); x2=x2.flatten(); t=np.linspace(0,2,Nt+1)
12
13 # basic operators
14 e=np.ones(N-1); I=sp.sparse.eye(Ns,format='csc')

```

```

15 A1D=1/h**2*sp.sparse.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
16 A2D=sp.sparse.kron(A1D,sp.sparse.eye(N-1,format='csc'),format='csc')\
17     +sp.sparse.kron(sp.sparse.eye(N-1,format='csc'),A1D,format='csc')
18
19 # snapshot, free DoFs, No IC included
20 U=np.zeros((Ns,no_para*Nt))
21 for p in range(no_para):
22     A=A2D-sp.sparse.diags(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2),format='csc'); invA=splu(I-k*A)
23     f=(1/np.sqrt((x1-mu1[p])**2+(x2-mu2[p])**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
24     u=np.zeros((Ns,Nt+1)) # IC is zero
25     for n in range(Nt):
26         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
27     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
28
29 W,S,VT=np.linalg.svd(U) # POD of solution snapshot
30
31 # test parameter space
32 nparam1=15; nparam2=15; ParamD1,ParamD2=np.meshgrid(np.linspace(-1.7,-0.2,nparam1),np.linspace(-1.7,-0.2,nparam2))
33 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
34
35 # variables to store (also store ParamD1 and ParamD2)
36 UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
37 AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))
38
39 # Construct Phi_s
40 ns=5; PHIs=W[:,ns]; PHIsT=PHIs.T
41
42 # construct Djk
43 nt=3; D=np.zeros((ns,nt*Nt))
44 for i in range(ns):
45     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
46     Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,nt]
47     for j in range(nt):
48         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
49
50 # construct PHIt for post processing, i.e., reconstruction
51 PHIt=np.zeros((Ns*Nt,ns*nt))
52 for i in range(Nt):
53     for j in range(nt):
54         PHItij=np.zeros((Ns,ns))
55         Dij=sp.sparse.diags(D[:,j*Nt+i],format='csc')
56         PHIt[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
57 PHItT=PHIt.T
58
59 # predictive cases
60 for test_count in range(num_test):
61     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter
62
63     # set model
64     A=A2D-sp.sparse.diags(1/np.sqrt((x1-param1)**2+(x2-param2)**2),format='csc')
65     f=(1/np.sqrt((x1-param1)**2+(x2-param2)**2).reshape(-1,1))@(np.sin(2*np.pi*t).reshape(1,-1))
66
67     # construct Ast and fst using block structure (ust0 is zero)
68     As=PHIsT@A@PHIs; fs=PHIsT@f
69     ATAs=PHIsT@A@PHIs; AIs=PHIsT@A@PHIs; ATIs=PHIsT@A@PHIs; ATIfs=PHIsT@A@PHIs; ATIfs=PHIsT@A@PHIs
70     Ast=np.zeros((ns*nt,ns*nt))
71     for i in range(nt):
72         for j in range(nt):
73             Astij=np.zeros((ns,ns))
74             for kk in range(Nt):
75                 Dik=sp.sparse.diags(D[:,i*Nt+kk],format='csc')
76                 Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
77                 Astij += Dik@ATAs@Djk
78                 if kk != Nt-1:
79                     Dik_next=sp.sparse.diags(D[:,i*Nt+kk+1],format='csc')
80                     Djk_next=sp.sparse.diags(D[:,j*Nt+kk+1],format='csc')
81                     Astij += Dik@AIs@Djk_next - Dik_next@ATIs@Djk
82             Ast[ns*i:ns*(i+1),ns*j:ns*(j+1)]=Astij
83     fst=np.zeros(ns*nt)
84     for j in range(nt):
85         for kk in range(Nt):
86             Djk=sp.sparse.diags(D[:,j*Nt+kk],format='csc')
87             fst[ns*j:ns*(j+1)] += k*Djk@ATIfs[:,kk+1]
88             if kk != Nt-1:
89                 fs_next = fs[:,kk+1+1]
90                 fst[ns*j:ns*(j+1)] -= k*Djk@fs_next
91
92 # Space-Time ROM (online phase)
93 UstROM=PHIt@np.linalg.solve(Ast,fst)
94
95 # FOM

```

```

96     u=np.zeros((Ns,Nt+1)); invA=splu(I-k*A)
97     for n in range(Nt):
98         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
99     UstFOM=u[:,1:].flatten(order='F')
100
101     AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
102
103     rstROM=np.zeros(Ns*Nt) # ST residual
104     for n in range(1,Nt+1):
105         if n==1:
106             rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
107         else:
108             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])
109     STResROM_pg2[test_count]=np.linalg.norm(rstROM)
110
111     # store solutions for each param
112     urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
113     urom[0]=np.zeros((N+1,N+1)); ufom[0]=np.zeros((N+1,N+1))
114     for n in range(1,Nt+1):
115         urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
116         ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
117     UROM_pg2[test_count]=urom; UFOM_2[test_count]=ufom

```

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

```

1  import numpy as np; import scipy.sparse as sp
2  from scipy.sparse.linalg import spsolve, splu
3
4  # train parameter space
5  mu1,mu2=np.meshgrid(np.linspace(0.03,0.05,2),np.linspace(0.33,0.35,2),indexing='ij')
6  mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)
7
8  # space and time domain
9  N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt
10 x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
11
12 # IC
13 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()
14
15 # basic operators
16 e=np.ones(N-1); I=sp.eye(Ns,format='csc')
17 A1D_diff=1/h**2*sp.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
18 A2D_diff=sp.kron(A1D_diff,sp.eye(N-1,format='csc'),format='csc')\
19     +sp.kron(sp.eye(N-1,format='csc'),A1D_diff,format='csc')
20 A1D_conv=1/(h)*sp.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
21 A2D_conv=sp.kron(A1D_conv,sp.eye(N-1,format='csc'),format='csc')\
22     +sp.kron(sp.eye(N-1,format='csc'),A1D_conv,format='csc')
23
24 # snapshot, free DoFs, No IC included
25 U=np.zeros((Ns,no_para*Nt))
26 for p in range(no_para):
27     A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff; invA=splu(I-k*A)
28     u=np.zeros((Ns,Nt+1)); u[:,0]=u0
29     for n in range(Nt):
30         u[:,n+1]=invA.solve(u[:,n])
31     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
32
33 # POD of solution snapshot
34 W,S,VT=np.linalg.svd(U)
35
36 # test parameter space
37 nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.01,0.07,nparam1),np.linspace(0.31,0.37,nparam2))
38 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
39
40 # variables to store (also store ParamD1 and ParamD2)
41 UROM_g2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
42 AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))
43
44 # Construct Phi_s
45 ns=5; PHIs=W[:,ns]; PHIsT=PHIs.T
46
47 # construct Dj_k
48 nt=3; D=np.zeros((ns,nt*Nt))
49 for i in range(ns):
50     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
51     Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,nt]
52     for j in range(nt):
53         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
54

```



```

55 # construct PHist for post processing, i.e., reconstruction
56 PHist=np.zeros((Ns*Nt,ns*nt))
57 for i in range(Nt):
58     for j in range(nt):
59         PHistij=np.zeros((Ns,ns))
60         Dij=sp.diags(D[:,j*Nt+i],format='csc')
61         PHist[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHis@Dij
62 PHistT=PHist.T
63
64 # precompute (parameter separation)
65 A1s=PHisT@A2D_conv@PHis; A2s=PHisT@A2D_diff@PHis;
66 DD=np.zeros((ns*nt,ns*nt)); DA1sD=np.zeros((ns*nt,ns*nt)); DA2sD=np.zeros((ns*nt,ns*nt))
67 for i in range(nt):
68     for j in range(nt):
69         DDij=np.zeros((ns,ns)); DA1sDij=np.zeros((ns,ns)); DA2sDij=np.zeros((ns,ns))
70         for kk in range(Nt):
71             Dik=sp.diags(D[:,i*Nt+kk],format='csc'); Djk=sp.diags(D[:,j*Nt+kk],format='csc')
72             DDij+=Dik@Djk; DA1sDij+=k*Dik@A1sDjk; DA2sDij+=k*Dik@A2sDjk
73             if kk != Nt-1:
74                 Dik_next=sp.diags(D[:,i*Nt+kk+1],format='csc')
75                 DDij+=Dik_next@Djk
76             DD[Ns*i:ns*(i+1),ns*j:ns*(j+1)]=DDij; DA1sD[Ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sDij
77             DA2sD[Ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sDij
78
79 # construct Ust
80 ust=np.zeros(ns*nt); us0=PHisT@u0
81 for j in range(nt):
82     Djk=sp.diags(D[:,j*Nt],format='csc')
83     ust[ns*j:ns*(j+1)] = Djk@us0
84
85 # predictive cases
86 for test_count in range(num_test):
87     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu
88
89     # construct Ast (fst is zero)
90     Ast=DD-param1*DA1sD+param2*DA2sD
91
92     A = -param1*A2D_conv + param2*A2D_diff # set model
93
94     # Space-Time ROM (online phase)
95     UstROM=PHist@np.linalg.solve(Ast,ust)
96
97     # FOM
98     u=np.zeros((Ns,Nt+1)); u[:,0] = u0; invA=splu(I-k*A)
99     for n in range(Nt):
100         u[:,n+1]=invA.solve(u[:,n])
101     UstFOM=u[:,1:].flatten(order='F')
102
103     AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
104
105     rstROM=np.zeros(Ns*Nt) # ST residual
106     for n in range(1,Nt+1):
107         if n==1:
108             rstROM[(n-1)*Ns:n*Ns]=u0-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
109         else:
110             rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
111     STResROM_g2[test_count]=np.linalg.norm(rstROM)
112
113     # store solutions for each param
114     urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
115     urom[0,1:-1,1:-1]=u0.reshape((N-1,N-1)); ufom[0,1:-1,1:-1]=u0.reshape((N-1,N-1))
116     for n in range(1,Nt+1):
117         urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
118         ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
119     UROM_g2[test_count]=urom; UFOM_2[test_count]=ufom

```

Appendix A.4. LSPG Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation

```

1 import numpy as np; import scipy.sparse as sp; from scipy.sparse.linalg import spsolve, splu
2
3 # train parameter space
4 mu1,mu2=np.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
7 # space and time domain
8 N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt
9 x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='ij')
10
11 # IC

```

```

12 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()
13
14 # basic operators
15 e=np.ones(N-1); I=sp.eye(Ns,format='csc')
16 A1D_diff=1/h**2*sp.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
17 A2D_diff=sp.kron(A1D_diff,sp.eye(N-1,format='csc'),format='csc')+sp.kron(sp.eye(N-1,format='csc'),A1D_diff,format='csc')
18 A1D_conv=1/(h)*sp.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
19 A2D_conv=sp.kron(A1D_conv,sp.eye(N-1,format='csc'),format='csc')+sp.kron(sp.eye(N-1,format='csc'),A1D_conv,format='csc')
20
21 # snapshot, free DoFs, No IC included
22 U=np.zeros((Ns,no_para*Nt))
23 for p in range(no_para):
24     A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff; invA=splu(I-k*A); u=np.zeros((Ns,Nt+1)); u[:,0]=u0
25     for n in range(Nt):
26         u[:,n+1]=invA.solve(u[:,n])
27     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
28
29 # POD of solution snapshot
30 W,S,VT=np.linalg.svd(U)
31
32 # test parameter space
33 nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.01,0.07,nparam1),np.linspace(0.31,0.37,nparam2))
34 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
35
36 # variables to store (also store ParamD1 and ParamD2)
37 UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
38 AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))
39
40 # Construct Phi_s
41 ns=5; PHIs=W[:, :ns]; PHIsT=PHIs.T
42
43 # construct Dj_k
44 nt=3; D=np.zeros((ns,nt*Nt))
45 for i in range(ns):
46     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F'); Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:, :nt]
47     for j in range(nt):
48         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
49
50 # construct PHIST for post processing, i.e., reconstruction
51 PHIST=np.zeros((Ns*Nt,ns*nt))
52 for i in range(Nt):
53     for j in range(nt):
54         PHISTij=np.zeros((Ns,ns)); Dij=sp.diags(D[:,j*Nt+i],format='csc')
55         PHIST[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
56 PHIST=PHIST.T
57
58 # precompute (parameter separation)
59 PHIsTA1T=PHIsT@A2D_conv.T; PHIsTA2T=PHIsT@A2D_diff.T; A1s=PHIsT@A2D_conv@PHIs; A2s=PHIsT@A2D_diff@PHIs
60 A1A2s=PHIsTA1T@A2D_diff@PHIs; A1A1s=PHIsTA1T@A2D_conv@PHIs; A2A2s=PHIsTA2T@A2D_diff@PHIs
61 DD=np.zeros((ns*nt,ns*nt)); DA1sD=np.zeros((ns*nt,ns*nt)); DA2sD=np.zeros((ns*nt,ns*nt));
62 DA1sA2sD=np.zeros((ns*nt,ns*nt)); DA1sA1sD=np.zeros((ns*nt,ns*nt)); DA2sA2sD=np.zeros((ns*nt,ns*nt))
63 for i in range(nt):
64     for j in range(nt):
65         DDij=np.zeros((ns,ns)); DA1sDij=np.zeros((ns,ns)); DA2sDij=np.zeros((ns,ns))
66         DA1sA2sDij=np.zeros((ns,ns)); DA1sA1sDij=np.zeros((ns,ns)); DA2sA2sDij=np.zeros((ns,ns))
67         for kk in range(Nt):
68             Dik=sp.diags(D[:,i*Nt+kk],format='csc'); Dj_k=sp.diags(D[:,j*Nt+kk],format='csc')
69             DDij += Dik@Dj_k; DA1sDij += k*Dik@(A1s.T+A1s)@Dj_k; DA2sDij += k*Dik@(A2s.T+A2s)@Dj_k
70             DA1sA2sDij += k**2*Dik@(A1A2s.T+A1A2s)@Dj_k; DA1sA1sDij += k**2*Dik@(A1A1s)@Dj_k; DA2sA2sDij += k**2*Dik@(A2A2s)@Dj_k
71             if kk != Nt-1:
72                 Dik_next=sp.diags(D[:,i*Nt+kk+1],format='csc'); Dj_k_next=sp.diags(D[:,j*Nt+kk+1],format='csc')
73                 DDij += Dik@Dj_k - Dik@Dj_k_next - Dik_next@Dj_k; DA1sDij += k*(Dik@A1s@Dj_k_next+Dik_next@A1s.T@Dj_k)
74                 DA2sDij += k*(Dik@A2s@Dj_k_next+Dik_next@A2s.T@Dj_k)
75                 DD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DDij; DA1sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sDij
76                 DA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sDij; DA1sA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sA2sDij
77                 DA1sA1sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sA1sDij; DA2sA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sA2sDij
78 DPHIsTu0=np.zeros(ns*nt); DA1sTu0=np.zeros(ns*nt); DA2sTu0=np.zeros(ns*nt)
79 us0=PHIsT@u0; PHIsTA1Tu0=PHIsTA1T@u0; PHIsTA2Tu0=PHIsTA2T@u0
80 for j in range(nt):
81     Dj_k=sp.diags(D[:,j*Nt],format='csc')
82     DPHIsTu0[ns*j:ns*(j+1)]=Dj_k@us0; DA1sTu0[ns*j:ns*(j+1)]=-k*Dj_k@PHIsTA1Tu0; DA2sTu0[ns*j:ns*(j+1)]=-k*Dj_k@PHIsTA2Tu0
83
84 # predictive cases
85 for test_count in range(num_test):
86     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu
87
88     # construct Ast, ust (fst is zero)
89     Ast=DD-param1*DA1sD+param2*DA2sD-param1*param2*DA1sA2sD+param1**2*DA1sA1sD+param2**2*DA2sA2sD
90     ust=DPHIsTu0-param1*DA1sTu0+param2*DA2sTu0
91
92     A= -param1*A2D_conv + param2*A2D_diff # set model

```

```

93
94 # Space-Time ROM (online phase)
95 UstROM=PHIst@np.linalg.solve(Ast,ust)
96
97 # FOM
98 u=np.zeros((Ns,Nt+1)); u[:,0] = u0; invA=splu(I-k*A)
99 for n in range(Nt):
100     u[:,n+1]=invA.solve(u[:,n])
101 UstFOM=u[:,1:].flatten(order='F')
102
103 AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
104
105 rstROM=np.zeros(Ns*Nt) # ST residual
106 for n in range(1,Nt+1):
107     if n==1:
108         rstROM[(n-1)*Ns:n*Ns]=u0-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
109     else:
110         rstROM[(n-1)*Ns:n*Ns]=UstROM[(n-2)*Ns:(n-1)*Ns]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
111 STResROM_pg2[test_count]=np.linalg.norm(rstROM)
112
113 # store solutions for each param
114 urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
115 urom[0,1:-1,1:-1]=u0.reshape((N-1,N-1)); ufom[0,1:-1,1:-1]=u0.reshape((N-1,N-1))
116 for n in range(1,Nt+1):
117     urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
118     ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
119 UROM_pg2[test_count]=urom; UFOM_2[test_count]=ufom

```

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

```

1 import numpy as np; import scipy.sparse as sp; from scipy.sparse.linalg import spsolve, splu
2
3 # train parameter space
4 mu1,mu2=np.meshgrid(np.linspace(0.195,0.205,2),np.linspace(0.018,0.022,2),indexing='xy')
5 mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)
6
7 # space and time domain
8 N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 2.; k=Tfinal/Nt; t=np.linspace(0,Tfinal,Nt+1)
9 x1,x2=np.meshgrid(np.linspace(0,1,N+1)[1:-1],np.linspace(0,1,N+1)[1:-1],indexing='xy')
10 x1=x1.flatten(); x2=x2.flatten()
11
12 # basic operators
13 e=np.ones(N-1); I=sp.eye(Ns,format='csc')
14 A1D_diff=1/h**2*sp.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
15 A2D_diff=sp.kron(A1D_diff,sp.eye(N-1,format='csc'),format='csc')+sp.kron(sp.eye(N-1,format='csc'),A1D_diff,format='csc')
16 A1D_conv=1/(h)*sp.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
17 A2D_conv=sp.kron(A1D_conv,sp.eye(N-1,format='csc'),format='csc')+0.1*sp.kron(sp.eye(N-1,format='csc'),A1D_conv,format='csc')
18
19 # snapshot, free DoFs, No IC included
20 f=np.zeros((Ns,Nt+1))
21 for i in range(Nt+1):
22     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))
23 U=np.zeros((Ns,no_para*Nt))
24 for p in range(no_para):
25     A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff; invA=splu(I-k*A); u=np.zeros((Ns,Nt+1))
26     for n in range(Nt):
27         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
28     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
29
30 # POD of solution snapshot
31 W,S,VT=np.linalg.svd(U)
32
33 # test parameter space
34 nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.160,0.240,nparam1),np.linspace(0.016,0.024,nparam2))
35 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
36
37 # variables to store (also store ParamD1 and ParamD2)
38 UROM_g2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_2=np.zeros((num_test,Nt+1,N+1,N+1))
39 AvgRelErr_g2=np.zeros((num_test)); STResROM_g2=np.zeros((num_test))
40
41 # Construct Phi_s
42 ns=19; PHIs=W[:,ns]; PHIsT=PHIs.T
43
44 # construct Djk
45 nt=3; D=np.zeros((ns,nt*Nt))
46 for i in range(ns):
47     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F')
48     Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,nt]

```

```

49     for j in range(nt):
50         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
51
52 # construct PHist for post processing, i.e., reconstruction
53 PHist=np.zeros((Ns*Nt,ns*nt))
54 for i in range(Nt):
55     for j in range(nt):
56         PHistj=np.zeros((Ns,ns))
57         Dij=sp.diags(D[:,j*Nt+i],format='csc')
58         PHist[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
59 PHIST=PHist.T
60
61 # precompute (parameter separation)
62 A1s=PHIST@A2D_conv@PHIs; A2s=PHIST@A2D_diff@PHIs;
63 DD=np.zeros((ns*nt,ns*nt)); DA1sD=np.zeros((ns*nt,ns*nt)); DA2sD=np.zeros((ns*nt,ns*nt))
64 for i in range(nt):
65     for j in range(nt):
66         DDij=np.zeros((ns,ns)); DA1sDij=np.zeros((ns,ns)); DA2sDij=np.zeros((ns,ns))
67         for kk in range(Nt):
68             Dik=sp.diags(D[:,i*Nt+kk],format='csc'); Djk=sp.diags(D[:,j*Nt+kk],format='csc')
69             DDij+=Dik@Djk; DA1sDij+=k*Dik@A1s@Djk; DA2sDij+=k*Dik@A2s@Djk
70             if kk != Nt-1:
71                 Dik_next=sp.diags(D[:,i*Nt+kk+1],format='csc'); DDij-=Dik_next@Djk
72                 DD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DDij; DA1sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sDij
73                 DA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sDij
74
75 # construct fst
76 fst=np.zeros(ns*nt); fs=PHIST@f
77 for j in range(nt):
78     for kk in range(Nt):
79         Djk=sp.diags(D[:,j*Nt+kk],format='csc'); fst[ns*j:ns*(j+1)]=k*Djk@fs[:,kk+1]
80
81 # predictive cases
82 for test_count in range(num_test):
83     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu
84
85     # construct Ast (ust is zero)
86     Ast=DD-param1*DA1sD+param2*DA2sD
87
88     A= -param1*A2D_conv + param2*A2D_diff # set model
89
90     # Space-Time ROM (online phase)
91     UstROM=PHIST@np.linalg.solve(Ast,fst)
92
93     # FOM
94     u=np.zeros((Ns,Nt+1)); invA=splu(I-k*A)
95     for n in range(Nt):
96         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
97     UstFOM=u[:,1:].flatten(order='F')
98
99     AvgRelErr_g2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
100
101     rstROM=np.zeros(Ns*Nt) # ST residual
102     for n in range(1,Nt+1):
103         if n==1:
104             rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
105         else:
106             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])
107     STResROM_g2[test_count]=np.linalg.norm(rstROM)
108
109     # store solutions for each param
110     urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
111     for n in range(1,Nt+1):
112         urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
113         ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
114     UROM_g2[test_count]=urom; UFOM_g2[test_count]=ufom

```

Appendix A.6. LSPG Reduced Order Model for 2D Implicit Linear Convection Diffusion Equation with Source Term

```

1 import numpy as np; import scipy.sparse as sp; from scipy.sparse.linalg import spsolve,splu
2
3 # train parameter space
4 mu1,mu2=np.meshgrid(np.linspace(0.195,0.205,2),np.linspace(0.018,0.022,2),indexing='xy')
5 mu1=mu1.flatten(); mu2=mu2.flatten(); no_para=np.size(mu1,axis=0)
6
7 # space and time domain
8 N=70; Ns=(N-1)**2; Nt=50; h=1./N; Tfinal = 1.; k=Tfinal/Nt; t=np.linspace(0,Tfinal,Nt+1)
9 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()

```

```

10
11 # basic operators
12 e=np.ones(N-1); I=sp.eye(Ns,format='csc')
13 A1D_diff=1/h**2*sp.spdiags(np.vstack((e,-2*e,e)),[-1,0,1],N-1,N-1,format='csc')
14 A2D_diff=sp.kron(A1D_diff,sp.eye(N-1,format='csc'),format='csc')+sp.kron(sp.eye(N-1,format='csc'),A1D_diff,format='csc')
15 A1D_conv=1/(h)*sp.spdiags(np.vstack((-1*e,1*e,0*e)),[-1,0,1],N-1,N-1,format='csc')
16 A2D_conv=sp.kron(A1D_conv,sp.eye(N-1,format='csc'),format='csc')+0.1*sp.kron(sp.eye(N-1,format='csc'),A1D_conv,format='csc')
17
18 # snapshot, free DoFs, No IC included
19 f=np.zeros((Ns,Nt+1))
20 for i in range(Nt+1):
21     f[:,i] = 1e5*np.exp(-((x1-0.5+(0.2*np.sin(2*np.pi*i))))/0.1)**2 + ((x2-0)/0.05)**2)
22 U=np.zeros((Ns,no_para*Nt))
23 for p in range(no_para):
24     A=-mu1[p]*A2D_conv + mu2[p]*A2D_diff; invA=splu(I-k*A); u=np.zeros((Ns,Nt+1))
25     for n in range(Nt):
26         u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
27     U[:,np.arange(p*Nt,(p+1)*Nt)]=u[:,1:]
28
29 # POD of solution snapshot
30 W,S,VT=np.linalg.svd(U)
31
32 # test parameter space
33 nparam1=12; nparam2=12; ParamD1,ParamD2=np.meshgrid(np.linspace(0.160,0.240,nparam1),np.linspace(0.016,0.024,nparam2))
34 ParamD1=ParamD1.flatten(); ParamD2=ParamD2.flatten(); num_test=np.prod(ParamD1.shape)
35
36 # variables to store (also store ParamD1 and ParamD2)
37 UROM_pg2=np.zeros((num_test,Nt+1,N+1,N+1)); UFOM_pg2=np.zeros((num_test,Nt+1,N+1,N+1))
38 AvgRelErr_pg2=np.zeros((num_test)); STResROM_pg2=np.zeros((num_test))
39
40 # Construct Phi_s
41 ns=19; PHIs=W[:,ns]; PHIsT=PHIs.T
42
43 # construct Djk
44 nt=3; D=np.zeros((ns,nt*Nt))
45 for i in range(ns):
46     Ri=VT[i,:]; Ri=Ri.reshape(-1,no_para,order='F'); Wi,Si,ViT=np.linalg.svd(Ri); PHIti=Wi[:,nt]
47     for j in range(nt):
48         D[i,Nt*j:Nt*(j+1)]=PHIti[:,j]
49
50 # construct PHIt for post processing, i.e., reconstruction
51 PHIt=np.zeros((Ns*Nt,ns*nt))
52 for i in range(Nt):
53     for j in range(nt):
54         PHItij=np.zeros((Ns,ns)); Dij=sp.diags(D[:,j*Nt+i],format='csc'); PHIt[Ns*i:Ns*(i+1),ns*j:ns*(j+1)]=PHIs@Dij
55 PHItT=PHIt.T
56
57 # precompute (parameter separation)
58 PHIsTA1T=PHIsT@A2D_conv.T; PHIsTA2T=PHIsT@A2D_diff.T; A1s=PHIsT@A2D_conv@PHIs; A2s=PHIsT@A2D_diff@PHIs
59 A1A2s=PHIsTA1T@A2D_diff@PHIs; A1A1s=PHIsTA1T@A2D_conv@PHIs; A2A2s=PHIsTA2T@A2D_diff@PHIs
60 DD=np.zeros((ns*nt,ns*nt)); DA1sD=np.zeros((ns*nt,ns*nt)); DA2sD=np.zeros((ns*nt,ns*nt))
61 DA1sA2sD=np.zeros((ns*nt,ns*nt)); DA1sA1sD=np.zeros((ns*nt,ns*nt)); DA2sA2sD=np.zeros((ns*nt,ns*nt))
62 for i in range(nt):
63     for j in range(nt):
64         DDij=np.zeros((ns,ns)); DA1sDij=np.zeros((ns,ns)); DA2sDij=np.zeros((ns,ns))
65         DA1sA2sDij=np.zeros((ns,ns)); DA1sA1sDij=np.zeros((ns,ns)); DA2sA2sDij=np.zeros((ns,ns))
66         for kk in range(Nt):
67             Dik=sp.diags(D[:,i*Nt+kk],format='csc'); Djk=sp.diags(D[:,j*Nt+kk],format='csc')
68             DDij += Dik@Djk; DA1sDij -= k*Dik@(A1s.T+A1s)@Djk; DA2sDij -= k*Dik@(A2s.T+A2s)@Djk
69             DA1sA2sDij += k**2*Dik@(A1A2s.T+A1A2s)@Djk; DA1sA1sDij += k**2*Dik@(A1A1s)@Djk; DA2sA2sDij += k**2*Dik@(A2A2s)@Djk
70             if kk != Nt-1:
71                 Dik_next=sp.diags(D[:,i*Nt+kk+1],format='csc'); Djk_next=sp.diags(D[:,j*Nt+kk+1],format='csc')
72                 DDij += Dik@Djk - Dik@Djk_next - Dik_next@Djk; DA1sDij += k*(Dik@A1s@Djk_next+Dik_next@A1s.T@Djk)
73                 DA2sDij += k*(Dik@A2s@Djk_next+Dik_next@A2s.T@Djk)
74                 DD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DDij; DA1sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sDij
75                 DA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sDij; DA1sA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sA2sDij
76                 DA1sA1sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA1sA1sDij; DA2sA2sD[ns*i:ns*(i+1),ns*j:ns*(j+1)]=DA2sA2sDij
77 Dfs=np.zeros(ns*nt); DA1sTf=np.zeros(ns*nt); DA2sTf=np.zeros(ns*nt); fs=PHIsT@f; A1sTf=PHIsTA1T@f; A2sTf=PHIsTA2T@f
78 for j in range(nt):
79     for kk in range(Nt):
80         Djk=sp.diags(D[:,j*Nt+kk],format='csc'); Dfs[ns*j:ns*(j+1)] += k*Djk@fs[:,kk+1]
81         DA1sTf[ns*j:ns*(j+1)] -= k**2*Djk@A1sTf[:,kk+1]; DA2sTf[ns*j:ns*(j+1)] -= k**2*Djk@A2sTf[:,kk+1]
82         if kk != Nt-1:
83             fs_next = fs[:,kk+1+1]; Dfs[ns*j:ns*(j+1)] -= k*Djk@fs_next
84
85 # predictive cases
86 for test_count in range(num_test):
87     param1,param2=ParamD1[test_count],ParamD2[test_count] # set parameter mu
88
89     # construct Ast, fst (ust is zero)
90     Ast=DD-param1*DA1sD+param2*DA2sD-param1*param2*DA1sA2sD+param1**2*DA1sA1sD+param2**2*DA2sA2sD

```

```

91     fst=Dfs-param1*DA1sTf+param2*DA2sTf
92
93     A= -param1*A2D_conv + param2*A2D_diff # set model
94
95     # Space-Time ROM (online phase)
96     UstROM=PHist@np.linalg.solve(Ast,fst)
97
98     # FOM
99     u=np.zeros((Ns,Nt+1)); invA=splu(I-k*A)
100    for n in range(Nt):
101        u[:,n+1]=invA.solve(u[:,n]+k*f[:,n+1])
102    UstFOM=u[:,1:].flatten(order='F')
103
104    AvgRelErr_pg2[test_count]=np.linalg.norm(UstROM-UstFOM)/np.linalg.norm(UstFOM) # Avg. rel. error
105
106    rstROM=np.zeros(Ns*Nt) # ST residual
107    for n in range(1,Nt+1):
108        if n==1:
109            rstROM[(n-1)*Ns:n*Ns]=k*f[:,n]-(I-k*A).dot(UstROM[(n-1)*Ns:n*Ns])
110        else:
111            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])
112    STResROM_pg2[test_count]=np.linalg.norm(rstROM)
113
114    # store solutions for each param
115    urom=np.zeros((Nt+1,N+1,N+1)); ufom=np.zeros((Nt+1,N+1,N+1))
116    for n in range(1,Nt+1):
117        urom[n,1:-1,1:-1]=UstROM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
118        ufom[n,1:-1,1:-1]=UstFOM[(n-1)*Ns:n*Ns].reshape((N-1,N-1))
119    UROM_pg2[test_count]=urom; UFOM_pg2[test_count]=ufom

```

References

- Mullis, C.; Roberts, R. Synthesis of minimum roundoff noise fixed point digital filters. *IEEE Trans. Circuits Syst.* **1976**, *23*, 551–562. [\[CrossRef\]](#)
- Moore, B. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Trans. Autom. Control* **1981**, *26*, 17–32. [\[CrossRef\]](#)
- Willcox, K.; Peraire, J. Balanced model reduction via the proper orthogonal decomposition. *AIAA J.* **2002**, *40*, 2323–2330. [\[CrossRef\]](#)
- Willcox, K.; Megretski, A. Fourier series for accurate, stable, reduced-order models in large-scale linear applications. *SIAM J. Sci. Comput.* **2005**, *26*, 944–962. [\[CrossRef\]](#)
- Heinkenschloss, M.; Sorensen, D.C.; Sun, K. Balanced truncation model reduction for a class of descriptor systems with application to the Oseen equations. *SIAM J. Sci. Comput.* **2008**, *30*, 1038–1063. [\[CrossRef\]](#)
- Sandberg, H.; Rantzer, A. Balanced truncation of linear time-varying systems. *IEEE Trans. Autom. Control* **2004**, *49*, 217–229. [\[CrossRef\]](#)
- Hartmann, C.; Vulcanov, V.M.; Schütte, C. Balanced truncation of linear second-order systems: A Hamiltonian approach. *Multiscale Model. Simul.* **2010**, *8*, 1348–1367. [\[CrossRef\]](#)
- Petreczky, M.; Wisniewski, R.; Leth, J. Balanced truncation for linear switched systems. *Nonlinear Anal. Hybrid Syst.* **2013**, *10*, 4–20. [\[CrossRef\]](#)
- Ma, Z.; Rowley, C.W.; Tadmor, G. Snapshot-based balanced truncation for linear time-periodic systems. *IEEE Trans. Autom. Control* **2010**, *55*, 469–473.
- Bai, Z. Krylov subspace techniques for reduced-order modeling of large-scale dynamical systems. *Appl. Numer. Math.* **2002**, *43*, 9–44. [\[CrossRef\]](#)
- Gugercin, S.; Antoulas, A.C.; Beattie, C. H₂ model reduction for large-scale linear dynamical systems. *SIAM J. Matrix Anal. Appl.* **2008**, *30*, 609–638. [\[CrossRef\]](#)
- Astolfi, A. Model reduction by moment matching for linear and nonlinear systems. *IEEE Trans. Autom. Control* **2010**, *55*, 2321–2336. [\[CrossRef\]](#)
- Chiprout, E.; Nakhla, M. Generalized moment-matching methods for transient analysis of interconnect networks. In Proceedings of the 29th ACM/IEEE Design Automation Conference, Anaheim, CA, USA, 8–12 June 1992; pp. 201–206.
- Pratesi, M.; Santucci, F.; Graziosi, F. Generalized moment matching for the linear combination of lognormal RVs: application to outage analysis in wireless systems. *IEEE Trans. Wirel. Commun.* **2006**, *5*, 1122–1132. [\[CrossRef\]](#)
- Ammar, A.; Mokdad, B.; Chinesta, F.; Keunings, R. A new family of solvers for some classes of multidimensional partial differential equations encountered in kinetic theory modeling of complex fluids. *J. Non-Newton. Fluid Mech.* **2006**, *139*, 153–176. [\[CrossRef\]](#)
- Ammar, A.; Mokdad, B.; Chinesta, F.; Keunings, R. 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. *J. Non-Newton. Fluid Mech.* **2007**, *144*, 98–121. [\[CrossRef\]](#)
- Chinesta, F.; Ammar, A.; Cueto, E. Proper generalized decomposition of multiscale models. *Int. J. Numer. Methods Eng.* **2010**, *83*, 1114–1132. [\[CrossRef\]](#)

18. Pruliere, E.; Chinesta, F.; Ammar, A. On the deterministic solution of multidimensional parametric models using the proper generalized decomposition. *Math. Comput. Simul.* **2010**, *81*, 791–810. [\[CrossRef\]](#)
19. Chinesta, F.; Ammar, A.; Leygue, A.; Keunings, R. An overview of the proper generalized decomposition with applications in computational rheology. *J. Non-Newton. Fluid Mech.* **2011**, *166*, 578–592. [\[CrossRef\]](#)
20. Giner, E.; Bognet, B.; Ródenas, J.J.; Leygue, A.; Fuenmayor, F.J.; Chinesta, F. The proper generalized decomposition (PGD) as a numerical procedure to solve 3D cracked plates in linear elastic fracture mechanics. *Int. J. Solids Struct.* **2013**, *50*, 1710–1720. [\[CrossRef\]](#)
21. Barbarulo, A.; Ladevèze, P.; Riou, H.; Kovalevsky, L. Proper generalized decomposition applied to linear acoustic: A new tool for broad band calculation. *J. Sound Vib.* **2014**, *333*, 2422–2431. [\[CrossRef\]](#)
22. Amsallem, D.; Farhat, C. Stabilization of projection-based reduced-order models. *Int. J. Numer. Methods Eng.* **2012**, *91*, 358–377. [\[CrossRef\]](#)
23. Amsallem, D.; Farhat, C. Interpolation method for adapting reduced-order models and application to aeroelasticity. *AIAA J.* **2008**, *46*, 1803–1813. [\[CrossRef\]](#)
24. Thomas, J.P.; Dowell, E.H.; Hall, K.C. Three-dimensional transonic aeroelasticity using proper orthogonal decomposition-based reduced-order models. *J. Aircr.* **2003**, *40*, 544–551. [\[CrossRef\]](#)
25. Hall, K.C.; Thomas, J.P.; Dowell, E.H. Proper orthogonal decomposition technique for transonic unsteady aerodynamic flows. *AIAA J.* **2000**, *38*, 1853–1862. [\[CrossRef\]](#)
26. Simoncini, V. A new iterative method for solving large-scale Lyapunov matrix equations. *SIAM J. Sci. Comput.* **2007**, *29*, 1268–1288. [\[CrossRef\]](#)
27. Benner, P.; Li, J.R.; Penzl, T. Numerical solution of large-scale Lyapunov equations, Riccati equations, and linear-quadratic optimal control problems. *Numer. Linear Algebra Appl.* **2008**, *15*, 755–777. [\[CrossRef\]](#)
28. Rowley, C.W. Model reduction for fluids, using balanced proper orthogonal decomposition. *Int. J. Bifurc. Chaos* **2005**, *15*, 997–1013. [\[CrossRef\]](#)
29. Ma, Z.; Ahuja, S.; Rowley, C.W. Reduced-order models for control of fluids using the eigensystem realization algorithm. *Theor. Comput. Fluid Dyn.* **2011**, *25*, 233–247. [\[CrossRef\]](#)
30. Lall, S.; Marsden, J.E.; Glavaški, S. A subspace approach to balanced truncation for model reduction of nonlinear control systems. *Int. J. Robust Nonlinear Control. IFAC-Affil. J.* **2002**, *12*, 519–535. [\[CrossRef\]](#)
31. Gosea, I.V.; Gugercin, S.; Beattie, C. Data-driven balancing of linear dynamical systems. *arXiv* **2021**, arXiv:2104.01006.
32. Chinesta, F.; Ladeveze, P.; Cueto, E. A short review on model order reduction based on proper generalized decomposition. *Arch. Comput. Methods Eng.* **2011**, *18*, 395. [\[CrossRef\]](#)
33. Mayo, A.; Antoulas, A. A framework for the solution of the generalized realization problem. *Linear Algebra Its Appl.* **2007**, *425*, 634–662. [\[CrossRef\]](#)
34. Scariotti, G.; Astolfi, A. Data-driven model reduction by moment matching for linear and nonlinear systems. *Automatica* **2017**, *79*, 340–351. [\[CrossRef\]](#)
35. Schmid, P.J. Dynamic mode decomposition of numerical and experimental data. *J. Fluid Mech.* **2010**, *656*, 5–28. [\[CrossRef\]](#)
36. Chen, K.K.; Tu, J.H.; Rowley, C.W. Variants of dynamic mode decomposition: Boundary condition, Koopman, and Fourier analyses. *J. Nonlinear Sci.* **2012**, *22*, 887–915. [\[CrossRef\]](#)
37. Williams, M.O.; Kevrekidis, I.G.; Rowley, C.W. A data—Driven approximation of the koopman operator: Extending dynamic mode decomposition. *J. Nonlinear Sci.* **2015**, *25*, 1307–1346. [\[CrossRef\]](#)
38. Takeishi, N.; Kawahara, Y.; Yairi, T. Learning Koopman invariant subspaces for dynamic mode decomposition. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; pp. 1130–1140.
39. Askham, T.; Kutz, J.N. Variable projection methods for an optimized dynamic mode decomposition. *SIAM J. Appl. Dyn. Syst.* **2018**, *17*, 380–416. [\[CrossRef\]](#)
40. Schmid, P.J.; Li, L.; Juniper, M.P.; Pust, O. Applications of the dynamic mode decomposition. *Theor. Comput. Fluid Dyn.* **2011**, *25*, 249–259. [\[CrossRef\]](#)
41. Kutz, J.N.; Fu, X.; Brunton, S.L. Multiresolution dynamic mode decomposition. *SIAM J. Appl. Dyn. Syst.* **2016**, *15*, 713–735. [\[CrossRef\]](#)
42. Li, Q.; Dietrich, F.; Bollt, E.M.; Kevrekidis, I.G. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the Koopman operator. *Chaos Interdiscip. J. Nonlinear Sci.* **2017**, *27*, 103111. [\[CrossRef\]](#)
43. Proctor, J.L.; Brunton, S.L.; Kutz, J.N. Dynamic mode decomposition with control. *SIAM J. Appl. Dyn. Syst.* **2016**, *15*, 142–161. [\[CrossRef\]](#)
44. Tu, J.H.; Rowley, C.W.; Luchtenburg, D.M.; Brunton, S.L.; Kutz, J.N. On dynamic mode decomposition: Theory and applications. *arXiv* **2013**, arXiv:1312.0041.
45. Kutz, J.N.; Brunton, S.L.; Brunton, B.W.; Proctor, J.L. *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*; SIAM: Philadelphia, PA, USA, 2016.
46. Choi, Y.; Coombs, D.; Anderson, R. SNS: A solution-based nonlinear subspace method for time-dependent model order reduction. *SIAM J. Sci. Comput.* **2020**, *42*, A1116–A1146. [\[CrossRef\]](#)
47. Hoang, C.; Choi, Y.; Carlberg, K. Domain-decomposition least-squares Petrov-Galerkin (DD-LSPG) nonlinear model reduction. *arXiv* **2020**, arXiv:2007.11835.

48. Carlberg, K.; Choi, Y.; Sargsyan, S. Conservative model reduction for finite-volume models. *J. Comput. Phys.* **2018**, *371*, 280–314. [\[CrossRef\]](#)
49. Berkooz, G.; Holmes, P.; Lumley, J.L. The proper orthogonal decomposition in the analysis of turbulent flows. *Annu. Rev. Fluid Mech.* **1993**, *25*, 539–575. [\[CrossRef\]](#)
50. Gubisch, M.; Volkwein, S. Proper orthogonal decomposition for linear-quadratic optimal control. *Model Reduct. Approx. Theory Algorithms* **2017**, *5*, 66.
51. Kunisch, K.; Volkwein, S. Galerkin proper orthogonal decomposition methods for parabolic problems. *Numer. Math.* **2001**, *90*, 117–148. [\[CrossRef\]](#)
52. Hinze, M.; Volkwein, S. Error estimates for abstract linear—Quadratic optimal control problems using proper orthogonal decomposition. *Comput. Optim. Appl.* **2008**, *39*, 319–345. [\[CrossRef\]](#)
53. Kerschen, G.; Golinval, J.C.; Vakakis, A.F.; Bergman, L.A. The method of proper orthogonal decomposition for dynamical characterization and order reduction of mechanical systems: An overview. *Nonlinear Dyn.* **2005**, *41*, 147–169. [\[CrossRef\]](#)
54. Bamer, F.; Bucher, C. Application of the proper orthogonal decomposition for linear and nonlinear structures under transient excitations. *Acta Mech.* **2012**, *223*, 2549–2563. [\[CrossRef\]](#)
55. Atwell, J.A.; King, B.B. Proper orthogonal decomposition for reduced basis feedback controllers for parabolic equations. *Math. Comput. Model.* **2001**, *33*, 1–19. [\[CrossRef\]](#)
56. Rathinam, M.; Petzold, L.R. A new look at proper orthogonal decomposition. *SIAM J. Numer. Anal.* **2003**, *41*, 1893–1925. [\[CrossRef\]](#)
57. Kahlbacher, M.; Volkwein, S. Galerkin proper orthogonal decomposition methods for parameter dependent elliptic systems. *Discuss. Math. Differ. Inclusions Control Optim.* **2007**, *27*, 95–117. [\[CrossRef\]](#)
58. Bonnet, J.P.; Cole, D.R.; Delville, J.; Glauser, M.N.; Ukeiley, L.S. Stochastic estimation and proper orthogonal decomposition: Complementary techniques for identifying structure. *Exp. Fluids* **1994**, *17*, 307–314. [\[CrossRef\]](#)
59. Placzek, A.; Tran, D.M.; Ohayon, R. Hybrid proper orthogonal decomposition formulation for linear structural dynamics. *J. Sound Vib.* **2008**, *318*, 943–964. [\[CrossRef\]](#)
60. LeGresley, P.; Alonso, J. Airfoil design optimization using reduced order models based on proper orthogonal decomposition. In Proceedings of the Fluids 2000 Conference and Exhibit, Denver, CO, USA, 19–22 June 2000; p. 2545.
61. Efe, M.O.; Ozbay, H. Proper orthogonal decomposition for reduced order modeling: 2D heat flow. In Proceedings of the 2003 IEEE Conference on Control Applications, (CCA 2003), Istanbul, Turkey, 25–25 June 2003; Volume 2, pp. 1273–1277.
62. Urban, K.; Patera, A. An improved error bound for reduced basis approximation of linear parabolic problems. *Math. Comput.* **2014**, *83*, 1599–1615. [\[CrossRef\]](#)
63. Yano, M.; Patera, A.T.; Urban, K. A space-time hp-interpolation-based certified reduced basis method for Burgers' equation. *Math. Model. Methods Appl. Sci.* **2014**, *24*, 1903–1935. [\[CrossRef\]](#)
64. Yano, M. A space-time Petrov–Galerkin certified reduced basis method: Application to the Boussinesq equations. *SIAM J. Sci. Comput.* **2014**, *36*, A232–A266. [\[CrossRef\]](#)
65. Baumann, M.; Benner, P.; Heiland, J. Space-time Galerkin POD with application in optimal control of semilinear partial differential equations. *SIAM J. Sci. Comput.* **2018**, *40*, A1611–A1641. [\[CrossRef\]](#)
66. Towne, A.; Schmidt, O.T.; Colonius, T. Spectral proper orthogonal decomposition and its relationship to dynamic mode decomposition and resolvent analysis. *J. Fluid Mech.* **2018**, *847*, 821–867. [\[CrossRef\]](#)
67. Towne, A. Space-time Galerkin projection via spectral proper orthogonal decomposition and resolvent modes. In Proceedings of the AIAA Scitech 2021 Forum, San Diego, CA, USA, 3–7 January 2021; p. 1676.
68. Towne, A.; Lozano-Durán, A.; Yang, X. Resolvent-based estimation of space-time flow statistics. *J. Fluid Mech.* **2020**, *883*. [\[CrossRef\]](#)
69. Choi, Y.; Carlberg, K. Space-Time Least-Squares Petrov–Galerkin Projection for Nonlinear Model Reduction. *SIAM J. Sci. Comput.* **2019**, *41*, A26–A58. [\[CrossRef\]](#)
70. Parish, E.J.; Carlberg, K.T. Windowed least-squares model reduction for dynamical systems. *J. Comput. Phys.* **2021**, *426*, 109939. [\[CrossRef\]](#)
71. Shimizu, Y.S.; Parish, E.J. Windowed space-time least-squares Petrov-Galerkin method for nonlinear model order reduction. *arXiv* **2020**, arXiv:2012.06073.
72. Choi, Y.; Brown, P.; Arrighi, W.; Anderson, R.; Huynh, K. Space-time reduced order model for large-scale linear dynamical systems with application to Boltzmann transport problems. *J. Comput. Phys.* **2020**, *424*, 109845. [\[CrossRef\]](#)
73. Barone, M.F.; Kalashnikova, I.; Segalman, D.J.; Thornquist, H.K. Stable Galerkin reduced order models for linearized compressible flow. *J. Comput. Phys.* **2009**, *228*, 1932–1946. [\[CrossRef\]](#)
74. Rezaian, E.; Wei, M. A hybrid stabilization approach for reduced-order models of compressible flows with shock-vortex interaction. *Int. J. Numer. Methods Eng.* **2020**, *121*, 1629–1646. [\[CrossRef\]](#)
75. Carlberg, K.; Bou-Mosleh, C.; Farhat, C. Efficient nonlinear model reduction via a least-squares Petrov–Galerkin projection and compressive tensor approximations. *Int. J. Numer. Methods Eng.* **2011**, *86*, 155–181. [\[CrossRef\]](#)
76. Huang, C.; Wentland, C.R.; Duraisamy, K.; Merkle, C. Model reduction for multi-scale transport problems using structure-preserving least-squares projections with variable transformation. *arXiv* **2020**, arXiv:2011.02072.

77. Yoon, G.H. Structural topology optimization for frequency response problem using model reduction schemes. *Comput. Methods Appl. Mech. Eng.* **2010**, *199*, 1744–1763. [[CrossRef](#)]
78. Amir, O.; Stolpe, M.; Sigmund, O. Efficient use of iterative solvers in nested topology optimization. *Struct. Multidiscip. Optim.* **2010**, *42*, 55–72. [[CrossRef](#)]
79. Amsallem, D.; Zahr, M.; Choi, Y.; Farhat, C. Design optimization using hyper-reduced-order modelsvd. *Struct. Multidiscip. Optim.* **2015**, *51*, 919–940. [[CrossRef](#)]
80. Gogu, C. Improving the efficiency of large scale topology optimization through on-the-fly reduced order model construction. *Int. J. Numer. Methods Eng.* **2015**, *101*, 281–304. [[CrossRef](#)]
81. Choi, Y.; Boncoraglio, G.; Anderson, S.; Amsallem, D.; Farhat, C. Gradient-based constrained optimization using a database of linear reduced-order models. *J. Comput. Phys.* **2020**, *423*, 109787. [[CrossRef](#)]
82. Choi, Y.; Oxberry, G.; White, D.; Kirchdoerfer, T. Accelerating design optimization using reduced order models. *arXiv* **2019**, arXiv:1909.11320.
83. White, D.A.; Choi, Y.; Kudo, J. A dual mesh method with adaptivity for stress-constrained topology optimization. *Struct. Multidiscip. Optim.* **2020**, *61*, 749–762. [[CrossRef](#)]
84. Najm, H.N. Uncertainty quantification and polynomial chaos techniques in computational fluid dynamics. *Annu. Rev. Fluid Mech.* **2009**, *41*, 35–52. [[CrossRef](#)]
85. Walters, R.W.; Huyse, L. *Uncertainty Analysis for Fluid Mechanics with Applications*; Technical Report; National Aeronautics and Space Administration Hampton va Langley Research Center: Hampton, VA, USA, 2002.
86. Zang, T.A. *Needs and Opportunities for Uncertainty-Based Multidisciplinary Design Methods for Aerospace Vehicles*; National Aeronautics and Space Administration, Langley Research Center: Hampton, VA, USA, 2002.
87. Petersson, N.A.; Garcia, F.M.; Copeland, A.E.; Rydin, Y.L.; DuBois, J.L. Discrete Adjoints for Accurate Numerical Optimization with Application to Quantum Control. *arXiv* **2020**, arXiv:2001.01013.
88. Choi, Y.; Farhat, C.; Murray, W.; Saunders, M. A practical factorization of a Schur complement for PDE-constrained distributed optimal control. *J. Sci. Comput.* **2015**, *65*, 576–597. [[CrossRef](#)]
89. Choi, Y. Simultaneous Analysis and Design in PDE-Constrained Optimization. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2012.
90. Sirovich, L. Turbulence and the dynamics of coherent structures. I. Coherent structures. *Q. Appl. Math.* **1987**, *45*, 561–571. [[CrossRef](#)]
91. Lee, K.; Carlberg, K.T. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders. *J. Comput. Phys.* **2020**, *404*, 108973. [[CrossRef](#)]
92. Kim, Y.; Choi, Y.; Widemann, D.; Zohdi, T. A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder. *arXiv* **2020**, arXiv:2009.11990.
93. Kim, Y.; Choi, Y.; Widemann, D.; Zohdi, T. Efficient nonlinear manifold reduced order model. *arXiv* **2020**, arXiv:2011.07727.
94. McBane, S.; Choi, Y. Component-wise reduced order model lattice-type structure design. *Comput. Methods Appl. Mech. Eng.* **2021**, *381*, 113813. [[CrossRef](#)]