Cite as: Camacho, J.: Intrusive Polynomial Chaos for Uncertainty Quantification of Supersensitivity in the Burgers' Equation Using OpenFOAM. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Intrusive Polynomial Chaos for Uncertainty Quantification of Supersensitivity in the Burgers' Equation Using OpenFOAM

Developed for OpenFOAM-v2306
Requires: pyFoam, chaosPy

Author:
Javier I. Camacho
Lund University
javier.camacho@brand.lth.se

Peer reviewed by:
Assoc. Prof. Marcus Runefors
Dr. Saeed Salehi
Khoder Alhamwi Alshaar

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like to learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 21, 2025

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- Step-by-step guide on running a modified deterministic (`burgersFoam`) and stochastic (`gPCBurgersFoam`) Burger's equation solver in OpenFOAM.

The theory of it:

- Explanation of the Burgers equation as a model for Uncertainty Quantification (UQ) in non-linear PDEs, focusing on the influence of uncertainties in boundary conditions.
- Overview of UQ in CFD, focusing on Intrusive Generalised Polynomial Chaos (gPC).
- The formulation of Generalised Polynomial Chaos Expansion (gPCE) and Galerkin projection.
- Description of Galerkin projection and coefficients.

How it is implemented[1]:

- `burgersFoam`: Detailed breakdown of the deterministic base solver structure (scalarTransportFoam) within OpenFOAM used to implement Burger's equation solver. Overview of the solver's fundamental steps, focusing on deterministic aspects before adding stochastic elements.
- `gPCBurgersFoam`: Explanation of each step on how to implement a solver to apply intrusive gPC methods, including gPCE integration and the Galerkin projection.

How to modify it:

- Instructions on modifying the solver to incorporate precomputed Galerkin coefficient.
- Addition of pre- and post-processing tools to compute the Galerkin coefficients ,mean, variance, and uncertainty for analysing effects on the quantity of interest from boundaries condition under random perturbation.

[1] Solvers `burgersFoam` and `gPCBurgersFoam` from the repository in Ref. [1]

Prerequisites

For readers to fully benefit from this report guide, they should be familiar with the following areas:

- Fundamentals of Partial Differential Equations (PDEs) and Fluid Mechanics.
- Basic understanding of Probability and Statistics is required. It is recommended to be familiar with topics such as random variables, probability distributions, and statistical moments. For further details, refer to the chapters "Basic Concepts of Probability Theory" in [2] and "Measure and Probability Theory" in [3].
- Hands-on Experience in OpenFOAM: Including navigating its environment, configuring case files, and locating files, classes, and functions within the source code.
- Basic C++ and Python Programming Skills

Contents

Nomenclature

Acronyms

1D	One-Dimension
CFD	Computational Fluid Dynamics
FVM	Finite Volume Method
gPC	Generalised Polynomial Chaos
gPCE	Generalised Polynomial Chaos Expansion
IPC	Intrusive Polynomial Chaos
PDE	Partial Differential Equation
PDF	Probability Density Function
QoI	Quantity of Interest
RDE	Random Differential Equation
SDE	Stochastic Differential Equation
SiP	Simulation Input Parameters
UQ	Uncertainty Quantification
V&V	Verification and Validation

English symbols

$\mathcal{U}$	Uniform Distribution
$\tilde{u}(x,t;\omega)$	Approximated Stochastic Solution
A	Slope at Transition Layer Location
a, b	Support of the Probability Density Function Upper and Lower Bound
P	Polynomial Order
S	Support of the Probability Density Function
$u(x,t)$	Deterministic Solution
$u(x,t;\omega)$	Stochastic Solution
z	Transition Layer Location
$\mathbb{E}[\cdot]$	Expectation Operator

Greek symbols

δ	Perturbation Applied to Boundary
δ_{ij}	Kronecker Delta
ϵ	The Maximum Value within the Uniform Distribution of Perturbation δ
γ_i	Normalisation Non-Zero Constant Factor
μ	Mean or Expected Value
ν	Kinematic Viscosity
ω	Random Parameter
Φ	Orthogonal Polynomial Basis Functions
ϕ	Flux
$\rho(\xi)$	Probability Density Function used as a Weight
σ	Standard Deviation
ξ	Random Variable or Germ

Superscripts

n	previous iteration step
$n+1$	current iteration step

Subscripts

i, j, k	Polynomial or Coefficient Index, non-negative integers
ex	exact

"Essentially, all models are wrong, but some are useful."

— George E.P. Box [4].

Chapter 1

Introduction

This chapter briefly introduces the concepts of Verification and Validation (V&V), which are essential to ensuring the accuracy and reliability of numerical simulations for their intended applications. The significance of Uncertainty Quantification (UQ) in Computational Fluid Dynamics (CFD) is discussed, providing a general overview of the topic.

1.1 Verification and Validation (V&V)

In CFD modelling and simulation, rigorous V&V are important for establishing accuracy, reliability, and credibility of the simulations. These processes address two fundamental questions: verification asks, *"Are we solving the equations, right?"* while validation focuses on, *"Are we solving the right equations?"*. Through the V&V process—viewed as a continuous improvement effort—confidence is established in the model's ability to computationally reproduce system behaviour that represents "reality" and predict it accurately within specified parameters, based on the model's intended use. While the interpretation of the validation definition in V&V is still under debate, readers are encouraged to consult ASME standards (Ref.[5, 6]) for a comprehensive understanding of V&V, alongside Roache's perspective in Ref. [7]. As illustrated in Figure 1.1, the V&V framework integrates model development, simulation, and UQ. Verification ensures that the mathematical model is implemented correctly within the computational framework, while the validation process compares model simulation outputs against empirical data to determine the extent of agreement.

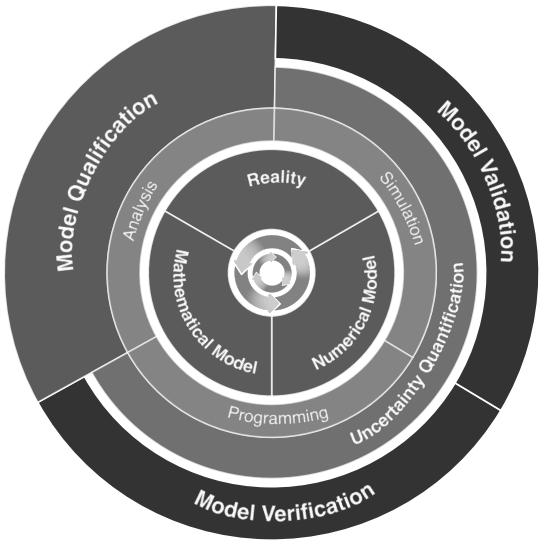

Figure 1.1: Overview of the CFD Modelling and Simulation Process. (Diagram inspired from Ref. [8, 9])

1.2 Uncertainty Quantification in CFD

1.2.1 What is Uncertainty Quantification?

"...uncertainty quantification can be broadly defined as the science of identifying, quantifying, and reducing uncertainties associated with models, numerical algorithms, experiments, and predicted outcomes or quantities of interest." [10]

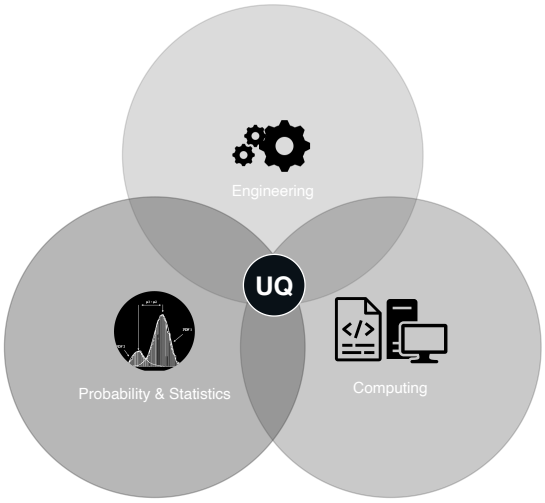

Figure 1.2: Main Disciplines Involved in UQ (Diagram inspired from Ref. [11])

To address the inherent uncertainties in simulations, which may arise from input variability (*Aleatory Uncertainties*), numerical errors, or limited knowledge of a physical phenomena (*Epistemic Uncertainties*). UQ involves several key steps as described by McClarren in Ref. [12]:

1. Identifying quantities of interest (QoIs).
2. Modelling uncertainties in inputs.
3. Narrowing down uncertain inputs.
4. Propagating uncertainties through simulations
5. Assessing their impact on the QoIs.

As illustrated in Figure 1.2, UQ is inherently multidisciplinary, requiring expertise from three key domains: Engineering, Probability and Statistics, and Computing. The synergy between these disciplines ensures that uncertainties are rigorously characterised, efficiently propagated, and effectively analysed within a robust computational framework.

1.2.2 Uncertainty Propagating Methods (Intrusive vs Non-Intrusive)

Within the context of UQ for CFD simulations, two approaches or techniques for UQ are commonly discussed in the literature [10, 13]: *non-intrusive* and *intrusive methods*. Deterministic simulations at particular parameter values (realisations) are used by non-intrusive methods (e.g., Monte Carlo and Sampling-Based Methods) to create approximations of the output function. The deterministic solver is treated as a "black box," thus necessitating no code modifications, this constitutes a significant advantage. While relatively easier to implement, these methods necessitate extensive sampling for accurate uncertainty quantification.

Conversely, intrusive approaches directly modify the governing equations or the solution algorithm to incorporate uncertainty quantification, for example, through *generalised polynomial chaos expansions* (gPC). Although computationally efficient and adaptable to uncertainty propagation, this method necessitates substantial modifications to the simulation code and a comprehensive understanding of the underlying numerical and mathematical principles.

1.2.3 A Word of Caution: Challenges in UQ Theory

As noted by Sullivan in 2015 on page 6 of Ref. [3], Uncertainty Quantification (UQ) is a relatively young discipline compared to mature fields like linear algebra and single-variable complex analysis, which are founded on established classical theorems by Cauchy, Gauss, and Hamilton. UQ's development is closely tied to practical applications, focusing on solving specific problems with appropriate methods instead of adhering to a unified theoretical framework. While the UQ framework employs sophisticated methods, a unifying theoretical structure remains absent, a characteristic that Sullivan highlighted and which continues to hold relevance today. instead of adhering to

1.3 Illustrative Scenario for UQ Potential Application

A CFD simulation problem and the corresponding UQ, may involve multiple, interconnected, and complex phenomena. Besides numerical uncertainties (e.g., round-off and discretisation errors), two main sources contribute to uncertainties in the model. Intrinsic model errors, including simplifying assumptions, approximations of physical processes, and grid-scale limitations (e.g., turbulence representation), form the primary source of inaccuracies. Assessing model errors often requires a problem-specific approach and additional expertise in the relevant field. The second source of uncertainty arises from input parameters, forcing functions, and initial and boundary conditions.

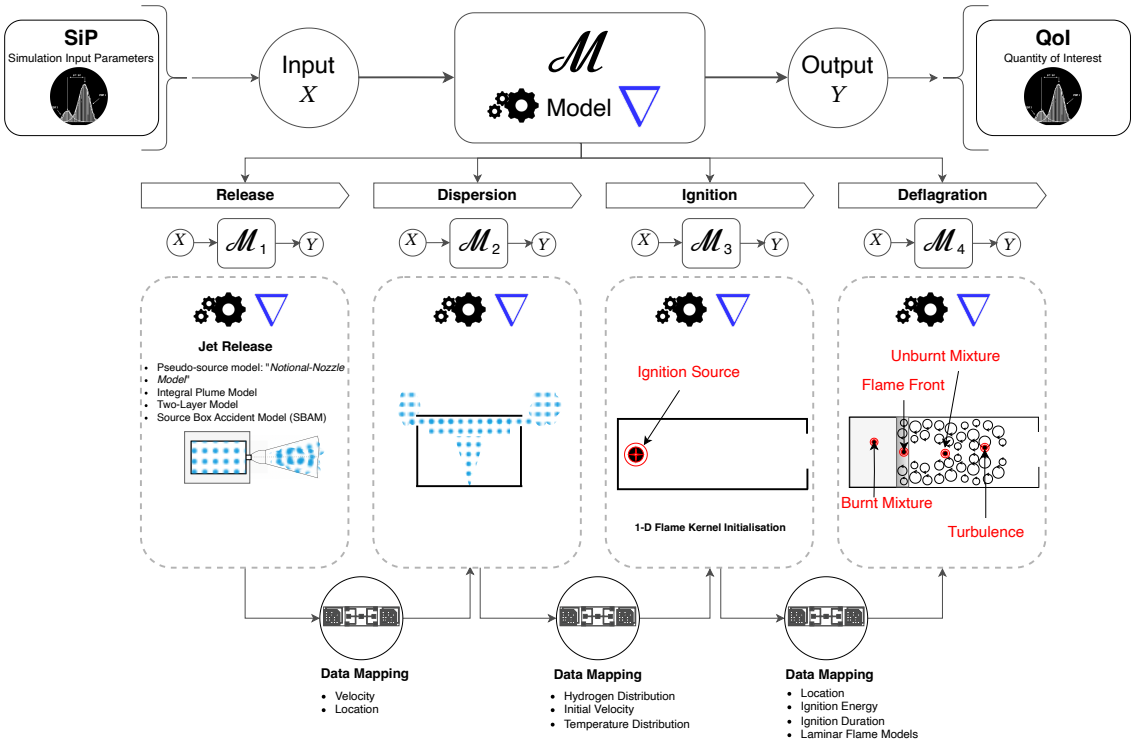

Figure 1.3: Illustration example of the complexity in simulating deflagration phenomena.
(Release phenomena simulation diagram inspired from Ref. [14], dispersion diagram inspired from Ref. [15]).

As an example, Figure 1.3 illustrates the challenges inherent in deflagrations [1] simulations within partially confined spaces. This complex nature emphasises the significance of UQ in CFD. The simulation input parameters (SiP) are subject to uncertainties at each stage of the process, which propagate through the system, impacting the final QoI. To enhance credibility and accuracy during the validation process of simulations, robust uncertainty quantification methods are indispensable.

1.3.1 Simulation Input Parameters

Sources of uncertainty include, but are not limited to, simulation input parameters (SiP) such as boundary conditions, initial conditions, and material/fluid properties (e.g., enclosure, fluid properties). The quantification and systematic propagation of uncertainties through the CFD model enable an assessment of their influence on the resulting outputs.

1.3.2 Sequential Modelling Stages

The simulation process involves multiple interconnected stages, each contributing to the overall uncertainty of the system. These stages represent the physical processes involved in hydrogen combustion, from its release to its deflagration, and highlight the uncertainties associated with each step. By quantifying and propagating these uncertainties, the sequential modelling framework ensures a comprehensive understanding of the system's behaviour under variable conditions. Below is a breakdown example of the key stages in the simulation process:

- Release ($\mathscr{M}_1$): Models, such as the "Notional-Nozzle Model"[14] are employed to simulate the release of hydrogen. Data uncertainties related to velocity and release location are quantified and passed to subsequent stages.
- Dispersion ($\mathscr{M}_2$): For example, hydrogen dispersion models are employed to predict the gas distribution. Uncertainties in the concentration and velocity fields can affect downstream outcomes.
- Ignition ($\mathscr{M}_3$): Combustion initiation is represented by models that account for uncertainties in ignition energy, location, and laminar flame speed.
- Deflagration ($\mathscr{M}_4$): The combustion phase is simulated, incorporating uncertainties related to flame propagation, turbulence, and the interaction between burned and unburnt gases.

Note

It should be noted that the diagram in Figure 1.3 is subject to further revision and currently omits the finer details of individual steps to maintain the clarity of the overall workflow.

1.3.3 Propagation of Uncertainty

At every stage, uncertainties arising from the outputs of the preceding simulation model, such as the hydrogen concentration derived from dispersion, are incorporated as inputs for the following stage. The cumulative propagation of uncertainty significantly amplifies the overall uncertainty, highlighting the importance of a systematic procedure for uncertainty quantification for effective management and quantification of these effects.

1.3.4 QoI Analysis

The QoI encompasses simulation outputs such as peak overpressure, flame speed, and location. Integrating UQ into CFD is crucial for evaluating its influence on simulation results, especially in critical applications such as hydrogen storage and transportation, where safety is paramount. UQ facilitates more informed decision-making within design optimisation and risk assessment, particularly when confronting the intricacies of multi-scale scenarios.

[1] Deflagration: combustion wave propagating at subsonic velocity.[16]

1.4 Objective, Scope, and Report Structure

1.4.1 Objectives

This report aims to provide a comprehensive tutorial for integrating UQ methods into CFD solvers in OpenFOAM, focusing on the one-dimensional viscous Burgers' equation. By bridging theoretical concepts of gPC with practical implementation in OpenFOAM, the report intends to:

- Equip users with a foundational understanding of UQ and intrusive gPC.
- Describe a systematic verification of deterministic and stochastic solvers using exact solutions.
- Offer pre- and post-processing tools to streamline the implementation and analysis of gPC frameworks.
- Support future improvements and expansions of the implemented solvers in OpenFOAM by enhancing adaptability and scalability.

1.4.2 Scope

The scope of this report includes:

- Specific Problem: A special case of the 1D viscous Burgers' equation, exhibiting *Supersensitivity*, is examined, building upon Xiu and Karniadakis's work on "Supersensitivity due to Uncertain Boundary Conditions" (Ref. [17]).
- Modifications Implementation and Verification: Detailed guidance on modifying and verifying two different solvers, `burgersFoam` and `gPCBurgersFoam`, for deterministic and stochastic studies.
- Tool Development: Introduction of pre- and post-processing tools designed to facilitate efficient execution and analysis of intrusive gPC solvers in OpenFOAM.
- Uncertainty Quantification: Application of UQ methodologies to study the supersensitivity phenomenon in the viscous Burgers' equation, with an emphasis on gPC for random input boundary condition scenario.
- Practical Relevance: Providing a structured approach that supports broader adoption of UQ methods by users, with potential extensions to more complex CFD scenarios.

1.4.3 Report Structure

This report is divided into six chapters, including this introduction. The structure is as follows:

- Chapter 2: Background on Intrusive Generalised Polynomial Chaos
- Chapter 3: Existing Solvers: Structure and Implementation Details
- Chapter 4: Solvers Modifications: Implementation Details and Supporting Tools
- Chapter 5: Verification and Uncertainty Quantification
- Chapter 6: Conclusions and Future Work

Chapter 2

Background on Intrusive Generalised Polynomial Chaos

This chapter outlines a comprehensive framework for the formulation of the viscous Burgers' equation, subjected to perturbed boundary conditions. It aims to establish a background in both the viscous Burgers' equation and the Generalised Polynomial Chaos (gPC) method, with the objective of enhancing the existing OpenFOAM solvers implementation, which will be described in subsequent chapters. The work of Xiu and Karniadakis, titled *"Supersensitivity due to Uncertain Boundary Conditions"*(Ref. [17]), serves as a primary reference for gPC method formulations and results verification. Moreover, auxiliary references, such as the book titled *"Numerical Methods for Stochastic Computations: A Spectral Method Approach"* [2], along with other documented sources, are incorporated as relevant information. Two types of perturbations ($\delta(0, \epsilon)$) are considered: deterministic and random.

For deterministic perturbations, the boundary conditions are modified by introducing predefined perturbations or deviations from their nominal values. Two formulations are described: an exact steady-state solution, which serves as a benchmark for verification, and the conservative form of the Burgers' equation for a Finite Volume Method (FVM) numerical implementation in OpenFOAM. In the case of random perturbations, the boundary conditions are represented using a random variable component and the intrusive *generalised polynomial chaos* method, as introduced by Xiu and Karniadakis in Ref. [18]. This approach expands a stochastic process, such as random perturbations in boundary conditions, using orthogonal polynomial functionals from the Askey scheme (see Ref. [19]). The probability density function of the random variable (e.g., uniform or normal) is incorporated through the selection of an appropriate polynomial basis.

As demonstrated by Xiu and Karniadakis in Ref. [17], both deterministic and stochastic perturbations exhibit the phenomenon of *supersensitivity* in the context of the viscous Burgers' equation (with $x \in [-1, 1]$). Even small perturbations (e.g., 10% of the nominal value) can cause substantial shifts in the transition layer's location—the spatial position where the velocity is zero—in the steady-state solution. In this report, the transition layer's location is used as a key reference metric for comparing implementation results against those reported in Ref. [17], with a series of verification and uncertainty quantification case studies outlined in Chapter 5.

2.1 The Burgers' Equation: An Important Tool in CFD

The Burgers' equation is a nonlinear partial differential equation, analogous to the Navier-Stokes equation, omitting the pressure term. This equation serves as a prototype model in fluid dynamics, facilitating the study of fundamental phenomena, including shock formation and diffusion, and provides a valuable framework for verifying novel numerical implementations. A comprehensive analysis of the Burgers' equation is provided in Ref. [20]. The 1D viscous Burgers' equation in its non-conservative form can be expressed as,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in \mathbb{R}, \quad t \geq 0, \tag{2.1}$$

where

$\frac{\partial u}{\partial t}$: Temporal rate of change of the velocity $u(x,t)$ term, which provide information on how the velocity changes with respect to time at a fixed spatial location.

$u\frac{\partial u}{\partial x}$: Nonlinear convective term, which accounts for the advection (i.e., transport) of velocity due to the flow itself,, which provide information on how the velocity changes spatially ($\frac{\partial u}{\partial x}$) and is scaled by the velocity u.

$\nu\frac{\partial^2 u}{\partial x^2}$: Viscous diffusion of velocity term, where $\nu > 0$ is the kinematic viscosity. It smooths out velocity gradients over time and introduces parabolic behaviour to the equation.

2.1.1 Modelling Versatility of Burgers' Equation

The versatility of the Burgers' equation arises from its dual nature, exhibiting wave-like (hyperbolic) and diffusion-like (parabolic) characteristics, depending on the comparative influence of convective (inertial) and viscous forces. The hyperbolic regime, dominated by convective forces, involves wave propagation and sharp fronts, such as shocks in the inviscid Burgers' equation. Although it can model nonlinear acoustic waves and shocks, the model lacks the complexity to model contact discontinuities. The parabolic regime is driven by viscous forces, leading to smooth, diffusion-like solutions. For more details about the Burgers' equation see Ref. [20, 21] and Section 4.8 in Ref. [22]. Figure 2.1 illustrates the impact of viscosity variations, which may be modelled as an uncertainty parameter, which is characterised by its range and probability distribution (e.g., uniform, Gaussian).

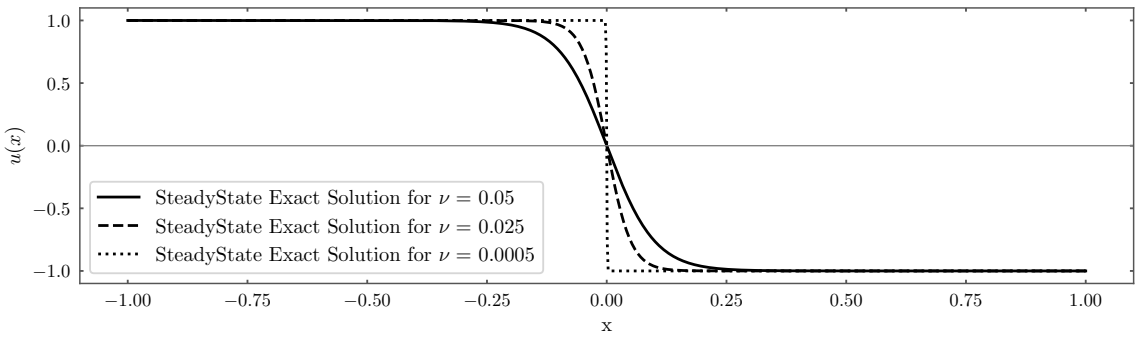

Figure 2.1: Viscosity Effect on Steady-State Burgers' Equation (with $x \in [-1, 1]$)

The ability to transition between these regimes makes the Burgers' equation a versatile benchmark problem in CFD for evaluating numerical methods. The versatility of the equation within the framework of UQ allows for the demonstration of how uncertainties in input parameters, such as initial/boundary conditions or viscosity, propagate through systems characterised by mixed dynamics. Through implementing techniques such as intrusive polynomial chaos, the Burgers' equation enables an analysis of the output of quantities of interest in simulations under uncertain conditions.

2.1.2 Burgers' Equation with Perturbed Boundary Conditions

This report details an analysis of a particular case of the one-dimensional viscous Burgers' equation (Eq. (2.2)) expressed in its non-conservative form, with the spatial domain restricted to $x \in [-1, 1]$, where small perturbations (δ) are introduced at the left boundary.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1], \quad t \geq 0. \tag{2.2}$$

This case is relevant for both understanding and quantifying the effects of uncertain boundary conditions, and allows for direct comparison with the results of Xiu and Karniadakis's work [17]. The boundary and initial conditions are given by

$$u(-1, t) = 1 + \delta, \quad u(1, t) = -1, \quad t \geq 0, \tag{2.3}$$

$$u(x, 0) = 0, \quad x \in [-1, 1], \tag{2.4}$$

where the Dirichlet boundary conditions (Eq. (2.3)) defines the velocity at the boundaries of the domain $x = -1$ and $x = 1$. Depending on the implementation case, the parameter δ introduces a small perturbation to the left boundary condition ($x = -1$), which can be deterministic $0 < \delta \ll O(1)$ or random where $\delta \in (0, \epsilon)$ is a random variable in $(0, \epsilon)$ with $\epsilon \ll O(1)$ and a predefined continuous probability distribution function (PDF) $f(\delta)$.

Transition Layer Location (z) and Supersensitivity Phenomenon

A transitional layer region of rapid variation is present in the solution of the viscous Burgers' equation (Eq. (2.2)), this region extends over a distance proportional to the viscosity ν, as ν tends to zero. The position of the transition layer, z, identified as the point where the solution profile u(z) is zero, exhibits temporal variability, and its final steady-state position demonstrates significant sensitivity to the imposed boundary conditions. Lorentz [23] first reported on the phenomenon, subsequently termed *supersensitivity* in *deterministic* asymptotic analysis.

2.2 Deterministic Supersensitivity Formulation

A solution formulation is presented in this section for the viscous Burgers' equation (Eq. (2.2)) under a small, deterministic perturbation ($\delta > 0$) of the upstream boundary condition. Two numerical methods are employed to address the deterministic supersensitivity problem. Initially, the exact steady-state solution to the viscous Burgers' equation is provided, which is defined implicitly as a non-linear equation. The iterative solution algorithm, incorporating chosen parameters and an initial estimate, is described in the verification chapter, which includes a Python implementation. The viscous Burgers' equation is subsequently recast in conservative form from its non-conservative counterpart, thus providing a suitable framework for the FVM implementation within OpenFOAM.

2.2.1 Exact Solution

The exact steady-state solution of the 1D viscous Burgers' equation (Eq. (2.2)), as described in Ref. [17], is expressed as follows,

$$u(x) = -A\tanh\left[\frac{A}{2\nu}(x - z_{ex})\right], \tag{2.5}$$

where z_{ex} is the location of the transition layer where $u(z_{ex}) = 0$, and $-A = \frac{\partial u}{\partial x}\Big|_{x=z_{ex}}$, represents the slope at this location. By applying the boundary conditions Eq. (2.3) to Eq. (2.5),

$$A\tanh\left[\frac{A}{2\nu}(1 + z_{ex})\right] = 1 + \delta, \quad A\tanh\left[\frac{A}{2\nu}(1 - z_{ex})\right] = 1, \tag{2.6}$$

then two unknowns, A and z_{ex}, are required to be solved for. First, by elimination of z_{ex} and obtaining a single equation for A,

$$(1+\delta+A^2)\tanh\left(\frac{A}{\nu}\right)=(2+\delta)A. \tag{2.7}$$

By first solving Eq. (2.7) for the slope A, the value of the location of the transition layer z_{ex} can subsequently be determined using one of the equations in Eq. (2.6). Iterative methods are needed to solve these non-linear equations, the details about the python implementation are provided in the Section 4.4.1. It is important to note that the convergence of the solution depends strongly on the initial guess, a consequence of the *supersensitivity* of the original problem defined in Eq. (2.2) and Eq. (2.3).

2.2.2 Finite Volume Method Formulation

Formulating the viscous Burgers' equation is a necessary preliminary step before engaging with the complexities of the intrusive polynomial chaos method, facilitating subsequent modifications within the OpenFOAM solver's implementation. This step establishes a fundamental comprehension of the equation's derivation. This clarifies the differences between deterministic and stochastic (intrusive polynomial chaos) formulations, thereby improving both the theoretical comprehension and practical implementation of the OpenFOAM method.

The numerical discretisation method implemented in OpenFOAM is the FVM[1], which is widely used for its conservation properties in solving fluid dynamics and multi-physics problems. FVM ensures the conservation of physical quantities across control volumes. As described in *Thermal Systems and Models*, Section 8.2.1.6 (page 420) in Ref. [25], the loss experienced by one cell results in a corresponding gain for another cell. Through a conservative scheme, the amplification of numerical errors affecting the conservation of physical quantities is prevented. However, this approach cannot mitigate (or dampen) the inherent instability present in the physical system.

To ensure the total conserved quantity remains consistent across the computational domain, conservative formulations are the preferred approach in numerical methods for scalar conservation laws. The implementation of the Eq. (2.2) in OpenFOAM, required the conservative form version of the viscous Burgers' equation (see Ref. [20, 26, 27, 28]) that can be expressed as

$$\frac{\partial u}{\partial t}+\frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right)=\nu\frac{\partial^2 u}{\partial x^2}. \tag{2.8}$$

The different equation forms still remain analogous (Eq. (2.2) and Eq. (2.8)), with the difference in the nonlinear term where the relationship between the Burgers' equation forms can be seen by applying the product rule to the nonlinear advection term $u\frac{\partial u}{\partial x}$ as shown in Eq. (2.9) and Eq. (2.10).

$$\frac{\partial(u.v)}{\partial x}=v\frac{\partial u}{\partial x}+u\frac{\partial v}{\partial x},\quad u=v,\quad \rightarrow\frac{\partial(u.u)}{\partial x}=2u\frac{\partial u}{\partial x} \tag{2.9}$$

$$u\frac{\partial u}{\partial x}=\frac{1}{2}\frac{\partial(u.u)}{\partial x}=\frac{1}{2}\nabla\cdot(u.u) \tag{2.10}$$

[1]For more details see OpenFOAM guide/Finite volume method (OpenFOAM) in Ref. [24].

2.3 Stochastic Supersensitivity Formulation

This section addresses the supersensitivity problem, as defined in equation Eq. (2.2), under conditions of stochastic perturbation. The formulation of the generalised polynomial chaos (gPC) expansion is presented, along with a detailed outline of the steps required to derive the corresponding deterministic system of equations.

2.3.1 Procedure Overview

A high-level schematic of the workflow is presented in Figure 2.2, providing an overview of how the implemented formulation will be utilised. This schematic precedes the detailed mathematical formulation of each step, aiming to give readers a clear and concise outline of the process. The probability density functions (PDFs) in Figure 2.2 are for illustration only and do not imply that inputs must always follow a normal distribution. In practice, generalised polynomial chaos (gPC) is capable of handling diverse probability distributions, including but not limited to normal and uniform distributions, depending on the estimated uncertainty characteristics of the problem. This distinguishes gPC from standard polynomial chaos (PC), which is dedicated to normal inputs, using *Hermite-chaos* polynomial functionals as basis functions. Regarding the solution, gPC provides the mean (μ) and standard deviation (σ) of the output fields. While the figure shows example PDFs, determining the actual solution PDF requires further analysis, which is out of the scope of this report.

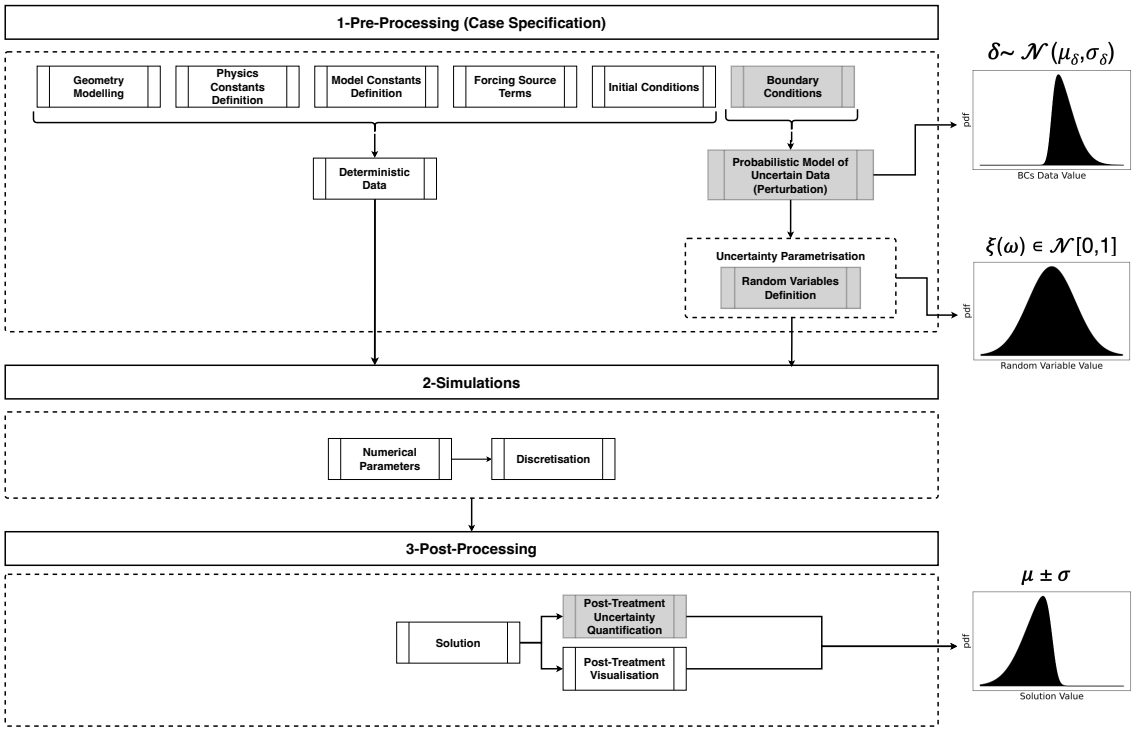

Figure 2.2: Illustration of the different steps of uncertainty propagation using intrusive gPC method.

The procedure commences with a pre-processing phase, in which deterministic data and probabilistic models of uncertain inputs, including boundary conditions (see Figure 2.2, top plot) based on a known probability of the boundary condition uncertain data and random variables (parametrisation), are established. In the parametrisation step, a probability distribution function (PDF) is assigned to each random variable used to model these uncertainties (see Figure 2.2, middle plot). In the simulation phase, numerical parameters and discretisation schemes are utilised to propagate uncertainties through the coupled deterministic equations, which are derived (see Section 2.3.2) and

implemented in the solver using the intrusive gPC method (see Section 4.2). The post-processing stage involves quantifying the moments (mean and standard deviation) of the quantities of interest (QoI), as illustrated in Figure 2.2 (bottom plot), enabling uncertainty quantification.

Note

Although the intrusive gPC method propagates uncertainties and provides statistical information (e.g., mean or expected value, variance, skewness) about the solution, it is not classified as a statistical method. Non-intrusive methods, such as Monte Carlo and stochastic collocation, are categorised as statistical methods because they generate statistical information from multiple simulation runs, unlike their intrusive counterparts.

2.3.2 Generalised Polynomial Chaos

The generalised polynomial chaos (gPC) is a spectral representation or expansion of a second-order (i.e., finite variance) stochastic processes by polynomial functionals $\Phi(\xi(\omega))$ of random variables $\xi(\omega)$ (where ξ is often called the *germ* [3, 13, 29]). It should be noted that, given that random variables are functions of the random parameter ω, the polynomial basis (also known as random trial basis functions) are, in fact, functionals of these functions.

The key advantage of the expansion lies in its ability to decompose a stochastic process into deterministic spatial-temporal functions, where each function is multiplied by a set of random basis polynomials that are entirely independent of both spatial and temporal variables. The solution of the Burgers' equation $u(x,t)$ is expanded to a random variable $u(x,t;\omega)$, considering that any second-order random field $u(x,t;\omega)$ can be expressed in this form (see page 14 on [30]),

$$u(x,t;\omega) = \sum_{i=0}^{\infty} \hat{u}_i(x,t)\Phi_i(\xi(\omega)), \tag{2.11}$$

where

$\xi(\omega)$ represents multi-dimensional random variables dependent on the random parameter ω.

$\Phi_i(\xi(\omega))$ are a set of orthogonal polynomial basis functions defined over the probability space of ξ.

$\hat{u}_i(x,t)$ are the deterministic coefficients or modes associated with the basis functions, where the index i is a non-negative integer.

Polynomial Truncation

Theoretically, the expansion exists within an infinite-dimensional stochastic space of ξ. The dimensionality of this space in practical applications is finite and defined by the number of random input parameters. Given that the boundary condition introduces δ as the only random input in Eq. (2.2), the infinite expansion in Eq. (2.11) is one-dimensional and it is truncated to a finite-dimensional form,

$$u(x,t;\omega) \approx \tilde{u}(x,t;\omega) = \sum_{i=0}^{P} \hat{u}_i(x,t)\Phi_i(\xi(\omega)), \tag{2.12}$$

where P is the highest degree (polynomial order) of the polynomial basis used.

Polynomial Basis Orthogonality Property

The generalised polynomial chaos basis functions are the orthogonal polynomial functions satisfying,

$$\langle \Phi_i(\xi), \Phi_j(\xi) \rangle = \mathbb{E}[\Phi_i(\xi)\Phi_j(\xi)] = \int_S \Phi_i(\xi)\Phi_j(\xi)\rho(\xi)d\xi = \gamma_i \delta_{ij}, \text{with} \quad \mathrm{i,j} \in \mathbb{N}, \tag{2.13}$$

$$\gamma_i = \mathbb{E}[\Phi_i(\xi)^2] = \langle \Phi_i^2 \rangle = \int_S \Phi_i^2(\xi)\rho(\xi)d\xi, \tag{2.14}$$

where

$\rho(\xi)$: weighting function[2], which is the PDF of the random variable ξ.

δ_{ij}: Kronecker delta.

$\Phi_i(\xi(\omega))$ and $\Phi_j(\xi(\omega))$: are orthogonal polynomial basis functions defined over the probability space of ξ.

$\langle \cdot, \cdot \rangle$: denotes the inner product (often referred to as the ensemble average), another notation commonly seen in literature is the expectation operator $\mathbb{E}[\cdot]$, as utilised in Section *6.5 Nonlinear Problems* page 76 in Ref. [2].)

γ_i: normalisation non-zero constant factor, which will be 1 if the basis functions are normalised, and the basis polynomial functions will be *orthonormal*. As suggested in Ref. [17], this factor admits an analytical calculation, alternatively, a numerical computation, implemented in this report using `chaoPy` and normalised (using `normed` functionality) to maintain *orthonormality* of the polynomial basis, is employed.

S: support of the PDF $\rho(\xi)$ that depends on the selected polynomial basis.

The basis functions $\Phi(\xi)$ constitute a set of orthogonal polynomials defined on $\xi \in \mathbb{R}$, with respect to the weight function $\rho(\xi)$, representing the probability density function of the random variable ξ. A relationship is established between the distribution of the random variable ξ and the orthogonal polynomial family that constitutes its gPC basis. The stochastic solution formulation will incorporate this property in the following steps. For notational simplicity and to facilitate the derivation, the explicit dependence of Φ on $\xi(\omega)$ is temporarily suppressed, while its functional dependence is implicitly preserved.

Polynomial Basis Selection

In practical applications, the selection of the basis type (Φ) and order (M) for gPCE implementation in uncertainty quantification is typically guided by engineering judgement supported by available data and the known properties of the random variable. This process leads to the choice of the orthogonal polynomial basis Φ, which depends on the known or assumed probability distribution of ξ. Table 2.1 summarised the relationship between common random variables probability distributions and their corresponding polynomial basis.

Table 2.1: Correspondence of the polynomial chaos type (*Wiener–Askey*) and continuous random variable distributions (adapted from [18]).

Random Variables (ξ)	Orthogonal Polynomials Basis ($\Phi(\xi)$)	Support(S)
Gaussian	*Hermite-chaos*	$(-\infty, \infty)$
Gamma	*Laguerre-chaos*	$[0, \infty)$
Beta	*Jacobi-chaos*	$[a, b]$
Uniform	*Legendre-chaos*	$[a, b]$

[2]**Remark** As has been noted by T.J. Sullivan (see page 136 in Ref. [3]), in many references, particularly those written by physicists, the weight function $e^{-x^2}\,dx$ is commonly used for *Hermite* polynomials, whereas probabilists' often prefer $(2\pi)^{-1/2}e^{-x^2/2}\,dx$ or $e^{-x^2/2}\,dx$. While straightforward, shifting between these normalisations requires careful consideration to identify the precise normalisation used, especially when employing third-party software. In `Chaospy`, this distinction can be specified by using the `physicist=False` option when defining the *Hermite* polynomials, ensuring the probabilists' normalisation is applied.

2.3.3 Random Differential Equation

In a random differential equation (RDE), randomness is introduced via the parameters, initial/boundary conditions, or external forcing functions. These random effects vary in a regular manner, such as being continuous over time and space. The random nature of the boundary conditions in this work propagates through the system, resulting in solutions exhibiting stochastic behaviour. The probabilistic nature of the solution is modelled and quantified through the application of intrusive gPC. By employing this approach, the uncertainty present in the solution space is captured without affecting the deterministic characteristics of the governing equations for fixed realisations of the random inputs. This yields a scenario in which the governing equation is classified as a RDE and, by substitution of Eq. (2.11) into Eq. (2.2), the original governing equation is redefined as,

$$\frac{\partial u(x,t;\omega)}{\partial t} + u(x,t;\omega)\frac{\partial u(x,t;\omega)}{\partial x} = \nu\frac{\partial^2 u(x,t;\omega)}{\partial x^2}, \quad x \in [-1,1], \quad t \geq 0. \tag{2.15}$$

Note

In contrast, a stochastic differential equation (SDE) is a type of differential equation in which uncertainty plays a distinct role. The system is influenced by irregular processes, such as Wiener processes or Brownian motion, which introduce randomness directly into the dynamics of the equation. For further details, the reader is referred to Section 4.7, *"Random versus Stochastic Differential Equations"*, on page 115 in Ref. [10] , as this topic is beyond the scope of this report.

Expanded Random Burgers' Equation

By substituting u for $\tilde{u}$ from the truncated gPCE from Eq. (2.12) into the RDE Eq. (2.15) gives

$$\frac{\partial}{\partial t}\sum_{i=0}^{P}\hat{u}_i(x,t)\Phi_i(\xi) + \left(\sum_{i=0}^{P}\hat{u}_i(x,t)\Phi_i(\xi)\right)\frac{\partial}{\partial x}\left(\sum_{j=0}^{P}\hat{u}_j(x,t)\Phi_j(\xi)\right) = \nu\frac{\partial^2}{\partial x^2}\sum_{i=0}^{P}\hat{u}_i(x,t)\Phi_i(\xi). \tag{2.16}$$

The summation can be moved outside the differentiation because of the *linearity of differentiation property*, $(\alpha \cdot f + \beta \cdot g)' = \alpha \cdot f' + \beta \cdot g'$. Additionally, the basis functions $\Phi_i(\xi)$ are independent of the differentiation variables (e.g., t or x) and can be treated as constants during differentiation. Therefore, differentiation is applicable solely to terms involving the differentiation variables. The order of summation and differentiation may be reversed without altering the result; the reorganised final expression is,

$$\sum_{i=0}^{P}\frac{\partial\hat{u}_i(x,t)}{\partial t}\Phi_i(\xi) + \sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i(x,t)\frac{\partial\hat{u}_j(x,t)}{\partial x}\Phi_i(\xi)\Phi_j(\xi) = \nu\sum_{i=0}^{P}\frac{\partial^2\hat{u}_i(x,t)}{\partial x^2}\Phi_i(\xi). \tag{2.17}$$

2.3.4 Galerkin Projection

The stochastic Galerkin projection[3] involves multiplying Eq. (2.17) by the same basis function $\Phi_k(\xi)$, and then integrating the result over the probability support S. (i.e., taking the expectation (inner product) with respect to the basis random variable ξ). The following provides a compact expression for the projection,

$$\mathbb{E}[\mathcal{L}[\tilde{u}(x,t,\omega)]\Phi_k(\xi(\omega))] = \langle\mathcal{L}[\tilde{u}(x,t,\omega)], \Phi_k(\xi(\omega))\rangle = 0, \quad k = 0,\ldots,P, \tag{2.18}$$

where $\mathbb{E}[\cdot]$ represents the expectation operator (equivalent notation to the ensemble average $\langle\cdot,\cdot\rangle$) and $\mathcal{L}[\cdot]$ the differential operator in the governing equation. By making use of the orthogonality property of the basis polynomials the final formulation result in a system of coupled deterministic

[3]The Galerkin projection method is a specific case of the *Method of Weighted Residuals* (MWR), where the weighting functions are chosen to be the same as the basis (trial) functions.

equations, where the unknowns are the expansion coefficients or modes $\hat{u}_k(x,t)$. The Figure 2.3 visually represents the Galerkin projection process for a two-term expansion, inspired by Figure 17.1 on page 621 in Ref. [29]. The random variable u lies in the full function space, while $\tilde{u}$ (the approximate solution) is its projected onto the subspace spanned by the polynomial chaos basis functions Φ_0, Φ_1. The expansion coefficients u_0, u_1 represent the weight of each polynomial basis function. The key property shown is that the residual ϵ (the difference between u and $\tilde{u}$) is orthogonal to the space covered by the basis functions, ensuring the best approximation in the L_2-norm sense. For more details, see Chapter 17 *"Intrusive Polynomial Chaos Methods for Forward Uncertainty Propagation"* in Ref. [29].

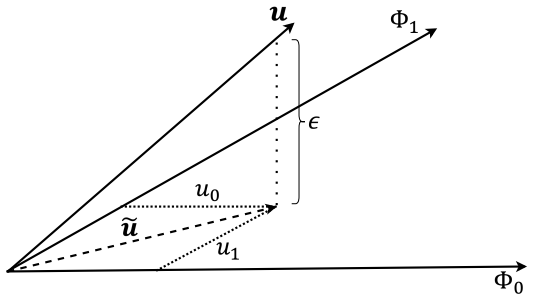

Figure 2.3: Illustration of the Galerkin projection

For notational simplicity and to facilitate the derivation, the explicit dependence of Φ on ξ, and $\hat{u}$ on (x,t) is temporarily suppressed, while its functional dependence is implicitly preserved. The Galerkin projection of the expanded Burgers' equation (Eq. (2.17)) can be expressed as follows, where the process will involve formulating each term of the Burgers' equation individually in a series of intermediate steps.

$$\mathbb{E}\left[\sum_{i=0}^{P}\frac{\partial \hat{u}_i}{\partial t}\Phi_i\Phi_k + \sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i\frac{\partial \hat{u}_j}{\partial x}\Phi_i\Phi_j\Phi_k\right] = \mathbb{E}\left[\nu\sum_{i=0}^{P}\frac{\partial^2 \hat{u}_i}{\partial x^2}\Phi_i\Phi_k\right]. \tag{2.19}$$

Term-by-Term Breakdown

Time Derivative Term

$$\mathbb{E}\left[\sum_{i=0}^{P}\frac{\partial \hat{u}_i}{\partial t}\Phi_i\Phi_k\right] \tag{2.20}$$

The linearity of the expectation $\mathbb{E}[\cdot]$, allow us to move the operator inside the summations since only the basis functions Φ depend on ξ, which leads to

$$\sum_{i=0}^{P}\frac{\partial \hat{u}_i}{\partial t}\mathbb{E}\left[\Phi_i\Phi_k\right], \quad \text{and by orthogonality property} \quad \mathbb{E}[\Phi_i\Phi_k] = \gamma_k\delta_{ki}, \tag{2.21}$$

$$\gamma_k = \mathbb{E}[\Phi_k^2] = \langle\Phi_k^2\rangle = \int_S \Phi_k^2(\xi)\rho(\xi)d\xi, \tag{2.22}$$

where γ_k is the normalisation non-zero constant factor (equal 1 for *orthonormal* basis functions) , then the time derivative term reduces to,

$$\gamma_k\frac{\partial \hat{u}_k}{\partial t}. \tag{2.23}$$

Nonlinear Advection Term

$$\mathbb{E}\left[\sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i\frac{\partial\hat{u}_j}{\partial x}\Phi_i\Phi_j\Phi_k\right] \tag{2.24}$$

Similarly, as in the previous term, we can apply the linearity of expectation to move the $\mathbb{E}[\cdot]$ operator inside the summations, thus obtaining

$$\sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i\frac{\partial\hat{u}_j}{\partial x}\mathbb{E}\left[\Phi_i\Phi_j\Phi_k\right], \tag{2.25}$$

and the triple product coefficients can be expressed as e_{ijk},

$$e_{ijk} = \mathbb{E}[\Phi_i\Phi_j\Phi_k] = \int_S \Phi_i(\xi)\Phi_j(\xi)\Phi_k(\xi)\rho(\xi)\,d\xi, \tag{2.26}$$

where,

$\rho(\xi)$: weight function, which is the probability density function (PDF) of the random variable

S: support of the PDF $\rho(\xi)$ that depends on the selected polynomial basis.

The triple product coefficients e_{ijk} within these equations remain invariant with respect to $\hat{u}$ and ξ, as they are constant. Pre-calculation of these values is feasible at the start of the simulation, given a priori knowledge of the basis functions and the corresponding PDF (see Section 2.3.4). This calculation can be conducted analytically, as suggested in Ref. [17], or numerically as has been implemented for this report using a *Gaussian* quadrature. Thus, the nonlinear advection term becomes,

$$\sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i\frac{\partial\hat{u}_j}{\partial x}e_{ijk}. \tag{2.27}$$

Diffusion Term

$$\mathbb{E}\left[\nu\sum_{i=0}^{P}\frac{\partial^2\hat{u}_i}{\partial x^2}\Phi_i\Phi_k\right]. \tag{2.28}$$

Moving the expectation $\mathbb{E}[\cdot]$ operator inside the summations, as explained before, the products yield

$$\nu\sum_{i=0}^{P}\frac{\partial^2\hat{u}_i}{\partial x^2}\mathbb{E}\left[\Phi_i\Phi_k\right], \quad \text{and by orthogonality property} \quad \mathbb{E}[\Phi_i\Phi_k] = \gamma_k\delta_{ki}, \tag{2.29}$$

$$\gamma_k = \langle\Phi_k^2\rangle = \int_S \Phi_k^2(\xi)\rho(\xi)d\xi. \tag{2.30}$$

Thus, the diffusion term simplifies to

$$\nu\gamma_k\frac{\partial^2\hat{u}_k}{\partial x^2}. \tag{2.31}$$

Deterministic System of Coupled Equations

Combining all terms, the deterministic system of equations for $u_k(x,t)$ is defined as,

$$\gamma_k\frac{\partial\hat{u}_k}{\partial t} + \sum_{i=0}^{P}\sum_{j=0}^{P}\hat{u}_i\frac{\partial\hat{u}_j}{\partial x}e_{ijk} = \nu\gamma_k\frac{\partial^2\hat{u}_k}{\partial x^2}. \tag{2.32}$$

Following division by γ_k,

$$\frac{\partial \hat{u}_k}{\partial t} + \sum_{i=0}^{P}\sum_{j=0}^{P} \hat{u}_i \frac{\partial \hat{u}_j}{\partial x}\frac{e_{ijk}}{\gamma_k} = \nu \frac{\partial^2 \hat{u}_k}{\partial x^2}, \tag{2.33}$$

$$\frac{\partial \hat{u}_k}{\partial t} + \sum_{i=0}^{P}\sum_{j=0}^{P} \hat{u}_i \frac{\partial \hat{u}_j}{\partial x} M_{ijk} = \nu \frac{\partial^2 \hat{u}_k}{\partial x^2}, \tag{2.34}$$

where M_{ijk} is the Galerkin Tensor. Subsequent rearrangement, the final system is expressed in conservative form, analogous to Eq. (2.8),

$$\frac{\partial \hat{u}_k}{\partial t} + \frac{1}{2}\sum_{i=0}^{P}\sum_{j=0}^{P} \frac{\partial(\hat{u}_i \hat{u}_j)}{\partial x} M_{ijk} = \nu \frac{\partial^2 \hat{u}_k}{\partial x^2}, \quad \forall k \in [0, P], \tag{2.35}$$

Galerkin Tensor (Pre-Processing Phase)

$$M_{ijk} = \frac{e_{ijk}}{\gamma_k} = \frac{\mathbb{E}\left[\Phi_i \Phi_j \Phi_k\right]}{\mathbb{E}\left[\Phi_k^2\right]} = \frac{\langle \Phi_i \Phi_j \Phi_k \rangle}{\langle \Phi_k^2 \rangle} = \frac{\int_S \Phi_i(\xi)\Phi_j(\xi)\Phi_k(\xi)\rho(\xi)\, d\xi}{\int_S \Phi_k^2(\xi)\rho(\xi) d\xi} \tag{2.36}$$

The term M_{ijk} in Eq. (2.36) represents the multiplication or *Galerkin* tensor. Throughout the derivation of the intrusive generalized Polynomial Chaos (gPC) and Galerkin projection formulations, different notations have been introduced for various components of this tensor. These multiple representations aim to bridge the gap between different literature sources. By recognising these notations and their equivalence, it is expected that the readers will be better equipped to navigate the diverse literature on UQ, gPC and the Galerkin projection method when applying these formulations in theoretical and computational settings. The derivation procedure outlined in this report encountered significant challenges in identifying and reconciling diverse notations. In this report, the tensor M_{ijk} simplifies to the triple product coefficients e_{ijk} due to the normalisation factor $\gamma_k = 1$.

Remarks

As described in Section *"Galerkin Multiplication"* (page 253 in Ref. [3]) some relevant aspects of the multiplication tensor are summarised below.

Symmetry: The Galerkin tensor M_{ijk} is symmetric in its first two indices (i, j), i.e., $M_{ijk} = M_{jik}$. However, no general symmetry exists involving the third index (k).

Sparsity: Since $\{\Phi_i\}_{i=0}^{P}$ form an orthogonal basis, many entries of M_{ijk} are zero, making it a sparse tensor. This sparsity can be exploited for computational efficiency.

Pre-computation: The computation of M_{ijk} depends entirely on the polynomial basis $\{\Phi_i\}$ and the probability density function $\rho(\xi)$. Once computed, the tensor can be stored and reused for various applications, significantly reducing computational cost in repeated evaluations.

Numerical Evaluation: While M_{ijk} can be computed analytically in certain cases, numerical methods such as Gaussian quadrature are often used for practical applications.

The deterministic 1D viscous Burgers' equation (Eq. (2.2)) has been transformed into a system of coupled deterministic equations (Eq. (2.35)), which is an expansion representation of the original single PDE into a RDE, through application of generalised polynomial chaos expansion (gPCE) and Galerkin projection. This system in Eq. (2.35), comprising a set of $P + 1$ coupled equations analogous to the Burgers' equation, propagates uncertainty from the boundary conditions through the nonlinear coupling term. A semi-implicit approach will be employed to solve the system, using FVM in OpenFOAM, treating nonlinear coupling terms explicitly (for unsolved coefficients at the current iteration step) and diffusion terms implicitly.

However, conventional OpenFOAM solvers are unsuitable for such coupled systems, necessitating the development or adaptation of solvers to handle uncertainty propagation. This exemplifies an intrusive UQ method, where the solution process differs from that of deterministic equations. Further details on Galerkin methods for uncertainty quantification are provided in Chapter 4 of *"Spectral Methods for Uncertainty Quantification With Applications to Computational Fluid Dynamics"* [13].

2.3.5 Boundary Conditions Expansion (Pre-Processing Phase)

This phase involves the implementation of probabilistic models to account for the uncertainty associated with input parameters, namely the boundary conditions, using established probability distributions for the uncertain data and random variables (parametrisation). The gPC framework from Eq. (2.12) is also utilised to expand the boundary conditions. The polynomial chaos expansion of the boundary conditions yields the values of the expansion coefficients (or modes) $\hat{u}_k$, such as $\hat{u}_0(-1)$ and $\hat{u}_1(-1)$, which serve as inputs at the boundary. These coefficients represent the deterministic values needed by the solver to impose the boundary condition in each mode of the polynomial chaos expansion. A Step-by-Step derivation of the boundary conditions is described below.

Left Boundary Condition

For this report, similarly to Xiu and Karniadakis' work in Ref. [17], the small imposed perturbation (δ) is assumed to be a $\delta \in (0, \epsilon) \sim \mathcal{U}(0, \epsilon)$ uniform distribution between 0 and ϵ. Therefore, *Legendre-chaos* polynomials are used, which are a special case of the *Jacobi-chaos* polynomials for uniform random variables.

Define the Boundary Condition with Random Perturbation

The left-hand boundary condition ($\tilde{u}(-1)$) is initially represented by a nominal value (e.g., 1) to which a perturbation (δ) is added.

$$\tilde{u}(-1) = 1 + \delta, \tag{2.37}$$

Expand the Boundary Condition Using Polynomial Chaos Expansion

The left boundary condition $\tilde{u}(-1)$ is expanded in the same form of Eq. (2.12)

$$\tilde{u}(-1) = \sum_{k=0}^{P} \hat{u}_k(-1)\Phi_k(\xi), \quad \forall k \in [0, P], \tag{2.38}$$

where

> $\Phi_k(\xi)$ are the orthogonal polynomials in terms of the random variable ξ, chosen based on its probability distribution. Since δ is uniformly distributed, ξ is a standardized random variable uniformly distributed in the interval $[-1, 1]$, and the appropriate orthogonal polynomials are *Legendre polynomials*.
>
> $\hat{u}_k(-1)$ are the deterministic coefficients (also called modes) of the expansion at the left boundary.

Legendre Polynomial Expressions

For a uniformly distributed random variable ξ in the interval $[-1, 1]$, the first three Legendre polynomials are

$$\Phi_0(\xi) = 1, \quad \Phi_1(\xi) = \xi, \quad \Phi_2(\xi) = \frac{1}{2}(3\xi^2 - 1), \ldots \tag{2.39}$$

A more comprehensive list of Legendre polynomials can be found in Table 9.6 of Ref. [12].

Apply the First-Order Polynomial Chaos Expansion

In a first-order expansion ($P = 1$), the approximation is truncated to the first two terms, as only the mean and standard deviation are required to define the coefficients at the left boundary condition.

$$\tilde{u}(-1) = \hat{u}_0(-1)\Phi_0(\xi) + \hat{u}_1(-1)\Phi_1(\xi). \tag{2.40}$$

By substituting the first two Legendre polynomials $\Phi_0(\xi) = 1$, and $\Phi_1(\xi) = \xi$ from Eq. (2.39) into Eq. (2.40) we obtain

$$\tilde{u}(-1) = \hat{u}_0(-1) + \hat{u}_1(-1)\xi. \tag{2.41}$$

Compute Expansion Deterministic Coefficients/Modes

The expansion coefficients $\hat{u}_0(-1)$ and $\hat{u}_1(-1)$ are obtained by expressing the perturbation δ in terms of its mean (μ_δ) and standard deviation (σ_δ). Given a uniformly distributed perturbation δ with support $(0, \epsilon)$, the values of μ_δ and σ_δ are derived directly from their definitions [4].

$$\mu_\delta = \frac{a+b}{2} = \frac{0+\epsilon}{2} = \frac{\epsilon}{2}, \tag{2.42}$$

$$\sigma_\delta = \frac{b-a}{2\sqrt{3}} = \frac{\epsilon-0}{2\sqrt{3}} = \frac{\epsilon}{2\sqrt{3}}. \tag{2.43}$$

The expansion coefficients can be expressed as follows

$$\hat{u}_0(-1) = 1 + \mu_\delta = 1 + \frac{\epsilon}{2}, \tag{2.44}$$

$$\hat{u}_1(-1) = \sigma_\delta = \frac{\epsilon}{2\sqrt{3}}. \tag{2.45}$$

Final Expression of the Deterministic Coefficients/Modes

Since this is a first-order expansion, the coefficients for higher-order terms ($k \geq 2$) are defined as zero. All coefficient terms to be used as left boundary condition inputs in Chapter 5 for the stochastic study are summarised as follows.

$$\hat{u}_k(-1) = \begin{cases} 1 + \mu_\delta = 1 + \frac{\epsilon}{2}, & \text{if } k = 0, \\ \sigma_\delta = \frac{\epsilon}{2\sqrt{3}}, & \text{if } k = 1, \\ 0, & \text{if } k \geq 2. \end{cases} \tag{2.46}$$

Right Boundary Condition

The boundary condition at $x = 1$ simplifies to $\tilde{u}(1) = -1$ since no random perturbation is imposed. Consequently, the zeroth-order term is $\hat{u}_0(1) = -1$, and all higher-order terms vanish, i.e., $\hat{u}_k(1) = 0$ for $k \geq 1$. The polynomial expansion for the right boundary condition can thus be expressed as

$$\hat{u}_k(1) = \begin{cases} -1, & \text{if } k = 0, \\ 0, & \text{if } k \geq 1. \end{cases} \tag{2.47}$$

[4] By definition from page 333-334 (A.11 Uniform Distribution A.11.3 Properties) in Ref. [12].

2.3.6 Statistical Moments of the Solution (Post-Processing Phase)

The intrusive gPC method facilitates the propagation of uncertainties and yields statistical moments of the solution, including the mean and variance. Given the calculated coefficients or modes $\hat{u}_k$ by using an implemented solver in OpenFOAM (or other software) and independently of the selected basis functions $\Phi_k(\xi)$ type (see Table 2.1), the statistical moments of the solution can be estimated. The mean (μ) and variance (σ^2) of the solution $\tilde{u}(x,t)$ are determined by the following formulas:

Expected or Mean Value (μ)

The mean of the solution is determined by the first coefficient $\hat{u}_0$, due to the orthonormality condition $\langle \Phi_0, \Phi_k \rangle = \delta_{0k}$, the mean value can be obtained by

$$\mathbb{E}[\tilde{u}(x,t,\omega)] = \langle \tilde{u}(x,t,\omega) \rangle = \sum_{k=0}^{P} \hat{u}_k(x,t) \langle \Phi_0, \Phi_k \rangle = \hat{u}_0(x,t). \tag{2.48}$$

Variance (σ^2)

The variance of the solution is obtained as the weighted sum of the squared coefficients for $k \geq 1$, leveraging the orthonormality of the basis functions,

$$\mathbb{V}[\tilde{u}(x,t,\omega)] = \sigma^2 = \mathbb{E}\left[(\tilde{u}(x,t,\omega) - \mathbb{E}[\tilde{u}(x,t,\omega)])^2\right] = \sum_{k=1}^{P} \hat{u}_k^2(x,t) \langle \Phi_k^2 \rangle = \sum_{k=1}^{P} \hat{u}_k^2(x,t), \tag{2.49}$$

where $\langle \Phi_k^2 \rangle = 1$ due to normalisation. These moments summarise the key statistical properties of the uncertain solution and rely on the orthonormality property of the basis functions. Further details on moments formulation are provided in Ref. [13, 30, 3], on page 39, 14 and 241 respectively.

2.4 Some Advantages and Limitations of gPC

2.4.1 Advantages

Mean and Variance Representation: In generalised Polynomial Chaos (gPC) expansions, the coefficient corresponding to the zeroth-order term represents the mean (expected value) of the QoI, while the variance is obtained as the sum of the squared higher-order coefficients. This property of gPC expansions is particularly significant for Uncertainty Quantification (UQ), as it allows the standard deviation (σ) of the QoI to be directly quantified. While gPC provides access to these quantities, the method is not inherently statistical, unlike the Monte Carlo method. Therefore, any additional interpretation involving confidence intervals would require further assumptions and analyses beyond the scope of this report.

Efficient Computation: Following offline pre-computation of the Galerkin tensor (M_{ijk}) (see Section 4.3 and Appendix C.1.2), the resulting tensor may be stored for subsequent reuse across diverse applications, thereby substantially mitigating computational expense in repetitive evaluations.

2.4.2 Limitations

Curse of Dimensionality: The exponential growth in computational cost as the number of random variables increases is a key drawback of gPC. This "curse of dimensionality" limits its applicability to high-dimensional problems unless mitigation strategies like sparse quadrature or low-rank approximations are employed.

High-Order Truncation Error: Significant limitations inherent in High-Order (i.e., over two random variables) Galerkin multiplication (i.e., calculation of Galerkin Tensor Coefficients) arise from its non-associative property, a direct result of the compounded truncation errors. Further details on Chapter *"Stochastic Galerkin Methods"* page 255 in Ref. [3].

Polynomial-Order Sensitivity Analysis: Unlike deterministic simulations, where mesh, time step, and domain sensitivity analyses are standard practices, gPC introduces an additional layer of sensitivity analysis related to the polynomial order. This analysis involves systematically increasing the polynomial order while monitoring the convergence of the solution and key statistical quantities of interest (e.g., mean and variance).

2.4.3 Recommendations for the Reader

While this chapter aims to provide a solid background, it cannot comprehensively address the vast and interdisciplinary fields of probability, engineering, and computational science. Additionally, the methodology presented is specifically tailored to the application under consideration. Readers are therefore encouraged to supplement this material with the cited references to gain a deeper understanding of the relevant theoretical frameworks, including their advantages and limitations for particular applications. The described method, while suitable for the present application, does not address more complex cases, such as High-Order(multivariate) (e.g., more than two random variables and dealing with variables' correlation) Uncertainty Quantification (UQ), which may require expanded methodologies and special limitations considerations.

Chapter 3

Existing Solvers: Structure and Implementation Details

This chapter presents a description of how deterministic and stochastic solvers are implemented in OpenFOAM. The deterministic solver serves as a baseline for solving the viscous Burgers' equation, while the stochastic solver incorporates the generalised Polynomial Chaos (gPC) method for uncertainty quantification. The solvers `burgersFoam` and `gPCBurgersFoam` have been made available by Robert Manson-Sawko through a public repository on GitHub in Ref. [1]. Additionally, key aspects of the solvers' structure, numerical schemes, and configuration files will be described to familiarise the reader with their implementation. Within this chapter, solver modifications are categorised and presented as either critical or optional modifications. Critical modifications are those necessary for the solver's proper operation, while optional modifications aim to enhance performance, readability (e.g.,user experience), and flexibility. Lastly, the listed modifications will be implemented in Chapter 4.

3.1 Solvers Structure Overview

3.1.1 Directory Structure of Repository Files

Downloaded from GitHub in Ref.[1][1], the solver package is structured according to the standard OpenFOAM directory hierarchy, ensuring user-friendly accessibility. The main directories are listed below:

run directory: Contains example case files and configuration setups necessary for executing the solvers. These cases provide a starting point for running both deterministic and stochastic simulations.

applications directory: This directory includes the implementation of the two solvers:

- `burgersFoam`: The deterministic solver for the standard Burgers' equation.
- `gPCBurgersFoam`: The stochastic solver implementing the generalised Polynomial Chaos (gPC) method for uncertainty quantification.

Each solver is located in its respective subdirectory and follows the OpenFOAM convention, including:

- `Make directory`: Contains the compilation instructions, including the `files` and `options` files required for compile the solvers.

[1] git clone https://github.com/robertsawko/pdesoft2016-burgers-uq.git

This structure ensures a modular and organised layout, allowing users to easily navigate between solvers, modify case files, and compile the solvers as needed.

Remark

At the time of writing this report, the implementation of the deterministic and stochastic solvers, as well as their associated cases, exhibited certain deficiencies. Specifically, the implemented equations and Galerkin coefficients in the `gPCBurgersFoam` solver required corrections due to improper initialisation. These modifications will be discussed in detail in the subsequent chapter. Nevertheless, despite these issues, the solvers, in particular the stochastic solver `gPCBurgersFoam` serve as an excellent foundation and a valuable source of inspiration for understanding the implementation process of intrusive polynomial chaos methods in OpenFOAM. These solvers provide a solid starting point for further development and refinement.

3.1.2 Base Solver Structure

The solvers `burgersFoam` and `gPCBurgersFoam` are implemented following the general structure of the basic `scalarTransportFoam` solver in OpenFOAM. While the equations implemented in each solver differ, the overall structure remains consistent and adheres to the OpenFOAM conventions. The key components of the solver structure are as follows:

Header Files

In the solvers `.C` files (`burgersFoam.C` and `gPCBurgersFoam.C`) the standard OpenFOAM classes are included, such as `fvCFD.H` for the finite volume framework (includes the class declarations) and `simpleControl.H` for preparing to read the SIMPLE sub-dictionary.

Setup and Initialisation

- `setRootCaseLists.H` and `createTime.H` Set the correct path and initialise the simulation time settings.
- `createMesh.H` create/loads the computational mesh into memory.
- `createFields.H` defines and initialises the necessary fields, such as the solution variable (e.g., T for `scalarTransportFoam` or $U/Uhat$ for our solvers) and associated parameters.

Main Solution Loop

The solvers execute the following steps iteratively until convergence:

- Time Loop.
  Controlled by the `simpleControl` class to iterate over time steps.
- Non-Orthogonal Correction Loop.
  Ensures robustness for non-orthogonal meshes by performing multiple corrections.
- Matrix Assembly and Solution.
  - Constructs the finite volume matrix system for the governing equation using `fvm::` and `fvc::` operators (e.g., time derivative, convection, and diffusion terms).
  - Solves the assembled matrix system using OpenFOAM's linear solver libraries.
- Field Update and Write: Updates the fields, applies any constraints or sources term (`fvOptions`), and writes the results at specified intervals.

Finalisation

After the solution converges or the maximum time is reached, the solver outputs a summary and terminates. The structure ensures modularity, flexibility, and ease of extension. The solvers `burgersFoam` and `gPCBurgersFoam` follow this template but implement different governing equations tailored for deterministic and stochastic simulations, respectively. A detailed explanation of the equations and their implementation will be provided in subsequent sections.

3.2 Numerical Implementation of the Deterministic Solver

The implementation in OpenFOAM uses Eq. (2.8) in a semi-implicit finite volume framework, where implicit terms are discretized and included in the matrix system using `fvm::` ('finite volume method'), while explicit terms are evaluated using `fvc::` ('finite volume calculus') based on previously computed field values. The convective term, represented by the divergence operator `fvm::div(phi, U)` in Listing 3.1 (line 64), is treated semi-implicitly. During the solve step, the flux ϕ is calculated explicitly using the velocity field from the previous time step or iteration and is treated as a known coefficient. The velocity field u^{n+1} is then solved implicitly, after which the flux ϕ is updated to ensure consistency with the newly computed velocity field for subsequent calculations. As only the steady-state solution is of interest, a first-order time discretisation scheme (the Euler method) will be utilised in the different example cases. Discretisation schemes for the remaining terms are detailed in the verification cases presented in Chapter 5. The discrete version of the viscous Burgers' equation (Eq. (2.8)) can be expressed as follows,

$$\frac{u^{n+1} - u^n}{\Delta t} + \frac{1}{2}\nabla \cdot \left(u^n u^{n+1}\right) = \nu \nabla^2 u^{n+1}, \tag{3.1}$$

where

u^{n+1}: Velocity at the current iteration step $(n+1)$.

u^n: Velocity at the previous iteration step (n).

It is important to note that a high-resolution mesh is required for accurately resolving the transition layer's position, given its sensitivity to minor boundary condition fluctuations. Enhancement of the solution is achievable via a non-uniform redistribution of elements within the computational domain.

Note (Implicit vs Explicit)

In the matrix form, $[\mathbf{A}][\mathbf{\Psi}] = [\mathbf{b}]$, discretisation is considered *implicit* when terms contribute to $[\mathbf{A}]$, treating $\mathbf{\Psi}$ as the unknown. However, in OpenFOAM's `fvm` namespace, implicit algorithms often include explicit contributions. In contrast, *explicit* discretisation calculates coefficients in $[\mathbf{b}]$ only, using the current values of the fields. For more details on discretisation schemes and matrix construction, see *"Notes on Computational Fluid Dynamics: General Principles"* [31].

3.2.1 Description of Solver File `burgersFoam.C`

The main computational process in the `burgersFoam` solver occurs within the *simple loop* (Listing 3.1), where the governing equation is implemented and solved iteratively. The main source code for the solver is described below.

Listing 3.1: `burgersFoam.C(simple.loop)`

```
55      while (simple.loop())
56      {
57          Info<< "Time = " << runTime.timeName() << nl << endl;
58
59          while (simple.correctNonOrthogonal())
60          {
61              solve
62              (
63                  fvm::ddt(U)
64                + fvm::div(phi, U)
65                - fvm::laplacian(nu, U)
66                ==
67                fvOptions(U)
68              );
69          }
70          phi = linearInterpolate(U) & mesh.Sf();
71          runTime.write();
72      }
```

Implementation in the Solver

The *simple loop* is structured as follows:

Time Loop (`simple.loop()`)
The solver iterates over time steps until the final simulation time is reached.

Correct Non-Orthogonal Iterations (`simple.correctNonOrthogonal()`)
Ensures accuracy in handling non-orthogonal meshes.

Equation Solving
The numerical implementation of the Burgers' equation is listed in code 3.1 in line 40–44 where the different terms and its role in the solver matrix construction can be describes as follow

`fvm::ddt(U)`: Time derivative discretised implicitly.

`fvm::div(phi, U)`: Divergence of the convective flux, where ϕ is the flux field.

`fvm::laplacian(nu, U)`: Diffusion term discretised implicitly.

`fvOptions(U)` Source terms (optional)

Flux Update
After solving for U, the flux ϕ is explicitly updated using

```
phi = linearInterpolate(U) & mesh.Sf();
```

where

`linearInterpolate(U)` interpolates the velocity field U from cell centres to the cell faces.

`mesh.Sf()` represents the face area vector, which combines the face's magnitude (area) and direction (normal vector).

The & operator performs the scalar (dot) product between the interpolated velocity vector and the face area vector.

This step ensures consistency between the updated velocity field and the flux field, which is essential for accurate flux calculations in the finite volume method.

Writing Results
Simulation data is written to the output files at each time step using `runTime.write();`.

Remark

The implemented equation in Listing 3.1 line 64 is missing a factor of $\frac{1}{2}$ for a conservative form required in OpenFOAM (see Eq. (2.8)). This will be listed as a critical modification.

3.2.2 Description of `createFields.H` File

The `createFields.H` file is responsible for initialising the key fields and input parameters required for the solver. It includes the following lines of code:

Reading the Velocity Field U

The velocity field U is read from the case files using the following lines of code (Listing 3.2):

Listing 3.2: `burgersFoam(createFields.H (U Field))`

```
1   Info<< "Reading field U\n" << endl;
2
3   volVectorField U
4   (
5       IOobject
6       (
7           "U",
8           runTime.timeName(),
9           mesh,
10          IOobject::MUST_READ,
11          IOobject::AUTO_WRITE
12      ),
13      mesh
14  );
```

The instruction `IOobject::MUST_READ` associated with the `IOobject` class ensures that the velocity field U is read from the input files, and `AUTO_WRITE` allows the field to be written to output during the simulation.

Reading the Transport Properties (Kinematic Viscosity ν) and Creating Flux Field

A dictionary named `transportProperties` is read, which contains the physical parameters required for the solver (Listing 3.3).

Listing 3.3: `burgersFoam(createFields.H (Transport Properties & Flux))`

```
17  Info<< "Reading transportProperties\n" << endl;
18
19  IOdictionary transportProperties
20  (
21      IOobject
22      (
23          "transportProperties",
24          runTime.constant(),
25          mesh,
26          IOobject::MUST_READ_IF_MODIFIED,
27          IOobject::NO_WRITE
28      )
29  );
30
31
32  Info<< "Reading viscosity nu\n" << endl;
33
34  dimensionedScalar nu
35  (
36      transportProperties.lookup("nu")
37  );
38
39  #include "createPhi.H"
```

The `MUST_READ_IF_MODIFIED` flag ensures that updates to the transport properties are recognised during the simulation.

Reading the Kinematic Viscosity ν
The kinematic viscosity ν is extracted from the `transportProperties` dictionary(Listing 3.3 line 34-37). This value is essential for defining the diffusion term in the governing equation.

Creating the Flux Field(`#include "createPhi.H"`)
The flux field ϕ is initialised using the `createPhi.H` file (Listing 3.3 line 39), which computes the flux based on the velocity field U and the mesh surface area vector `mesh.Sf()`.

Compilation Warning Message

During the compilation of the solver, the following warning is generated in the `createFields.H` file:

```
   createFields.H:37:5:  warning:  'Foam::dimensioned<Type>
::dimensioned(Foam::Istream&) [with Type = double]' is deprecated:
Since 2018-11 [-Wdeprecated-declarations] 37 | );
```

This warning indicates that the use of the `dimensioned<Type>` constructor from the `Foam::Istream` interface has been deprecated since November 2018. Although the solver will still function correctly, this issue highlights the need for updating deprecated constructs to ensure compatibility with future versions of OpenFOAM. This warning will be added to the list of Critical Modifications to address and update the solver implementation.

3.2.3 Make Folder Files

The `Make` folder in the solver directory contains the `files` and `options` files.

`files`

The `files` file lists the source files to be compiled and specifies the name of the executable generated (see Listing 3.4):

Listing 3.4: `files`

```
1 burgersFoam.C
2
3 EXE = $(FOAM_APPBIN)/burgersFoam
```

The executable `burgersFoam` is placed in the default OpenFOAM application binary directory (`$(FOAM_APPBIN)`), which is generally not considered a best practice.

`options`

The `options` file includes the required include directories (`EXE_INC`) and the linked libraries (`EXE_LIBS`) necessary for compilation:

Listing 3.5: `options`

```
1  EXE_INC = \
2      -I$(LIB_SRC)/finiteVolume/lnInclude \
3      -I$(LIB_SRC)/fvOptions/lnInclude \
4      -I$(LIB_SRC)/meshTools/lnInclude \
5      -I$(LIB_SRC)/sampling/lnInclude
6
7  EXE_LIBS = \
8      -lfiniteVolume \
9      -lfvOptions \
10     -lmeshTools \
11     -lsampling
```

`EXE_INC`: Specifies the include directories for necessary OpenFOAM libraries.

`EXE_LIBS`: Links the required OpenFOAM libraries for finite volume operations, mesh tools, options handling, and sampling.

3.2.4 Modifications for `burgersFoam`: Critical and Optional

Critical Modifications

All necessary modifications will be incorporated in the subsequent chapter.

1. The implemented equation in Listing 3.1, line 64 , is analogous to the conservative form of the Burgers' equation (Eq. (2.8)), differing only by the omission of the factor $\frac{1}{2}$.
2. Using `phi = fvc::flux(U)` instead of `phi = linearInterpolate(U) & mesh.Sf()`, is generally more beneficial because it is a built-in OpenFOAM function that ensures consistent flux calculation with the selected numerical schemes. It reduces redundancy in the code, avoids potential errors, and aligns with OpenFOAM discretisation framework.
3. Address "warning message" during solver compilation.
4. Changing to `$(FOAM_USER_APPBIN)`: By modifying the `EXE` line in the `files` file, `EXE = $(FOAM_USER_APPBIN)/burgersFoam`, the executable will be placed in the **user-specific** binary directory (`$(FOAM_USER_APPBIN)`), instead of the global application directory. This approach has the following advantages:

   Customisation: It keeps user-defined solvers separate from OpenFOAM's default solvers, avoiding any potential conflicts.

   Ease of Development: Facilitates testing and debugging of custom solvers without affecting existing OpenFOAM installations.

   This modification is a good practice recommendation when developing a new solver based on an existing one.

Optional Modifications

1. By introducing `fvVectorMatrix UEqn` instead of directly solving the equation with `solve`, the equation is first encapsulated into a finite volume matrix system. This approach allows users to construct the coefficients and source terms into a linear system of equations that can be inspected, manipulated, and controlled before solving. It provides flexibility to apply operations such as relaxation, constraints, or additional corrections (e.g., `UEqn.relax()` or `fvOptions.constrain(UEqn)`) prior to calling the solver, ensuring greater control over the solution process.
2. Adding `runTime.printExecutionTime(Info)` after `runTime.write()`, it does not impact the solver's numerical results, accuracy, or stability. It is purely for monitoring, profiling, and user feedback, which can be beneficial during development or testing but is not strictly necessary for the solver to function.

3.3 Numerical Implementation of the Stochastic Solver

3.3.1 Description of Solver File `gPCBurgersFoam.C`

The main computational process for the stochastic solver is implemented within the `simple.loop()` structure illustrated in Listing 3.6, where the main components are in lines 66-83. The solver implements the gPC method described in the previous chapter to decompose the velocity field U into its mode coefficients $\hat{U}_k$, which will correspond to an equation per mode, from the coupled system of equations (see Eq. (2.35)).

Listing 3.6: `gPCBurgersFoam.C (simple.loop)`

```
60      while (simple.loop())
61      {
62          Info<< "Time = " << runTime.timeName() << nl << endl;
63
64          while (simple.correctNonOrthogonal())
65          {
66              forAll(Uhat, k)
67              {
68                  fvVectorMatrix UhatEqn(
69                      fvm::ddt(Uhat[k])
70                      -
71                      fvm::laplacian(nu, Uhat[k])
72                  );
73
74                  forAll(Uhat, i){
75                      forAll(Uhat, j){
76                          if(j == k)
77                              UhatEqn += e[i][j][k] * fvm::div(phihat[i], Uhat[j]);
78                          else
79                              UhatEqn += e[i][j][k] * fvc::div(phihat[i], Uhat[j]);
80                      }
81                  }
82                  solve(UhatEqn);
83                  phihat[k] = linearInterpolate(Uhat[k]) & mesh.Sf();
84              }
85
86          }
87          runTime.write();
88      }
```

Remark

As with the deterministic solver, the implemented equation in Listing 3.6 (line 77 and 79) a factor $\frac{1}{2}$ is missing for a conservative form required in OpenFOAM (see Eq. (2.35)). This will be categorised as a critical modification.
The system of equations is iteratively solved as follows:

Time Loop (`simple.loop()`)
The solver iterates over time steps until the final simulation time is reached.

Correct Non-Orthogonal Iterations (`simple.correctNonOrthogonal()`)
Ensures accuracy in handling non-orthogonal meshes.

Modes Loop
Iterates over each mode $\hat{U}_k$, where the following operations are performed:

- Solve the time derivative and diffusion terms *implicitly* for the current mode:

  $$\texttt{UhatEqn} = \texttt{fvm::ddt(Uhat[k]) - fvm::laplacian(nu, Uhat[k])}.$$

- Add contributions from the nonlinear convection term, introducing coupling between the modes:

  $$\texttt{UhatEqn += e[i][j][k]} \cdot \begin{cases} \texttt{fvm::div(phihat[i], Uhat[j])} & \text{if } j = k \text{ (implicit)} \\ \texttt{fvc::div(phihat[i], Uhat[j])} & \text{if } j \neq k \text{ (explicit)} \end{cases}.$$

For example, when solving mode $\hat{U}_0$ is treated *implicitly*, while the contributions from other modes $\hat{U}_j$ (where $j \neq k$) are treated *explicitly* using their values from the previous iteration step, as these modes have not yet been updated. The term `e[i][j][k]` correspond to the pre-computed triple product coefficients (e_{ijk}) (see Eq. (2.26)) which are the same as the Galerkin Tensor M_{ijk} (see Eq. (2.36)) since $\gamma_k = 1$ due to normalisation. In summary, `gPCBurgersFoam` is solving one mode equation at the time, using the corresponding constant coefficients.

– Solve the system for $\hat{U}_k$ and update the flux $\hat{\phi}_k$:

`phihat[k] = linearInterpolate(Uhat[k]) & mesh.Sf().`

Output
Results are written at each time step using `runTime.write()`.

Key Features

The stochastic solver solves a coupled system of equations resulting from the gPC expansion.

Mode coupling arises through the nonlinear convection term, handled partially implicitly and explicitly.

Fluxes are updated after solving the mode coefficients to ensure consistency with the velocity field.

3.3.2 Description of `createFields.H` File

The `createFields.H` file for the stochastic solver `gPCBurgersFoam` is responsible for reading user-defined input properties and initializing the required fields, including the mode coefficients and fluxes. Key elements of this file are described as follows:

UQProperties Dictionary

A dictionary named `UQProperties` (Listing 3.7) is defined to read the required parameters for uncertainty quantification, in this case, the polynomial order. The polynomial order for the gPC expansion is extracted using `order = readLabel(UQProperties.lookup("order"))`.

Listing 3.7: `gPCBurgersFoam (createFields.H (UQProperties))`

```
1   IOdictionary UQProperties
2   (
3       IOobject
4       (
5           "UQProperties",
6           runTime.constant(),
7           mesh,
8           IOobject::MUST_READ,
9           IOobject::NO_WRITE
10      )
11  );
12
13  label order
14  (
15      readLabel(UQProperties.lookup("order")) //need readLabel!
16  );
```

Galerkin Tensor Coefficients (`e[3][3][3]`)

The array `e[3][3][3]` in Listing 3.8 (line 18) defines the precomputed triple product coefficients (i.e., the Galerkin tensor for the normalised case) that multiply the nonlinear convective term. These precomputed coefficients, which are hardcoded in the solver implementation, correspond to a normally distributed random variable and are associated with a second-order *Hermite* polynomial basis. However, apart from the hardcoded nature of the coefficients, the current polynomial type is

not suitable for verification purposes. Therefore, it will be necessary to implement a version based on *Legendre-chaos* polynomials.

Listing 3.8: gPCBurgersFoam (createFields.H (Tensor Coefficients))

```
18    scalar e[3][3][3];
19    e[0][0][0] = 1.0;
20    e[1][1][0] = 1.0;
21    e[2][2][0] = 1.0;
22
23    e[0][1][1] = 1.0;
24    e[1][0][1] = 1.0;
25    e[1][2][1] = Foam::sqrt(2.0);
26    e[2][1][1] = Foam::sqrt(2.0);
27
28    e[0][2][2] = 1.0;
29    e[2][0][2] = 1.0;
30    e[1][1][2] = Foam::sqrt(2.0);
31    e[2][2][2] = 2.0 * Foam::sqrt(2.0);
```

Note

While directly hard-coding the coefficients into the OpenFOAM solver or dictionary is simpler, it bypasses an important step of linking the mathematical derivation of the coefficients to their implementation. This report details the process of Galerkin Coefficient Calculation (see Section 4.3 and Appendix C.1.2), including the mathematical framework, Python-based computation, and integration into the OpenFOAM workflow. Python-based coefficient calculation provides flexibility, reproducibility, and potential for modification, overcoming limitations inherent in hard-coded values while enhancing transparency and learning process.

Mode Fields Initialization

In the solver `createFields.H` file (Listing 3.9) it is implemented a `PtrList`(pointer list) to initialise multiple velocity modes `Uhat` and corresponding fluxes `phihat`, where the number of modes depends on the polynomial `order`. Each mode field (`Uhat[i]` and `phihat[i]`) is created using a loop over the number of equation needed to solved (`order +1`). The velocity fields `Uhat` are initialised as `volVectorField`, and the flux fields `phihat` are computed explicitly using,

```
phihat[i] = linearInterpolate(Uhat[i]) & mesh.Sf().
```

Listing 3.9: gPCBurgersFoam (createFields.H (Uhat,phihat))

```
33    PtrList<volVectorField> Uhat(order + 1);
34    PtrList<surfaceScalarField> phihat(order + 1);
35    forAll(Uhat, i)
36    {
37        Info<< "Reading field U" << i <<endl;
38        Uhat.set(i, new volVectorField(
39            IOobject
40            (
41                "Uhat" + std::to_string(i),
42                runTime.timeName(),
43                mesh,
44                IOobject::MUST_READ,
45                IOobject::AUTO_WRITE
46            ),
47            mesh));
48        phihat.set(i, new surfaceScalarField(
49            IOobject
50            (
51                "phihat" + std::to_string(i),
52                runTime.timeName(),
53                mesh,
54                IOobject::READ_IF_PRESENT,
55                IOobject::AUTO_WRITE
56            ),
57            linearInterpolate(Uhat[i]) & mesh.Sf()));
58    }
```

Reading Transport Properties (ν) and Compilation Warning Message

The lines of code in the file `createFields.H (transportProperties)` of the solver `gPCBurgersFoam` are the same as `burgersFoam`. The compilation warning message will be handled in a similar manner as the deterministic solver.

3.3.3 Make Folder Files

Modifications to the files contained in the `Make` folder are similar to those applied to the deterministic solver.

3.3.4 Modifications for `gPCBurgersFoam`: Critical and Optional

All necessary modifications will be incorporated in the subsequent chapter.

Critical Modifications

1. As with the deterministic solver, the implemented equation in Listing 3.6 (line 77 and 79) a factor $\frac{1}{2}$ is missing for a conservative form required in OpenFOAM.
2. Using `phihat[k] = fvc::flux(Uhat[k])` instead of `phihat[k] =linearInterpolate (Uhat[k]) & mesh.Sf()`, is generally more advantageous, as it utilises a built-in OpenFOAM function that ensures consistent flux calculation with the selected numerical schemes. It reduces redundancy in the code, avoids potential errors, and aligns with OpenFOAM discretisation framework.
3. The solver incorporate the hardcoded polynomial type and order for the precomputed non-zero coefficients (`e[ijk]`). This design restricts the solver's flexibility, as any change in the random variable distribution or the polynomial order requires recompilation of the solver. This limitation reduces its adaptability for broader applications involving other distributions or higher-order expansions.
4. Uninitialised zero components in the coefficient tensor (`e[ijk]`). This causes the solver to crash, as it has been observed that OpenFOAM assigns arbitrary values to the tensor coefficient locations where zeros are expected.
5. Address "warning message" during solver compilation.

Optional Modifications

1. Syntax consistency: `solve(UhatEqn))` can be change for `UhatEqn.solve()`, to keep consistency with OpenFOAM.
2. Adding `runTime.printExecutionTime(Info)` after `runTime.write()`, it is mainly for monitoring, and user feedback, which can be beneficial during development or testing but is not strictly necessary for the solver to function.

Chapter 4

Solvers Modifications: Implementation Details and Supporting Tools

This chapter focuses on the critical and optional modifications required for the deterministic solver (`burgersFoam`) and the stochastic solver (`gPCBurgersFoam`), intended to improve functionality and consistency within OpenFOAM framework. The modifications address critical issues such as equation formulation, flux computation, and solver robustness, while optional improvements provide recommendations to enhance usability.

Furthermore, this chapter also includes the development of supplementary Python-based pre- and post-processing tools designed to facilitate both the implementation and verification of the modified solvers. The pre-processing tools compute the Galerkin coefficients necessary for the stochastic solver. These tools are designed to enhance flexibility by allowing users to modify the expansion order for *Legendre-chaos* polynomials and are readily adaptable to other polynomial types with minor adjustments. By pre-computing the coefficients, the solvers can be reused without rerunning the coefficient generation process, saving time and computational resources.

In addition, the modified implementation in this report, enables the coefficients to be automatically recomputed with every execution of the solver by incorporating a flag (`coeffCalc=true`) in the `Allrun` scripts that are designed to run all codes in an automated manner. This feature could provide an advantage when conducting sensitivity analyses across multiple instances involving different polynomial orders. Post-processing scripts compute the analytical solution of the steady-state viscous Burgers' equation, providing a benchmark for verification. Additionally, the corresponding moments and uncertainty quantification calculations are implemented for data analysis by extracting key results of the QoI (in this case the transition layer's location (z)).

4.1 Modifications to `burgersFoam`

4.1.1 Creating `myBurgersFoam`

After downloading the solver from the repository in Ref. [1], it is recommended to create a copy of the solver, rename the relevant folders (e.g., renaming `burgersFoam` to `myBurgersFoam`), and update all occurrences of the original name within the code. This section provides a step-by-step guide for creating a new OpenFOAM solver, `myBurgersFoam`, derived from the downloaded `burgersFoam` solver. Initiate a terminal session, source your OpenFOAM-v2306 application, and then execute the commands provided below. Execution of the provided commands results in navigation to the designated OpenFOAM user directory, followed by the creation of a `1D_ViscousBurgersEquation directory` within the `applications/solvers` hierarchy. Following this, the `burgersFoam` solver directory is replicated into a newly created directory named `myBurgersFoam`. Modifications to the

Make/files file guarantee the solver's inclusion within the user application's binary directory (no changes are needed for the Make/options file). In the final step, the main solver file is renamed from burgersFoam.C to myBurgersFoam.C.

```
cd $WM_PROJECT_USER_DIR
mkdir -p applications/solvers/1D_ViscousBurgersEquation
cd applications/solvers/1D_ViscousBurgersEquation
cp -r /path/to/downloaded/files/burgersFoam myBurgersFoam
cd myBurgersFoam
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
find . -type f -exec sed -i 's/burgersFoam/myBurgersFoam/g' {} +
mv burgersFoam.C myBurgersFoam.C
```

As a recommended coding practice, it is advised to compile the solver using wmake and run a simple case after each modification. This ensures that any errors introduced during the implementation are identified early, facilitating a smoother debugging and development process.

4.1.2 Modifying Solver File myBurgersFoam.C

The Listing 4.1 details modifications to myBurgersFoam.C, the key change being the introduction of a 0.5 factor in the nonlinear term for representing the conservative form of the Burgers' equation. For consistency with OpenFOAM's discretisation framework and to eliminate redundant code, the computation of phi now utilises fvc::flux(U). Utilising fvVectorMatrix UEqn, rather than a direct solution with solve, facilitates the incorporation of under-relaxation factors where necessary by calling UEqn.relax().

Listing 4.1: myBurgersFoam.C(simple.loop)

```
84      while (simple.loop())
85      {
86          Info<< "Time = " << runTime.timeName() << nl << endl;
87
88          while (simple.correctNonOrthogonal())
89          {
90
91              fvVectorMatrix UEqn
92              (
93                  fvm::ddt(U)
94                + 0.5 * fvm::div(phi, U)
95                - fvm::laplacian(nu, U)
96                ==
97                fvOptions(U)
98              );
99              UEqn.relax();
100             UEqn.solve();
101         }
102         phi = fvc::flux(U);
103         runTime.write();
104         runTime.printExecutionTime(Info);
105     }
```

4.1.3 Modifying createFields.H File of myBurgersFoam.C

To avoid the deprecation warning during compilation, the definition of the kinematic viscosity nu in createFields.H can be updated as follows:

Listing 4.2: myBurgersFoam(createFields.H (Read Kinematic Viscosity))

```
34      dimensionedScalar nu
35      (
36          "nu",  // Name: kinematic viscosity
37          dimensionSet(0, 2, -1, 0, 0, 0, 0),  // Units: m^2/s
38          transportProperties  // Read from dictionary
39      );
```

4.2 Modifications to gPCBurgersFoam

4.2.1 Creating myGPCBurgersFoam

The procedure for creating myGPCBurgersFoam is similar to that described for myBurgersFoam in Section 4.1.1, involving renaming folders, updating references, and modifying the Make/files file accordingly. As before, no changes are required for the options file.

```
cd $WM_PROJECT_USER_DIR
cd applications/solvers/1D_ViscousBurgersEquation
cp -r /path/to/downloaded/files/gPCBurgersFoam myGPCBurgersFoam
cd myGPCBurgersFoam
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
find . -type f -exec sed -i 's/gPCBurgersFoam/myGPCBurgersFoam/g' {} +
mv gPCBurgersFoam.C myGPCBurgersFoam.C
```

4.2.2 Modifying Solver File myGPCBurgersFoam.C

Similar to the modifications described in Section 4.1.2 for myBurgersFoam.C, the key changes in Listing 4.3 include the introduction of a 0.5 factor (Listing 4.3 line 89 and 93) in the nonlinear term. Additionally, the computation of phi using fvc::flux(U), and the incorporation of UEqn.relax() has been included for code flexibility.

Listing 4.3: myGPCBurgersFoam.C(simple.loop)

```
67      Info<< "\nCalculating gPC 1D-Burgers Equation\n" << endl;
68      while (simple.loop())
69      {
70          Info<< "Time = " << runTime.timeName() << nl << endl;
71
72          while (simple.correctNonOrthogonal())
73          {
74              forAll(Uhat, k)
75              {
76                  fvVectorMatrix UhatEqn
77                  (
78                      fvm::ddt(Uhat[k])
79                      -
80                      fvm::laplacian(nu, Uhat[k])
81                  );
82
83                  forAll(Uhat, i)
84                  {
85                      forAll(Uhat, j)
86                      {
87                          if(j == k)
88                          {
89                              UhatEqn += (*e)[i][j][k] * 0.5 * fvm::div(phihat[i], Uhat[j]);
90                          }
91                          else
92                          {
93                              UhatEqn += (*e)[i][j][k] * 0.5 * fvc::div(phihat[i], Uhat[j]);
94                          }
95                      }
96                  }
97                  UhatEqn.relax();
98                  UhatEqn.solve();
99                  phihat[k] = fvc::flux(Uhat[k]);
100             }
101
102         }
103         runTime.write();
104         runTime.printExecutionTime(Info);
105     }
```

4.2.3 Modifying `createFields.H` File of `myGPCBurgersFoam.C`

The required modifications to the `createFields.H` file in the `myGPCBurgersFoam` solver, to ensure correct field initialisation and handling for stochastic simulations, are detailed in this section. These modifications include dynamically allocating the Galerkin tensor values, reading pre-computed coefficients, and correctly reading transport properties, such as kinematic viscosity in this case. Each change ensures flexibility, adaptability to varying polynomial orders, and alignment with the requirements of the gPC framework. The `UQProperties` dictionary (Listing 3.7) and Mode Fields Initialisation (Listing 3.9) remain unmodified. These components are already configured appropriately for the solver's functionality and do not require changes.

Initialisation and Reading of the Galerkin Tensor Values

The following lines of code that are required to be implemented in `createFields.H` file, initialise and populate the Galerkin tensor with precomputed coefficients required for the stochastic solver. This tensor is a key term for coupling the system of deterministic equations (Eq. (2.35)) throughout the nonlinear convection term in the viscous Burgers' equation.

Dynamic Allocation of the Tensor `e[i][j][k]`

The entire array `e[3][3][3]` precomputed triple product (hardcoded) in Listing 3.8 should be removed and substituted for a much more flexible line of code (see Listing 4.4 and 4.5) that will not require re-compilation of the solver when changing polynomial type or expansion order by initialising the tensor and reading the non-zero coefficients from `gPCCoeff` dictionary file generated by a Python script, described in Section 4.3.

Listing 4.4: `myGPCBurgersFoam (createFields.H (Tensor Coeff. Initialisation))`

```
21 Info<< "Initialize Galerkin Tensor based on order\n" << endl;
22 autoPtr<Foam::List<Foam::List<Foam::List<scalar>>>> e(
23     new Foam::List<Foam::List<Foam::List<scalar>>>(
24         order + 1,
25         Foam::List<Foam::List<scalar>>(
26             order + 1,
27             Foam::List<scalar>(
28                 order + 1, 0.0
29             )
30         )
31     )
32 );
```

- The template class `autoPtr` (Listing 4.4 in line 22) is a smart pointer designed to manage and control memory allocation and deallocation automatically, ensuring proper cleanup of dynamically allocated objects. This ensure automatic memory management to prevent leaks. This approach is concise, OpenFOAM-compliant, and avoids the complexity of manual memory allocation.
- To ensure flexibility and scalability, the tensor structure (`e[i][j][k]`) is dynamically allocated based on the polynomial expansion order using nested `Foam::List` constructors. The tensor values are initialized to zero (Listing 4.4, line 28) to prevent arbitrary default values that may otherwise be assigned by OpenFOAM.

Reading Precomputed Galerkin Coefficients

The IOdictionary object gPCCoeff (Listing 4.5 lines 37-47) is used to read the precomputed Galerkin coefficients from a file named gPCCoeff, that is automatically created during pre-processing steps and saved to the constant directory.

Listing 4.5: myGPCBurgersFoam (createFields.H (Reading Precomputed Coeff.))

```
58 Info<< "Reading gPCCoeff\n" << endl;
59     IOdictionary gPCCoeff
60     (
61         IOobject
62         (
63             "gPCCoeff",
64             runTime.constant(),
65             mesh,
66             IOobject::MUST_READ,
67             IOobject::NO_WRITE
68         )
69     );
70 
71     for (int i = 0; i <= order; ++i)
72     {
73         for (int j = 0; j <= order; ++j)
74         {
75             for (int k = 0; k <= order; ++k)
76             {
77                 word entryName = "e[" + Foam::name(i) + "][" +
78                 Foam::name(j) + "][" + Foam::name(k) + "]";
79                 if (gPCCoeff.found(entryName))
80                 {
81                     // Assign value to the tensor
82                         (*e)[i][j][k] = readScalar(gPCCoeff.lookup(entryName));
83                     // Verification: Print the entry name and value
84                     Info << "Loaded coefficient " << entryName << ": "
85                         << (*e)[i][j][k] << endl;
86                 }
87                 // else
88                 // {
89                 //     // Debugging: Warn if the entry is missing
90                 //     // Warning << "Coefficient " << entryName
91                 //     // << " not found in gPCCoeff!" << endl;
92                 // }
93             }
94         }
95     }
```

The IOobject configuration specifies:

- IOobject::MUST_READ: Ensures the file is mandatory for the solver to read.
- IOobject::NO_WRITE: Prevents modification of the file.

Nested loops (Listing 4.5 lines 71-95) iterate over all combinations of indices i, j, k up to the polynomial order:

- For each combination, an entry name (e.g., e[0][0][0]) is constructed.
- If the entry exists in gPCCoeff, its value is read using readScalar and stored in the tensor e.

Although alternative, more efficient implementations of this nested loop structure may exist, the current approach was chosen for its practicality and clarity. This implemented method (see Listing 4.5) dynamically reads the coefficients and populates the Galerkin tensor e[i][j][k] within the specific constraints of this implementation. Future optimizations could focus on exploring more compact or performance-oriented solutions.

Transport Properties (Read Kinematic Viscosity)

To eliminate compiler warnings, the code segment in Listing 4.2 must be replicated within the myGPCBurgersFoam solver in the createFields.H file.

4.3 Pre-Processing Tools

The pre-processing tools[1] automate the generation and verification of coefficients required for Uncertainty Quantification (UQ) using generalised Polynomial Chaos (gPC) expansions. While the tool comprises four main steps (in addition to handling input/output data), these steps are:

Input Data: Read `UQProperties` dictionary (Listing C.1).

Step 1: Generate Distribution and Polynomials (Listing C.1 or stand-alone script Listing C.2).

Step 2: Orthonormality Verification (Listing C.3).

Step 3: Calculate Tensor Coefficients (Listing C.1 or stand-alone script Listing C.2).

Step 4: Galerkin Coefficient Verification (Listing C.3).

Output Data: Write output file (`gPCCoeff`) and save to the `constant` directory (Listing C.1)

This section prioritises Steps 1 and 3, given their importance to the pre-processing workflow detailed in this report. Steps 2 and 4 involve verification tasks that serve as manual "sanity checks" to ensure correctness during initial development, particularly given the author's first experience with `Chaospy`. These steps include orthonormality verification using the *Gram* matrix (a matrix of inner products using the precomputed polynomials in `Chaospy`), which is compared against the identity matrix, and the calculation of the triple product coefficients (e_{ijk}) using the *Rodrigues'* formula (for *Legendre* polynomial from Appendix A.1.3 in Ref. [18]) to create the orthogonal polynomials instead of relying only on built-in polynomial functionalities of `Chaospy` libraries. Necessary for establishing confidence in the implementation's correctness, these elements are procedural and of supplementary value. Conversely, the core functionality of the pre-processing tool, Steps 1 and 3, will be detailed in this section.

4.3.1 Step 1: Generate Distribution and Polynomials

The function `generate_distribution_and_polynomials` in Listing 4.6 creates a standardised uniform distribution over the interval $[-1, 1]$ for the random variable (ξ) and generates the corresponding *Legendre* polynomials (Φ) based on the specified polynomial order. The use of the standardised uniform distribution ensures consistency with the orthogonality properties of *Legendre* polynomials within this range. While the current implementation focuses on *Legendre* polynomials, the code is designed to be extensible, allowing for the inclusion of other polynomial types, such as *Hermite*, by extending the conditional logic. If an unsupported polynomial type is provided, the function raises a `ValueError`. The generated distribution and polynomials are returned for further use in the pre-processing workflow.

Listing 4.6: `gPCCoeff_v003.py(Function:generate_distribution_and_polynomials)`

```
16 # Function to get the distribution and polynomials based on type
17 # (always normalised)
18 def generate_distribution_and_polynomials(order, poly_type):
19     if poly_type.lower() == "legendre":
20         lower_std = -1 # standard lower bound
21         upper_std = 1  # standard upper bound
22         distribution = cp.Uniform(lower=lower_std, upper=upper_std)
23         polynomials = cp.expansion.legendre(order, lower=lower_std,
24                                             upper=upper_std,
25                                             physicist=False,
26                                             normed=True)
27     else:
28         raise ValueError("Unsupported polynomial type. Use 'legendre'.")
29     return distribution, polynomials
```

[1]Required Tools: `pyFoam` is **NOT** a part of the OpenFOAM distribution, users will have to install it separately [32], to install use: `pip install PyFoam`. `chaospy` installation link[33], to install use: `pip install chaospy`.

4.3.2 Step 3: Calculate Tensor Coefficients

The function `calculate_triple_product` (Listing 4.7) computes the triple product coefficients e_{ijk}, which are required for constructing the *Galerkin* tensor used in polynomial chaos expansions. As derived in the report (Eq. (2.36)), the orthonormality of the polynomial basis simplifies the actual *Galerkin* tensor coefficients to the triple product. These coefficients are calculated using *Gaussian* quadrature, leveraging the orthogonality properties of the polynomials over the specified probability distribution. The computed triple product coefficients form a critical input to the pre-processing phase of the solver. For more details on *Gauss-Legendre Quadrature*, see Chapter 9, titled *Stochastic Projection and Collocation*, on page 202 of Ref. [12].

Listing 4.7: `gPCCoeff_v003.py(Function:calculate_triple_product)`

```
31 def calculate_triple_product(order, distribution, polynomials):
32     """
33     Calculate triple product coefficients e_ijk using Gaussian Quadrature.
34 
35     Args:
36         order: polynomial order
37         distribution: probability density distribution
38         polynomials: polynomials expansion coefficients (e.g., Legendre).
39 
40     Returns:
41         e_ijk: Triple product coefficients using Gauss Quadrature.
42     """
43     e_ijk = {}
44 
45     num_polynomials = len(polynomials)
46     c = 1 # Classical Gauss Quadrature (c=1)
47     quadrature_order = math.ceil((3 * order + c)/2)
48     nodes, weights = cp.generate_quadrature(quadrature_order,
49                                             distribution, rule = "gaussian")
50     for i in range(num_polynomials):
51         for j in range(num_polynomials):
52             for k in range(num_polynomials):
53                 # Using Quadrature
54                 integrand = (polynomials[i](nodes)
55                              * polynomials[j](nodes)
56                              * polynomials[k](nodes))
57                 e_ijk[(i, j, k)] = np.sum(weights * integrand)
58     return e_ijk
```

4.4 Post-Processing Tools

4.4.1 Steady-State Burgers' Equation Exact Solution

This section describes the computation of the exact solution for the steady-state Burgers' equation under specified boundary conditions. The mathematical expressions for the exact solution are described in Section 2.2.1 and implemented numerically in Python for verification purposes. This solution depends on the boundary system defined by the hyperbolic tangent function and parameters such as the kinematic viscosity (ν), transition layer location (z_{ex}), and perturbation (δ). The boundary system is solved using numerical root-finding functionality is Python (see Listing 4.8 and 4.9).

Listing 4.8: `burgersEqExactSolution_v000.py` (exact solution implementation)

```
20 def burgers_exact_solution(delta, nu, tol, initial_guess):
21     def boundary_system(vars):
22         A, z_ex = vars
23         u_left = A * np.tanh((A / (2 * nu)) * (1 + z_ex)) - (1 + delta)
24         u_right = A * np.tanh((A / (2 * nu)) * (1 - z_ex)) - 1
25         return [u_left, u_right]
26 
27     solution = root(boundary_system, initial_guess, tol=tol)
28     return solution.x if solution.success else (None, None)
29 
30 def exact_solution_u(x, A, z_ex, nu):
31     return -A * np.tanh((A / (2 * nu)) * (x - z_ex))
```

Listing 4.9: `stochasticSolverVerification_v000.py` (input parameters)

```
70 #--Define parameters in this main script---------------------------------------
71 # nu = 0.05
72 exacSolTol = 1e-12
73 delta_unperturbed = case_1_lower  # unperturbed case
74 delta_perturbed = case_2_upper  # perturbed case
75 x_values = np.linspace(-1, 1, 10000) # Generate x-values for exact solution
76 #--Setup cases with delta and initial guess----------------------------------
77 casesData = {
78     cases[0]:
79         {"delta": delta_unperturbed, "initial_guess": [-1.0, 0.0],
80          "order": case_1_order, "nu": case_1_nu_value},
81     cases[1]:
82         {"delta": delta_perturbed, "initial_guess": [-1.0, 1.0],
83          "order": case_2_order, "nu": case_2_nu_value}
84 }
```

The script in Listing 4.10 calculates the exact velocity profile for given input parameters (δ, ν), by executing the Python code in Listing 4.8, determines the transition layer location (z_{ex}) and output the results to a `.txt` file (see Listing 4.10, line 175 and 178).

Listing 4.10: `stochasticSolverVerification_v000.py` (function output)

```
174         #--Calculate exact solution parameters---------------------------------
175         A, z_ex = burgers_exact_solution(delta, nu, exacSolTol, initial_guess)
176 
177         if A is not None and z_ex is not None:
178             u_values_exact = exact_solution_u(x_values, A, z_ex, nu)
179             output_file.write(f"\n{case_name} ('Exact Solution'): nu = {nu}, "
180                               f"delta = {delta}, "
181                               f"z_ex = {z_ex:.8f} (Transition Layer Location)\n")
182             print(f"\n{case_name} ('Exact Solution'): nu = {nu} , delta = {delta},"
183                   f" z_ex = {z_ex:.8f} (Transition Layer Location)")
184         else:
185             print(f"Failed to compute exact solution for {case_name}.")
186             output_file.write(f"Failed to compute"
187                               f" exact solution for {case_name}.\n")
188             continue
```

For the purpose of subsequent error analysis and solver verification, the exact solution is interpolated to correspond with the mesh points (x_{foam}) of the case simulation results (see Listing 4.11).

Listing 4.11: `stochasticSolverVerification_v000.py` (Exact solution interpolation)

```
238                 # Calculate exact_interpolated for the relative error calculation
239                 interpolation_exact = interp1d(x_values, u_values_exact,
240                                                kind='linear')
241                 U_exact_interpolated = interpolation_exact(x_foam)
```

4.4.2 Calculation of Transition Layer Location (z_{foam})

The transition layer location, z_{foam}, is determined during the post-processing of the solvers' simulation results by analysing the velocity profile $U(x)$. This location corresponds to the point where $U(x) = 0$, marking the transition between two distinct flow regions. However, since the velocity profile is represented by discrete data points, interpolation is required to refine the location of z_{foam}. This procedure is implemented in the post-processing script Listing 4.12. This method guarantees precise calculation of z_{foam} from discrete simulation data, thus offering a metric for verifying the simulated transition layer location against the exact solution.

Note

The calculation of the transition layer location is performed in the same manner for both solvers, as shown in the full code in Appendices C.7 and C.8. A key difference between the post-processing of the deterministic and stochastic solver solutions lies in the approach to input information retrieval. In the deterministic solver, $u(x)$ is accessed directly, while in the stochastic solver, it requires the special treatment of the modes to determine the statistical moments (μ and σ). Furthermore, the objective is to maintain a clear distinction between the post-processing tools used for the different solvers.

Listing 4.12: `stochasticSolverVerification_v000.py` (Transition Layer Location (z_{foam})

```
250                 #--Calculate Transition Layer Location (z_foam)----------------
251                 zero_crossings = np.where(np.diff(np.sign(U_x_foam)))[0]
252                 if zero_crossings.size > 0:
253                     idx = zero_crossings[0]
254                     z_foam = interp1d(U_x_foam[idx:idx + 2],
255                                       x_foam[idx:idx + 2], kind='linear')(0)
```

4.4.3 Velocity Mean Value, Variance and Uncertainty Calculations

Listing 4.13: `stochasticSolverUQ_v002.py` (statistical moments and UQ calculation)

```
170             if os.path.exists(sample_file):
171                 foam_data = np.loadtxt(sample_file)
172
173                 # Mean Value extraction from Uhat0-----------------------------
174                 x_foam, U_x_foam = foam_data[:, 0], foam_data[:, 1]
175                 Uhats_x = foam_data[:, 4::3] # Assuming Uhat coefficients are in columns [1, 4,
    7,...] for x-components only
176
177                 # Variance (sigma^2)-------------------------------------------
178                 # Square each mode to get the variance contribution along the spatial domain
179                 U_x_modeVar_foam = Uhats_x**2  # Shape: (number of spatial points, number of
    modes)
180
181                 # Total Variance (Sum the variances along the spatial domain)
182                 U_x_totalVar_foam = np.sum(U_x_modeVar_foam, axis=1)
183
184                 # Standard Deviation (sigma)-----------------------------------
185                 # Calculate the standard deviation based on higher-order terms
186                 U_x_sd_foam = np.sqrt(U_x_totalVar_foam)  # Sum squares of higher-order terms
    only
187
188                 # Calculate the uncentainty based on standard deviation--------
189                 k = 1 #k coverage factor ( k = 2 for 95% CI but this required normality
    assumption, this need further analysis)
190                 U_x_Unc = k * U_x_sd_foam
```

The mean (μ) and variance (σ^2) (see Listing 4.13 line 173–182) of the solution are calculated based on the statistical moments derived from the gPC expansion, as described in Section 2.3.6. The mean value corresponds to the first mode coefficient ($\hat{u}_0$), leveraging the orthonormality condition of the basis functions, while the variance is computed as the sum of the squared higher-order coefficients ($\hat{u}_k^2$) for $k \geq 1$.

The uncertainty quantification (Listing 4.13, lines 188–190) in this implementation uses a coverage factor of $k = 1$, which use a direct measure of the standard deviation (σ) as the standard uncertainty bound. Considering that gPC is not inherently a statistical method, expanding the uncertainty to a 95% confidence interval (CI) using $k = 1.96$ would require further analysis, as this requires assuming that the solution at each position in the domain follows a normal distribution. If the solution does not follow a normal distribution, additional methods would be needed to determine the appropriate confidence interval. However, this falls beyond the scope of the analysis presented in this report. For further details on standard and expanded uncertainties, see Section 2.5.7, *Standard and Expanded Uncertainties*, in Ref. [34].

Chapter 5

Verification and Uncertainty Quantification

This chapter presents a verification cases of the modified solvers by comparing the results with the study "Supersensitivity due to Uncertain Boundary Conditions" by Xiu and Karniadakis, as referenced in Ref. [17]. Their study provides a benchmark for verification through its well-defined and detailed approach to uncertainty propagation. Moreover, the study by Xiu and Karniadakis incorporates a higher-order spectral method, yielding highly accurate results. It serves as an excellent reference, particularly for assessing uncertainty quantification. The cases serve two distinct purposes. Initially, they prioritise verification of the solvers' implementation by comparing the results against the exact solution. These case designs prioritise rapid computational performance. Consequently, this verification should be considered as preliminary, since a comprehensive sensitivity analysis—considering mesh size, time step size, and polynomial order sensitivity—has not yet been conducted. The second objective is to describe the use of the stochastic solver in exploring uncertainty quantification. By providing detailed comparisons with the reference study and highlighting any potential deviation, this chapter not only supports solver verification but also acts as a practical guide for users, reinforcing the reproducibility and accuracy of the implementation.

5.1 Case Studies: Structure and Insights

The studies and cases directory (`1D_ViscousBurgersEquation`), available at [35] and illustrated in Figure 5.1, is structured to support two primary types of studies: deterministic and stochastic. These studies are designed to test the solver by simulating the *supersensitivity* phenomenon in the one-dimensional viscous Burgers' equation. Each study includes a set of two verification cases, enabling a systematic analysis of the deterministic and stochastic behaviours of the equation. This structure emphasises verification by comparing results with exact solutions and facilitates robust uncertainty quantification (UQ). To streamline execution and management, each study incorporates its own `Allrun` and `Allclean` scripts for automated execution and clean-up, respectively. Additionally, a `Results` directory is included, organised into several subdirectories to ensure efficient and systematic data storage and management.

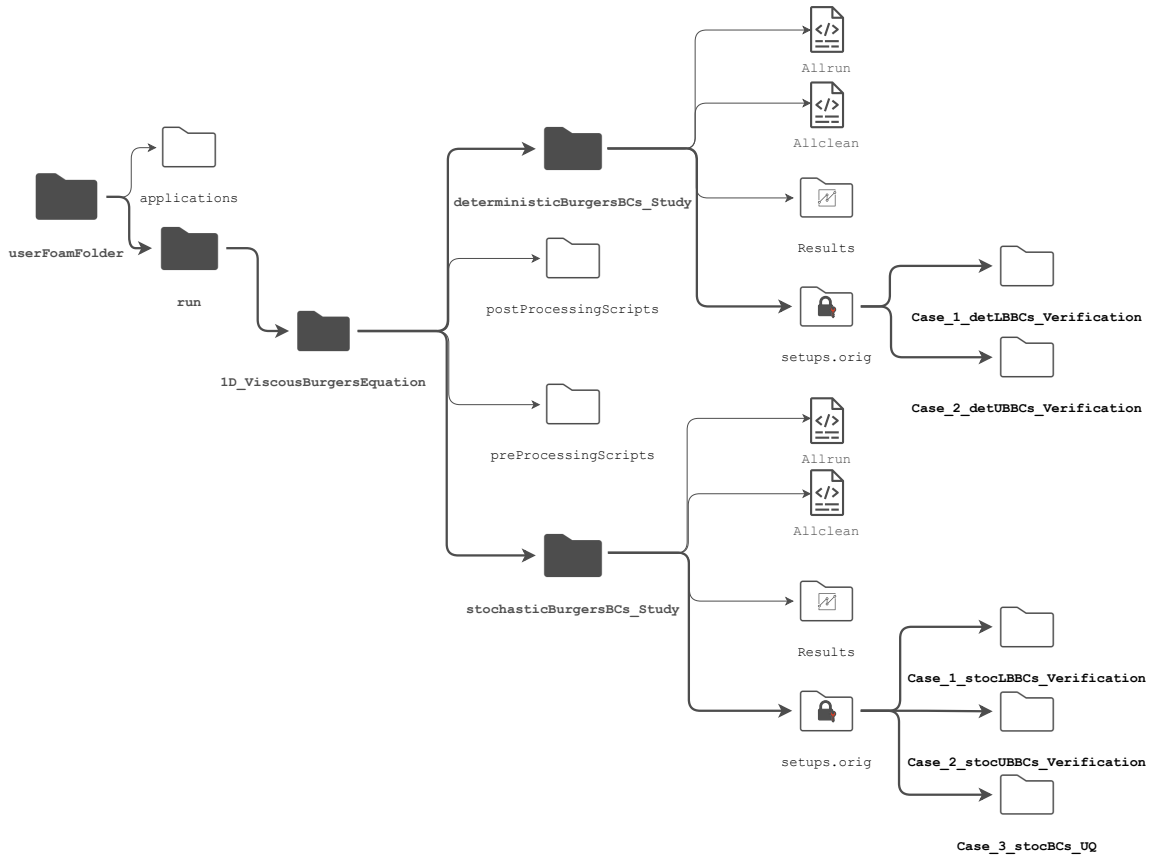

Figure 5.1: Studies and Cases Directory Structure

Warning: Directory Structure and Naming Dependency

The functionality of the `Allrun`, `Allclean`, and all pre-processing and post-processing scripts relies heavily on the directory structure and naming conventions presented in the diagram in Figure 5.1. Any modifications to this directory structure, including changes to directory names, locations, or the removal of empty directories (such as the Results directory), may cause the scripts to malfunction or crash.

5.2 Deterministic Study Using `myBurgersFoam`

The configuration, execution steps, and presentation of verification results are discussed for the two deterministic cases.The cases are designed, similarly as in Ref. [17], for verification (i.e., a comparison against the steady-state exact solution) of the implemented solver `myBurgersFoam`. The results analysis focus on the quantification of the transition layer location (z) at the lower-bound-boundary (`LBBCs`) and upper-bound-boundary (`UBBCs`) conditions, whit a perturbation $\delta \in (0, 0.1)$ respectively and a kinematic viscosity $\nu = 0.05$. Both cases utilise an identical configuration, differing only in the perturbation applied to the left boundary condition ($x = -1$).

5.2.1 Configuration and Execution of Deterministic Cases

This section presents two verification cases for the deterministic study of the 1D viscous Burgers' equation using the `myBurgersFoam` solver. The reader is guided through running the provided scripts (`Allrun`, `Allclean`) and understanding key settings. The following procedure offers a generalised

description for both cases, outlining the main relevant files and highlighting the specific differences between them.

Configuration

0.orig

U

Configuration the boundary conditions of the cases `Case_2_detUBBCs_Verification` is the same as for `Case_1_detLBBCs_Verification`, except, that the perturbation `delta $lower` in Listing B.3 0.orig (U) is replaced by `delta $upper`.

constant

UQProperties

It should be noted that, to maintain consistency in the structure of all studies, a dictionary named `UQProperties` (Listing B.5) has been created for each case located at the `constant` directory to store information about the perturbation applied to the boundaries, and this directory is included in Listing B.3 line 20 (`#include "../constant/UQProperties"`).

System

blockMeshDict

To accurately capture the transition layer position (z), which is highly sensitive to boundary condition perturbations, the mesh in `blockMeshDict` employs a non-uniform distribution (as previously noted in Section 2.2.2). The domain is divided into a coarse block ($x = -1$ to $x = -0.2$) with 5000 cells to minimise computational cost and a refined block ($x = -0.2$ to $x = 1$) with 25,000 cells to ensure precise resolution of the transition dynamics. This setup balances computational efficiency and solution accuracy, as shown in Listing B.6 (line 38-47).

controlDict

The `controlDict` (Listing B.7) file serves as the central configuration for running simulations in OpenFOAM. A sampling functionality allowing the extraction of velocity information along a defined line (`lineX`) within the computational domain has been added (see Listing B.7, lines 50–79). To facilitate detailed post-processing analysis of the velocity field, the sampling will output 10,000 uniformly distributed points along the x-axis.

decomposeParDict

The configuration of domain decomposition for parallel computing in OpenFOAM relies on the `decomposeParDict` file (Listing B.8). In this case setup, the domain is divided into 4 subdomains using the `hierarchical` method, with the specified decomposition along the x-axis (n=(4 1 1)). This configuration ensures efficient utilisation of computational resources during simulations.

fvSchemes

Numerical discretisation schemes are defined in the `fvSchemes` dictionary (Listing B.9). Time discretisation uses the Euler scheme (ddtSchemes, as previously noted in Section 2.2.2). Gradient calculations (`gradSchemes`) apply a Gauss linear scheme, while divergence terms (`divSchemes`) utilise a `Gauss linearUpwind gradient` scheme[1]. This second-order accurate approach improves accuracy

[1] For additional information on the OpenFOAM implementation, please consult [36]

compared to first-order schemes, maintaining numerical stability through gradient-based corrections. For Laplacian terms (`laplacianSchemes`), a `Gauss linear orthogonal` scheme is used, and interpolation applies linear schemes (interpolationSchemes).

`fvSolution`

The `fvSolution` dictionary (Listing B.10) specifies the solver and algorithm settings for the simulation. The velocity field (`U`) uses the `PBiCGStab` solver (Preconditioned Bi-Conjugate Gradient Stabilised for more details see Ref. [37] with a `DILU` preconditioner (see Ref. [38]) that has been shown to be the most efficient option. A tight absolute tolerance (`1e-12`) is set to prioritise high accuracy and ensure the steady-state solution is reached, with `relTol` set to zero. The `SIMPLE` algorithm applies no non-orthogonal correctors (`nNonOrthogonalCorrectors` = 0) (while not strictly necessary, for this simple case), and the relaxation factor for velocity is `1.0`, indicating no under-relaxation.

Execution

An automation script has streamlined the execution of deterministic cases, making it simpler for users. The process involves executing `Allclean` and `Allrun` in the OpenFOAM environment for version 2306, with prior installation of `pyFoam`[2] and `chaospy`[3].

`Allrun`

The `Allrun` script (Listing B.1) automates the execution of simulation cases and manages the associated pre- and post-processing tasks in OpenFoam. It prepares the working directory by copying setup files, generates and verifies the computational mesh using `blockMesh` and `checkMesh`, and supports both serial and parallel execution modes. Simulation results, including time directories, logs, and post-processing outputs, are collected into dedicated directories for each case. The script also converts results to VTK format for visualisation and executes a Python post-processing script for further analysis. After each case, temporary files are cleaned, ensuring consistency in the workflow.

`Allclean`

The `Allclean` script (Listing B.2) is designed to systematically clean up the working directory after running simulations. It removes all temporary and intermediate files, including original case setups (`0.orig`, `constant`, and `system`), VTK files, dynamic code, and simulation results for individual cases, as well as sampled data, plots, and results stored in specific directories. By ensuring the removal of unnecessary files, this script helps maintain a clean and organised directory structure, making it ready for subsequent simulations or modifications.

[2]Installation command: `pip install PyFoam`
[3]Installation command: `pip install chaospy`

5.2.2 Verification Results (`myBurgersFoam`)

Upon completion of cases execution, the post-processing tools will conduct the required calculations, saving the results and their graphical visualisations to the `Results` directory. The results verify the accuracy of the solver by demonstrating excellent agreement with the exact solution, as shown in Figure 5.2, both the velocity profile $u(x)$ (upper plot) and the relative error plots (lower plot).

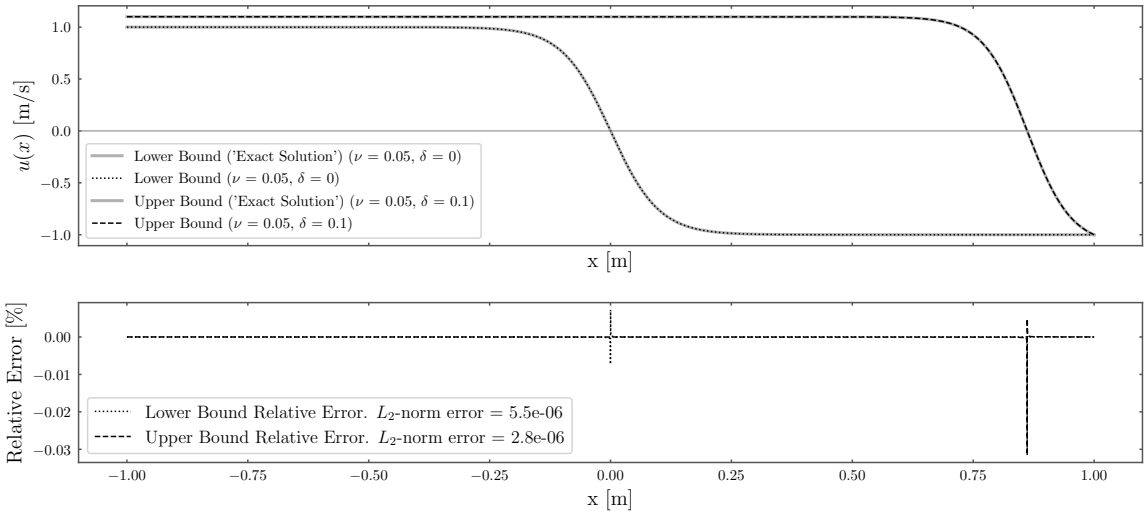

Figure 5.2: Deterministic *supersensitivity* simulation using `myBurgersFoam`

Table 5.1: Transition layer location, exact solution (z_{ex}) and `myBurgersFoam` solver (z_{det}) solution

δ	z_{ex}	z_{det}
0	0.0000000**0**	-0.0000000**1**
10^{-1}	0.8616126**2**	0.8616126**1**

The results of the transition layer location are presented in Table 5.1 that provides a numerical comparison of the computed transition layer locations (z_{det}) with the exact values (z_{ex}), highlighting minimal discrepancies. The agreement with the exact solution is accurate up to the seventh decimal digit, verifying the correct implementation of the solver subjected to two different deterministic perturbation δ on the left boundary condition ($x = -1$) with $\nu = 0.05$.

5.3 Stochastic Study Using `myGPCBurgersFoam`

This section describes the configuration, execution steps, and presents the results for a stochastic study using `myGPCBurgersFoam` solver. Its purpose is to verify the accuracy of the solver and quantify the propagated uncertainty in the solution resulting from randomly perturbed boundary conditions. The study is divided into three cases:

- `Case_1_stocLBBCs_Verification`
- `Case_2_stocUBBCs_Verification`
- `Case_3_stocBCs_UQ`

In the verification cases 1 and 2, of the `myGPCBurgersFoam` solver, no uncertainty is introduced in the polynomial chaos expansion beyond the zeroth mode ($\hat{u}_0$). This implies that all higher-order coefficients ($\hat{u}_k$ for $k \geq 1$) are set to zero, effectively reducing the solver to behave as a deterministic solver. Consequently, the `myGPCBurgersFoam` solution should align with the results of the deterministic `myBurgersFoam` solver when compared to the exact solution. This comparison ensures the correctness of the stochastic solver in deterministic conditions. In contrast, `Case_3_stocBCs_UQ` is a stochastic case that propagates imposed random perturbation throughout the system using the gPC framework that allows us to quantify the statistical moments of the solution.

5.3.1 Configuration and Execution of Stochastic Cases

The configuration and execution of the stochastic cases are largely similar to those for the deterministic cases, with minor adjustments specific to the stochastic study. These differences include modifications in the `0.orig` directory (`Uhat0` and `Uhat1`), where the initial conditions incorporate uncertainty, and a slight change in the `Allrun` script (see Appendix B.11) to account for stochastic settings (i.e., incorporating a flag (coeffCalc=true) to allow the user to select if the Galerkin coefficients should be calculated or use the pre-computed values). This section will address only the differences in configuration and execution to avoid unnecessary repetition, citing the deterministic study design where relevant.

Configuration

`0.orig`

`Uhat0`

The Listing 5.1 represents the boundary condition for the deterministic mode `Uhat0`, where the mean value of the perturbation (μ_δ) on the left boundary, with ($\delta \sim U(a, b) = \delta \sim U(0, 0.1)$) should be set as follows depending on the stochastic case:

- `Case_1_stocLBBCs_Verification`
  → `deltaMean $a;`
- `Case_2_stocUBBCs_Verification`
  → `deltaMean $b;`
- `Case_3_stocBCs_UQ`
  → `deltaMean #calc "(($a + $b)/2.0)";`

Listing 5.1: `0.orig (Uhat0)`

```
a  $lower; // a 0.0;
b  $upper; // b 0.1;
Ux_LBC $Ux_LBC;   // Specify a fixed velocity at x = -1
Ux_RBC $Ux_RBC;  // Specify a fixed velocity at x = 1
deltaMean #calc "(($a + $b)/2.0)"; // Uncomment for a real stochastic solver.

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform (#calc "($Ux_LBC + $deltaMean)" 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform ($Ux_RBC 0.0 0.0);
    }

    other
    {
        type            empty;  // Ignore y and z directions for 1D simulation
    }
}
```

`Uhat1`

The Listing 5.2, represents the boundary condition for the deterministic mode `Uhat1`, where the standard deviation value of the perturbation (σ_δ) on the left boundary, with ($\delta \sim U(a,b) = \delta \sim U(0,0.1)$) should be set as follows depending on the stochastic case:

- `Case_1_stocLBBCs_Verification`
  → `deltaMean 0;`
- `Case_2_stocUBBCs_Verification`
  → `deltaMean 0;`
- `Case_3_stocBCs_UQ`
  → `deltaSigma #calc "($b - $a) / (2.0 * sqrt(3.0))";`

Listing 5.2: `0.orig (Uhat1)`

```
#include "../constant/UQProperties"

a  $lower; // a 0.0;
b  $upper; // b 0.1;
deltaSigma #calc "($b - $a) / (2.0 * sqrt(3.0))";

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform ($deltaSigma 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }
    other
    {
        type            empty;
    }
}
```

5.3.2 Verification Results (`myGPCBurgersFoam`)

As noted earlier, during the verification process, the solver `myGPCBurgersFoam` should behave as a deterministic solver, if no uncertainty is introduced in the polynomial chaos expansion of the initial and boundary conditions. When all coefficients $\hat{u}_k = 0, \forall k \geq 1$, reducing the required system to be solved similar to the deterministic case (i.e., the system of equations is reduced to a single equation). This particular scenario allows for the verification of the solver implementation by comparing it to the exact solution.

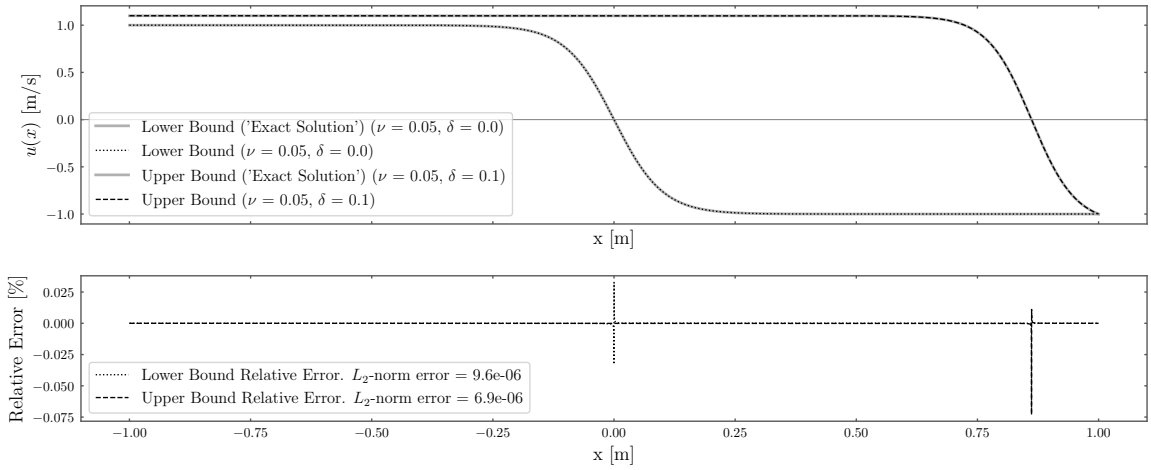

Figure 5.3: Cross-comparison of the exact solution and `myGPCBurgersFoam` results.

Figure 5.3 presents a comparison of the `myGPCBurgersFoam` solver's steady-state Burgers' equation solution with the exact solution. This comparison is only possible because the perturbation is limited to $\hat{u}_0$, making the solver function as a pseudo-deterministic solver, thus excluding higher-order term contributions. Upper and lower bounds correspond to the pseudo-deterministic solutions for the extreme values of the perturbation input, $\delta = 0.1$ and $\delta = 0$, respectively, with a kinematic viscosity of $\nu = 0.05$. The velocity profile $u(x)$ (Figure 5.3 upper plot) and the relative error [%] (Figure 5.3 lower plot) are presented to visually compare the exact solution with the two different solutions produced by the solver under the given boundary perturbation. The L_2-norm error values, for the lower and the upper bound, are relatively small, indicating a high level of agreement between the exact solution and the `myGPCBurgersFoam` results. This agreement will be further examined and corroborated at the transition layer location in the subsequent analysis.

Table 5.2: Transition layer (z) cross-comparison

δ	z_{ex}	z_{stoch}
0	0.0000000**0**	-0.0000000**3**
10^{-1}	0.8616126**2**	0.8616126**0**

TableFigure 5.2 shows the results of the transition layer cross-comparison between exact solution in Eq. (2.5) (z_{ex}) and `myGPCBurgersFoam` solver (z_{stoch}) solution with $\nu = 0.05$, subject to two different deterministic perturbations δ on left boundary condition ($x = -1$). The results shows high level of agreement between the solver solution and the exact solution, within seven significant digits for the given conditions.

5.3.3 Uncertainty Quantification Results (`myGPCBurgersFoam`)

Figure 5.4 presents the stochastic solutions of the velocity profile $u(x)$ for $\delta \sim U(0, 0.1)$ obtained using Legendre polynomial chaos with $M = 3$. The shaded region in Figure 5.4, representing the uncertainty range $(\mu \pm \sigma)$, illustrates the dispersion in the quantity of interest $(u(x))$ caused by the propagation of boundary condition uncertainties. The assumed symmetry of the uncertainty range $(\mu \pm \sigma)$ arises from the definition of the standard deviation, which is a measure of the average dispersion of data points around the mean.
The impact of boundary conditions on the dispersion of the quantity of interest $u(x)$, is effectively illustrated in Figure 5.4. A comprehensive analysis of the solution's PDF, including potential asymmetries in its distribution, would be valuable but is beyond the scope of this work. The upper and lower bounds correspond to the deterministic solutions for the extreme values of δ (0.1 and 0, respectively). Figure 5.5 illustrates the contributions of individual modes $u_k(x)$, highlighting the diminishing impact of higher-order terms beyond $M = 3$, scaled relative to the mean value $\hat{u}_0(x)$.

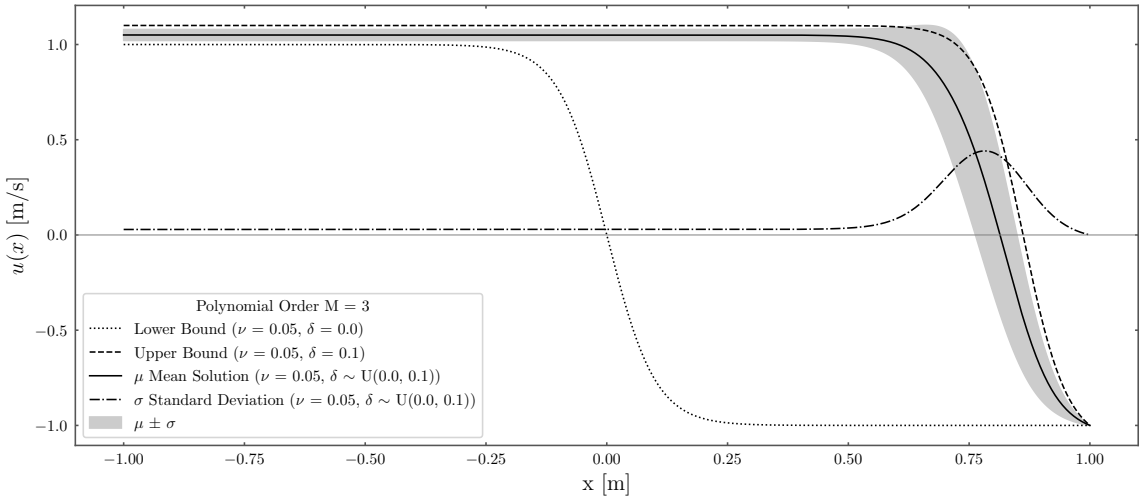

Figure 5.4: Stochastic solutions obtained using Legendre polynomial chaos for $\delta \sim U(0, 0.1)$ and $\nu = 0.05$.

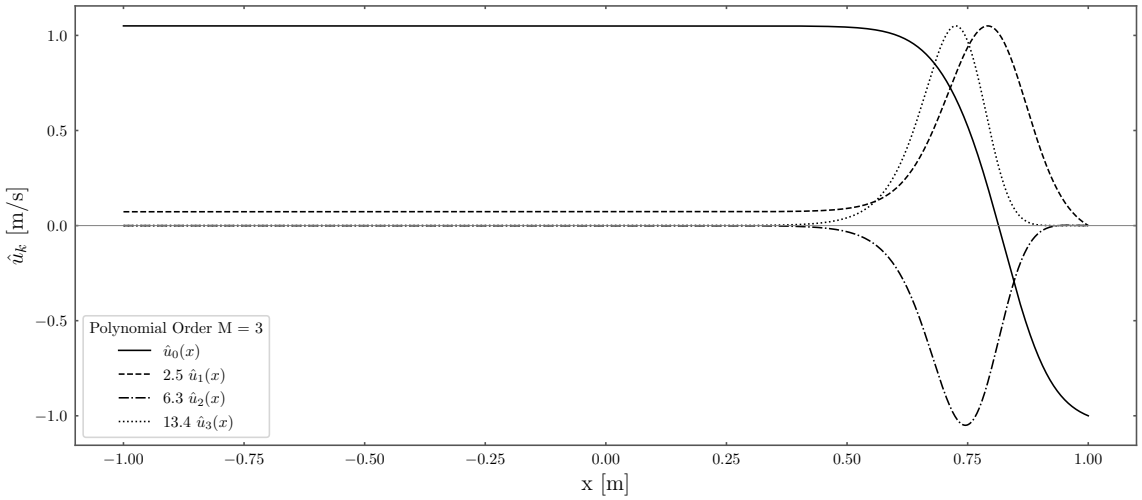

Figure 5.5: Modes Contribution (scaled to mean value $\hat{u}_0$)

Table 5.3: Results comparison of the mean transition layer location ($\overline{z}_{ref}$,$\overline{z}$) and their corresponding standard deviations (σ_{zref}, σ_z)

Polynomial Order (P)	$\overline{z}_{ref}$	$\overline{z}$	σ_{zref}	σ_z
$P = 1$	0.81459294	0.81459291	0.37660751	0.37660741
$P = 2$	0.81394090	0.81394087	0.41099350	0.41099333
$P = 3$	0.81390671	0.81390668	0.41382035	0.41382014

Table 5.3 presents the average transition layer location ($\overline{z}$) and its standard deviation (σ_z) for $\nu = 0.05$, considering a uniform random perturbation $\delta \sim U(0, 0.1)$ applied to the left boundary ($x = -1$). These results are systematically compared to the reference values ($\overline{z}_{ref}$, σ_{zref}) documented in Table B1 of Ref. [17], across different polynomial orders up to $P = 3$ and spectral element orders $N = 20$. The results highlight the ability of the `myGPCBurgersFoam` solver to accurately simulate the mean ($\bar{z}$) and standard deviation (σ_z) of the transition layer location under stochastic (i.e., random) boundary perturbations, $\delta \sim U(0, 0.1)$, for varying polynomial orders (P). The comparison between the computed mean and standard deviation ($\bar{z}$ and σ_z) with reference values (z_{ref} and $\sigma_{z_{\text{ref}}}$) demonstrates a high degree of agreement consistently within six significant digits.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Designed as a tutorial, this report not only guides readers through the methods in OpenFOAM but also provides essential pre- and post-processing tools for the efficient implementation and analysis of intrusive generalised polynomial chaos frameworks. The report details the implementation, modification, and verification of the `burgersFoam` and `gPCBurgersFoam` solvers, with a focus on uncertainty quantification within the context of the one-dimensional viscous Burgers equation. In both deterministic and stochastic studies, the solvers were systematically verified against the exact steady-state solution of the Burgers equation, demonstrating high accuracy. In summary, this report aims to support other users in improving or expanding the current implementation, contributing to integrating uncertainty quantification into computational fluid dynamics in OpenFOAM.

6.2 Future Work

To enhance the solver's functionality and broaden its range of applications, several directions for future research, building upon the present implementation, are proposed.

Generalised Polynomial Chaos Methods: Enhance the existing implementation to include other orthogonal polynomial types, e.g., Hermite polynomials, thus broadening the scope of uncertainty quantification (UQ) to various probability distributions.

Problem Complexity Expansion: Investigate the solver's performance with increased dimensionality and more sophisticated governing equations.

Application to Hydrogen Deflagration: Explore the applicability of the framework to hydrogen deflagration scenarios by introducing uncertainties in turbulence model parameters. This could provide valuable insights into safety-critical hydrogen combustion simulations.

Further investigations could build upon this report by exploring the broader implications of UQ methodologies in CFD applications to generate practical benefits.

Bibliography

[1] "pdesoft2016-burgers-uq," https://github.com/robertsawko/pdesoft2016-burgers-uq, accessed: 2024-11-18.

[2] D. Xiu, *Numerical methods for stochastic computations: A spectral method approach*, ser. Numerical Methods for Stochastic Computations: A Spectral Method Approach. Princeton University Press, 2010. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84883987013&partnerID=40&md5=03ab957548352bae56469d22b2529ae7

[3] T. Sullivan, *Introduction to Uncertainty Quantification*. Springer, 2015. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-23395-6

[4] G. E. Box and N. R. Draper, *Empirical model-building and response surfaces*. John Wiley & Sons, 1987.

[5] ASME, "Standard for verification and validation in computational fluid dynamics and heat transfer v v 20 - 2009(r2021)," 2009, aSME (American Society of Mechanical Engineers). [Online]. Available: https://www.asme.org/codes-standards/find-codes-standards/v-v-20-standard-verification-validation-computational-fluid-dynamics-heat-transfer/2009/print-book

[6] ——, "Verification, validation, and uncertainty quantification terminology in computational modeling and simulation vvuq 1 - 2022," p. 24, 2022, iSBN: 9780791875506. [Online]. Available: https://www.asme.org/codes-standards/find-codes-standards

[7] P. J. Roache, *Fundamentals of verification and validation*. hermosa publ., 2009. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/2815368242400390/FVV_Roache_2009.pdf

[8] S. Schlesinger, "Terminology for model credibility," *SIMULATION*, vol. 32, no. 3, pp. 103–104, 1979. [Online]. Available: https://doi.org/10.1177/003754977903200304

[9] SUSANA-Project, "Report on verification and validation procedures," EU Framework Program, Report SUSANA Final Report D4.2 December 2016.p, 2016. [Online]. Available: https://www.h2fc-net.eu/app/download/10705642983/SUSANA+Final+Report+D4.2+December+2016.pdf?t=1555511013

[10] R. C. Smith, *Uncertainty Quantification: Theory, Implementation, and Applications*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2013. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611973228

[11] J. Warner, "Mini tutorial 6: An introduction to uncertainty quantification for modeling & simulation," 2023. [Online]. Available: https://youtu.be/7w-K_EF2j64?t=418

[12] R. G. McClarren, *Uncertainty Quantification and Predictive Computational Science. A Foundation for Physical Scientists and Engineers*. Springer Cham, 2018. [Online]. Available: https://link-springer-com.ludwig.lub.lu.se/book/10.1007/978-3-319-99525-0

[13] O. P. L. Maître and O. M. Knio, *Spectral Methods for Uncertainty Quantification With Applications to Computational Fluid Dynamics*, ser. Scientific Computation. Springer Dordrecht, 2010. [Online]. Available: https://link.springer.com/book/10.1007/978-90-481-3520-2#bibliographic-information

[14] J. J. Keenan, D. V. Makarov, and V. V. Molkov, "Modelling and simulation of high-pressure hydrogen jets using notional nozzle theory and open source code openfoam," *International Journal of Hydrogen Energy*, vol. 42, no. 11, pp. 7447–7456, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360319916301598

[15] T. Lu and J. Gong, "Affecting mechanism of partition boards on hydrogen dispersion in confined space with symmetrical lateral openings," *International Journal of Hydrogen Energy*, vol. 46, no. 78, pp. 38 944–38 958, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360319921036582

[16] S. I. f. Standarder and S.-E. I. 13943:2017, "Fire safety – vocabulary (iso 13943:2017)," 2017. [Online]. Available: https://www.sis.se/produkter/terminologi-och-dokumentation/ordlistor/miljo-och-halsoskydd/ss-en-iso-139432017/

[17] D. Xiu and G. E. Karniadakis, "Supersensitivity due to uncertain boundary conditions," *International Journal for Numerical Methods in Engineering*, vol. 61, no. 12, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1152

[18] ——, "The wiener–askey polynomial chaos for stochastic differential equations," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 619–644, 2002, doi: 10.1137/S1064827501387826. [Online]. Available: https://doi.org/10.1137/S1064827501387826

[19] R. Askey and J. Wilson, *Some basic hypergeometric orthogonal polynomials that generalize Jacobi polynomials*. American Mathematical Society, 1985. [Online]. Available: https://www.ams.org/books/memo/0319/

[20] M. P. Bonkile, A. Awasthi, C. Lakshmi, V. Mukundan, and V. S. Aswin, "A systematic literature review of burgers' equation with recent advances," *Pramana - Journal of Physics*, vol. 90, no. 6, 2018. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85046302650&doi=10.1007%2fs12043-018-1559-4&partnerID=40&md5=c29acd331cea3278444820cc9810b848

[21] A. E. Taigbenu, *Burgers Equation*. Boston, MA: Springer US, 1999, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-1-4757-6738-4_7

[22] C. B. Laney, Ed., *Scalar Conservation Laws*. Cambridge: Cambridge University Press, 1998, pp. 48–70. [Online]. Available: https://www.cambridge.org/core/product/114E78D6BC0FF6691B3CA280A0CF04A2

[23] L. J., "Nonlinear singular perturbation problems and the engquist–osher difference scheme," 1981, technical Report 8115.

[24] "Openfoam guide/finite volume method (openfoam)." [Online]. Available: https://openfoamwiki.net/index.php/OpenFOAM_guide/Finite_volume_method_(OpenFOAM)

[25] J.-L. Peube, *Thermal Systems and Models*. Wiley, 2009, pp. 405–475. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470611500.ch8

[26] P. Pettersson, G. Iaccarino, and J. Nordström, "Numerical analysis of the burgers' equation in the presence of uncertainty," *Journal of Computational Physics*, vol. 228, no. 22, pp. 8394–8412, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999109004471

[27] R. J. LeVeque, *Numerical methods for conservation laws*, 2nd ed. Birkhäuser, 1992, vol. 214. [Online]. Available: https://link.springer.com/book/10.1007/978-3-0348-8629-1

[28] E. F. Toro, *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*, 3rd ed. Springer Science & Business Media, 2013. [Online]. Available: https://link.springer.com/book/10.1007/b79761

[29] R. Ghanem, H. Owhadi, and D. Higdon, *Handbook of uncertainty quantification*, ser. Handbook of Uncertainty Quantification. Springer International Publishing, 2017, export Date: 16 May 2023; Cited By: 58. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85020310085&doi=10.1007%2f978-3-319-12385-1&partnerID=40&md5=0fd5931878488a2bb56f1c7fcb55dfd6

[30] M. P. Pettersson, G. Iaccarino, and J. Nordstrom, *Polynomial chaos methods for hyperbolic partial differential equations*, ser. Springer Math Eng. Springer, 2015, vol. 10. [Online]. Available: https://link-springer-com.ludwig.lub.lu.se/book/10.1007/978-3-319-10714-1

[31] C. Greenshields and H. Weller, *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022. [Online]. Available: https://doc.cfd.direct/notes/cfd-general-principles/

[32] "Pyfoam 2023.7," accessed: 2024-11-18. [Online]. Available: https://pypi.org/project/PyFoam/

[33] "Chaospy — chaospy 4.3.13 documentation," accessed: 2024-11-18. [Online]. Available: https://chaospy.readthedocs.io/en/master/

[34] National Institute of Standards and Technology (NIST), *Engineering Statistics Handbook*, U.S. Department of Commerce, 2023. [Online]. Available: https://www.itl.nist.gov/div898/handbook/mpc/section5/mpc57.htm

[35] "CFD with OpenSource Software," http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/, accessed: 2020-07-21.

[36] "Linear-upwind divergence scheme,openfoam." [Online]. Available: https://www.openfoam.com/documentation/guides/latest/doc/guide-schemes-divergence-linear-upwind.html

[37] "Preconditioned bi-conjugate gradient (pbicgstab),openfoam." [Online]. Available: https://www.openfoam.com/documentation/guides/latest/doc/guide-solvers-cg-pbicgstab.html

[38] "Dilu preconditioner,openfoam." [Online]. Available: https://www.openfoam.com/documentation/guides/latest/doc/guide-solvers-cg-preconditioner-dilu.html

Study questions

1. What is the difference between Verification and Validation (V&V)?
2. Which are the key steps in Uncertainty Quantification (UQ)?
3. What is the main difference between intrusive and non-intrusive UQ propagation methods?
4. What is the main significant aspect of generalised Polynomial Chaos (gPC)?
5. Why is normalising orthogonal polynomials relevant?

Appendix A

Solvers Source Code

A.1 myBurgersFoam Solver

A.1.1 myBurgersFoam.C

Listing A.1: myBurgersFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2019-2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    myBurgersFoam

Group
    grpBasicSolvers

Description
    This implementation provides a solution to the one-dimensional viscous
    Burgers' equation, a fundamental partial differential equation in fluid
    mechanics modelling the behaviour of a viscous fluid. The equation, which
    incorporates nonlinear convection and diffusion effects, constitutes a
    fundamental model for investigating diverse physical phenomena, including
    shock waves and turbulence.

    The 1D viscous Burgers' equation is given by:

    \f[
```

```
        \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} =
        \nu \frac{\partial^2 u}{\partial x^2}
    \f]

    where ( u(x,t) ) is the velocity field, ( \nu ) is the kinematic viscosity,
    and ( x ) and ( t ) are the spatial and temporal coordinates, respectively.

SourceFiles
    burgersFoam.C

SourceLiterature
    - Book chapter 7 "Burgers Equation" "The Green Element Method" Taigbenu,
    Akpofure E. (1999)

    -UEqn from EQ.(1) "Supersensitivity due to uncertain boundary conditions"
    Xiu, Dongbin Karniadakis, George Em (2004)
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"// Include the class declarations
#include "fvOptions.H"// Include the class declarations
#include "simpleControl.H"// Prepare to read the SIMPLE sub-dictionary

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "setRootCase.H"// Set the correct path
    #include "createTime.H"// Create the time
    #include "createMesh.H"// Create the mesh

    simpleControl simple(mesh);

    #include "createFields.H"
    #include "createFvOptions.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nCalculating Deterministic 1D-Burgers Equation\n" << endl;

    #include "CourantNo.H"

    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        while (simple.correctNonOrthogonal())
        {

            fvVectorMatrix UEqn
            (
                fvm::ddt(U)
              + 0.5 * fvm::div(phi, U)
              - fvm::laplacian(nu, U)
              ==
              fvOptions(U)
            );
            UEqn.relax();
            UEqn.solve();
        }
        phi = fvc::flux(U);
        runTime.write();
        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;
    return 0;
}
```

```
// ************************************************************************* //
```

A.1.2 createFields.H

Listing A.2: myBurgersFoam.C (createFields.H)

```
    Info<< "Reading field U\n" << endl;

    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );


    Info<< "Reading transportProperties\n" << endl;

    IOdictionary transportProperties
    (
        IOobject
        (
            "transportProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE
        )
    );


    Info<< "Reading viscosity nu\n" << endl;
    // Define kinematic viscosity 'nu' (read from transportProperties)
    dimensionedScalar nu
    (
        "nu",  // Name: kinematic viscosity
        dimensionSet(0, 2, -1, 0, 0, 0, 0),  // Units: m^2/s
        transportProperties  // Read from dictionary
    );

    #include "createPhi.H"
```

A.1.3 Make Folder

Listing A.3: myBurgersFoam.C (files)

```
myBurgersFoam.C

EXE = $(FOAM_USER_APPBIN)/myBurgersFoam
```

Listing A.4: myBurgersFoam.C (options)

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude
```

```
EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling
```

A.2 myGPCBurgersFoam Solver

A.2.1 myGPCBurgersFoam.C

Listing A.5: myGPCBurgersFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2019-2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    myGPCBurgersFoam

Group
    grpStochasticSolvers (there is no a group yet as far as I know)

Description
    Solves Viscous Burgers Equation using General Polynomial Chaos Method

SourceFiles
    myGPCBurgersFoam.C

SourceLiterature

    - UhatEqn from EQ.(6.33) page 76 section 6.5 Nonlinear Problems
    Book"Numerical methods for stochastic computations: A spectral method
    approach" Xiu, D.(2010)
    -"Supersensitivity due to uncertain boundary conditions"
    Xiu, Dongbin Karniadakis, George Em (2004)

\*---------------------------------------------------------------------------*/

#include "fvCFD.H" // Include the class declarations
#include "fvOptions.H" // Include the class declarations
#include "simpleControl.H" // Prepare to read the SIMPLE sub-dictionary

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H" // Set the correct path
    #include "createTime.H" // Create the time
    #include "createMesh.H" // Create the mesh

    simpleControl simple(mesh);// Read the SIMPLE sub-dictionary

    #include "createFields.H"
    #include "createFvOptions.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nCalculating gPC 1D-Burgers Equation\n" << endl;
    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        while (simple.correctNonOrthogonal())
        {
            forAll(Uhat, k)
            {
                fvVectorMatrix UhatEqn
                (
                    fvm::ddt(Uhat[k])
                    -
                    fvm::laplacian(nu, Uhat[k])
                );

                forAll(Uhat, i)
                {
                    forAll(Uhat, j)
                    {
                        if(j == k)
                        {
                            UhatEqn += (*e)[i][j][k] * 0.5 * fvm::div(phihat[i], Uhat[j]);
                        }
                        else
                        {
                            UhatEqn += (*e)[i][j][k] * 0.5 * fvc::div(phihat[i], Uhat[j]);
                        }
                    }
                }
                UhatEqn.relax();
                UhatEqn.solve();
                phihat[k] = fvc::flux(Uhat[k]);
            }

        }
        runTime.write();
        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;
    return 0;
}

// ************************************************************************* //
```

A.2.2 createFields.H

Listing A.6: myGPCBurgersFoam.C (createFields.H)

```
//-----------------------------------------------------------------------------
Info<< "Reading UQProperties\n" << endl;
IOdictionary UQProperties
(
    IOobject
    (
        "UQProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    )
);

label order
(
    readLabel(UQProperties.lookup("order")) //need readLabel!
);

//-----------------------------------------------------------------------------
Info<< "Initialize Galerkin Tensor based on order\n" << endl;
autoPtr<Foam::List<Foam::List<Foam::List<scalar>>>> e(
    new Foam::List<Foam::List<Foam::List<scalar>>>(
        order + 1,
        Foam::List<Foam::List<scalar>>(
            order + 1,
            Foam::List<scalar>(
                order + 1, 0.0
            )
        )
    )
);

// // Verification of the initialization of the tensor-------------------------
// Info << "Verifying Tensor e Initialization...\n" << endl;

// for (int i = 0; i <= order; ++i)
// {
//     for (int j = 0; j <= order; ++j)
//     {
//         for (int k = 0; k <= order; ++k)
//         {
//             // Print the value of each element
//             Info << "e[" << i << "][" << j << "][" << k << "] = "
//             << (*e)[i][j][k] << endl;

//             // Optional: Check if the value is zero
//             if ((*e)[i][j][k] == 0.0)
//             {
//                 Info << "Coefficient e[" << i << "][" << j << "]["
//                 << k << "] is correctly initialized to 0.0." << endl;
//             }
//         }
//     }
// }

//-----------------------------------------------------------------------------
Info<< "Reading gPCCoeff\n" << endl;
    IOdictionary gPCCoeff
    (
        IOobject
        (
            "gPCCoeff",
            runTime.constant(),
```

```
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    for (int i = 0; i <= order; ++i)
    {
        for (int j = 0; j <= order; ++j)
        {
            for (int k = 0; k <= order; ++k)
            {
                word entryName = "e[" + Foam::name(i) + "][" +
                Foam::name(j) + "][" + Foam::name(k) + "]";
                if (gPCCoeff.found(entryName))
                {
                    // Assign value to the tensor
                        (*e)[i][j][k] = readScalar(gPCCoeff.lookup(entryName));
                    // Verification: Print the entry name and value
                    Info << "Loaded coefficient " << entryName << ": "
                        << (*e)[i][j][k] << endl;
                }
                // else
                // {
                //     // Debugging: Warn if the entry is missing
                //     // Warning << "Coefficient " << entryName
                //     // << " not found in gPCCoeff!" << endl;
                // }
            }
        }
    }


//---------------------------------------------------------------------------
    PtrList<volVectorField> Uhat(order + 1);
    PtrList<surfaceScalarField> phihat(order + 1);
    forAll(Uhat, i)
    {
        Info<< "Reading field Uhat" << i <<endl;
        Uhat.set(i, new volVectorField(
            IOobject
            (
                "Uhat" + std::to_string(i),
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh));
        phihat.set(i, new surfaceScalarField(
            IOobject
            (
                "phihat" + std::to_string(i),
                runTime.timeName(),
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::AUTO_WRITE
            ),
            linearInterpolate(Uhat[i]) & mesh.Sf()));
    }

//---------------------------------------------------------------------------
    Info<< "Reading transportProperties\n" << endl;

    IOdictionary transportProperties
    (
        IOobject
        (
```

```
            "transportProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE
        )
    );

//-----------------------------------------------------------------
    Info<< "Reading viscosity nu\n" << endl;
    // Define kinematic viscosity 'nu' (read from transportProperties)
    dimensionedScalar nu
    (
        "nu",  // Name: kinematic viscosity
        dimensionSet(0, 2, -1, 0, 0, 0, 0),  // Units: m^2/s
        transportProperties  // Read from dictionary
    );

//-----------------------------------------------------------------------------
```

A.2.3 Make Folder

Listing A.7: myGPCBurgersFoam.C (files)

```
myGPCBurgersFoam.C

EXE = $(FOAM_USER_APPBIN)/myGPCBurgersFoam
```

Listing A.8: myBurgersFoam.C (options)

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling
```

Appendix B

Case Studies Files

B.1 deterministicBurgersBCs_Study

B.1.1 Allrun

Listing B.1: Allrun

```
#!/bin/sh
#------------------------------------------------------------------------------
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/RunFunctions        # Tutorial run functions
. ${WM_PROJECT_DIR:?}/bin/tools/CleanFunctions      # Tutorial clean functions

#--Settings--------------------------------------------------------------------

    # Cases
    setups="
     Case_1_detLBBCs_Verification
     Case_2_detUBBCs_Verification
    "
    # flag to enable computations in parallel mode
    parallel=true

#--Functions-------------------------------------------------------------------

# Collect results into a given path
# and clean the case for the next run
# Arguments:
#    $1 = Path to move results
# Outputs:
#    Writes info to stdout
collect()
{

    [ $# -eq 0 ] && { echo "Usage: $0 dir-model"; exit 1; }

    collection="$1"

    dirResult=./"$collection"
    dirResultsSampledData=./Results/sampledData/"$collection"

    if [ ! -d "$dirResult" ]
    then

        echo "      #--Collecting results and settings into $dirResult"

        mkdir -p "$dirResult"
        mkdir -p "$dirResultsSampledData"
```

```
        endTime=$(foamListTimes -latestTime)
        cp -rf postProcessing "$dirResultsSampledData"
        mv -f $(foamListTimes) "$dirResult"
        [ -d postProcessing ] && mv -f postProcessing "$dirResult"
        [ -d processor0 ] && mv -f processor* "$dirResult"
        mv -f log.* "$dirResult"
            #mv -f profiles.dat "$dirResult"
            #cp -f system/{fv*,controlDict} constant/*Properties "$dirSettings"
        cp -rf system/ "$dirResult"
        cp -rf constant/ "$dirResult"
        mv -f 0/ "$dirResult"
        mv -f VTK/ "$dirResult"
        #mv -f dynamicCode

        runApplication paraFoam -builtin -touch -case "$dirResult"
        echo "      #--Cleaning up the case"

        cleanTimeDirectories
        cleanAuxiliary
        cleanPostProcessing

    else

        echo "      #--Directory $dirResult already exists"
        echo "      #--Skipping the computation"

    fi

}

#------------------------------------------------------------------------------

for setup in $setups
do
    #--Status Message--
    echo ""
    echo "#--Computations for Model: $setup--"
    echo ""

    dirSetup="setups.orig/$setup"
    #---Alerts--
    if [ ! -d "$dirSetup" ]
    then
        echo "Setup directory: $dirSetup" \
             "could not be found - skipping execution" 1>&2
        exit 1
    fi
    #--Copy Files--
    cp -rfL "$dirSetup/0.orig" .
    cp -rfL "$dirSetup/constant" .
    cp -rfL "$dirSetup/system" .
    cp -rf 0.orig/ 0/

    canCompile || exit 0    # Dynamic code

  if [ ! -d constant/polyMesh ]
   then
#--Geometry & Mesh-------------------------------------------------------------

        echo ""
        echo "#--Geometry & Mesh Creation"
        echo ""
        runApplication blockMesh | tee log.blockMesh
        echo ""
        echo "#--Mesh Conversion & Check"
        echo ""
        runApplication checkMesh -allTopology -allGeometry -constant | tee log.checkMesh
```

```
   fi
#--Parallel Mode-----------------------------------------------------------------

    if [ "$parallel" = true ]
    then
        echo ""
        echo "#--Parallel Mode"
        echo ""
        runApplication decomposePar
        runParallel -s parallel renumberMesh -overwrite
        runParallel  $(getApplication) | tee log.solver
        runApplication reconstructPar
        runApplication foamToVTK

    else
#--Serial Mode-------------------------------------------------------------------

        echo ""
        echo "#--Serial Mode"
        echo ""
        runApplication $(getApplication)
        #runApplication pyFoamPlotRunner.py myBurgersFoam #with residual monitor
        runApplication foamToVTK

    fi

    collect "$setup"

#--Clean Folder-------------------------------------------------------------------

rm -rf system
rm -rf 0.orig
rm -rf constant
rm -rf dynamicCode
done

#--Run Plot Files-----------------------------------------------------------------
chmod 755 ./../postProcessingScripts/deterministicSolverVerification_v003.py
python3 ./../postProcessingScripts/deterministicSolverVerification_v003.py
```

B.1.2 Allclean

Listing B.2: Allclean

```
#!/bin/sh
#------------------------------------------------------------------------------
# Source the OpenFOAM environment
#source /OpenFOAM/OpenFOAM-v2112/etc/bashrc
# Confirm the sourcing by printing OpenFOAM version
#foamVersion
#------------------------------------------------------------------------------
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/CleanFunctions      # Tutorial clean functions
# ============================================================================

cleanCase0

#------------------------------------------------------------------------------
rm -rf 0.orig
rm -rf constant
rm -rf system
rm -rf VTK
rm -rf Case_1_detLBBCs_Verification
rm -rf Case_2_detUBBCs_Verification
rm -rf dynamicCode
rm -rf ./Results/sampledData/Case_1_detLBBCs_Verification
```

```
rm -rf ./Results/sampledData/Case_2_detUBBCs_Verification
rm -rf Results/plots/*.png
rm -rf Results/plots/*.eps
rm -rf Results/plots/*.pdf
rm -rf Results/resultsData/*.txt
rm -rf Py*
rm -rf Gn*
# ========================================================================
```

B.1.3 Case_1_detLBBCs_Verification

Note

Implementation of `Case_2_detUBBCs_Verification` is the same as for `Case_1_detLBBCs_Verification`, except, that the perturbation `delta $lower` in Listing B.3 0.orig (U) should replaced by `delta $upper`. The source code, including the solver and accompanying case studies, is available to the public at [35]

0.orig

Listing B.3: `0.orig (U)`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}

dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);

#include "../constant/UQProperties"
Ux 1;// Replace with your desired velocity value
delta $lower;

boundaryField
{
    left
    {
        type            fixedValue;  // Specify a fixed velocity at x = -1
        value           uniform (#calc "($Ux + $delta)" 0 0);
    }

    right
    {
        type            fixedValue;  // Specify a fixed velocity at x = 1
        value           uniform (-1 0 0);  // Replace with your desired velocity value
    }

    other
    {
        type            empty;  // Ignore y and z directions for 1D simulation
    }
}
```

constant

Listing B.4: transportProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [0 2 -1 0 0 0 0] 0.05;// Replace with your desired viscosity value

// ************************************************************************* //
```

Listing B.5: UQProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      UQProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//--Legendre Polynomial-------------------------------------------------------
// Specify parameters
order 3;      // Adjust as needed
poly_type legendre;
lower 0;      // Adjust as needed "a" lower bound
upper 0.1;    // Adjust as needed "b" upper bound

// Output "tensorCoeff" (3rd order tensor) to file "gPCCoeff".
calculation_type  tensorCoeff;

// BCs
Ux_LBC 1;   // Specify a fixed velocity at x = -1
Ux_RBC -1;  // Specify a fixed velocity at x = 1
```

System

Listing B.6: blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
```

```
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

// Scaling factor applied to all coordinates
scale 1;

// Vertex definitions for the 1D domain along the x-axis.
// Transition refinement starts from x = -0.2 to cover the region of interest.
vertices
(
    (-1 0 0)       // Vertex 0: Start of the domain (left boundary)
    (-0.2 0 0)     // Vertex 1: Start of refined region based on the paper approach
    (1 0 0)        // Vertex 2: End of the domain (right boundary)
    (-1 1 0)       // Vertex 3: y-direction boundary for empty boundary
    (-0.2 1 0)     // Vertex 4
    (1 1 0)        // Vertex 5
    (-1 0 1)       // Vertex 6: z-direction boundary for empty boundary
    (-0.2 0 1)     // Vertex 7
    (1 0 1)        // Vertex 8
    (-1 1 1)       // Vertex 9
    (-0.2 1 1)     // Vertex 10
    (1 1 1)        // Vertex 11
);

// Block definitions:
// - Coarse mesh on the left side up to x = -0.2
// - Refined mesh from x = -0.2 to x = 1
blocks
(
    // Coarse block from x = -1 to x = -0.2
    hex (0 1 4 3 6 7 10 9) (5000 1 1) simpleGrading (1 1 1) // Adjust cell count as needed
     for coarse mesh

    // Refined block from x = -0.2 to x = 1
    hex (1 2 5 4 7 8 11 10) (25000 1 1) simpleGrading (1 1 1) // Refined block covering the
     transition region

);

// Edges section: Empty for this case, as there are no curved edges.
edges
(
);

// Boundary conditions for the mesh:
// - `left` and `right` patches define the physical boundaries at x = -1 and x = 1.
// - `other` boundary is set to empty to ignore y and z directions in a 1D simulation.
boundary
(
    left
    {
        type            patch;
        faces
        (
            (0 3 9 6)
        );
    }
    right
    {
```

```
        type            patch;
        faces
        (
            (2 5 11 8)
        );
    }

    other
    {
        type            empty;
        faces
        (
            (0 1 4 3)
            (1 2 5 4)
            (0 1 7 6)
            (1 2 8 7)
            (3 4 10 9)
            (4 5 11 10)
            (6 7 10 9)
            (7 8 11 10)
        );
    }
);

// No merging of patches specified
mergePatchPairs
(
);
```

Listing B.7: controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     myBurgersFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         1000;

deltaT          1e-4;

writeControl    timeStep;

writeInterval   1e6;

purgeWrite      0;

writeFormat     ascii;
```

```
writePrecision  10;

writeCompression off;

timeFormat      general;

timePrecision   8;

runTimeModifiable true;

// ************************************************************************* //

functions
{
//U Sampling
    Usample
    {
    libs            (fieldFunctionObjects);
        type            sets;
        libs            (sampling);
        writeControl    writeTime;
        result          Usample;
        setFormat       raw;
        interpolationScheme cellPoint;
    sets
    (
        lineX  // Sampling line along the x-axis
        {
            type        uniform;
            axis        x;
            start       (-1 0 0);  // Start point of the line
            end         (1 0 0);   // End point of the line
            nPoints     10000;       // Number of points along the line
        }
    );

    fields
    (
        U
    );
    }
}
```

Listing B.8: decomposeParDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains  4;

method          hierarchical;

coeffs
```

```
{
    n           (4 1 1);
    // delta       0.001;  //< default value = 0.001
    // order       xyz;    //< default order = xyz
}


// ************************************************************************* //

/*numberOfSubdomains 4;

method          scotch;
// ************************************************************************* //
```

Listing B.9: `fvSchemes`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default          Gauss linear;
}

divSchemes
{
    default          none;

    "div(phi,U)"     Gauss linearUpwind grad(U);

}

laplacianSchemes
{
    default          none;
        "laplacian(nu,U)" Gauss linear orthogonal;
}

interpolationSchemes
{
    default          linear;
}

snGradSchemes
{
    default          corrected;
}
```

```
// ************************************************************************* //
```

Listing B.10: fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    U
    {

        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-12;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
       U               1e-12;
    }
    relaxationFactors
    {
    equations
    {
       U               1;
    }
}
}
```

B.2 stochasticBurgersBCs_Study

B.2.1 Allrun

Listing B.11: Allrun

```
#!/bin/sh
#------------------------------------------------------------------------------
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/RunFunctions        # Tutorial run functions
. ${WM_PROJECT_DIR:?}/bin/tools/CleanFunctions      # Tutorial clean functions

#--Settings--------------------------------------------------------------------

    # Cases
    setups="
     Case_1_stocLBBCs_Verification
     Case_2_stocUBBCs_Verification
     Case_3_stocBCs_UQ
    "
    # flag to enable computations in parallel mode
    parallel=true
    coeffCalc=true

#--Functions-------------------------------------------------------------------

# Collect results into a given path
# and clean the case for the next run
# Arguments:
#    $1 = Path to move results
# Outputs:
#    Writes info to stdout
collect()
{

    [ $# -eq 0 ] && { echo "Usage: $0 dir-model"; exit 1; }

    collection="$1"

    dirResult=./"$collection"
    dirResultsSampledData=./Results/sampledData/"$collection"

    if [ ! -d "$dirResult" ]
    then

        echo "      #--Collecting results and settings into $dirResult"

        mkdir -p "$dirResult"
        mkdir -p "$dirResultsSampledData"

        endTime=$(foamListTimes -latestTime)
        cp -rf postProcessing "$dirResultsSampledData"
        mv -f $(foamListTimes) "$dirResult"
        [ -d postProcessing ] && mv -f postProcessing "$dirResult"
        [ -d processor0 ] && mv -f processor* "$dirResult"
        mv -f log.* "$dirResult"
            #mv -f profiles.dat "$dirResult"
            #cp -f system/{fv*,controlDict} constant/*Properties "$dirSettings"
        cp -rf system/ "$dirResult"
        cp -rf constant/ "$dirResult"
        mv -f 0/ "$dirResult"
        mv -f VTK/ "$dirResult"
        #mv -f dynamicCode

        runApplication paraFoam -builtin -touch -case "$dirResult"
        echo "      #--Cleaning up the case"

        cleanTimeDirectories
```

```
        cleanAuxiliary
        cleanPostProcessing

    else

        echo "      #--Directory $dirResult already exists"
        echo "      #--Skipping the computation"

    fi

}

#------------------------------------------------------------------------------

for setup in $setups
do
    #--Status Message--
    echo ""
    echo "#--Computations for Model: $setup--"
    echo ""

    dirSetup="setups.orig/$setup"
    #---Alerts--
    if [ ! -d "$dirSetup" ]
    then
        echo "Setup directory: $dirSetup" \
             "could not be found - skipping execution" 1>&2
        exit 1
    fi
    #--Copy Files--
    cp -rfL "$dirSetup/0.orig" .
    cp -rfL "$dirSetup/constant" .
    cp -rfL "$dirSetup/system" .
    cp -rf 0.orig/ 0/

    canCompile || exit 0    # Dynamic code

  if [ ! -d constant/polyMesh ]
   then
#--Geometry & Mesh-------------------------------------------------------------

        echo ""
        echo "#--Geometry & Mesh Creation"
        echo ""
        runApplication blockMesh | tee log.blockMesh
        echo ""
        echo "#--Mesh Conversion & Check"
        echo ""
        runApplication checkMesh -allTopology -allGeometry -constant | tee log.checkMesh

#--Generate gPCCoeff-----------------------------------------------------------
        if [ "$coeffCalc" = true ]
        then
            echo ""
            echo "#--Generating and Verfying gPCCoeff..."
            echo ""
            chmod 755 ./../preProcessingScripts/gPCCoeff_v003.py
            python3 ./../preProcessingScripts/gPCCoeff_v003.py
        else
            echo ""
            echo "#--Using Pre-Computed Values gPCCoeff"
            echo ""
        fi
   fi
#--Parallel Mode---------------------------------------------------------------

    if [ "$parallel" = true ]
    then
```

```
        echo ""
        echo "#--Parallel Mode"
        echo ""
        runApplication decomposePar
        runParallel -s parallel renumberMesh -overwrite
        runParallel  $(getApplication) | tee log.solver
        runApplication reconstructPar
        runApplication foamToVTK

    else
#--Serial Mode-------------------------------------------------------------------

        echo ""
        echo "#--Serial Mode"
        echo ""
        runApplication $(getApplication)
        #runApplication pyFoamPlotRunner.py myGPCBurgersFoam #with residual monitor
        runApplication foamToVTK

    fi

    collect "$setup"

#--Clean Folder------------------------------------------------------------------

rm -rf system
rm -rf 0.orig
rm -rf constant
rm -rf dynamicCode
done

#--Run Plot Files----------------------------------------------------------------
chmod 755 ./../postProcessingScripts/stochasticSolverVerification_v000.py
python3 ./../postProcessingScripts/stochasticSolverVerification_v000.py

chmod 755 ./../postProcessingScripts/stochasticSolverUQ_v002.py
python3 ./../postProcessingScripts/stochasticSolverUQ_v002.py
```

B.2.2 Allclean

Listing B.12: Allclean

```
#!/bin/sh
#------------------------------------------------------------------------------
# Source the OpenFOAM environment
#source /OpenFOAM/OpenFOAM-v2112/etc/bashrc
# Confirm the sourcing by printing OpenFOAM version
#foamVersion
#------------------------------------------------------------------------------
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/CleanFunctions      # Tutorial clean functions
# ========================================================================

cleanCase0

#------------------------------------------------------------------------------
rm -rf 0.orig
rm -rf constant
rm -rf system
rm -rf VTK
rm -rf Case_1_stocLBBCs_Verification
rm -rf Case_2_stocUBBCs_Verification
rm -rf Case_3_stocBCs_UQ
rm -rf dynamicCode
rm -rf ./Results/sampledData/Case_1_stocLBBCs_Verification
rm -rf ./Results/sampledData/Case_2_stocUBBCs_Verification
```

```
rm -rf ./Results/sampledData/Case_3_stocBCs_UQ
rm -rf Results/plots/*.png
rm -rf Results/plots/*.eps
rm -rf Results/plots/*.pdf
rm -rf Results/resultsData/*.txt
rm -rf Py*
rm -rf Gn*
# ========================================================================
```

B.2.3 Case_3_stocBCs_UQ

Note

Implementations of Case_1_stocLBBCs_Verification and Case_2_stocUBBCs_Verification is the same as for Case_3_stocBCs_UQ, except for the minor differences:

Case_1_stocLBBCs_Verification

Uhat0: deltaMean is set deltaMean $a;

Uhat1,Uhat2,Uhat3:deltaSigma is set to deltaSigma 0;

Case_2_stocUBBCs_Verification

Uhat0: deltaMean is set deltaMean $b;

Uhat1,Uhat2,Uhat3:deltaSigma is set to deltaSigma 0;

The source code, including the solver and accompanying case studies, is available to the public at [35]

0.orig

.

Listing B.13: 0.orig (Uhat0)

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      Uhat0;
}

/*---------------------------------------------------------------------------
Note:

-Uniform Distributed Random Variable (mean and standard deviation)
delta approx. U(0,epsilon) is a uniform random variable in (0,epsilon)
delta(a,b) -> delta(0,0.1)-> a = 0, b = 0.1
deltaMean = (b - a)/2
delta_sigma = (b-a)/(2.0*sqrt(3.0)

source: Book page 333-334 (A.11 Uniform Distribution A.11.3 Properties)
```

```
"Uncertainty Quantification and Predictive Computational Science.
A Foundation for Physical Scientists and Engineers" Ryan G. McClarren (2018)
-------------------------------------------------------------------------------*/
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0.0 0.0 0.0);

#include "../constant/UQProperties"

a  $lower; // a 0.0;
b  $upper; // b 0.1;
Ux_LBC $Ux_LBC;   // Specify a fixed velocity at x = -1
Ux_RBC $Ux_RBC;  // Specify a fixed velocity at x = 1
deltaMean #calc "(($a + $b)/2.0)"; // Uncomment for a real stochastic solver.

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform (#calc "($Ux_LBC + $deltaMean)" 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform ($Ux_RBC 0.0 0.0);
    }

    other
    {
        type            empty;  // Ignore y and z directions for 1D simulation
    }
}
```

Listing B.14: `0.orig (Uhat1)`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      Uhat1;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

/*---------------------------------------------------------------------------
Note:

-Uniform Distributed Random Variable (mean and standard deviation)
delta approx. U(0,epsilon) is a uniform random variable in (0,epsilon)
delta(a,b) -> delta(0,0.1)-> a = 0, b = 0.1
deltaMean = (b - a)/2
delta_sigma = (b-a)/(2.0*sqrt(3.0)

source: Book page 333-334 (A.11 Uniform Distribution A.11.3 Properties)
"Uncertainty Quantification and Predictive Computational Science.
A Foundation for Physical Scientists and Engineers" Ryan G. McClarren (2018)
-------------------------------------------------------------------------------*/
```

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0.0 0.0 0.0);

#include "../constant/UQProperties"

a  $lower; // a 0.0;
b  $upper; // b 0.1;
deltaSigma #calc "($b - $a) / (2.0 * sqrt(3.0))";

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform ($deltaSigma 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }
    other
    {
        type            empty;
    }
}


// ************************************************************************* //
```

Listing B.15: `0.orig (Uhat2)`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      Uhat2;
}

/*---------------------------------------------------------------------------
Note:

-More Higher Order Polynomial Terms should be added if the order of the
polynomials are higher than 3rd order

-Higher Order Polynomial Terms set to zero at left BC.
source: Equation (16)"Supersensitivity due to uncertain boundary conditions"
Xiu, Dongbin Karniadakis, George Em (2004)
---------------------------------------------------------------------------*/

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0.0 0.0 0.0);

boundaryField
```

```
{
    left
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }
    other
    {
        type            empty;
    }
}


/*--------------------------------------------------------------------------*/
```

Listing B.16: `0.orig (Uhat3)`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      Uhat3;
}

/*---------------------------------------------------------------------------
Note:

-More Higher Order Polynomial Terms should be added if the order of the
polynomials are higher than 3rd order

-Higher Order Polynomial Terms set to zero at left BC.
source: Equation (16)"Supersensitivity due to uncertain boundary conditions"
Xiu, Dongbin Karniadakis, George Em (2004)
---------------------------------------------------------------------------*/

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0.0 0.0 0.0);

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }

    right
    {
        type            fixedValue;
        value           uniform (0.0 0.0 0.0);
    }
```

```
    other
    {
        type            empty;
    }
}


/*---------------------------------------------------------------------------*/
```

constant

Listing B.17: transportProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [0 2 -1 0 0 0 0] 0.05;// Replace with your desired viscosity value


// ************************************************************************* //
```

Listing B.18: UQProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      UQProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//--Legendre Polynomial-------------------------------------------------------
// Specify parameters
order 3;      // Adjust as needed
poly_type legendre;
lower 0.0;      // Adjust as needed "a" lower bound
upper 0.1;    // Adjust as needed "b" upper bound

// Output "tensorCoeff" (3rd order tensor) to file "gPCCoeff".
calculation_type  tensorCoeff;

// BCs
Ux_LBC 1.0;   // Specify a fixed velocity at x = -1
```

```
Ux_RBC -1.0;  // Specify a fixed velocity at x = 1
```

System

Listing B.19: blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

// Scaling factor applied to all coordinates
scale 1;

// Vertex definitions for the 1D domain along the x-axis.
// Transition refinement starts from x = -0.2 to cover the region of interest.
vertices
(
    (-1 0 0)       // Vertex 0: Start of the domain (left boundary)
    (-0.2 0 0)     // Vertex 1: Start of refined region based on the paper approach
    (1 0 0)        // Vertex 2: End of the domain (right boundary)
    (-1 1 0)       // Vertex 3: y-direction boundary for empty boundary
    (-0.2 1 0)     // Vertex 4
    (1 1 0)        // Vertex 5
    (-1 0 1)       // Vertex 6: z-direction boundary for empty boundary
    (-0.2 0 1)     // Vertex 7
    (1 0 1)        // Vertex 8
    (-1 1 1)       // Vertex 9
    (-0.2 1 1)     // Vertex 10
    (1 1 1)        // Vertex 11
);

// Block definitions:
// - Coarse mesh on the left side up to x = -0.2
// - Refined mesh from x = -0.2 to x = 1
blocks
(
    // Coarse block from x = -1 to x = -0.2
    hex (0 1 4 3 6 7 10 9) (5000 1 1) simpleGrading (1 1 1) // Adjust cell count as needed
     for coarse mesh

    // Refined block from x = -0.2 to x = 1
    hex (1 2 5 4 7 8 11 10) (15000 1 1) simpleGrading (1 1 1) // Refined block covering the
     transition region
);

// 2500 7500  coarse mesh
// 2500 12500 medium mesh
// 5000 15000 fine mesh
// 5000 15000 superfine mesh
// (
//     // Coarse block from x = -1 to x = -0.2
//     hex (0 1 4 3 6 7 10 9) (2500 1 1) simpleGrading (1 1 1) // Adjust cell count as
     needed for coarse mesh
```

```
//     // Refined block from x = -0.2 to x = 1
//     hex (1 2 5 4 7 8 11 10) (5000 1 1) simpleGrading (1 1 1) // Refined block covering
    the transition region
// );

// Edges section: Empty for this case, as there are no curved edges.
edges
(
);

// Boundary conditions for the mesh:
// - `left` and `right` patches define the physical boundaries at x = -1 and x = 1.
// - `other` boundary is set to empty to ignore y and z directions in a 1D simulation.
boundary
(
    left
    {
        type            patch;
        faces
        (
            (0 3 9 6)
        );
    }
    right
    {
        type            patch;
        faces
        (
            (2 5 11 8)
        );
    }

    other
    {
        type            empty;
        faces
        (
            (0 1 4 3)
            (1 2 5 4)
            (0 1 7 6)
            (1 2 8 7)
            (3 4 10 9)
            (4 5 11 10)
            (6 7 10 9)
            (7 8 11 10)
        );
    }
);

// No merging of patches specified
mergePatchPairs
(
);
```

Listing B.20: controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
```

```
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     myGPCBurgersFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         1000;

deltaT          5e-3;

writeControl    timeStep;

writeInterval   1e7;

purgeWrite      0;

writeFormat     ascii;

writePrecision  10;

writeCompression off;

timeFormat      general;

timePrecision   8;

runTimeModifiable true;

// adjustTimeStep  yes;

// maxCo 0.5;

// ************************************************************************* //
functions
{
//U Sampling
    Usample
    {
        libs            (fieldFunctionObjects);
        type            sets;
        libs            (sampling);
        writeControl    writeTime;
        result          Usample;
        setFormat       raw;
        interpolationScheme cellPoint;
    sets
    (
        lineX  // Sampling line along the x-axis
        {
            type        uniform;
            // object      lineX_Uhats;
            axis        x;
            start       (-1 0 0);    // Start point of the line
            end         (1 0 0);     // End point of the line
            nPoints     10000;       // Number of points along the line
        }
    );

    fields
    (
```

```
        "Uhat0"
        "Uhat1"
        "Uhat2"
        "Uhat3"
    );
    }
}
```

Listing B.21: decomposeParDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains  4;

method          hierarchical;

coeffs
{
    n           (4 1 1);
    // delta       0.001;  //< default value = 0.001
    // order       xyz;    //< default order = xyz
}


// ************************************************************************* //

/*numberOfSubdomains 4;

method          scotch;
// ************************************************************************* //
```

Listing B.22: fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
```

```
    default          Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    "div\(phihat.*,Uhat.*\)" Gauss linearUpwind grad(Uhat);
}

laplacianSchemes
{
    "laplacian\(nu,Uhat.*\)" Gauss linear orthogonal;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}


// ************************************************************************* //
```

Listing B.23: fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2306                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    "Uhat.*"
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-12;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
```

```
    {
        "Uhat.*"              1e-12;
    }
    relaxationFactors
    {
    equations
    {
        "Uhat.*"              1;
    }
}
}


// ************************************************************************* //
```

Appendix C

Processing Tools

C.1 Pre-processing Tools

C.1.1 Polynomial Triple Product Coefficients (e_{ijk})

Listing C.1: gPCCoeff_v003.py

```
import os
import chaospy as cp
import numpy as np
import math
from datetime import datetime
from read_UQProperties_v001 import read_UQProperties
from gPCVerification_v002 import (orthonormality_verification,
galerkin_coefficient_verification)

# Locate UQProperties in the same directory as the script
script_dir = os.path.dirname(os.path.abspath(__file__))
# Function to read properties from UQProperties using pyFoam
uqproperties_path = os.path.join(script_dir, "..", "stochasticBurgersBCs_Study",
                                 "constant", "UQProperties")

# Function to get the distribution and polynomials based on type
# (always normalised)
def generate_distribution_and_polynomials(order, poly_type):
    if poly_type.lower() == "legendre":
        lower_std = -1 # standard lower bound
        upper_std = 1  # standard upper bound
        distribution = cp.Uniform(lower=lower_std, upper=upper_std)
        polynomials = cp.expansion.legendre(order, lower=lower_std,
                                                   upper=upper_std,
                                                   physicist=False,
                                                   normed=True)
    else:
        raise ValueError("Unsupported polynomial type. Use 'legendre'.")
    return distribution, polynomials

def calculate_triple_product(order, distribution, polynomials):
    """
    Calculate triple product coefficients e_ijk using Gaussian Quadrature.

    Args:
        order: polynomial order
        distribution: probability density distribution
        polynomials: polynomials expansion coefficients (e.g., Legendre).

    Returns:
        e_ijk: Triple product coefficients using Gauss Quadrature.
    """
```

```
    e_ijk = {}

    num_polynomials = len(polynomials)
    c = 1 # Classical Gauss Quadrature (c=1)
    quadrature_order = math.ceil((3 * order + c)/2)
    nodes, weights = cp.generate_quadrature(quadrature_order,
                                            distribution, rule = "gaussian")
    for i in range(num_polynomials):
        for j in range(num_polynomials):
            for k in range(num_polynomials):
                # Using Quadrature
                integrand = (polynomials[i](nodes)
                             * polynomials[j](nodes)
                             * polynomials[k](nodes))
                e_ijk[(i, j, k)] = np.sum(weights * integrand)
    return e_ijk

# Main function to calculate tensor coefficients and write output
def calculate_and_write_tensorCoeffs(order, poly_type, calculation_type,
                                     output_path):

    # Step 1: Generate distribution and polynomials
    distribution, polynomials = generate_distribution_and_polynomials(
        order, poly_type)
    # Step 2: Orthonormality Verification
    orthonormality_verif_message = orthonormality_verification(polynomials,
                                                               distribution,
                                                               tol=1e-10,
                                                               verbose=True)
    # Step 3: Calculate tensor coefficients
    # Calculate tensor coefficients directly due to normalisation
    if calculation_type == "tensorCoeff":
        tensorCoeffs = calculate_triple_product(order, distribution, polynomials)
    else:
        raise ValueError("Invalid calculation_type. Choose 'tensorCoeff'.")

    # Step 4: Galerkin Coefficient Verification
    galerkin_coeff_verif_message = galerkin_coefficient_verification(tensorCoeffs, order,
     tol=1e-10, verbose=True)
    # Write output to file in OpenFOAM-friendly format
    with open(output_path, "w") as output_file:
        output_file.write("/*--------------------------------*- C++
     -*----------------------------------*\\n")
        output_file.write("| =========                 |
                  |\n")
        output_file.write("| \\\\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox           |\n")
        output_file.write("|  \\\\    /   O peration     | Version:  v2306
                    |\n")
        output_file.write("|   \\\\  /    A nd           | Website:  www.openfoam.com
                    |\n")
        output_file.write("|    \\\\/     M anipulation  |
                    |\n")
        output_file.write("
     \\*---------------------------------------------------------------------------*/\n")
        output_file.write("FoamFile\n{\n")
        output_file.write("    version     2.0;\n")
        output_file.write("    format      ascii;\n")
        output_file.write("    class       dictionary;\n")
        output_file.write("    location    \"constant\";\n")
        output_file.write("    object      gPCCoeff;\n")
        output_file.write("}\n")
        output_file.write(f"// Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\
     n")
        output_file.write("//
     ************************************************************************ //\n\n")

        # Append Verification Message
```

```
        output_file.write(f"// {orthonormality_verif_message}\n\n")
        output_file.write(f"// {galerkin_coeff_verif_message}\n\n")
        # Write computed tensor coefficients in OpenFOAM format
        for (i, j, k), value in tensorCoeffs.items():
            if not  np.isclose(value, 0.0, rtol=1e-6, atol=1e-10):
                output_file.write(f"e[{i}][{j}][{k}] {value};\n")

# Execute the process by reading UQProperties and generating coefficients
order, poly_type, calculation_type, additional_params = read_UQProperties(uqproperties_path)
output_path = os.path.join(script_dir,"..","stochasticBurgersBCs_Study","constant","gPCCoeff
    ")
calculate_and_write_tensorCoeffs(order, poly_type, calculation_type, output_path)
```

C.1.2 Polynomial Triple Product Coefficients (e_{ijk}) (Stand-alone Script)

Listing C.2: gPCCoeff_standalone_v002.py

```
import os
import chaospy as cp
import numpy as np
import math
from datetime import datetime
from gPCVerification_v002 import (orthonormality_verification,
galerkin_coefficient_verification)
# User-defined inputs
POLY_ORDER = 3  # Polynomial expansion order
POLY_TYPE = "legendre"  # Polynomial type (e.g., "legendre")

# Function to generate distribution and polynomials
def generate_distribution_and_polynomials(order, poly_type):
    if poly_type.lower() == "legendre":
        lower_std, upper_std = -1, 1  # Standard bounds for Legendre polynomials
        distribution = cp.Uniform(lower=lower_std, upper=upper_std)
        polynomials = cp.expansion.legendre(
            order, lower=lower_std, upper=upper_std, physicist=False, normed=True
        )
    else:
        raise ValueError("Unsupported polynomial type. Use 'legendre'.")
    return distribution, polynomials

# Function to calculate triple product coefficients
def calculate_triple_product(order, distribution, polynomials):
    """
    Calculate triple product coefficients e_ijk using Gaussian Quadrature.

    Args:
        order: polynomial order
        distribution: probability density distribution
        polynomials: polynomials expansion coefficients (e.g., Legendre).

    Returns:
        e_ijk: Triple product coefficients using Gauss Quadrature.
    """
    e_ijk = {}
    num_polynomials = len(polynomials)
    c = 1  # Classical Gauss Quadrature (c=1)
    quadrature_order = math.ceil((3 * order + c) / 2)
    nodes, weights = cp.generate_quadrature(
        quadrature_order, distribution, rule="gaussian"
    )
    for i in range(num_polynomials):
        for j in range(num_polynomials):
            for k in range(num_polynomials):
                integrand = (
                    polynomials[i](nodes)
                    * polynomials[j](nodes)
```

```
                    * polynomials[k](nodes)
                )
                e_ijk[(i, j, k)] = np.sum(weights * integrand)
    return e_ijk

# Main function for coefficient generation and file output
def generate_galerkin_coefficients(order, poly_type):
    # Step 1: Generate the distribution and polynomials
    distribution, polynomials = generate_distribution_and_polynomials(order, poly_type)

    # Step 2: Orthonormality Verification
    orthonormality_verif_message = orthonormality_verification(polynomials,
                                                               distribution,
                                                               tol=1e-10,
                                                               verbose=True)
    # Step 3: Calculate tensor coefficients
    tensorCoeffs = calculate_triple_product(order, distribution, polynomials)

    # Determine output path in the same folder as this script
    script_dir = os.path.dirname(os.path.abspath(__file__))
    output_path = os.path.join(script_dir, "gPCCoeff")

    # Step 4: Galerkin Coefficient Verification
    galerkin_coeff_verif_message = galerkin_coefficient_verification(tensorCoeffs, order,
     tol=1e-10, verbose=True)

    # Step 5: Write output in OpenFOAM-friendly format
    with open(output_path, "w") as output_file:
        output_file.write("/*--------------------------------*- C++
     -*----------------------------------*\\n")
        output_file.write("| =========                 |
                  |\n")
        output_file.write("| \\\\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox           |\n")
        output_file.write("|  \\\\    /   O peration     | Version:  v2306
                    |\n")
        output_file.write("|   \\\\  /    A nd           | Website:  www.openfoam.com
                    |\n")
        output_file.write("|    \\\\/     M anipulation  |
                    |\n")
        output_file.write("
     \\*---------------------------------------------------------------------------*/\n")
        output_file.write("FoamFile\n{\n")
        output_file.write("    version     2.0;\n")
        output_file.write("    format      ascii;\n")
        output_file.write("    class       dictionary;\n")
        output_file.write("    location    \"constant\";\n")
        output_file.write("    object      gPCCoeff;\n")
        output_file.write("}\n")
        output_file.write(f"// Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\
     n")
        output_file.write("//
     ********************************************************************** //\n\n")
        # Append Verification Message
        output_file.write(f"// {orthonormality_verif_message}\n\n")
        output_file.write(f"// {galerkin_coeff_verif_message}\n\n")
        # Write computed tensor coefficients in OpenFOAM format
        for (i, j, k), value in tensorCoeffs.items():
            if not  np.isclose(value, 0.0, rtol=1e-6, atol=1e-10):
                output_file.write(f"e[{i}][{j}][{k}] {value};\n")

    print(f"Galekin coefficients written to: {output_path}")
    print("You can copy the file to your OpenFOAM case directory under 'constant/'.")

# Automatically run the function
if __name__ == "__main__":
    print(f"Generating Galerkin coefficients with order={POLY_ORDER} and type='{POLY_TYPE}'"
     )
```

```
    generate_galerkin_coefficients(POLY_ORDER, POLY_TYPE)
```

C.1.3 Auxiliary File Orthonormality and Galerkin Coefficient Verification

Listing C.3: gPCVerification_v002.py

```
"""
gPCVerification.py

This module provides verification functions for generalised Polynomial Chaos
(gPC) expansions. It combines functionality from Chaospy for precomputing
coefficients and orthonormal polynomials with manual verification methods
based on Rodrigues' formula.

Key Features:
1. Orthonormality Verification:
   Verify the orthonormality of polynomial bases used in gPC expansions
   (via Chaospy-generated polynomials and distributions). This includes the
   computation of the **Gram matrix**, which is verifies against the
   identity matrix to confirm orthonormality.
2. Galerkin Coefficient Verification:
   Compare manually computed triple product coefficients (using Rodrigues'
   formula) with precomputed coefficients (from Chaospy or other sources).
3. Simplified Polynomial Normalisation:
   Directly normalise Legendre polynomials based on their known theoretical
   properties to ensure properties like mean = 0 and variance = 1.

Key Functions:
- orthonormality_verification:
  Computes the **Gram matrix** for the polynomial basis and verifies
  its orthonormality against the distribution.
- galerkin_coefficient_verification:
  Computes triple product coefficients manually and verifies them against
  precomputed values for correctness.

Usage:
1. Use Chaospy to precompute coefficients in your main script.
2. Pass the precomputed coefficients as input to the verification functions.
3. Perform orthonormality checks (via the Gram matrix) and
Galerkin coefficient verification.

Example:
    Precomputed coefficients can be generated using Chaospy:
    >>> precomputed_coeffs = {...}
    >>> galerkin_coefficient_verification(precomputed_coeffs, order=3, tol=1e-10, verbose=
     True)

This module enables both numerical and manual verification for robust
gPC expansions, supporting higher polynomial orders and adaptability
to different orthogonal polynomial systems.
"""

import chaospy as cp
import numpy as np
from scipy.integrate import quad
from scipy.special import factorial

"Orthonormality Verification----------------------------------------------------"

# Verify the orthonormality of the generated polynomials.
# Checks that <Ii, Ij> = Deltaij (orthonormality).
def orthonormality_verification(polynomials, distribution, tol=1e-10,
                                verbose=False):
    """
    Verify the orthonormality of the generated polynomials using the
```

```
    Gram matrix.
    Ensures that <Ii, Ij> = Deltaij (orthonormality).
    Parameters:
    - polynomials: List of Chaospy-generated polynomials to verificate.
    - distribution: Chaospy distribution used to compute the inner products.
    - verbose: If True, prints the Gram matrix and verification status.
    Returns:
    - A message indicating the result of the verification.
    """
    # Compute the Gram matrix
    num_polys = len(polynomials)
    G = np.zeros((num_polys, num_polys))
    for i in range(num_polys):
        for j in range(num_polys):
            G[i, j] = cp.E(polynomials[i] * polynomials[j], distribution)

    # Optionally print the Gram matrix
    if verbose:
        print("Gram matrix (Gij):")
        print(G)

    # Verify if Gram matrix is an identity matrix
    # tol = 1e-10  # Adjust this tolerance if needed
    is_identity = np.allclose(G, np.eye(num_polys), atol=tol)

    if not is_identity:
        if verbose:
            print("\nVerification failed: Gram matrix is"
                  "not an identity matrix.")
            print("Deviation from identity matrix:")
            print(G - np.eye(num_polys))
        raise ValueError("Verification failed: Gram matrix is"
                         "not an identity matrix.")

    if verbose:
        print("\nVerification of OrthoNormality Passed.")

    return "Verification of OrthoNormality Passed."

# ------------------------------------------------------------------------
"Galerkin Coefficient Verification-----------------------------------------"

# Step 1: Define Legendre polynomials using Rodrigues' formula
def legendre_poly(n, x):
    """
    Compute the nth Legendre polynomial using Rodrigues' formula.
    """
    coeff = 1 / (2**n * factorial(n))
    poly = coeff * np.polyder(np.poly1d([1, 0, -1])**n, n)
    return np.polyval(poly, x)

# Step 2: Directly normalise Legendre polynomials

def normalised_legendre_poly(n, x):
    """
    Compute the normalised nth Legendre polynomial.
    Uses theoretical properties to ensure variance=1 and mean=0 under
    uniform distribution [-1, 1].
    """
    P_n = legendre_poly(n, x)  # Standard Legendre polynomial
    return np.sqrt((2 * n + 1)) * P_n

# Step 3: Compute triple product coefficients manually
def compute_triple_product(i, j, k):
    """
    Compute the triple product coefficient M_ijk for manually normalised polynomials.
    """
    def integrand(x):
```

```
        return (normalised_legendre_poly(i, x) *
                normalised_legendre_poly(j, x) *
                normalised_legendre_poly(k, x)) * 0.5  # Weight function f(x) = 0.5

    result, _ = quad(integrand, -1, 1)
    return result

# Step 4: Galerkin Coefficient verification
def galerkin_coefficient_verification(precomputed_coeffs, order, tol, verbose=False):
    """
    Verify Galerkin coefficients using manually computed triple products and compare with
    precomputed coefficients provided as input.

    Args:
        precomputed_coeffs (dict): Precomputed triple product coefficients.
        order (int): Maximum polynomial order for verification.
        tol (float): Tolerance for coefficient differences.
        verbose (bool): If True, prints detailed results.

    Returns:
        dict: Manually computed tensor coefficients for the polynomials.
    """
    num_polys = order + 1
    if verbose:
        print(f"Verifying Galerkin coefficients up to order {order}...")

    # Step 4.1: Compute all triple product coefficients manually
    coefficients_manual = {}
    for i in range(num_polys):
        for j in range(num_polys):
            for k in range(num_polys):
                coefficients_manual[(i, j, k)] = compute_triple_product(i, j, k)

    # Step 4.2: Compare with precomputed coefficients
    max_diff = 0
    for key, manual_value in coefficients_manual.items():
        # Retrieve the corresponding precomputed coefficient
        precomputed_value = precomputed_coeffs.get(key, 0.0)

        # Compute the difference
        diff = abs(precomputed_value - manual_value)

        # Print detailed comparison if verbose
        if verbose:
            print(f"M_{key}: Manual = {manual_value:.12f},"
                  f"Chaospy-Precomputed = {precomputed_value:.12f}, Diff = {diff:.12e}")
        # Track the largest difference
        if diff > tol:
            max_diff = max(max_diff, diff)

    # Raise an error if the difference exceeds tolerance
    if max_diff > tol:
        raise ValueError(f"Galerkin Coefficients Verification failed: max_diff = {max_diff}
     exceeds tolerance {tol}.")

    if verbose:
        print(f"Galerkin Coefficients Verification successful within tolerance {tol}.")
    # return coefficients_manual
    return f"Galerkin Coefficients Verification successful within tolerance {tol}."

# ------------------------------------------------------------------------------
# # Example Usage----------------------------------------------------------------
# if __name__ == "__main__":
#     # Example: Precomputed coefficients for up to 2nd-order Legendre polynomials
#     # Explicitly include all coefficients, setting zero coefficients to 0.0
#     precomputed_coeffs_example = {
#         (0, 0, 0): 1.0,       # Triple product <P0, P0, P0>
#         (0, 1, 1): 1.0,       # Triple product <P0, P1, P1>
```

```
#         (0, 2, 2): 1.0,       # Triple product <P0, P2, P2>
#         (1, 0, 1): 1.0,       # Triple product <P1, P0, P1>
#         (1, 1, 0): 1.0,       # Triple product <P1, P1, P0>
#         (1, 1, 2): 0.894427190999916,  # Triple product <P1, P1, P2>
#         (1, 2, 1): 0.894427190999916,  # Triple product <P1, P2, P1>
#         (2, 0, 2): 1.0,       # Triple product <P2, P0, P2>
#         (2, 2, 0): 1.0,       # Triple product <P2, P2, P0>
#         (2, 1, 1): 0.894427190999916,  # Triple product <P2, P1, P1>
#         (2, 2, 2): 0.6388765649999391,  # Triple product <P2, P2, P2>
#         # Coefficients that should theoretically be zero
#         (0, 0, 1): 0.0,
#         (0, 0, 2): 0.0,
#         (1, 1, 1): 0.0,
#         (1, 2, 2): 0.0,
#     }

#     # Polynomial order and tolerance
#     max_order = 2  # Set maximum polynomial order to 2
#     tol = 1e-10    # Set a tolerance for verification

#     # Verify manually computed coefficients against precomputed ones
#     verified_coefficients = galerkin_coefficient_verification(
#         precomputed_coeffs=precomputed_coeffs_example,
#         order=max_order,
#         tol=tol,
#         verbose=True
#     )

#     # Print verified coefficients
#     print("\n=== Verified Coefficients ===")
#     for key, value in verified_coefficients.items():
#         if not  np.isclose(value, 0.0, rtol=1e-6, atol=1e-10):
#             print(f"M_{key} = {value:.12f}")
```

C.1.4 Auxiliary File Read UQProperties from OpenFOAM

Listing C.4: `read_UQProperties_v001.py`

```
import os
import chaospy as cp
import numpy as np
# from datetime import datetime
from PyFoam.RunDictionary.ParsedParameterFile import ParsedParameterFile

# Locate UQProperties in the same directory as the script
# script_dir = os.path.dirname(os.path.abspath(__file__))
# uqproperties_path = os.path.join(script_dir, "..", "stochasticBurgersBCs_Study", "constant
    ", "UQProperties")

# Function to read properties from UQProperties using pyFoam
def read_UQProperties(uqproperties_path):
    properties = ParsedParameterFile(uqproperties_path)
    order = properties["order"]
    poly_type = properties["poly_type"]
    calculation_type = properties["calculation_type"]

    additional_params = {}
    if poly_type.lower() == "legendre":
        if "lower" not in properties or "upper" not in properties:
            raise KeyError("Legendre polynomials require 'lower' and 'upper' in UQProperties
    .")
        additional_params["lower"] = properties["lower"]
        additional_params["upper"] = properties["upper"]
    else:
        raise ValueError("Unsupported polynomial type. Use 'legendre'.")
```

```
    return order, poly_type, calculation_type, additional_params
```

C.1.5 Auxiliary File Read Transport Properties from OpenFOAM

Listing C.5: read_transportProperties_v000.py

```
import os
from PyFoam.RunDictionary.ParsedParameterFile import ParsedParameterFile

# Locate transportProperties in the same directory as the script
# script_dir = os.path.dirname(os.path.abspath(__file__))
# transportProperties_path = os.path.join(script_dir, "transportProperties")

# Function to read properties from transportProperties using pyFoam
def read_transportProperties(transportProperties_path):
    try:
        # Check if the file exists
        if not os.path.exists(transportProperties_path):
            raise FileNotFoundError(f"File not found: {transportProperties_path}")

        # Read the file using ParsedParameterFile
        properties = ParsedParameterFile(transportProperties_path)

        # Check if 'nu' is in the properties
        if "nu" not in properties:
            raise KeyError("'nu' not found in transportProperties")

        # Extract only the numeric value of 'nu'
        nu = properties["nu"][-1]  # Access the last element which is the value (0.05 in
    this case)
        # print(f"Extracted 'nu' value: {nu}")
        return nu
    except Exception as e:
        print(f"Error reading transportProperties: {e}")
        return None

# Call the function
# nu_value = read_transportProperties(transportProperties_path)

# if nu_value is not None:
#     print(f"Successfully extracted 'nu': {nu_value}")
# else:
#     print("Failed to extract 'nu'.")
```

C.2 Post-processing Tools

C.2.1 Burgers Equation Steady-state Exact Solution

Listing C.6: burgersEqExactSolution_v000.py

```
import numpy as np
from scipy.optimize import root

#--Note on Burgers' Exact Solution-------------------------------------------
# source: from section 2.1 "Exact Solution" on "Supersensitivity due to
# uncertain boundary conditions" by Xiu, Dongbinand Karniadakis, George Em 2004

# This function `burgers_exact_solution` finds values of A (slope) and
# z_ex (transition layer location)
# that satisfy the steady-state boundary conditions of the viscous Burgers'
# equation.
# Specifically, it solves the two boundary equations for u(-1) = (1 + delta)
# and u(1) = -1.
# These boundary conditions are implemented as equations (u_left and u_right)
# in the inner function `boundary_system`.
# The scipy `root` function then finds values of A and z_ex that make both
# equations true, ensuring that the solution meets the boundary conditions
# specified in the problem.
#-----------------------------------------------------------------------------
def burgers_exact_solution(delta, nu, tol, initial_guess):
    def boundary_system(vars):
        A, z_ex = vars
        u_left = A * np.tanh((A / (2 * nu)) * (1 + z_ex)) - (1 + delta)
        u_right = A * np.tanh((A / (2 * nu)) * (1 - z_ex)) - 1
        return [u_left, u_right]

    solution = root(boundary_system, initial_guess, tol=tol)
    return solution.x if solution.success else (None, None)

def exact_solution_u(x, A, z_ex, nu):
    return -A * np.tanh((A / (2 * nu)) * (x - z_ex))
```

C.2.2 Deterministic Solver Verification

Listing C.7: deterministicSolverVerification_v003.py

```
import re
import os
import sys
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
from burgersEqExactSolution_v000 import burgers_exact_solution, exact_solution_u

#--Get the directory where the script is located-----------------------------
script_dir = os.path.dirname(os.path.abspath(__file__))
preprocessing_dir = os.path.join(script_dir, "..","preProcessingScripts")
sys.path.append(os.path.abspath(preprocessing_dir))

#--Import External Modules---------------------------------------------------
from read_UQProperties_v001 import read_UQProperties
from read_transportProperties_v000 import read_transportProperties
#--Get the directory containing the cases------------------------------------
study = "deterministicBurgersBCs_Study"
cases_dir = os.path.join(script_dir, "..", study)

#--Function to extract the numeric part from case names and sort them accordingly
def get_case_names(cases_dir):
    # List all directories that start with "Case"
    case_names = [
```

```
        name for name in os.listdir(cases_dir)
        if os.path.isdir(os.path.join(cases_dir, name))
        and name.startswith("Case")
    ]

    # Sort case names by extracting the numeric part after "Case_"
    sorted_case_names = sorted(case_names,
                               key=lambda x: int(re.search(r'\d+', x).group()))
    return sorted_case_names

# Automatically get the case names from the folder
cases = get_case_names(cases_dir)

#--UQProperties-------------------------------------------------------------
#--Function to read UQ properties for a given case-----------------------------
def get_UQProperties(case_name, cases_dir):
    uqproperties_path = os.path.join(cases_dir, case_name, "constant",
                                     "UQProperties")
    return read_UQProperties(uqproperties_path)

#--transportProperties------------------------------------------------------
def get_transportProperties(case_name, cases_dir):
    transportProperties_path = os.path.join(cases_dir, case_name, "constant",
                                            "transportProperties")
    return read_transportProperties(transportProperties_path)

#--Accessing the properties for a specific case-------------------------------
#--Case-All-----------------------------------------------------------------
# (here the Case 1 Properties are used for all of the cases)
UQProperties = get_UQProperties(cases[0], cases_dir)
order = UQProperties[0]   # Polynomial order
bounds = UQProperties[3]  # Lower and upper bounds
lower = bounds ['lower']
upper = bounds ['upper']
#--transportProperties
nu_value = get_transportProperties(cases[0], cases_dir)

#--Define parameters in this main script-------------------------------------
nu = nu_value
delta_lower = lower  # unperturbed case
delta_upper = upper   # perturbed case
exacSolTol = 1e-12
x_values = np.linspace(-1, 1, 10000) # Generate x-values for exact solution

#--Setup cases with delta and initial guess----------------------------------
cases = {
    "Case_1_detLBBCs_Verification":
        {"delta": delta_lower, "initial_guess": [-1.0, 0.0]},
    "Case_2_detUBBCs_Verification":
        {"delta": delta_upper, "initial_guess": [-1.0, 1.0]}
}

#--Plot Configuration
fontSize = 10
fontSizeLegend = 11
fontsizeAxisLabel = 11
lineWidth = 1.0
tickPad = 5
labelPadY = 4
labelPadX = 4
boxPad = 2
tickLength = 2
markerSize = 4
#--HTML code for colours----------------------------------------------------
black = '#000000'
grey = '#555555'
greyLight = '#B0B0B0'
blue = '#011231'
```

```
blueLight = '#cddefd'
red = '#E50037'
redLight = '#FCC8CF'

#--Enable LaTeX rendering in Matplotlib for consistency-----------------------
plt.rcParams['text.usetex'] = True
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['Computer Modern Roman']  # LaTeX's default font
plt.rcParams['axes.linewidth'] = lineWidth

#--Create a figure with controlled aspect ratio--------------------------------
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(11, 5), gridspec_kw={'height_ratios': [1.5,
    1]},sharex=True)
fig.subplots_adjust(hspace=0.1)

#--Get the directory where the script is located-------------------------------
script_dir = os.path.dirname(os.path.abspath(__file__))

#--Prepare output directory for results data-----------------------------------
results_data_dir = os.path.join(script_dir, "..",
                                "deterministicBurgersBCs_Study", "Results", "resultsData")
os.makedirs(results_data_dir, exist_ok=True)  # Ensure directory exists
output_file_path = os.path.join(results_data_dir, "
    deterministicBurgers_BCs_Study_Verification.txt")

#--Open the output file in write mode------------------------------------------
with open(output_file_path, "w") as output_file:
    #--Define colour and linestyle dictionaries for each case------------------
    case_styles = {
    "Case_1_detLBBCs_Verification": {"exact_color": greyLight,"exact_line": "-"
                                    , "foam_color": black, "foam_line": ":"},
    "Case_2_detUBBCs_Verification": {"exact_color": greyLight , "exact_line" :"-",
                                   "foam_color": black,"foam_line": "--"}
    }
    #---------------------------------------------------------------------------
    case_labels = {
    "Case_1_detLBBCs_Verification": {"exactSol_label":
        f"Lower Bound ('Exact Solution')", "foamSol_label":
                                      f"Lower Bound"},
    "Case_2_detUBBCs_Verification": {"exactSol_label":
        f"Upper Bound ('Exact Solution')", "foamSol_label":
                                      f"Upper Bound"}
    }

    #--Loop through each case------------------------------------------------
    for case_name, params in cases.items():
        delta = params["delta"]
        initial_guess = params["initial_guess"]

        # Get colours for each case from the dictionary-----------------------
        exact_color = case_styles[case_name]["exact_color"]
        exact_line = case_styles[case_name]["exact_line"]
        foam_color = case_styles[case_name]["foam_color"]
        foam_line = case_styles[case_name]["foam_line"]

        # Get labels for each case from the dictionary------------------------
        exactSol_label = case_labels[case_name]["exactSol_label"]
        foamSol_label = case_labels[case_name]["foamSol_label"]

        #--Calculate exact solution parameters-------------------------------
        A, z_ex = burgers_exact_solution(delta, nu, exacSolTol, initial_guess)

        if A is not None and z_ex is not None:
            u_values_exact = exact_solution_u(x_values, A, z_ex, nu)
            output_file.write(f"\n{case_name} ('Exact Solution'): nu = {nu}, "
                              f"delta = {delta}, "
                              f"z_ex = {z_ex:.8f} (Transition Layer Location)\n")
            print(f"\n{case_name} ('Exact Solution'): nu = {nu}  delta = {delta},"
```

```
                f" z_ex = {z_ex:.8f} (Transition Layer Location)")
    else:
        print(f"Failed to compute exact solution for {case_name}.")
        output_file.write(f"Failed to compute"
                          f" exact solution for {case_name}.\n")
        continue

    #--Construct post-processing path--------------------------------------
    post_process_dir = os.path.join(script_dir, "..",
                    "deterministicBurgersBCs_Study", "Results",
                    "sampledData", case_name, "postProcessing", "Usample")

    #--Check if the directory exists---------------------------------------
    if not os.path.isdir(post_process_dir):
        print(f"Directory not found for {case_name}: {post_process_dir}")
        output_file.write(f"Directory not found for "
                          f"{case_name}: {post_process_dir}\n")
        continue

    #--Locate latest time folder in Usample--------------------------------
    time_dirs = [d for d in os.listdir(post_process_dir)
                 if d.replace(".", "", 1).replace("e-", "", 1).isdigit()]
    if time_dirs:
        latest_time = sorted(time_dirs, key=lambda x: float(x))[-1]

        sample_file = os.path.join(post_process_dir,
                                   latest_time, "lineX_U.xy")
        if os.path.exists(sample_file):
            foam_data = np.loadtxt(sample_file)
            x_foam, U_x_foam = foam_data[:, 0], foam_data[:, 1]

            #--Plot the exact solution and simulation results--------------
            ax1.plot(x_values, u_values_exact, label=f"{exactSol_label} ($\\nu$ = {nu},
$\\delta$ = {delta})",
                     linestyle=exact_line, color=exact_color,
                     linewidth=2)

            ax1.plot(x_foam, U_x_foam, label=f"{foamSol_label} ($\\nu$ = {nu}, $\\delta$
 = {delta})", linestyle=foam_line,
                     color=foam_color, linewidth=1)
            ax1.grid(color=grey, linewidth=lineWidth, alpha=0.3)
            ax1.tick_params(which='both', direction='in', length=tickLength, width=
lineWidth,  pad=tickPad, color=grey)
            ax1.yaxis.set_ticks_position('both')
            ax1.xaxis.set_ticks_position('both')
            ax1.spines['bottom'].set_color(grey)
            ax1.spines['top'].set_color(grey)
            ax1.spines['right'].set_color(grey)
            ax1.spines['left'].set_color(grey)
            #-----------------------------------------------------------------
            # Calculate exact_interpolated for the relative error calculation
            interpolation_exact = interp1d(x_values, u_values_exact, kind='linear')
            U_exact_interpolated = interpolation_exact(x_foam)

            #--Calculate and plot L2 norm and relative error---------------
            L2_norm = np.sqrt(np.sum((U_exact_interpolated - U_x_foam) ** 2))
            relative_error = ((U_x_foam - U_exact_interpolated) / U_exact_interpolated)
*100
            ax2.plot(x_foam, relative_error, label=f"{foamSol_label} Relative Error. "
                     f"$L_2$-norm error = {L2_norm:.1e}",
                     linestyle=foam_line, linewidth=1, color=foam_color)

            #--Calculate Transition Layer Location (z_foam)----------------
            zero_crossings = np.where(np.diff(np.sign(U_x_foam)))[0]
            if zero_crossings.size > 0:
                idx = zero_crossings[0]
                z_foam = interp1d(U_x_foam[idx:idx + 2],
                                  x_foam[idx:idx + 2], kind='linear')(0)
```

```
                    #--Format & output the OpenFOAM solver transition layer data
                    output_text = (f"\n{case_name} ('myBurgersFoam' Simulation):"
                                   f" nu = {nu}, delta = {delta}, "
                                   f"z_foam = {z_foam:.8f} "
                                   "(Transition Layer Location)\n")
                    print(output_text)
                    output_file.write(output_text)  # Write to file
                else:
                    output_text = f"{case_name} ('myBurgersFoam' Simulation):"
                    f"No transition layer (zero-crossing) found.\n"
                    print(output_text)
                    output_file.write(output_text)  # Write to file
            else:
                output_text = f"Sample file {sample_file} not found.\n"
                print(output_text)
                output_file.write(output_text)  # Write to file
        else:
            output_text = f"No time directories found in: {post_process_dir}\n"
            print(output_text)
            output_file.write(output_text)  # Write to file

#--Plotting Data-------------------------------------------------------------
ax1.axhline(y=0, color='grey', linestyle='-', linewidth=0.5)
ax1.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
ax1.set_ylabel(r"$u(x)$ [m/s]", fontsize=fontsizeAxisLabel)
ax1.legend()
ax1.grid(False)
# ax1.tight_layout()
fig.tight_layout(pad=boxPad)

#--Configure the subplot (relative error)------------------------------------
ax2.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
ax2.set_ylabel("Relative Error [$\%$]", fontsize=fontsizeAxisLabel)
ax2.legend(loc='lower left', fontsize=fontSizeLegend)
ax2.grid(color=grey, linewidth=lineWidth, alpha=0.3)
ax2.grid(False)
ax2.tick_params(which='both', direction='in', length=tickLength, width=lineWidth,  pad=
    tickPad, color=grey)
ax2.yaxis.set_ticks_position('both')
ax2.xaxis.set_ticks_position('both')
ax2.spines['bottom'].set_color(grey)
ax2.spines['top'].set_color(grey)
ax2.spines['right'].set_color(grey)
ax2.spines['left'].set_color(grey)

#--Construct Plots path------------------------------------------------------
plot_dir = os.path.join(script_dir, "..",
                        "deterministicBurgersBCs_Study", "Results", "plots")
os.makedirs(plot_dir, exist_ok=True)  # Ensure directory exists

#--Show and Save Plot--------------------------------------------------------
filename = 'myBurgersFoam_Verification.pdf'
plt.savefig(os.path.join(plot_dir, filename), dpi=600, format='pdf')

plt.show()
```

C.2.3 Stochastic Solver Verification

Listing C.8: `stochasticSolverVerification_v000.py`

```
import re
import os
import sys
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
```

```
from burgersEqExactSolution_v000 import burgers_exact_solution, exact_solution_u

#--Get the directory where the script is located-------------------------------
script_dir = os.path.dirname(os.path.abspath(__file__))
preprocessing_dir = os.path.join(script_dir, "..","preProcessingScripts")
sys.path.append(os.path.abspath(preprocessing_dir))

#--Import External Modules-----------------------------------------------------
from read_UQProperties_v001 import read_UQProperties
from read_transportProperties_v000 import read_transportProperties
#--Get the directory containing the cases--------------------------------------
study = "stochasticBurgersBCs_Study"
cases_dir = os.path.join(script_dir, "..", study)

#--Function to extract the numeric part from case names and sort them accordingly
def get_case_names(cases_dir):
    # List all directories that start with "Case"
    case_names = [
        name for name in os.listdir(cases_dir)
        if os.path.isdir(os.path.join(cases_dir, name))
        and name.startswith("Case")
    ]

    # Sort case names by extracting the numeric part after "Case_"
    sorted_case_names = sorted(case_names,
                               key=lambda x: int(re.search(r'\d+', x).group()))
    return sorted_case_names

# Automatically get the case names from the folder
cases = get_case_names(cases_dir)

#--UQProperties----------------------------------------------------------------
#--Function to read UQ properties for a given case-----------------------------
def get_UQProperties(case_name, cases_dir):
    uqproperties_path = os.path.join(cases_dir, case_name, "constant",
                                     "UQProperties")
    return read_UQProperties(uqproperties_path)

#--transportProperties---------------------------------------------------------
def get_transportProperties(case_name, cases_dir):
    transportProperties_path = os.path.join(cases_dir, case_name, "constant",
                                            "transportProperties")
    return read_transportProperties(transportProperties_path)

#--Accessing the properties for a specific case--------------------------------
#--Case-1----------------------------------------------------------------------
case_1_UQProperties = get_UQProperties(cases[0], cases_dir)
case_1_order = case_1_UQProperties[0]   # Polynomial order
case_1_bounds = case_1_UQProperties[3]  # Lower and upper bounds
case_1_lower = case_1_bounds ['lower']
case_1_upper = case_1_bounds ['upper']
#--transportProperties
case_1_nu_value = get_transportProperties(cases[0], cases_dir)

#--Case-2----------------------------------------------------------------------
case_2_UQProperties = get_UQProperties(cases[1], cases_dir)
case_2_order = case_2_UQProperties[0]   # Polynomial order
case_2_bounds = case_2_UQProperties[3]  # Lower and upper bounds
case_2_lower = case_2_bounds ['lower']
case_2_upper = case_2_bounds ['upper']
#--transportProperties
case_2_nu_value = get_transportProperties(cases[1], cases_dir)

#--Define parameters in this main script--------------------------------------
# nu = 0.05
exacSolTol = 1e-12
delta_unperturbed = case_1_lower  # unperturbed case
delta_perturbed = case_2_upper  # perturbed case
```

```
x_values = np.linspace(-1, 1, 10000) # Generate x-values for exact solution
#--Setup cases with delta and initial guess-----------------------------------
casesData = {
    cases[0]:
        {"delta": delta_unperturbed, "initial_guess": [-1.0, 0.0],
         "order": case_1_order, "nu": case_1_nu_value},
    cases[1]:
        {"delta": delta_perturbed, "initial_guess": [-1.0, 1.0],
         "order": case_2_order, "nu": case_2_nu_value}
}

#--Prepare output directory for results data----------------------------------
results_data_dir = os.path.join(script_dir, "..",
                                "stochasticBurgersBCs_Study", "Results",
                                "resultsData")
os.makedirs(results_data_dir, exist_ok=True)  # Ensure directory exists
output_file_path = os.path.join(results_data_dir,
                                "stochasticBurgers_BCs_Study_Verification.txt")

#--Prepare input file name----------------------------------------------------
def generate_filename(base_name, order, extension):
    # Create the filename dynamically based on the polynomial order
    variable_parts = [f"Uhat{i}" for i in range(order + 1)]
    filename = f"{base_name}_" + "_".join(variable_parts) + f".{extension}"
    return filename

#--Plot Configuration---------------------------------------------------------
fontSize = 10
fontSizeLegend = 11
fontsizeAxisLabel = 11
lineWidth = 1.0
tickPad = 5
labelPadY = 4
labelPadX = 4
boxPad = 2
tickLength = 2
markerSize = 4

#--HTML code for colours------------------------------------------------------
black = '#000000'
grey = '#555555'
greyLight = '#B0B0B0'
blue = '#011231'
blueLight = '#cddefd'
red = '#E50037'
redLight = '#FCC8CF'

#--Enable LaTeX rendering in Matplotlib for consistency-----------------------
plt.rcParams['text.usetex'] = True
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['Computer Modern Roman']  # LaTeX's default font
plt.rcParams['axes.linewidth'] = lineWidth

#--Create a figure with controlled aspect ratio-------------------------------
# Create figure and subplots
fig, (ax1, ax2) = plt.subplots(2, 1,
                               figsize=(12, 5),
                               gridspec_kw={'height_ratios': [1.5, 1]},
                               sharex=True)
fig.subplots_adjust(hspace=0.15)

#--Open the output file in write mode-----------------------------------------
with open(output_file_path, "w") as output_file:
    #--Define colour and linestyle dictionaries for each case-----------------
    case_styles = {
    "Case_1_stocLBBCs_Verification":
        {"exact_color": greyLight,"exact_line": "-"
                                , "foam_color": black, "foam_line": ":"},
```

```
    "Case_2_stocUBBCs_Verification": {"exact_color": greyLight ,
                                     "exact_line" :"-",
                                "foam_color": black,"foam_line": "--"}
    }

    #------------------------------------------------------------------------
    case_labels = {
    "Case_1_stocLBBCs_Verification": {"exactSol_label":
        f"Lower Bound ('Exact Solution')", "foamSol_label":
                                     f"Lower Bound"},
    "Case_2_stocUBBCs_Verification": {"exactSol_label":
        f"Upper Bound ('Exact Solution')", "foamSol_label":
                                     f"Upper Bound"}
    }

    #--Loop through each case-------------------------------------------------
    for case_name, params in casesData.items():
        delta = params["delta"]
        initial_guess = params["initial_guess"]
        order = params["order"]
        nu = params["nu"]

        # Get colours and linestyle for each case from the dictionary----------
        exact_color = case_styles[case_name]["exact_color"]
        exact_line = case_styles[case_name]["exact_line"]
        foam_color = case_styles[case_name]["foam_color"]
        foam_line = case_styles[case_name]["foam_line"]
        # Get labels for each case from the dictionary-----------------------
        exactSol_label = case_labels[case_name]["exactSol_label"]
        foamSol_label = case_labels[case_name]["foamSol_label"]

        #--Calculate exact solution parameters-------------------------------
        A, z_ex = burgers_exact_solution(delta, nu, exacSolTol, initial_guess)

        if A is not None and z_ex is not None:
            u_values_exact = exact_solution_u(x_values, A, z_ex, nu)
            output_file.write(f"\n{case_name} ('Exact Solution'): nu = {nu}, "
                              f"delta = {delta}, "
                              f"z_ex = {z_ex:.8f} (Transition Layer Location)\n")
            print(f"\n{case_name} ('Exact Solution'): nu = {nu} , delta = {delta},"
                  f" z_ex = {z_ex:.8f} (Transition Layer Location)")
        else:
            print(f"Failed to compute exact solution for {case_name}.")
            output_file.write(f"Failed to compute"
                              f" exact solution for {case_name}.\n")
            continue

        #--Construct post-processing path-----------------------------------
        post_process_dir = os.path.join(script_dir, "..",
                         "stochasticBurgersBCs_Study", "Results",
                        "sampledData", case_name, "postProcessing", "Usample")

        #--Check if the directory exists------------------------------------
        if not os.path.isdir(post_process_dir):
            print(f"Directory not found for {case_name}: {post_process_dir}")
            output_file.write(f"Directory not found for "
                              f"{case_name}: {post_process_dir}\n")
            continue

        #--Locate latest time folder in Usample-----------------------------
        inputFilename = generate_filename("lineX", order, "xy")
        time_dirs = [d for d in os.listdir(post_process_dir)
                     if d.replace(".", "", 1).replace("e-", "", 1).isdigit()]
        if time_dirs:
            latest_time = sorted(time_dirs, key=lambda x: float(x))[-1]

            sample_file = os.path.join(post_process_dir,
                                       latest_time, inputFilename)
```

```
            if os.path.exists(sample_file):
                foam_data = np.loadtxt(sample_file)
                x_foam, U_x_foam = foam_data[:, 0], foam_data[:, 1]

                #--Plot the exact solution and simulation results--------------
                ax1.plot(x_values, u_values_exact,
                         label=f"{exactSol_label} ($\\nu$ = {nu},"
                         f" $\\delta$ = {delta})",
                         linestyle=exact_line, color=exact_color,
                         linewidth=2)

                ax1.plot(x_foam, U_x_foam,
                         label=f"{foamSol_label} ($\\nu$ = {nu},"
                         f" $\\delta$ = {delta})", linestyle=foam_line,
                         color=foam_color, linewidth=1, markeredgewidth=1,
                         markeredgecolor=exact_color)
                ax1.grid(color=grey, linewidth=lineWidth, alpha=0.3)
                ax1.tick_params(which='both', direction='in',
                                length=tickLength, width=lineWidth,
                                pad=tickPad, color=grey)
                ax1.yaxis.set_ticks_position('both')
                ax1.xaxis.set_ticks_position('both')
                ax1.spines['bottom'].set_color(grey)
                ax1.spines['top'].set_color(grey)
                ax1.spines['right'].set_color(grey)
                ax1.spines['left'].set_color(grey)
                #-------------------------------------------------------------
                # Calculate exact_interpolated for the relative error calculation
                interpolation_exact = interp1d(x_values, u_values_exact,
                                               kind='linear')
                U_exact_interpolated = interpolation_exact(x_foam)

                #--Calculate and plot L2 norm and relative error---------------
                L2_norm = np.sqrt(np.sum((U_exact_interpolated - U_x_foam) ** 2))
                relative_error = ((U_x_foam - U_exact_interpolated) / U_exact_interpolated)
    *100
                ax2.plot(x_foam, relative_error,
                         label=f"{foamSol_label} Relative Error. "
                         f"$L_2$-norm error =  {L2_norm:.1e}",
                         linestyle=foam_line, linewidth=1, color=foam_color)
                #--Calculate Transition Layer Location (z_foam)----------------
                zero_crossings = np.where(np.diff(np.sign(U_x_foam)))[0]
                if zero_crossings.size > 0:
                    idx = zero_crossings[0]
                    z_foam = interp1d(U_x_foam[idx:idx + 2],
                                      x_foam[idx:idx + 2], kind='linear')(0)
                    #--Format & output the OpenFOAM solver transition layer data
                    output_text = (f"\n{case_name} ('myGPCBurgersFoam' Simulation):"
                                   f" nu = {nu}, delta = {delta}, "
                                   f"z_foam = {z_foam:.8f} "
                                   "(Transition Layer Location)\n")
                    print(output_text)
                    output_file.write(output_text)  # Write to file
                else:
                    output_text = f"{case_name} ('myGPCBurgersFoam' Simulation):"
                    f"No transition layer (zero-crossing) found.\n"
                    print(output_text)
                    output_file.write(output_text)  # Write to file
            else:
                output_text = f"Sample file {sample_file} not found.\n"
                print(output_text)
                output_file.write(output_text)  # Write to file
        else:
            output_text = f"No time directories found in: {post_process_dir}\n"
            print(output_text)
            output_file.write(output_text)  # Write to file

#--Plotting Data-----------------------------------------------------------
```

```
ax1.axhline(y=0, color='grey', linestyle='-', linewidth=0.5)
ax1.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
ax1.set_ylabel(r"$u(x)$ [m/s]", fontsize=fontsizeAxisLabel)
# ax1.set_suptitle("Burger's Equation", fontsize=14)
# ax1.set_title(r"Stochastic Solver 'myGPCBurgersFoam' Verification", fontsize=10)
ax1.legend(loc='lower left', fontsize=fontSizeLegend)
ax1.grid(False)
# ax1.tight_layout()
fig.tight_layout(pad=boxPad)

#--Configure the subplot (relative error)-------------------------------------
ax2.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
ax2.set_ylabel("Relative Error [$\%$]", fontsize=fontsizeAxisLabel)
# ax2.set_yscale('log')
# ax2.set_title(r"Stochastic Solver 'myGPCBurgersFoam' Relative Error [$\%$]", fontsize=10)
ax2.legend(loc='lower left', fontsize=fontSizeLegend)
ax2.grid(color=grey, linewidth=lineWidth, alpha=0.3)
ax2.grid(False)
ax2.tick_params(which='both', direction='in', length=tickLength, width=lineWidth,
                pad=tickPad, color=grey)
ax2.yaxis.set_ticks_position('both')
ax2.xaxis.set_ticks_position('both')
ax2.spines['bottom'].set_color(grey)
ax2.spines['top'].set_color(grey)
ax2.spines['right'].set_color(grey)
ax2.spines['left'].set_color(grey)

#--Construct Plots path-------------------------------------------------------
plot_dir = os.path.join(script_dir, "..",
                    "stochasticBurgersBCs_Study", "Results", "plots")
os.makedirs(plot_dir, exist_ok=True)  # Ensure directory exists

#--Show and Save Plot---------------------------------------------------------
filename= 'myGPCBurgersFoam_Verification.pdf'
plt.savefig(os.path.join(plot_dir, filename), dpi=600, format='pdf')

plt.show()
```

C.2.4 Stochastic Solver Uncertainty Quantification

Listing C.9: `stochasticSolverUQ_v002.py`

```
import re
import os
import sys
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

#--Get the directory where the script is located-------------------------------
script_dir = os.path.dirname(os.path.abspath(__file__))
preprocessing_dir = os.path.join(script_dir, "..","preProcessingScripts")
sys.path.append(os.path.abspath(preprocessing_dir))

#--Import External Modules----------------------------------------------------
from read_UQProperties_v001 import read_UQProperties
from read_transportProperties_v000 import read_transportProperties
#--Get the directory containing the cases-------------------------------------
study = "stochasticBurgersBCs_Study"
cases_dir = os.path.join(script_dir, "..", study)

#--Function to extract the numeric part from case names and sort them accordingly
def get_case_names(cases_dir):
    # List all directories that start with "Case"
    case_names = [
        name for name in os.listdir(cases_dir)
        if os.path.isdir(os.path.join(cases_dir, name))
        and name.startswith("Case")
    ]

    # Sort case names by extracting the numeric part after "Case_"
    sorted_case_names = sorted(case_names,
                               key=lambda x: int(re.search(r'\d+', x).group()))
    return sorted_case_names

# Automatically get the case names from the folder
cases = get_case_names(cases_dir)

#--UQProperties---------------------------------------------------------------
#--Function to read UQ properties for a given case----------------------------
def get_UQProperties(case_name, cases_dir):
    uqproperties_path = os.path.join(cases_dir, case_name, "constant",
                                     "UQProperties")
    return read_UQProperties(uqproperties_path)

#--transportProperties--------------------------------------------------------
def get_transportProperties(case_name, cases_dir):
    transportProperties_path = os.path.join(cases_dir, case_name, "constant",
                                            "transportProperties")
    return read_transportProperties(transportProperties_path)

#--Accessing the properties for a specific case-------------------------------
#--Case-All-------------------------------------------------------------------
# (here the Case 3 Properties are used for all of the cases)
UQProperties = get_UQProperties(cases[2], cases_dir)
order = UQProperties[0]   # Polynomial order
bounds = UQProperties[3]  # Lower and upper bounds
lower = bounds ['lower']
upper = bounds ['upper']
#--transportProperties
nu_value = get_transportProperties(cases[0], cases_dir)

#--Define parameters in this main script--------------------------------------
nu = nu_value
delta_lower = lower  # unperturbed case
delta_upper = upper   # perturbed case
```

```
order = order

#--Setup cases with delta and initial guess-----------------------------------
cases = {
    "Case_1_stocLBBCs_Verification":
        {"delta": delta_lower},
    "Case_2_stocUBBCs_Verification":
        {"delta": delta_upper},
    "Case_3_stocBCs_UQ":
        {"delta": (delta_lower, delta_upper)}
}

#--Plot Configuration---------------------------------------------------------
fontSize = 10
fontSizeLegend = 11
fontsizeAxisLabel = 11
lineWidth = 1.0
tickPad = 5
labelPadY = 4
labelPadX = 4
boxPad = 2
tickLength = 2
markerSize = 4

#--HTML code for colours------------------------------------------------------
black = '#000000'
grey = '#555555'
greyLight = '#B0B0B0'
blue = '#011231'
blueLight = '#cddefd'
red = '#E50037'
redLight = '#FCC8CF'

#--Enable LaTeX rendering in Matplotlib for consistency-----------------------
plt.rcParams['text.usetex'] = True
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['Computer Modern Roman']  # LaTeX's default font
plt.rcParams['axes.linewidth'] = lineWidth
plt.rcParams['mathtext.fontset'] = 'cm'

#--Create a figure with controlled aspect ratio-------------------------------
figUQ = plt.figure(figsize=(12, 5))  # Adjust width and height to control aspect ratio
axUQ = figUQ.add_subplot(111)
# Create a new figure for the mode contributions and a subplot for the percentage
    contribution of each mode
figModes, (axModes) = plt.subplots(1, 1, figsize=(12, 5), gridspec_kw={'height_ratios':
    [1]})

#--Get the directory where the script is located------------------------------
script_dir = os.path.dirname(os.path.abspath(__file__))

#--Prepare output directory for results data----------------------------------
results_data_dir = os.path.join(script_dir, "..",
                                "stochasticBurgersBCs_Study", "Results", "resultsData")
os.makedirs(results_data_dir, exist_ok=True)  # Ensure directory exists
output_file_path = os.path.join(results_data_dir, f"stochasticBurgers_BCs_Study_UQ_M{order}.
    txt")

#--Prepare input file name----------------------------------------------------
def generate_filename(base_name, order, extension):
    # Create the filename dynamically based on the polynomial order
    variable_parts = [f"Uhat{i}" for i in range(order + 1)]
    filename = f"{base_name}_" + "_".join(variable_parts) + f".{extension}"
    return filename
inputFilename = generate_filename("lineX", order, "xy")

#--Open the output file in write mode-----------------------------------------
with open(output_file_path, "w") as output_file:
```

```
    #--Define colour and linestyle dictionaries for each case------------------
    case_styles = {
        "Case_1_stocLBBCs_Verification": {"foam_color": black, "foam_line": ":"},
        "Case_2_stocUBBCs_Verification": {"foam_color": black, "foam_line": "--"},
        "Case_3_stocBCs_UQ": {"foam_color": black, "foam_line": "-"}
    }

    case_labels = {
        "Case_1_stocLBBCs_Verification": {"foamSol_label": "Lower Bound"},
        "Case_2_stocUBBCs_Verification": {"foamSol_label": "Upper Bound"},
        "Case_3_stocBCs_UQ": {"foamSol_label": "$\mu$ Mean Solution"}
    }

    #--Loop through each case--------------------------------------------------
    for case_name, params in cases.items():
        delta = params["delta"]
        # ---------------------------------------------------------------------
        # Get colors and linestyles for each case from the dictionaries
        foam_color = case_styles[case_name]["foam_color"]
        foam_line = case_styles[case_name]["foam_line"]
        foamSol_label = case_labels[case_name]["foamSol_label"]

        #--Construct post-processing path-------------------------------------
        post_process_dir = os.path.join(script_dir, "..",
                        "stochasticBurgersBCs_Study", "Results",
                        "sampledData", case_name, "postProcessing", "Usample")

        #--Check if the directory exists--------------------------------------
        if not os.path.isdir(post_process_dir):
            print(f"Directory not found for {case_name}: {post_process_dir}")
            output_file.write(f"Directory not found for {case_name}: {post_process_dir}\n")
            continue

        #--Locate latest time folder in Usample-------------------------------
        time_dirs = [d for d in os.listdir(post_process_dir)
                     if d.replace(".", "", 1).replace("e-", "", 1).isdigit()]
        if time_dirs:
            latest_time = sorted(time_dirs, key=lambda x: float(x))[-1]

            sample_file = os.path.join(post_process_dir, latest_time, inputFilename)
            if os.path.exists(sample_file):
                foam_data = np.loadtxt(sample_file)

                # Mean Value extraction from Uhat0-----------------------------
                x_foam, U_x_foam = foam_data[:, 0], foam_data[:, 1]
                Uhats_x = foam_data[:, 4::3] # Assuming Uhat coefficients are in columns [1,
     4, 7,...] for x-components only

                # Variance (sigma^2)-------------------------------------------
                # Square each mode to get the variance contribution along the spatial domain
                U_x_modeVar_foam = Uhats_x**2  # Shape: (number of spatial points, number of
     modes)

                # Total Variance (Sum the variances along the spatial domain)
                U_x_totalVar_foam = np.sum(U_x_modeVar_foam, axis=1)

                # Standard Deviation (sigma)-----------------------------------
                # Calculate the standard deviation based on higher-order terms
                U_x_sd_foam = np.sqrt(U_x_totalVar_foam)  # Sum squares of higher-order
    terms only

                # Calculate the uncentainty based on standard deviation--------
                k = 1 #k coverage factor ( k = 2 for 95% CI but this required normality
    assumption, this need further analysis)
                U_x_Unc = k * U_x_sd_foam

                #--Plot the mean solution 'U_x_foam'--------------------------
                if case_name != "Case_3_stocBCs_UQ":
```

```
                axUQ.plot(x_foam, U_x_foam, label=f"{foamSol_label} ($\\nu$ = {nu}, $\\
delta$ = {delta})",
                      linestyle=foam_line, color=foam_color, linewidth=1)
                # axUQ.grid(color=grey, linewidth=lineWidth, alpha=0.3)
                axUQ.tick_params(which='both', direction='in', length=tickLength, width=
lineWidth,  pad=tickPad, color=grey)
                axUQ.yaxis.set_ticks_position('both')
                axUQ.xaxis.set_ticks_position('both')
                axUQ.spines['bottom'].set_color(grey)
                axUQ.spines['top'].set_color(grey)
                axUQ.spines['right'].set_color(grey)
                axUQ.spines['left'].set_color(grey)
            else:
            #--Plot the mean solution 'U_x_foam'---------------------------
                axUQ.plot(x_foam, U_x_foam, label=f"{foamSol_label} ($\\nu$ = {nu}, $\\
delta$ $\\sim$ U{delta})",
                linestyle=foam_line, color=foam_color, linewidth=lineWidth)
            # Additional plot for standard deviation in 'Case_3_stocBCs_UQ'
                axUQ.plot(x_foam, U_x_sd_foam, label=r"$\sigma$" f" Standard Deviation (
$\\nu$ = {nu}, $\\delta$ $\\sim$ U{delta})", linestyle="-.", color=foam_color,
linewidth=1)
            # Plot mean solution +- standard deviation as a shaded region
                axUQ.fill_between(x_foam, U_x_foam - U_x_Unc, U_x_foam + U_x_Unc, color=
foam_color, alpha=0.2, label="$\mu$ $\pm$ $\sigma$")

            #-------------------------------------------------------------
            #--Calculate maximum absolute value for U_x_foam (Uhat_0)------
                max_Uhat0 = np.max(np.abs(U_x_foam))

                # Calculate maximum absolute values for each higher mode in Uhats_x
                max_values = np.max(np.abs(Uhats_x), axis=0)

                # Define scaling factors for each higher mode relative to Uhat_0
                scaling_factors = max_Uhat0 / max_values  # Scaling all higher modes
based on Uhat_0
                # scaling_factors = [1, 1, 1]
                # Plot Uhat_0 without scaling
                axModes.plot(x_foam, U_x_foam, label=r"$\hat{u}_0(x)$", linestyle='-',
color='black', linewidth=lineWidth)

                # Plot each higher mode with its calculated scaling factor
                # Define a list of linestyles for each mode
                linestyles = ['--', '-.', ':', (0, (3, 1, 1, 1)), (0, (5, 5))]  # Add
more if you have more modes
                # Plot each higher mode with its calculated scaling factor and different
 linestyles, all in black
                for mode_index in range(Uhats_x.shape[1]):  # Iterate through each
higher mode
                    mode_contribution = scaling_factors[mode_index] * Uhats_x[:,
mode_index]
                    axModes.plot(x_foam, mode_contribution, linewidth=lineWidth,
                                 label=f"{scaling_factors[mode_index]:.1f} " r"$\hat{u}_"
 f"{mode_index + 1}(x)$",
                                 color='black', linestyle=linestyles[mode_index % len(
linestyles)])  # Cycle through linestyles

                # Customize the plot appearance
                axModes.grid(color=grey, linewidth=lineWidth, alpha=0.3)
                axModes.tick_params(which='both', direction='in', length=tickLength,
width=lineWidth, pad=tickPad, color=grey)
                axModes.yaxis.set_ticks_position('both')
                axModes.xaxis.set_ticks_position('both')
                axModes.spines['bottom'].set_color(grey)
                axModes.spines['top'].set_color(grey)
                axModes.spines['right'].set_color(grey)
                axModes.spines['left'].set_color(grey)

            #-------------------------------------------------------------
```

```
                #--Calculate transition layer location using zero crossing-----
                zero_crossings = np.where(np.diff(np.sign(U_x_foam)))[0]
                if zero_crossings.size > 0:
                    idx = zero_crossings[0]
                    z_foam = interp1d(U_x_foam[idx:idx + 2], x_foam[idx:idx + 2], kind='
    linear')(0)

                    # Calculate the standard deviation at the transition layer location
                    sd_interpolation = interp1d(x_foam, U_x_sd_foam, kind='linear')
                    sd_at_transition = sd_interpolation(z_foam)

                    # Write transition location and standard deviation to file
                    output_text = (f"\n{case_name} ('myGPCBurgersFoam' Simulation):"
                                   f" nu = {nu}, delta = {delta}, "
                                   f"z_foam = {z_foam:.8f}, "
                                   f"SD at Transition Layer = {sd_at_transition:.8f}\n")
                    print(output_text)
                    output_file.write(output_text)  # Write to file
                else:
                    output_text = f"{case_name} ('myGPCBurgersFoam' Simulation): No
    transition layer (zero-crossing) found.\n"
                    print(output_text)
                    output_file.write(output_text)  # Write to file
            else:
                output_text = f"Sample file {sample_file} not found.\n"
                print(output_text)
                output_file.write(output_text)  # Write to file
        else:
            output_text = f"No time directories found in: {post_process_dir}\n"
            print(output_text)
            output_file.write(output_text)  # Write to file

#--Plotting Data-------------------------------------------------------------
#--UQ-----------------------------------------------------------------------
axUQ.axhline(y=0, color='grey', linestyle='-', linewidth=0.5)
axUQ.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
axUQ.set_ylabel(r"$u(x)$ [m/s]", fontsize=fontsizeAxisLabel)
# axUQ.set_title(r"Stochastic Solver 'myGPCBurgersFoam' Solution (Legendre-Chaos Expansion)
    ", fontsize=10)
axUQ.legend(loc='lower left', fontsize=fontSizeLegend)
# Include the polynomial order in the legend
axUQ.legend(title=f"Polynomial Order M = {order}")
axUQ.grid(False)

#--Modes--------------------------------------------------------------------
axModes.set_xlabel("x [m]", fontsize=fontsizeAxisLabel)
axModes.set_ylabel(r"$\hat{u}_k$ [m/s]", fontsize=fontsizeAxisLabel)
# axModes.set_title(r"Mode Contributions in Stochastic Solution (Scaled Relative to $\hat{U}
    _0(x)$)", fontsize=10)
axModes.axhline(y=0, color='grey', linestyle='-', linewidth=0.5)
axModes.legend(loc='lower left', fontsize=fontSizeLegend)
# Include the polynomial order in the legend
axModes.legend(title=f"Polynomial Order M = {order}")
axModes.grid(False)

#--Construct Plots path-----------------------------------------------------
plot_dir = os.path.join(script_dir, "..",
                        "stochasticBurgersBCs_Study", "Results", "plots")
os.makedirs(plot_dir, exist_ok=True)  # Ensure directory exists

#--Show and Save Plot-------------------------------------------------------
#--UQ-----------------------------------------------------------------------
filename = f'myGPCBurgersFoam_UQ_M{order}.pdf'
figUQ.savefig(os.path.join(plot_dir, filename), dpi=600,format='pdf')
plt.show()
#--Modes--------------------------------------------------------------------
filename = f'myGPCBurgersFoam_ModeContributionsScaled_M{order}.pdf'
```

```
figModes.savefig(os.path.join(plot_dir, filename), dpi=600,format='pdf')
plt.show()
```