Study in Modern Uncertainty Quantification Methods

Master Thesis

Emil Kjær Nielsen

Technical University of Denmark
Department of Applied Mathematics and Computer Science
DK-2800 Lyngby

Technical University of Denmark

DTU Compute
Department of Applied Mathematics and Computer Science
Asmussens Alle, bygning 303B
DK-2800 Lyngby
http://compute.dtu.dk

Title:
Study in Modern Uncertainty Quantification Methods

Theme:
Uncertainty Quantification Methods

Project Period:
December 2012 - May 2013

Participant:
Emil Kjær Nielsen

Supervisors:
Allan Peter Engsig-Karup

Copies: 3

Page Numbers: 132

Date of Completion:
May 3, 2013

Abstract:

Uncertainty Quantification (UQ) is a relatively new research area where there over the past years have been an ongoing development in techniques to improve the existing UQ methods. As the demand on quantifying uncertainties are increasing the methods becomes more widely used.
The goal with the thesis is to apply UQ using generalized Polynomial Chaos (gPC) expansion together with spectral numerical methods on differential equation. Furthermore experiences with the programming language `Python` must be gained in order to implement the UQ methods.
The thesis starts by introducing the mathematical background of the spectral method including e.g. orthogonal polynomials and quadrature rules. Three differently UQ methods is, after the introduction of gPC, presented.
To illustrate uncertainty, the UQ method are applied on two stochastic differential equations showing the beneficial by using spectral methods illustrated by the spectral convergence.
The final part dealing with the illustration of the *curse of dimensionality.* It also contains 1 technique handling the dimensionality which is satisfied to some extend.

Danmarks Tekniske Universitet

DTU Compute
Institut Matematik og Computer Science
Asmussens Alle, bygning 303B
DK-2800 Lyngby
http://compute.dtu.dk

Titel:
Studie i moderne metoder til kvantificering af usikkerhed

Tema:
Metoder til kvantificering af usikkerhed

Projektperiode:
December 2012 - Maj 2013

Deltager:
Emil Kjær Nielsen

Vejledere:
Allan Peter Engsig-Karup

Oplagstal: 3

Sidetal: 132

Afleveringsdato:
3. maj, 2013

Abstract:

Uncertainty Quantification er et forholdvist nyt forskningsområde, hvor der over de seneste år er blevet udviklet teknikker til at forbedre de existerende UQ metoder. Siden flere og flere stiller krav om kvantificering af usikkerheder metoderne bliver mere og mere udbredte.
Målet med afhandlingen er at anvende UQ (kvantificering af usikkerhed) (UQ) ved at bruge generalized Polynomial Chaos (gPC) ekspansion sammen med spektrale numeriske metoder på differential ligninger. Endvidere erfaringer i programmeringssproget `Python` skal opnås sådan at UQ metoderne kan implementeres.
Afhandlingen starter med en introduktion af den bagvedliggende matematiske teori for de spektrale metoder indeholdende f.eks. ortogonale polynomier og kvadratur regler. Tre forskellige UQ metoder vil, efter en introduktion af gPC, blive præsenteret.
For at illustrere usikkerhederne, UQ metoderne er afprøvet på to stokastiske differential ligninger, der viser fordelene ved at bruge de spektrale metoder illustreret ved den spektrale konvergence.
Den sidste del beskæftiger sig med at illustrere *curse of dimensionality*. Dette indeholder en teknik der håndtere denne dimensionalitet, som viser at denne teknik er tilstrækkelig til en vis grad.

Contents

Preface

This thesis is prepared as a master thesis at the department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU) and supervised by Associate professor Allan Peter Engsig-Karup. The thesis is conducted in the period from December the 3rd 2012 to May the 3rd 2013 by Emil Kjær Niesen.

The thesis is an introduction to the area Uncertaincy Quantification and will require knowledges to numerical method for differential equations. The applied programming language is `Python`, but basic knowledge to some programming language might be sufficient to understand the implementations. The thesis contains theory of generalized Polynomial Chaos, orthogonal polynomials and basic probability theory. All simulations and test are performed on a computer with 8.00 GB RAM with a Intel Core i7 processor with 64 bit Windows 8.

I will like to thank my supervisor Allan Peter Engsig-Karup for introducing me to the area and supervision when needed. Also I like to thank Ph.D. Daniele Bigoni for helping with the practical questions I had throughout the thesis. A special thank to fellow student Emil Brandt Kærgaard for the constructive discussions in many fields we had throughout the period. Johan Kjær Nielsen and Kristian Rye Jensen also have to be thanked for the critical eyes and feedback on my written report.

Technical University of Denmark, May 3, 2013

Emil Kjær Nielsen
<s072248@student.dtu.dk>

Chapter 1

Introduction

Solving differential equation by use of numerical models is continuously in developing and is used in many engineering and science areas. The development focusing on reducing the time consumption and the precision of the solutions. Today the numerical tools are extremely important as simulations can reduce the need for costly physical test in for example design processes [1].

However, numerical simulations must be handled carefully as assumptions and approximations leading to error which are unavoidable. The errors introduced between the numerical simulations and 'the real world' can be classified into 3 main groups [1].

Model error

The approximation of the physics in the real world is done by mathematical models which contains the physical laws and principles. However, the mathematical models are often simplified in order to make the simulations possible or easier. Hence some physical laws are disregarded as its effects are assumed to be negligible. Therefore the mathematical models often will not contain all aspects from the physics and cannot exactly replicate the behaviour in 'the real world'.

Numerical errors

Using numerical methods to solve mathematical models will introduce numerical errors because of the finite representation of numbers on computers. This involves both truncation errors (e.g. truncation of an infinite sum) and rounding errors (e.g. rounding π to 16 digits). Some numerical methods ensures small numerical errors by taking advantage of convergence and stability, but numerical errors will practically always exist.

Data errors

To find a solution to a given mathematical model, the parameters and data related to the model have to be known. This could for example be a temperature or velocity expressed as model constants, boundary conditions or initial condition. Measurements or identifications are used to determine these data and are very often flawed. These errors are called data errors.

This thesis will focus on the influence data errors have on the solutions for mathematical models in terms of differential equations. The uncertainty in the data will be represented by random variables and hence it all boils down to solving stochastic differential equations quantifying the uncertainty introduced by the data by numerical methods. Uncertainty Quantification (UQ) is a relatively new research area and gets more and more attention, as it is demanded that solutions for the differential equations contains uncertainty.

In figure 1.1 an illustrative example is shown. It shows two deterministic solutions for the viscous Burger's equation (introduced later on) for two different boundary conditions. One boundary condition is $u_1(t,-1) = 1$ while the other is $u_2(t,-1) = 1.01$.

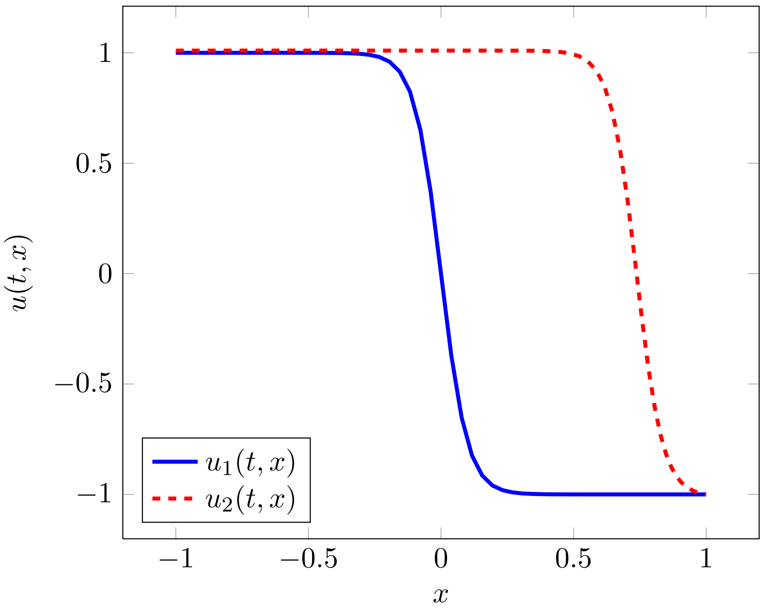

Figure 1.1: Deterministic solutions of viscous Burgers equation for boundary condtions $u_1(t,-1) = 1$ and $u_2(t,-1) = 1.01$.

The solutions in figure 1.1 show that a very small change in the boundary condition can have a huge impact on the solution. Assume that the boundary condition is found by measurements and follows a normal distribution $\mathcal{N}(1, 0.5)$ (from $\mathcal{N}(\mu, \sigma^2)$). Then two different measurements can give two very different solutions as illustrated in figure 1.1.

1.1 Objectives and goals

As UQ is a relatively new area and will likely be a widely used area in the future it will be advantageously to get knowledge to UQ method, but also to get knowledges to the theoretical background and techniques.

The overall goal with the project is the achieve knowledge to UQ area. To obtain this goal the theoretical background and numerical tools have to be investigated and understand. Since this thesis is an introduction to UQ the focus area will be to illustrate UQ methods, but also to validate and compare them. It is chosen only to use relatively simple test problem, such that a technique handling the *curse of dimensionality* can be treated. The thesis should be presented such that it is possible to imagine how it can be applied on more complex systems. The thesis is also limited only to handle systems with random variables and not with random processes. Other objectives are to illustrate the issues in form of huge systems which have led to different techniques to handle them. Finally the thesis ends with an illustration one technique that reduces the computational work.

A secondary goal of this thesis is to get knowledge and experiences to the programming language `Python`. Previously in my studies the programming language `Matlab` has been used, but due to experiences with time consuming calculations it was temping to try out `Python`. Another reason to use `Python` instead of `Matlab` is that `Python` is an open source product.

1.2 Outlines of the thesis

The thesis is mainly based on [2], which is an introduction to UQ. Many of the numerical tools used throughout the thesis are inspired by [3] and [4].

In the thesis chapter 2 starts covering the notation and expressions used throughout the thesis and a short introduction to `Python` will be given. Chapter 3 will move on with the mathematical background and numerical tools needed by the UQ methods. This lead to chapter 4 where three UQ methods are presented and shortly discussed. Chapter 5 introduced the two test problems where some exact expressions are computed for the simplest problem.

This is followed by a test chapter (chapter 6) where the three methods are tested on the two problems (containing 1 random variable) and finally the methods are discussed. Hereafter today's topics within UQ are shortly presented. In chapter 8 multidimensional cases are addressed and hereby the *curse of dimensionality* is illustrate. One of the method presented in chapter 7 will lastly be illustrated on a multidimensional case.

Chapter 2

Notation and expressions

Before going into details with the methods and the underlying tools the used notation and expressions will be clarified. This will be recurrent throughout the project.

First the general formulation of a Stochastic Differential Equation (SDE) where the goal is to find the solution $u(t, \boldsymbol{x}, \boldsymbol{Z})$. A SDE is a Partial Differential Equation (PDE) added some uncertainty coming from the measured data. The system is defined on a spatial domain $S \subset \mathbb{R}^\ell, \ell = 1, 2, 3, \dots$ and a time domain $t \subset [0, T]$. The formulation for a SDE with solution $u(\boldsymbol{x}, t,)$ is given by

$$\begin{aligned} &u_t(\boldsymbol{x}, t, \boldsymbol{Z}) = \mathcal{L}(u), && S \times (0, T] \times \mathbb{R}^d \\ &\mathcal{B}(u) = 0, && \partial S \times [0, T] \times \mathbb{R}^d \\ &u(\boldsymbol{x}, 0, \boldsymbol{Z}) = \boldsymbol{u}_0, && S \times \mathbb{R}^d \end{aligned} \tag{2.1}$$

where $\mathcal{L}$ is a differential operator, $\mathcal{B}$ is the boundary condition operator, $\boldsymbol{u}_0$ is the initial condition and $\boldsymbol{Z} = (Z_1, Z_2, \dots, Z_d) \in \mathbb{R}^d$, $d \geq 1$ is a set of independent random variables [2]. The most simple SDE is the case where $d = 1$ - a SDE having 1 random variable, which will be the main case in the first part of the thesis.

A (deterministic) solution to the SDE system will be denoted $u(t, \boldsymbol{x}, \boldsymbol{Z})$. The exact mean and variance solution are designated with μ_u and σ_u^2 respectively (and hereby the standard deviation is denoted σ_u). The corresponding estimated mean and variance solution will have the notation $\bar{u}$ and s_u^2.

In order to describe the uncertainty in the SDE probability theory is needed. The uncertainty will be described by the aforementioned random variable $Z = Z(z)$ where z belongs to the outcome space Ω. E.g. by tossing a coin the outcome space will be $\Omega = \{\text{head,tail}\}$ and hereby the random variable becomes Z belongs to $\{0, 1\}$. In this way the SDE can contain uncertainty on the measured parameters. However, the focus in this thesis will only be on continuously random variables and not on discrete discrete variables as in the coin example.

The random variables will follow a certain distribution with a corresponding probability $P(Z = z)$ which takes values between 0 and 1, where 1 means that the outcome z will always happen. If the random variable is continuous then $P(Z = z) = 0$ for $z \in \Omega$.

Instead the probability can be described by the cumulative distribution function (CDF) which is given by (from [2])

$$P(Z \leq a) = F_Z(a) = \int_{-\infty}^{a} f(z)\, dz,$$

where $f(z)$ is the probability density function (PDF) corresponding to the given distribution and has the condition

$$\int_{-\infty}^{\infty} f(z)\, dz = 1.$$

The PDF's that will be used throughout this thesis are those belonging to the Normal distribution and to the Uniform distribution. For a random variable Z following $\mathcal{N}(\mu, \sigma^2)$ the PDF is

$$f(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(z-\mu)^2}{2\sigma^2}},$$

and for the random variable following $\mathcal{U}(a, b)$ the PDF is

$$f(z) = \begin{cases} \frac{1}{b-a}, & z \in [a, b] \\ 0, & \text{otherwise} \end{cases}.$$

Another important property in probability theory that will be used here is the moment of the continuous random variable. The m'th moment is given by

$$\mathbb{E}[Z^m] = \int_{-\infty}^{\infty} z^m f(z)\, dz$$

for $m \in \mathbb{N}$ [2]. The first moment $m = 1$ is the formula for the expected value

$$\mu_Z = \mathbb{E}[Z] = \int_{-\infty}^{\infty} z f(z)\, dz.$$

The variance of the random variable Z is defined by

$$\sigma_Z^2 = \int_{-\infty}^{\infty} (z - \mu_Z)^2 f(z)\, dz.$$

Through the thesis the expectation will be determined from a function of a random variable $g(Z)$ and here the expectation is found by [2] to be

$$\mathbb{E}[g(Z)] = \int_{-\infty}^{\infty} g(z) f(z)\, dz.$$

The weighted inner product notation between two functions $u(Z)$ and $v(Z)$ is defined as

$$\langle u, v \rangle_w = \mathbb{E}[u(Z)v(Z)] = \int_{-\infty}^{\infty} u(z)v(z)w(z)\, dz. \tag{2.2}$$

All these recurrent expressions will be widely used throughout the thesis. With the basic notation and expressions introduced the mathematical background and some numerical tools, which makes the basis for the UQ methods, will be explained in the next chapter. First an introduction to the used programming language `Python`.

2.1 Programming language: `Python`

All implementations and test will be conducted in the language `Python` which is a open source programming language. This is the first time the language is used by the author, but since `Python` supports a list of packages which mimic the procedures in `Matlab` where the author has great experiences.

`Python` is language where the user do not need to handling memory management and the choice of variable types. Therefore the attention remains on programming the mathematics. The two packages mainly used is `NumPy` and `SciPy`. The `NumPy` package is for scientific computing and it provides a multidimensional arrays. It includes a large number of the same functions declared in `Matlab`. This package makes it a lot easier to used when coming from `Matlab`. The `SciPy` package contains scientific tools depending on `NumPy`. For example it contains a time integration solver presented in the next chapter.

Chapter 3

Mathematical background and numerical tools

Before introducing the specific UQ methods, the mathematical background and the needed numerical tools have to be stated. A very important ingredient are the orthogonal polynomials which will be examined first. Afterwards the corresponding quadrature rules for these polynomials will be introduced. Further on the used time-integration methods will be explained and finally the generalized Polynomial Chaos expansion is examined.

3.1 Orthogonal Polynomials

To explain orthogonal polynomials, the starting point will be a general polynomial of degree n given by

$$Q_n(x) = k_n x^n + k_{n-1}x^{n-1} + \cdots + k_1 x + k_0 \tag{3.1}$$

where $k_i, i = 0, 1, \ldots, n$ are the coefficients. The polynomial can be written into a monic form which is defined to be the polynomial where the coefficient in front of the leading term x^n equals 1. The polynomial $Q_n(x)$ can be rewritten into monic form as

$$P_n(x) = \frac{Q_n(x)}{k_n} = x^n + \frac{k_{n-1}}{k_n}x^{n-1} + \cdots + \frac{k_1}{k_n}x + \frac{k_0}{k_n}.$$

An orthogonal polynomial is defined as a sequence of polynomials where any pair of polynomials are orthogonal under some inner product [2]. By using the weighted inner product (2.2), this can be expressed by

$$\langle Q_m, Q_n\rangle_w = \int_{-\infty}^{\infty} Q_m(x)Q_n(x)w(x)\, dx = \gamma_n\delta_{mn}, \quad m, n \in \mathbb{N}$$

where

$$\delta_{mn} = \begin{cases} 0, & m \neq n \\ 1, & m = n, \end{cases}$$

known as the Kronecker delta function. The constant γ_n, called the normalizing constant, is given by

$$\gamma_n = \langle Q_n, Q_n \rangle_w.$$

A sequence of orthogonal polynomials will satisfy a three-term recurrence relation for $x \in \mathbb{R}$. In general the three-term recurrence is on the form

$$Q_{n+1}(x) = (A_n x + B_n)Q_n(x) - C_n Q_{n-1}(x), \quad n = 0, 1, \dots,$$

with $Q_0(x) = 0$, $Q_1(x) = 1$ [2]. Further it must be respected that $A_n \neq 0$, $C_n \neq 0$ and $C_n A_n A_{n-1} > 0$ for all n [2]. In monic form, the three term recurrence is on the following form (from [5]).

$$\varphi_{n+1}(x) = (x + a_n)\varphi_n(x) - b_n \varphi_{n-1}(x), \quad n = 0, 1, \dots, \tag{3.2}$$

where

$$a_n = \frac{\langle x\varphi_n, \varphi_n \rangle}{\langle \varphi_n, \varphi_n \rangle}, \quad b_n = \frac{\langle \varphi_n, \varphi_n \rangle}{\langle \varphi_{n-1}, \varphi_{n-1} \rangle}$$

The three-term recurrence in monic form is used for the quadrature rule presented later on.

3.1.1 Examples of orthogonal polynomials

Two types of orthogonal polynomials are the Legendre polynomials and the Hermite polynomials. These will shortly be presented below.

Legendre polynomials

The Legendre polynomials $L_n(x)$ are defined on the interval $x \in [-1, 1]$ and from [2] the orthogonality relation is

$$\langle L_n, L_m \rangle_w = \int_{-1}^{1} L_n(x) L_m(x)\, dx = \frac{2}{2n+1}\delta_{mn},$$

where the weight is $w(x)$ equals 1. The corresponding three-term recurrence relation is on the form

$$L_{n+1}(x) = \frac{2n+1}{n+1} x L_n(x) - \frac{n}{n+1} L_{n-1}(x), \qquad n = 1, 2, \dots, \tag{3.3}$$

where it is defined that $L_{-1}(x) = 0$ and $L_0(x) = 1$. The implementation of the Legendre polynomials is shown in appendix B.1.1. Using this the first five Legendre polynomials are determined to be

$$\begin{aligned}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= \tfrac{3}{2}x^2 - \tfrac{1}{2} \\
L_3(x) &= \tfrac{5}{2}x^3 - \tfrac{3}{2}x \\
L_4(x) &= \tfrac{35}{8}x^4 - \tfrac{30}{8}x^2 + \tfrac{3}{8}
\end{aligned}$$

In figure 3.1 these 5 polynomials are shown.

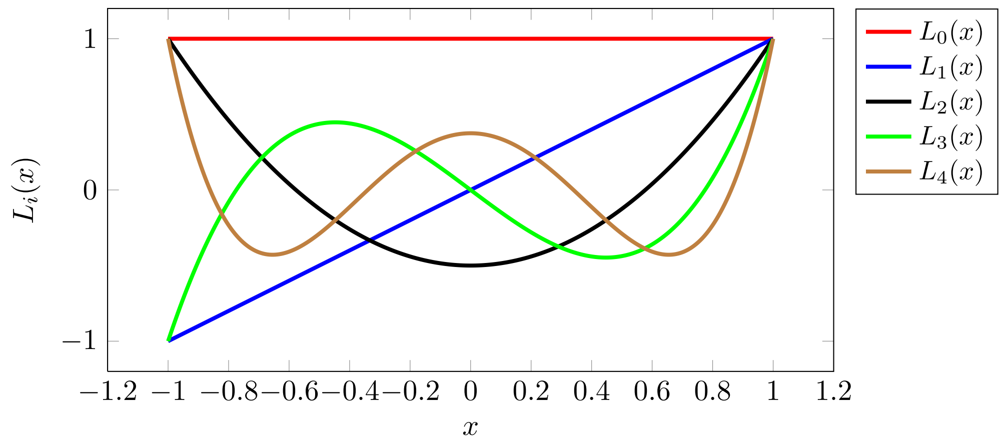

Figure 3.1: First 5 Legendre polynomial

Hermite polynomials

The other type of orthogonal polynomials that will be used are the Hermite polynomials $H_n(x)$ which are defined on $\mathbb{R}$. The orthogonal relation is given by

$$\langle H_n, H_m \rangle_w = \int_{-\infty}^{\infty} H_n(x) H_m(x) w(x) \, dx = n! \delta_{mn}.$$

where the weight is $w(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ [2]. The three-term recurrence for these polynomials takes the form

$$H_{n+1}(x) = xH_n(x) - nH_{n-1}(x), \qquad n = 0, 1, \ldots.$$

where it is defined that $H_{-1}(x) = 0$ and $H_0(x) = 1$. The implementation of the Hermite polynomials is shown in appendix B.1.2. The first five Hermite polynomials are then

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= x \\ H_2(x) &= x^2 - 1 \\ H_3(x) &= x^3 - 3x \\ H_4(x) &= x^4 - 6x^2 + 3, \end{aligned}$$

and these are shown in figure 3.2 on the interval $x \in [-3, 3]$.

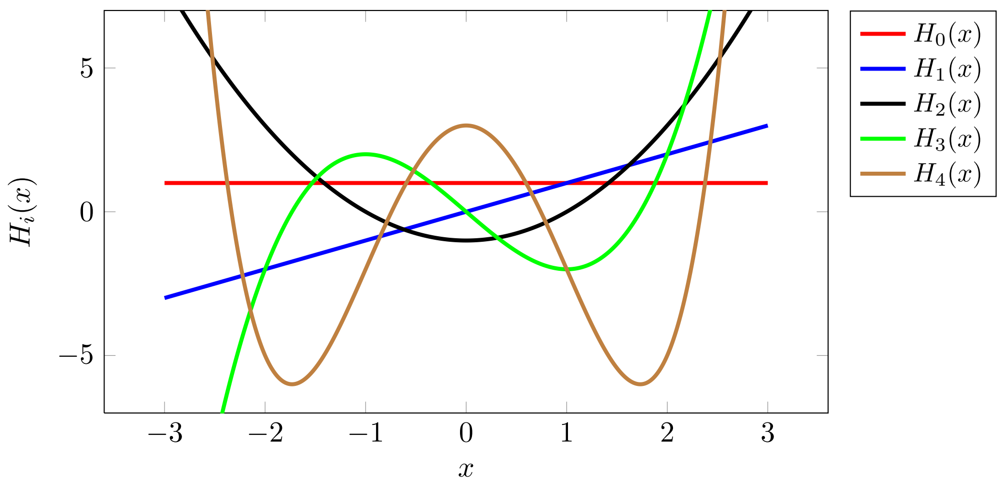

Figure 3.2: First 5 Hermite polynomial

3.2 Quadrature rules

An important tool of the UQ methods is the quadrature rule. It is used for approximating an integral over a given domain and in general the approximation is defined by

$$\int f(x)\,dx \approx \sum_{k=1}^{n} w_k f(x_k), \tag{3.4}$$

where x_k is the nodes (or abscissas) and w_k is the corresponding weights [5]. There exist methods to compute these and one of those methods will be presented after some preliminaries. First the representation of a function will be discussed.

A function $f(x) \in \mathbb{R}$ can be approximated with a polynomial $Q(x)$ times some known weight function $w(x)$

$$f(x) \approx w(x)Q(x), \tag{3.5}$$

from [5]. For the orthogonal polynomials just presented the corresponding weight functions $w(x)$ are known, so these polynomials are possible to use for the approximation of a continuous function. By applying this on (3.5) the approximation can be written by (from [5])

$$\int f(x)\,dx \approx \int w(x)Q(x)\,dx \approx \sum_{k=1}^{n} w_k Q(x_k).$$

In the following the method used to determine the abscissas x_k and weights w_k are presented by a theorem from [5], where the relation between $w(x)$ and w_k are also stated.

Theorem 1. *The w_k and x_k can be obtained from the eigenvalue decomposition of the symmetric, tridiagonal Jacobi matrix*

$$J_n = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \sqrt{b_2} & \ddots & \ddots & \\ & & \ddots & a_{n-2} & \sqrt{b_{n-1}} \\ & & & \sqrt{b_{n-1}} & a_{n-1} \end{bmatrix}$$

where the a_i and b_i for $i = 0, 1, \ldots, n-1$ are as in the three-term recurrence in (3.2)*. If $V^T J_n V = \Lambda = diag(\lambda_1, \ldots, \lambda_n)$, where $V^T V = I$ is the $n \times n$ identity matrix, then $x_k = \lambda_k$ and $w_k = v_{k,0}^2 \int_{-\infty}^{\infty} w(x)\, dx$, where v_k is the kth column of V and $v_{k,0}$ is the first component of v_k.*

In the theorem λ_k is the kth eigenvalue and v_k is the corresponding eigenvector. Below the theorem is used to determine the abscissas and weights for the Legendre polynomials and Hermite polynomials, respectively.

Legendre-Gauss quadrature

The starting point for finding the abscissas and weights is in the three-term recurrence for the Legendre polynomials (3.3)

$$L_{n+1}(x) = \frac{2n+1}{n+1} x L_n(x) - \frac{n}{n+1} L_{n-1}(x) \qquad n > 0. \tag{3.6}$$

This recurrence is required to be on the three-term recurrence monic form if Theorem 1 can be used. Therefore the coefficient in front of the leading term for Legendre polynomials has to be found. From the 5 first Legendre polynomials listed earlier the leading coefficients are $1, 1, \frac{3}{2}, \frac{5}{2}, \frac{35}{8}$ which fit with the general leading coefficient $\frac{(2n)!}{2^n (n!)^2}$ [5]. By dividing through with this coefficient in (3.6) the monic form is obtained to be the following

$$L_{n+1}(x) \frac{2^{n+1}((n+1)!)^2}{(2(n+1))!} = \frac{2n+1}{n+1} x L_n(x) \frac{2^{n+1}((n+1)!)^2}{(2(n+1))!} - \frac{n}{n+1} L_{n-1}(x) \frac{2^{n+1}((n+1)!)^2}{(2(n+1))!} \Leftrightarrow$$

$$\varphi_{n+1}(x) = \frac{2n+1}{n+1} x L_n(x) \frac{2 \cdot 2^n ((n+1)n!)^2}{(2n+2)!} - \frac{n}{n+1} L_{n-1}(x) \frac{4 \cdot 2^{n-1}(((n+1)n)(n-1)!)^2}{(2n+2))!} \Leftrightarrow$$

$$\varphi_{n+1}(x) = \frac{2n+1}{n+1} x \frac{2(n+1)^2}{(2n+1)(2n+2)} L_n(x) \frac{2^n (n!)^2}{(2n)!}$$
$$- \frac{n}{(n+1)} \frac{4(n+1)^2 n^2}{(2n+2)(2n+1)2n(2n-1)} L_{n-1}(x) \frac{2^{n-1}((n-1)!)^2}{(2n-2)!} \Leftrightarrow$$

$$\varphi_{n+1}(x) = x\varphi_n(x) - \frac{n^2}{4n^2-1} \varphi_{n-1}(x) \tag{3.7}$$

where

$$\varphi_n(x) = \tfrac{2^n (n!)^2}{(2n)!} L_n(x).$$

Comparing (3.7) with (3.2) it is seen that $a_n = 0$ and $b_n = \frac{n^2}{4n^2-1}$ and hence Jacobi matrix are

$$J = \begin{bmatrix} 0 & \sqrt{\frac{1}{3}} & & & \\ \sqrt{\frac{1}{3}} & 0 & \sqrt{\frac{4}{15}} & & \\ & \sqrt{\frac{4}{15}} & 0 & \sqrt{\frac{9}{35}} & \\ & & \ddots & \ddots & \ddots \end{bmatrix}.$$

Hereby the eigenvalue analysis can be conducted and the eigenvectors v_k and corresponding eigenvalues λ_k can be obtained. The quadrature abscissas are directly the eigenvalues $x_k = \lambda_k$ while the weights w_k are found by weight a function, in this case $w(x) = 1$ [5], and hereby it follows that

$$w_k = v_{0,k}^2 1 \int_{-1}^{1} 1 \, dx = v_{0,k}^2 [x]_{-1}^{1} = 2v_{0,k}^2.$$

All this is implemented in a function which is shown in appendix B.1.3

Hermite-gauss quadrature

Again the starting point is the three-term recurrence

$$H_{n+1}(x) = xH_n(x) - nH_{n-1}(x) \qquad n = 0, 1, 2 \ldots.$$

The leading coefficient has to be found to get the recurrence on monic form. The 5 Hermite polynomials listed earlier have the leading coefficients $1, 1, 1, 1, 1$. This means that the above recurrence already is on monic form and it is seen directly that $a_n = 0$ and $b_n = n$. The Jacobi matrix is hereby

$$J = \begin{bmatrix} 0 & 1 & & & \\ 1 & 0 & \sqrt{2} & & \\ & \sqrt{2} & 0 & \sqrt{3} & \\ & & \ddots & \ddots & \ddots \end{bmatrix}.$$

The weight function are $w(x) = e^{-x^2/2}$, which gives the weights

$$w_k = v_{0,k}^2 \int_{-\infty}^{\infty} e^{-x^2/2} \, dx = v_{0,k}^2 \sqrt{2\pi}$$

where the last step is found in [6]. The implementation in Python is listed in appendix B.1.4. The next section will deals with polynomial interpolation. Here the used polynomials are the Lagrange polynomials which belongs to the nodal representation where the presented polynomials belongs to the modal representation [3]. The reason the Lagrange polynomials are used is the beneficial construction. Since it is possible to transform between the nodal and modal representation verifying the use of the Lagrange polynomials.

3.3 Polynomial interpolation

In general interpolation is use to construct a solution in between the solution points $u(x_i)$. The simplest form of interpolating is the linear interpolation which makes straight line segments between the solution points. To achieve a good result with linear interpolation a huge number of points often is required.

Another way to interpolate is to use polynomial interpolation. Here the final solution is the polynomial $P(x)$ where it in the nodes x_i is required that $P(x_i) = u(x_i)$. This can in the nodal representation be written by

$$\mathcal{I}_N u(x) = \sum_{i=0}^{N} u(x_i) h_i(x), \tag{3.8}$$

following [3], where $h_i(x)$ is the Lagrange polynomial defined as

$$h_i(x) = \prod_{j=0, j\neq i}^{N} \frac{x - x_j}{x_i - x_j},$$

and in the points satisfy

$$h_i(x_j) = \delta_{ij} \tag{3.9}$$

The corresponding modal representation interpolation function is in the same way constructed by

$$\mathcal{I}_N u(x) = \sum_{i=0}^{N} \hat{u}_i \Phi_i(x), \tag{3.10}$$

where $\Phi_i(x)$ is a model basis polynomial function e.g. Legendre or Hermite polynomials. The coefficients $\hat{u}(x_i)$, on the interval $x \in [a, b]$, are defined by (from [3])

$$\hat{u}_i = \frac{1}{\gamma_i} \langle u, \Phi_i \rangle_w = \frac{1}{\gamma_i} \int_a^b u(x) \Phi_i(x) w(x)\, dx$$

By using the proper quadrature rule on this integral the expression is discretized and hence an solvable expression for the coefficients are obtained. The interpolation for both the nodal and modal representation are used to develop the deterministic solver and is presented next.

3.4 Deterministic Collocation method

In order to solve a SDE, a solver for the spatial part of a system is needed. Here the spectral Collocation method will be used and will from now on be referred to as the Deterministic Collocation method because of a later introduction of the UQ method - the Stochastic Collocation method.

3.4.1 Construction of the Deterministic Collocation method

As the name indicates the method uses collocation points $x_i, i = 1, 2, \ldots, N$. The method require that the solution in these points has to be exact. The idea behind the method is that the differentiated parts are determined e.g. by

$$\frac{d\boldsymbol{u}}{dx} = \boldsymbol{D}\boldsymbol{u}. \tag{3.11}$$

where $\boldsymbol{u} = [u(x_1), u(x_2), \ldots, u(x_N)]$. To construct the $\boldsymbol{D}$ matrix the so-called Vandermonde matrix is needed. This matrix arises from the relationship between the nodal and modal representation. From [3] the relation in the collocation points x_i between the nodal and modal representation can be written by

$$u(x_i) = \sum_{j=0}^{N} u(x_j)h_j(x_i) = \sum_{j=0}^{N} \hat{u}_j \Phi_j(x_i), \quad i = 0, 1, \ldots, N.$$

From this the relationship can be expressed by

$$\boldsymbol{u} = \boldsymbol{V}\hat{\boldsymbol{u}}, \tag{3.12}$$

where the indexes in $\boldsymbol{V}$ are $V_{ij} = \Phi_j(x_i)$. With the relation between the modal and nodal coefficient the corresponding relationship between the nodal and modal basis polynomials can also be related by

$$\boldsymbol{u}^T\boldsymbol{h}(x) = \hat{\boldsymbol{u}}^T\boldsymbol{\Phi}(x).$$

From (3.12) this can be manipulated into the following.

$$\begin{aligned}
\boldsymbol{u}^T\boldsymbol{h}(x) &= \hat{\boldsymbol{u}}^T\boldsymbol{\Phi}(x) \Leftrightarrow \\
(\boldsymbol{V}\hat{\boldsymbol{u}})^T\boldsymbol{h}(x) &= \hat{\boldsymbol{u}}^T\boldsymbol{\Phi}(x) \Leftrightarrow \\
\hat{\boldsymbol{u}}^T\boldsymbol{V}^T\boldsymbol{h}(x) &= \hat{\boldsymbol{u}}^T\boldsymbol{\Phi}(x) \Leftrightarrow \\
\boldsymbol{V}^T\boldsymbol{h}(x) &= \boldsymbol{\Phi}(x).
\end{aligned}$$

This makes it clear that the transformation between the nodal and modal basis is quite simple in form of the Vandermonde matrix. The goal is to find an operator to approximate the derivatives in the nodal space. So by taking the first derivative of first of the modal space it is obtained that

$$\frac{du(x)}{dx} = \frac{d}{dx}\left(\sum_{j=0}^{N} \hat{u}_j \Phi_j(x)\right) = \sum_{j=0}^{N} \hat{u}_j \frac{d}{dx}\Phi_j(x) = \boldsymbol{V}_x\hat{\boldsymbol{u}},$$

where $\boldsymbol{V}_x$ is the differentiated Vandermonde matrix. In the same way the first derivative of the nodal space gives

$$\frac{du(x)}{dx} = \frac{d}{dx}\left(\sum_{j=0}^{N} u(x_j)h_j(x)\right) = \sum_{j=0}^{N} u(x_j)\frac{d}{dx}h_j(x) = \boldsymbol{D}\boldsymbol{u},$$

where $\boldsymbol{D}$ is the differentiation matrix. Further manipulation of the above shows that

$$\frac{d\boldsymbol{u}}{dx} = \boldsymbol{D}\boldsymbol{u} = \boldsymbol{D}(\boldsymbol{V}\hat{\boldsymbol{u}}) = \boldsymbol{V}_x\hat{\boldsymbol{u}}.$$

From this the differentiation matrix is obtained to be $\boldsymbol{D} = \boldsymbol{V}_x\boldsymbol{V}^{-1}$. The question is how these matrices are constructed in practise. It requires both knowledge of the basis polynomials $\Phi_n(x)$ and the corresponding derivative basis polynomials $\Phi_n'(x)$. In this project the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$ are used as basis polynomials, which are defined on the interval $x \in [-1,1]$, where α and β are parameters. The recurrence for the Jacobi polynomials are given as

$$\begin{aligned}
P_0^{(\alpha,\beta)}(x) &= 1\\
P_1^{(\alpha,\beta)}(x) &= \tfrac{1}{2}(\alpha - \beta + (\alpha+\beta+2)x)\\
P_{n+1}^{(\alpha,\beta)}(x) &= \frac{(a_{n,n}^{(\alpha,\beta)} + x)P_n^{(\alpha,\beta)}(x) - a_{n-1,n}^{(\alpha,\beta)}P_{n-1}^{(\alpha,\beta)}(x)}{a_{n+1,n}^{(\alpha,\beta)}},
\end{aligned}$$

with the coefficients for $n > 0$

$$\begin{aligned}
a_{n-1,n}^{(\alpha,\beta)} &= \frac{2(n+\alpha)(n+\beta)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta)}\\
a_{n,n}^{(\alpha,\beta)} &= \frac{\alpha^2-\beta^2}{(2n+\alpha+\beta+2)(2n+\alpha+\beta)}\\
a_{n+1,n}^{(\alpha,\beta)} &= \frac{2(n+1)(n+\alpha+\beta+1)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta+2)},
\end{aligned}$$

and $a_{-1,0}^{(\alpha,\beta)} = 0$ [3]. If the parameters are chosen to be $\alpha = \beta = 0$ the Legendre polynomials are constructed. For practical computations it can be useful to use the normalized Jacobi polynomials $\tilde{P}_n^{(\alpha,\beta)}(x)$ which from [3] is determined by

$$\tilde{P}_n^{(\alpha,\beta)}(x) = \gamma_n^{(\alpha,\beta)}P_n^{(\alpha,\beta)}(x) \tag{3.13}$$

where the normalizing factor is

$$\gamma_n^{(\alpha,\beta)} = 2^{\alpha+\beta+1}\frac{(n+\alpha)!(n+\beta)!}{n!(2n+\alpha+\beta+1)(n+\alpha+\beta)!}.$$

The kth derivatives of the normalized Jacobi polynomial, which is important in order to construct $\boldsymbol{V}_x$, can be described by [3]

$$\frac{d^k}{dx^k}\tilde{P}_n^{(\alpha,\beta)}(x) = \frac{(\alpha+\beta+n+k)!}{2^k(\alpha+\beta+n)!}\sqrt{\frac{\gamma_{n-k}^{(\alpha+k,\beta+k)}}{\gamma_n^{(\alpha,\beta)}}}\tilde{P}_{n-k}^{(\alpha+k,\beta+k)}(x).$$

Hereby the first derivative is obtained to be

$$\frac{d}{dx}\tilde{P}_n^{(\alpha,\beta)}(x) = \tilde{P}_n^{(\alpha,\beta)}(x)' = \sqrt{n(n+\alpha+\beta+1)}\tilde{P}_{n-1}^{(\alpha+1,\beta+1)}(x). \tag{3.14}$$

The Vandermonde matrix and the differentiated Vandermonde matrix can now be constructed from (3.13) and (3.14).

$$\boldsymbol{V} = \begin{bmatrix} \tilde{P}_0^{(\alpha,\beta)}(x_0) & \tilde{P}_1^{(\alpha,\beta)}(x_0) & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_0) \\ \tilde{P}_0^{(\alpha,\beta)}(x_1) & \tilde{P}_1^{(\alpha,\beta)}(x_1)' & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{P}_0^{(\alpha,\beta)}(x_N) & \tilde{P}_1^{(\alpha,\beta)}(x_N) & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_N) \end{bmatrix}$$

$$\boldsymbol{V}_x = \begin{bmatrix} \tilde{P}_0^{(\alpha,\beta)}(x_0)' & \tilde{P}_1^{(\alpha,\beta)}(x_0)' & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_0)' \\ \tilde{P}_0^{(\alpha,\beta)}(x_1)' & \tilde{P}_1^{(\alpha,\beta)}(x_1)' & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_1)' \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{P}_0^{(\alpha,\beta)}(x_N)' & \tilde{P}_1^{(\alpha,\beta)}(x_N)' & \dots & \tilde{P}_N^{(\alpha,\beta)}(x_N)' \end{bmatrix}.$$

From these two matrices the differentiation matrix $\boldsymbol{D}$ are determined and used to approximate the derivatives in the given differential equation.

The collocation points $x_i, i = 0, 1, \dots, N$ are found by the corresponding Jacobi-gauss quadrature. This is done by the same procedure as described in section 3.2. These collocation points are called Gauss-Lobatto nodes if the boundary points (-1 and 1) are included and the interior points are found by Jacobi-Gauss quadrature. The exact procedure for this quadrature and the Gauss-Lobatto nodes can be found in [3]. All the functions used for the Deterministic Collocation method are listed in appendix B.1.5.

3.5 Time-integration solver

In the many SDE or PDE there exist a time-dependent term and a time stepping solver is therefore needed. Throughout this project the Python function `odeint` will be used. The background for this solver will shortly be presented in this section.

The function `odeint` uses the so-called LSODE (Livermore Solver for Ordinary Differential Equations) and exists from the `FORTRAN` ODE solver pack `ODEPACK` [7]. LSODE can solve both stiff and non-stiff ODE systems on the general from

$$\frac{\partial u(\boldsymbol{x}, t)}{\partial t} = \mathcal{L}(u).$$

The solver uses both Adams-Moulton (AM) method and the Backward Differentiation Formula (BDF) method in a combination [7]. This combination can both take form as an implicit and explicit solver. To find the solution in each step an iterative predictor-corrector method is used, where the explicit form is used to the predictor step and the implicit form is used to the corrector step [7]. The implicit form is in general faster and more precise compared with the explicit method, since the implicit form can use larger time steps because of the larger stability area. However, if for the same time step the explicit solver is faster due to less complexity.

The LSODE method will handle the time stepping size which makes it easy to use. It only requires a right hand side as a function, the initial condition, the times for the desired solutions and finally the parameters. More information about the solver can be found in [7].

3.6 Generalized Polynomial Chaos

The original Polynomial Chaos (PC) formulation was proposed by Wiener [8]. He suggested that Hermite polynomials can be used to represent a Gaussian random variable in a stochastic process. The generalized Polynomials Chaos (gPC) is an extension to this where other orthogonal polynomials are used to represent random variables with different distributions. In table 3.1 the distributions of the random variable and the corresponding optimal orthogonal polynomials are shown.

It is also possible to use e.g. Hermite polynomials to represent a random variable with a uniform distribution as shown in [2]. This choice of representation leads to a slower convergence and higher orders of the orthogonal polynomials is needed to obtain the same precision compared to using the optimal polynomial.

Distribution of Z	gPC basis polynomials	support
Gaussian	Hermite	$[-\infty, \infty]$
Gamma	Laguerre	$[0, \infty]$
Beta	Jacobi	$[a, b]$
Uniform	Legendre	$[a, b]$

Table 3.1: Relation between the gPC and the continuous distribution of the random variable Z

For one dimension ($d = 1$) the random variable Z can be represented in the probability space $(\Omega, \mathcal{F}, P)$ (where Ω is the sample space, $\mathcal{F}$ a σ-algebra of subsets of Ω and P is the probability measure [2]) by (from [8])

$$Z = \sum_{i=0}^{\infty} c_i \Phi_i(Z),$$

where c_i is the coefficient which in general is given by

$$c_i = \frac{1}{\gamma_i} \mathbb{E}[Z \Phi_i(Z)]$$

and $\Phi_i(Z)$ is the orthogonal polynomial basis is chosen from the distribution of Z [2]. As a small example on how the representation is used the following simple equation will be rewritten

$$\frac{du(t, Z)}{dt} = \alpha(Z) u(t, Z). \tag{3.15}$$

Since the solution $u(t, Z)$ is affected by the uncertainty of the variable $\alpha(Z)$ (assume that $\alpha(Z)$ follows a normal distribution) a gPC expansion can be produced on the $u(t, Z)$ by using the Hermite polynomials $H_i(Z)$ by

$$u(t,Z) = \sum_{i=0}^{m} u_i(t) H_i(Z).$$

Here the summation is truncated to the desired number of polynomial orders m. In the same way $\alpha(Z)$ can be represented by a gPC expansion and (3.15) becomes

$$\sum_{i=0}^{m} \frac{du_i H_i(Z)}{dt} = \sum_{i=0}^{m} a_i H_i(Z) \sum_{i=0}^{m} u_i(t) H_i(Z).$$

From this the methods used in this project takes different approaches to obtain the desired solution. Next the multi dimensional gPC shortly will be presented.

3.6.1 Multi dimensional gPC

Here it is assumed that the system contains more than 1 random variable ($d > 1$) meaning that the gPC expansion gets more complex. However, it will have the same type of representation

$$u(t,\boldsymbol{x},\boldsymbol{Z}) = \sum_{|\boldsymbol{i}|\le M} u_{\boldsymbol{i}}(\boldsymbol{x},t)\Psi_{\boldsymbol{i}}(\boldsymbol{Z}). \tag{3.16}$$

where the $\boldsymbol{i}$ is a multi-index which is defined to be $\boldsymbol{i} = (i_1, i_2, \dots, i_d) \in \mathbb{N}^d$ where $|\boldsymbol{i}| = i_1 + i_2 + \cdots + i_d$. In table 3.2 this is illustrated up to $d = 3$. In equation (3.16)

$\lvert\boldsymbol{i}\rvert$	Multi-index $\boldsymbol{i}$	Single index
0	$(0,0,0,0)$	1
1	$(1,0,0,0)$	2
	$(0,1,0,0)$	3
	$(0,0,1,0)$	4
	$(0,0,0,1)$	5
2	$(2,0,0,0)$	6
	$(1,1,0,0)$	7
	$(1,0,1,0)$	8
	$(1,0,0,1)$	9
	$(0,2,0,0)$	10
	$(0,1,1,0)$	11
	$(0,1,0,1)$	12
	$(0,0,2,0)$	13
	$(0,0,1,1)$	14
	$(0,0,0,2)$	15

Table 3.2: Multi-index $\boldsymbol{i}$ for $d = 3$

$\Psi_{\boldsymbol{i}}(Z)$ is a collection of the orthogonal polynomials which from [2] is given by

$$\Psi_{\boldsymbol{i}}(\boldsymbol{Z}) = \Phi_{i_1}(Z_1)\Phi_{i_2}(Z_2)\cdots\Phi_{i_d}(Z_d), \qquad 0 \le |\boldsymbol{i}| \le M,$$

and in the same way for the deterministic coefficient

$$u_{\boldsymbol{i}} = u_{i_1} u_{i_2} \cdots u_{i_d}, \qquad 0 \leq |\boldsymbol{i}| \leq M.$$

So through preliminary work by setting up $\Psi_{\boldsymbol{i}}$ and $u_{\boldsymbol{i}}$ the representation can be proceeded in a similar way.

This chapter have presented some numerical tools in form of orthogonal polynomials and quadrature rules which plays an essential role for the UQ method. Also both a method to solve deterministic systems have been established and the used time step integrator is also introduced. Lastly the gPC, a tool for representing the uncertain variables, is presented and the reader has sufficient background to move on with the UQ methods.

Chapter 4

Uncertainty Quatification methods

Different UQ methods are developed to solve SDE. These methods can be divided into Non-intrusive methods and Intrusive methods and the most common of these will be in presented here. The simplest and most intuitive is the Monte Carlo Method which is presented first. Afterwards the Stochastic Collocation Method (SCM) and the Stochastic Galerkin Method (SGM) in general will be presented.

4.1 Non-intrusive methods

The Non-intrusive methods are defined as methods that rely on a set of deterministic solutions to determine the statistical parameters for the SDE. In this thesis two Non-intrusive methods will be used - The Monte Carlo Method and the SCM.

4.1.1 Monte Carlo Method

As mentioned before the Monte Carlo Method is the simplest and most intuitive method to solve a SDE. The method relies m_i random samples fitting the random variables $\boldsymbol{Z} = Z_i^{(j)}, i = 1, \dots, d$ and $j = 1, \dots, m_i$. For the Monte Carlo method all random variables have the same representation number $M = m_1 = m_2 = \cdots = m_d$ since each single outcome is a random draw from each distribution. For each sample a solution $u(\boldsymbol{x}, t, \boldsymbol{Z}^{(j)})$ is obtained by a deterministic solver where $k = 1, 2, \dots, M$. For the M solutions an estimate of the mean solution $\bar{u}(\boldsymbol{x}, t)$ and the variance $s_u^2(\boldsymbol{x}, t)$ can be computed by

$$\bar{u}(\boldsymbol{x}, t) = \frac{1}{M} \sum_{k=1}^{M} u(\boldsymbol{x}, t, \boldsymbol{Z}^{(k)}). \tag{4.1}$$

where k is a single index running through all combinations of the nodes in the random space similar to the single index in table 3.2. When the estimated mean is determined the estimate of the variance follows by

$$s_u^2(\boldsymbol{x},t) = \frac{1}{M}\sum_{k=1}^{M}(u(\boldsymbol{x},t,\boldsymbol{Z}^{(j)})^2 - \bar{u}^2. \tag{4.2}$$

To make it more clear how it in practise is executed assume a SDE is containing $d = 2$ random variables, Z_1 and Z_2. For the distributions belonging to these random variables a single random outcome, denoted z_1 and z_2, is drawn. By including these two numbers into the differential equation a deterministic solution can be found. This procedure is performed M times and hereafter the statistics can be determined.

As described in [2] this method is very inefficient. In fact the convergence rate is $O(M^{-1/2})$ which mean that an increase of the precision of one digit will lead to 100 times more calculations. Later this will be illustrated.

4.1.2 Stochastic Collocation Method

Another Non-intrusive method is the Stochastic Collocation Method (SCM). Overall the method can be characterized as a clever Monte Carlo method where the random space is represented with much fewer points, with corresponding weights, which finally are used to determine the estimated mean and variance solution.

The name 'Collocation' refers to the collocation points $z_i^{(j)}, j = 1, 2, \ldots, m_i$. In this part the notation $\boldsymbol{Z}^{(k)}$ will refer to the set of prescribed nodes in the random space of dimension d containing M nodes. In between the collocation points the solution will be interpolated. The interpolation function $\mathcal{I}(u)(\boldsymbol{Z})$ is then required to find a polynomial approximation such that the solution in the collocation points is exact. $\mathcal{I}(u)(\boldsymbol{Z})$ is from [9] defined by

$$\mathcal{I}(u)(\boldsymbol{Z}) = \sum_{k=1}^{M} u(\boldsymbol{Z}^{(k)})h_j(\boldsymbol{Z})$$

where $h_j(Z)$ is the Lagrange polynomial which in the collocation points is given by

$$h_k(\boldsymbol{Z}^{(l)}) = \delta_{kl}, \quad 1 \le l, k \le M. \tag{4.3}$$

To require that the solution is exact in the collocation points means that [9]

$$\sum_{k=1}^{M} u_t(\boldsymbol{x},t,\boldsymbol{Z}^{(k)})h_k(\boldsymbol{Z}^{(l)}) - \mathcal{L}\Big(\sum_{k=0}^{M} u(Z^{(k)})h_k(\boldsymbol{Z}^{(l)})\Big) = 0, \quad l = 1, 2, \ldots, M. \tag{4.4}$$

Hereby the system (2.1) by using (4.3) and [9] can be written into the following system

$$\begin{aligned}
&u_t(\boldsymbol{x},t,Z^{(k)}) = \mathcal{L}(u), && S \times (0,T] \times \mathbb{R} \\
&\mathcal{B}(u) = 0, && \partial S \times [0,T] \times \mathbb{R} \\
&u(\boldsymbol{x},0,Z^{(k)}) = u_0, && S \times \mathbb{R}.
\end{aligned}$$

For each k a deterministic solution is obtained $u(Z^{(k)}) = u(\boldsymbol{x}, t, Z^{(k)})$. To compute the mean from these solution the following is used

$$\mathbb{E}[\mathcal{I}(u(Z))] = \sum_{k=1}^{M} u(Z^{(k)})) \int_\Gamma h_j(Z)\rho(Z)\, dz,$$

where Γ is the random space and $\rho(Z)$ is the pdf of the random variable Z. The integral can be approximated by the quadrature rule with the abscissa y_i and corresponding weights w_i. E.g. if Z (for $d = 1$) is uniformly distributed then the Legendre-Gauss quadrature rule is used and if it follows a normal distributed the Hermite-gauss quadrature rule is used. Hereby the estimated expectation $\mathbb{E}[\mathcal{I}(u(Z))] = \bar{u}(\boldsymbol{x}, t)$ is

$$\bar{u}(\boldsymbol{x}, t) = \sum_{k=1}^{M} u(\boldsymbol{Z}^{(k)}) \sum_{l=1}^{M} h_k(y_l)\rho(y_l)w_l$$

These points found by the quadrature rule can also be used as the collocation points to represent the random space hence $Z^{(k)} = y_k, k = 1, 2, \ldots, M$. By using that $h_k(y_l) = \delta_{kl}$ the above can be reduced to

$$\bar{u}(\boldsymbol{x}, t) = \sum_{k=1}^{M} u(Z^{(k)})\rho(Z^{(k)})w_k. \tag{4.5}$$

The estimated variance can from the estimated mean also be found by

$$s_u^2(t, \boldsymbol{x}) = \sum_{k=1}^{M} \left(u(Z^{(k)}) - \bar{u}\right)^2 w_k \rho(Z^{(k)})). \tag{4.6}$$

This will be applied when solving different types of differential equations.

4.2 Intrusive methods

Another way of solving SDE is the so-called Intrusive methods. An intrusive methodology incorporates stochastic information directly into the governing differential equation [10]. The intrusive method that will be presented here is the Stochastic Galerkin Method (SGM).

4.2.1 Stochastic Galerkin Method

The review of this method will for simplicity be done for 1 random variable ($d = 1$). The SGM uses the gPC expansion on the solution $u(\boldsymbol{x}, t, Z)$ and at the random parameters and apply the expansion onto the given SDE.

Again the starting point is the SDE given in (2.1) where the gPC expansion,

$$u(\boldsymbol{x}, t, Z) = \sum_{k=0}^{M} u_k(\boldsymbol{x}, t)\Phi_k(Z),$$

is performed. In this expansion there are $M+1$ terms. Including the multi dimensional case the total number of terms $M+1$ depends on the dimension M_Z of the multivariate random parameter $\boldsymbol{Z}$ and the highest order M_Φ of the set of polynomials in $\Psi_{\boldsymbol{i}}$ [8]. In this reference the total number is stated to be

$$M+1=\frac{(M_Z+M_\Phi)!}{M_Z!M_\Phi!} \tag{4.7}$$

The choices of the orthogonal polynomials depend on how the random variable is distributed as mentioned earlier. By imposing gPC expansion into the differentiation equation (first line) (2.1) the following is obtained.

$$\sum_{k=0}^{M}\frac{\partial u_k}{\partial t}(\boldsymbol{x},k)\Phi_k(Z)=\mathcal{L}\left(\sum_{k=0}^{M}u_k(\boldsymbol{x},t)\Phi_k(Z)\right)$$

In this equation the so-called Galerkin projection is conducted. A Galerkin projection implies to successively take the inner product of the differential equation above with orthogonal polynomials $\Phi_k(Z), k=0,1,\dots M$ [8]. This can be written as

$$\left\langle\sum_{k=0}^{M}\frac{\partial u_k}{\partial t}(\boldsymbol{x},t)\Phi_k(Z),\Phi_l(Z)\right\rangle=\left\langle\mathcal{L}\left(\sum_{k=0}^{M}u_k(\boldsymbol{x},t)\Phi_k(Z)\right),\Phi_l(Z)\right\rangle,\quad l=0,1,\dots,M.$$

By exploiting the orthogonality it ends up with a system having $M+1$ deterministic coupled equations, which can be solved with an appropriate numerical solver.

The statistics of the $M+1$ solutions from the system above are found by [8]

$$\bar{u}(\boldsymbol{x},t)=u_0(\boldsymbol{x},t) \tag{4.8}$$

$$s_u^2(\boldsymbol{x},t)=\sum_{k=1}^{M}\left\langle\Phi_k^2(Z)\right\rangle u_k^2(\boldsymbol{x},t) \tag{4.9}$$

This ends the chapter of the widely used UQ methods. The Monte Carlo method is included in order to have a great reference. The method is very inefficient but on the other hand very intuitive and easier to implement compared with the other methods. Therefore it can be use to give indication of solutions.

The two other methods are expected to have spectral (exponential) convergence meaning that a high precision can be obtained by relatively few discretisation points. The SCM is based on the deterministic solver also making the method relatively easy to implement while the implementation difficulty of the SGM depends on how complex the SDE is. Hence SCM and SGM will be preferable for different systems.

Chapter 5

Establishment of differential equations

To explain and understand UQ different problems in terms of stochastic Ordinary Differential Equation (ODE) and stochastic Partial Differential Equations (PDE) will be establish. This chapter will present the differential equations which will be solved later by the different UQ methods. First a simple stochastic ODE is presented and the exact expectations and variances are derived for different distributions on both the parameter and on the boundary condition.

5.1 The Test equation

The differential equation that will be established first is the so-called Test equation which is given by

$$\frac{du(t)}{dt} = \lambda u(t), \quad u(0) = k, \qquad t \geq 0, \tag{5.1}$$

where λ is a parameter and k is the Initial Condition(IC). This is a deterministic system and does not contain any random variable. (5.1) can be written in the following stochastic system

$$\frac{du(t, \boldsymbol{Z})}{dt} = \alpha(Z_1)u(t, Z), \quad u(0, \boldsymbol{Z}) = \beta(Z_2), \qquad t \geq 0, \tag{5.2}$$

where $\alpha(Z_1)$ is distributed with some distribution $\mathcal{F}_1$ and $\beta(Z_2)$ is distributed with some distribution $\mathcal{F}_2$. The exact solution to this ODE can be determined analytically. This is done by using separation of variables by (using u as integration variable)

$$\frac{du}{dt} = \alpha(Z_1)u \Leftrightarrow \frac{1}{u}\, du = \alpha(Z_1)\, dt.$$

Then integrate on both sides gives

$$\int_{u(0,\boldsymbol{Z})}^{u(t,\boldsymbol{Z})} \frac{1}{u}\, du = -\alpha(Z_1) \int_0^t 1\, dt \Leftrightarrow [\ln(u)]_{u(0,\boldsymbol{Z})}^{u(t,\boldsymbol{Z})} = \alpha(Z_1)t$$

and the exact solution is then obtained to be

$$\begin{aligned}\ln(u(t,\boldsymbol{Z})) - \ln(u(0,\boldsymbol{Z})) &= \alpha(Z_1)t\\ \ln\left(\frac{u(t,\boldsymbol{Z})}{u(0,\boldsymbol{Z})}\right) &= \alpha(Z_1)t\\ \frac{u(t,\boldsymbol{Z})}{u(0,\boldsymbol{Z})} &= e^{\alpha(Z_1)t}\\ u(t,\boldsymbol{Z}) &= u(0,\boldsymbol{Z})e^{\alpha(Z_1)t}.\end{aligned}$$

By insertion of the boundary condition the exact solution is

$$u(t,\boldsymbol{Z}) = \beta(Z_2)e^{\alpha(Z_1)t}.$$

5.1.1 Statistical parameters for the Normal distribution

The statistical parameters are advantageous to know in order to investigate the correctness of the used UQ method but also to compare the different methods. These parameters (the exact expectation $\mathbb{E}[u] = \mu_u$ and the exact variance $\mathrm{Var}[u] = \sigma_u^2$) will here be determined analytically for different cases

Case 1: $\alpha(Z_1) \sim \mathcal{N}(\mu_1, \sigma_1^2)$ **and** $\beta(Z_2) = k$

In this first case the parameter $\alpha(Z_1)$ follows a normal distribution and the IC $\beta(Z_2)$ is kept constant at $k \in \mathbb{R}$. The outcome will be denoted ω. Recall that the expectation of a function is determined by

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x)\,dx. \tag{5.3}$$

Since $u(t,\boldsymbol{Z}) = \beta(Z_2)e^{\alpha(Z_1)t}$ then $\mathbb{E}[u] = \mathbb{E}[\beta]\mathbb{E}[e^{\alpha t}]$, if it is assumed that $\alpha(Z_1)$ and $\beta(Z_2)$ are independent. The hard part is to compute $\mathbb{E}[e^{\alpha t}]$ as it contains the uncertainty. It is done below.

$$\begin{aligned}\mathbb{E}[e^{\alpha t}] &= \int_{-\infty}^{\infty} e^{\omega t}\frac{1}{\sqrt{2\pi\sigma_1^2}}e^{-\frac{(\omega-\mu_1)^2}{2\sigma_1^2}}\,d\omega\\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\int_{-\infty}^{\infty} e^{\omega t}e^{-\frac{(\omega-\mu_1)^2}{2\sigma_1^2}}\,d\omega\\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\int_{-\infty}^{\infty} e^{\omega t-\frac{(\omega-\mu_1)^2}{2\sigma_1^2}}\,d\omega.\end{aligned} \tag{5.4}$$

The exponent in the exponential function can be manipulated further

$$\begin{aligned}\omega t - \frac{(\omega-\mu_1)^2}{2\sigma_1^2} &= \omega t - \frac{1}{2\sigma_1^2}(\omega^2+\mu_1^2-2\omega\mu_1) \\ &= -\frac{1}{2\sigma_1^2}(\omega^2+\mu_1^2-2\omega\mu_1-2\sigma_1^2\omega t) \\ &= -\frac{1}{2\sigma_1^2}(\omega^2+\mu_1^2-2\omega(\mu_1+\sigma_1^2 t)). \qquad (5.5)\end{aligned}$$

By the calculations in (5.6) a connection to (5.5) is seen.

$$\begin{aligned}(\omega-(\mu_1+\sigma_1^2 t))^2 &= \omega^2+(\mu_1+\omega^2 t)^2-2\omega(\mu_1+\sigma_1^2 t) \\ &= \omega^2+\mu_1^2+\sigma_1^4 t^2+2\mu_1\sigma_1^2 t-2\omega(\mu_1+\sigma_1^2 t) \\ &= \omega^2+\mu_1^2-2\omega(\mu_1+\sigma_1^2 t)+\sigma_1^4 t^2+2\mu_1\sigma_1^2 t. \qquad (5.6)\end{aligned}$$

The first 3 terms in (5.6) is the same as the parentheses in (5.5). Therefore (5.5) can be expressed through

$$\begin{aligned}-\frac{1}{2\sigma_1^2}(\omega^2+\mu_1^2-2\omega(\mu_1+\sigma_1^2 t)) &= -\frac{1}{2\sigma_1^2}((\omega-(\mu_1+\sigma_1^2 t))^2-\sigma_1^4 t^2-2\mu_1\sigma_1^2 t) \\ &= -\frac{1}{2\sigma_1^2}(\omega-(\mu_1+\sigma_1^2 t))^2+\frac{1}{2}\sigma_1^2 t^2+\mu_1 t.\end{aligned}$$

This can be inserted into (5.4) and further manipulated to

$$\begin{aligned}&\frac{1}{\sqrt{2\pi\sigma_1^2}}\int_{-\infty}^{\infty} e^{-\frac{1}{2\sigma_1^2}(\omega-(\mu_1+\sigma_1^2 t))^2+\frac{1}{2}\sigma_1^2 t^2+\mu_1 t}\, d\omega \\ &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\int_{-\infty}^{\infty} e^{-\frac{1}{2\sigma_1^2}(\omega-(\mu_1+\sigma_1^2 t))^2} e^{\frac{1}{2}(\sigma_1^2 t^2+2\mu_1 t)}\, d\omega \\ &= e^{\frac{1}{2}(\sigma_1^2 t^2+2\mu_1 t)}\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{1}{2\sigma_1^2}(\omega-(\mu_1+\sigma_1^2 t))^2}\, d\omega.\end{aligned}$$

The terms inside the integration sign will integrate to 1 because it is the density function for $\mathcal{N}(\mu_1+\sigma_1^2 t, \sigma_1^2)$. Therefore it ends up with

$$\mathbb{E}[e^{\alpha t}] = e^{\frac{1}{2}(\sigma_1^2 t^2+2\mu_1 t)}.$$

Since $\mathbb{E}[\beta]=b$ the final exact expectation for $u(t,\boldsymbol{Z})$ is

$$\mu_u = \mathbb{E}[u] = \mathbb{E}[\beta]\mathbb{E}[e^{\alpha t}] = k e^{\frac{1}{2}(\sigma_1^2 t^2+2\mu_1 t)}. \qquad (5.7)$$

The variance is calculated from the mean in general by

$$\sigma_X^2 = \mathrm{Var}(X) = \mathbb{E}[X^2]-(\mathbb{E}[X])^2, \qquad (5.8)$$

and $\mathbb{E}[X^2]$ can be calculated by

$$\mathbb{E}[g(X)^2] = \int_{-\infty}^{\infty} g(x)^2 f_X(x)\, dx.$$

Translating this into this case the following is obtained to be

$$\mathbb{E}[u^2] = \mathbb{E}[\beta^2]\mathbb{E}[(e^{\alpha t})^2] = k^2 \int_{-\infty}^{\infty} e^{\omega 2t} \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{\frac{(\omega-\mu_1)^2}{2\sigma_1^2}}\, d\omega.$$

The integral is almost the same as the integral in (5.4). The only difference is the $2t$ instead of the t in the first exponential function. Therefore the solution for $E[e^{\alpha t}]$ can be used and by substitute t with $2t$ the following is achieved

$$\begin{aligned}\mathbb{E}[e^{\alpha(Z)2t}] &= e^{\frac{1}{2}(\sigma_1^2(2t)^2+2\mu_1 2t)}\\ &= e^{\frac{1}{2}(\sigma_1^2 4t^2+4\mu_1 t)}\\ &= e^{2\sigma_1^2 t^2+2\mu_1 t}.\end{aligned}$$

The second term in (5.8) is calculated by

$$(\mathbb{E}[u])^2 = (ke^{\frac{1}{2}(\sigma_1^2 t^2+2\mu_1 t)})^2 = k^2 e^{(\sigma_1^2 t^2+2\mu_1 t)}.$$

Therefore the final expression for the exact variance for the Test equation where $\alpha(Z) \sim \mathcal{N}(\mu_1, \sigma_1^2)$ is

$$\begin{aligned}\sigma_u^2 &= k^2 e^{2\sigma_1^2 t^2+2\mu_1 t} - k^2 e^{\sigma_1^2 t^2+2\mu_1 t}\\ &= k^2(e^{2\sigma_1^2 t^2+2\mu_1 t} - e^{\sigma_1^2 t^2+2\mu_1 t}).\end{aligned} \tag{5.9}$$

Case 2: $\alpha(Z_1) = k$ **and** $\beta(Z_2) \sim \mathcal{N}(\mu_2, \sigma_2^2)$

In this case it is the IC $\beta(Z_2)$ which follows a normal distribution and $\alpha(Z_2)$ is kept constant. Again the exact expectation μ_u and the exact variance σ_u^2 should be calculated. The μ_u is determined by

$$\mu_u = \mathbb{E}[u] = \mathbb{E}[\beta]\mathbb{E}[e^{kt}],$$

where $\mathbb{E}[e^{kt}] = e^{kt}$ since $\alpha(Z_1)$ is a constant. The other expectation $\mathbb{E}[\beta] = \mu_2$ because this is how $\beta(Z_2)$ is defined. Therefore the final expectation where $\beta(Z_2)$ follows a normal distribution is given by

$$\mu_u = \mathbb{E}[u] = \mu_2 e^{kt}.$$

The exact variance for this case σ_u^2 can again be calculated by (5.8). The two terms is given by

$$\begin{aligned}
(\mathbb{E}[u])^2 &= \mu_2^2 e^{2kt}, \\
\mathbb{E}[u^2] &= \int_{-\infty}^{\infty} \omega^2 e^{2kt} \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(\omega-\mu_2)^2}{2\sigma_2^2}}\, d\omega \\
&= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}} \int_{-\infty}^{\infty} \omega^2 e^{-\frac{(\omega-\mu_2)^2}{2\sigma_2^2}}\, d\omega.
\end{aligned}$$

The last integral have to be manipulated in order to obtain a useful expression. This is done by first creating a variable transformation by $\varpi = \omega - \mu_2$. Since $\frac{d\varpi}{d\omega} = \frac{d\omega+\mu_2}{d\omega} = 1$ then $d\omega = d\varpi$. The above can hereby be rewritten to

$$\begin{aligned}
\mathbb{E}[u^2] &= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}} \int_{-\infty}^{\infty} (\varpi+\mu_2)^2 e^{-\frac{(\varpi+\mu_2-\mu_2)^2}{2\sigma_2^2}}\, d\varpi \\
&= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}} \int_{-\infty}^{\infty} (\varpi^2+\mu_2^2+2\varpi\mu_2) e^{-\frac{\varpi^2}{2\sigma_2^2}}\, d\varpi.
\end{aligned}$$

This can be divided into 3 integrals

$$\mathbb{E}[u^2] = \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}} \left(\int_{-\infty}^{\infty} \varpi^2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\, d\varpi + \int_{-\infty}^{\infty} \mu_2^2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\, d\varpi + \int_{-\infty}^{\infty} 2\varpi\mu_2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\, d\varpi \right).$$

By substituting back in the last integral it is obtained that

$$\begin{aligned}
\int_{-\infty}^{\infty} 2\varpi\mu_2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\, d\varpi &= 2\mu_2 \int_{-\infty}^{\infty} (\omega-\mu_2) e^{-\frac{(\omega-\mu_2)^2}{2\sigma_2^2}}\, d\omega \\
&= 2\mu_2 \left(\int_{-\infty}^{\infty} (\omega e^{-\frac{(\omega-\mu_2)^2}{2\sigma_2^2}}\, d\omega - \mu_2 \int_{-\infty}^{\infty} e^{-\frac{(\omega-\mu_2)^2}{2\sigma_2^2}}\, d\omega \right) \\
&= 2\mu_2 \left(\mu_2\sqrt{2\sigma_2^2\pi} - \mu_2\sqrt{2\sigma_2^2\pi} \right) = 0.
\end{aligned}$$

The last step is done by an integral reference from [6]. The two remaining integrals can also by a manipulated from [6] to

$$\begin{aligned}\mathbb{E}[u^2] &= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}}\left(\int_{-\infty}^{\infty}\varpi^2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\,d\varpi + \int_{-\infty}^{\infty}\mu_2^2 e^{-\frac{\varpi^2}{2\sigma_2^2}}\,d\varpi\right)\\ &= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}}\left(\frac{1}{2}\sqrt{\pi(2\sigma_2^2)^3} + \mu_2^2\sqrt{2\pi\sigma_2^2}\right)\\ &= \frac{e^{2kt}}{\sqrt{2\pi\sigma_2^2}}\left(\frac{1}{2}2\ \sigma_2^2\sqrt{2\pi\sigma_2^2} + \mu_2^2\sqrt{2\pi\sigma_2^2}\right)\\ &= e^{2kt}\left(\sigma_2^2+\mu_2^2\right).\end{aligned}$$

Hereby the exact variance solution is found to be

$$\sigma_u^2 = \mathbb{E}[u^2] - (\mathbb{E}[u])^2 = e^{2kt}\left(\sigma_2^2+\mu_2^2\right) - \mu_2^2 e^{2kt} = e^{2kt}\sigma_2^2.$$

Case 3: $\alpha(Z_1) = \mathcal{N}(\mu_1,\sigma_1^2)$ **and** $\beta(Z_2) \sim \mathcal{N}(\mu_2,\sigma_2^2)$

In this case both $\alpha(Z_1)$ and $\beta(Z_2)$ are normal distributed. The exact expectation μ_u is again calculated by

$$\mu_{\alpha\beta} = \mathbb{E}[u] = \mathbb{E}[\beta]\mathbb{E}[e^{\alpha t}].$$

From earlier the two expectation terms in the expression above is found to be

$$\begin{aligned}\mathbb{E}[\beta] &= \mu_2\\ \mathbb{E}[e^{\alpha t}] &= e^{\frac{1}{2}(\sigma_1^2 t^2+\mu_1 t)},\end{aligned}\tag{5.10}$$

which gives the exact expectation as

$$\mu_u = \mu_2 e^{\frac{1}{2}(\sigma_1^2 t^2+\mu_1 t)}.$$

The exact variance σ_u^2 is similar to earlier found by

$$\sigma_{\alpha\beta}^2 = \mathbb{E}[u^2] - (\mathbb{E}[u])^2 = \mathbb{E}[\beta^2]\mathbb{E}[(e^{\alpha t})^2] - (\mathbb{E}[\beta])^2(\mathbb{E}[e^{\alpha t}])^2.$$

All these 4 expectations have been derived earlier to be

$$\begin{aligned}\mathbb{E}[\beta^2] &= \mu_2^2+\sigma_2^2,\\ \mathbb{E}[(e^{\alpha t})^2] &= e^{2\sigma_1^2 t^2+2\mu_1 t},\\ (\mathbb{E}[\beta])^2 &= \mu_2^2,\\ (\mathbb{E}[e^{\alpha t}])^2 &= e^{\sigma_1^2 t^2+2\mu_1 t},\end{aligned}$$

and by insertion

$$\sigma_u^2 = (\mu_2^2+\sigma_2^2)e^{2\sigma_1^2 t^2+2\mu_1 t} - \mu_2^2 e^{\sigma_1^2 t^2+2\mu_1 t}.$$

5.1.2 Statistical parameters for the Uniform distribution

The random variables could also be distributed by the Uniform distribution $\mathcal{U}[a,b]$. If so the expectation and variance for the Test equations will take another expression which will be determined analytical in this section.

Case 1: $\alpha(Z_1) \sim \mathcal{U}(a_1,b_1)$ and $\beta(Z_2) = k$

The parameter $\alpha(Z_1)$ will now follow a Uniform distribution on a interval $[a_1,b_1]$ while the IC is a constant k. By (5.3) the expectation for $\mathbb{E}[e^{\alpha t}]$ in this case can be expressed by

$$\begin{aligned}\mathbb{E}[e^{\alpha t}] &= \int_{a_1}^{b_1} e^{\omega t}\frac{1}{b_1-a_1}\,d\omega\\ &= \frac{1}{b_1-a_1}\int_{a_1}^{b_1} e^{\omega t}\,d\omega\\ &= \frac{1}{b_1-a_1}\left[\frac{1}{t}e^{\omega}\right]_{a_1}^{b_1}\\ &= \frac{1}{t(b_1-a_1)}(e^{b_1 t}-e^{a_1 t}).\end{aligned}$$

So the final expression for the expectation is

$$\mu_u = \mathbb{E}[u] = \frac{k}{t(b_1-a_1)}(e^{b_1 t}-e^{a_1 t}). \tag{5.11}$$

Again the variance is derived by (5.8) where the terms are

$$\begin{aligned}\mathbb{E}[u^2] = \mathbb{E}[(ke^{\alpha t})^2] = k^2\int_{a_1}^{b_1}(e^{\omega t})^2\frac{1}{b_1-a_1}\,d\omega &= \frac{k^2}{2t(b_1-a_1)}(e^{b_1 2t}-e^{a_1 2t}),\\ (\mathbb{E}[u])^2 = \left(\frac{k}{t(b_1-a_1)}(e^{b_1 t}-e^{a_1 t})\right)^2 &= \frac{k^2}{(t(b_1-a_1))^2}(e^{b_1 t}-e^{a_1 t})^2.\end{aligned}$$

By this the exact variance is

$$\sigma_\alpha^2 = \mathbb{E}[u^2]-(\mathbb{E}[u])^2 = k^2\left(\frac{1}{2t(b_1-a_1)}(e^{b_1 2t}-e^{a_1 2t})-\frac{1}{(t(b_1-a_1))^2}(e^{b_1 t}-e^{a_1 t})^2\right). \tag{5.12}$$

Other cases where the exact mean and variance solution are determined analytically for the random variable(s) following the Uniform distribution can be found in appendix A.

Now the exact solutions for different cases are found and can be used to investigate the convergence of the different UQ methods, but also to compare the estimated mean and variance solutions between the UQ methods.

5.2 The Burger's equation

A bit more complicated differential equation is the Burger's equation, because it has a spatial dimension and contains a non-linear term. The deterministic Burger's equation with specified boundary condition and initial conditions for $x \in [-1, 1]$ and $t \geq 0$ is given by

$$\begin{aligned}
&\frac{\partial u(x,t)}{\partial t} + u(x,t)\frac{\partial u(x,t)}{\partial x} = \nu \frac{\partial^2 u(x,t)}{\partial x^2},\\
&u(-1,t) = 1,\\
&u(1,t) = -1,\\
&u(x,0) = -x,
\end{aligned}$$

where ν is the viscosity [2]. The deterministic equation with $\nu = 0.05$ can be solved by using the Deterministic Collocation method explained in section 3.4. An exact analytic mean solution for the Burger's equation is not known here, so the solution of the deterministic system will give an indication of what can be expected of the mean solution. The implementation to solve the deterministic Burger's equation can be seen in appendix B.1.6. The solution for different times t are shown in figure 5.1.

From the figure it is seen that over time the solution goes against a steady state solution that seems to be reached at $t = 20$. However, this steady state point is not constant and depend on the input parameters to the system.

The stochastic Burger's equation that will be solved by the UQ methods is the system where some uncertainty is the following equation.

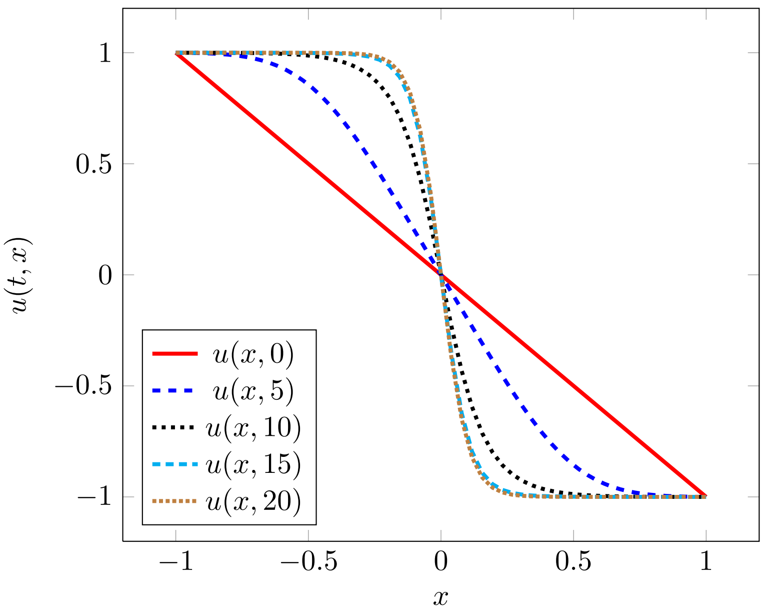

Figure 5.1: Deterministic solution for the Burger's equation with initial condition $u(x,0) = -x$ to different times t

$$
\begin{aligned}
&\frac{\partial u(x,t,\boldsymbol{Z})}{\partial t} + u(x,t,\boldsymbol{Z})\frac{\partial u(x,t,\boldsymbol{Z})}{\partial x} = \nu(Z_3)\frac{\partial^2 u(x,t,\boldsymbol{Z})}{\partial x^2},\\
&u(-1,t,\boldsymbol{Z}) = 1 + \delta_1(Z_1),\\
&u(1,t,\boldsymbol{Z}) = -1 + \delta_2(Z_2),\\
&u(x,0,\boldsymbol{Z}) = -x.
\end{aligned}
\tag{5.13}
$$

Here there are added uncertainty on both boundary conditions with $\delta_1(Z_1)$ and $\delta_2(Z_2)$ which each follows a given distribution. It is also possible that the parameter $\nu(Z_3)$ contains uncertainty. The main test case in the first test section will be where uncertainty only is on the left boundary. This ends the establishment of the test problems in this thesis which the different UQ method must solve. The stochastic Test equation will mainly be used to validate the UQ methods as the analytical mean and variance solutions know are known. The stochastic Burger's equation will on the other hand be used to challenge the methods on dimensions and efficiency. The next chapter will test the methods with the two problems for $d = 1$.

Chapter 6

Test of Uncertainty Quantification methods

The different methods presented earlier in chapter 4 will know be tested, evaluated and discussed on the relative simple SDE's presented in chapter 5. First the Monte Carlo method is tested on the Stochastic Test equation with a random variable which follows different distributions.

6.1 Stochastic Test equations - Monte Carlo method

The Stochastic Test equation (5.2) with the parameter $\alpha(Z) \sim \mathcal{N}(0, 1)$ and with the boundary condition $\beta = 1$ will be the first test example for the Monte Carlo method. From (5.7) and (5.9) an exact expression for the mean and variance solution are given which will be used for comparison and to determine the error of the Monte Carlo method.

In the implementation of the method $\alpha(Z)$ is created as an $N \times 1$ vector consisting of N random numbers from the distribution $\mathcal{N}(0, 1)$. These random numbers are given to the time-integrator function `odeint` together with the initial condition $u(0, Z) = 1$, the time domain $t = [0, 1]$ with time step $\Delta t = 0.01$ and the right hand side formulated as a function by

```
def rhs(u,t,a):
    u = a*u
    return u
```

`odeint`'s output is the solution $u(t, Z)$ to each time t and for each random variable $\alpha(Z)$. An estimate of the mean and variance solution can then be solved from all the samples by (4.1) and (4.2), respectively. The entire implementation of this test case is shown in appendix B.2.1

In figure 6.1 the mean $\bar{u}(t)$ and variance $s_u^2(t)$ solution for $M = 100$ and $M = 1000$ samples are shown together with the exact mean $\mu_u(t)$ and variance $\sigma_u^2(t)$ solution.

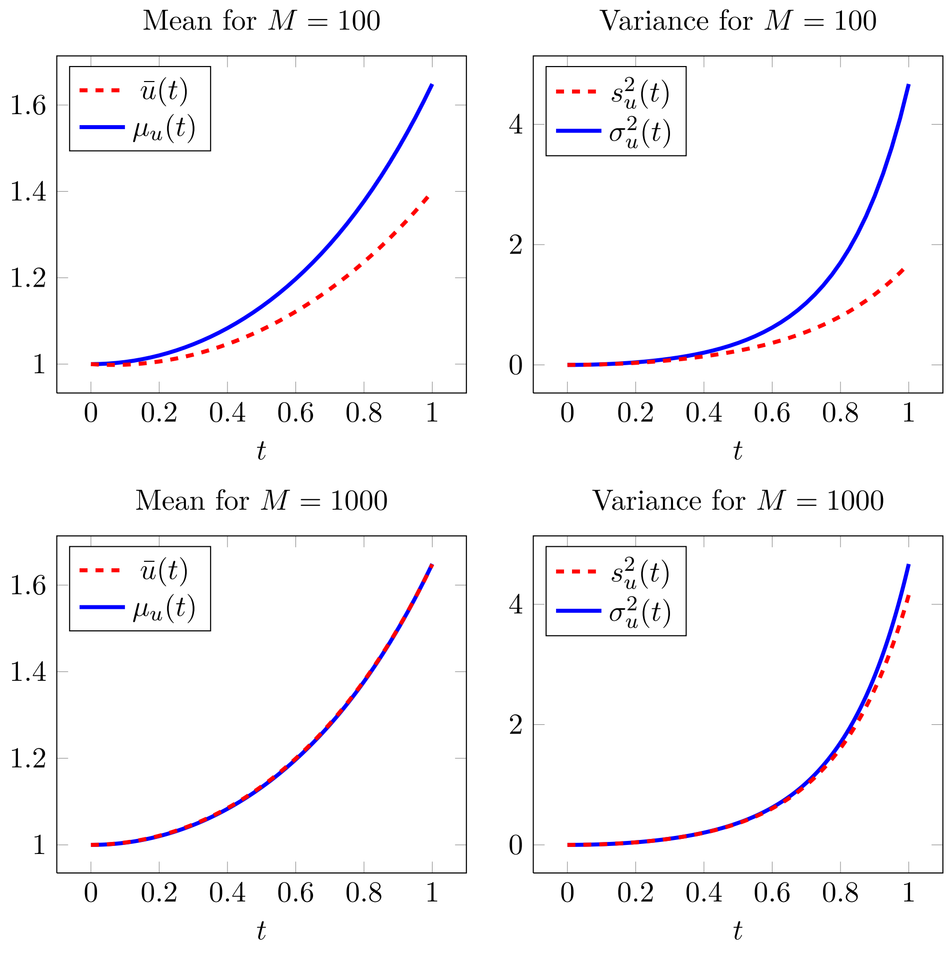

Figure 6.1: Estimated mean and variance for $M = 100$ and $M = 1000$.

As expected the increase of M gives a better precision of the estimated mean and variance. It should be mentioned that if these plots were produced again they could have given a different result that varies from what is seen and it it possible to get a relatively precise result for a low M. However, the chance to obtain a result with high precision is much larger for increasing M. The reason for this is the random draw from the distribution which makes the solution a bit fluctuating.

A more illustrative way to show the estimated mean and standard deviation solution in the same figure where the standard deviation is added and subtracted from the mean solution. In this way it is easier to compare the two solutions point wise. For $M = 1000$ this is produced and illustrated in figure 6.2 for the same problem just solved.

Figure 6.2 illustrates that uncertainty grows as the time goes on, but that should also be the outcome since the variance shown in figure 6.1 also grows over time. This also makes good sense because a small error in the beginning will affect the result in the following time steps and therefore the increase in uncertainty over time.

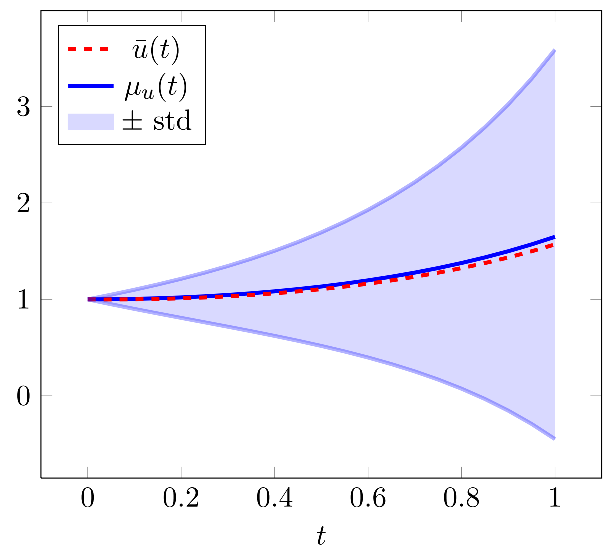

Figure 6.2: Estimated mean for $M = 1000$ with the corresponding standard deviation for $\alpha(Z) \sim \mathcal{N}(0, 1)$.

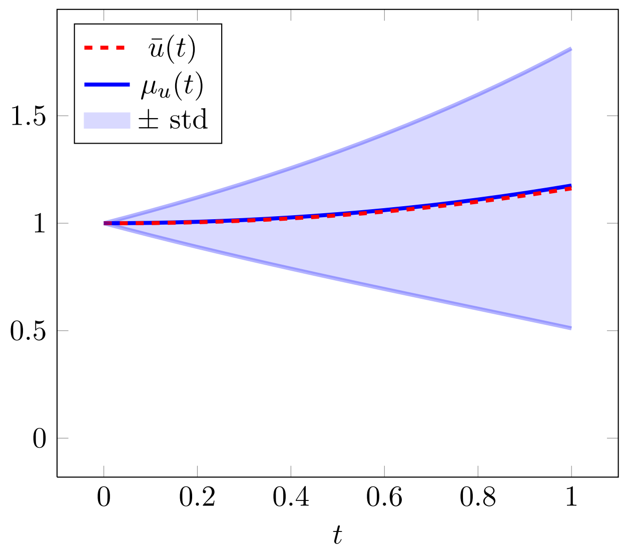

Figure 6.3: Estimated mean for $M = 1000$ with the standard deviation for $\alpha(Z) \sim \mathcal{U}(-1, 1)$.

Now $\alpha(Z)$ is given another distribution - the uniform distribution $\mathcal{U}(-1, 1)$ while all other parameters are kept the same. For this case the exact mean solution $\mu_u(t)$ (5.11) is known and is used to the comparison. The implementation is very similar and can be seen in appendix B.2.1.

Figure 6.3 shows the solutions and the difference is not that big compared with $\alpha(Z)$ following a normal distribution but there seems to be a kind of scaling factor difference between the two solutions. The mean solution does not increase that much where $\alpha(Z)$ following the uniform distribution and the standard deviation area are also smaller. The estimated mean solution is close to the exact mean solution but that was also expected for $M = 1000$. However, the comparison of the solutions in figure 6.2 and 6.3 should not be too concluding, since it is hard to compare if a normal and uniform distribution are similar. On the other hand the two figures illustrates the affect another distribution has.

Lastly an investigation of the convergence rate of the Monte Carlo method solving the stochastic Test equation. As mentioned earlier it is expected that the method have a convergence rate on $O(M^{-1/2})$ where M is the number of simulations. The convergence test will produce for $\alpha(Z) \sim \mathcal{N}(0,1)$ and with the number of simulations $M \in [100, 500000]$. All other parameters are the same as in the first example. The error for M simulations is calculated by

$$E_M = \max_t(|\mu_{u_M}(t) - \bar{u}_M(t)|). \tag{6.1}$$

The implementation of the convergence test is listed in appendix B.2.1.

The convergence rate in figure 6.4 is first of all seen to follow the expected rate (the blue line) roughly. However, the precision does not follow the line perfectly and varies a lot. For one of the biggest M the error could be of size 10^{-4} but for $M+1$, just a number larger, the error could be of size 10^{-2}. This is a quite big variation that occurs by the random selection from the distribution. However, the trend in figure 6.4 is that for increasing M the error will decrease.

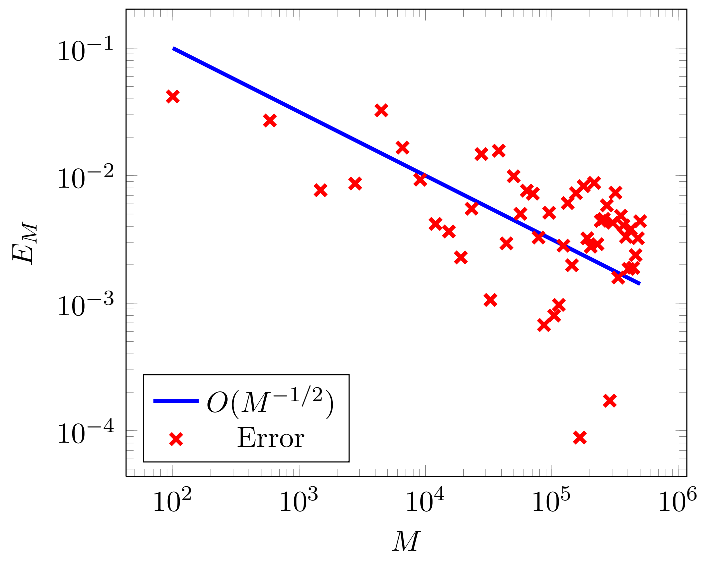

Figure 6.4: Convergence for the Monte Carlo method with $\alpha(Z) \sim \mathcal{N}(0,1)$.

6.2 Stochastic Test equations - Stochastic Collocation Method

More efficient method is needed, since the Monte Carlo method was illustrated to be very inefficient, so now the Stochastic Collocation Method (SCM) is tried out on the stochastic Test equation.

Again the parameter $\alpha(Z)$ is chosen to follow the standard normal distribution $\mathcal{N}(0,1)$, so it is the exact same problem, as solved by the Monte Carlo method. The implementation is very similar as the implementation of the Monte Carlo method. The only difference is how the random variable $\alpha(Z)$ is represented and how the expectation and the variance solution are calculated. $\alpha(Z)$ is represented by M values from the Hermite-Gauss quadrature, since $\alpha(Z)$ is normal distributed. From the quadrature the abscissas $\boldsymbol{z}$ and weights $\boldsymbol{w}$ are returned for the standard normal distribution. The abscissas must therefore be transformed to actual distribution of $\alpha(Z) \sim \mathcal{N}(\mu, \sigma^2)$ by

$$\boldsymbol{\alpha} = \mu + \sigma \boldsymbol{z}.$$

In this case where $\mu = 0$ and $\sigma = 1$ it doesn't change the abscissas. For each element in $\alpha(Z^{(j)}), j = 1, 2, \ldots, M$ a deterministic solution $u(t, \alpha(Z^{(j)})$ are solved by the Deterministic Collocation method and `odeint`. Hence the estimated mean and variance solution can be found by (4.5) and (4.6) which can be compared with the analytical solutions (5.7) and (5.9). The implementation is listed in appendix B.2.1.

In figure 6.5 the result are shown for $M = 5$ nodes representing the random variable. To the left the 5 deterministic solutions are shown and on the right the mean solution $\bar{u}(t)$ and the corresponding standard deviation are shown together with the exact mean solution $\mu_u(t)$.

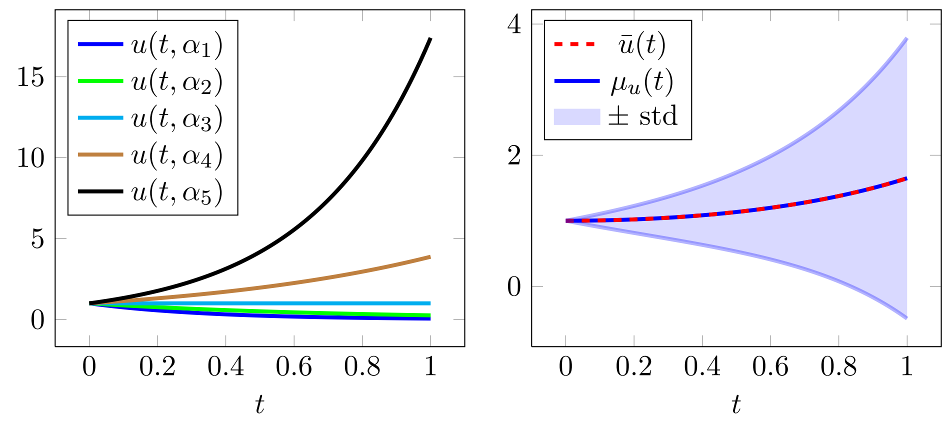

Figure 6.5: Left is the 5 deterministic solutions for the 5 different $\alpha(Z) \sim \mathcal{N}(0,1)$. To the right the corresponding estimated mean for $M = 5$ with the standard deviation.

First of all the estimated mean solution in figure 6.5 seems to be almost the same as the exact mean solution and also better than the result found by the Monte Carlo method with $M = 1000$ samples. The standard deviation seems to be the same. So with only $M = 5$ the precision is improved. The figure to the left shows the 5 deterministic solutions that together with the quadrature weights are the elements for determine the estimated mean and variance solution.

Now it is tried out with $\alpha(Z) \sim \mathcal{U}(-1,1)$ to see if the same efficiency improvement also is obtained for this distribution. It should hopefully not affect the method, so it is expected the same efficiency. To represent $\alpha(Z)$ by the uniform distribution the Legendre polynomials must be used. By the Legendre quadrature the abscissas $\boldsymbol{z}$ and $\boldsymbol{w}$ are found. To transform from the interval $[-1,1]$, where the Legendre polynomials are defined, to the wanted interval $[a,b]$ the following transformation is used

$$\boldsymbol{\alpha} = \frac{b-a}{2}\boldsymbol{z} + \frac{a+b}{2}. \tag{6.2}$$

Again in this case the transformation doesn't change anything because the transformation is from $[-1,1]$ to $[-1,1]$. The implementation of this test is shown in appendix B.2.1. Figure 6.6 shows the results for this case with $M = 5$.

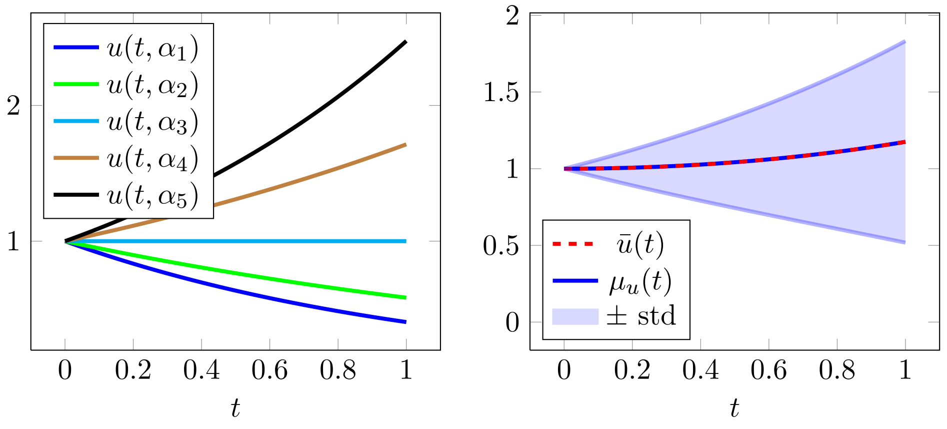

Figure 6.6: Left is the $M = 5$ deterministic solutions for the 5 different $\alpha(Z) \sim \mathcal{U}(-1,1)$. To the right the corresponding estimated mean with the standard deviation.

Again the mean solution $\bar{u}(t)$ is very close to the exact mean solution $\mu_u(t)$ and by comparison of the standard deviation with the corresponding standard deviation found with the Monte Carlo method they seem very similar.

So it seems like that the SCM is a very efficient method to find a precise solutions at least for the Test Equation. To investigate the actual precision the mean and variance solution are determined for different values of M and hereby the convergence can be obtained. This is shown in figure 6.7. M is running on the interval $[1,20]$ and the parameter $\alpha(Z) \sim \mathcal{N}(0,1)$. The error is determined in the same way as in (6.1). The code for the convergence test is listed in appendix B.2.1.

As expected the SCM have shows spectral (exponential) convergence. To compare with the solutions in figure 6.5 where $M = 5$ the error on the mean solution is seen to be approximately 10^{-4}. To obtained this precision by using the Monte Carlo method approximately $M = 5 \cdot 10^5$ have to be used and this guaranties not the desired precision. It is also seen that for $M > 9$ the estimated mean will not be improved and for the estimated variance it is for $M > 13$. The reason for this difference is that the variance increase faster over time compared with the mean solution. This means that the maximum M, where the precision not will be increased, will variate for other choices of random variables and other SDE's. It also depends on how large the time t is. It is therefore not easy to decide the maximum M.

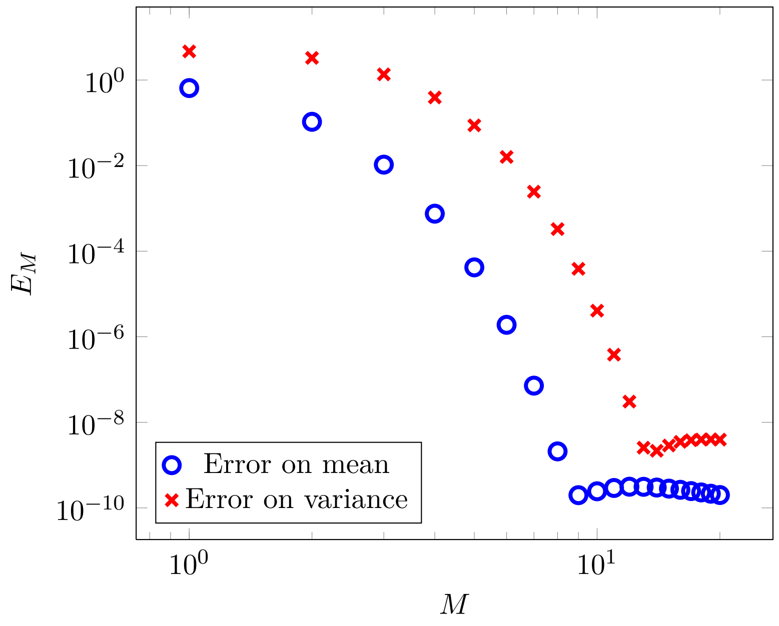

Figure 6.7: Convergence for both mean and variance for the SCM with $\alpha(Z) \sim \mathcal{N}(0, 1)$.

6.3 Stochastic Test equations - Stochastic Galerkin Method

The Test Equation will now be solved by the Stochastic Galerkin Method (SGM) which takes another approach compared to the two other methods. The SGM will be illustrated for the same case where $\alpha(Z) \sim \mathcal{N}(0, 1)$ for easy comparison. Afterwards the convergence test also is conducted for this case.

Before the implementation the system first have to be manipulated. The Test equation is listed here again.

$$\frac{du}{dt}(t, Z) = -\alpha(Z)u(t, Z), \qquad u(0, Z) = \beta. \tag{6.3}$$

To rewrite this system the gPC expansions for $u(t, Z)$ and the random variable $\alpha(Z)$ are first established. In the expansion the Hermite polynomials $H_i(Z)$ are used since $\alpha(Z)$ follows a Normal distribution and this is the only random variable in the SDE.

$$u(t, Z) = \sum_{i=0}^{M} u_i(t)H_i(Z), \quad \alpha(Z) = \sum_{i=0}^{M} a_i H_i(Z),$$

where $a_0 = \mu$, $a_1 = \sigma$ and all other elements are 0, since $\alpha(Z)$ can be represented by $\mu + \sigma Z$. By substituting these expansions into the differential equation the following system is obtained

$$\sum_{i=0}^{M} \frac{du_i}{dt} H_i(Z) = -\sum_{i=0}^{M}\sum_{j=0}^{M} a_i H_i(Z) u_j H_j(Z).$$

Now by using a Galerkin projection that project the above equation onto the random space spanned by the polynomial basis will now be conducted. This is done by successively evaluate the inner product and exploit the orthogonality. From [2] the system then becomes

$$\frac{du_k}{dt} = -\frac{1}{\gamma_k}\sum_{i=0}^{M}\sum_{j=0}^{M} a_i u_j e_{ijk}, \quad \forall k = 0, 1, \ldots, M \tag{6.4}$$

where $\gamma_k = k!$ is the normalization factor for the Hermite polynomials and

$$e_{ijk} = \mathbb{E}[H_i(Z)H_j(Z)H_k(Z)], \quad 0 \le i, j, k \le M.$$

e_{ijk} can be found by the Hermite-gauss quadrature but in this case it can also be found exact by (from [2])

$$e_{ijk} = \frac{i!j!k!}{(s-i)!(s-j)!(s-k)!}, \quad s \ge i, j, k \text{ and } 2s = i + j + k \text{ is even}$$

If the 2 conditions are not satisfied then $e_{ijk} = 0$. The implemented function calculating e_{ijk} is listed in appendix B.2.1. The system (6.4) can be written into a vector system [2] by

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{A}^T \boldsymbol{u}$$

where the elements in $\boldsymbol{A}$ is computed by

$$A_{jk} = -\frac{1}{k!}\sum_{i=0}^{M} a_i e_{ijk}$$

and with the initial condition being $\boldsymbol{u} = \boldsymbol{b}$, where $\boldsymbol{b} = [\beta, 0, 0, \ldots]^T$.

This system is now ready to be implemented where these vectors and the matrix are build. The right hand side is different from the two other methods implemented since the system is set up to a matrix vector product. The right hand side function is implemented as

```
import numpy as np

def rhs(u,t,A):
    u = np.dot(A.T,u)
    return u
```

which is given to the time integration function `odeint` together with the initial condition vector $\boldsymbol{u}$ and the matrix $\boldsymbol{A}$. The mean and variance solution is found from the output from `odeint` by (4.8) and (4.9). All code for this test can be seen in appendix B.2.1.

In figure 6.8 the estimated and exact mean solution is shown together with the estimated standard deviation. The figure is produced for 5 expansions, so $M = 5$.

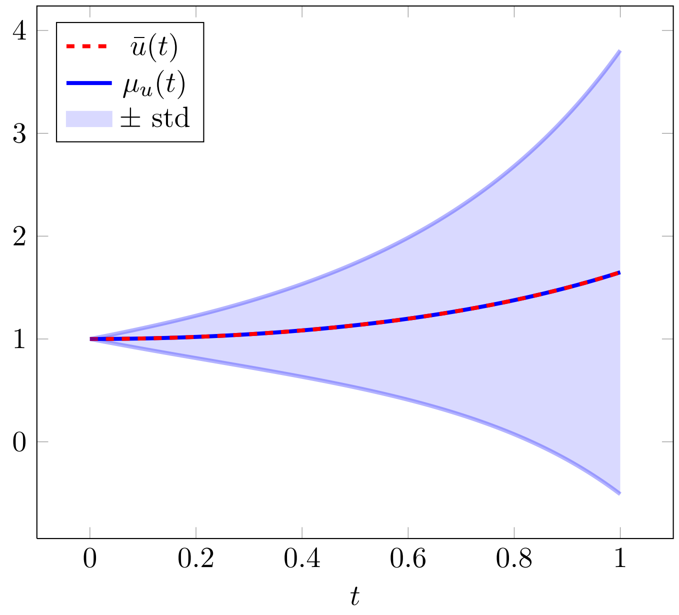

Figure 6.8: Estimated mean solution for $M = 5$ with the corresponding standard deviation for $\alpha(\omega) \sim \mathcal{N}(0, 1)$.

The SGM also produces a very fine result in this case and there seems to be no difference compared to the solution found with the SCM. Hereby the outcome from the two methods seems to be similar.

The SGM could also be used to solve the Test Equation with the $\alpha(Z) \sim \mathcal{U}(-1, 1)$ where the result will be similar to the result solved with the SCM. Instead the convergence of the SGM will be investigated in the same way as for the SCM. So by running through $M = 1, 2 \ldots, 20$ and compute the error E_M in the same way as in (6.1) figure 6.9 were produced. The actual code for this convergence test can be seen in appendix B.2.1.

In figure 6.9 spectral convergence is again obtained for both the mean and variance solutions as expected of SGM. Compared with the convergence of the SCM, where the lowest precision is around 10^{-10} for $M = 9$, the lowest precision for SGM is around 10^{-8}. The reason could be that more numerical errors enters from the vector matrix calculations. Besides that the convergence of SCM and SGM seems to be very similar.

Now the tree methods have been tried out on a very simple problem and it is interesting to try with something a bit more complicated - Burger's equation. So far it is seen that the two spectral method can estimate mean and variance solution quite precise and efficient, but some difference might appear when solving the stochastic Burger's equation.

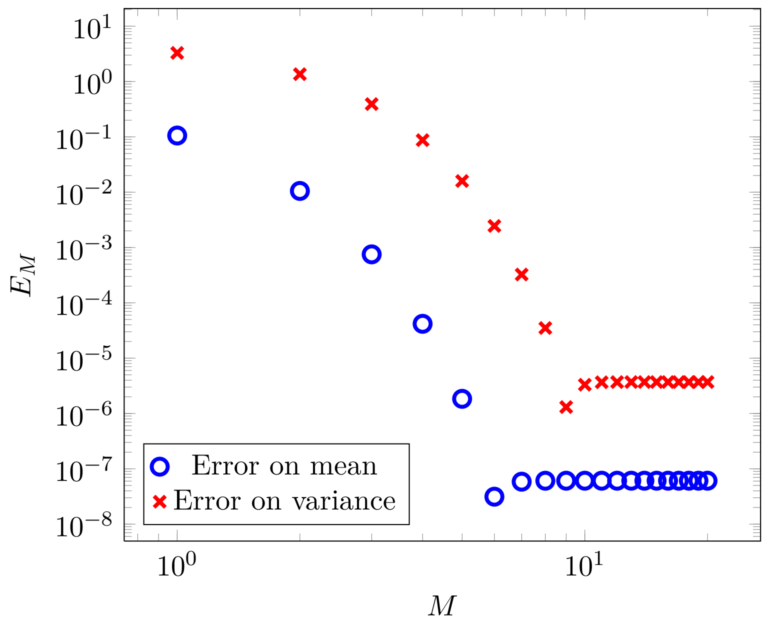

Figure 6.9: Convergence for both mean and variance for the SGM with $\alpha(\omega) \sim \mathcal{N}(0,1)$.

6.4 Stochastic Burger's equation - Monte Carlo method

The stochastic Burger's equation will first be solved by the Monte Carlo method. Before the Monte Carlo method can be applied to the problem a deterministic solver should be developed to the system. Here the Deterministic Collocation method will be used as examined in general earlier.

The Deterministic Collocation method is simple to construct for the SDE (5.13). Since the differential part can be approximated with the differentiation matrix $\boldsymbol{D}$ by (3.11)

$$\frac{d\boldsymbol{u}}{dx} = \boldsymbol{D}\boldsymbol{u}$$

where $\boldsymbol{u}$ is a $N \times 1$ vector corresponding to the Gauss-Lobatto nodes. The stochastic Burger's equation can be written into the following system

$$\frac{d\boldsymbol{u}}{dt} = -\boldsymbol{u}\left(\boldsymbol{D}\boldsymbol{u}\right) + \nu \boldsymbol{D}^2\boldsymbol{u}.$$

The initial condition will be $u(x,0) = -x$ which ensures that $u(-1,0) = 1$ and $u(1,0) = -1$, when no uncertainty are added to the boundary. Therefore the boundary condition only have to be $\frac{du}{dx}(-1,t) = 0$ and $\frac{du}{dx}(1,t) = 0$, such that the slope on the boundary points are 0.

From the above the right hand side can be established and is implemented as follows.

```
import numpy as np

def rhs(u,t,v,D,D2):
    u = -u*np.dot(D,u) + v*np.dot(D2,u)
    u[0]  = 0.0
    u[-1] = 0.0
    return u
```

where D2 are the squared differentiation matrix $\boldsymbol{D}^2$ and v is the parameter ν. The technique handling that steady state will be reached is by iteratively solving the problem for times t_i and t_{i+k} and then compare the solutions. This could be done for each time step ($k = 1$), meaning a comparison between $u(x, t_i, Z)$ and $u(x, t_{i+1}, Z)$ until the difference is below be below e.g. 10^{-6}. Then it will be assumed that steady state is reached.

Another way is to compare the solution $u(x, t_i, Z)$ and $u(t_{x,i+1000}, Z)$ and see if the difference between these two solutions is below 10^{-6}. In this way the number of loops is reduced a lot. On the other hand the steady state time t_s is possible to be 999 time steps past the steady state point (for the specified tolerance). The implementation of this iterative control is shown below, to show how it is implemented in practice.

```
# Initial tolerance, which must be too big.
tol  = 1
max_iter = 1
# Time step
dt  = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

while tol> 10**(-6) and max_iter < 10**5:
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs,u_init,t_span,tuple([nu,D,D2]))

    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol  = max(np.abs(u_init-u_check))

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init = u_check

    max_iter += 1
```

Now the deterministic part of the system is established and the stochastic part is the only thing remaining. For the Monte Carlo method it is simply just done by picking M random numbers from the desired distribution and solve the deterministic system M times. The overall implementation is shown in appendix B.2.2.

The parameters chosen to solve the stochastic Burger's equation are $\nu = 0.05$, $u(-1,t) = 1 + \delta_1(Z), \delta_1(Z) \sim \mathcal{U}[0, 0.1]$ and $u(1,t) = -1$, the number of spacial nodes $N = 80$ and the number of samples $M = 1000$. The result of this run is shown in figure 6.10.

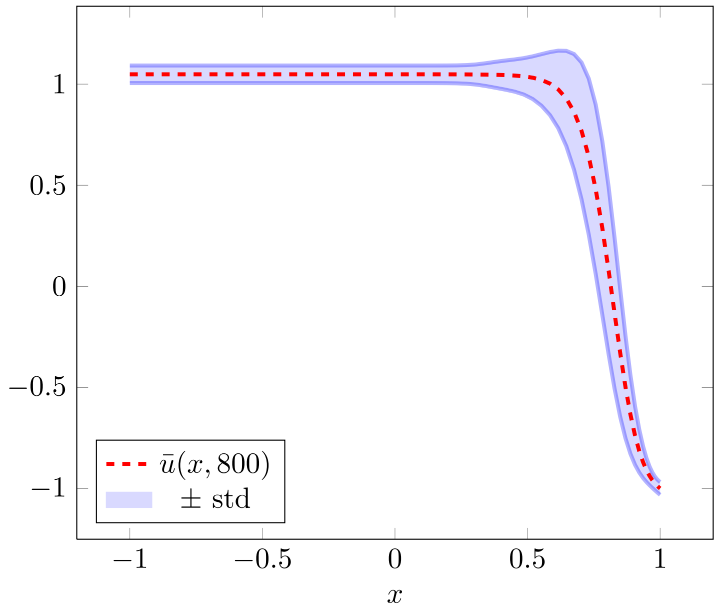

Figure 6.10: Estimated mean solution for the Burger's equation for $M = 1000$ samples with the corresponding standard deviation with uncertainty $\mathcal{U}(0, 0.1)$ on the left boundary.

For this problem the exact mean solution is not known, but by comparing the solution with the solution of the same problem in [2] it is seen to quite similar. Keeping the results from solving the Test equation by the Monte Carlo method in mind, $M = 1000$ realizations gives variations in the solutions. This means that the solution must be expecting some errors.

To analyse the result it is seen in figure 6.10 that when $\bar{u}(x, 800)$ decreases the standard deviation increases so in this area the mean solution is most uncertain. In the other areas the standard deviation is quite small. Next the exact same SDE is solved by the SCM.

6.5 Stochastic Burger's equation - Stochastic Collocation method

The exact same problem shall also be solved by the SCM. The implementation is based on the same spatial solver (the Deterministic Collocation method) as used before. The difference is the way the random variable ($\delta_1(Z)$) is represented. Since $\delta_1(Z)$ is uniform distributed the Legendre polynomials are used to represent the random variable. This is done by the Legendre-gauss quadrature which gives the abscissas $\boldsymbol{z}$ and weights $\boldsymbol{w}$ on the interval $[-1, 1]$. For this problem these abscissas have to be scaled to the desired interval $[0, 0.1]$ by using the transformation (6.2).

For each abscissa $z^{(i)}$ the deterministic solutions could be determined by running a for-loop around the deterministic solver. Another way to determine all deterministic systems is to build one large system by the Kronecker product. This means that the system is repeated with the number of realizations M (the number of abscissas in $\boldsymbol{z}$). In short notation the system is written as

$$\frac{d\boldsymbol{u}}{dt} = -\boldsymbol{u}_e\boldsymbol{D}_e\boldsymbol{u}_e + \nu\boldsymbol{G}_e\boldsymbol{u}_e$$

where $\boldsymbol{u}_e$ is a $MN \times 1$ vector and $\boldsymbol{D}_e$ and $\boldsymbol{G}_e$ are $MN \times MN$ matrices given by

$$\boldsymbol{u}_e = \begin{bmatrix} u(x_1, t, z^{(1)}) \\ u(x_2, t, z^{(1)}) \\ \vdots \\ u(x_N, t, z^{(1)}) \\ u(x_1, t, z^{(2)}) \\ \vdots \\ u(x_N, t, z^{(M)}) \end{bmatrix}, \quad \boldsymbol{D}_e = \begin{bmatrix} \boldsymbol{D} & & & \\ & \boldsymbol{D} & & \\ & & \ddots & \\ & & & \boldsymbol{D} \end{bmatrix}, \quad \boldsymbol{G}_e = \begin{bmatrix} \boldsymbol{D}^2 & & & \\ & \boldsymbol{D}^2 & & \\ & & \ddots & \\ & & & \boldsymbol{D}^2 \end{bmatrix}$$

For $M = 5$ and $N = 80$ this system is implemented and the mean solution and the standard deviation solution for the stochastic Burger's equation are solved to each time steps by `odeint`. The results are shown in figure 6.11 together with the 5 individual deterministic solution for each $z^{(i)}$. It should be mentioned that the solutions are shown for $t = 800$ where the steady state is reached. The code is listed in appendix B.2.2.

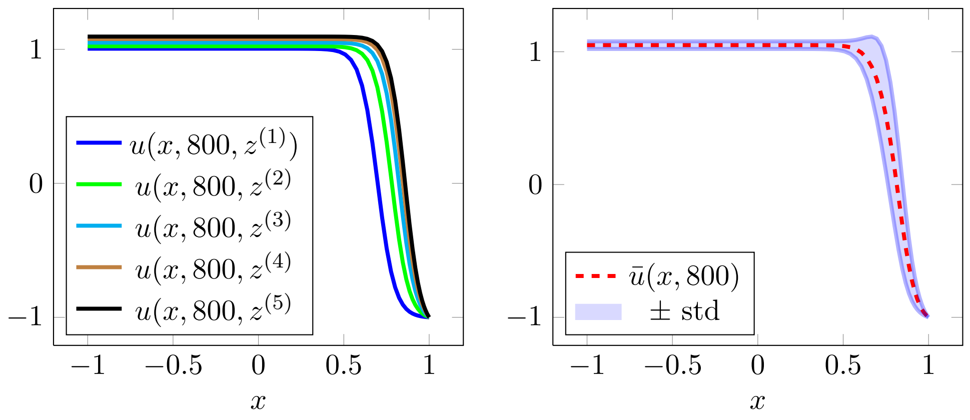

Figure 6.11: Left is the $N = 5$ deterministic solutions for the 5 different $\delta \sim \mathcal{U}(0, 0.1)$. To the right the corresponding estimated mean solution and the estimated standard deviation.

For only $M = 5$ the mean solution is determined with at least the same precision as the Monte Carlo method. This means that a reduction of solving approximately 1000 deterministic system and in the same time achieve, most likely, more precise solutions. It shows the strength of the SCM. To the left the 5 deterministic solutions is shown that illustrates how the random space is spanned.

6.6 Stochastic Burger's equation - Stochastic Galerkin method

The SGM will as the last method being used to solve the stochastic Burger's equation. As for the Test equation some manipulations of the system first have to be made before the implementation can be conducted.

Starting from the stochastic Burger's equation given in (4.8) and by exploitation of the gPC expansion with the Legendre polynomials $L_i(Z)$, since the uncertainty is uniform distributed by

$$u(x, t, Z) = \sum_{i=0}^{M} u_i L_i(Z).$$

By insertion of this expansion into the differential equation the following equation is obtained.

$$\sum_{i=0}^{M} \frac{\partial u_i}{\partial t} L_i(Z) = -\sum_{i=0}^{M}\sum_{j=0}^{M} u_i L_i(Z) \frac{\partial u_j}{\partial x} L_j(Z) + \nu \sum_{i=0}^{M} \frac{\partial^2 u_i}{\partial x^2} L_i(Z).$$

On this system the Galerkin projection is performed by taking the inner product between each term and $L_k(Z), k = 0, 1, \ldots, M$. By taking the orthogonality into account the system ends up with (from [2])

$$\frac{\partial u_k}{\partial t} = -\frac{1}{\gamma_k} \sum_{i=0}^{M}\sum_{j=0}^{M} u_i \frac{\partial u_j}{\partial x} e_{ijk} + \nu \frac{\partial^2 u_k}{\partial x^2}, \quad k = 0, 1, \ldots, M.$$

This time the differentiated parts are approximated with finite difference stencils where they will be approximated by [4]

$$
\begin{aligned}
\frac{\partial u}{\partial x}(x_i,t) &= \frac{u(x_{i+1},t)-u(x_{i-1},t)}{2\Delta x},\\
\frac{\partial^2 u}{\partial x^2}(x_i,t) &= \frac{u(x_{i+1},t)-2u(x_i,t)+u(x_{i-1},t)}{(\Delta x)^2}.
\end{aligned}
$$

In short form notation the whole system will be as below

$$
\frac{\partial u_k}{\partial t} = -\frac{1}{\gamma_k}\boldsymbol{U}_I\boldsymbol{B}_k\boldsymbol{u}_{vec} + \nu\boldsymbol{A}\boldsymbol{u}_k + \boldsymbol{g}_k, \quad k=0,1,\dots,M,
$$

where the matrices and vectors are defined as below. The indexes $u_{i,j}$ refers to the i'th spatial point and the j'th expansion.

$$
\boldsymbol{u}_k = \begin{bmatrix} u_{1,k}\\ u_{2,k}\\ \vdots\\ u_{N,k}\end{bmatrix} \qquad
\boldsymbol{u}_{vec} = \begin{bmatrix} u_{1,0}\\ u_{2,0}\\ \vdots\\ u_{N,0}\\ u_{1,1}\\ u_{2,1}\\ \vdots\\ \vdots\\ u_{N-1,M}\\ u_{N,M}\end{bmatrix}, \quad
\boldsymbol{g}_k = \begin{bmatrix} \frac{1}{2\Delta x\gamma_k}\sum_{i=0}^{M}\sum_{j=0}^{M} u_{0,j}u_{1,i}e_{ijk} + u_{0,k}\frac{\nu}{(\Delta x)^2}\\ 0\\ \vdots\\ 0\\ -\frac{1}{2\Delta x\gamma_k}\sum_{i=0}^{M}\sum_{j=0}^{M} u_{N+1,j}u_{N,i}e_{ijk} + u_{N+1,k}\frac{\nu}{(\Delta x)^2}\end{bmatrix}
$$

$$
\boldsymbol{U}_I = \begin{bmatrix} u_{1,0} & 0 & & \cdots & u_{1,M} & 0 & \\ 0 & u_{2,0} & & & 0 & u_{2,M} & \\ & & \ddots & & & & \ddots & \\ & & & u_{N,0} & \cdots & & & u_{N,M}\end{bmatrix}
\boldsymbol{A} = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & \ddots & \ddots & \ddots \\ & & & 1 & -2\end{bmatrix}
$$

$$
\boldsymbol{B}_k = \begin{bmatrix}
\begin{bmatrix} 0 & e_{0,0,k} & & \\ -e_{0,0,k} & 0 & \ddots & \\ & \ddots & \ddots & e_{0,0,k}\\ & & -e_{0,0,k} & 0\end{bmatrix} & \cdots & \begin{bmatrix} 0 & e_{0,M,k} & & \\ -e_{0,M,k} & 0 & \ddots & \\ & \ddots & \ddots & e_{0,M,k}\\ & & -e_{0,M,k} & 0\end{bmatrix}\\
\vdots & \ddots & \\
\begin{bmatrix} 0 & e_{M,0,k} & & \\ -e_{M,0,k} & 0 & \ddots & \\ & \ddots & \ddots & e_{M,0,k}\\ & & -e_{M,0,k} & 0\end{bmatrix} & & \begin{bmatrix} 0 & e_{M,M,k} & & \\ -e_{M,M,k} & 0 & \ddots & \\ & \ddots & \ddots & e_{M,M,k}\\ & & -e_{M,M,k} & 0\end{bmatrix}
\end{bmatrix}
$$

The sizes of the matrices are that $\boldsymbol{u}_k$ is a $N \times 1$ vector, $\boldsymbol{u}_{vec}$ is a $NM \times 1$ vector, $\boldsymbol{U}_I$ is a $N \times NM$ matrix, $\boldsymbol{A}$ is a $N \times N$ matrix, $\boldsymbol{B}_k$ is a $NM \times NM$ matrix and $\boldsymbol{g}_k$, which handles the boundary conditions, is a $N \times 1$ vector.

This system will be tested with $N = 38$, $M = 5$, $\nu = 0.05$ and the initial condition $u(x, 0, Z) = -x$. The entire implementation can be seen in appendix B.2.2. The mean and standard deviation solution are shown i figure 6.12.

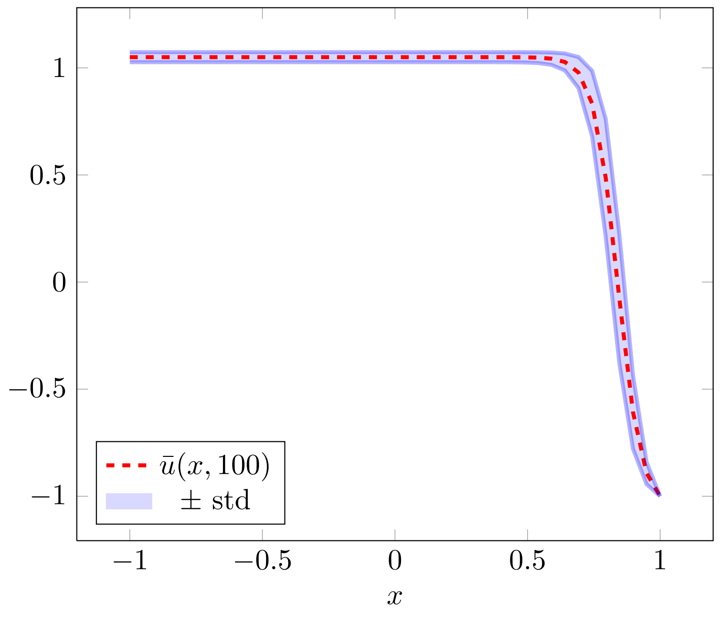

Figure 6.12: Mean and standard deviation solution for SGM with $\delta_1(Z) \sim \mathcal{U}[0, 0.1]$ and with $M = 5$.

Compared with the solutions solve by SCM the solutions of the SGM are not the same. The reason for the difference must be a small error in the implementation which not was found. Even though the solutions are wrong, they still have the same trends and the correct solutions seems to be estimated by some scaling. This could illustrate that the implementation of SGM a least not is easy to implement.

6.7 Comparison between the UQ methods

A comparison and evaluation of the 3 different UQ methods based on the tests done in the previous sections but also from the discussion in [2] will be presented. This will end with a selection of 1 of the UQ methods for the further experiments in this projects.

It is obvious from the tests that the Monte Carlo method is not preferable since it is very inefficient. Already for the simplest test example it was very time-consuming. For complex SDE's the time consumption will be extremely huge. However, it is a very intuitive and relatively easy method to implement and use and have therefore been used as a reference for the correctness of the other methods.

The SCM is compared to SGM easy to implement as long as the deterministic solver is implemented for the problem. The applicability of SCM is not affected of the complexity or non-linear terms in the differential equation which is handled by the deterministic solvers [2]. The efficiency for the SCM was for these simple problems extremely good, but also the precision was very satisfactory.

The Stochastic Galerkin Method (SGM) is on the other hand not trivial to implement and depend very much on the complexity of the differential equation. The experience from the implementation of the SGM method for the Burger's equation were time-consuming and is guaranteed not optimal implemented with respect to efficiency. The convergence is very similar to the convergence for the SCM.

All this indicates that the SCM is the preferable method, but in [2] it is explained that the accuracy of the SGM is better than SCM for multidimensional random spaces ($d > 1$) because of the so called aliasing error which is introduced by the interpolation. If the same precision have to be achieved for the multidimensional random space then higher order of orthogonal polynomials is needed for the SCM compared to the SGM [2].

The choice of method therefore depends on different factors but overall the SGM method preferable for differential equation with multiple random variables and the implementation can be done. For few random variables or for a very complex differential equation the SCM is preferable.

Taking all this into account and the experience working with the methods the SCM will be the selected method for further experiments mainly because that it is easier to implement and that the number of variables in the further experiments not will be extremely large.

Chapter 7

Topics in Uncertainty Quantification

In this part of the project some of the research areas and topics in UQ will be presented. So far in this project the problems have been quite simple. This part will illustrate the terms of the systems which causing the problems.

So far the solved systems (stochastic Test equation and stochastic Burger's equation) have only included $d = 1$ random variable into the SDE. This keep the systems relatively small and relatively easy solved by the UQ methods. As described in [11] it is possible to have practical stochastic problem with more than hundred random variables $d > 100$ in addition to the traditional space and time dimensions. This lead to a huge or almost impossible amount of work if such a problem is solved by the presented methods.

To illustrate the amount of work in the high dimensional case lets say that d random variable exist in a given SDE and each of these are represented in m nodes. From [11] the number of times the deterministic solver have to be performed is

$$M = m^d.$$

If it is assumed that there exist $d = 10$ random variables, all represented with lets say $m = 5$ nodes leading to $5^{10} = 9,765,625$ times the deterministic system have to be solved. The number grows extremely fast as the number of random variables increases.

This have naturally resulted in a research and developing of method which makes it possible to solve systems with several random variables solved in a reasonable time horizon. There is still a limit in how large d can be by using SCM [12] and there is an ongoing research for pushing to this limit. Below are techniques handling high dimensional random space in SDE presented very shortly to give a brief overview of the topics. References are given if more details are desired.

ℓ_1-minimization

This short description is based on [11]. The overall idea comes from the generation of a underdetermined system and from this system the goal is to find the computational cheapest solution by ℓ_1-minimization. The underdetermined system is often encountered for high dimensions ($d >> 1$). The base is taking in the gPC expansion, for the Stochastic Collocation method, used to represent the target function $u(t, \boldsymbol{x}, \boldsymbol{Z})$ by

$$u(t, \boldsymbol{x}, \boldsymbol{Z}) = \sum_{|\boldsymbol{i}|<K} \hat{\boldsymbol{u}}_{\boldsymbol{i}} \boldsymbol{\Psi}_{\boldsymbol{i}}(\boldsymbol{Z})$$

with the multi-index $\boldsymbol{i}$ explained in gPC section 3.6. (In the following the single-index l will be used which runs through a ordered scheme for the multi-index). This can be rewritten into the linear system [11]

$$\boldsymbol{V}\hat{\boldsymbol{u}} = \boldsymbol{f}$$

where $\hat{\boldsymbol{u}}$ containing the K (highest polynomial degree of Ψ_l) coefficients. $\boldsymbol{f}$ is the sample of the deterministic solutions $u(t, \boldsymbol{x}, \boldsymbol{Z}^{(m)})$ and the matrix $\boldsymbol{V} = v_{m,l}$ is a Vandermonde-type interpolation matrix ([11]) given as

$$v_{m,l} = \boldsymbol{\Psi}_l(\boldsymbol{Z}^{(m)}), \quad m = 1, \ldots, M, \quad l = 1, \ldots, K.$$

For $M > K$ the underdetermined system is obtained which means that the sample of deterministic solution M is larger then the total polynomial degree M. For a big difference between these two numbers the system becomes severely underdetermined and here the ℓ_1-minimization can be used to find the solution by

$$\min \|\hat{\boldsymbol{u}}\|_1 = \sum_{l=1}^{M} |\hat{u}_l|, \quad \text{subject to} \quad \boldsymbol{V}\hat{\boldsymbol{u}} = \boldsymbol{f}.$$

This is the convex relaxation of $\min \|\hat{\boldsymbol{u}}\|_0 := \#\{l : \hat{u}_l \neq 0\}$ finding the coefficient vector containing most zeros and hereby the most reduced.

ANOVA decomposition

The abbreviation ANOVA comes from the statistical method 'analysis of variance'. The main idea behind this method according to the high dimensional problems ($d >> 1$) is to split the multi-dimensional random space into smaller subdimensions [13]. This way of decomposing the dimensionality is a effective way to break the curse of dimensionality for certain systems by solving several low-dimensional systems. However, if the ANOVA decomposition still results in a component with relatively high dimensionality nothing is gained with respect to the computational complexity [13].

The ANOVA decomposition method can be combined with the Multi-Element Probabilistic Collocation Method (MEPCM) in order to be able to solve additional problems [13]. The MEPCM solves each component of the ANOVA decomposition efficiently and the total statistics can be obtained from these subsolutions. To read more see [13].

Space Grid Collocation

This section is based on [2] and [14]. A third method to handle the *curse of dimensionality* is the sparse grid method also called the Smolyak sparse grid. The sparse grid method can be used in the multivariate case to reduce the computational cost and keep the approximate same precision. The reduction of computational cost from the reduction of deterministic system needed to be solved. Hence the sparse grid method can only be used for the SCM out of the introduced methods. Therefore sparse grid can overcome the *curse of dimension* to a certain extent.

The sparse grid method is a sparse quadrature rule where fewer abscissas and weights are obtained in correspondence to the original quadrature rules. Therefore there also here exists several sparse quadrature e.g. gauss-Legendre, Clenshaw Curtis and gauss-Hermite (see [14]) which corresponds to the different distributions in the same way as described earlier. Because of the fewer nodes and weights the number of deterministic systems to be solved decreases and hence a reduction of the total dimension.

A more theoretical description will be presented later in the thesis, but first the Tensor Product Collocation will be introduced which uses the full tensor grid.

This chapter have illustrated shortly the ongoing research that have been the past years and further some difference approaches to handle the *curse of dimensionality.* Overall these methods are relatively complex and demands a strong theoretical understanding, so at this point it might be abstract methods. Hence the next chapter will in some way illustrate the *curse of dimensionality* and finally use the sparse grid based on gauss-Legendre quadrature to illustrate it is beneficial.

Chapter 8

Multidimensional problems

As highlighted in the study of the literature the *curse of dimensionality* is the main big topic of today's research. The systems with relatively many random variables results in huge systems which are very time consuming. In this chapter this is illustrated and further some tools to handle the complexity will be introduced. These tools will enable the systems to have several random variables, but not systems which contains a huge amount of random variables.

Based on chapter 6 (where the systems only contained 1 random variable) the used method will be the SCM. First the Tensor Product Collocation will be introduced and afterwards the solutions for the stochastic Test equation and stochastic Burger's equation will be shown to verify and illustrate the effects of the Tensor Product Collocation and to illustrate the *curse of dimensionality.*

8.1 Tensor Product Collocation method

Recall the general system (2.1) now with $d > 1$. In other words there will exist at least 2 random variables having the notation $\boldsymbol{Z} = \{Z_1, \dots, Z_d\}$, where the i'th random variable will be represented by $\Theta_i = \{z_i^{(1)}, \dots, z_i^{(m_i)}\}$ which is m_i long.

Section 4.1.2 describes how the mean solution is determined with use of interpolation based on Lagrange polynomials $h_i(x)$. In the multivariate case the total interpolation is a tensor product between the one-dimensional interpolations which is notated by

$$\mathcal{I}^i(u) = \sum_{k=1}^{m_i} u(z_i^{(k)}) h_k(Z_i) \qquad i = 1, 2, \dots, d.$$

The set of the these interpolation can by [9] be formulated via the tensor product

$$\boldsymbol{I}(u) = (\mathcal{I}^1 \otimes \dots \otimes \mathcal{I}^d)(u) = \sum_{k_1=1}^{m_1} \cdots \sum_{k_d=1}^{m_d} u(z_1^{(k_1)}, \dots, z_d^{k_d})(h_{k_1}(Z^{(1)}) \otimes \dots \otimes h_{k_d}(Z^{(d)})) \tag{8.1}$$

By the same procedure as in section 4.1.2 (using the quadrature rule) the estimated mean solution is solved from [9]

$$\bar{u}(\boldsymbol{x},t) = \sum_{k_1=1}^{m_1} \cdots \sum_{k_d=1}^{m_d} u\Big(z_1^{(k_1)},\ldots,z_d^{(k_d)}\Big)\Big(\rho(z_1^{(k_1)})w_1^{(k_1)} \cdots \rho(z_d^{(k_d)})w_d^{(k_d)}\Big). \tag{8.2}$$

This shows that all combinations between the weights for each random variable and coefficients for every combination of the nodes $z_i^{(j)}$. The corresponding estimated variance solution can from this mean expression also be determined as in previous sections by $s_u^2 = \mathbb{E}[u^2] - (\mathbb{E}[u])^2$.

The number of nodes to represent the random space is $M = m_1 m_2 \cdots m_d$. If it is assumed that all random variables is represented with the same number of nodes $m_1 = m_2 = \cdots = m_d = m$ then the total number of nodes is $M = m^d$. This number will of course grow very fast if the number of random variables increases and/or m increases. M is desired to be small because the deterministic system has to be solved M times.

The practical implementation of the multi-dimensional Tensor Grid Collocation will be illustrated in context with the visualization of the results in the next section.

8.1.1 Test of Tensor Grid Collocation method

The Test equation will here consist of two random variables ($d = 2$), where the parameter in the differential equation $\alpha(Z_1)$ and the boundary condition $\beta(Z_2)$ both will contain uncertainty. The SDE will take the following form

$$\frac{du(t,\boldsymbol{Z})}{dt} = \alpha(Z_1)u(t,\boldsymbol{Z}), \quad u(0,\boldsymbol{Z}) = \beta(Z_2), \quad t \geq 0$$

The parameter $\alpha(Z_1)$ will in this example follow a $\mathcal{N}(0,1)$ while the boundary condition $\beta(Z_2)$ will follow $\mathcal{N}(1,0.1^2)$ and both variables will be represented by $m_1 = m_2 = 6$ nodes. This leads to $M = m_1^2 = 36$ deterministic systems, which have to be solved in order to determine the mean and variance solution. The nodes from the two distributions (Θ_1,Θ_2) and the corresponding weights $(\boldsymbol{w}_1,\boldsymbol{w}_2)$ are determined from the Hermite-gauss quadrature. The nodes are then transformed to the desired interval by

$$\begin{aligned} \boldsymbol{z}_1 &= 0 + 1 \cdot \Theta_1, \\ \boldsymbol{z}_2 &= 1 + 0.1 \cdot \Theta_2. \end{aligned}$$

To solve the 36 different deterministic differential equations all the the combinations of the elements in $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ have to be constructed. To do this a python function called `cartesian` [15] is used which makes all possible combinations between all elements in the d vectors, which is given as inputs to the function. The same function is also used to make all the combinations between the weights. The code for this are shown below.

```
# Create all combinations of the nodes
nodes = cartesian((Theta_1,Theta_2))
```

```
# Split into two different vectors
A = nodes[:,0]
B = nodes[:,1]

# Create all combinations of the weights
W_all = cartesian((w1,w2))

# Split into two different vectors
W1 = W_all[:,0]
W2 = W_all[:,1]

# Multiple all the combinations of weights
W = W1*W2
```

Note that the output from the `cartesian` function is a matrix with the i'th columns being different repetition of e.g. w_i. In this test case `A`, `B` and `W` have the size of 36 fitting the number of deterministic systems that should be solved. On this form the vectors `A` and `B` can be given to the time-integration solver `odeint` together with the right-hand side and the initial condition and hereby the solutions for the 36 deterministic differential equations can be obtained. The complete code can be seen in appendix B.3.1.

The estimated mean is computed by (8.2) and can be compared with the exact mean solution determined analytical to (**??**). The final solution by running this test can be seen in figure **??**.

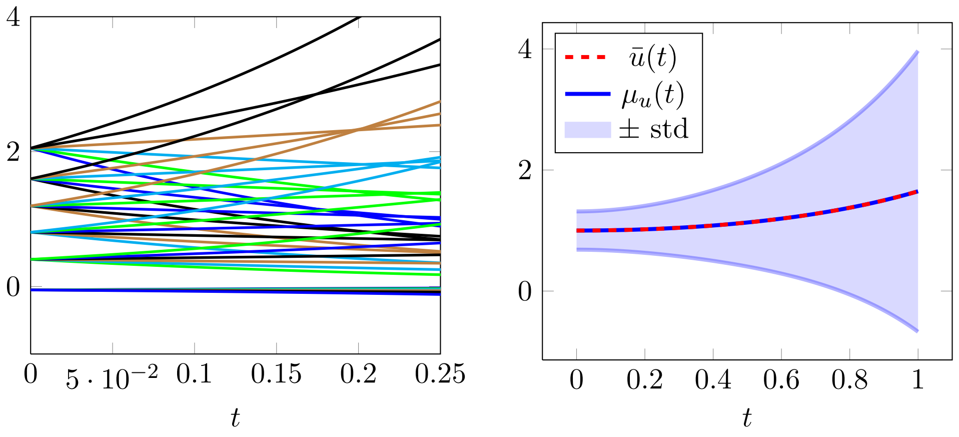

Figure 8.1: Left: The 36 deterministic solutions for $0 \leq t \leq 0.25$. Right: The estimated and the exact mean solution together with the estimated standard deviation.

Left in figure (**??**) it is illustrated how the different deterministic solutions are distributed - like spanning the random space. At $t = 0$ the 6 nodes to represent the uncertainty on the boundary are apparent. From each of these nodes 6 solutions take their base corresponding to the 6 nodes representing $\alpha(Z_1)$. This gives all together the 36 different deterministic solutions which makes the base of the statistics shown to the right. Compared with figure 6.2 the big difference is seen on the boundary ($t = 0$) where the standard deviation already is of a certain level. The standard deviation does not differ much otherwise. It is only slightly larger.

It is interesting to find out if the TPC method has an effect on the convergence, so an equivalent convergence test as performed earlier will be done. It takes the exact same parameters as above but will be running through the total number of nodes $M = m_1 m_2$ with $m_1 = m_2 \in [2, 3, \ldots, 20]$. This implementation is listed in appendix B.3.2. In figure 8.2 the convergence are shown for both the mean and variance solution.

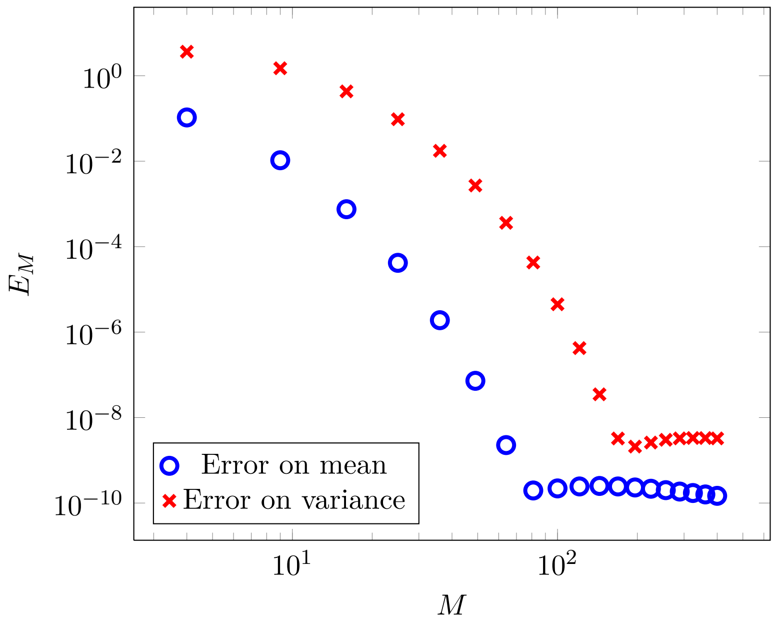

Figure 8.2: Convergence for both mean and variance solution for the SCM with $\alpha(Z_1) \sim \mathcal{N}(0, 1)$ and $\beta(Z_2) \sim \mathcal{N}(1, 0.1^2)$.

The convergence process is exactly the same as in figure 6.7. The only difference is the M axis. For $M = 81$ the mean have reached an optimal precision with respect to the mean solution. In the convergence process with only 1 random variable this point is at $M = 9$. The relationship between these two fits very well as $9^2 = 81$. This means that the representation of a single random variable does not increase with d - for this case. This result should therefore only be used as an indication.

8.1.2 Test with 3 random variables

To illustrate the *curse of dimensionality* the stochastic Burger's equation will be solved containing three random variables - two on the boundary and one describing the parameter ν in (5.13). The three random variables will in the test be the following

$$\begin{aligned} \delta_1(Z_1) =& \delta_2(Z_2) = \mathcal{U}[0, 0.1], \\ \nu(Z_3) =& \mathcal{N}[0.2, 0.001]. \end{aligned}$$

This means that the two boundary conditions will take the following form

$$
\begin{aligned}
u(-1, x, \boldsymbol{Z}) &= 1 + \delta_1(Z_1), \\
u(1, x, \boldsymbol{Z}) &= -1 + \delta_2(Z_2).
\end{aligned}
$$

The abscissas and weights are found by Legendre-gauss quadrature for the boundary condition and by Hermite quadrature for $\nu(Z_3)$. In the same way as for the Test equation all the combinations between the abscissas and between the weights are found by `cartesian`. The number of abscissas to represent the 3 random variables will be $M = m^3$ (assuming $m_1 = m_2 = m_3$). For Burger's equation a space dimension also exist which is represented by N nodes.

Therefore M deterministic differential equations have to be solved each with the differentiation matrix $\boldsymbol{D}$ of size $N \times N$. Hence the total system to be solved, constructed by the Kronecker product, consists of $MN \times MN$ matrices. The number of nonzero elements in this matrix will be around $nnz = NNM$ out of N^2M^2 elements.

This gives occasion to consider sparse matrices to decrease the computational cost. In `Python` this is simply done by importing a sparse package where the creation of the sparse matrices and corresponding matrix multiplication function for sparse matrices are included. The right hand side therefore is rewritten to

```
import scipy.sparse as sparse

def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u
```

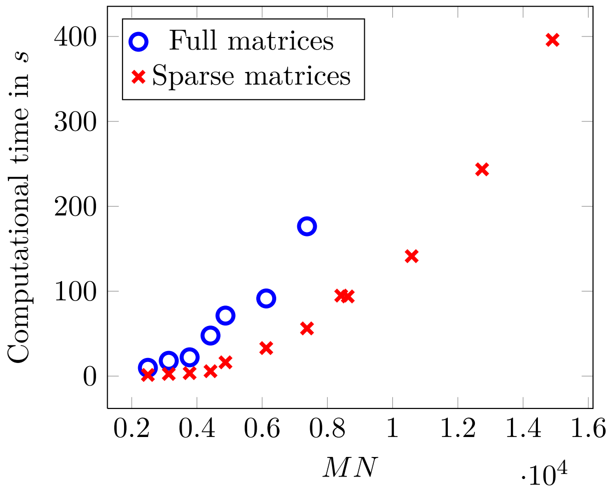

Figure 8.3: Computational time in seconds solving the stocastic Burger's equation with 3 with random variables for different sizes ($MN \times MN$) of both full and sparse differentiation matrices.

To show the benefit by using sparse matrices instead of full matrices a combinations of $N = [40, 50, 60, 70]$ and $M = [3^3, 4^3, 5^3] = [27, 64, 125]$ are conducted and the solution times is obtained for the desired combinations. The entire implementation of this sparse test is shown in appendix B.3.3

It is clear from figure 8.3 that sparse matrices reduces the computation time as expected. It should be noticed that there are fewer points for the full matrices because full matrices larger than around 7500×7500 results in memory error. This means yet another advantage by using sparse matrices instead of full matrices.

So by using the sparse matrices it is possible to solve Burger's equation with $M = 125$ and $N = 60$. For these numbers the estimated mean and standard deviation solution are determined and are shown for $t = 500$ in figure 8.4. The implementation to produce this results is given in appendix B.3.4.

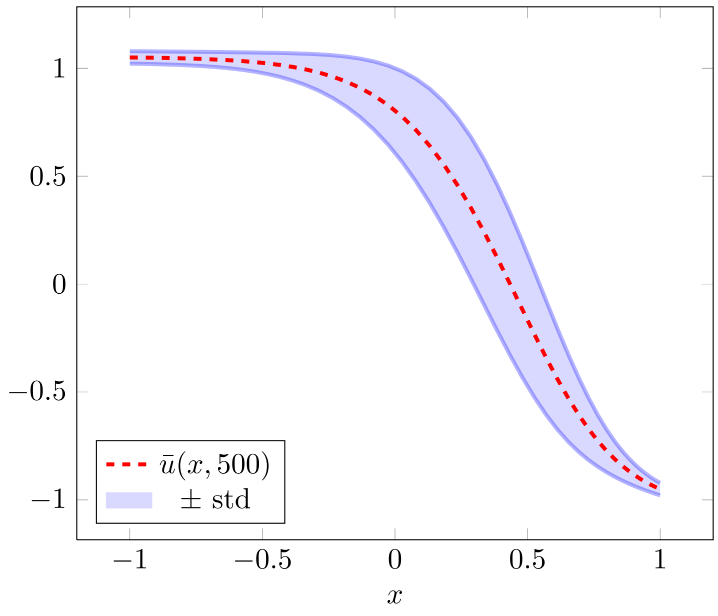

Figure 8.4: Mean and standard deviation solution for the stochastic Burger's equation with $\delta_1(Z_1) = \delta_2(Z_2) \sim \mathcal{U}[0, 0.1]$ and $\nu(Z_3) \sim \mathcal{N}[0.2, 0.001]$.

The solution can not directly be compared with the previous solutions of the stochastic Burger's equation, since the parameter ν (which having affect on the slope of the solution) takes higher values. The reason why the mean of $\nu(Z_3)$ is not chosen to be 0.05 is that smaller values will make the solution less smooth which the time-integration function `odeint` cannot handle.

About the standard deviation solution it is not increasing that much when in taking into account that there are uncertainty on 3 parameters. However, the variance on $\nu(Z_1)$ is in this case not that big, so other choices will probably increase the area of the standard deviation.

Introducing 3 random variables into Burger's equation has shown that the computational work grows massively and even when sparse matrices is used larger systems are almost impossible to solve with this set up. To solve larger systems smarter methods need to be taking into account.

8.2 Sparse Grid Collocation

In order to be able to compute SDE's even faster but also to make it possible solve larger systems (higher d or higher dimensions of $\boldsymbol{x}$) a sparse grid method now will be introduced also called Smolyak sparse collocation. It takes base from the short description in the literature study (chapter 7). After the basic theory some examples will be shown in order to illustrate the efficiency but also to illustrate that the precision is kept. First shortly the theoretical background.

The Sparse Grid Collocation (SGC) takes base in the tensor product presented in (8.1) but SGC (or from know on sparse grid) will only be a subset of the full tensor grid. From [2] the construction of the sparse grid is

$$\sum_{l-d+1\leq|\boldsymbol{i}|\leq l}(-1)^{l-|\boldsymbol{i}|}\cdot\binom{d-1}{N-|\boldsymbol{i}|}\cdot\left(\mathcal{I}^{i_1}\otimes\cdots\otimes\mathcal{I}^{i_d}\right), \tag{8.3}$$

where $l \geq d$ is an integer corresponds to the level of the construction. To specify further it means that the level l can be at most the maximum order of the 1 dimensional polynomials $\Phi(Z)$. If e.g. the order for two polynomials are 4 and 5 then l can maximum take the value 5.

The expression (8.3) is rather complex but in short words it is a combination of subsets of the full tensor grid. This can be expressed (from [2]) as the nodal set by

$$\theta_M = \bigcup_{l-d\leq|\boldsymbol{i}|}\left(\Theta^{i_1}\times\cdots\times\Theta^{i_d}\right).$$

This shows the collection of subset of the full tensor grid. Depending on the choice of the Gauss quadrature there are multiple sparse grid constructions which have different accuracy and efficiency. The different choices are mentioned earlier. The sparse grid construction that will be used here is the one based on the Legendre-Gauss quadrature. This sparse grid quadrature is not the most effective but has a better accuracy than most of the others. In the next section the sparse grid will be used to illustrate their effects.

8.2.1 Test of Sparse Grid Collocation

From the theory it is illustrated that the sparse grid have fewer points compared with the corresponding to the full tensor grid constructed by the Tensor Product Collocation method. In the literature study some of these were highlighted. In this test section the sparse grid will be based on Legendre-Gauss quadrature. All the implementations for this section are listed in appendix B.4.

An example on the difference between the full tensor grid and the sparse grid can be shown by a spanned random space with $d = 2$. For the full tensor grid each random variable will be represented with $m = 9$ where the corresponding level $l = 2$. The two grids are illustrated in figure 8.5.

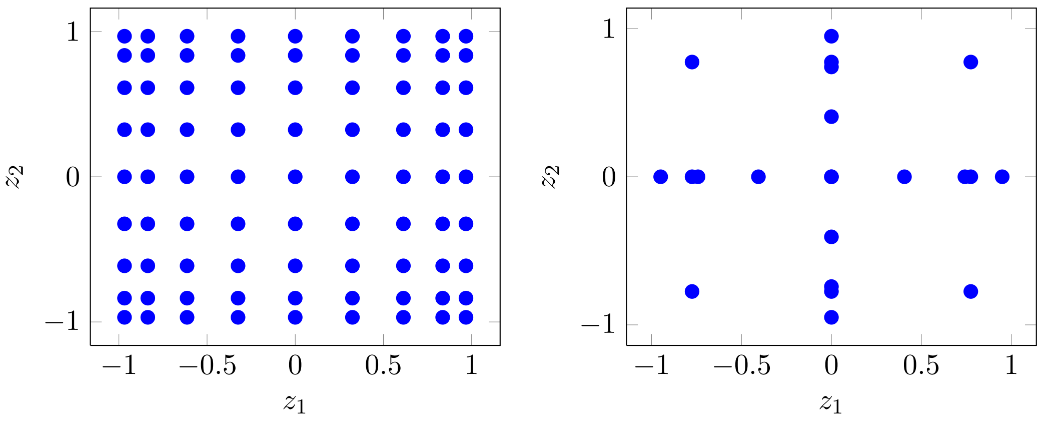

Figure 8.5: Left: The full grid with $m_1 = m_2 = 9$ with totally 81 points. Right: The sparse grid with corresponding level $l = 2$ with totally 21 points.

For each combination of the representations (of the random variables) z_1 and z_2 a deterministic solution has to be solved. Hence 81 deterministic systems have to be solved for the full tensor grid while only 21 are determined for the sparse grid. Therefore the sparse grid always will be more efficient in relation to the corresponding full tensor grid. The question is what effect the sparse grid will have on the precision.

Before the investigation of the precision the number of grid points for different dimensions will to be illustrated. This is shown in table 8.1 where the number of nodes for the sparse grid is found by using the code in [16]. The tables illustrates very well

$m\backslash d$	1	2	3	4
1	1	1	1	1
3	3	9	27	81
9	9	81	729	$6,561$
23	23	529	$12,167$	$279,841$
53	53	$2,809$	$148,877$	$7,890,481$

$l\backslash d$	1	2	3	4
0	1	1	1	1
1	3	5	7	9
2	9	21	37	57
3	23	73	159	289
4	53	225	597	$1,265$

Table 8.1: Left: The number of nodes in the random space using full tensor grid. Right: The number of nodes in the random space using sparse grid.

that the number of points are reduced drastically by using the sparse grid method. It should be mentioned that m increases in the table such that it corresponds to the level l which is seen from the $d = 1$ columns. In this columns the number of nodes are the same and hence there is nothing to gain in the one dimensional cases using sparse grid. If the precision is kept for the sparse grid it will enable to determine much larger systems.

Hence the stochastic Burger's equation with $d = 3$ random variables will be solved both for the full tensor grid and the sparse grid. For convenience, all the 3 random variables will follow a uniform distribution as the following

$$\begin{aligned}\delta_1(Z_1) =&\delta_2(Z_2) = \mathcal{U}[0, 0.1],\\ \nu(Z_3) =&\mathcal{U}[0.05, 0.35],\end{aligned}$$

according to the general system (5.13). For the full tensor product the number of representation of the random variables is $m_1 = m_2 = m_3 = 5$ leading to $M = 125$ nodes. For the sparse grid method the level is chosen to be $l = 2$ which from table 8.1 corresponds to $M = 37$ nodes. These two node representations are shown in figure 8.6.

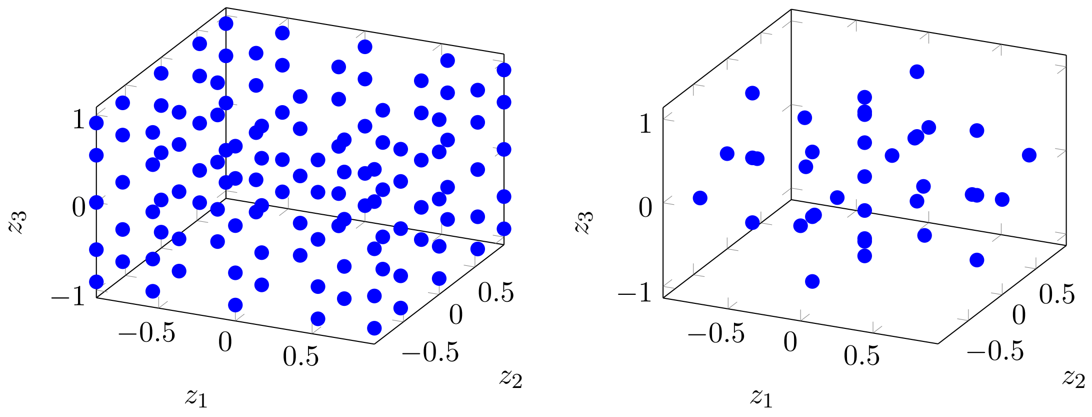

Figure 8.6: Left:Full tensor grid containing $M = 125$ nodes. Right: Sparse grid containing $M = 37$ nodes corresponding to level $l = 2$.

This illustrates further that as d grows the computational cost is greatly reduced. The level $l = 2$ corresponds to the full tensor grid where all random variables are represented by $m = 9$ nodes. This case will lead to memory error and is therefore not possible to solve with the used computer.

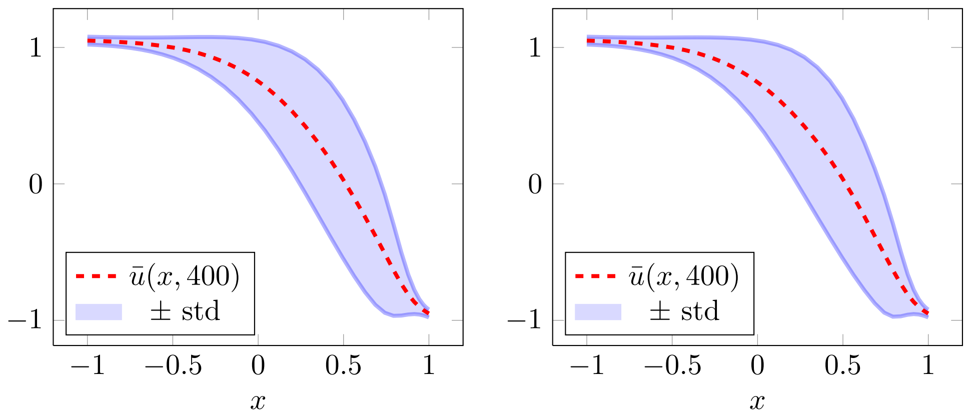

Figure 8.7: Left: Mean and standard deviation solutions for the full tensor grid. Right: Mean and standard deviation solutions for the sparse grid.

In figure 8.7 the results of storchastic Burger's equation are shown for both the full tensor grid and the sparse grid, with $N = 40$ points to represent the spatial space. The steady state is reached at $t = 400$. The solutions in the figure seems to very similar so by using the sparse grid method the almost same solution can be produced much faster.

I order to compare the solutions the difference between them are calculated. This will clarify how similar the solutions are. The difference both between the mean solutions and between the standard deviation solutions are illustrated in figure 8.8.

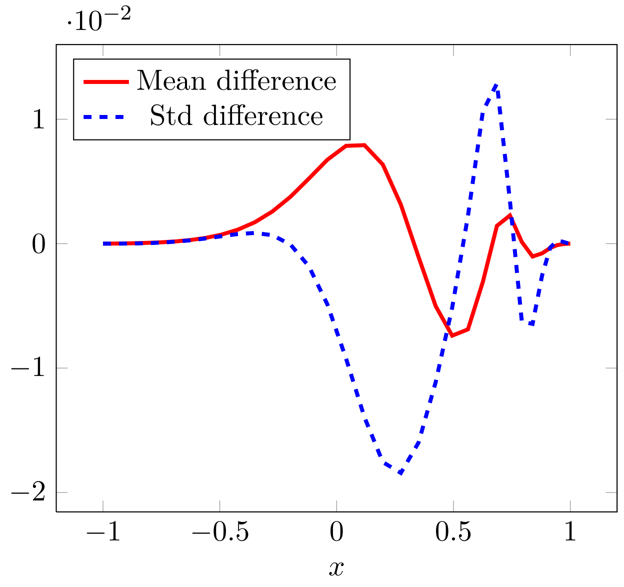

Figure 8.8: Difference between the mean and standard deviation solution for the full tensor grid and the sparse grid.

This figure shows that the largest difference is of the order of approximately 10^{-2}. The figure does not illustrates which solution is the better, because the exact mean and standard deviation solution are not determined for the stochastic Burger's equation. Intuitive the full tensor grid could be assumed to be the most precise solution as more points in the random space is taken into account. On the other hand the used sparse grid with level $l = 2$ corresponds to the tensor grid with $M = 9^3 = 729$ nodes and this have higher precision than the tensor grid with $M = 5^3 = 125$.

However, the overall conclusion from this is that a similar solution can be obtained by using sparse grids. Hence this makes it possible to determine the existent SDE's based on full tensor grids much faster, but also to determine SDE's which not are possible to solve with the full tensor grid.

The final results presented are the calculation times for the same stochastic Burger's equation just solved. The calculation times both found for the full tensor grid and the sparse grid. The full tensor grid will run through $M = [2^3, 3^3, 4^3, 5^3, 6^3]$ while for the sparse grid runs through $l = [1, 2, 3]$. In table 8.2 the corresponding number of nodes/grid points are illustrated together with the corresponding calculation times. To explain the variables in the table to the right, M_s is how many nodes there is in the

M	time/s
2^3	2.26
3^3	18.63
4^3	333.73
5^3	2527.32
6^3	9003.26

l	M_s	M_f	Time/s
1	7	3^3	1.48
2	27	9^3	32.59
3	159	23^3	2669.91

Table 8.2: Left: Calculation times using full tensor grid for different choices of M to represent the 3 random variables. Right: Calculation times using sparse grid for different levels l (and corresponding number of sparse grid points M_s and corresponding full tensor grid points M_f).

sparse grid while M_f is the number of nodes in the corresponding full tensor grid. The two tables shows a huge reduction in time by using the sparse grid. It also shows that it is possible to solve the stochastic Burger's equation where each 3 random variable is represented by 23 nodes in about 45 minutes.

This illustrates that larger representations are possible by using sparse grids and it also can be an important method if larger systems ($d > 3$) are needed to be solved. Overall, the chapter has given an idea of the *curse of dimensionality* and 1 method (sparse grid) that deal with the problem. The sparse grid has makes it possible to solve system which not is possible to solve with the full tensor grid, but the sparse grid also has a upper limit on how large the systems can be.

8.3 Future works

The work presented in this thesis is introducing materials and additional methods, systems and techniques could be investigated and tested. One of the techniques presented (sparse grid) in chapter 7 has been illustrated with one type of sparse grid. Other types of sparse grids could also be investigated and compared with the processed sparse grids.

It will also be of great interest to implement and use the other techniques presented in chapter 7 (ANOVA and ℓ_1- minimization) in order to improve the solution times further. For implementations of these techniques larger systems also can be tested as the systems in this thesis not will be sufficient challenging. As an extension to this systems the random processes also could be taken into account where the Karhunen-Loeve expansions are needed.

Chapter 9

Conclusion

This thesis has first of all showed that the spectral numerical methods can be used to quantifying uncertainty relatively efficient as spectral convergence is obtained. The theory based on the orthogonal polynomials and the corresponding quadratures together with the knowledges to generalized Polynomial Chaos makes the basic of the used Uncertainty Quantification methods.

Throughout the thesis two stochastic differential equations (stochastic Test equation and stochastic Burger's equation) have been solved in many different combinations of random variables. Three methods have be used to determined the statistics of these different systems. The stochastic Test equation is solved satisfactory by all these methods and the expected convergence is obtained which validates all the methods. For the stochastic Burger's equation the Monte Carlo method and the Stochastic Collocation Method (SCM) solves the statistics as expected, while the implementation of the Stochastic Galerkin Method (SGM) ended with wrong (almost correct) statistics due to a complex implementation.

It can be concluded that the SCM is the preferable UQ method in this thesis due to the relatively ease of the implementation but also because of the strong convergence and efficiency. The Monte Carlo method is too inefficient, but the method will for some very large systems be the only method to estimate the statistics. Furthermore it has been a great reference method. The SGM was deselected due to the relatively complex implementation but with the correct implementation the method in some cases still would be preferable.

With SCM the *curse of dimensionality* was illustrated using the full tensor grid to construct the nodes in the random space. In the same chapter the sparse grid was tested and shown that SDE's will be solved much faster and also larger systems are possible to solve compared to the full tensor grid constructed by Tensor Product Collocation method. Additional techniques could be added to this work in order to further improvements of the methods.

The experiences with the programming language `Python` has been positive after few initial difficulties. Many operations and function calls is very much like the corresponding in `Matlab`. Overall, the experiences with `Python` is that it is a bit more efficient compared to `Matlab` but particular the efficiency of the function `odeint` is very high. The usability of `Python` is not at the same level as in `Matlab`.

Appendix A

Additional analytical statistical solutions for the Test equation

Here additional calculation for obtaining an exact analytical mean and variance solutions for the random variables following a uniform distributions. It is in addition to section 5.1.2.

$\alpha(Z) = k$ **and** $\beta(Z) \sim \mathcal{U}(a_2, b_2)$

The expectation and the variance is here determined in the opposite case. The expectation for an uniform distributed variable on the interval $[a_2, b_2]$ is given to be

$$\begin{aligned}\mathbb{E}[\beta] &= \frac{b_2 + a_2}{2}\\ \mathbb{E}[e^{\alpha t}] &= e^{kt}\end{aligned}$$

By this the expectation μ_u in this case is

$$\mu_u = \frac{b_2 + a_2}{2} e^{kt} \tag{A.1}$$

In order to determined the corresponding variance solution $\mathbb{E}[\beta^2]$ have to be solved as in all other cases.

$$\begin{aligned}\mathbb{E}[\beta^2] &= \int_{a_2}^{b_2} \omega^2 \frac{1}{b_2 - a_2}\, d\omega\\ &= \frac{1}{b_2 - a_2}\left[\frac{1}{3}\omega^3\right]_{a_2}^{b^2}\\ &= \frac{1}{b_2 - a_2}\left(\frac{1}{3}b_2^3 - \frac{1}{3}a_2^3\right)\\ &= \frac{1}{3}\left(\frac{b_2^3 - a_2^3}{b_2 - a_2}\right)\end{aligned}$$

By the rule $(b_2^3 - a_2^3) = (b_2 - a_2)(b_2^2 + a_2b_2 + a_2^2)$ the following is obtained

$$\mathbb{E}[\beta^2] = \frac{1}{3}(b_2^2 + a_2b_2 + a_2^2)$$

The other term is the variance expression $(\mathbb{E}[\beta])^2$ is determined by

$$(\mathbb{E}[\beta])^2 = \left(\frac{b_2 + a_2}{2}\right)^2 = \frac{(b_2 + a_2)^2}{4}$$

and hereby the exact variance solution is found to be

$$\begin{aligned}\sigma_u^2 = \mathbb{E}[\beta^2] - (\mathbb{E}[\beta])^2 &= \frac{1}{3}(b_2^2 + a_2b_2 + a_2^2) - \frac{(b_2 + a_2)^2}{4} \\ &= \frac{1}{3}(b_2^2 + a_2b_2 + a_2^2) - \frac{1}{4}(b_2^2 + a_2^2 + 2a_2b_2) \\ &= \frac{4}{12}b_2^2 - \frac{3}{12}b_2^2 + \frac{4}{12}a_2^2 - \frac{3}{12}a_2^2 + \frac{4}{12}a_2b_2 - \frac{6}{12}a_2b_2 \\ &= \frac{1}{12}(b_2^2 + a_2^2 - 2a_2b_2) = \frac{1}{12}(b_2 - a_2)^2\end{aligned}$$

Next the case where both random variables following an uniform distributed are outlined.

$\alpha(Z_1) \sim \mathcal{U}(a_1, b_1)$ **and** $\beta(Z_2) \sim \mathcal{U}(a_2, b_2)$

Here both parameters follows an uniform distribution and the expectation and variance solution is also in this case determined. First the expectation is determined by

$$\mu_u = (\mathbb{E}[u]) = (\mathbb{E}[\beta])(\mathbb{E}[e^{\alpha t}])$$

From earlier these expectations is determined to be

$$\begin{aligned}(\mathbb{E}[\beta]) &= \frac{b_2 + a_2}{2}, \\ (\mathbb{E}[e^{\alpha t}]) &= \frac{1}{t(b_1 - a_1)}(e^{b_1 t} - e^{a_1 t}),\end{aligned}$$

and the final expectation is

$$\mu_u = \frac{b_2 + a_2}{2t(b_1 - a_1)}(e^{b_1 t} - e^{a_1 t})$$

The variance can be determined by earlier computations seen by

$$\begin{aligned}\mathbb{E}[u^2] &= \mathbb{E}[\beta^2]\mathbb{E}[(e^{\alpha t})^2] \\ (\mathbb{E}[u])^2 &= (\mathbb{E}[\beta])^2(\mathbb{E}[e^{\alpha t}])^2\end{aligned}$$

All these four parts have been determined previously and by insertion the final expression for the variance it ends up with

$$\begin{aligned}\sigma_u^2 &= \mathbb{E}[u^2] - (\mathbb{E}[u])^2 \\ &= \frac{1}{3}(b_2^2 + a_2 b_2 + a_2^2)\frac{1}{2t(b_1 - a_1)}(e^{b_1 2t} - e^{a_1 2t}) - \frac{(b_2 + a_2)^2}{4}\frac{1}{t^2(b_1 - a_1)^2}(e^{b_1 t} - e^{a_1 t})^2 \\ &= \frac{b_2^2 + a_2 b_2 + a_2^2}{6t(b_1 - a_1)}(e^{b_1 2t} - e^{a_1 2t}) - \frac{(b_2 + a_2)^2}{4t^2(b_1 - a_1)^2}(e^{b_1 t} - e^{a_1 t})^2\end{aligned}$$

Appendix B

Implemented code

All relevant used Python code is presented in this appendix. This is divided into three sections - 'Toolbox code', '1 dimensional test code' and 'Multidimensional test code'

B.1 Toolbox code

B.1.1 Legendre polynomials

The implementation of Legendre polynomials.

```
import numpy as np

def legendrepol(x,n):
    x = x[:,np.newaxis]
    x = x[:,0]
    # Preallocate for the n+1 polynomials
    L = np.zeros((len(x),n+1))

    if n == 0:
        # Zero order polynomial
        L[:,0] = 1.
    elif n == 1:
        # Zero and first order polynomial
        L[:,0] = 1.
        L[:,1] = x
    else:
        # Zero and first order polynomial
        L[:,0] = 1.
        L[:,1] = x
        i = 2
        while i <= n:
            # Using the three-term recurssion to determine the n>1 order
            # polynomials
            L[:,i] = ((2.*(i-1.) + 1.)/((i-1.) +1.))*x*L[:,i-1]-\
                    ((i-1.)/((i-1.)+1.))*L[:,i-2]
            i += 1
    return L
```

B.1.2 Hermite polynomials

The implementation of Hermite polynomials.

```
import numpy as np

def hermitepol(x,n):
    # Preallocate for the n+1 polynomials.
    H = np.zeros((len(x),n+1))
    if n == 0:
        # The zero order polynomial
        H[:,0] = 1
    elif n == 1:
        # The zero and first order polynomial
        H[:,0] = 1.
        H[:,1] = 2.*x
    else:
        # The zero and first order polynomial
        H[:,0] = 1.
        H[:,1] = 2.*x

        i = 2
        while i <= n:
            # Using the three-terms recurrence to determine the n>1 order
            # polynomials
            H[:,i] = 2.*x*H[:,i-1] - 2.*(i-1)*H[:,i-2]
            i += 1
    return H
```

B.1.3 Lengendre quadrature

Below is the code for the Legendre quadrature shown.

```
import numpy as np

def legendrequad(n):
        # Coefficient a_n
        a = np.zeros((n,1))
        # Coefficient b_n
        vecn = np.arange(1.,n)[:, np.newaxis]
        b = vecn**2/(4*vecn**2-1)

        # Set up the Jacobi matrix
        J = np.diag(np.sqrt(b[:,0]),1)
        J = J + J.T
        # Determine the eigenvalues and eigenvectors
        lambda_n,V = np.linalg.eig(J)

        # Sorting index
        i = lambda_n.argsort(axis = None)
        # The abscissas
        x = np.sort(lambda_n)

        # Determine the weigts
        w = 2*V[0,:]**2
        w = w[i]
        return x,w
```

B.1.4 Hermite quadrature

Here is the implemented code for the Hermite quadrature.

```
import numpy as np

def hermitequad(n):
        # Coefficient a_n
    a = np.zeros((n,1))
    # Coefficient b_n
    b = np.arange(1.,n )[:,  np.newaxis]

    # Set up the Jacobi matrix
    J = np.diag(np.sqrt(b [:,0]),1)
    J = J + J.T

        # Determine the eigenvalues and eigenvectors
        lambda_n,V = np.linalg.eig(J)

    # Sorting index
        i = lambda_n.argsort(axis = None)
        # The abscissas
    x = np.sort(lambda_n)

    # Determine the weigts
    w = np.sqrt(2*np.pi)*V[0,:]**2
    w = w[i]
    return x,w
```

B.1.5 Deterministic Collocation functions

Here the `Python` functions that makes the basis for the Deterministic Collocation method are shown.

```
import numpy as np
import scipy as sc
import scipy.misc as scm
from pylab import *

 factorial  = scm.factorial

# The three constants a_{n-1,n}, a_{n,n} and a_{n+1,n} are defined as a
# function.
def aconst(n,alpha,beta):
    ab = alpha + beta
    ab1 = ab + 1
    ab2 = ab1 + 1
    aminus = (2*(n+alpha)*(n+beta))/((2*n+ab1)*(2*n+ab))
    a = (alpha**2-beta**2)/((2*n+ab2)*(2*n+ab))
    aplus = (2*(n+1)*(n+ab1))/((2*n+ab2)*(2*n + ab1))
    return aminus,a,aplus

# Jacobi polynomial function
def JacobiP(x,alpha,beta,n):
    # Solved often used terms
    ab = alpha + beta
    ab1 = ab + 1
    ab2 = ab1 + 1

    # Preallocate
    P = np.zeros((n+1,len(x)))
    # Zero order polynomial
    P [0,:]  = 1

    if n>0:
        # First order polynomial is calculated.
        P [1,:]  = 0.5*(alpha-beta+ab2*x)

    if n>1:
        for i in xrange(2,n+1):
            amin,a,amax = aconst(i-1,alpha,beta)
            # Calculates the n>1 order polynomials
            P[i ,:]  = ((a+x)*P[i-1,:]-amin*P[i-2,:])/amax

    N = np.linspace(0,n,n+1)
    # Normalizing factor is calculated.
    gamma = 2**(ab1)*(factorial(N+alpha)*factorial(N+beta))/\
                    ( factorial (N)*(2*N+ab1)*factorial(N+ab));
    # Calculates the normalized Jacobi polynomials.
    P = 1/np.sqrt(gamma)*P.T
    return P
```

```
# Differentiated polynomial function.
def GradJacobiP(x,alpha,beta,n):
    N = np.linspace(0,n,n+1)
    const = np.sqrt(N*(N+alpha+beta+1.))
    if n == 0:
        # Zero order polynomial.
        P = np.zeros((len(x),1))
    else:
        # Zero order polynomial.
        P1 = np.zeros((len(x),1))
        # n>0 order polynomials by calling the JacobiP function.
        P2 = JacobiP(x,alpha+1,beta+1,n-1)
        P = const*c_[P1,P2]
    return P

# Quadrature function for the Jacobi polynomial.
def JacobiGQ(alpha,beta,n):
    if n == 0:
        # Abscissas
        x = (alpha+beta)/(alpha+beta+2)
        # Weights
        w = 2;
    else:
        N = np.linspace(0,n,n+1)
        h1 = 2*N+alpha+beta
        # Constructing the Jacobi matrix.
        if alpha +beta ==0.0:
            J = np.zeros((n+1,n+1))
            J [1:,1:]  = np.diag(-0.5*(alpha**2-beta**2)/(h1[1:]+2)/h1[1:])
            J = J + np.diag(2/(h1[0:-1]+2)*np.sqrt(N[1:]*(N[1:]+alpha+beta)*\
                (N[1:]+alpha)*(N[1:]+beta)/(h1[0:-1]+1)/(h1[0:-1]+3)),1)
        else:
            J = np.diag(-0.5*(alpha**2-beta**2)/(h1+2)/h1)+\
            np.diag(2/(h1[0:-1]+2)*np.sqrt(N[1:]*(N[1:]+alpha+beta)*\
                (N[1:]+alpha)*(N[1:]+beta)/(h1[0:-1]+1)/(h1[0:-1]+3)),1)
        J = J + J.T

        # Solves the eigenvalues and eigenvectors.
        D,V = np.linalg.eig(J)
        # Sort the eigenvalues.
        i = D.argsort(axis = None)
        x = np.sort(D)
        # Calculates the corresponding weights.
        w = V[0,:]**2*2**(alpha+beta+1)/(alpha+beta+1)*factorial(alpha)*\
            factorial (beta)/ factorial (alpha+beta)
        # Sort the weights.
        w = w[i]
        return x,w
```

```
# Gauss-Lobatto node function.
def JacobiGL(alpha,beta,n):
    # Preallocate.
    x = np.zeros((n+1,1))
    if n ==1:
        # Boundary nodes.
        x[0] = -1.0
        x[1] = 1.0
    else:
        # Finds the interior nodes by the JacobiGQ function.
        x,w = JacobiGQ(alpha+1,beta+1,n-2)
        x = x[:, np.newaxis]
        # Adds the boundary nodes.
        x = np.vstack((-1,x,1))
        x=x[:,0]
    return x,w
```

B.1.6 Implementation of the deterministic Burger's equation

Below the implementation of the deterministic Burger's equation is shown.

```
import numpy as np
import jacobipol
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from time import *
from pylab import *

JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# The right hand side defined as a function.
def rhs(u,t,v,D,D2):
    u = -u*np.dot(D,u) + nu*np.dot(D2,u)
    u[0] = 0.0
    u[-1] = 0.0
    return u

# Function to create the initial condition.
def init(x):
    u_init = -x
    return u_init

# Vandermonde matrix defined as a function.
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
```

```
# Differentiated Vandermonde matrix defined as a function.
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx

# Left spacial bound
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 80
# Space step
dx = (x_right-x_left)/n_space

# Initial time
t_init = 0.0
# End time
t_final = 1000.0
# Times the solution is desired.
t_span = np.linspace(t_init,t_final,n_time)

# parameter nu
nu = 0.05

# Parameters for the deterministic collocation method
alpha = 0.0
beta = 0.0

# Gauss-Lobatto nodes
x,w = JacobiGL(alpha,beta,n_space)

# Construction of the Vandermonde matrix and the differentiated
# Vandermonde matrix.
V = vanderMat(x)
Vx = vanderMatx(x)

# Determined the differentiation matrix D and D^2.
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Initial condition.
u_init = init(x)

# Solve the system.
u = odeint(rhs,u_init,t_span,tuple([nu,D,D2]))
```

B.2 1 dimensional test code

B.2.1 Stochastic Test equation

Code for the Monte Carlo method used at the Stochastic Test equation with $\alpha(Z) \sim \mathcal{N}(0,1)$.

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint

# Number of realizations from the distribution.
n_expan = 1000

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.1

t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)
# Number of time steps - 1 extra for initial condition.
num_steps = np.floor((t_final - t_start)/delta_t) + 1

# Set initial condition.
beta = 1.0

# Parameters for the distribution.
mu = 0.0
var = 1.0
sigma = sqrt(var)

# Draw n random numbers.
alpha = np.random.normal(mu,sigma,n_expan)

# Initial condition.
u_init = beta*np.ones((n_expan,1))

# Determine the solution.
u = odeint(rhs.rhs,u_init[:,0], t_span,tuple([alpha]))

# Calculates the statisicals parameters.
mu_estimate = np.mean(u, axis = 1)
var_estimate = np.mean(u**2, axis = 1)-mu_estimate**2

# Exact mean and variance solution.
mu_exact = beta*exp(0.5*(sigma**2*t_span**2 + 2.*mu*t_span))
var_exact = beta**2*exp(2*sigma**2*t_span**2 + 2*mu*t_span)\
              - beta**2*exp(sigma**2*t_span**2 + 2*mu*t_span)
```

```
std = np.sqrt(var_estimate)
```

Code for the Monte Carlo method used at the Stochastic Test equation with $\alpha(Z) \sim \mathcal{U}(-1, 1)$.

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint

# Number of realizations from the distribution.
n_expan = 1000

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.1
t_span = np.linspace(t_start,t_final,int (2.0/delta_t)+1)
num_steps = np.floor((t_final - t_start)/delta_t) + 1

# Set initial condition
beta = 1.0

# Parameters for the distribution.
a = -1
b = 1

# Draw n random numbers.
alpha = np.random.uniform(a,b,n_expan)

# Initial condition.
u_init = beta*np.ones((n_expan,1))

# Determine the solutions.
u = odeint(rhs.rhs,u_init [:,0], t_span,tuple([alpha]))

# Calculates the statisicals parameters.
mu_estimate = np.mean(u, axis = 1)
var_estimate = np.mean(u**2, axis = 1)-mu_estimate**2

t = t_span.copy()
t_span = t_span[1:]

# Exact mean and variance solution.
mu_exact = beta/(t_span*(b-a))*(exp(b*t_span)-exp(a*t_span))
var_exact = beta**2*(1/(2*t_span*(b-a))*(exp(b*2*t_span)-exp(a*2*t_span))-\
                        1/(t_span*(b-a))**2*(exp(b*t_span)-exp(a*t_span))**2)
mu_exact = r_[beta,mu_exact]
var_exact = r_[0.0,var_exact]
```

```
std = np.sqrt(var_estimate)
```

Convergence test where $\alpha(Z) \sim \mathcal{N}(0,1)$.

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint

N = np.linspace(10,np.sqrt(500000),50)
N = N**2

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.1
t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Parameters for the distribution.
mu = 0.0
var = 1.0
sigma = np.sqrt(var)

# Preallocate vector to containing errors.
E = np.zeros((len(N),1))
i = 0
for n in N:
    n = int(n)
    alpha = np.random.normal(mu,sigma,n)

    # All initial condition.
    u_init = beta*np.ones((n,1))
    u = np.zeros((len(t_span),n))
    nn = 10

    # Split the system up in order to avoid large matrices.
    for j in xrange(0, n/nn):
        # Initial condition.
        u0 = u_init[j*nn:j*nn+nn,0]
        # Random parameters
        alpha_in = alpha[j*(nn):j*(nn)+nn]
        # Solve the subsystem.
        u = odeint(rhs.rhs,u0,t_span,tuple([alpha_in]))

        # Calculates the estimated mean solution
        # for the subsystem.
        mu_estimate_nn = np.mean(u, axis = 1)
        var_estimate_nn = np.var(u, axis = 1)

        # Overall estimated mean solution.
        if j!=0:
            mu_estimate = (mu_estimate*(j-1)*nn+mu_estimate_nn*nn)/(j*nn)
        else:
            mu_estimate = mu_estimate_nn

    mu_exact = beta*exp(0.5*(sigma**2*t_span**2 + 2.*mu*t_span))

    # Calculates the estimated mean solution.
    E[i] = np.max(np.abs(mu_estimate-mu_exact))
    i+=1
```

Code for the SCM used to solve stochastic Test equation with $\alpha(Z) \sim \mathcal{N}(0,1)$.

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint
import hermitepol as hep
import hermitequad as heq

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.01
t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Set the statistical parameters.
mu = 0.0
var = 1.0
sigma = np.sqrt(var)

# Number of nodes to represent the random variable.
n_expan = 5

# Calls the hermite-gauss quadrature to obtaine nodes and weights.
x,w = heq.hermitequad(n_expan,2)

# Transforming the nodes.
alpha = mu + sigma*x

# Initial condition.
u_init = beta*np.ones((n_expan,1))
```

```
# Determine the solutions.
u = odeint(rhs.rhs,u_init [:,0], t_span,tuple([alpha]))

# Calculates the estimated mean and variance solution.
mu_estimate = 1/np.sqrt(2*np.pi)*sum(w*u,axis=1)
var_estimate = 1/np.sqrt(2*np.pi)*\
            sum(w*((u-np.tile(mu_estimate,[n_expan,1]).T))**2,axis=1)

# Exact mean and variance solutions.
mu_exact = beta*exp(0.5*(sigma**2*t_span**2 + 2.*mu*t_span))
var_exact = beta**2*exp(2*sigma**2*t_span**2 + 2*mu*t_span)\
            - beta**2*exp(sigma**2*t_span**2 + 2*mu*t_span)
std = np.sqrt(var_estimate)
```

Code for the SCM to solve the stochastic Test equation with $\alpha(Z) \sim \mathcal{U}(-1,1)$.

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint
import legendrepol as lep
import legendrequad as leq

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.01
t_span = np.linspace(t_start,t_final,int (2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Number of nodes in random space.
n_expan = 5

# Uniform distribution parameters.
a = -1.0
b = 1.0
x,w = leq.legendrequad(n_expan)

# Transformation from [-1,1] to [a,b]
alpha = 0.5*(b-a)*x + 0.5*(b+a)

# Calculates initial condition
u_init = beta*np.ones((n_expan,1))

# Determine the solutions
u = odeint(rhs.rhs,u_init [:,0], t_span,tuple([alpha]))
```

```
# Statistical parameters.
mu_estimate = 0.5*sum(w*u,axis=1)
var_estimate = 0.5*\
               sum(w*((u-np.tile(mu_estimate,[n_expan,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)

t = t_span.copy()
t_span = t_span[1:]

# Exact mean and variance solution.
mu_exact = beta/(t_span*(b-a))*(exp(b*t_span)-exp(a*t_span))
var_exact = beta**2*(1/(2*t_span*(b-a))*(exp(b*2*t_span)-exp(a*2*t_span))-\
                     1/(t_span*(b-a))**2*(exp(b*t_span)-exp(a*t_span))**2)
mu_exact = r_[beta,mu_exact]
var_exact = r_[0.0,var_exact]
```

Convergence test where $\alpha(Z) \sim \mathcal{N}(0,1)$

```
import numpy as np
from matplotlib.pylab import *
import rhs as rhs
from scipy.integrate import odeint
import hermitepol as hep
import hermitequad as heq

N = np.linspace(1,20,20)

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.01
t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Normal distribution parameters.
mu = 0.0
var = 1.0
sigma = np.sqrt(var)

# Preallocation.
E_mu = np.zeros((len(N),1))
E_var = np.zeros((len(N),1))
i = 0

for n in N:
    n_expan = int(n)
    # Determines nodes and weights.
    x,w = heq.hermitequad(n_expan,2)

    # Transformation from [-1,1] to [a,b]
    alpha = mu + sigma*x

    # Calculates initial condition
    u_init = beta*np.ones((n_expan,1))

    # Determine the solutions
    u = odeint(rhs.rhs,u_init [:,0], t_span,tuple([alpha]))

    # Statistical parameters.
    mu_estimate = 1/np.sqrt(2*np.pi)*sum(w*u,axis=1)
    mu_exact = beta*exp(0.5*(sigma**2*t_span**2 + 2.*mu*t_span))

    # Exact mean and variance solution.
    var_estimate = 1/np.sqrt(2*np.pi)*\
                sum(w*((u-np.tile(mu_estimate,[n_expan,1]).T))**2,axis=1)
    var_exact = beta**2*exp(2*sigma**2*t_span**2 + 2*mu*t_span)\
                 - beta**2*exp(sigma**2*t_span**2 + 2*mu*t_span)

    # Determine the errors.
    E_mu[i] = np.max(np.abs(mu_estimate-mu_exact))
    E_var[i] = np.max(np.abs(var_estimate-var_exact))
    i+=1
```

Code for the SGM used to solve stochastic Test equation with $\alpha(Z) \sim \mathcal{N}(0,1)$.

```
import numpy as np
import scipy as sp
import scipy.misc as spm
from pylab import *
import he_product
import rhs as rhs
from scipy.integrate import odeint

triple_product = he_product.he_tpi
double_product = he_product.he_dpi

# Initial time
t_init = 0.0
# End time
t_final = 1.0
# Number of time steps
dt = 0.01
# Time step
n_time = int((t_final-t_init)/dt)
t_span = np.linspace(t_init,t_final,n_time)
# Initial condition
```

```
beta = 1.0
# Number of expansion
n_expan = 5

# Parameters of the stochastic variable
mu = 0.0
var = 1.0
sigma = np.sqrt(var)

# Coefficients a_i
a = np.zeros((n_expan+1,1))
a[0,0] = mu
a[1,0] = sigma

# Preallocating
A = np.zeros((n_expan+1,n_expan+1))
u0 = np.zeros((n_expan+1,1))

# Initial vector.
u0[0,0] = beta

# Construct the matrix containing e_{ijk}.
for j in xrange(0,n_expan+1):
    for k in xrange(0,n_expan+1):
        for i in xrange(0,n_expan+1):
            A[j,k] = A[j,k] - a[i,0]*triple_product(i,j,k)/double_product(k,k)

# Determine the solutions.
u = odeint(rhs.rhs,u0[:,0], t_span,tuple([-A]))

n = range(1,n_expan+1)

# Statistical parameters.
mu_estimate = u[:,0]
var_estimate = sum(double_product(n,n)*u[:,1:]**2,axis=1)

# Exact statistical parameters
mu_exact = beta*exp(0.5*(sigma**2*t_span**2 + 2.*mu*t_span))
var_exact = beta**2*exp(2*sigma**2*t_span**2 + 2*mu*t_span)\
                - beta**2*exp(sigma**2*t_span**2 + 2*mu*t_span)
std = np.sqrt(var_estimate)
```

Function that calculates e_{ijk}. Convergence test where $\alpha(Z) \sim \mathcal{N}(0,1)$

```
import numpy as np
import scipy as sp
import scipy.misc as spm

fac = sp.misc.factorial
```

```
def he_tpi(i,j,k):

    s = np.floor((i+j+k)/2)
    if (s<i or s<j or s<k):
        value = 0.0
    elif (np.mod(i+j+k,2) != 0):
        value = 0.0
    else:
        value = (fac(i)*fac(j)*fac(k))/(fac(s-i)*fac(s-j)*fac(s-k))
    return value

def he_dpi(i,j):
    if i != j:
          value = 0.0
    else:
          value = fac(i)
    return value
```

Convergence test where $\alpha(Z) \sim \mathcal{N}(0,1)$

```
import numpy as np
import scipy as sp
import scipy.misc as spm
from pylab import *
import he_product
import rhs as rhs
from scipy.integrate import odeint

triple_product = he_product.he_tpi
double_product = he_product.he_dpi

N = np.linspace(1,20,20)

# Initial time
t_init = 0.0
# End time
t_final = 1.0
# Number of time steps
dt = 0.01
# Time step
n_time = int((t_final-t_init)/dt)
t_span = np.linspace(t_init,t_final,n_time)
# Initial condition
beta = 1.0

# Parameters of the stochastic variable
mu = 0.0
var = 1.0
sigma = np.sqrt(var)
```

```
# Preallocate error vectors
E_mu = np.zeros((len(N),1))
E_var = np.zeros((len(N),1))
l = 0

for n in N:
    # Number of expansion
    n_expan = n

    # Coefficients a_i
    a = np.zeros((n_expan+1,1))
    a [0,0]  = mu
    a [1,0]  = sigma

    # Preallocating
    A = np.zeros((n_expan+1,n_expan+1))
    u0 = np.zeros((n_expan+1,1))

    # Initial vector.
    u0 [0,0]  = beta

    # Construct the matrix containing e_{ijk}.
    for j in xrange(0,int(n_expan)+1):
        for k in xrange(0,int(n_expan)+1):
            for i in xrange(0,int(n_expan)+1):
                A[j,k]  = A[j,k]  − a[i,0]∗triple_product(i,j,k)/double_product(k,k)

    # Determine the solutions.
    u = odeint(rhs.rhs,u0 [:,0], t_span,tuple([−A]))

    # Statistical parameters.
    mu_estimate = u[:,0]
    mu_exact = beta∗exp(0.5∗(sigma∗∗2∗t_span∗∗2 + 2.∗mu∗t_span))

    n = range(1,int(n_expan)+1)
    var_estimate = sum(double_product(n,n)∗u[:,1:]∗∗2,axis=1)
    var_exact = beta∗∗2∗exp(2∗sigma∗∗2∗t_span∗∗2 + 2∗mu∗t_span)\
                − beta∗∗2∗exp(sigma∗∗2∗t_span∗∗2 + 2∗mu∗t_span)
    std  = np.sqrt(var_estimate)

    # Calculates the errors.
    E_mu[l] = np.max(np.abs(mu_estimate−mu_exact))
    E_var[l] = np.max(np.abs(var_estimate−var_exact))
    l+=1
```

B.2.2 Stochastic Burger's equation

Code for the Monte Carlo method used to solve stochastic Burger's equation with $\delta_1(Z) \sim \mathcal{U}(0, 0.1)$.

```
import numpy as np
import jacobipol
from matplotlib.pyplot import *
from scipy.integrate import odeint
import time

JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Right hand side function
def rhs(u,t,v,D,D2):
    u = -u*np.dot(D,u) + v*np.dot(D2,u)
    u[0]  = 0.0
    u[-1] = 0.0
    return u

# Initial condition function.
def init(x):
    u_init = -x
    return u_init

# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V

# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx

# Number of expansions
n_expan = 1000

# Left spacial bound
x_left  = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 80
# Space step
dx = (x_right-x_left)/n_space
```

```
# Initial time
t_init = 0.0
# End time
t_final = 1000.0
# Time step
dt = 0.31
# Number of time steps
n_time = int(np.ceil((t_final/dt)))

# parameter nu
nu = 0.05

# Uniform distribution parameters.
a = 0.0
b = 0.1

# Time step vector.
t_span = np.linspace(0,t_final,n_time)

# Spatial nodes is determined.
alpha = 0.0
beta = 0.0
x,w = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Determine the Differential matrices and the squared differential matrix.
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Preallocating.
u_final = np.zeros((len(x),n_expan))

# Creating the M random numbers
delta = np.random.uniform(a,b,n_expan)

for n in range(1,len(delta)+1):
    # Initial condition.
    u_init = init(x)
    # Adding the uncertainty.
    u_init[0] = u_init[0] + delta[n-1]

    # Initial tolerance, which must be too big.
    tol = 1
    max_iter = 1
    # Time step
    dt = 0.1
    # Number of time jumps
    nt_jump = 1000
    # The last t in the first t_span
    t_end = dt*nt_jump
    # Create the t_span
    t_span = np.linspace(0,t_end,nt_jump)

    while tol> 10**(-6) and max_iter < 10**5:
        # Solve the system in the next nt_jump time steps
        u = odeint(rhs,u_init,t_span,tuple([nu,D,D2]))

        # Find the difference between the solution to the first and the last
        # element in t_span
        u_check = u[-1,:]
        tol = max(np.abs(u_init-u_check))

        # Update t_span and the initial condition
        t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
        u_init = u_check
        max_iter += 1
    # Save the steady state solution.
    u_final [:, n-1] = u_check

# Calculates the statistical parameters.
mu_estimate = np.mean(u_final,axis =1)
var_estimate = np.var(u_final,axis =1)
std = np.sqrt(var_estimate)
```

Code for the SCM used to solve stochastic Burger's equation with $\delta_1(Z) \sim \mathcal{U}(0, 0.1)$.

```
import numpy as np
from scipy.sparse. linalg .dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
from time import *
import scipy.sparse as sparse

legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Initial condition function.
def init (x):
    return -x
# Vandermonde matrix function
```

```
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx
# Right hand side function
def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*np.dot(D,u) + nu*np.dot(D2,u)
    u = u*B
    return u

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu
nu = 0.05

# Interval of the uniform boundary
a = 0.0
b = 0.1

# Number of boundary nodes
n_expan = 5
z,w = legendrequad(n_expan)

# Transform from interval [-1,1] to [a,b]
delta = (b-a)/2*z + (b+a)/2

# Spatial nodes is determined.
alpha = 0.0
beta = 0.0
x,w_nu = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Determine the Differential matrices and the squared differential matrix.
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)
```

```
# Initial condition.
u_init = init(x)
u_init = u_init

# Expanding the system so in contains the deterministic systems.
D_expan = np.kron(np.identity(n_expan),D)
D2_expan = np.kron(np.identity(n_expan),D2)
u_init_expan = np.tile(u_init,n_expan)
delta_expan = np.zeros(((n_expan)*(n_space+1),1))

for i in xrange(0,n_expan):
    delta_expan[i*(n_space+1):(i*(n_space+1)+n_space+1),:]\
    =delta[i]*np.ones((n_space+1,1))

u_init_expan[u_init_expan>0] = u_init_expan[u_init_expan>0]\
                              *(1+delta_expan[u_init_expan>0,0])

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
B = np.tile(B,n_expan)

# Initial tolerance, which must be too big.
tol = 1
max_iter = 1
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

while tol> 10**(-6) and max_iter < 10**5:
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
               tuple([nu,D_expan,D2_expan,B]))
    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol = max(np.abs(u_init_expan-u_check))

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check
    max_iter += 1

# Save the steady state solution.
u_sol = u_check.reshape(n_space+1,n_expan,order='F')

# Expand the weights to the right size.
w_expan = np.tile(w,(n_space+1,1))

# Calculates the statistical parameters.
mu_estimate = 0.5*np.sum(w_expan*u_sol,axis=1)
var_estimate = 0.5*np.sum(w_expan*\
                 ((u_sol-np.tile(mu_estimate,[n_expan,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

Code for the SGM used to solve stochastic Burger's equation with $\delta_1(Z) \sim \mathcal{U}(0, 0.1)$.

```
import numpy as np
import scipy as sp
import scipy.misc as spm
from pylab import *
import mtx_e as mtx
import le_product
import legendrequad
import rhs_burger as rhsb
from scipy.integrate import odeint
import jacobipol as jac

mtx = mtx.mtx
legendrequad = legendrequad.legendrequad
double_product = le_product.legendre_double_product

def init(x):
    return -x

nu = 0.05
# Initial time
t_init = 0.0
# End time
t_final = 100.0
# Number of time steps
n_time = 1000
# Time step
#dt = (t_final-t_init)/n_time
#dt = 0.001
n_time = int(n_time)
# Left spacial bound
x_left = -1.0
# Right spacial bound
x_right = 1.0
```

```
# Number of spacial steps
n_space = 39

# Space step
dx = (x_right−x_left)/n_space
# Number of expansion
n_expan = 6

t_span = np.linspace(t_init,t_final,n_time+1)

# Setup time and space
t  = np.zeros((n_time+1,1))
x = linspace(−1,1,n_space+1)

b_mean =0.05
#b_std = np.sqrt(1.0/12.0*(0.1-0.0)**2)
b_std = 1.0/np.sqrt(12)∗(1.1−1.0)

# Initial condition computed
u_init = init(x[1:−1])

# Pre-allocation
u_space = np.zeros((n_space+1,n_expan+1))
u_time = np.zeros((n_space+1,n_time+1))

# Impose boundary condition and initial condition
u_space[0,0] = 1.0+b_mean
u_space[0,1] = b_std
u_space[1:−1,0] = u_init
u_space[−1,0] = −1.0

# Store mean solution for t=0.
u_time[:,0]  = u_space[:,0]

# Setup all expansions on vectorform
u_space_vec = u_space[1:−1,:].flatten(1)[:,np.newaxis]

one = np.ones((n_space−1,1))

# Stencilmatrix for the double differentiation term
A = nu/(dx∗∗2.)∗(np.diag(one[1:,0],−1) + np.diag(one[1:,0],1)\
     + np.diag(−2.∗one[:,0],0))

# Stencilmatrix for the sigle differentiation term
B = 1./(2.∗dx)∗(np.diag(−one[1:,0],−1) + np.diag(one[1:,0],1))

# Vector conpemsate with the boundary
g = np.zeros((n_space−1,1))
```

```
# Repetition og the u matrix
I = np.tile(np.eye(n_space−1),(1,n_expan+1))
u_space_I = u_space_vec.T*I

#Pre-allocation
C = zeros(((n_expan+1),(n_expan+1),n_expan+1))
CB = zeros(((n_space−1)*(n_expan+1),(n_space−1)*(n_expan+1),n_expan+1))

for l in xrange(0,n_expan+1):
    # Matrix containing e_{i,j,k}
    C [:,:, l] = mtx(l,n_expan+1)
    # Kronecker product between the stencilmatrix B and the e_{i,j,k}-matrix
    CB [:,:, l ] = np.kron(C [:,:, l ], B)

for i in xrange(0,n_time):
    u1 = u_space[1,:]
    u2 = u_space[−2,:]
    for k in xrange(0,n_expan+1):
        g [0,0] = 0.0
        g[−1,0] = 0.0

        for kk in xrange(0,n_expan+1):
            g [0,0] = g[0,0]+ u_space[0,kk]*(1./(2.*dx)*sum(C[:,kk,k]\
                                    *u1))
            g[−1,0] = g[−1,0] + u_space[−1,kk]*(−1./(2.*dx)*sum(C[:,kk,k]\
                                    *(u2)))

        g [0,0] = 1./double_product(k,k)*g[0,0] + u_space[0,k]*nu/(dx**2.)
        g[−1,0] = 1./double_product(k,k)*g[−1,0] + u_space[−1,k]*nu/(dx**2.)

        u = u_space[1:−1,k][:,np.newaxis]

        u_temp = odeint(rhsb.rhs,u[:,0],t_span[i:i+2],\
            tuple ([u_space_vec[:,0],u_space_I,A,CB[:,:,k],g [:,0], k,n_space]))
        u_temp = u_temp[−1,:]
        u_space[1:−1,k] = u_temp

    print t_span[i]
    u_time[:,i+1] = u_space[:,0]
    u_space_vec = u_space[1:−1,:].flatten(1)[:,np.newaxis]
    u_space_I = u_space_vec.T*I

mu_estimate = u_time[:,−1]
n = np.linspace(1,n_expan,n_expan)
gamma_n = 2.0/(2.0*n+1.0)

var_estimate = np.sum(gamma_n*u_space[:,1:]**2,1)
std = np.sqrt(var_estimate)
```

B.3 Multidimensional test code

B.3.1 Stochastic Test equation ($d = 2$) - SCM

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint
import hermitequad as heq
import legendrequad as leq
from cartesian import *

# Initial condition function.
def init(t,a,b):
    u = b*np.exp(a*t)
    return u

# Set the time range
t_start = 0.0
t_final = 1.0
delta_t = 0.01
t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Statistical parameters for the random variables.
mu1 = 0.0
var1 = 1.0
sigma1 = np.sqrt(var1)

mu2 = 1.0
var2 = 0.1
sigma2 = np.sqrt(var2)

# Number of realizations to represent the random variables.
n_expan = 6

x1,w1 = heq.hermitequad(n_expan,2)
x2,w2 = heq.hermitequad(n_expan,2)

# Transformation.
Theta_1 = mu1 + sigma1*x1
Theta_2 = mu2 + sigma2*x2

# Create all combinations of the nodes
nodes = cartesian((Theta_1,Theta_2))
```

```
# Split into two different vectors
A = nodes[:,0]
B = nodes[:,1]

# Create all combinations of the weights
W_all = cartesian((w1,w2))

# Split into two different vectors
W1 = W_all[:,0]
W2 = W_all[:,1]

# Multiple all the combinations of weights
W = W1*W2

# Determine the initial condition.
u_init = init(0,A,B)

# Determine the solutions.
u = odeint(rhs.rhs,u_init,t_span,tuple([A]))

# Calculates the estimate statistics
mu_estimate = (1/np.sqrt(2*np.pi))**2*np.sum(np.tile(W,[len(t_span),1])*u,1)
var_estimate = 1/np.sqrt(2*np.pi)**2*\
               sum(W*((u-np.tile(mu_estimate,[n_expan*n_expan,1]).T))**2,axis=1)

# Calculates the exact statistics.
mu_exact = mu2*exp(0.5*(sigma1**2*t_span**2 + 2.*mu1*t_span))
var_exact = (mu2**2 + sigma2**2)*exp(2*sigma1**2*t_span**2 + 2*mu1*t_span)\
                - mu2**2*exp(sigma1**2*t_span**2 + 2*mu1*t_span)
std = np.sqrt(var_estimate)
```

B.3.2 Stochastic Test equation ($d = 2$) - Convergence test

```
import numpy as np
from matplotlib.pylab import *
import time
import rhs as rhs
from scipy.integrate import odeint
import hermitequad as heq
import legendrequad as leq
from cartesian import *

# Initial condition function.
def init(t,a,b):
    u = b*np.exp(a*t)
    return u

# Set the time range
```

```
t_start = 0.0
t_final = 1.0
delta_t = 0.01
t_span = np.linspace(t_start,t_final,int(2.0/delta_t)+1)

# Set initial condition
beta = 1.0

# Statistical parameters for the random variables.
mu1 = 0.0
var1 = 1.0
sigma1 = np.sqrt(var1)

mu2 = 1.0
var2 = 0.1
sigma2 = np.sqrt(var2)

E = np.zeros((19,1))
V = np.zeros((19,1))
M = np.zeros((19,1))

for i in range(2,21):
    n_expan = i

    x1,w1 = heq.hermitequad(n_expan,2)
    x2,w2 = heq.hermitequad(n_expan,2)

    # Transformation.
    Theta_1 = mu1 + sigma1*x1
    Theta_2 = mu2 + sigma2*x2

    # Create all combinations of the nodes
    nodes = cartesian((Theta_1,Theta_2))

    # Split into two different vectors
    A = nodes[:,0]
    B = nodes[:,1]

    # Create all combinations of the weights
    W_all = cartesian((w1,w2))

    # Split into two different vectors
    W1 = W_all[:,0]
    W2 = W_all[:,1]

    # Multiple all the combinations of weights
    W = W1*W2

    # Determine the initial condition.
    u_init = init(0,A,B)

    # Determine the solutions.
    u = odeint(rhs.rhs,u_init,t_span,tuple([A]))

    # Calculates the estimate statistics
    mu_estimate = (1/np.sqrt(2*np.pi))**2*np.sum(np.tile(W,[len(t_span),1])*u,1)
    var_estimate = 1/np.sqrt(2*np.pi)**2*\
                sum(W*((u-np.tile(mu_estimate,[n_expan*n_expan,1]).T))**2,axis=1)

    # Calculates the exact statistics.
    mu_exact = mu2*exp(0.5*(sigma1**2*t_span**2 + 2.*mu1*t_span))
    var_exact = (mu2**2 + sigma2**2)*exp(2*sigma1**2*t_span**2 + 2*mu1*t_span)\
                 - mu2**2*exp(sigma1**2*t_span**2 + 2*mu1*t_span)
    std  = np.sqrt(var_estimate)

    # Calculates the errors.
    E[i-2,0] = np.max(np.abs(mu_estimate-mu_exact))
    V[i-2,0] = np.max(np.abs(var_estimate-var_exact))
    M[i-2,0] = n_expan**2
```

B.3.3 Sparse matrix test

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
import scipy.sparse as sparse
from cartesian import *

# Initial condition function.
def init(x):
    return -x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx
```

```
# Right hand side function for full matrices
def rhs_full(u,t,nu,D,D2,B):
    u = -u*np.dot(D,u) + nu*np.dot(D2,u)
    u = u*B
    return u
# Right hand side function for sparse matrices
def rhs_sp(u,t,nu,D,D2,B):
    u = -u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0

t_span = 0

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

# Statistical parameters for v.
mu_nu = 0.2
var_nu = 0.001
std_nu = np.sqrt(var_nu)

# Define the combinations that the timings must be performed at.
M = np.array((39,49,59,69))
N1 = np.array((4,5,6))

MN = cartesian((M,N1))
MN = c_[MN[:,0],MN[:,1],MN[:,1],MN[:,1]]
P = np.product(MN,axis=1)
MNP = c_[MN,P]

I = np.argsort(MNP.T)

MNP1 = MNP[I[-1,:]]
```

```
T_full = np.zeros((len(MNP1),1))
T_sp = np.zeros((len(MNP1),1))

for i in range(len(MNP1)):
    # Number of nodes representing randomvariables
    n_expan1 = MNP1[i,1]
    n_expan2 = MNP1[i,2]
    n_expan3 = MNP1[i,3]
    # Number of spacial steps
    n_space = MNP1[i,0]

    z1,w1 = legendrequad(n_expan1-1)
    z2,w2 = legendrequad(n_expan2-1)
    nu,w_nu = hermitequad(n_expan3-1,2)

    # Transforming weights.
    w1 = w1/2
    w2 = w2/2
    w3 = 1/np.sqrt(2*np.pi)*w_nu

    # Transform from interval [-1,1] to [a,b]
    delta1 = (b1-a1)/2*z1 + (b1+a1)/2
    delta2 = (b2-a2)/2*z2 + (b2+a2)/2
    # Transforming the v parameter
    nu = mu_nu + std_nu*nu

    # Makes all possible combinations between all abscissas.
    delta_temp = cartesian((delta1,delta2,nu))
    Delta1 = delta_temp[:,0]
    Delta2 = delta_temp[:,1]
    Nu = delta_temp[:,2]

    # Makes all possible combinations between all weights.
    W_temp = cartesian((w1,w2,w3))
    W1 = W_temp[:,0]
    W2 = W_temp[:,1]
    W3 = W_temp[:,2]

    W = W1*W2*W3

    # Determine the total number of deterministic systems that have to be
    # solved.
    new_dim = (n_expan1-1)*(n_expan2-1)*(n_expan3-1)

    # To the deterministic solver
    alpha = 0.0
    beta = 0.0
    x,w_nu = JacobiGL(alpha,beta,n_space)
```

```
# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Differential matrices
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(Nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
    u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
            u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0]  = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol  = 1
max_iter = 1
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)
t1 = time()
while tol> 10**(-6) and max_iter < 10**5:

    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_sp,u_init_expan,t_span,\
            tuple([Nu_expan,D_expan,D2_expan,B]))
    # Find the difference between the solution to the first and the last
```

```
        # element in t_span
        u_check = u[−1,:]
        tol  = max(np.abs(u_init_expan−u_check))

        # Update t_span and the initial condition
        t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
        u_init_expan = u_check

        max_iter += 1
    T_sp[i,0]  = time()−t1
```

B.3.4 Stochastic Burger's equation ($d = 3$) - SCM

```
import numpy as np
from scipy.sparse. linalg .dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
import scipy.sparse as sparse
from cartesian import *

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Initial condition function.
def init (x):
    return −x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N−1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N−1)
    return Vx
# Right hand side function
def rhs_SCM(u,t,nu,D,D2,B):
    u = −u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

# Statistical parameters for v.
mu_nu = 0.2
var_nu = 0.001
std_nu = np.sqrt(var_nu)

# Number of nodes representing random variables
n_expan = 5
z1,w1 = legendrequad(n_expan-1)
z2,w2 = legendrequad(n_expan-1)
nu,w_nu = hermitequad(n_expan-1,2)

# Transforming weights.
w1 = w1/2
w2 = w2/2
w3 = 1/np.sqrt(2*np.pi)*w_nu

# Transform from interval [-1,1] to [a,b]
delta1 = (b1-a1)/2*z1 + (b1+a1)/2
delta2 = (b2-a2)/2*z2 + (b2+a2)/2
# Transforming the v parameter
nu = mu_nu + std_nu*nu

# Makes all possible combinations between all abscissas.
delta_temp = cartesian((delta1,delta2,nu))
Delta1 = delta_temp[:,0]
Delta2 = delta_temp[:,1]
Nu = delta_temp[:,2]

# Makes all possible combinations between all weights.
W_temp = cartesian((w1,w2,w3))
```

```
W1 = W_temp[:,0]
W2 = W_temp[:,1]
W3 = W_temp[:,2]

W = W1*W2*W3

# Determine the total number of deterministic systems that have to be
# solved.
new_dim = (n_expan-1)**3

# To the deterministic solver
alpha = 0.0
beta = 0.0
x,w_nu = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Differential matrices
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(Nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol = 1
max_iter = 1
```

```
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

while tol> 10**(-6) and max_iter < 10**5:
    t1 = time()
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
               tuple([Nu_expan,D_expan,D2_expan,B]))
    t3 = time()-t1
    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol  = max(np.abs(u_init_expan-u_check))

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check

    max_iter += 1

# Save the steady state solution.
u_sol = u_check.reshape(n_space+1,new_dim,order='F')

# Calculates the statistical parameters.
mu_estimate = np.sum(W*u_sol,axis=1)
var_estimate = np.sum(W*\
                ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

B.4 Sparse grid implementations and tests

Code construction two dimensional full grid and tensor grid.

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
```

```
import scipy.sparse as sparse
from cartesian import *

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

def init(x):
    return -x

def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V

def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx

def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*np.dot(D,u) + nu*np.dot(D2,u)
    u = u*B
    return u
# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 29
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu
nu = 0.05

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

# Number of boundary nodes
n_expan = 10
z1,w1 = legendrequad(n_expan-1)
z2,w2 = legendrequad(n_expan-1)
```

```
#x1 = np.array([0.0,-1.0,1.0,0.0,0.0,-np.sqrt(0.5),np.sqrt(0.5)\
#,-1.0,1.0,-1.0,1.0,0.0,0.0])
#x2 = np.array([0.0,0.0,0.0,-1.0,1.0,0.0,0.0,-1.0,-1.0,1.0,1.0,\
#-np.sqrt(0.5),np.sqrt(0.5)])

y1 = np.array([-0.774596669241483,0.0,0.774596669241483,0.0,0.0,0.0,\
-0.949107912342758,-0.741531185599394,-0.405845151377397,0.405845151377397,\
0.741531185599394,0.949107912342758,-0.774596669241483,0.774596669241483,\
-0.774596669241483,0.774596669241483,0.0,0.0,0.0,0.0,0.0,0.0])

y2 = np.array([0.0,0.0,0.0,-0.774596669241483,0.0,0.774596669241483,0.0\
,0.0,0.0,0.0,0.0,0.0,-0.774596669241483,-0.774596669241483,\
0.774596669241483,0.774596669241483,-0.949107912342758,-0.741531185599394,\
-0.405845151377397,0.405845151377397,0.741531185599394,0.949107912342758])


w1 = w1/2
w2 = w2/2

# Transform from interval [-1,1] to [a,b]
delta1 = (b1-a1)/2*z1 + (b1+a1)/2
delta2 = (b2-a2)/2*z2 + (b2+a2)/2

z_temp = cartesian((z1,z2))

plot(z_temp[:,0],z_temp[:,1],'.')

delta_temp = cartesian((delta1,delta2))

Delta1 = delta_temp[:,0]
Delta2 = delta_temp[:,1]

Delta1 = (b1-a1)/2*y1 + (b1+a1)/2
Delta2 = (b2-a2)/2*y2 + (b2+a2)/2

W_temp = cartesian((w1,w2))

W1 = W_temp[:,0]
W2 = W_temp[:,1]

W = W1*W2

#new_dim = (n_expan-1)**2
new_dim = len(Delta1)
# To the deterministic solver
alpha = 0.0
beta = 0.0

x,w_nu = JacobiGL(alpha,beta,n_space)
```

```
V = vanderMat(x)
Vx = vanderMatx(x)

D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

D_expan = np.kron(np.identity(new_dim),D)
D2_expan = np.kron(np.identity(new_dim),D2)

u_init = init(x)

u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

tol = 1
max_iter = 1
dt = 0.1
nt_jump = 1000
t_end = dt*nt_jump
t_span = np.linspace(0,t_end,nt_jump)

while tol> 10**(-6) and max_iter < 10**5:
    u = odeint(rhs_SCM,u_init_expan,t_span,\
               tuple([nu,D_expan,D2_expan,B]))
    u_check = u[-1,:]
    tol = max(np.abs(u_init_expan-u_check))
    print tol

    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check

    max_iter += 1


u_sol = u_check.reshape(n_space+1,new_dim,order='F')

Wx = np.array([-0.0888888888888889,-0.0222222222222222,\
-0.0222222222222222,-0.0222222222222222,-0.0222222222222222\
,0.266666666666667,0.266666666666667,0.0277777777777778,\
0.0277777777777778,0.0277777777777778,0.0277777777777778,\
0.266666666666667,0.266666666666667])

Wy = np.array([-0.617283950617284,0.684182413706223,-0.617283950617284,\
-0.617283950617284,-1.77777777777778,-0.617283950617284,\
0.258969932337739,0.559410782978553,0.763660101010238,0.763660101010238,\
0.559410782978553,0.258969932337739,0.308641975308642,0.308641975308642,\
0.308641975308642,0.308641975308642,0.258969932337739,0.559410782978553,\
0.763660101010238,0.763660101010238,0.559410782978553,0.258969932337739])/4

mu_estimate = np.sum(Wy*u_sol,axis=1)
var_estimate = np.sum(Wy*\
                  ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

The implementation with full tensor grid for the stochastic Burger's equation where the 3 random variables follows an Uniform distribution.

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
import scipy.sparse as sparse
from cartesian import *

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Initial condition function.
def init(x):
    return -x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx
# Right hand side function
```

```
def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

# Statistical parameters for v.
a3 = 0.05
b3 = 0.35

# Number of nodes representing random variables
n_expan = 7
z1,w1 = legendrequad(n_expan)
z2,w2 = legendrequad(n_expan)
z3,w3 = legendrequad(n_expan)

# Transforming weights.
w1 = w1/2
w2 = w2/2
w3 = w3/2

# Transform from interval [-1,1] to [a,b]
delta1 = (b1-a1)/2*z1 + (b1+a1)/2
delta2 = (b2-a2)/2*z2 + (b2+a2)/2
nu = (b3-a3)/2*z3 + (b3+a3)/2
# Makes all possible combinations between all abscissas.
delta_temp = cartesian((delta1,delta2,nu))
Delta1 = delta_temp[:,0]
Delta2 = delta_temp[:,1]
Nu = delta_temp[:,2]

# Makes all possible combinations between all weights.
W_temp = cartesian((w1,w2,w3))
```

```
W1 = W_temp[:,0]
W2 = W_temp[:,1]
W3 = W_temp[:,2]

W = W1*W2*W3

# Determine the total number of deterministic systems that have to be
# solved.
new_dim = (n_expan)**3

# To the deterministic solver
alpha = 0.0
beta = 0.0
x,w_nu = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Differential matrices
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(Nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol = 1
max_iter = 1
```

```
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

t1 = time()
while tol> 10**(-6) and max_iter < 10**5:
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
                tuple([Nu_expan,D_expan,D2_expan,B]))

    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol = max(np.abs(u_init_expan-u_check))

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check
    print 'tol:',tol,'  index:',n
    max_iter += 1
Timing[n,0] = time()-t1
# Save the steady state solution.
u_sol = u_check.reshape(n_space+1,new_dim,order='F')

# Calculates the statistical parameters.
mu_estimate = np.sum(W*u_sol,axis=1)
var_estimate = np.sum(W*\
                ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

The implementation with sparse grid for the stochastic Burger's equation where the 3 random variables follows an Uniform distribution.

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
import scipy.sparse as sparse
from cartesian import *
import scipy.io as sio
```

```
hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

mat = sio.loadmat('C:/Users/miv_kjaer/Dropbox/Kun mig − speciale UQ/\
Python/Methods/SCM/Burgers equation/Multidimensional/sparsegrid3d.mat')
data = mat['zw']

y1 = data [:,0]
y2 = data [:,1]
y3 = data [:,2]
W = data[:,3]/8

# Initial condition function.
def init (x):
    return −x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N−1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N−1)
    return Vx
# Right hand side function
def rhs_SCM(u,t,nu,D,D2,B):
    u = −u∗D.dot(u) + nu∗D2.dot(u)
    u = u∗B
    return u

# Number of time steps
x_left = −1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right−x_left)/n_space
# The parameter nu

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1
```

```
a2 = 0.0
b2 = 0.1

a3 = 0.05
b3 = 0.35

# Transform from interval [-1,1] to [a,b]
Delta1 = (b1-a1)/2*y1 + (b1+a1)/2
Delta2 = (b2-a2)/2*y2 + (b2+a2)/2
nu = (b3-a3)/2*y3 + (b3+a3)/2

# Determine the total number of deterministic systems that have to be
# solved.
new_dim = (len(y1))

# To the deterministic solver
alpha = 0.0
beta = 0.0
x,w_nu = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Differential matrices
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
```

```
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol = 1
max_iter = 1
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

while tol> 10**(-6) and max_iter < 10**5:
    t1 = time()
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
               tuple([Nu_expan,D_expan,D2_expan,B]))
    t3 = time()-t1
    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol = max(np.abs(u_init_expan-u_check))

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check

    max_iter += 1

# Save the steady state solution.
u_sol = u_check.reshape(n_space+1,new_dim,order='F')

# Calculates the statistical parameters.
mu_estimate = np.sum(W*u_sol,axis=1)
var_estimate = np.sum(W*\
                 ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

The implementation of the timings for the full tensor grid.

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
```

```
from time import *
import scipy.sparse as sparse
from cartesian import *

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

# Initial condition function.
def init(x):
    return -x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx
# Right hand side function
def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

# Statistical parameters for v.
a3 = 0.05
b3 = 0.35
```

```
N = range(2,8)
Timing = np.zeros((len(N),1))
mu_e = np.zeros((len(N),n_space+1))
std_e = np.zeros((len(N),n_space+1))
# Number of nodes representing random variables
for n in range(2,8):
    n_expan = n
    z1,w1 = legendrequad(n_expan)
    z2,w2 = legendrequad(n_expan)
    z3,w3 = legendrequad(n_expan)

    # Transforming weights.
    w1 = w1/2
    w2 = w2/2
    w3 = w3/2

    # Transform from interval [-1,1] to [a,b]
    delta1 = (b1-a1)/2*z1 + (b1+a1)/2
    delta2 = (b2-a2)/2*z2 + (b2+a2)/2
    nu = (b3-a3)/2*z3 + (b3+a3)/2
    # Makes all possible combinations between all abscissas.
    delta_temp = cartesian((delta1,delta2,nu))
    Delta1 = delta_temp[:,0]
    Delta2 = delta_temp[:,1]
    Nu = delta_temp[:,2]

    # Makes all possible combinations between all weights.
    W_temp = cartesian((w1,w2,w3))
    W1 = W_temp[:,0]
    W2 = W_temp[:,1]
    W3 = W_temp[:,2]

    W = W1*W2*W3

    # Determine the total number of deterministic systems that have to be
    # solved.
    new_dim = (n_expan)**3

    # To the deterministic solver
    alpha = 0.0
    beta = 0.0
    x,w_nu = JacobiGL(alpha,beta,n_space)

    # Determine the Vandermonde matrices.
    V = vanderMat(x)
    Vx = vanderMatx(x)

    # Differential matrices
```

```
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(Nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0]  = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol  = 1
max_iter = 1
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

t1 = time()
while tol> 10**(-6) and max_iter < 10**5:
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
            tuple([Nu_expan,D_expan,D2_expan,B]))

    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol  = max(np.abs(u_init_expan-u_check))
```

```
        # Update t_span and the initial condition
        t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
        u_init_expan = u_check
        print 'tol:',tol,'  index:',n
        max_iter += 1
    Timing[n-2,0] = time()-t1
    # Save the steady state solution.
    u_sol = u_check.reshape(n_space+1,new_dim,order='F')

    # Calculates the statistical parameters.
    mu_estimate = np.sum(W*u_sol,axis=1)
    mu_e[n-2,:] = mu_estimate
    var_estimate = np.sum(W*\
                    ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
    std  = np.sqrt(var_estimate)

    std_e[n-2,:] = std
```

The implementation of the measured time for sparse grids.

```
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import matplotlib.pyplot as plt
from pylab import *
from scipy.integrate import odeint
import legendrequad
import jacobipol
import hermitequad
from time import *
import scipy.sparse as sparse
from cartesian import *
import scipy.io as sio

hermitequad = hermitequad.hermitequad
legendrequad = legendrequad.legendrequad
JacobiP = jacobipol.JacobiP
GradJacobiP = jacobipol.GradJacobiP
JacobiGL = jacobipol.JacobiGL

mat = sio.loadmat('C:/Users/miv_kjaer/Dropbox/Kun mig - speciale UQ/\
Python/Methods/SCM/Burgers equation/Multidimensional/sparsegridd3k3.mat')
data = mat['zw']

y1 = data[:,0]
y2 = data[:,1]
y3 = data[:,2]
W = data[:,3]/8

# Initial condition function.
def init(x):
    return -x
# Vandermonde matrix function
def vanderMat(x):
    N = len(x)
    V = JacobiP(x,0.0,0.0,N-1)
    return V
# Differentiated vandermonde matrix function
def vanderMatx(x):
    N = len(x)
    Vx = GradJacobiP(x,0.0,0.0,N-1)
    return Vx
# Right hand side function
def rhs_SCM(u,t,nu,D,D2,B):
    u = -u*D.dot(u) + nu*D2.dot(u)
    u = u*B
    return u

# Number of time steps
x_left = -1.0
# Right spacial bound
x_right = 1.0
# Number of spacial steps
n_space = 39
# Space step
dx = (x_right-x_left)/n_space
# The parameter nu

# Interval of the uniform boundary
a1 = 0.0
b1 = 0.1

a2 = 0.0
b2 = 0.1

a3 = 0.05
b3 = 0.35

# Transform from interval [-1,1] to [a,b]
Delta1 = (b1-a1)/2*y1 + (b1+a1)/2
Delta2 = (b2-a2)/2*y2 + (b2+a2)/2
nu = (b3-a3)/2*y3 + (b3+a3)/2

# Determine the total number of deterministic systems that have to be
# solved.
new_dim = (len(y1))

# To the deterministic solver
alpha = 0.0
beta = 0.0
```

```
x,w_nu = JacobiGL(alpha,beta,n_space)

# Determine the Vandermonde matrices.
V = vanderMat(x)
Vx = vanderMatx(x)

# Differential matrices
D = np.linalg.solve(V.T,Vx.T).T
D2 = np.dot(D,D)

# Expand the differential matrices.
D_expan = sparse.kron(sparse.eye(new_dim,new_dim),D)
D2_expan = sparse.kron(sparse.eye(new_dim,new_dim),D2)

# Initial condtion.
u_init = init(x)
# Expand the nu parameter.
Nu_expan = np.repeat(nu,n_space+1)

# Expand the initial condtion.
u_init_expan = np.tile(u_init,new_dim)
u_init_expan[0:-1:(n_space+1)] =\
        u_init_expan[0:-1:(n_space+1)] + Delta1
u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] = \
                u_init_expan[n_space:(n_space+1)*new_dim:n_space+1] + Delta2

# Boundary slope condition.
B = np.ones((n_space+1,1))
B = B[:,0]
B[0] = 0.0
B[-1] = 0.0
B = np.tile(B,new_dim)

# Initial tolerance, which must be too big.
tol = 1
max_iter = 1
# Time step
dt = 0.1
# Number of time jumps
nt_jump = 1000
# The last t in the first t_span
t_end = dt*nt_jump
# Create the t_span
t_span = np.linspace(0,t_end,nt_jump)

t1 = time()
while tol> 10**(-6) and max_iter < 10**5:
    # Solve the system in the next nt_jump time steps
    u = odeint(rhs_SCM,u_init_expan,t_span,\
                tuple([Nu_expan,D_expan,D2_expan,B]))
    t3 = time()-t1
    # Find the difference between the solution to the first and the last
    # element in t_span
    u_check = u[-1,:]
    tol = max(np.abs(u_init_expan-u_check))
    print tol

    # Update t_span and the initial condition
    t_span = np.linspace(t_end*max_iter,t_end*(max_iter+1),nt_jump)
    u_init_expan = u_check

    max_iter += 1

# Save the steady state solution.
t3 = time()-t1
u_sol = u_check.reshape(n_space+1,new_dim,order='F')

# Calculates the statistical parameters.
mu_estimate = np.sum(W*u_sol,axis=1)
var_estimate = np.sum(W*\
                 ((u_sol-np.tile(mu_estimate,[new_dim,1]).T))**2,axis=1)
std = np.sqrt(var_estimate)
```

Bibliography

[1] O. P. Le Maître and O. M. Knio. *Spectral Methods for Uncertainty Quantification.* Springer, 2010.

[2] Dongbin Xiu. *Numerical Methods for Stochastic Computations: A Spectral Method Approach.* Princeton University Press, 2010.

[3] Allan P. Engsig-Karup. *Slides for 02689 – polynomial methods*, 2009.

[4] Lars Eldén, Linde Wittmeyer-Koch, and Hans Brunn Nielsen. *Introduction to Numerical Computation - analysis and MATLAB illustrations.* Studentlitteratur, 2004.

[5] John A. Gubner. *Gaussian Quadrature and the Eigenvalue Problem*, 2009.

[6] Murray R. Spiegel and John Liu. *Mathematical Handbook of Formulars and Tables.* Schaum's Outlines, 1999.

[7] Alan C. Hindmarsh and Krishnan Radhakrishnan. *Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations.* NASA Reference Publication, 1993.

[8] John Jakeman. *General procedure of Storchastic Galerkin Methods.* `http://maths-people.anu.edu.au/~jakeman/QuantifyingUncertainty/Tutorials/SGtutorial.html`.

[9] Dongbin Xiu and Jan S. Hesthaven. *High-Order Collocation Methods for Differential Equation with Random Inputs.* SIAM J. Sci. Comput., 2005.

[10] M. T. Reagan, H. N. Najm, B. J. Debusschere, O P Le Maire, O. M. Knio, and R. G. Ghanem. *Spectral Stochastic Uncertainty Quantification in Chemical Systems.* `http://lmee.univ-evry.fr/~olm/biblio_dwnload/reagan_uq.pdf`.

[11] Liang Yang, Ling Guo, and Dongbin Xiu. *Stochastic Collocation Algorithms Using ℓ_1-minimization*, 2012.

[12] Guang Lin. Uncertainty quantification algorithms, analysis and applications for high dimensional stochastic pde systems.

[13] Jasmine Foo and George Em Karniadakis. *Multi-element probabilistic collocation method in high dimensions*. Elsevier, 2009.

[14] John Burkardt. *Sparse Grid Collocation for Uncertaincy Quantififation*. `http://people.sc.fsu.edu/~jburkardt/presentations/scala_2012.pdf`. Accessed: 2013-05-01.

[15] Stackoverflow. *Cartesian Python code*. `http://stackoverflow.com/questions/1208118/using-numpy-to-build-an-array-of-all-combinations-of-two-arrays`. Accessed: 2013-04-22.

[16] John Burkardt. *Sparse Grid Based on Gauss-Legendre Rules*. `http://people.sc.fsu.edu/~jburkardt/f_src/sparse_grid_gl/sparse_grid_gl.html`. Accessed: 2013-04-30.