Innovations in Reduced-Order Modeling: A Comprehensive Study on the 1D Burgers Equation

S. Ares de Parga[a,b,*]

[a] *Department of Civil and Environmental Engineering, Universitat Politècnica de Catalunya, Building B0, Campus Nord, Jordi Girona 1-3, Barcelona 08034, Spain*

[b] *Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE), Universitat Politècnica de Catalunya, Building C1, Campus Nord, Jordi Girona 1-3, Barcelona 08034, Spain*

Keywords: Reduced Order Modeling

1. Introduction

Projection-based reduced order models (PROMs) can significantly reduce the time and storage required for the solution of high-fidelity (high-dimensional) discretized PDE-based computational models, also known as full-order models (FOMs). PROMs are particularly valuable in applications where time-critical models, such as digital twins, design optimization, and control problem optimization, are needed. These models project the high-dimensional system onto a lower-dimensional subspace, enabling rapid and efficient simulations while maintaining a high level of accuracy.

The creation of a PROM typically involves an offline-online decomposition strategy. In the offline phase, computationally intensive tasks are performed, such as generating a reduced-order basis (ROB) through methods like Proper Orthogonal Decomposition (POD) [**? ?**]. In the online phase, the precomputed ROB is used to rapidly project the FOM onto the reduced subspace, allowing for efficient simulations [**? ?**]. This approach has been shown to be effective for a variety of linear and nonlinear problems, particularly when the Kolmogorov n-width decays rapidly, as in the case of smooth solution manifolds [**? ?**].

In this review paper, we aim to provide an overall comparison of the state-of-the-art reduced order modeling techniques, with a primary focus on intrusive models. These models, which require access to the underlying system equations, offer high accuracy by closely interacting with the full-order model's computational framework. We will also compare these intrusive methods with some non-intrusive, data-driven models that do not require access to the underlying physics but instead rely on historical data and advanced algorithms to approximate the system's behavior.

Specifically, we will discuss various projection-based ROM techniques, such as Proper Orthogonal Decomposition (POD), local-POD, and Quadratic Approximation Manifolds, as well as recent advancements like POD-ANN (Neural-Network-Augmented) and nonlinear manifold approximations using dense and convolutional autoencoders. Furthermore, we will explore both Galerkin and Least-Squares Petrov-Galerkin (LSPG) projection methods and highlight the strengths and weaknesses of each approach in different application scenarios.

By providing a detailed comparison of these techniques, this review paper aims to serve as a comprehensive guide for researchers and practitioners looking to select the most appropriate reduced order modeling approach for their specific applications.

2. Strong Form of the 1D Burgers' Equation

The 1D Burgers' equation with viscosity and a source term is given by:

$$\frac{\partial u(x,t)}{\partial t} + u(x,t)\frac{\partial u(x,t)}{\partial x} - \nu\frac{\partial^2 u(x,t)}{\partial x^2} = f(x,t), \quad \forall x \in [0,L],\ \forall t \in [0,T] \tag{1}$$

where:

- $u(x,t)$ is the velocity field.

*Corresponding author
Email address: sebastian.ares@upc.edu (S. Ares de Parga)

- ν is the kinematic viscosity.
- $f(x,t)$ is the source term.
- L is the length of the spatial domain.
- T is the final time.

2.1. Boundary and Initial Conditions

The equation is subject to the following boundary and initial conditions:

$$\begin{aligned} u(0,t) &= u_{\text{left}}(t) = \mu_1, \quad && \forall t \in [0,T], \\ u(x,0) &= u_{\text{initial}}(x) = 1, && \forall x \in [0,L]. \end{aligned} \tag{2}$$

2.2. Source Term

In the specific case we are considering, the source term $f(x,t)$ is given by:

$$f(x,t) = 0.02e^{\mu_2 x} \tag{3}$$

where μ_2 is a parameter that defines the spatial variation of the source term.

3. Weak Form of the 1D Burgers' Equation

To derive the weak form of the 1D Burgers' equation, we start with the strong form:

$$\frac{\partial u(x,t)}{\partial t} + u(x,t)\frac{\partial u(x,t)}{\partial x} = \nu \frac{\partial^2 u(x,t)}{\partial x^2} + f(x,t) \tag{4}$$

Multiplying by a Test Function and Integrating Over the Domain

We multiply by a test function $v(x)$ and integrate over the domain $[0,L]$:

$$\int_0^L \left(\frac{\partial u(x,t)}{\partial t} + u(x,t)\frac{\partial u(x,t)}{\partial x} - \nu \frac{\partial^2 u(x,t)}{\partial x^2} - f(x,t) \right) v(x)\, dx = 0 \tag{5}$$

Applying Integration by Parts

We handle the second-order derivative term $-\nu \frac{\partial^2 u}{\partial x^2}$ using integration by parts:

$$-\nu \int_0^L \frac{\partial^2 u}{\partial x^2} v\, dx = \left[-\nu \frac{\partial u}{\partial x} v \right]_0^L + \nu \int_0^L \frac{\partial u}{\partial x} \frac{\partial v(x)}{\partial x}\, dx \tag{6}$$

Assuming homogeneous Dirichlet boundary conditions (i.e., $u = 0$ on $\partial\Omega$), the boundary term vanishes:

$$\nu \frac{\partial u}{\partial x} v \Big|_0^L = 0 \tag{7}$$

Thus, we obtain:

$$\int_0^L \left(\frac{\partial u}{\partial t} v + u\frac{\partial u}{\partial x} v + \nu \frac{\partial u}{\partial x}\frac{\partial v(x)}{\partial x} - f(x,t)v \right) dx = 0 \tag{8}$$

The weak form of the 1D Burgers' equation with viscosity and a source term is:

$$\int_0^L \left(\frac{\partial u}{\partial t} v + u\frac{\partial u}{\partial x} v + \nu \frac{\partial u}{\partial x}\frac{\partial v(x)}{\partial x} \right) dx = \int_0^L f(x,t) v\, dx \tag{9}$$

This equation must be satisfied for all test functions $v(x)$ in the appropriate function space.

4. Finite Element Discretization

4.1. Discretizing the Domain

Discretize the domain $[0, L]$ into E finite elements with nodes x_i, where $i = 1, 2, \ldots, n$ and n is the total number of nodes.

4.2. Approximating the Solution and Test Functions

Approximate the solution $u(x,t)$ and the test function $v(x)$ using finite element basis functions $N_i(x)$:

$$u(x,t) \approx \sum_{j=1}^{n} U_j(t) N_j(x), \quad v(x) = N_i(x) \tag{10}$$

Here, $U_j(t)$ are the time-dependent coefficients (nodal values) of the solution, and $N_j(x)$ are the shape functions associated with each node.

4.3. Substituting into the Weak Form

Substitute these approximations into the weak form and simplify:

$$\sum_{j=1}^{n} \frac{dU_j(t)}{dt} \int_0^L N_j(x) N_i(x) dx + \sum_{j=1}^{n} U_j(t) \int_0^L U(x,t) \frac{\partial N_j(x)}{\partial x} N_i(x) dx + \nu \sum_{j=1}^{n} U_j(t) \int_0^L \frac{\partial N_j(x)}{\partial x} \frac{\partial N_i(x)}{\partial x} dx = \int_0^L f(x,t) N_i(x) dx \tag{11}$$

4.4. Defining the Matrices

This can be rewritten in matrix form:

- **Mass Matrix M:**

$$\mathbf{M}_{ij} = \int_0^L N_i(x) N_j(x) dx \tag{12}$$

- **Convection Term C(U):**

$$\mathbf{C}_{ij}(\mathbf{U}) = \int_0^L U(x,t) \frac{\partial N_j(x)}{\partial x} N_i(x) dx \tag{13}$$

- **Diffusion Matrix K:**

$$\mathbf{K}_{ij} = \nu \int_0^L \frac{\partial N_i(x)}{\partial x} \frac{\partial N_j(x)}{\partial x} dx \tag{14}$$

- **Load Vector F:**

$$\mathbf{F}_i(t) = \int_0^L f(x,t) N_i(x) dx \tag{15}$$

4.5. Discretized System of Equations

The overall system of equations is:

$$\mathbf{M} \frac{d\mathbf{U}(t)}{dt} + \mathbf{C}(\mathbf{U})\mathbf{U} + \mathbf{K}\mathbf{U} = \mathbf{F}(t) \tag{16}$$

where:

- $\mathbf{U}(t)$ is the vector of unknowns at the nodes.

4.6. Time Discretization

To solve this system over time, we apply a time discretization method such as the implicit Euler scheme:

$$\mathbf{M} \frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} + \mathbf{C}(\mathbf{U}^{n+1})\mathbf{U}^{n+1} + \mathbf{K}\mathbf{U}^{n+1} = \mathbf{F}^{n+1} \tag{17}$$

This forms a nonlinear system at each time step, which can be solved iteratively.

4.7. Defining the Residual

The nonlinear system at each time step can be rearranged to define a residual $\mathbf{R}(\mathbf{U}^{n+1})$ that we seek to minimize. The residual is given by:

$$\mathbf{R}(\mathbf{U}^{n+1}) = \mathbf{M}\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} + \mathbf{C}(\mathbf{U}^{n+1})\mathbf{U}^{n+1} + \mathbf{K}\mathbf{U}^{n+1} - \mathbf{F}^{n+1}. \tag{18}$$

Equivalently, this can be expressed as:

$$\mathbf{R}(\mathbf{U}^{n+1}) = \mathbf{A}(\mathbf{U}^{n+1})\mathbf{U}^{n+1} - \mathbf{b}^{n+1}, \tag{19}$$

where:

$$\mathbf{A}(\mathbf{U}^{n+1}) = \mathbf{M} + \Delta t\mathbf{C}(\mathbf{U}^{n+1}) + \Delta t\mathbf{K} \tag{20}$$

$$\mathbf{b}^{n+1} = \mathbf{M}\mathbf{U}^n + \Delta t\mathbf{F}^{n+1} \tag{21}$$

The residual $\mathbf{R}(\mathbf{U}^{n+1})$ represents the difference between the left-hand side and the right-hand side of the discretized equation, and our goal is to find $\mathbf{U}^{n+1}$ such that $\mathbf{R}(\mathbf{U}^{n+1}) = 0$.

4.7.1. Residual Minimization via Picard Iteration

To solve this nonlinear system iteratively, we use a Picard iteration method, which iteratively refines the solution by minimizing the residual. The steps are as follows:

1. **Initial Guess:** Start with an initial guess $\mathbf{U}_0^{n+1}$, typically taken as the solution from the previous time step, $\mathbf{U}^n$.
2. **Compute the Residual:** At each iteration k, compute the residual based on the current approximation:

$$\mathbf{R}(\mathbf{U}_k^{n+1}) = \mathbf{M}\frac{\mathbf{U}_k^{n+1} - \mathbf{U}^n}{\Delta t} + \mathbf{C}(\mathbf{U}_k^{n+1})\mathbf{U}_k^{n+1} + \mathbf{K}\mathbf{U}_k^{n+1} - \mathbf{F}^{n+1} \tag{22}$$

3. **Solve for the Correction:** Solve for the correction $\delta\mathbf{U}_{k+1}^{n+1}$ that reduces the residual:

$$\mathbf{A}(\mathbf{U}_k^{n+1})\delta\mathbf{U}_{k+1}^{n+1} = -\mathbf{R}(\mathbf{U}_k^{n+1}) \tag{23}$$

   where $\mathbf{A}(\mathbf{U}_k^{n+1})$ is the linearized system matrix evaluated using the current guess $\mathbf{U}_k^{n+1}$.
4. **Update the Solution:** Update the solution using the computed correction:

$$\mathbf{U}_{k+1}^{n+1} = \mathbf{U}_k^{n+1} + \delta\mathbf{U}_{k+1}^{n+1} \tag{24}$$

5. **Convergence Check:** Compute the error between successive iterations:

$$\text{error}_U = \frac{\|\delta\mathbf{U}_{k+1}^{n+1}\|}{\|\mathbf{U}_{k+1}^{n+1}\|} \tag{25}$$

6. **Stopping Criterion:** The iteration continues until the error falls below a specified tolerance, or a maximum number of iterations is reached.

This iterative process continues until the residual $\mathbf{R}(\mathbf{U}^{n+1})$ is sufficiently small, indicating that the solution $\mathbf{U}^{n+1}$ satisfies the nonlinear system within the desired accuracy.

4.7.2. Residual Minimization via Newton-Raphson

Alternatively, the nonlinear system can be solved using the Newton-Raphson method, which is another iterative approach that directly utilizes the Jacobian matrix for faster convergence.

Given the residual $\mathbf{R}(\mathbf{U}^{n+1})$, the Newton-Raphson method seeks to find the solution $\mathbf{U}^{n+1}$ that satisfies:

$$\mathbf{R}(\mathbf{U}^{n+1}) = 0 \tag{26}$$

This is achieved by iteratively solving the following linearized system at each iteration k:

$$\mathbf{J}(\mathbf{U}_{n+1}^{k})\delta\mathbf{U} = -\mathbf{R}(\mathbf{U}_{n+1}^{k}), \tag{27a}$$

$$\mathbf{U}_{n+1}^{k+1} = \mathbf{U}_{n+1}^{k} + \delta\mathbf{U}, \tag{27b}$$

where:

- $\mathbf{J}(\mathbf{U}_{n+1}^{k}) = \frac{\partial \mathbf{R}}{\partial \mathbf{U}}$ is the Jacobian matrix, representing the derivative of the residual with respect to the solution.
- $\delta\mathbf{U}$ is the correction term that adjusts the current solution estimate $\mathbf{U}_{n+1}^{k}$.

In comparison, the Picard iteration method can be viewed as a special case of the Newton-Raphson method where the Jacobian matrix is approximated by the system matrix $\mathbf{A}(\mathbf{U})$. Specifically, this can be expressed as:

$$\mathbf{J}(\mathbf{U}) \approx \mathbf{A}(\mathbf{U}), \tag{28}$$

where $\mathbf{J}(\mathbf{U})$ is the exact Jacobian, and $\mathbf{A}(\mathbf{U})$ is the system matrix used in the Picard iteration.

This approximation simplifies the iteration process and generally makes Picard iteration more robust, especially for initial guesses that are far from the true solution. However, this robustness often comes at the cost of slower convergence compared to the Newton-Raphson method, which utilizes the exact Jacobian.

The Picard iteration is also computationally less expensive, as it avoids the need to calculate the full Jacobian matrix. Therefore, while Newton-Raphson offers faster convergence, Picard iteration is preferred when computational efficiency is a priority, or when the problem's nonlinearity is moderate and robustness is a concern.

From now on, for simplicity of exposition, I will refer to the left-hand side matrix as $\mathbf{J}$, even when it is an approximation of the Jacobian as in the Picard method, although the results discussed here are obtained using the Picard method.

4.8. Boundary Conditions

Boundary conditions are crucial in ensuring that the numerical solution adheres to the physical constraints of the problem.

Dirichlet Boundary Condition at $x = 0$. For the 1D Burgers' equation, the Dirichlet boundary condition is:

$$u(0, t) = \mu_1, \quad \forall t \in [0, T] \tag{29}$$

To enforce this in the FEM system:

- **Modify the system matrix J**:

$$\mathbf{J}[0, :] = 0 \quad \text{and} \quad \mathbf{J}_{00} = 1 \tag{30}$$

  This sets the first row of $\mathbf{J}$ to ensure the solution at $x = 0$ is fixed.

- **Adjust the vector b**:

$$\mathbf{b}[0] = \mu_1 \tag{31}$$

  This forces the solution at the boundary node to be μ_1.

Initial Condition. The initial condition is:

$$u(x, 0) = 1, \quad \forall x \in [0, L] \tag{32}$$

This initializes the solution vector $\mathbf{U}(0)$ at time $t = 0$.

5. Reduced Order Modeling (ROM)

Let us consider a fully discrete set of governing equations in the form of a $\boldsymbol{\mu}-$parametric operator $\mathbf{R} : \mathbb{R}^{\mathcal{N}} \times \mathcal{P} \to \mathbb{R}^{\mathcal{N}}$, as

$$\mathbf{R}(\mathbf{U}; \boldsymbol{\mu}) = \mathbf{0} \ , \tag{33}$$

where $\mathbf{U} \in \mathbb{R}^{\mathcal{N}}$ is the FOM solution vector containing the value of the solution on every degree of freedom of the spatial discretization, and $\boldsymbol{\mu} \in \mathcal{P} \subset \mathbb{R}^p$ is the parameter vector encapsulating, for example, geometrical variations, material properties, or boundary conditions.

For emphasis, let us reprint the iteration method used to obtain the solution at the next step, given the FOM solution at step n, symbolically represented as $\mathbf{U}_n$:

$$\mathbf{J}(\mathbf{U}^k_{n+1})\delta\mathbf{U} = \mathbf{R}(\mathbf{U}^k_{n+1}; \boldsymbol{\mu}) \tag{34a}$$

$$\mathbf{U}^{k+1}_{n+1} = \mathbf{U}^k_{n+1} + \delta\mathbf{U}, \tag{34b}$$

where:

- $\mathbf{U} \in \mathbb{R}^{\mathcal{N}}$ is the solution vector at the current step.
- $\mathbf{J} = \frac{\partial \mathbf{R}}{\partial \mathbf{U}} \in \mathbb{R}^{\mathcal{N} \times \mathcal{N}}$ is the Jacobian matrix (or its approximation).
- $\delta\mathbf{U} \in \mathbb{R}^{\mathcal{N}}$ is the differential increment.
- k is the current iterate.

Solving this linear system of equations can be computationally expensive, especially for large-scale problems where the dimension $\mathcal{N}$ is very high. This is where reduced-order modeling becomes crucial, as it aims to achieve significant computational efficiency by reducing the dimensionality of the problem while preserving the essential dynamics of the original system.

5.1. Manifold Projection-Based ROMs

We begin by approximating the solution state variable through a general decoder function defined as:

$$\mathbf{U} \approx D_u(\mathbf{q}), \tag{35}$$

where $D_u : \mathbb{R}^k \to \mathbb{R}^{\mathcal{N}}$ maps $\mathbf{q}$ to $\mathbf{U}$.

Correspondingly, we introduce its encoder:

$$E_u : \mathbb{R}^{\mathcal{N}} \to \mathbb{R}^k \qquad \mathbf{U} \mapsto \mathbf{q}. \tag{36}$$

When Eq.35 is substituted into Eq.33, we obtain a residual characterized by:

$$\mathbf{R}(\mathbf{U}; \boldsymbol{\mu}) = \mathbf{R}(D_u(\mathbf{q}); \boldsymbol{\mu}). \tag{37}$$

We now aim to minimize the non-linear residual in some norm $\mathbf{G}$ in order to meet the minimum-residual optimality of the projection approximation. Specifically, we seek to solve the following minimization problem:

$$\min_{\mathbf{q} \in \mathbb{R}^n} \frac{1}{2} \left\| \mathbf{R}(D_u(\mathbf{q}); \boldsymbol{\mu}) \right\|^2_{\mathbf{G}} \tag{38}$$

where $\| \cdot \|_{\mathbf{G}}$ denotes a norm defined by a symmetric, positive definite (SPD) matrix $\mathbf{G} \in \mathbb{R}^{N \times N}$. Alternatively, we can write the above problem as follows:

$$\min_{\mathbf{q} \in \mathbb{R}^n} \frac{1}{2} \mathbf{R}(D_u(\mathbf{q}))^T \mathbf{G} \mathbf{R}(D_u(\mathbf{q})). \tag{39}$$

By satisfying the minimum-residual optimality of Eq.38, a general relationship between projection-based reduced-order models and minimum-residual reduced-order models can be established, as follows:

$$\begin{aligned}\left[\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}\right]^T \mathbf{G}\mathbf{R} = \mathbf{0} \\ \left[\mathbf{J}\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}\right]^T \mathbf{G}\mathbf{R} = \mathbf{0}.\end{aligned} \tag{40}$$

This version of the projected residual can be seen as a general encoder of the residual, that is:

$$E_r(\mathbf{R}) = \mathbf{\Psi}^T\mathbf{R} = \frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}^T\mathbf{G}\mathbf{R} \tag{41}$$

We can then propose to minimize the following expression depending on small dimensional quantities $\mathbf{r} \in \mathbb{R}^\ell$, and $\mathbf{q} \in \mathbb{R}^k$:

$$\mathbf{r}(\mathbf{q};\boldsymbol{\mu}) = E_r\left(\mathbf{R}(D_u(\mathbf{q});\boldsymbol{\mu})\right) = \mathbf{0} \tag{42}$$

where $\mathbf{r}(\mathbf{q};\boldsymbol{\mu}) := E_r\left(\mathbf{R}(D_u(\mathbf{q})\,;\boldsymbol{\mu})\right)$. Moreover, given a ROM solution at a step n, symbolically represented as $\tilde{\mathbf{U}}_n \in \mathbb{R}^{\mathcal{N}}$, the solution corresponding to the next step can be obtained via a fixed-point iteration method like

$$\frac{\partial \mathbf{r}}{\partial \mathbf{q}}\delta\mathbf{q} = -\mathbf{r}, \tag{43}$$

$$\frac{\partial E_r(\mathbf{R})}{\partial \mathbf{R}}\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\frac{\partial D_u(\mathbf{q})}{\partial \mathbf{q}}\delta\mathbf{q}^k = -E_r\left(\mathbf{R}\right) \tag{44a}$$

$$\mathbf{q}^{k+1} = \mathbf{q}^k + \delta\mathbf{q}\ , \tag{44b}$$

Elaborating on equation 44a given the definition of the encoder of the residual in eq. 41, we obtain:

$$\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}^T\mathbf{G}\mathbf{J}\frac{\partial D_u(\mathbf{q})}{\partial \mathbf{q}}\delta\mathbf{q}^k = -\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}^T\mathbf{G}\mathbf{R} \tag{45}$$

5.1.1. Galerkin Projection

The Galerkin projection is particularly effective when dealing with symmetric positive definite (SPD) Jacobians. In such scenarios, it yields search directions that are minimum-residual optimal. This optimality is achieved with $\mathbf{G} = \mathbf{J}^{-T}$, leading to the following formulation:

$$\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}\frac{\partial D_u(\mathbf{q})}{\partial \mathbf{q}}\delta\mathbf{q}^k = -\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{R} \tag{46}$$

5.1.2. Least-Squares Petrov-Galerkin (LSPG) Projection

In nonlinear problems, Jacobians often are not SPD. The LSPG method addresses this by setting $\mathbf{G} = \mathbf{I}$, the identity matrix. This choice ensures that the LSPG method maintains the minimum-residual property, crucial for the stability and accuracy of the solution. The projection equation in this context is:

$$\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}^T\mathbf{J}\frac{\partial D_u(\mathbf{q})}{\partial \mathbf{q}}\delta\mathbf{q}^k = -\frac{\partial (D_u(\mathbf{q}))}{\partial \mathbf{q}}^T \mathbf{J}^T\mathbf{R} \tag{47}$$

6. Methods for Constructing the Decoder Function

In the context of Reduced Order Modeling (ROM), the decoder function $D_U(\mathbf{q})$ plays a crucial role in mapping the reduced coordinates $\mathbf{q}$ to the full-order space $\mathbf{U}$. The effectiveness of a ROM largely depends on the choice of this decoder function. There are several methods available for constructing $D_U(\mathbf{q})$, ranging from traditional linear approaches to more advanced nonlinear techniques. Each method offers different trade-offs in terms of accuracy, computational efficiency, and applicability to various types of problems.

Before diving into the specific methods for constructing the decoder function, it is essential to discuss how the data, or snapshots, used in these methods are generated. The quality and representativeness of these snapshots are critical for the success of any ROM approach, as they directly influence the accuracy and robustness of the resulting reduced-order model.

6.1. Parametric Exploration and Snapshot Collection

To build an effective decoder function, we first need a diverse set of solution snapshots from the full-order model (FOM). These snapshots are collected by exploring the parameter space $\boldsymbol{\mu}$, which may represent physical properties, boundary conditions, or other problem-specific parameters.

The parameters μ_1 and μ_2 are varied within specified ranges to observe their influence on the solution $u(x,t)$. The solutions are recorded at multiple time steps to capture the dynamic behavior of the system.

Figure 1 illustrates the effect of varying μ_1 while keeping μ_2 fixed. The 3D plot provides a comprehensive view of how changes in μ_1 impact the solution profile across the spatial domain over time. Different time steps are represented by different positions along the depth axis, offering insights into the temporal evolution of the solution.

Figure 2 shows the effect of varying μ_2 while μ_1 is held constant. Similar to the previous figure, this plot demonstrates the solution's sensitivity to changes in μ_2, which can be crucial for understanding the system's response under different conditions.

To provide a more detailed view of the solution evolution over time, Figures 3 and 4 present the solution profiles at specific time steps (e.g., $t = 5$, $t = 10$, $t = 15$, and $t = 20$). These subplots highlight the differences in solution behavior at different times, allowing for a clear comparison between the effects of varying μ_1 and μ_2.

These figures collectively serve as a foundation for understanding how the parameter space $\boldsymbol{\mu}$ influences the solution dynamics, which is critical for constructing an accurate and robust reduced-order model.

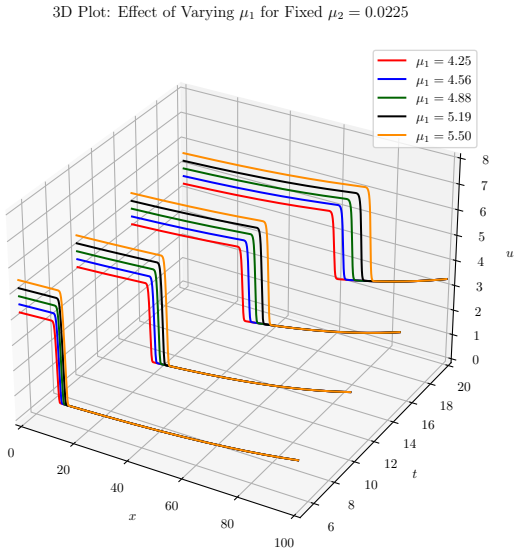

Figure 1: 3D plot showing the effect of varying μ_1 on the solution $u(x,t)$ over time, with μ_2 fixed at 0.0225.

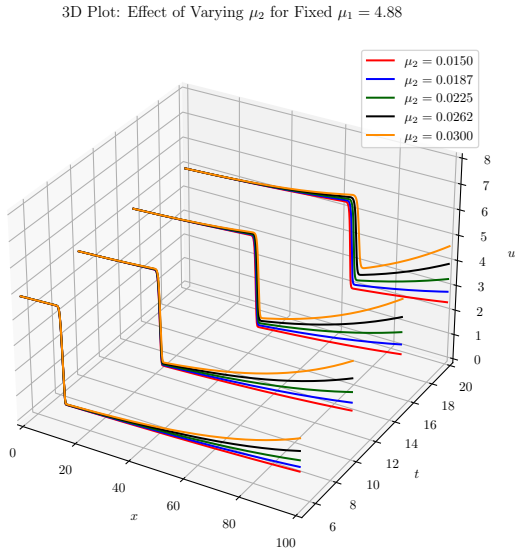

Figure 2: 3D plot illustrating the effect of varying μ_2 on the solution $u(x,t)$ over time, with μ_1 fixed at 4.88.

6.1.1. Latin Hypercube Sampling (LHS)

Latin Hypercube Sampling (LHS) is employed to generate a well-distributed set of parameter values $\boldsymbol{\mu}$ across the defined parameter space. In our study, the parameters μ_1 and μ_2 are varied within the ranges $[4.25, 5.5]$ and $[0.015, 0.03]$, respectively. LHS ensures that even with a relatively small number of samples, the parameter space is efficiently and thoroughly explored, providing a comprehensive set of scenarios for which the FOM is solved.

We performed 500 LHS samples, generating a series of parameter vectors, $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_{500}$, where each vector represents a unique combination of parameter values within the specified ranges. These parameter vectors are then used to run FOM simulations, yielding the corresponding solution snapshots.

6.1.2. Snapshot Collection

Once the LHS-generated parameters are defined, the FOM is solved for each parameter set $\boldsymbol{\mu}_p$ over a specified time interval. The solutions at different time steps for each $\boldsymbol{\mu}_p$ are collected and organized into a snapshot matrix $\mathbf{S}$. This matrix contains the full-order solutions that capture the system's behavior under various conditions, forming the foundation for constructing the decoder function.

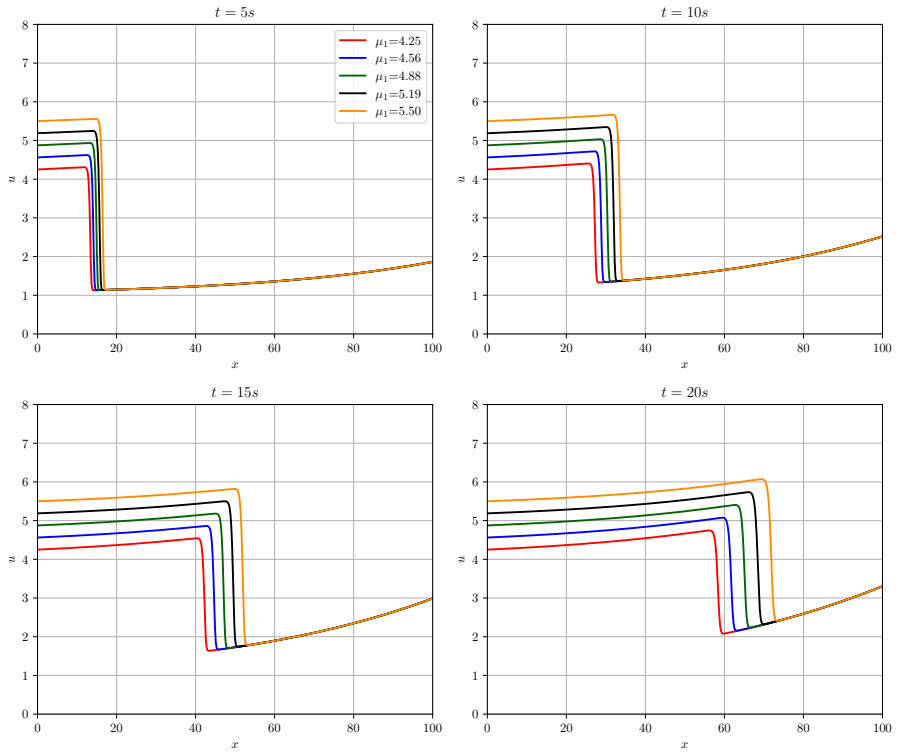

Figure 3: Subplots displaying the solution profiles at different time steps for varying μ_1 values, with μ_2 fixed at 0.0225.

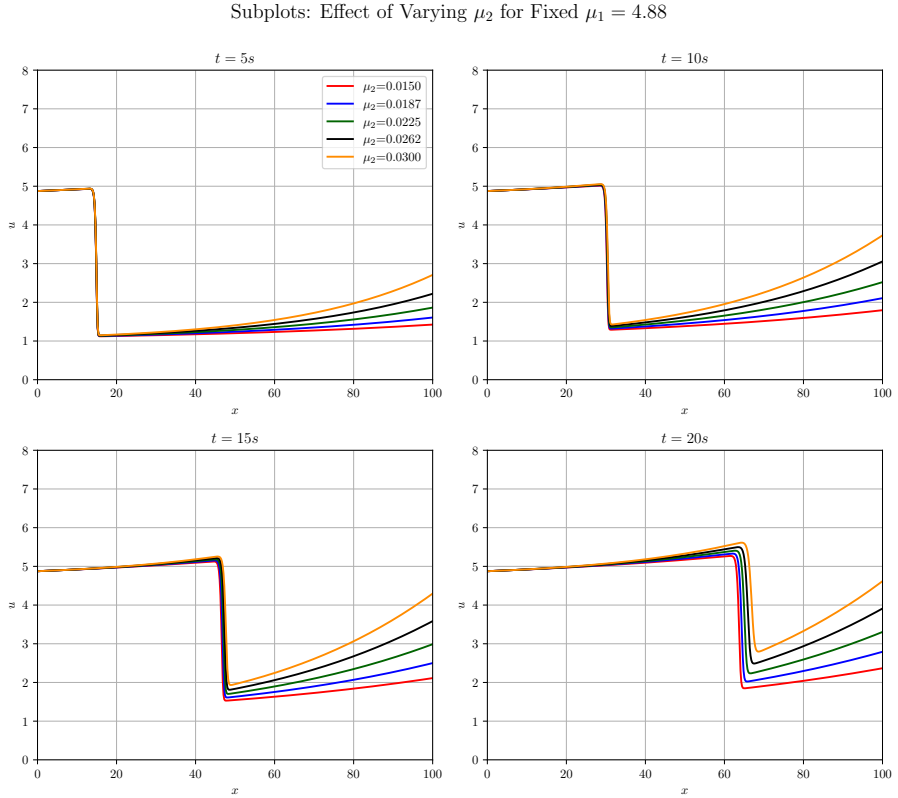

Figure 4: Subplots showing the solution profiles at different time steps for varying μ_2 values, with μ_1 fixed at 4.88.

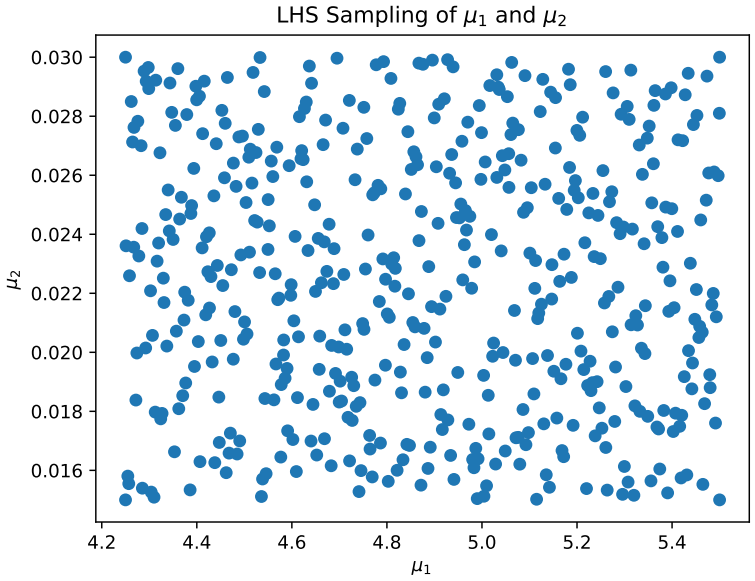

Figure 5: Latin Hypercube Sampling (LHS) of parameters μ_1 and μ_2. The plot shows the distribution of sampling points used to generate the solution snapshots. The corners represent extreme cases, while the inner points represent intermediate parameter combinations.

6.2. Proper Orthogonal Decomposition (POD)

With the snapshot matrix $\mathbf{S}$ prepared, we can proceed to construct the decoder function using Proper Orthogonal Decomposition (POD). POD is a classical method that involves performing a Singular Value Decomposition (SVD) on the snapshot matrix to identify the most energetically significant modes of the system. The matrix $\mathbf{\Phi}$ represents the optimal basis that captures the dominant modes of the system's behavior (see Figure 6).

6.2.1. Snapshot Collection and Basis Construction

The snapshot matrix $\mathbf{S}$ is defined as:

$$\mathbf{S} = \left[\mathbf{U}_1(\boldsymbol{\mu}_1), \quad \mathbf{U}_2(\boldsymbol{\mu}_1), \quad \ldots, \quad \mathbf{U}_T(\boldsymbol{\mu}_1), \quad \mathbf{U}_1(\boldsymbol{\mu}_2), \quad \mathbf{U}_2(\boldsymbol{\mu}_2), \quad \ldots, \quad \mathbf{U}_T(\boldsymbol{\mu}_P)\right], \tag{48}$$

where $\mathbf{U}_t(\boldsymbol{\mu}_p)$ is the snapshot at time t for parameter $\boldsymbol{\mu}_p$.

To construct the reduced basis, we perform Singular Value Decomposition (SVD) on the snapshot matrix $\mathbf{S}$:

$$\mathbf{S} = \mathbf{\Phi}\mathbf{\Sigma}\mathbf{V}^T, \tag{49}$$

where:

- $\mathbf{\Phi} \in \mathbb{R}^{\mathcal{N}\times n}$ contains the left singular vectors, which form the reduced basis.
- $\mathbf{\Sigma} \in \mathbb{R}^{n\times n}$ is a diagonal matrix containing the singular values.
- $\mathbf{V} \in \mathbb{R}^{T\times n}$ contains the right singular vectors.

The matrix $\mathbf{\Phi}$ represents the optimal basis that captures the dominant modes of the system's behavior.

6.2.2. Truncation Based on Singular Value Tolerance

In practice, not all singular values contribute significantly to the energy of the system. To create a more efficient reduced basis, we often truncate the SVD by retaining only the singular values that exceed a certain tolerance level ϵ. The number of modes r retained is chosen such that:

$$\sum_{i=1}^{r} \sigma_i^2 \geq (1-\epsilon^2) \sum_{i=1}^{\text{rank}(\mathbf{S})} \sigma_i^2, \tag{50}$$

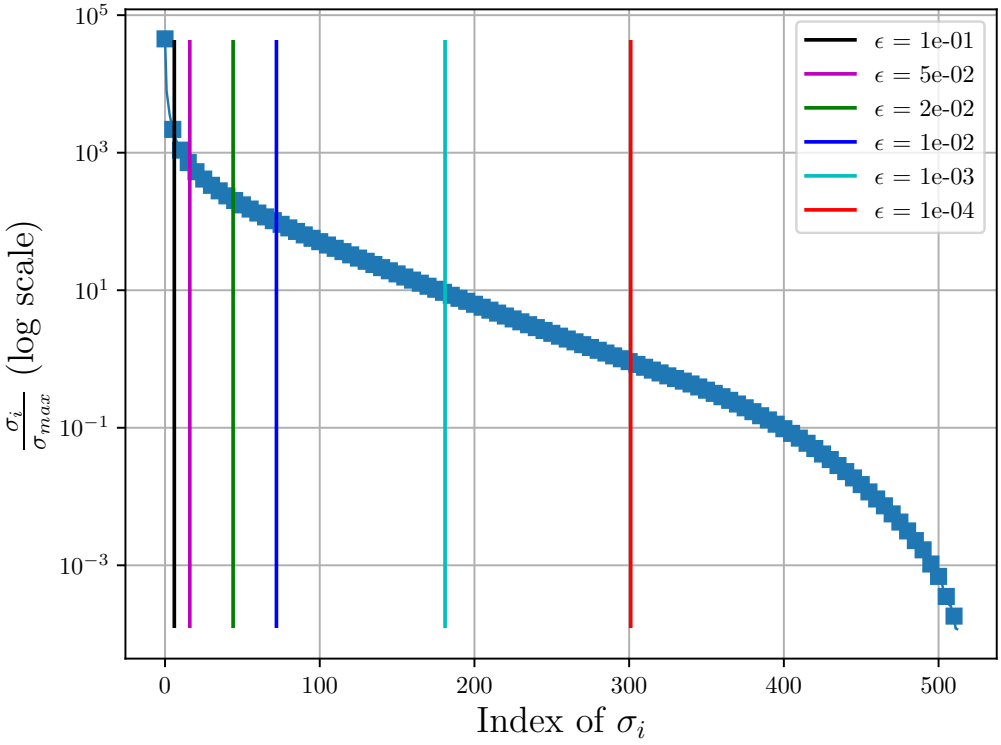

Figure 6: Singular value decay of the snapshot matrix $\mathbf{S}$. The vertical lines indicate the number of modes retained for different tolerance levels ϵ.

where σ_i are the singular values, and ϵ is the prescribed tolerance. This ensures that the retained modes capture the majority of the system's energy, while significantly reducing the dimensionality.

The truncated basis matrix $\mathbf{\Phi}$ is then constructed from the first r columns of $\mathbf{\Phi}$:

$$\mathbf{\Phi} = \mathbf{\Phi}[:, :r], \tag{51}$$

where $\mathbf{\Phi} \in \mathbb{R}^{\mathcal{N} \times r}$. This results in a reduced basis of size r, where $r \ll \mathcal{N}$, significantly reducing the computational complexity of the ROM (see Figure 7).

This reduced basis is used to approximate the full-order solution $\mathbf{U}$ by projecting it onto the reduced space:

$$\mathbf{U} \approx D_U(\mathbf{q}) = \mathbf{\Phi}\mathbf{q}, \tag{52}$$

where $\mathbf{q} \in \mathbb{R}^r$ now represents the reduced coordinates. The final size of the reduced basis is r, which is determined by the tolerance ϵ applied during the truncation process.

The derivative of the decoder function with respect to $\mathbf{q}$ is defined as:

$$\frac{\partial(D_U(\mathbf{q}))}{\partial \mathbf{q}} = \mathbf{\Phi}. \tag{53}$$

We can then substitute this derivative into equation (45) to obtain:

$$\mathbf{\Phi}^T \mathbf{J}^T \mathbf{G} \mathbf{J} \mathbf{\Phi} \delta \mathbf{q}^k = -\mathbf{\Phi}^T \mathbf{J}^T \mathbf{G} \mathbf{R}, \tag{54}$$

This equation represents the Projected Reduced Order Model (PROM) for the POD-based ROM, where the reduced coordinates $\mathbf{q}$ evolve in a lower-dimensional space $\mathbb{R}^r$ based on the reduced basis $\mathbf{\Phi}$.

Finally, one can substitute $\mathbf{G} = \mathbf{J}^{-T}$ for the Galerkin projection and $\mathbf{G} = \mathbf{I}$ for the LSPG projection. Specifically, these substitutions lead to the following forms:

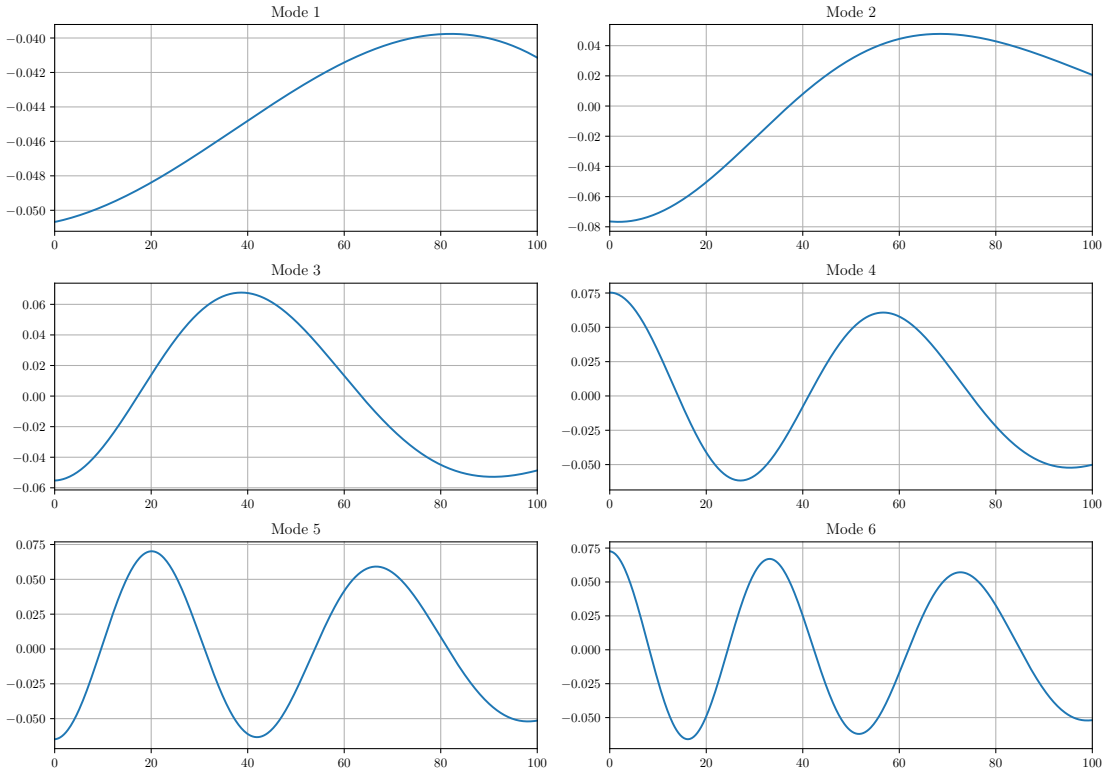

Figure 7: First six POD modes $\mathbf{\Phi}_1, \ldots, \mathbf{\Phi}_6$ obtained from the truncated SVD of the snapshot matrix. These modes form part of the reduced basis used in the POD-based ROM.

For the Galerkin projection:

$$\mathbf{\Phi}^T \mathbf{J} \mathbf{\Phi} \delta \mathbf{q}^k = -\mathbf{\Phi}^T \mathbf{R}, \tag{55}$$

For the LSPG projection:

$$\mathbf{\Phi}^T \mathbf{J}^T \mathbf{J} \mathbf{\Phi} \delta \mathbf{q}^k = -\mathbf{\Phi}^T \mathbf{J}^T \mathbf{R}. \tag{56}$$

From now on, we will make use of the LSPG method, as the Jacobian matrix is generally non-SPD (Symmetric Positive Definite).

To evaluate the performance of the POD-based reduced-order models (ROMs) using the Galerkin and LSPG projections, we consider the parameter values $\mu_1 = 4.76$ and $\mu_2 = 0.0182$. We compare the full-order model (FOM) with the ROMs for both methods at different time instances ($t = 7$, $t = 14$, and $t = 21$ seconds) for various tolerance levels.

The comparisons between the FOM and ROM solutions are illustrated in Figure 8. Each subplot shows the solutions for a specific tolerance level, with the number of modes indicated in the figure. The L2 norm errors for the Galerkin and LSPG ROMs across these tolerance levels are presented in Table 1.

Table 1: L2 Norm Errors and Number of Modes for Galerkin and LSPG ROMs at Different Tolerances

Tolerance	**Number of Modes**	**Galerkin L2 Error**	**LSPG L2 Error**
1e-1	6	1.78e-1	1.94e-1
5e-2	16	1.08e-1	1.04e-1
2e-2	44	3.42e-2	2.76e-2
1e-2	72	1.52e-2	1.24e-2
1e-3	181	1.34e-3	1.23e-3
1e-4	301	1.22e-4	1.16e-4

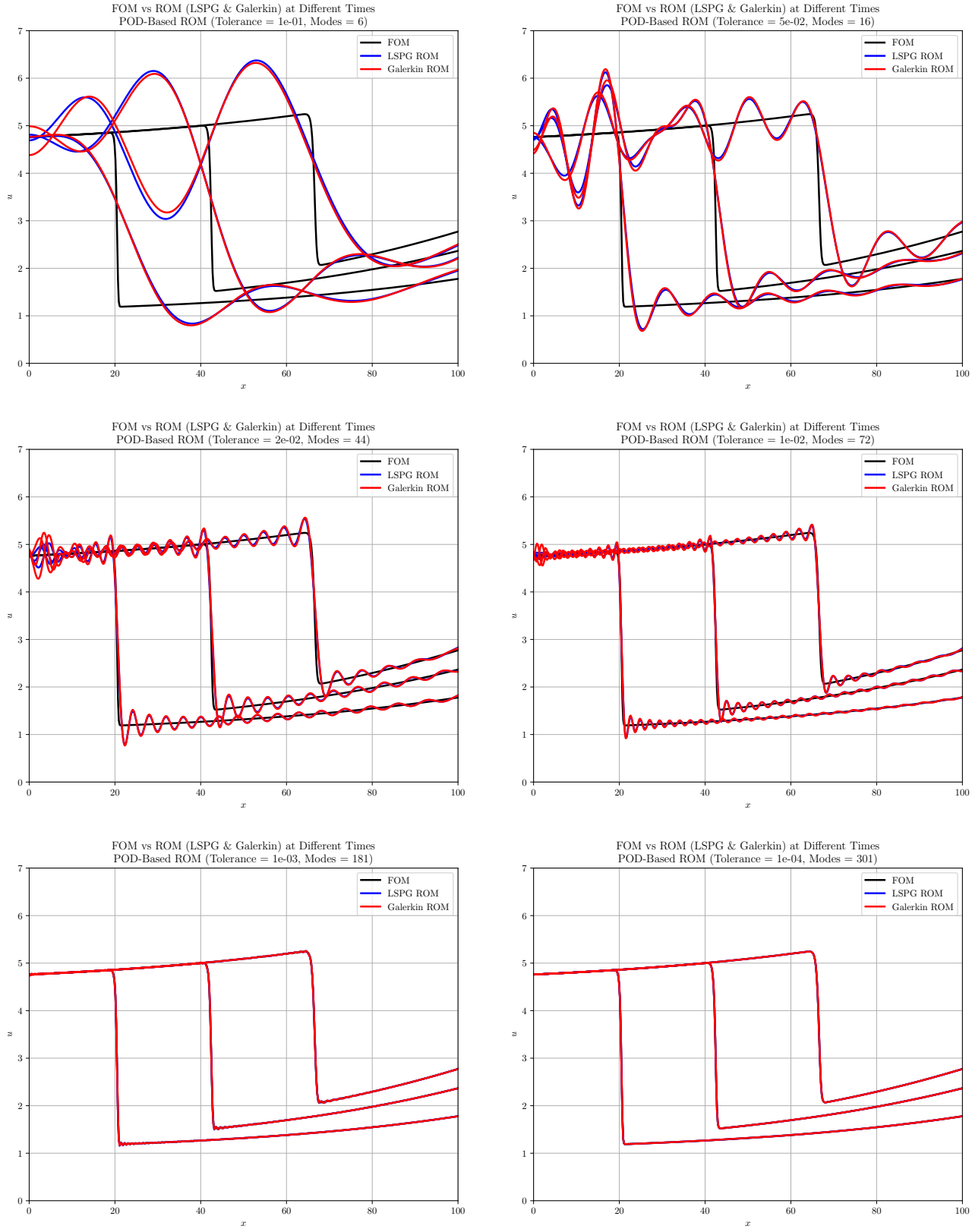

Figure 8: Comparison of FOM, Galerkin, and LSPG ROMs at different tolerances. Each subplot shows the full-order model (FOM) and the reduced-order models (ROMs) for both Galerkin and LSPG at times $t = 7$, $t = 14$, and $t = 21$ seconds.

The L2 norm error is computed as follows:

$$\text{L2 Error} = \frac{\|\mathbf{U}_{\text{FOM}} - \mathbf{U}_{\text{ROM}}\|_2}{\|\mathbf{U}_{\text{FOM}}\|_2}, \tag{57}$$

where $\mathbf{U}_{\text{FOM}}$ is the full-order model (FOM) solution vector, and $\mathbf{U}_{\text{ROM}}$ is the reduced-order model (ROM) solution vector (either Galerkin or LSPG). The L2 norm $\|\cdot\|_2$ represents the Euclidean norm.

6.3. Local Proper Orthogonal Decomposition (Local POD)

In the previous sections, we observed that applying Proper Orthogonal Decomposition (POD) to the 1D Burgers equation required retaining a significant number of modes to achieve the desired accuracy. This indicates a large Kolmogorov n-width, suggesting that the solution manifold is complex and spans multiple regimes. Consequently, a single global basis $\mathbf{\Phi}_{\text{global}}$ may not sufficiently capture the localized features of the solution. To address this challenge, we introduce Local POD, a technique that constructs multiple local bases, each tailored to specific regions of the solution space.

6.3.1. Limitations of Global POD

The necessity of retaining a large number of modes in the global POD framework highlights the inherent complexity of the solution manifold. A single global basis $\mathbf{\Phi}_{\text{global}}$, derived from the entire snapshot matrix $\mathbf{S}$, may fail to capture localized features of the solution efficiently, leading to either reduced accuracy or increased computational cost.

6.3.2. The Concept of Local POD

To overcome these limitations, Local POD partitions the solution space into smaller, more manageable regions. Each region is characterized by its own local basis, constructed using snapshots specific to that region.

Global Basis Construction:. First, a global basis $\mathbf{\Phi}_{\text{global}}$ is constructed using Singular Value Decomposition (SVD) applied to the entire snapshot matrix $\mathbf{S}$:

$$\mathbf{S} = \mathbf{\Phi}_{\text{global}} \mathbf{\Sigma}_{\text{global}} \mathbf{V}_{\text{global}}^T, \tag{58}$$

where $\mathbf{\Phi}_{\text{global}} \in \mathbb{R}^{\mathcal{N} \times r_{\text{global}}}$ contains the left singular vectors forming the global basis, and r_{global} is the number of modes retained based on a given tolerance.

Snapshot Clustering:. The snapshots are projected onto the global basis to obtain reduced coordinates $\mathbf{q}_{\text{global}}$:

$$\mathbf{q}_{\text{global}} = \mathbf{\Phi}_{\text{global}}^T \mathbf{U}, \tag{59}$$

where $\mathbf{U}$ represents the full-order solution snapshots. Clustering algorithms, such as K-Means, are then applied to $\mathbf{q}_{\text{global}}$ to partition the solution space into n_{clusters} regions. This clustering assigns each snapshot to a specific cluster, resulting in submatrices $\mathbf{S}_c$ for each cluster c:

$$\mathbf{S} = [\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_{n_{\text{clusters}}}], \tag{60}$$

where $\mathbf{S}_c$ represents the snapshots within cluster c.

Local Basis Construction:. For each cluster c, we perform SVD on the submatrix $\mathbf{S}_c$ to obtain the local basis $\mathbf{\Phi}_c$:

$$\mathbf{S}_c = \mathbf{\Phi}_c \mathbf{\Sigma}_c \mathbf{V}_c^T, \tag{61}$$

where $\mathbf{\Phi}_c \in \mathbb{R}^{\mathcal{N} \times r_c}$ contains the left singular vectors forming the local basis, and r_c is the number of modes retained for cluster c.

Selection of Local Bases:. During the online phase, the appropriate local basis is selected based on the proximity of the current state to the cluster centers in the reduced global space. Let $\mathbf{u}$ be the current state vector in the full-order space, and $\mathbf{q}_{\text{global}}$ be its representation in the global reduced space:

$$\mathbf{q}_{\text{global}} = \mathbf{\Phi}_{\text{global}}^T \mathbf{u}. \tag{62}$$

The cluster c is selected by finding the closest cluster center to $\mathbf{q}_{\text{global}}$ using a function K_m:

$$c = K_m(\mathbf{q}_{\text{global}}), \tag{63}$$

where K_m represents the clustering algorithm, such as K-Means, which predicts the cluster c to which $\mathbf{q}_{\text{global}}$ belongs.

Once the cluster c is determined, the corresponding local basis $\mathbf{\Phi}_c$ is used to project the system's equations. The reduced coordinates $\mathbf{q}_c$ in the local basis are given by:

$$\mathbf{q}_c = \mathbf{\Phi}_c^T \mathbf{u}, \tag{64}$$

and the full-order solution $\mathbf{u}$ is approximated using the local decoder function:

$$\mathbf{u} \approx D_U(\mathbf{q}_c) = \mathbf{\Phi}_c \mathbf{q}_c. \tag{65}$$

The derivative of the full-order solution with respect to the reduced coordinates $\mathbf{q}_c$ is given by the local basis $\mathbf{\Phi}_c$:

$$\frac{\partial(D_U(\mathbf{q}_c))}{\partial \mathbf{q}_c} = \mathbf{\Phi}_c. \tag{66}$$

For example, in the context of the Least-Squares Petrov-Galerkin (LSPG) projection, the system is projected as follows:

$$\mathbf{\Phi}_c^T \mathbf{J}^T \mathbf{J} \mathbf{\Phi}_c \delta \mathbf{q}^k = -\mathbf{\Phi}_c^T \mathbf{J}^T \mathbf{R}, \tag{67}$$

where $\mathbf{J}$ is the Jacobian matrix, $\mathbf{R}$ is the residual, and $\delta \mathbf{q}^k$ represents the correction to the reduced coordinates at iteration k.

Overlapping Snapshots and Smooth Transitions:. To ensure smooth transitions between clusters, snapshots near cluster boundaries can be included in multiple clusters. The overlap ensures continuity and reduces the risk of discontinuities in the solution:

$$\mathbf{S}_{\text{overlap}} = [\mathbf{S}_c \cap \mathbf{S}_{c'}], \tag{68}$$

where $\mathbf{S}_{\text{overlap}}$ contains snapshots shared by adjacent clusters c and c'.

6.3.3. Analysis of Clustering Configurations

To optimize the trade-off between computational efficiency and accuracy, we performed an analysis of the clustering process by varying the number of clusters n_{clusters}. The key objective was to determine the appropriate number of clusters that would sufficiently capture the localized features of the solution space while minimizing the number of modes retained in each cluster.

The analysis involved applying K-Means clustering to the reduced representation $\mathbf{q}_{\text{global}}$ obtained from the global POD, where a tolerance of 10^{-4} was used to construct both the global and local bases. For each clustering configuration, ranging from 1 to 100 clusters, we computed the local bases and evaluated the average number of modes required per cluster to meet this predefined tolerance level. The average number of modes for each clustering configuration is plotted in Figure 9, along with the vertical range of modes across clusters.

Interpretation of Results:. As observed in Figure 9, the average number of modes decreases significantly as the number of clusters increases, especially in the initial range. This behavior is expected since increasing the number of clusters allows each local basis to capture more specific and less complex features of the solution space, reducing the need for a large number of modes.

However, beyond approximately 20 clusters, the rate of decrease in the average number of modes slows down considerably. This indicates diminishing returns in further subdividing the solution space, as the complexity within each cluster becomes minimal. The vertical range of modes also narrows with an increasing number of clusters, suggesting that the variance in the required number of modes across clusters reduces, leading to a more uniform distribution of complexity across the solution space.

Optimal Number of Clusters:. To balance accuracy and computational efficiency, we suggest using a configuration with around 20 clusters. This number of clusters achieves a significant reduction in the average number of modes, indicating a more efficient representation of the solution space. Additionally, it avoids the diminishing returns observed when the number of clusters exceeds this range.

This choice of 20 clusters provides a practical trade-off: it captures the localized features of the solution while maintaining computational efficiency, ensuring that the reduced-order model remains manageable in both offline and online phases.

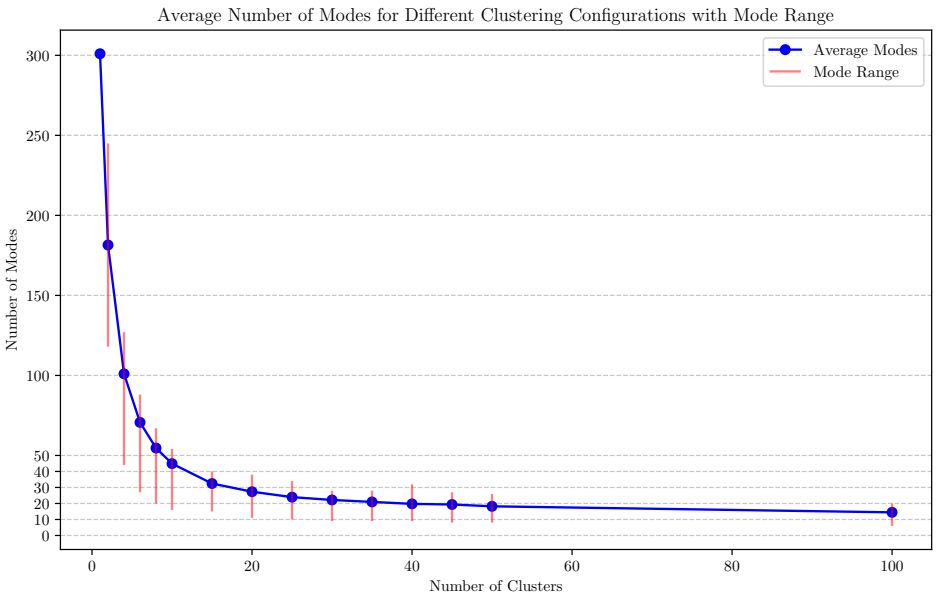

Figure 9: Average number of modes for different clustering configurations, with the vertical bars indicating the range of modes across clusters.

6.3.4. Results for the 20-Cluster Configuration

To validate the effectiveness of the selected clustering configuration, we applied the Local POD approach using 20 clusters. The results, in terms of the L2 norm error for both the Galerkin and LSPG ROMs, are summarized in Table 2. Additionally, Figure 10 illustrates the comparison between the full-order model (FOM) and the reduced-order models (ROMs) for both Galerkin and LSPG methods.

Table 2: L2 Norm Errors and Average Number of Modes for Galerkin and LSPG ROMs with 20 Clusters

Number of Clusters	**Galerkin L2 Error**	**LSPG L2 Error**	**Average Number of Modes**
20	1.72e-4	1.77e-4	27.25

6.3.5. Comparison with Global POD

To further assess the efficiency of the Local POD with 20 clusters, we compared it with the Global POD using 28 modes, which is a comparable number to the average of 27.5 modes per cluster in the Local POD configuration. The L2 norm error for the LSPG ROMs was calculated for both methods.

Table 3: L2 Norm Errors for Global and Local POD with LSPG ROMs

POD Method	**LSPG L2 Error**	**Number of Modes**
Global POD	5.20e-2	28
Local POD (20 clusters)	1.77e-4	27.5 (avg)

Figure 11 shows a direct comparison between the FOM, Local POD (with 20 clusters), and Global POD (with 28 modes) for the LSPG ROMs. The comparison indicates a significant improvement in accuracy with the Local POD approach, despite using a similar number of modes.

The results demonstrate that the Local POD with 20 clusters not only achieves a lower L2 error but also offers a more efficient and accurate representation of the solution space compared to the Global POD, even with a comparable number of modes.

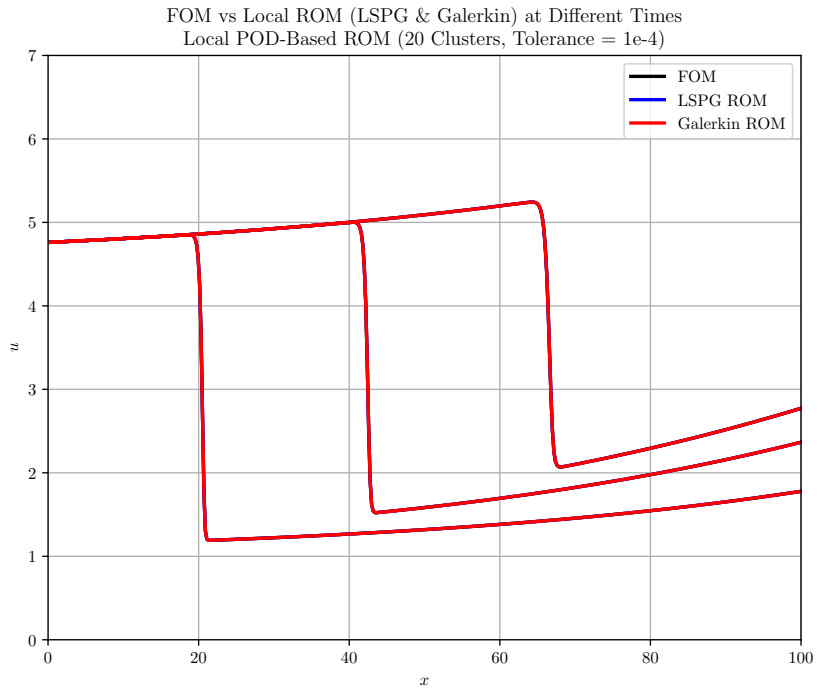

Figure 10: Comparison of FOM, Galerkin, and LSPG Local ROMs at Different Times using 20 clusters. The plot shows the solutions at times $t = 7$, $t = 14$, and $t = 21$ seconds.

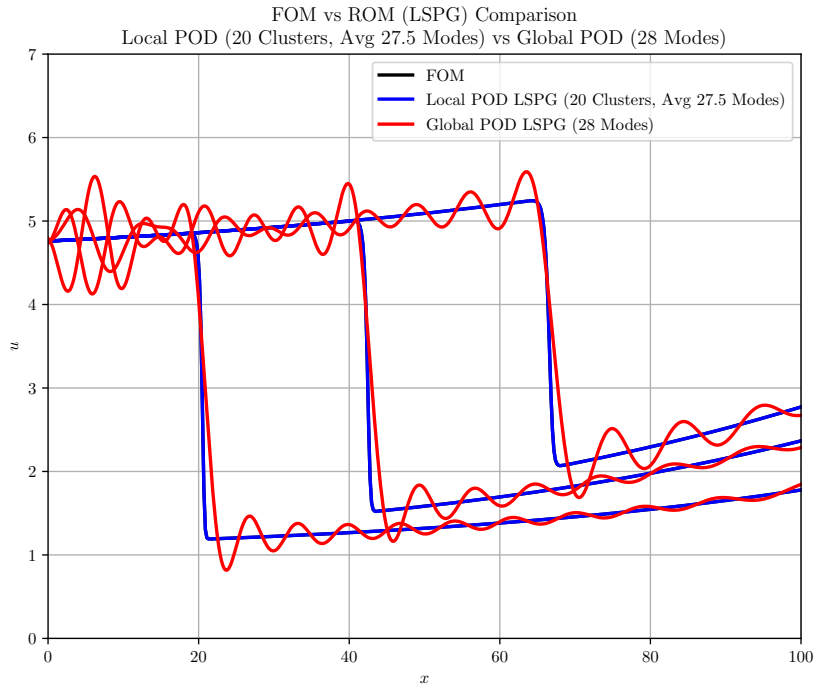

Figure 11: Comparison of FOM, Global POD (28 modes), and Local POD (20 clusters, avg 27.5 modes) using LSPG at Different Times. The plot shows the solutions at times $t = 7$, $t = 14$, and $t = 21$ seconds.

6.4. Quadratic Manifold Approximation

In reduced-order modeling, Proper Orthogonal Decomposition (POD) typically provides a linear approximation of the solution space. This linear model often struggles to capture nonlinear dynamics, especially in complex systems. To address this, we introduce a quadratic manifold approximation that extends the POD framework, capturing second-order interactions between the modes and improving the representation of nonlinearities.

6.4.1. Mathematical Formulation

Given a set of snapshots $\mathbf{S} \in \mathbb{R}^{N \times N_s}$, where N is the number of spatial points and N_s is the number of snapshots, we perform Singular Value Decomposition (SVD) to obtain the full basis:

$$\mathbf{S} = \mathbf{\Phi}_p \mathbf{\Sigma}_p \mathbf{V}_p^T + \mathbf{\Phi}_s \mathbf{\Sigma}_s \mathbf{V}_s^T, \tag{69}$$

where $\mathbf{\Phi}_p \in \mathbb{R}^{N \times r_p}$ contains the first r_p modes retained for the linear approximation, and $\mathbf{\Phi}_s$ contains the remaining modes that are discarded from the linear approximation but approximated in the quadratic manifold.

The traditional linear POD approximation is expressed as:

$$\mathbf{U} \approx D_U(\mathbf{q}_p) = \mathbf{\Phi}_p \mathbf{q}_p, \tag{70}$$

where $\mathbf{q}_p = \mathbf{\Phi}_p^T \mathbf{U}$ are the reduced coordinates in the primary space.

To account for the secondary modes $\mathbf{\Phi}_s$, we introduce a quadratic term:

$$\mathbf{U} \approx D_U(\mathbf{q}_p) = \mathbf{\Phi}_p \mathbf{q}_p + \mathbf{H}\mathbf{Q}(\mathbf{q}_p), \tag{71}$$

where $\mathbf{Q}(\mathbf{q}_p)$ is a vector containing all unique quadratic terms formed by $\mathbf{q}_p$, and $\mathbf{H}$ is a matrix designed to approximate the effect of the secondary modes $\mathbf{\Phi}_s$.

6.4.2. Construction of the Quadratic Approximation

To compute the matrix $\mathbf{H}$, we first calculate the error matrix $\mathbf{E}$ representing the difference between the original snapshots and their linear POD approximation:

$$\mathbf{E} = \mathbf{S} - \mathbf{\Phi}_p \mathbf{q}_p. \tag{72}$$

Next, we form the matrix $\mathbf{Q}$ by stacking the quadratic terms $\mathbf{Q}(\mathbf{q}_p)$ for each snapshot:

$$\mathbf{Q} = \left[\mathbf{Q}(\mathbf{q}_p^{(1)}) \quad \mathbf{Q}(\mathbf{q}_p^{(2)}) \quad \dots \quad \mathbf{Q}(\mathbf{q}_p^{(N_s)}) \right]. \tag{73}$$

The matrix $\mathbf{H}$ is then obtained by solving the following regularized least-squares problem:

$$\mathbf{H} = \text{argmin}_{\mathbf{H}} \|\mathbf{E} - \mathbf{H}\mathbf{Q}\|_F^2 + \alpha \|\mathbf{H}\|_F^2, \tag{74}$$

where α is a regularization parameter used to ensure stability.

6.4.3. Construction of the Matrix $\mathbf{H}$

To compute the matrix $\mathbf{H}$, we first calculate the error matrix $\mathbf{E}$ that represents the difference between the original snapshots and their linear POD approximation:

$$\mathbf{E} = \mathbf{S} - \mathbf{\Phi}_p \mathbf{q}_p. \tag{75}$$

Here, $\mathbf{S} \in \mathbb{R}^{N \times N_s}$ is the snapshot matrix with N spatial points and N_s snapshots, $\mathbf{\Phi}_p \in \mathbb{R}^{N \times r_p}$ contains the primary modes, and $\mathbf{q}_p \in \mathbb{R}^{r_p \times N_s}$ represents the reduced coordinates.

Next, we form the matrix $\mathbf{Q}$ by stacking the symmetric quadratic terms $\mathbf{Q}(\mathbf{q}_p)$ for each snapshot:

$$\mathbf{Q} = \left[\mathbf{Q}(\mathbf{q}_p^{(1)}) \quad \mathbf{Q}(\mathbf{q}_p^{(2)}) \quad \dots \quad \mathbf{Q}(\mathbf{q}_p^{(N_s)}) \right], \tag{76}$$

where $\mathbf{Q}(\mathbf{q}_p) \in \mathbb{R}^k$ contains the unique quadratic terms formed by $\mathbf{q}_p$, and $k = \frac{r_p(r_p+1)}{2}$ is the number of unique quadratic terms. Consequently, $\mathbf{Q} \in \mathbb{R}^{k \times N_s}$.

To solve for the matrix $\mathbf{H}$, we first perform a Singular Value Decomposition (SVD) on $\mathbf{Q}$:

$$\mathbf{Q} = \mathbf{U}_q \mathbf{\Sigma}_q \mathbf{V}_q^T, \tag{77}$$

where $\mathbf{U}_q \in \mathbb{R}^{k\times k}$, $\mathbf{\Sigma}_q \in \mathbb{R}^{k\times N_s}$, and $\mathbf{V}_q \in \mathbb{R}^{N_s\times N_s}$.

The matrix $\mathbf{H} \in \mathbb{R}^{N\times k}$ is then obtained by solving the following weighted least-squares problem:

$$\mathbf{H}_i = \sum_{l=1}^{N_q} \left(\frac{\sigma_q^2[l]}{\sigma_q^2[l] + \alpha^2} \right) \left(\frac{\mathbf{V}_q[l,:] \cdot \mathbf{E}_i^T}{\sigma_q[l]} \right) \mathbf{U}_q[:,l], \tag{78}$$

where $\mathbf{H}_i$ represents the i-th row of $\mathbf{H}$, and $\sigma_q[l]$ are the singular values from $\mathbf{\Sigma}_q$. Here, α is the regularization parameter that ensures the stability of the solution.

Finally, the matrix $\mathbf{H}$ encapsulates the influence of the secondary modes $\mathbf{\Phi}_s$ on the primary modes $\mathbf{\Phi}_p$ through the quadratic terms, allowing for a more accurate reconstruction of the original snapshots.

6.4.4. Derivative of the Quadratic Manifold Approximation

Given the quadratic manifold approximation:

$$\mathbf{u} \approx D_U(\mathbf{q}_p) = \mathbf{\Phi}_p \mathbf{q}_p + \mathbf{H}\mathbf{Q}(\mathbf{q}_p), \tag{79}$$

where $\mathbf{Q}(\mathbf{q}_p)$ contains the quadratic terms of $\mathbf{q}_p$, the derivative of $\mathbf{u}$ with respect to $\mathbf{q}_p$ is given by:

$$\frac{\partial(D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p} = \mathbf{\Phi}_p + \mathbf{H}\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p}. \tag{80}$$

The derivative $\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p}$ can be expressed using the identity matrix $\mathbf{I}$ and tensor operations:

$$\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p} = \mathbf{I} \otimes \mathbf{q}_p + \mathbf{q}_p \otimes \mathbf{I}, \tag{81}$$

where $\otimes$ denotes the Kronecker product.

Thus, the derivative of the quadratic manifold approximation becomes:

$$\frac{\partial(D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p} = \mathbf{\Phi}_p + \mathbf{H}\left(\mathbf{I} \otimes \mathbf{q}_p + \mathbf{q}_p \otimes \mathbf{I}\right). \tag{82}$$

6.4.5. Practical Derivative Calculation

While the theoretical derivative of the quadratic manifold approximation involves tensor operations and the identity matrix, in practice, we calculate the derivative directly by considering the symmetric part of the quadratic terms.

Given the quadratic manifold approximation:

$$\mathbf{u} \approx D_U(\mathbf{q}_p) = \mathbf{\Phi}_p \mathbf{q}_p + \mathbf{H}\mathbf{Q}(\mathbf{q}_p), \tag{83}$$

where $\mathbf{Q}(\mathbf{q}_p)$ contains the unique quadratic terms formed by $\mathbf{q}_p$.

For a practical example, consider $k = 6$, meaning $\mathbf{q}_p$ has 6 components:

$$\mathbf{q}_p = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \end{pmatrix}.$$

The quadratic terms $\mathbf{Q}(\mathbf{q}_p)$ include:

$$\mathbf{Q}(\mathbf{q}_p) = \begin{pmatrix} q_1^2, & q_1q_2, & q_1q_3, & q_1q_4, & q_1q_5, & q_1q_6, & q_2^2, & \ldots, & q_6^2 \end{pmatrix}^T.$$

The derivative $\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p}$ for this case is a matrix where each row corresponds to the derivative of a quadratic term with respect to each component of $\mathbf{q}_p$. Specifically:

$$\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p} = \begin{pmatrix} 2q_1 & q_2 & q_3 & q_4 & q_5 & q_6 \\ q_1 & q_2 & 0 & 0 & 0 & 0 \\ q_1 & 0 & q_3 & 0 & 0 & 0 \\ q_1 & 0 & 0 & q_4 & 0 & 0 \\ q_1 & 0 & 0 & 0 & q_5 & 0 \\ q_1 & 0 & 0 & 0 & 0 & q_6 \\ 0 & 2q_2 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 2q_6 \end{pmatrix}.$$

This matrix is typically sparse and has non-zero entries where the quadratic terms involve the corresponding components of $\mathbf{q}_p$.

Thus, in practice, the derivative of the quadratic manifold approximation is calculated as:

$$\frac{\partial (D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p} = \mathbf{\Phi}_p + \mathbf{H}\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p}, \tag{84}$$

where $\mathbf{H}\frac{\partial \mathbf{Q}(\mathbf{q}_p)}{\partial \mathbf{q}_p}$ is computed directly using the structure of the quadratic terms.

For the specific case of $k = 6$, the resulting derivative matrix provides the gradient of the reconstructed solution $\mathbf{u}$ with respect to the reduced coordinates $\mathbf{q}_p$, incorporating both the linear and quadratic contributions.

6.4.6. Results and Discussion

Despite the theoretical promise of the quadratic manifold approximation, our results show that the improvement in reconstruction accuracy over the traditional POD method is relatively modest. The total L2 norm error for the POD reconstruction was 0.034, while the quadratic manifold approximation reduced the error to 0.013. However, this improvement comes with a significant risk of overfitting, especially for different values of the regularization parameter α.

Moreover, attempts to derive a PROM (Projection-based Reduced Order Model) using this quadratic manifold were unsuccessful. The high norm of the decoder's derivative (the sensitivity of the full-order model to changes in the reduced coordinates) made the PROM approach unstable. This instability likely arises from the complexity introduced by the quadratic terms, which significantly amplify small errors in the reduced coordinates.

Method	**Total L2 Norm Error**
POD	0.034
Quadratic Manifold Approximation	0.013

Table 4: Comparison of Total L2 Norm Errors for POD and Quadratic Manifold Approximation.

The figure shows the reconstructed solutions at different time steps (7s, 14s, and 21s) for the FOM, POD, and Quadratic Manifold Approximation. While the quadratic approximation provides a slightly better fit to the full-order model (FOM) than the POD, the overall gain is limited, and the complexity of the quadratic approach introduces challenges in stability and generalization.

6.5. ANN-Augmented Manifold Approximation

In reduced-order modeling, POD typically provides a linear approximation of the solution space. However, for complex systems, especially those with strong nonlinearities, a purely linear model might struggle to capture the underlying dynamics. To address this, we introduce an ANN-augmented manifold approximation that extends the traditional POD framework by incorporating nonlinear corrections using a neural network (ANN).

6.5.1. Mathematical Formulation

Given a set of snapshots $\mathbf{S} \in \mathbb{R}^{N \times N_s}$, where N is the number of spatial points and N_s is the number of snapshots, we begin by performing a Singular Value Decomposition (SVD) to obtain the primary modes $\mathbf{\Phi}_p$ and secondary modes $\mathbf{\Phi}_s$:

$$\mathbf{S} = \mathbf{\Phi}_p \mathbf{\Sigma}_p \mathbf{V}_p^T + \mathbf{\Phi}_s \mathbf{\Sigma}_s \mathbf{V}_s^T, \tag{85}$$

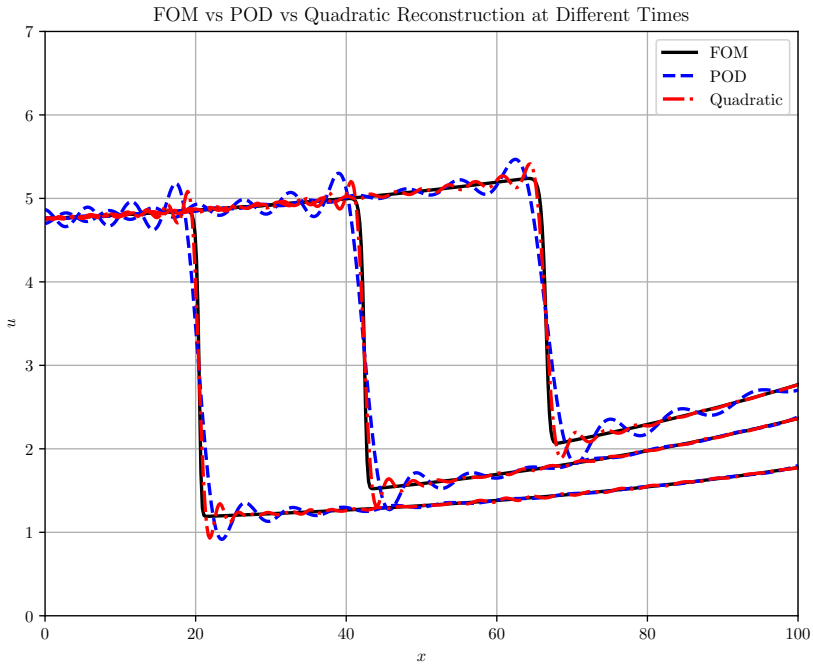

Figure 12: Comparison of FOM, POD, and Quadratic Manifold reconstructions at different time steps (7s, 14s, and 21s).

where $\mathbf{\Phi}_p \in \mathbb{R}^{N \times r_p}$ contains the first r_p modes retained for the linear approximation, and $\mathbf{\Phi}_s$ contains the remaining modes. The traditional linear POD approximation can be expressed as:

$$\mathbf{U}(t;\mu) \approx \mathbf{\Phi}_p \mathbf{q}_p(t;\mu), \tag{86}$$

where $\mathbf{q}_p(t;\mu) = \mathbf{\Phi}_p^T \mathbf{U}(t;\mu)$ are the reduced coordinates corresponding to the principal modes.

To capture nonlinear dynamics more accurately, we augment this linear approximation with an ANN term, leading to the following expression:

$$\mathbf{U}(t;\mu) \approx \mathbf{\Phi}_p \mathbf{q}_p(t;\mu) + \mathbf{\Phi}_s \mathbf{N}(\mathbf{q}_p(t;\mu)), \tag{87}$$

where $\mathbf{\Phi}_s$ represents the secondary modes, and $\mathbf{N}(\mathbf{q}_p)$ is a nonlinear map approximated by a neural network.

6.5.2. Construction of the ANN-Augmented Approximation

To construct the ANN-augmented approximation, we begin with the linear POD approximation, which captures the contribution of the principal modes $\mathbf{\Phi}_p \mathbf{q}_p(t;\mu)$. The secondary modes, which are not captured by the principal modes, are then approximated by an ANN. Specifically, the nonlinear term modeled by the ANN should capture the residual contribution of the secondary modes $\mathbf{\Phi}_s$:

$$\mathbf{N}(\mathbf{q}_p) \approx \mathbf{\Phi}_s^T \left(\mathbf{U}(t;\mu) - \mathbf{\Phi}_p \mathbf{q}_p(t;\mu) \right). \tag{88}$$

The network is trained on the snapshot data to minimize the difference between the true solution and the linear approximation, effectively capturing the higher-order nonlinearities. This approach allows the ANN to model complex dynamics that are not adequately represented by the linear POD, enhancing the overall accuracy of the reduced-order model.

6.5.3. Derivative of the ANN-Augmented Approximation

To utilize the ANN-augmented approximation in PROMs, we must compute the derivative of the approximation with respect to the reduced coordinates $\mathbf{q}_p$. This derivative is crucial for projection-based methods like the Least-Squares Petrov-Galerkin (LSPG) approach.

The derivative of the ANN-augmented approximation is given by:

$$\frac{\partial(D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p} = \mathbf{\Phi}_p + \mathbf{\Phi}_s \frac{\partial \mathbf{N}(\mathbf{q}_p)}{\partial \mathbf{q}_p}, \tag{89}$$

where $\frac{\partial \mathbf{N}(\mathbf{q}_p)}{\partial \mathbf{q}_p}$ is the Jacobian of the ANN with respect to the principal reduced coordinates $\mathbf{q}_p$. This expression captures both the linear contributions from the primary modes and the nonlinear corrections provided by the ANN.

6.5.4. Incorporating the ANN-Augmented Approximation into LSPG

In the LSPG framework, we linearize the residual $\mathbf{R}$ of the system around the current estimate of $\mathbf{q}_p$ to iteratively update $\mathbf{q}_p$. The linearization leads to the following equation:

$$\frac{\partial(D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p}^T \mathbf{J}^T \mathbf{J} \frac{\partial D_U(\mathbf{q}_p)}{\partial \mathbf{q}_p} \delta \mathbf{q}_p^k = -\frac{\partial(D_U(\mathbf{q}_p))}{\partial \mathbf{q}_p}^T \mathbf{J}^T \mathbf{R}, \tag{90}$$

where $\delta \mathbf{q}_p^k$ is the update to the reduced coordinates in the k-th iteration, and $\mathbf{J}$ is the Jacobian of the full-order system. Substituting the derivative expression into this equation gives us:

$$\left(\mathbf{\Phi}_p^T + \left(\frac{\partial \mathbf{N}(\mathbf{q}_p)}{\partial \mathbf{q}_p}\right)^T \mathbf{\Phi}_s^T\right) \mathbf{J}^T \mathbf{J} \left(\mathbf{\Phi}_p + \mathbf{\Phi}_s \frac{\partial \mathbf{N}(\mathbf{q}_p)}{\partial \mathbf{q}_p}\right) \delta \mathbf{q}_p^k = -\left(\mathbf{\Phi}_p^T + \left(\frac{\partial \mathbf{N}(\mathbf{q}_p)}{\partial \mathbf{q}_p}\right)^T \mathbf{\Phi}_s^T\right) \mathbf{J}^T \mathbf{R}. \tag{91}$$

A. Finite Element Matrices in 1D Burgers' Equation

In the context of the 1D Burgers' equation, we need to compute several matrices using the Finite Element Method (FEM). Two key matrices are the mass matrix $\mathbf{M}$ and the convection matrix $\mathbf{C}(\mathbf{U})$.

A.1. Transformation to the Reference Element

To simplify integration, we map the physical element $[x_1, x_2]$ to a reference element $[-1, 1]$ using a linear transformation:

$$x(\xi) = \frac{(1-\xi)}{2} x_1 + \frac{(1+\xi)}{2} x_2 \tag{92}$$

where $\xi \in [-1, 1]$ is the coordinate in the reference element.

The integral over the physical element then transforms into an integral over the reference element:

$$\int_{x_1}^{x_2} f(x)\, dx = \int_{-1}^{1} f(\xi) \left|\frac{dx}{d\xi}\right| d\xi \tag{93}$$

where the Jacobian $\frac{dx}{d\xi}$ accounts for the transformation from ξ to x.

A.2. The Jacobian

In 1D, for a linear element, the Jacobian J is constant and relates the differential $d\xi$ in the reference element to the differential dx in the physical element:

$$J = \frac{dx}{d\xi} = \frac{x_2 - x_1}{2} \tag{94}$$

Here, x_1 and x_2 are the coordinates of the element's endpoints in the physical domain, and ξ is the coordinate in the reference element. The Jacobian J represents the stretching or compression of the reference element to fit the physical element.

A.3. Gaussian Quadrature

Gaussian quadrature is a numerical integration method used to approximate integrals over the interval $[-1, 1]$. The integral is approximated as a weighted sum of the function values at specific points (Gauss points):

$$\int_{-1}^{1} f(\xi)\, d\xi \approx \sum_{k=1}^{n_{\text{gauss}}} w_k f(\xi_k) \tag{95}$$

where:

- ξ_k are the Gauss points, the locations in the reference element where the function is evaluated.
- w_k are the corresponding weights, which scale the contribution of each Gauss point to the integral.

In the implementation we have $n_{\text{gauss}} = 2$:

$$\text{Gauss points: } \xi_1 = -\frac{\sqrt{3}}{3}, \quad \xi_2 = \frac{\sqrt{3}}{3}$$

$$\text{Weights: } w_1 = w_2 = 1$$

These are represented in the code by the arrays:

$$\text{Gauss points: } = \left[-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right]$$

$$\text{Weights: } = [1, 1]$$

The shape functions $N_i(\xi)$ and their derivatives $\frac{dN_i(\xi)}{d\xi}$ are evaluated at these Gauss points for the integration. For instance, the shape functions in your code are defined as:

$$N(\xi) = \left[\frac{1-\xi}{2}, \frac{1+\xi}{2}\right]$$

and their derivatives are:

$$\frac{dN(\xi)}{d\xi} = \left[-\frac{1}{2}, \frac{1}{2}\right]$$

These arrays and the corresponding operations provide the necessary components to compute integrals in the FEM discretization process accurately.

A.4. Mass Matrix in FEM

The mass matrix $\mathbf{M}$ is defined as:

$$\mathbf{M}_{ij} = \int_{\Omega} N_i(x) N_j(x)\, dx \tag{96}$$

where $N_i(x)$ and $N_j(x)$ are the shape functions associated with the ith and jth nodes, respectively.
Using the transformations and Gaussian quadrature:

$$\mathbf{M}_{ij} \approx \sum_{k=1}^{n_{\text{gauss}}} w_k N_i(\xi_k) N_j(\xi_k) |J| \tag{97}$$

Here:

- $N_i(\xi_k)$ and $N_j(\xi_k)$ are the values of the shape functions evaluated at the Gauss points.
- $|J|$ is the absolute value of the Jacobian, accounting for the scaling between the reference and physical element.
- w_k is the weight associated with the Gauss point.

A.5. Implementation Details of the Mass Matrix

The mass matrix $\mathbf{M}$ arises from the time-dependent term in the 1D Burgers' equation. Below is a detailed explanation of the implementation, relating each part of the code to its corresponding mathematical expression.

A.5.1. Initial Setup

```
n_nodes = len(self.X)  # Number of nodes
n_elements, n_local_nodes = self.T.shape  # Number of elements and nodes per element
M_global = sp.lil_matrix((n_nodes, n_nodes))  # Initialize global mass matrix
```

In mathematical terms:

- **Number of Nodes** (n_{nodes}): This is the total number of nodes in the finite element mesh.
- **Number of Elements** (n_{elements}) and **Nodes per Element** ($n_{\text{local nodes}}$): These define the structure of the mesh. For 1D linear elements, $n_{\text{local nodes}}$ is typically 2.
- **Global Mass Matrix** ($\mathbf{M}_{\text{global}}$): This matrix will store the assembled contributions from all elements:

$$\mathbf{M} = \sum_{e=1}^{n_{\text{elements}}} \mathbf{M}_e$$

  where $\mathbf{M}_e$ is the local mass matrix for the e-th element.

A.5.2. Element Loop

```
for elem in range(n_elements):
    element_nodes = self.T[elem, :] - 1  # Nodes of the current element (adjusted for 0-based indexing)
    x_element = self.X[element_nodes].reshape(-1, 1)  # Physical coordinates of the element's nodes

    M_element = np.zeros((n_local_nodes, n_local_nodes))  # Initialize local mass matrix
```

Mathematically:

- **Element Nodes** (element_nodes): These are the indices of the nodes that belong to the current element. If the e-th element is defined by nodes i and j, then:

$$\text{element\_nodes} = [i, j]$$

- **Physical Coordinates of Element Nodes** (x_{element}): These are the actual x-coordinates of the nodes of the element:

$$x_{\text{element}} = \begin{bmatrix} x_i \\ x_j \end{bmatrix}$$

- **Local Mass Matrix** ($\mathbf{M}_e$): This represents the element-wise contribution to the global mass matrix:

$$\mathbf{M}_e = \int_{\Omega_e} N_i(x) N_j(x)\, dx$$

A.5.3. Gauss Point Loop

```
for gauss_point in range(self.ngaus):
    N_gp = self.N[gauss_point, :]  # Shape functions at the Gauss point
    dN_dxi_gp = self.Nxi[gauss_point, :]  # Shape function derivatives wrt reference coordinate at the Gauss point

    # Compute the Jacobian
```

```
    J = dN_dxi_gp @ x_element

    # Compute the differential volume
    dV = self.wgp[gauss_point] * np.abs(J)

    # Update the local mass matrix
    M_element += np.outer(N_gp, N_gp) * dV
```

Mathematically:

- **Shape Functions** ($N_i(\xi)$): These are the shape functions evaluated at the Gauss point ξ_k. For linear elements:

$$N_1(\xi) = \frac{1-\xi}{2}, \quad N_2(\xi) = \frac{1+\xi}{2}$$

- **Shape Function Derivatives** ($\frac{dN_i(\xi)}{d\xi}$): These are the derivatives of the shape functions with respect to the reference coordinate ξ:

$$\frac{dN_1(\xi)}{d\xi} = -\frac{1}{2}, \quad \frac{dN_2(\xi)}{d\xi} = \frac{1}{2}$$

- **Jacobian** (J): The Jacobian J represents the transformation from the reference element ξ to the physical element x. It is computed as:

$$J = \frac{dx}{d\xi} = \frac{dN_1(\xi)}{d\xi} \cdot x_1 + \frac{dN_2(\xi)}{d\xi} \cdot x_2$$

- **Differential Volume** (dV): The differential volume (or length in 1D) is:

$$dV = w_k \cdot |J|$$

  where w_k is the weight associated with the Gauss point.

- **Update Local Mass Matrix** ($\mathbf{M}_e$): The local mass matrix is updated by summing the contributions from each Gauss point:

$$\mathbf{M}_e += (N_i(\xi_k) \cdot N_j(\xi_k)) \cdot dV$$

A.5.4. Assembly of the Global Matrix

```
for i in range(n_local_nodes):
    for j in range(n_local_nodes):
        M_global[element_nodes[i], element_nodes[j]] += M_element[i, j]
```

Mathematically:

- **Assembly**: After computing the local matrix $\mathbf{M}_e$, it is assembled into the global matrix $\mathbf{M}$ by adding the local contributions to the corresponding global positions:

$$\mathbf{M}_{ij} += \mathbf{M}_e^{ij} \quad \text{for } i, j = 1, 2$$

A.5.5. Return the Global Matrix

```
return M_global.tocsc()  # Convert to compressed sparse column format for efficiency
```

Finally, the global mass matrix $\mathbf{M}$ is converted to Compressed Sparse Column (CSC) format for efficient storage and computation.

A.6. Convection Matrix in FEM

The convection matrix $\mathbf{C}(\mathbf{U})$ arises from the advection term and is defined as:

$$\mathbf{C}_{ij}(\mathbf{U}) = \int_{\Omega} U(x,t) \frac{\partial N_j(x)}{\partial x} N_i(x)\, dx \tag{98}$$

Using the same transformations and Gaussian quadrature:

$$\mathbf{C}_{ij}(\mathbf{U}) \approx \sum_{k=1}^{n_{\text{gauss}}} w_k U(\xi_k) \frac{\partial N_j(\xi_k)}{\partial \xi} N_i(\xi_k) |J| \tag{99}$$

Here:

- $U(\xi_k)$ is the value of the solution field at the Gauss points.
- $N_i(\xi_k)$ and $N_j(\xi_k)$ are the values of the shape functions evaluated at the Gauss points.
- $\frac{\partial N_j(\xi_k)}{\partial \xi}$ is the derivative of the shape function $N_j(x)$ with respect to ξ.
- $|J|$ is the absolute value of the Jacobian.
- w_k is the weight associated with the Gauss point.

A.7. Implementation Details of the Convection Matrix

The convection matrix $\mathbf{C}$ arises from the convection term in the 1D Burgers' equation. Below is a detailed explanation of the implementation, relating each part of the code to its corresponding mathematical expression.

A.7.1. Initial Setup

```
n_nodes = len(self.X)  # Number of nodes
n_elements, n_local_nodes = self.T.shape  # Number of elements and nodes per element
C_global = sp.lil_matrix((n_nodes, n_nodes))  # Initialize global convection matrix
```

In mathematical terms:

- **Number of Nodes** (n_{nodes}): This is the total number of nodes in the finite element mesh.
- **Number of Elements** (n_{elements}) and **Nodes per Element** ($n_{\text{local nodes}}$): These define the structure of the mesh. For 1D linear elements, $n_{\text{local nodes}}$ is typically 2.
- **Global Convection Matrix** ($\mathbf{C}_{\text{global}}$): This matrix will store the assembled contributions from all elements:

$$\mathbf{C} = \sum_{e=1}^{n_{\text{elements}}} \mathbf{C}_e$$

where $\mathbf{C}_e$ is the local convection matrix for the e-th element.

A.7.2. Element Loop

```
for elem in range(n_elements):
    element_nodes = self.T[elem, :] - 1  # Nodes of the current element (adjusted for 0-based indexing)
    x_element = self.X[element_nodes].reshape(-1, 1)  # Physical coordinates of the element's nodes

    u_element = U_n[element_nodes]  # Solution values at the element's nodes
    C_element = np.zeros((n_local_nodes, n_local_nodes))  # Initialize local convection matrix
```

Mathematically:

- **Element Nodes** (element_nodes): These are the indices of the nodes that belong to the current element. If the e-th element is defined by nodes i and j, then:

$$\text{element\_nodes} = [i, j]$$

- **Physical Coordinates of Element Nodes** (x_{element}): These are the actual x-coordinates of the nodes of the element:

$$x_{\text{element}} = \begin{bmatrix} x_i \\ x_j \end{bmatrix}$$

- **Solution Values at Element Nodes** (u_{element}): These are the values of the solution u at the nodes of the element:

$$u_{\text{element}} = \begin{bmatrix} u_i \\ u_j \end{bmatrix}$$

- **Local Convection Matrix** ($\mathbf{C}_e$): This represents the element-wise contribution to the global convection matrix:

$$\mathbf{C}_e = \int_{\Omega_e} u(x,t) \frac{dN_j(x)}{dx} N_i(x)\, dx$$

A.7.3. Gauss Point Loop

```python
for gauss_point in range(self.ngaus):
    N_gp = self.N[gauss_point, :]  # Shape functions at the Gauss point
    dN_dxi_gp = self.Nxi[gauss_point, :]  # Shape function derivatives wrt reference coordinate at the Gauss point

    # Compute the Jacobian
    J = dN_dxi_gp @ x_element

    # Compute the differential volume
    dV = self.wgp[gauss_point] * np.abs(J)

    # Compute the derivative of shape functions with respect to the physical coordinate
    dN_dx_gp = dN_dxi_gp / J

    # Compute the solution value at the Gauss point
    u_gp = N_gp @ u_element

    # Update the local convection matrix
    C_element += np.outer(N_gp, u_gp * dN_dx_gp) * dV
```

Mathematically:

- **Shape Functions** ($N_i(\xi)$): These are the shape functions evaluated at the Gauss point ξ_k. For linear elements:

$$N_1(\xi) = \frac{1-\xi}{2}, \quad N_2(\xi) = \frac{1+\xi}{2}$$

- **Shape Function Derivatives** ($\frac{dN_i(\xi)}{d\xi}$): These are the derivatives of the shape functions with respect to the reference coordinate ξ:

$$\frac{dN_1(\xi)}{d\xi} = -\frac{1}{2}, \quad \frac{dN_2(\xi)}{d\xi} = \frac{1}{2}$$

- **Jacobian** (J): The Jacobian J represents the transformation from the reference element ξ to the physical element x. It is computed as:

$$J = \frac{dx}{d\xi} = \frac{dN_1(\xi)}{d\xi} \cdot x_1 + \frac{dN_2(\xi)}{d\xi} \cdot x_2$$

- **Differential Volume** (dV): The differential volume (or length in 1D) is:

$$dV = w_k \cdot |J|$$

where w_k is the weight associated with the Gauss point.

- **Derivative of Shape Functions with Respect to Physical Coordinate** ($\frac{dN_i(x)}{dx}$): The derivatives of the shape functions with respect to x are:

$$\frac{dN_i(x)}{dx} = \frac{dN_i(\xi)}{d\xi} \cdot \frac{1}{J}$$

- **Solution Value at the Gauss Point** ($u(\xi_k)$): The solution value at the Gauss point is:

$$u(\xi_k) = N_1(\xi_k)u_1 + N_2(\xi_k)u_2$$

- **Update Local Convection Matrix** ($\mathbf{C}_e$): The local convection matrix is updated by summing the contributions from each Gauss point:

$$\mathbf{C}_e += \left(N_i(\xi_k) \cdot u(\xi_k) \cdot \frac{dN_j(x)}{dx} \right) \cdot dV$$

A.7.4. Assembly of the Global Matrix

```
for i in range(n_local_nodes):
    for j in range(n_local_nodes):
        C_global[element_nodes[i], element_nodes[j]] += C_element[i, j]
```

Mathematically:

- **Assembly**: After computing the local matrix $\mathbf{C}_e$, it is assembled into the global matrix $\mathbf{C}$ by adding the local contributions to the corresponding global positions:

$$\mathbf{C}_{ij} += \mathbf{C}_e^{ij} \quad \text{for } i, j = 1, 2$$

A.7.5. Return the Global Matrix

```
return C_global.tocsc()  # Convert to compressed sparse column format for efficiency
```

Finally, the global convection matrix $\mathbf{C}$ is converted to Compressed Sparse Column (CSC) format for efficient storage and computation.

A.8. Diffusion Matrix in FEM

The diffusion matrix $\mathbf{K}$ arises from the diffusion term in the 1D Burgers' equation and is defined as:

$$\mathbf{K}_{ij} = \int_\Omega \frac{dN_i(x)}{dx} \frac{dN_j(x)}{dx} \, dx \tag{100}$$

where $\frac{dN_i(x)}{dx}$ and $\frac{dN_j(x)}{dx}$ are the derivatives of the shape functions associated with the ith and jth nodes, respectively. Using the transformations and Gaussian quadrature:

$$\mathbf{K}_{ij} \approx \sum_{k=1}^{n_{\text{gauss}}} w_k \frac{dN_i(\xi_k)}{d\xi} \frac{dN_j(\xi_k)}{d\xi} |J| \tag{101}$$

Here:

- $\frac{dN_i(\xi_k)}{d\xi}$ and $\frac{dN_j(\xi_k)}{d\xi}$ are the values of the derivatives of the shape functions evaluated at the Gauss points.
- $|J|$ is the absolute value of the Jacobian, which accounts for the scaling between the reference and physical element.
- w_k is the weight associated with the Gauss point.

A.9. Implementation Details of the Diffusion Matrix

The diffusion matrix $\mathbf{K}$ is computed in a step-by-step process. Below is a detailed explanation of the implementation, relating each part of the code to its corresponding mathematical expression.

A.9.1. Initial Setup

```
n_nodes = len(self.X)  # Number of nodes
n_elements, n_local_nodes = self.T.shape  # Number of elements and nodes per element
K_global = sp.lil_matrix((n_nodes, n_nodes))  # Initialize global diffusion matrix
```

In mathematical terms:

- **Number of Nodes** (n_{nodes}): This is the total number of nodes in the finite element mesh.
- **Number of Elements** (n_{elements}) and **Nodes per Element** ($n_{\text{local nodes}}$): These define the structure of the mesh. For 1D linear elements, $n_{\text{local nodes}}$ is typically 2.
- **Global Diffusion Matrix** ($\mathbf{K}_{\text{global}}$): This matrix will store the assembled contributions from all elements:

$$\mathbf{K} = \sum_{e=1}^{n_{\text{elements}}} \mathbf{K}_e$$

  where $\mathbf{K}_e$ is the local diffusion matrix for the e-th element.

A.9.2. Element Loop

```
for elem in range(n_elements):
    element_nodes = self.T[elem, :] - 1  # Nodes of the current element (adjusted for 0-based indexing)
    x_element = self.X[element_nodes].reshape(-1, 1)  # Physical coordinates of the element's nodes
    K_element = np.zeros((n_local_nodes, n_local_nodes))  # Initialize local diffusion matrix
```

Mathematically:

- **Element Nodes** (element_nodes): These are the indices of the nodes that belong to the current element. If the e-th element is defined by nodes i and j, then:

$$\text{element\_nodes} = [i, j]$$

- **Physical Coordinates of Element Nodes** (x_{element}): These are the actual x-coordinates of the nodes of the element:

$$x_{\text{element}} = \begin{bmatrix} x_i \\ x_j \end{bmatrix}$$

- **Local Diffusion Matrix** ($\mathbf{K}_e$): This represents the element-wise contribution to the global diffusion matrix:

$$\mathbf{K}_e = \int_{\Omega_e} \frac{dN_i(x)}{dx} \frac{dN_j(x)}{dx} \, dx$$

A.9.3. Gauss Point Loop

```
for gauss_point in range(self.ngaus):
    N_gp = self.N[gauss_point, :]  # Shape functions at the Gauss point
    dN_dxi_gp = self.Nxi[gauss_point, :]  # Shape function derivatives wrt reference coordinate at the Gauss point

    # Compute the Jacobian
    J = dN_dxi_gp @ x_element
```

```
    # Compute the differential volume
    dV = self.wgp[gauss_point] * np.abs(J)

    # Compute the derivative of shape functions with respect to the physical coordinate
    dN_dx_gp = dN_dxi_gp / J

    # Update the local diffusion matrix
    K_element += np.outer(dN_dx_gp, dN_dx_gp) * dV
```

Mathematically:

- **Shape Functions** ($N_i(\xi)$): These are the shape functions evaluated at the Gauss point ξ_k. For linear elements:
$$N_1(\xi) = \frac{1-\xi}{2}, \quad N_2(\xi) = \frac{1+\xi}{2}$$
- **Shape Function Derivatives** ($\frac{dN_i(\xi)}{d\xi}$): These are the derivatives of the shape functions with respect to the reference coordinate ξ:
$$\frac{dN_1(\xi)}{d\xi} = -\frac{1}{2}, \quad \frac{dN_2(\xi)}{d\xi} = \frac{1}{2}$$
- **Jacobian** (J): The Jacobian J represents the transformation from the reference element ξ to the physical element x. It is computed as:
$$J = \frac{dx}{d\xi} = \frac{dN(\xi)}{d\xi} \cdot x_e = \frac{dN_1(\xi)}{d\xi} \cdot x_1 + \frac{dN_2(\xi)}{d\xi} \cdot x_2$$
- **Differential Volume** (dV): The differential volume (or length in 1D) is:
$$dV = w_k \cdot |J|$$
where w_k is the weight associated with the Gauss point.
- **Derivative of Shape Functions with Respect to Physical Coordinate** ($\frac{dN_i(x)}{dx}$): The derivatives of the shape functions with respect to x are:
$$\frac{dN_i(x)}{dx} = \frac{dN_i(\xi)}{d\xi} \cdot \frac{1}{J}$$
- **Update Local Diffusion Matrix** ($\mathbf{K}_e$): The local diffusion matrix is updated by summing the contributions from each Gauss point:
$$\mathbf{K}_e += \left(\frac{dN_i(x)}{dx} \cdot \frac{dN_j(x)}{dx} \right) \cdot dV$$

A.9.4. Assembly of the Global Matrix

```
for i in range(n_local_nodes):
    for j in range(n_local_nodes):
        K_global[element_nodes[i], element_nodes[j]] += K_element[i, j]
```

Mathematically:

- **Assembly**: After computing the local matrix $\mathbf{K}_e$, it is assembled into the global matrix $\mathbf{K}$ by adding the local contributions to the corresponding global positions:
$$\mathbf{K}_{ij} += \mathbf{K}_e^{ij} \quad \text{for } i, j = 1, 2$$

A.9.5. Return the Global Matrix

```
return K_global.tocsc()  # Convert to compressed sparse column format for efficiency
```

Finally, the global diffusion matrix $\mathbf{K}$ is converted to Compressed Sparse Column (CSC) format for efficient storage and computation.

A.10. Load Vector (Source Term) in FEM

The load vector $\mathbf{F}$, which arises from the source term $f(x,t)$ in the 1D Burgers' equation, is defined as:

$$\mathbf{F}_i = \int_\Omega f(x,t) N_i(x)\, dx \tag{102}$$

where $N_i(x)$ is the shape function associated with the ith node, and $f(x,t)$ is the source term.
Using the transformations and Gaussian quadrature:

$$\mathbf{F}_i \approx \sum_{k=1}^{n_{\text{gauss}}} w_k f(\xi_k) N_i(\xi_k) |J| \tag{103}$$

Here:

- $f(\xi_k)$ is the value of the source term evaluated at the Gauss points.
- $N_i(\xi_k)$ is the value of the shape function evaluated at the Gauss points.
- $|J|$ is the absolute value of the Jacobian, accounting for the scaling between the reference and physical element.
- w_k is the weight associated with the Gauss point.

A.11. Local to Global Matrix Assembly

Once the matrices for each element are computed, these local matrices are assembled into the global matrices by summing up the contributions from all elements.

A.12. Implementation Details of the Forcing Vector

The forcing vector $\mathbf{F}$ arises from the external forcing term in the 1D Burgers' equation. Below is a detailed explanation of the implementation, relating each part of the code to its corresponding mathematical expression.

A.12.1. Initial Setup

```
n_nodes = len(self.X)  # Number of nodes
n_elements, n_local_nodes = self.T.shape  # Number of elements and nodes per element
F_global = np.zeros(n_nodes)  # Initialize global forcing vector
```

In mathematical terms:

- **Number of Nodes** (n_{nodes}): This is the total number of nodes in the finite element mesh.
- **Number of Elements** (n_{elements}) and **Nodes per Element** ($n_{\text{local nodes}}$): These define the structure of the mesh. For 1D linear elements, $n_{\text{local nodes}}$ is typically 2.
- **Global Forcing Vector** ($\mathbf{F}_{\text{global}}$): This vector will store the assembled contributions from all elements:

$$\mathbf{F} = \sum_{e=1}^{n_{\text{elements}}} \mathbf{F}_e$$

where $\mathbf{F}_e$ is the local forcing vector for the e-th element.

A.12.2. Element Loop

```
for elem in range(n_elements):
    element_nodes = self.T[elem, :] - 1  # Nodes of the current element (adjusted for 0-based indexing)
    x_element = self.X[element_nodes].reshape(-1, 1)  # Physical coordinates of the element's nodes

    F_element = np.zeros(n_local_nodes)  # Initialize local forcing vector
```

Mathematically:

- **Element Nodes** (element_nodes): These are the indices of the nodes that belong to the current element. If the e-th element is defined by nodes i and j, then:

$$\text{element\_nodes} = [i, j]$$

- **Physical Coordinates of Element Nodes** (x_{element}): These are the actual x-coordinates of the nodes of the element:

$$x_{\text{element}} = \begin{bmatrix} x_i \\ x_j \end{bmatrix}$$

- **Local Forcing Vector** ($\mathbf{F}_e$): This represents the element-wise contribution to the global forcing vector:

$$\mathbf{F}_e = \int_{\Omega_e} f(x) N_i(x)\, dx$$

  where $f(x)$ is the external forcing function.

A.12.3. Gauss Point Loop

```
for gauss_point in range(self.ngaus):
    N_gp = self.N[gauss_point, :]  # Shape functions at the Gauss point
    dN_dxi_gp = self.Nxi[gauss_point, :]  # Shape function derivatives wrt reference coordinate at the Gauss point

    # Compute the Jacobian
    J = dN_dxi_gp @ x_element

    # Compute the differential volume
    dV = self.wgp[gauss_point] * np.abs(J)

    # Compute the physical coordinate at the Gauss point
    x_gp = N_gp @ x_element

    # Compute the forcing function at the Gauss point
    f_gp = 0.02 * np.exp(mu2 * x_gp)

    # Update the local forcing vector
    F_element += f_gp * N_gp * dV
```

Mathematically:

- **Shape Functions** ($N_i(\xi)$): These are the shape functions evaluated at the Gauss point ξ_k. For linear elements:

$$N_1(\xi) = \frac{1-\xi}{2}, \quad N_2(\xi) = \frac{1+\xi}{2}$$

- **Shape Function Derivatives** ($\frac{dN_i(\xi)}{d\xi}$): These are the derivatives of the shape functions with respect to the reference coordinate ξ:

$$\frac{dN_1(\xi)}{d\xi} = -\frac{1}{2}, \quad \frac{dN_2(\xi)}{d\xi} = \frac{1}{2}$$

- **Jacobian** (J): The Jacobian J represents the transformation from the reference element ξ to the physical element x. It is computed as:

$$J = \frac{dx}{d\xi} = \frac{dN_1(\xi)}{d\xi} \cdot x_1 + \frac{dN_2(\xi)}{d\xi} \cdot x_2$$

- **Differential Volume** (dV): The differential volume (or length in 1D) is:

$$dV = w_k \cdot |J|$$

  where w_k is the weight associated with the Gauss point.

- **Physical Coordinate at the Gauss Point** ($x(\xi_k)$): The physical coordinate at the Gauss point is:

$$x(\xi_k) = N_1(\xi_k)x_1 + N_2(\xi_k)x_2$$

- **Forcing Function at the Gauss Point** ($f(\xi_k)$): The forcing function at the Gauss point is:

$$f(\xi_k) = 0.02 \cdot \exp(\mu_2 \cdot x(\xi_k))$$

- **Update Local Forcing Vector** ($\mathbf{F}_e$): The local forcing vector is updated by summing the contributions from each Gauss point:

$$\mathbf{F}_e += f(\xi_k) \cdot N_i(\xi_k) \cdot dV$$

A.12.4. Assembly of the Global Vector

```
for i in range(n_local_nodes):
    for j in range(n_local_nodes):
        F_global[element_nodes[i]] += F_element[i]
```

Mathematically:

- **Assembly**: After computing the local vector $\mathbf{F}_e$, it is assembled into the global vector $\mathbf{F}$ by adding the local contributions to the corresponding global positions:

$$\mathbf{F}_i += \mathbf{F}_e^i \quad \text{for } i = 1, 2$$

A.12.5. Return the Global Forcing Vector

```
return F_global  # Return the assembled global forcing vector
```

Finally, the global forcing vector $\mathbf{F}$ is returned, containing the contributions from all elements in the mesh.