Bachelorarbeit in Informatik

Uncertainty Quantification in Incompressible Flow using Sparse Grids

Friedrich Menhorn
Uncertainty Quantification in Incompressible Flow using Sparse Grids

Author: Friedrich Menhorn
Supervisor: Prof. Dr. Hans-Joachim Bungartz
Advisor: Dr. Tobias Neckel
Submission Date: June 16, 2014
I assure the single handed composition of this bachelor’s thesis only supported by declared resources.

Munich, June 16, 2014

Friedrich Menhorn
Acknowledgments

First, I would like to thank my advisor Dr. Tobias Neckel, who not only suggested this topic but was also always on hand with help and advice whenever I got stuck in a one way street. His feedback and topic related discussions always gave me new ideas for possible tasks. Furthermore, I would like to thank my supervisor Prof. Dr. Hans-Joachim Bungartz for giving me the opportunity to write this thesis at the Chair of Scientific Computing.

Additionally, I thank Alfredo Parra from the Chair of Scientific Computing and some members of the Institute for Parallel and Distributed Systems at the University of Stuttgart, namely Fabian Franzelin and Jun.-Prof. Dirk Pflüger who supported my thesis and invited me over for a day to discuss my work. With the help of their feedback the quality of this thesis improved a lot.

At last, but surely not least, I would like to thank my family who has always been supporting me. They encouraged me to try my best and always provided me with the freedom to do so, though maybe sometimes mowing the meadow was already long overdue in their eyes.
Abstract

Uncertainty Quantification is receiving evermore interest in many different fields of simulation of physical phenomena due to the increased introduction of random model data. This thesis investigates the upcoming topic of Uncertainty Quantification in the field of computational fluid dynamics with the help of Monte Carlo and different Stochastic Collocation methods, focusing on sparse grid Stochastic Collocation methods. Whereas Monte Carlo just uses multiple sample runs to approximate the distribution of the quantity of interest, Stochastic Collocation tries to efficiently choose collocation points to approximate different interesting properties, for example the mean, via interpolation. As has been shown in other works, Monte Carlo converges only in $O\left(\frac{1}{\sqrt{n}}\right)$ where $n$ is the number of samples. It is, however, independent of the number of random parameters. Stochastic collocation methods on the other hand, although converging much faster than Monte Carlo, have to cope with the curse of dimensionality. Each added random parameter adds one dimension to the collocation space. To reduce its impact on the performance, a Stochastic Collocation method which uses sparse grids will be presented in this thesis. For the grid, we use the SG++ library, a sparse grid library in C++, implemented in Matlab. By comparing the sparse grid collocation methods we see that a higher grid level results in a decrease of the L2Error for unsteady flow. Additionally, the simulation runs which underlie uncertainty show the same behavior as the deterministic run, for stationary as well as unsteady flows. An almost identical runtime as Monte Carlo and a drastic runtime decrease compared to full grid Stochastic Collocation methods was identifiable. Furthermore, our Stochastic Collocation method can cope with many random parameters and the curse of dimensionality.
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Introduction to Fluid Dynamics</td>
<td>5</td>
</tr>
<tr>
<td>2.1. Theoretical Background</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Navier-Stokes Equations for Incompressible Flow</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1. The Reynolds Transport Theorem</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2. Conservation of Mass</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3. Conservation of momentum</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4. Navier-Stokes-Equations</td>
<td>8</td>
</tr>
<tr>
<td>2.2.5. Quickfluid</td>
<td>9</td>
</tr>
<tr>
<td>3. Sparse Grid</td>
<td>11</td>
</tr>
<tr>
<td>3.1. Theoretical Introduction to Sparse Grids</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1. Full Grid Interpolation</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2. Hierarchical Basis</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3. Sparse Grid</td>
<td>15</td>
</tr>
<tr>
<td>3.2. Spatially Adaptive Sparse Grid Toolbox SG++</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1. SG++ in Matlab</td>
<td>17</td>
</tr>
<tr>
<td>3.3. Example application for Sparse Grids</td>
<td>23</td>
</tr>
<tr>
<td>4. Uncertainty Quantification</td>
<td>27</td>
</tr>
<tr>
<td>4.1. Probabilistic framework</td>
<td>27</td>
</tr>
<tr>
<td>4.2. Monte Carlo</td>
<td>28</td>
</tr>
<tr>
<td>4.3. Stochastic Collocation with sparse grids</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1. Monte Carlo quadrature</td>
<td>31</td>
</tr>
<tr>
<td>4.3.2. Sparse Grid quadrature</td>
<td>32</td>
</tr>
<tr>
<td>5. Results</td>
<td>33</td>
</tr>
<tr>
<td>5.1. Benchmark scenario</td>
<td>33</td>
</tr>
<tr>
<td>5.2. Results for RE=20</td>
<td>36</td>
</tr>
<tr>
<td>5.2.1. Drag coefficient</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2. Lift coefficient</td>
<td>36</td>
</tr>
<tr>
<td>5.3. Results for RE=100</td>
<td>38</td>
</tr>
<tr>
<td>5.3.1. Drag coefficient</td>
<td>38</td>
</tr>
<tr>
<td>5.3.2. Lift coefficient</td>
<td>40</td>
</tr>
</tbody>
</table>
1. Introduction

Uncertainty Quantification examines the influence of uncertainty in input parameters on the output. This topic has received an increased amount of interest over the last decade. One application of Uncertainty Quantification lies in the field of computational fluid dynamics. Whereas on its deterministic side many models have been developed and a lot of research has been done, uncertainty only slowly finds its way into research. Random components, however, can be found in many instances of a fluid dynamic simulation. These may come from various sources:

1. inaccuracy of initial conditions (experimental data, variability of operating environment)
2. bias/measurement errors in physical models
3. simplified assumptions in mathematical models (simplification or neglection of physical effects)
4. uncertainties in describing the physical reality (errors in geometry, boundary conditions)
5. discretization errors (round-off, algorithmic errors)

A more detailed review of sources of uncertainty in computational fluid dynamics can be found for example in [6] or [21].

These errors have to be taken into account if we want to determine the output of the model. Regarding a flow channel, for example, the input velocity may not be exactly known and varies around the desired value. The physical properties of the fluid, like its viscosity or density may furthermore be measured inaccurately or the friction at the borders of the channel was not exactly determined. This may lead to chaotic output values in the real life experiment, which do no match the computed deterministic simulation. As a result, Uncertainty Quantification is an important field of research to model physical phenomena accurately. In this work, we will concentrate on uncertainty in initial conditions, namely for example the inlet velocity or the density.

The most traditional Uncertainty Quantification method is Monte Carlo. Based on the assumption that a parameter has a specific random distribution, one takes samples of this distribution and runs a now deterministic simulation with each of these. Afterwards, the output can be examined with respect to different attributes, as for example its mean or variance. The Monte Carlo method offers some nice properties. First, it is rather straightforward to implement as we will see later. Second, it uses deterministic sample runs. That means we do not have to change the model itself. Third, the convergence rate does not
1. Introduction

depend on the number of independent random parameters [19]. Unfortunately, the convergence rate does depend on the number of samples with $O\left(\frac{1}{\sqrt{n}}\right)$ where $n$ is the number of samples [36]. That means that it becomes very expensive to reach a high accuracy, since a high number of sample runs are needed. There are some adjustments to the Monte Carlo methods, like Latin Hypercube Monte Carlo which try to tackle these problems and shall be mentioned here. Description about how to use these adapted Monte Carlo methods can for example be found in [13].

The Stochastic Collocation method was first introduced by [18] as a derivation of the Stochastic Galerkin method, both being based on the Polynomial Chaos Expansion. The theory of Polynomial Chaos originated from the work of Wiener in 1938 [32]. It was first applied mainly to the field of structural mechanics, but makes its way now into other fields like fluid dynamics. It was for example used by [31] and further developed by [37], so that it can be employed on different probability distributions. Many other works are also referring to this method, namely [38], [16] and [14]. We refer to these papers to get more information about the Stochastic Galerkin methods and Polynomial Chaos Expansion, since this paper focuses on the Stochastic Collocation method.

To this end, the Stochastic Galerkin method seemed superior to other non-sampling methods and often to several sampling methods. Its disadvantage, however, is that it is intrusive. That means, for complicated governing equations the approach becomes non-trivial and hard to implement. On the other hand, there are non-intrusive straightforward methods -like Monte Carlo- which are easy to implement, but have a slow convergence rate. The Stochastic Collocation method tries to combine the best of both sides, the good convergence and efficiency of Stochastic Galerkin and the non-intrusive approach of sampling methods like Monte Carlo. The idea is to approximate the stochastic space of the random parameter via an Polynomial Chaos interpolation with intelligently picked sample points. Afterwards, the sample points are used for independent solution runs, just as in Monte Carlo, and interesting quantities can be evaluated with the help of the interpolation.

In this thesis, we take another approach towards Stochastic Collocation than Polynomial Chaos Expansion. We use a sparse grid to approximate the solution function via a sparse grid interpolation. Using a sparse grid is advantageous, since for a high number of random parameters, the interpolation space becomes high dimensional and a full grid interpolation runs into the curse of dimensionality. As has for example been described in [4] and [26], sparse grid can cope better with high dimensions and still loses only slightly to the exactness of full grid. In specific, whereas the number of function evaluations decreases from $O(h_n^{-d})$ to $O(h_n^{-1}(\log h_n^{-1})^{d-1})$, error only increases from $O(h_n^2)$ to $O(h_n^2(\log h_n^{-1})^{d-1})$, where $h_n$ is the mesh width on level $n$ and $d$ is the dimension. A sparse grid Stochastic Collocation method can therefore cope better with high dimensional random input parameters than a full grid.

The thesis is structured in the following way: We start off by introducing the theory used throughout this work. First, we will give a short outlook about computational fluid dynamics and fluid mechanics in general in section 2. We will then move on to explain the
development of sparse grids in chapter 3 and will end the theory part with a chapter about Uncertainty Quantification in section 4. In 4, we will explain the Monte Carlo as well as the sparse grid Stochastic Collocation method used in this work step by step. Next, we will have a look at the results of this thesis in section 5 and will finally end with a short outlook in chapter 6.
1. Introduction
2. Introduction to Fluid Dynamics

In this section, we will give a short introduction to the physical background of fluid dynamics in order to better understand the used simulations in the field of computational fluid dynamics. We will start off with stating the model for an incompressible flow on various physical assumptions, thereby deriving the Navier-Stokes equation as the theoretical background. Next, we will shortly introduce the Matlab application quickfluid [24], which was developed at the Chair of Scientific Computing by Dr. Tobias Neckel, and which we used for our simulation runs.

2.1. Theoretical Background

This chapter will give a short introduction to fluid dynamics since the topic itself, in all its extent, exceeds the scope of this thesis. For the interested reader, we recommend [15] or [29] for a detailed introduction to fluid mechanics. Furthermore, the work of [12] gives a more application-oriented approach, explaining numerical simulations in fluid dynamics. This chapter mainly follows [23] and [12] for the description.

Computational fluid dynamics is an important field in numerical simulations. “Fluids” thereby not only describes liquids but also gases in every possible mixture. As one can imagine, there exist multiple applications for fluid dynamics which can be examined. It affects not only the more obvious areas of civil engineering, concerning for example the flow of a river which has to be straightened, the car industry investigating the air flow and irregularities around a car, or the aircraft industry which has to study turbulences and air resistance. It is also used in medical engineering, in simulating the blood flow through the heart or blood vessels. Fluid dynamics affects our daily life and is therefore a very fascinating and important field of research.

The reason for the behaviour of fluids lies in the interaction between different fluids and the interaction between the fluid particles themselves. Furthermore, the fluid interacts with solid bodies, for example the riverbank, slowing down the flow at the border or an object in the middle of the river, forcing the fluid to flow around. Moreover, moving objects in a resting fluid influence how the fluid itself behaves. The main physical property describing these characteristics is viscosity, which describes the inner and outer friction of the fluid.

The inner friction can be illustrated by imagining the fluid as a stack of cards lying onto each other. First the stack is moving with constant velocity and the stack is formed like a cuboid. However, when the lowest cards stops, because of for example friction with the ground, the upper cards still move a little bit further due to inertial force and friction be-
2. Introduction to Fluid Dynamics

between the cards. We then speak of laminar flow, when the fluid can be seen as a stack of layers which run in parallel, opposed to turbulent flow where the different layers blend. This is further illustrated in figure 2.1. An example for laminar flow due to high viscosity is honey.

The flow parameter describing the influence of viscosity and inertial force is called Reynolds number. The Reynolds number is named after the Physician O. Reynolds and depends on the fluids, as well as the velocity of the flow and the flow domain. The exact definition will be introduced later in this chapter. Another important property of a fluid is its compressibility. Everybody has already made the experience that compressing a plastic bottle filled only with air is much easier than compressing one filled with water. Of course, a fluid which is compressible has other characteristics than one which is not or only slightly. Regarding the physics, by compressing the fluid we change its density, as well as its pressure and temperature. In this thesis, we will use a physical model, which is based on incompressible fluids.

There are two different ways in fluid mechanics to describe a flow, the Lagrangian and the Eulerian view. The Lagrangian point of view follows a specific fluid particle and describes its properties. The Lagrangian point of view is therefore also called material or material related, representing for example the velocity and location of the particle at every point in time. The Eulerian point of view, on the other hand, observes a specific location. Hence, changes of properties like the velocity for a specific point in the domain can be related to two different events. Either to changes of properties of the currently observed particle at this location, or to material differences between two different particles traversing this location at two different points in time. The correlation between these two different views will be developed in subsection 2.2.1 introducing the Reynolds Transport Theorem.

For the visualisation of fluid flows, streamlines and pathlines play an important role. Thereby, pathlines are strongly connected to the Lagrangian point of view, illustrating the path which one particle takes through the domain. Streamlines, on the other hand, depict the Eulerian point of view, being defined as integral curves of the velocity field. They are tangential to the velocity vector at every location in the domain. In the case of the steady-state flow, that means a flow with no turbulences, where the fluid reaches a stationary state, the definitions of pathlines and streamlines are equivalent. We will use streamlines to illustrate our scenario in the upcoming chapters. Yet first, we will introduce the physical model leading to the Navier-Stokes equations.

![Figure 2.1: a) Laminar flow in channel. b) Turbulent flow in channel](image)
2.2. Navier-Stokes Equations for Incompressible Flow

The following chapter will shortly derive and introduce the partial differential equations which mathematically describe incompressible flow: the Navier-Stokes equations. For the description we mainly follow [12] and [23] but refer to [5] for a detailed mathematical approach. With these equations, we want to describe the incompressible flow in a domain $\Omega \subset \mathbb{R}^d$ with $d = 2, 3$, with the boundary $\Gamma = \partial \Omega$ in the time interval $t = [t_0, T] \subset \mathbb{R}_0^+$. Therefore, we will first introduce the so-called Reynolds transport theorem, which we need in order to derive the continuity and momentum equation. Finally, we present the Navier-Stokes equations in their dimensionless form.

2.2.1. The Reynolds Transport Theorem

The Reynolds transport theorem represents the relation between the Lagrangian and Eulerian view. In its general form it depicts how to calculate the time derivation of an integral over a time dependent domain

$$\frac{d}{dt} \int_{\Omega_t} f(\vec{x}, t) \, d\vec{x} = \int_{\Omega_t} \left( \frac{\partial}{\partial t} f + \text{div}(f \vec{u}) \right)(\vec{x}, t) \, d\vec{x}. \quad (2.1)$$

The first term on the right side of the equation describes the change of the regarded property $f$ in the time interval $t$. The second part describes the change of the property due to the change of the integration borders. $\Omega_t$ relates to the domain currently taken by the flow, in our case a constant volume $\Omega_c$ which matches the time dependant $\Omega_t$.

2.2.2. Conservation of Mass

The fluid, covering a specific domain $\Omega$, has a certain mass. The mass can be calculated with the help of the integral over the density. Since the fluid, which covers the domain $\Omega_0$ at the time $t = 0$, has to have the same mass as the same fluid covering $\Omega_t$ at the time $t > 0$, the following relations hold for $t \geq 0$

$$\int_{\Omega_0} \rho(\vec{x}, 0) \, d\vec{x} = \int_{\Omega_t} \rho(\vec{x}, t) \, d\vec{x} \quad (2.2)$$

and

$$\frac{d}{dt} \int_{\Omega_t} \rho(\vec{x}, t) \, d\vec{x} = 0, \forall \Omega_t, t \geq 0. \quad (2.3)$$

We now apply the Reynolds-Transport-Theorem [2.1] to [2.3] Further we use the fact that equation [2.3] has to hold for arbitrary sized domains $\Omega_t$ and, therefore, the integrand itself has to be zero. This leads to the continuity equation for compressible fluids

$$\frac{\partial}{\partial t} \rho + \text{div}(\rho \vec{u}) = 0. \quad (2.4)$$

Since we regard only incompressible flow in this work the change of the density over time is zero as well, which simplifies [2.4] to

$$\nabla \cdot \vec{u} = 0. \quad (2.5)$$
2.3 Conservation of momentum

Next, we will look at another conservation law in physics, which has to hold in fluid dynamics as well: the conservation of momentum. We will only give a short overview about the idea, which is used here. A more detailed derivation can be found in the already stated reading.

Since the velocity of the fluid also depends on the location, the momentum \( \mathbf{p} \) has to be calculated via the integral over the domain

\[
\mathbf{p} = \int_{\Omega} \rho(\mathbf{x}, t) \mathbf{u}(\mathbf{x}, t)d\mathbf{x}.
\] (2.6)

Using Newton’s second law, the derivation of \( \mathbf{p} \) is equal to the sum of acting forces \( F_i \)

\[
\frac{d}{dt} \mathbf{p} = \sum_i F_i.
\] (2.7)

These forces can either be body forces like gravitation, Coriolis force, magnetic forces or surface forces, as for example pressure or inner friction. Using these forces, and applying the transport theorem on the left side as well as using the continuity equation 2.4, we obtain the momentum equation

\[
\frac{\partial}{\partial t} \mathbf{u} + (\mathbf{u} \cdot \nabla) \mathbf{u} + \frac{1}{\rho_\infty} \nabla p = \frac{\mu}{\rho_\infty} \Delta \mathbf{u} + \mathbf{g}.
\] (2.8)

Here \( p \) is the pressure (scalar), \( \mu \) describes the dynamic viscosity and \( \rho_\infty \) is the constant density for the incompressible fluid.

2.4 Navier-Stokes-Equations

In the previous two chapters we have given a short excursion to fluid dynamics, deriving to the continuity equation and the momentum equation. These two represent the Navier-Stokes equations for unsteady, laminar flow of incompressible, viscous fluids. As recap and summary, we list them once again

\[
\frac{\partial}{\partial t} \mathbf{u} + (\mathbf{u} \cdot \nabla) \mathbf{u} + \nabla p = \frac{1}{Re} \Delta \mathbf{u} + \mathbf{g}
\]

\[
\nabla \cdot \mathbf{u} = 0.
\] (2.9)

Hereby, \( Re := \frac{\rho_\infty u_\infty L}{\mu} \in \mathbb{R} \) is the Reynolds number, where \( u_\infty \) and \( L \) depend on the scenario and are for example the dimensionless inlet velocity in a channel and the width of the channel respectively.

Having given a short theoretical insight into fluid dynamics, we will introduce the simulation we used. Therefore, we will introduce quickfluid, a Matlab program for simulating different scenarios of computational fluid dynamics in the next section.
2.2.5. Quickfluid

Quickfluid [24] is a Matlab library which offers the possibility to simulate a two dimensional laminar, incompressible fluid flow. It solves the previously described Navier-Stokes equations with the help of Cartesian grid. For discretizing space, it uses bilinear elements for the velocities, using constant pressure per cell as the corresponding Lagrangian multiplier for the continuity constraint. The time discretisation is done with the help of an explicit Euler. In each time step, the discrete pressure poisson equation is calculated, which is then used to update the velocities for the next time step. The method of consistent force is used to calculate force values, for example on obstacles. It can simulate different scenarios like driven cavity or free channel flow. Furthermore, it is possible to place obstacles in the channel, in order to simulate flow around an obstacle. The program is straightforward to use and the results can then be visualized with included methods.

In this thesis, we will work on a channel flow around an obstacle. The example scenarios are illustrated in section 5. We will work on two different scenarios with different Reynolds numbers and will use quickfluid for the simulation.
2. Introduction to Fluid Dynamics
3. Sparse Grid

Since having been developed, Grids have been a favored and efficient method to discretize a continuous space. There exist various different implementations like multigrids and adaptive grids with all kind of variations in between, each adapted for specific problems and fields of applications. Most of these methods however, efficient and precise as they are, have to tackle one problem: the curse of dimensionality. If we consider a uniform grid with a mesh size $h$ and a width of 1, there are $N = \frac{1}{h} + 1$ grid points for one dimension. The number of grid points increases exponentially with the number of dimensions, which leads do $N^d$ grid points for $d$ dimensions. Therefore it is almost impossible to tackle high-dimensional problems with the standard grid approach.

The sparse grid method was originally developed for the solution of partial differential equations introduced by Zenger in 1992 [39]. The main idea, however, was first suggested by Smolyak in 1963 [28] to overcome the curse of dimensionality. The sparse grid approach does not solve the problem completely, as we will see in the following. Nevertheless, due to less grid points per dimension, the number of function evaluations decreases drastically compared to the full grid, yet still almost preserving the accuracy rate of the standard grid. For that, it uses the hierarchical decomposition on the corresponding approximation space. After its first usage to solve partial differential equations, it has made its impact in various different fields. An overview of applications can be found in [3] or [4] in the problem of numerical quadrature, data mining or financial applications.

Section 3.1 will give a short theoretical introduction to the sparse grid method, so that the reader will be able to understand how it is used in the field of uncertainty quantification. In Section 3.2 we will present the SG++-Library. SG++ is a library written in C++ which offers the functionality of sparse grids and will be used throughout this thesis. Furthermore, we will present its implementation in Matlab via Java and the MEX-file in this section. At last, the technique is further illustrated with an example in Section 3.3.

3.1. Theoretical Introduction to Sparse Grids

In the following chapter we will give a short overview about how the sparse grid technique works. This chapter will by far not go into all details about sparse grids. We therefore refer the interested reader to [11] or [11], which give a more detailed introduction. We will follow these works and the work of [26] in this introduction as well.

The main idea about sparse grids is to apply a hierarchical decomposition by using hierarchical basis functions instead of uniform basis functions and thereby only selecting the most contributing subspaces for the approximation. Thus, there will be less grid points...
3. Sparse Grid

used than in the standard grid approach. We start off with the interpolation on a full grid
and continue to develop it using a hierarchical basis creating a sparse grid.

For this, we will introduce notations which will be helpful in the description later on.
Due to better readability, we write a vector variable \( \vec{k} := k \). We denote

\[
1 = (l_1, ..., l_d) \in \mathbb{N}^d
\]  

(3.1)

as multi-index which will describe the grid level and introduce the uniform grid \( \Omega_l \) on the
domain \( \Omega \) with mesh size

\[
h_l := 2^{-l} := (2^{-l_1}, ..., 2^{-l_d})
\]  

(3.2)

where \( \Omega_l \) has different, but equidistant mesh sizes \( h_l \) in each dimension directions \( i =
1, ..., d \). Therefore, we can describe the grid points

\[
\mathbf{x}_{l,j} = (x_{l_1,j_1}, ..., x_{l_d,j_d})
\]  

(3.3)

with

\[
x_{l,j_i} := j_i \cdot h_l = j_i \cdot 2^{-l_i}
\]  

(3.4)

and \( j_i = 0, ..., 2^{l_i} \). In addition, we define a very general function \( f : \Omega \to \mathbb{R} \), which we will
approximate by our grid. At last we introduce an index set

\[
I_l := \{ j_i \in \mathbb{N} : 1 \leq j_i \leq 2^{l_i} - 1, j_i \text{ odd} \}
\]  

(3.5)

which will be needed for the hierarchical approach in section 3.1.2. Without losing gener-
ality we restrict ourself to the \( d \)-dimensional unit-hypercube \( \Omega := [0, 1]^d \). This can be easily
expanded to any desired domain via transformation afterwards. Another simplification is
our focus on functions which are zero on the domain’s boundary \( \partial \Omega \).

3.1.1. Full Grid Interpolation

At first, we will look at the approximation \( u \) of a function \( f \) on a one-dimensional full grid.
For the full grid interpolation, we choose a specific level \( l \) which defines the mesh grid size
and therefore the accurateness. By discretizing the domain \( \Omega \) via a mesh grid size \( h \) we
get equidistant gridpoints \( x_{l,j} \) which we can simplify to \( x_j \) since all points are on the same
level. The higher the level, the more equidistant grid points exist. For the interpolation we
introduce the piecewise linear standard hat function

\[
\phi_{l,j}(x) = \phi_j(x) = \begin{cases}
  \frac{x-x_{j-1}}{x_j-x_{j-1}}, & x \in [x_{j-1}, x_j], \\
  \frac{x_j-x_{j+1}}{x_{j+1}-x_j}, & x \in [x_j, x_{j+1}], \\
  0, & \text{otherwise}.
\end{cases}
\]  

(3.6)

With the help of these standard hat functions, the function \( f \) can be approximated with
the help of the weights \( \alpha_j \) as

\[
f(x) \approx u(x) := \sum_j \alpha_j \phi_j(x).
\]  

(3.7)
3.1. Theoretical Introduction to Sparse Grids

Figure 3.1.: The one dimensional standard hat function weighted by the function values. The exact function \( f \) is on the left side, approximated by \( u \) on the right side.

This approximation is illustrated in Figure 3.1 for an example function.

As can be seen, the standard hat functions need many grid points to approximate the function as good as possible, although one hat function alone does not contribute much to the approximation. The idea behind the sparse grid approach is now to use only the most influential grid points and basic functions from multiple levels and omit the less important ones.

3.1.2. Hierarchical Basis

We therefore introduce the hierarchical basis on a one dimensional grid

\[ \varphi_{l,j}(x) = \phi\left(\frac{x - j \cdot h_l}{h_l}\right) \]  

which can be derived from the standard hat basis

\[ \phi(x) = \begin{cases} 
1 - |x|, & \text{if } x \in [-1, 1], \\
0, & \text{otherwise}
\end{cases} \]  

by dilatation and translation, whereas its influence is now depending on its actual level and index position. For higher dimensions the basis functions are extended by the tensor product approach

\[ \varphi_{l,j}(x) = \prod_{i=1}^{d} \varphi_{l_i,j_i}(x_i). \]

With the help of these hierarchical basis functions, we denote the so-called hierarchical subspace \( W_l \)

\[ W_l := \text{span} \left( \varphi_{l,j}(x) : j_i \in I_{l_i} \right) \]
3. Sparse Grid

Figure 3.2.: Juxtaposition between the hierarchical basis function for each level (left) and its corresponding standard hat function (right). [26]

where $I_l$ is the index set defined in 3.5. For a given level $n$ the piecewise linear functions $V_n$ of a full grid with mesh width $h_n$ can be described as

$$V_n = \bigoplus|l|_\infty \leq n W_l$$

(3.12)

where $|l|_\infty$ describes the infinity norm of $l$, defined as

$$|l|_\infty := \max_{1 \leq j \leq d} |l_j|.$$ 

(3.13)

As can be seen in figure 3.2, the hierarchical basis uses only odd grid points on each level. Furthermore, on each level the hierarchical hat functions have the same width and are not overlapping, whereas the standard basis uses every point and overlaps.

This concludes in a full grid with $(2^n - 1)^d$ grid points and the interpolant

$$u(x) = \sum_{|l|_\infty \leq n, i \in I_l} \alpha_{l,i} \varphi_{l,i}(x)$$

(3.14)

where $\alpha_{l,i}$ are the so-called hierarchical surpluses which are uniquely indexed by the same level and index as the corresponding basis function.

Referencing [26] we obtain an asymptotic error decay of

$$||f(x) - u(x)||_{L_2} \in O(h_n^2)$$

(3.15)

for a sufficiently smooth $f$ and its approximation $u \in V_n$. Nevertheless, since we are still working on a full grid we have

$$O(h_n^{-d}) = O(2^{-nd})$$

(3.16)

function evaluations, resulting in the curse of dimensionality for a high number of dimensions $d$. 

14
3.1. Theoretical Introduction to Sparse Grids

Figure 3.3.: The example function approximated by hierarchical basis functions on the left and the used hierarchical basis functions on each level on the right [26].

3.1.3. Sparse Grid

To overcome this disadvantage, this section will develop the idea of sparse grid out of the previously described approach. We will now only choose those subspaces which contribute the most in the approximation of 3.14. Based on the assumption that the regarded function fulfills a certain smoothness, the mixed second derivatives

\[ D^2 f := \frac{\partial^{|l_1|}}{\partial x_1^{l_1} \cdots \partial x_d^{l_d}} f \]  

have to be bounded, which can be done by deriving bound for the contributions of the different subspaces. By formulating the optimization problem as a continuous knapsack problem, we obtain the sparse grid space

\[ V_n^{(1)} := \bigoplus_{|l|_1 \leq n+d-1} W_l \text{[26].} \]  

As one can observe, we leave out the subspaces with many basis functions of small support. That is illustrated in Fig. 3.4 which shows the used subspaces for a two-dimensional grid colored black, the left out ones colored grey. It also shows the constructed grid for a level sparse grid on the right side, which has a significant lower amount of grid points compared to the full grid which would also include the grey subspaces.

The function can now be approximated with \( u(x_{1,1}) \in V_n^{(1)} \), as

\[ u(x_{1,1}) = \sum_{|l|_1 \leq n+d-1, i \in I_l} \alpha_l \varphi_{l,i}(x). \]  

The hierarchical surpluses \( \alpha \) can be calculated from the function values \( u = u(x_{1,1}) \) as

\[ \alpha_{1,1} = (\prod_{i=1}^{d} [-\frac{1}{2}, 1 - \frac{1}{2}]_{x_i, j_i, I_i}) u =: (\prod_{i=1}^{d} I_{x_i, j_i, I_i}) u =: I_{x_{1,1}} u. \]  

The function can now be approximated with \( u(x_{1,1}) \in V_n^{(1)} \), as
The quadrature formula used in sparse grids is based on the studies of Smolyak [28] and has the following form

\[ Q_n^{(d)} f := \left( \sum_{j=0}^{n} (Q_j^{(1)} - Q_j^{(1)}_{-1}) \right) \otimes Q_{n-1}^{(d-1)} f \]  

(3.21)

where \( Q_n^{(d)} \) denotes a \( d \)-dimensional quadrature formula based on the 1D rule \( Q_n^{(1)} \).

In conclusion, the sparse grid approach reduces the number of function evaluations from \( O(h_n^{-d}) \) to

\[ O(h_n^1 (\log h_n)^{-d+1}) \]  

(3.22)

and the asymptotic accuracy only decreases by a small number from \( O(h_n^2) \) to

\[ O(h_n^2 (\log h_n)^{-d+1}) \]  

(3.23)

as described in more detail in [4].

3.2. Spatially Adaptive Sparse Grid Toolbox SG++

The SG++ Toolbox was and is currently developed at the Institute for Parallel and Distributed Systems at the Universität Stuttgart in collaboration with the Department of Scientific Computing at the TU München. Its documentation, releases and contacts can be found in [25]. The toolbox is developed in C++ and can be used to implement sparse grids in many possible ways for many different applications. There are many different features offered by this toolbox, which can all be found in [25]. In the following we will name a few:
3.2. Spatially Adaptive Sparse Grid Toolbox SG++

- Different types of basis functions are supported which enable to cope with the characteristics of special applications.
- Modern shared-memory multi-core systems are supported via OpenMP to speed-up computations.
- Adaptive refinement criteria can easily be specified as functor objects.
- The underlying data structure (which is, per default, hash-based) is encapsulated so that it can be exchanged by other choices.
- Template-based support for fast realizations of typical sparse-grid algorithms that make use of the tensor product structure of the underlying basis.

It also offers libraries for the usage in Python, Java and Matlab. The implementation of these libraries was also part of this thesis. With the aid of a simple example, we will not only show the usage of SG++ in Matlab, but also deepen the understanding about sparse grids.

3.2.1. SG++ in Matlab

In this section, we will give a short introduction to using SG++ in Matlab by approximating the function

$$f = 16.0 \cdot (x_0 - 1) \cdot x_0 \cdot (x_1 - 1) \cdot x_1.$$  \hspace{1cm} (3.24)

In order to include the SG++ library in Matlab, there are two possibilities. Since SG++ is developed in C++, we can use the Matlab MEX-file [20] for direct implementation of C++ code. The second way of implementing SG++ in Matlab is by using a small detour. SG++ offers a library for Java, which can then be used in Matlab. In the following two subsections, we will introduce both ways of accessing the Sparse Grid library.

Implementation of SG++ in Matlab via Java

Matlab offers the possibility to include Java .class files or .jar archives and access these files with a Java syntax. The following chapter will show how to prepare SG++ for the usage in Matlab via Java and how to use it with the aid of the same example as in the previous section.

In the first step, we have to compile the SG++ Java library, with for example

`scons SG_ALL=1`

Listing 3.1: Compiling the Java SG++ library

creating all modules or

`scons JSGPP=True JNI_CPPPATH=<jnipath> JNI_OS=linux`

Listing 3.2: Compiling the Java SG++ library

compiling only the Java library, as described in [25]. It is necessary to call the method

```
3. Sparse Grid

```
System.loadLibrary("jsgpp");
```

Listing 3.3: System.loadLibrary() to load the jsgpp.so library

In Java to be able to access the library. In Matlab, to use the toolbox, this method cannot be called directly, as is explained in [1]. Thus, we first have to create a Java class which holds a method that calls System.loadLibrary(). Therefore, we change to the directory which holds the .java files of the SG++ library

```
/Path/To/SGpp/trunk/lib/jsgpp/sgpp
```

Listing 3.4: Path to the JSGPP lib

In this directory, we create a Java class, here the file is called LoadJSGPPLib.java, with the following content:

```
package sgpp;

public class LoadJSGPPLib {
    public static void loadJSGPPLib() {
        try {
            System.loadLibrary("jsgpp");
        } catch (UnsatisfiedLinkError error) {
            System.out.println("Error loading library");
        }
    }
}
```

Listing 3.5: LoadJSGPPLib.java

As can be seen and as described above, the class LoadJSGPPLib simply holds a method loadJSGPPLib(), which calls System.loadLibrary().

After having created this class, we create the .jar archive, which we can include and access in Matlab. The archive is built with the following commands

```
> javac -d . *.java
> jar cf SGPP.jar ./sgpp/*\_.class
```

Listing 3.6: Building the SGPP.jar archive

in /Path/To/SGpp/trunk/lib/jsgpp/sgpp. Here the archive is called SGPP.jar.

For using the library, we also have to specify the library path for Matlab. This can be done by adding the entry

```
/Path/To/SGpp/trunk/lib/jsgpp
```

Listing 3.7: Adding librarypath to librarypath.txt

to the librarypath.txt file, which can be found in /Path/To/Matlab/toolbox/local.

Now that we have specified the path to the jsgpp.so library, we can load the newly created jar archive in Matlab. We can either add it in a dynamic or a static path. For the dynamic path, we call
3.2. Spatially Adaptive Sparse Grid Toolbox SG++

Listing 3.8: Dynamically loading SGPP.jar to Matlab

```
javaaddpath ('/Path/To/SGpp/trunk/lib/jsgpp/jsgpp/SGPP.jar')
```

which has to be done after every restart of Matlab. For the static alternative, we add the entry

```
/Path/To/SGpp/trunk/lib/jsgpp/jsgpp/SGPP.jar
```

Listing 3.9: Statically loading SGPP.jar to Matlab

to the classpath file `classpath.txt` in `/Path/To/Matlab/toolbox/local`. With the command

```
javaclasspath
```

Listing 3.10: javaclasspath command in Matlab

we can check if the jar archive is loaded. At last, we have to call the before created method to load the `jsgpp.so` library with the command

```
sgpp.LoadJSGPPLib.loadJSGPPLib()
```

Listing 3.11: Call to load jsgpp.so library

Right now, all methods of SG++ are usable in Matlab. We will show how to use them in an example in the next section. The whole code can be found in Appendix A.

Example for Java in Matlab

The first line shows the function head. It is called by `interfaceJava` taking the dimension, the level, the function to approximate and the condition if to use boundary grid points, returning a result array and the number of grid points used.

```
function [ result, noGridPoints ] = interfaceJava( dim, level, funct, boundaries)
```

Listing 3.12: line 1 of `interfaceJava.m`

Regarding the body of the function, we first import the packages, which we will use. As in the MEX file we can either include all packages or only the ones we really use.

```
% INTERFACEJAVA Summary of this function goes here
% Detailed explanation goes here

%import all packages
%import sgpp.*;

% or, better, include only the ones needed
import sgpp.DataVector;
import sgpp.GridGenerator;
import sgpp.GridStorage;
import sgpp.Grid;
import sgpp.GridIndex;
import sgpp.jsgpp;
import sgpp.OperationEval;
```

Listing 3.13: lines 2-15 of `interfaceJava.m`
3. Sparse Grid

Line 17 calls the method `loadJSGPPLib()` of `LoadJSGPPLib` to make sure that the jsgpp.so file is loaded.

```java
sgpp.LoadJSGPPLib.loadJSGPPLib();
```

Listing 3.14: line 17 of interfaceJava.m

The next eight lines create the grid, either with or without grid points on the boundary, specified by the parameter. As one can see, the methods are called in Java syntax.

```java
if (boundaries == 0)
    grid = Grid.createLinearGrid(dim);
else
    grid = Grid.createLinearBoundaryGrid(dim);
end
gridStorage = grid.getStorage();
dimensionality = gridStorage.dim();
```

Listing 3.15: lines 19-26 of interfaceJava.m

Next, we use the class `GridGenerator` to generate a regular grid of the stated level. The number of grid points created can be returned by calling `gridStorage.size()`.

```java
gridGen = grid.createGridGenerator();
gridGen.regular(level);
Gridlevel = level
noGridPoints = gridStorage.size();
```

Listing 3.16: lines 28-32 of interfaceJava.m

Now, after having created the grid points, we can calculate the coefficient value `alpha[i]` for each grid point `gp[i]` by first receiving the grid point with `gridStorage.get(i)`, since every point has its own index. Afterwards, we call the function `funct()` with the corresponding parameters, depending on the dimension of our problem. At the end, we save the coefficients in `alpha_array`. Attention has to be laid on the fact that Java and Matlab use different array indexing. For an array of length `j`, in Java the first index is `0`, in Matlab it is `1`. That is why, for example in line 56, `alpha_array` is indexed with `i + 1`, while the for loop starts with `0`.

```java
alpha = DataVector(noGridPoints);
alpha.setAll(0);

alpha.size = alpha.getSize();
alpha_array = nan(1, alpha.size);
alpha_array_hier = nan(1, alpha.size);
tmp_point = nan(1, dim);
for i = 0 : noGridPoints-1
    gp = gridStorage.get(i)
    for j = 0 : dim-1
        tmp_point(j+1) = gp.abs(j);
    end
    if(dim==1)
        res = funct(tmp_point(1));
```
Just as in `interface.cpp` we save the grid points to a one-dimensional array, with the corresponding length and offset. Thus, grid points coordinates of the same grid point are saved with an offset of length `noGridPoints` between them.

We can now interpolate a function for example at an arbitrary point \( p \) with the call to `opEval.eval()`.

Lastly, we fill the result array, i.e. the first \((\text{noGridPoints} \times \text{dimension})\) are the grid points coordinates, the next \(\text{noGridPoints}\) are the coefficients and the last \(\text{noGridPoints}\) are the hierarchized coefficients.
3. Sparse Grid

```latex
\texttt{result} = \texttt{nan}(1, \texttt{noGridPoints}*(\texttt{dim}+2));
\texttt{x_array}
\texttt{result}(1,1: \texttt{noGridPoints}*\texttt{dim})=\texttt{x_array};
\texttt{alpha_array}
\texttt{result}(1,(\texttt{noGridPoints}+\texttt{dim}+1): \texttt{noGridPoints}*(\texttt{dim}+1))=\texttt{alpha_array};
\texttt{alpha_array_hier}
\texttt{result}(1,(\texttt{noGridPoints}+\texttt{dim}+1)+1: \texttt{noGridPoints}*(\texttt{dim}+2))=\texttt{alpha_array_hier};
```

Listing 3.21: lines 82-88 of interfaceJava.m

**Implementation of SG++ in Matlab via MEX-file**

A MEX-file lets you call C-code in Matlab. In this example, the file which holds the MEX-function is named `interface.cpp`. For calling `interface.cpp`, we have to compile the file with the `mex` command instead of for example `gcc`, linking to the SG++ library, which we will do with a `Makefile` holding the following input: In the first two lines we specify the include path to the `sgpp_base.hpp` file and to the `libsgppbase.so` library file. The fourth line uses `mex` to compile `interface.cpp` file. The resulting file is called `interface.mexa64`.

```latex
\texttt{CFLAGS} = \texttt{OMP=true -I../SGpp/trunk/src/sgpp}
\texttt{LIBS} = \texttt{-L../SGpp/trunk/lib/sgpp}
\texttt{interface : interface.cpp}
\texttt{mex $(CFLAGS) $(LIBS) interface.cpp -o interface -lsgppbase}
```

Listing 3.22: Makefile for MEX-file interface.cpp

We can now call `interface.cpp` in Matlab. Since the behaviour of the code is the same as in the Java approach, we will not give a step by step walk through. The code can be found in AppendixB. In this section, we will only mention some specific differences of the MEX file to the Java implementation.

First, we take a look at the method head itself. The method is always named `mexFunction` and has four input parameters `int nlhs, mxArray * plhs[], int nrhs, const mxArray * prhs[]`:

- `int nlhs` describes the number of output arguments/the length of `mxArray * plhs[]`
- `mxArray * plhs[]` holds the output arguments of `mexFunction`
- `int nrhs` states the number of input arguments/the length of `mxArray * prhs[]`
- `mxArray * prhs[]` holds the input arguments of `mexFunction`

```latex
void \texttt{mexFunction}(int \texttt{nlhs}, \texttt{mxArray * plhs[]}, \texttt{int \texttt{nrhs}}, \texttt{const \texttt{mxArray * prhs[]}}){}
```

Listing 3.23: mexFunction head from line 17 of interface.cpp

Attention has to be laid on the fact that `mexFunction` is called with the name of the file, in our case

```latex
[sparseGrid, noGridPoints] = \texttt{interface(dim, level, funct, boundaries)};
```

Listing 3.24: Matlab call for `mexFunction`
3.3. Example application for Sparse Grids

Here `int nrhs = 4` and `mxArray * prhs = [dim, level, funct, boundaries]`. Respectively, `int nlhs = 2` and `mxArray * plhs = [sparseGrid, noGridPoints]`.

If we want to call a function outside of the MEX-file, defined in Matlab, we use the function call `mexCallMATLAB` with the following syntax

```c
int mexCallMATLAB(int nlhs, mxArray * plhs[], int nrhs, mxArray * prhs[], const char * functionName)
```

Listing 3.25: `mexCallMATLAB` syntax

where:

- `int nlhs` describes the number of output arguments/the length of `mxArray * plhs[]`
- `mxArray * plhs[]` holds the output arguments of `mexCallMATLAB`
- `int nrhs` states the number of input arguments/the length of `mxArray * prhs[]`
- `mxArray * prhs[]` holds the input arguments of `mexCallMATLAB`
- `const char * functionName` is a character string containing the name of the MATLAB built-in function, operator, user-defined function, or MEX-file you are calling

Here, we expect one output argument, which will be the function value at grid point `gp[i]` and we have three input arguments `mxArray * prhs[] = [function_handle, gp[0], gp[1]]. The string "feval" tells `mexCallMATLAB` to call a user-defined function.

The function header and the call to an outside defined function are the two main differences to the Java implementation. The rest of the code should be self-explanatory. Nevertheless, a detailed line by line walk through can also be found in the manuals section of [25].

3.3. Example application for Sparse Grids

After having presented two possible ways of using the SG++ toolbox in Java, we want to illustrate how well the sparse grid approach approximates the function

\[
 f(x) = f(x_0, x_1) = 16.0 \cdot (x_0 - 1) \cdot x_0 \cdot (x_1 - 1) \cdot x_1
\]

which we have already used in the previous chapters. For comparison, we use a full grid with 441, uniformly distributed grid points compared to a sparse grid of level 2 and level 6. See figure 3.5, which illustrates the different grids.

As one can see, for the sparse grid, most of the grid points are in the center. Therefore, we expect to have the main interpolation error in the sparse area of the grid. We can show this by plotting the function itself and its L2 error. See figure 3.6, for a plot of the function on the full grid and on the two sparse grids with level 2 and level 6. Already at level 2, the function is approximated quite well, yet some corners and edges are still visible. The sparse grid with level 6, however, approximates the function almost exactly and it is hard
3. Sparse Grid

Figure 3.5.: Different grids used for the approximation of function $f$. Full uniform grid on the left with 441 grid points. Sparse grid level 2 with 5 grid points in the middle and sparse grid of level 6 with 321 grid points on the right.

Figure 3.6.: Approximated function with full uniform grid, sparse grid level 2 and sparse grid level 6.

to see any difference between the two.

This can also be seen in the result of the L2 error, comparing the accurate value to the value of the sparse grid interpolation. See figure 3.7 which has plots of the L2 error of the two sparse grids. Here we can notice that already on level 2, we have an L2 error smaller $10^{-2}$ than and for level 6 the error is in the range of $10^{-5}$. That means the sparse grid is quite effective in approximating the function with a small number of grid points.

Figure 3.7.: L2 error for the two sparse grids of level 2 and level 6 with respectivly 5 and 321 grid points compared to the exact value $f((x))$ in each grid point $(x)$. 
3.3. Example application for Sparse Grids

To decrease the approximation error even further, we could now use the SG++ toolbox to for example create grid points on the boundary or use a different basis. Albeit, we will leave the example be at this point, but not without encouraging the interested reader to try the toolbox for himself, for the best way to understand the purpose and functioning of sparse grids is to work with them.
3. Sparse Grid
4. Uncertainty Quantification

As we have already stated before, Uncertainty Quantification wants to quantify the influence of input parameters, which are afflicted with randomness, on the solution. Computer simulations of real life phenomena used to use a perfectly deterministic input to simulate the outcome. Since real life experiments always underlie fluctuations, it is rather important to take these into account as well. In the last few decades, many different UQ methods were developed and introduced. In this chapter, we will introduce two important methods in this field: Monte Carlo and Stochastic Collocation. We start off by developing an underlying probabilistic framework, which we will use in the following descriptions. Afterwards, we will describe the straightforward usage of Monte Carlo. We will end this chapter by introducing our Stochastic Collocation method, which uses a sparse grid to choose its collocation points.

4.1. Probabilistic framework

The probability space is defined as \((\Omega, F, P)\). Here, \(\Omega\) is a set of all possible outcomes, \(F\) is the \(\sigma\)-algebra of subsets of \(\Omega\) and \(P\) is the probability measure on \(F\). A random variable \(X\) is a map from \(\Omega\) to the real line \(\mathbb{R}\) and is written \(X : \Omega \to \mathbb{R}\). For the case of \(d\) random variables this extends to the mapping \(X = [X_1, ..., X_d] : \Omega \to \mathbb{R}^d\). A continuous random variable is a random variable that admits a probabilistic density function \(pdf_X(x)\).

Furthermore, for independent random variables the joint probability distribution can be expressed as

\[
\text{pdf}_X(x) = \prod_{j=1}^{d} \text{pdf}_{X_j}(x_j)
\]  

(4.1)

where \(x = [x_1, ..., x_d]\). The cumulative distribution function describes the probability that \(X\) will have a value less or equal to \(x\), formally

\[
cdf_X(x) = \text{pdf}(X \leq x) = \int_{-\infty}^{x} \text{pdf}_X(t)dt.
\]  

(4.2)

In the following chapter, we consider our random parameters normally distributed, that means \(X_j \sim N(\mu_j, \sigma_j)\). Hereby \(\mu_j\) is the mean and \(\sigma_j\) the standard deviation of the random variable \(X_j\). Normal distribution can be assumed, since it represents the deviations in physical parameters the most exact.

The problem we will solve is \(Y = u(X_1, ..., X_d)\), where \(u : \mathbb{R}^d \to \mathbb{R}\) is the underlying partial differential equation, solved for example with a numerical solver. Quantities we are for example interested in are the expected value or mean of \(Y\),

\[
\mathbb{E}(Y) = \mathbb{E}(u(X_1, ..., X_d)) = \int_{\Omega} \text{pdf}_Y(y)dy
\]  

(4.3)
4. Uncertainty Quantification

and the standard deviation

$$\sigma_Y = \sqrt{Var(Y)} = \sqrt{\mathbb{E}(Y^2) - (\mathbb{E}(Y))^2}. \tag{4.4}$$

Furthermore, the expected value of a variable $Y = g(X)$ can be acquired as

$$\mathbb{E}(Y) = \mathbb{E}(g(X)) = \int_{-\infty}^{\infty} g(x) \, pdf_X(x) \, dx \tag{4.5}$$

and the variance respectively, which we will use later on.

4.2. Monte Carlo

The Monte Carlo method is a simple but effective approach for Uncertainty Quantification. It basically uses multiple simulation runs with input parameters sampled with an underlying probability distribution to empirically approximate the probability distribution of the quantity of interest. More formally speaking the method takes the following steps

1. For a previously determined number of samples $N$, generate independent and identically distributed joint random variables $X_i = [X_i^{(1)}, ..., X_i^{(d)}]$ where $d$ is the number of random variables and $i = 1, ..., N$.

2. Use these $N$ realizations to receive $N$ deterministic solutions $Y_i = u(X_i^{(1)}, ..., X_i^{(d)})$.

3. Postprocess the results to empirically evaluate the solution’s statistics, e.g. the mean $\mathbb{E}(Y) = \frac{1}{N} \sum_{i=1}^{N} Y_i$ or the variance $Var(Y) = \frac{1}{N-1} \sum_{i=1}^{N} (Y_i - \mathbb{E}(Y))^2$. \[8\]

As can be seen, the Monte Carlo approach is straightforward in its application. It is further non-intrusive since we solve $N$ deterministic solutions and do not have to interfere with the solving process itself. Additionally, its convergence just depends on the number of samples and not on the number of random variables. Unfortunately, Monte Carlo only converges with $O(\frac{1}{\sqrt{N}})$ and gets very expensive for a high number of samples, since we have to solve $u$ each time.

4.3. Stochastic Collocation with sparse grids

In general, Monte Carlo can be seen as kind of a brute force collocation method. The idea behind Stochastic Collocation, on the other hand, is to choose collocation points in the random variable space, which is usually also the input parameter space, in the best possible way to approximate the solution. Various Stochastic Collocation methods exist, for example first introduced by \[18\] based on Polynomial Chaos Expansions. Two of the major approaches of high order Stochastic Collocation methods use the Lagrange interpolation, first presented in \[36\] and \[2\] and the pseudo-spectral gPC approach, first introduced by \[34\]. A detailed summary about Stochastic Collocation methods can be found in \[35\] in general or \[22\] for CFD applications in particular.

In this thesis, we will introduce a Stochastic Collocation method based on sparse grid
interpolation using a hierarchical approach. Similar work can for example be found in [10] or [17]. The idea behind our approach is to use the sparse grid interpolation to interpolate with the aid of sample solutions. Afterwards, we will use this interpolation function to be able to approximate the solution for multiple inputs instead of having to solve many time consuming simulation runs. The process of our algorithm is shown in figure 4.1 and will be explained step by step in the following chapter.

For the sparse grid collocation we consider having $d$ random variables $V^{(j)}$ corresponding to our input parameters, which are normally distributed $V^{(j)} \sim \mathcal{N}(\mu_j, \sigma_j)$, with $j = 1, ..., d$. We will use the sparse grid points as our collocation points. Therefore we create a sparse grid with dimension $d$ and arbitrary level $l$ with the help of the SG++ library, as described in [3]. Thus, we get a sparse grid on the unit domain $[0,1]^d$ with $K$ grid points $\mathbf{x}_i = [x^{(1)}_i, ..., x^{(d)}_i]$ where $i \in 1, ..., K$ is the number of the grid point and $j \in 1, ..., d$ its corresponding dimension.

We have to use a linear transformation $T : \mathbb{R}^d \rightarrow \mathbb{R}^d$ which transforms the grid point $x_i^{(j)}$ from the unit domain to the domain corresponding to its distribution in dimension $j$. To do this, we have to pay attention to the fact that the sparse grid is defined on the unit domain $[0,1]^d$, whereas the normal distribution is defined on $(-\infty, \infty)$. Therefore, we have to use determined limits $a_j$ on the left and $b_j$ on the right for each dimension $j$ and can thus only cover a relative area $(1 - \alpha)$ of the normal distribution. An illustration of the description given above for a normal distribution around zero can be seen in figure 4.2. The blue colored sector is the area, which we cover with our sparse grid transformation, thus neglecting a fraction $\alpha$ of the distribution.

![Figure 4.1: Process diagram of our sparse grid Stochastic Collocation algorithm.](image)
4. Uncertainty Quantification

Figure 4.2.: A one dimensional Gaussian distribution around its mean 0. The blue area shows the region which we take into account for the sparse grid approximation.

Hence, we define the transformation $T$, obtaining our collocation point $v_i = [v_i^{(1)}, ..., v_i^{(d)}]$ as follows

$$ v_i^{(j)} = T_j(x_i^{(j)}) = (b_j - a_j)x_i^{(j)} + a_j. \tag{4.6} $$

We receive the borders with the help of the cumulative distribution function. We can use the property $\lim_{x \to \infty} \text{cdf}_X(x) = 1$ to obtain

$$ a_j = \text{cdf}^{-1}_{V^{(j)}}(\frac{\alpha}{2}) \tag{4.7} $$

and

$$ b_j = \text{cdf}^{-1}_{V^{(j)}}(1 - \frac{\alpha}{2}) \tag{4.8} $$

where $\text{cdf}^{-1}_{V^{(j)}}$ is the inverse cumulative distribution function corresponding to the distribution of $V^{(j)}$. After having acquired the collocation points $v_i$, we use them as sample runs, just as in the Monte Carlo method, calculating $K$ sample solutions:

$$ y_i = u(v_i) = u(v_i^{(1)}, ..., v_i^{(d)}) \tag{4.9} $$

with $u$ being the underlying partial differential equation or numerical solver. These sample solutions $y_i$ represent our interpolation coefficients and we can approximate $u$ by defining a sparse grid interpolation function as derived in [3]

$$ u(v) \approx f_{sg}(x) = \sum_{|l| \leq K + d - 1, i \in I_l} \alpha_i \varphi_l(x_i) \tag{4.10} $$

where $\varphi_l$ are the hierarchical basis functions and $x_i$ are our sparse grid points.
Having acquired an approximation of the solution, we can now compute interesting properties of its distribution. For that, we use the relation 4.5 and get

$$
\mathbb{E}(u(v)) = \int_{\mathbb{R}^d} u(v) \text{pdf}_V(v) dv \approx \int_{\mathbb{R}} f_{sg}(x) \text{pdf}_V(T(v)) dv \tag{4.11}
$$

with \( \text{pdf}_V(v) = \prod_{j=1}^d \text{pdf}_{V_j}(v) \) being the joint probability distribution, since the random variables are considered independent. Since we cannot solve the integral analytically, we have to approximate it. We have implemented two different ways to do so: Monte Carlo and sparse grid quadrature.

### 4.3.1. Monte Carlo quadrature

The Monte Carlo quadrature is defined in [27] as

$$
\int_{a}^{b} f(w) dw \approx \frac{1}{n} \sum_{i=1}^{n} \frac{f(w_i)}{pdf_W(w_i)}. \tag{4.12}
$$

Here, \( n \) is the number of samples \( w_i \), independently drawn with a distribution \( pdf_W(w) \) in the interval \([a, b]\). In our case, we randomly draw \( n \) samples of \( v_{mq,i} = [v_{mq,i}^{(1)}, \ldots, v_{mq,i}^{(d)}] \) with \( V_{mq}^{(j)} = V^{(j)} \sim \mathcal{N}(\mu_j, \sigma_j) \). Since \( f_{sg} \) is defined on the grid points \( x \in [0, 1]^d \), we have to retransform the \( v_{mq,i}^{(j)} \) points with the inverse transformation and receive \( n \) samples points \( x_{mq,i} = [x_{mq,i}^{(1)}, \ldots, x_{mq,i}^{(d)}] \) on the sparse grid

$$
x_{mq,i}^{(j)} = T_j^{-1}(v_{mq,i}^{(j)}) = \frac{v_{mq,i}^{(j)} - a_j}{b_j - a_j} \tag{4.13}
$$

with \( j \in 1, \ldots, d \). We use the relation [4.11] by first defining a helper function

$$
g(v) = f_{sg}(x) \cdot \text{pdf}_V(v) = f_{sg}(T^{-1}(v)) \cdot \text{pdf}_V(v) \tag{4.14}
$$

and get

$$
\mathbb{E}(u(v)) \approx \int_{\mathbb{R}^d} g(v) dx. \tag{4.15}
$$

By using the Monte Carlo quadrature formula [4.12] and resubstituting [4.14] we can approximate the integral

$$
\int_{\mathbb{R}^d} g(v) dx \approx \frac{1}{n} \sum_{i=1}^{n} \frac{g(v_i)}{\text{pdf}_V(v_i)} = \frac{1}{n} \sum_{i=1}^{n} \frac{f_{sg}(T^{-1}(v_i)) \cdot \text{pdf}_V(v_i)}{\text{pdf}_V(v_i)} = \frac{1}{n} \sum_{i=1}^{n} f_{sg}(T^{-1}(v_i)) \tag{4.16}
$$

and obtain the expected value of \( u(v) \). The variance can be obtained similarly with relation

$$
\text{Var}(u(v)) = \mathbb{E}(u(v)^2) - \mathbb{E}(u(v))^2. \tag{4.17}
$$
4. Uncertainty Quantification

4.3.2. Sparse Grid quadrature

We can also use a sparse grid for the quadrature of 4.11. For this we have to define a new function

\[ h(x) = f_{sg}(x) \cdot \text{pdf}_{V}(v) = f_{sg}(x) \cdot \text{pdf}_{V}(T(x)) \]  

(4.18)

which is defined on the sparse grid and get the relation

\[ \mathbb{E}(u(v)) = \int_{\mathbb{R}^d} f_{sg}(x) \text{pdf}_{V}(T(x)) dv = \text{vol} \cdot \int_{[0,1]^d} h(x) dx. \]  

(4.19)

Here, we have to multiply with \( \text{vol} = \prod_{j=1}^{d} (b_j - a_j) \), since we are changing the integration area to \([0, 1]^d\). We interpolate \( h \) now with a sparse grid with dimension \( d_{sg} = d \), arbitrary level \( l_{sg} \) and \( M \) grid points \( x_{sg,i} = [x_{sg,i}^{(1)}, \ldots, x_{sg,i}^{(d)}], i = 1, \ldots, M \) and receive

\[ h(x_{sg}) = \sum_{|l_{sg}| \leq M + d_{sg} - 1, i \in I_{l_{sg}}} \alpha_{sg,i} \varphi_i(x_{sg,i}). \]  

(4.20)

The hierarchical surpluses \( \alpha_{sg,i} \) are here computed as

\[ \alpha_{i,sg} = I_{x_{sg}} u = I_{x_{sg}}(f_{sg}(x_{sg}) \cdot \text{pdf}_{V}(v)) = I_{x_{sg}}(f_{sg}(x_{sg,i}) \cdot \text{pdf}_{V}(T_j(x_{sg,i}))) \]  

(4.21)

corresponding to the equation 3.20. The quadrature formula is then defined by

\[ \mathbb{E}(u(v)) \approx \text{vol} \cdot \int_{[0,1]^d} h(x_{sg}) dx \approx \text{vol} \cdot Q_{l_{sg}}^{(d_{sg})} h, \]  

(4.22)

also using the relation in equation 3.21.

32
5. Results

In this section, we will present the results of our sparse grid Stochastic Collocation method on a computational fluid dynamics scenario. We will compare our method with a Monte Carlo Uncertainty Quantification. Furthermore, concerning the runtime of our approach, we also compare both of them to a full grid Stochastic Collocation method. We will start off by explaining the benchmark scenario that we are using. Afterwards, we will show results about two quantities of interest, the drag and lift coefficient on the cylinder, and finish the section with a runtime and error analysis. In this whole chapter, we will refer to Uncertainty Quantification methods in figures in the following way: Monte Carlo (MC), sparse grid Stochastic Collocation with Monte Carlo quadrature (SCMC), sparse grid Stochastic Collocation with sparse grid quadrature (SCSG), full grid Stochastic Collocation (FGSG).

5.1. Benchmark scenario

As we are working in a 2D domain, we use a channel flow scenario around a cylinder as described in section 2.2 of [30]. We use this scenario to receive a comparable benchmark. The geometry and boundary conditions of the benchmark are illustrated in figure 5.1. The channel length is \( L = 2.2 \text{m} \) and the height \( H = 0.41 \text{m} \). Furthermore, the cylinder is positioned 0.15m to the right side of the inlet and 0.16m or 0.15m respectively from the top and bottom boundary. The cylinder has a diameter \( D = 0.1 \text{m} \). At the boundaries and around the cylinder itself, we applied a no slip condition, which means that \( U = V = 0 \), where \( U \) and \( V \) are the \( x \)- and \( y \)-components of the velocity respectively.

As liquid we simulated water by implementing its physical properties. Therefore, den-
5. Results

Density is set to $\rho = 1.0 \text{kg/m}^3$ and the kinematic viscosity to $\eta = 10^{-3} \text{m}^2/\text{s}$. For our inlet velocity, we used different values $u_{in}$ to receive different reynolds numbers $Re = \frac{u_{in}D}{\eta}$ and different types of flow. Pay attention to the fact that we applied uncertainty to these scenarios, which means that the above properties are representing the mean of our random input parameters $u_{in} = \mu_{u_{in}}$, $\rho = \mu_{\rho}$ and $\eta = \mu_{\eta}$.

The quantities of interest are the drag and lift coefficient $c_D$ and $c_L$ on the cylinder. We got these by computing the drag and lift forces at the cylinder for each time step with

$$F_D = \int_C (\rho \eta \frac{\partial v_t}{\partial n} n_y - p n_x) dC \quad (5.1)$$

and

$$F_L = -\int_C (\rho \eta \frac{\partial v_t}{\partial n} n_x - p n_y) dC, \quad (5.2)$$

where $C$ is the cross sectional area of the cylinder, $v_t$ is the tangential velocity on $C$ with $t = (n_y, -n_x)$, $n_x$ and $n_y$ are the normal vectors on $C$ in $x$ and $y$ respectively and $p$ is the pressure. The drag and lift coefficients can then be computed via

$$c_D = 2 \frac{F_D}{\rho u_{in} n^2 D} \quad (5.3)$$

and

$$c_L = 2 \frac{F_L}{\rho u_{in}^2 D}. \quad (5.4)$$

To test our Stochastic Collocation method on different types of flow, we simulated a flow with $RE = 20$ and one flow with $RE = 100$. To receive a Reynolds number $RE = 20$, we used a mean inlet velocity of $u_{in} = 0.2 \frac{m}{s}$. The time step size $\delta t = 0.005s$ and the number of time steps is 1000, thus giving us a time period $t = [0s, 5s]$. Figure 5.2 shows the pressure and velocity distribution in the channel in the last time step of the simulation run. For this flow, we obtained a laminar behavior and no turbulences. For the simulation run with a Reynolds number $RE = 100$, we applied a mean inlet velocity of $u_{in} = 1.0 \frac{m}{s}$. The time step size was adapted to $\delta t = 0.001s$ and with a number of time steps of 5000, we also receive a time intervall $t = [0s, 5s]$. In figure 5.3 we illustrated the pressure and velocity distribution, again in the last time step of the simulation run. In contrast, to the simulation run with $RE = 20$, we can already see some unsteadiness and the first development of turbulences.

To include uncertainty, we applied a Gaussian distribution to inlet velocity $u_{in}$ and used deterministic values for the density $\rho$ and the dynamic viscosity $\eta$. That means, for the inlet velocity we utilized a standard deviation $\sigma_{u_{in}} = 0.005 \frac{m}{s}$, whereas the standard deviation of $\rho$ and $\eta$ were set to zero. The reasoning behind such rather low standard deviation is, that it simulates uncertainties which may be for example caused by small errors in the sensors at the inlet of the channel. For the quadrature in the sparse grid Stochastic Collocation method we used a quadrature grid level 7 and 129 Monte Carlo quadrature sample points respectively. In quickfluid, we used a discretization grid of $220 \times 41$. 

34
5.1. Benchmark scenario

Figure 5.2.: Simulation benchmark scenario: Channel flow for $RE = 20$ with a cylinder in the middle of the flow. On the top, the pressure field is illustrated for the flow. The velocity field for this scenario is plotted below.

Figure 5.3.: Simulation benchmark scenario: Channel flow for $RE = 100$ with a cylinder in the middle of the flow. On the top, the pressure field is illustrated for the flow. The velocity field for this scenario is plotted below.
5. Results

5.2. Results for RE=20

To receive a Reynolds number of \( RE = 20 \), we set the mean inlet velocity \( u_{in} = 0.2\text{m/s} \). As can be seen in figure 5.4, which illustrates the drag and lift coefficient on the cylinder for the deterministic run, the forces are turbulent in the beginning but calm down quite fast to a constant value. In this section, we will show that we received similar results for the Monte Carlo and sparse grid Stochastic Collocation method.

![Deterministic run for drag coefficient](image1)

![Deterministic run for lift coefficient](image2)

Figure 5.4.: Results for the drag coefficient \( c_D \) (left) and the lift coefficient \( c_L \) (right) for a deterministic simulation run applied on the computational fluid scenario described in 5.1 with \( RE = 20 \).

5.2.1. Drag coefficient

In this section, we will compare the drag coefficient obtained by our sparse grid Stochastic Collocation method for each time step to results from a deterministic run, where the mean values of the random parameters were used as input. Additionally, we will present the results of the Monte Carlo method with different numbers of samples.

In figure 5.5, we can see the results for the Monte Carlo method on the left and the ones for our sparse grid Stochastic Collocation method on the right. As can be seen, all runs are quite close to the deterministic solution. They all show a similar behavior and there are no outliers. That means, even if our simulation or scenario underlies small fluctuations in its input, we can still expect a predictable output and no chaotic behavior.

5.2.2. Lift coefficient

Regarding the lift coefficient, we will see qualitatively similar results as for the drag coefficient. We will again look at the result for a deterministic simulation run compared to the Monte Carlo simulation and our sparse grid Stochastic Collocation method. In figure 5.6 the results for the Monte Carlo method (left) and sparse grid methods (right) are illustrated. Again, we get likewise results as for the drag coefficient, the runs all showing similar behavior as the deterministic run. The influence of uncertainty is therefore predictable and can be simulated, and no chaotic outcome has to be expected.
5.2. Results for $RE=20$

Figure 5.5.: Monte Carlo Uncertainty Quantification (left) and sparse grid Stochastic Collocation (right) applied on the computational fluid scenario described in section 5.1 for the drag coefficient $c_D$ for different sample sizes 10, 50 and 100 and different levels 1, 2, 3, 4 and $RE = 20$.

Figure 5.6.: Monte Carlo Uncertainty Quantification (left) and sparse grid Stochastic Collocation (right) applied on the computational fluid scenario described in section 5.1 for the lift coefficient $c_L$ for different sample sizes 10, 50 and 100 and different levels 1, 2, 3, 4 and $RE = 20$. 
5. Results

5.3. Results for RE=100

To receive a Reynolds number of $RE = 100$, we set the mean inlet velocity $u_{in} = 1.0 \text{ m/s}$. In contrast to section 5.2, we can see that the coefficients do not approach a constant value. As illustrated in figure 5.7, which shows again the drag and lift coefficient for the deterministic run, they oscillate around a constant value. Since we chose this high Reynolds number to receive an unsteady flow, we expected these turbulences and will now look how our Stochastic Collocation method can cope with them. We will show that we receive similar results for the Monte Carlo and sparse grid Stochastic Collocation method and that the solutions do not tend to a chaotic behavior.

![Figure 5.7: Results for the drag coefficient $c_D$ (left) and the lift coefficient $c_L$ (right) for a deterministic simulation run applied on the computational fluid scenario described in 5.1 with $RE = 100$.](image)

5.3.1. Drag coefficient

First, we will compare the drag coefficient obtained by our sparse grid Stochastic Collocation method for each time step to results from a deterministic run. Additionally, we will show the results of the Monte Carlo method with different numbers of samples.

Regarding figure 5.8, we illustrated the results for the Monte Carlo method on the left and the ones for our sparse grid Stochastic Collocation method on the right. As can be seen, the runs all show the same behavior as the deterministic run. There are, however, some solutions which lie visibly below the deterministic run.

In figure 5.9, we enlarged a section of the plot. We see that the lines lying below the deterministic runs correspond to the the sparse grid Stochastic Collocation method with sparse grid quadrature. We can furthermore see that for both quadrature methods, for a higher level the lines approach the profile of the deterministic runs, with the amplitudes of the waves increasing. The offset of the sparse grid quadrature methods may be due to the approximation error of the sparse grid. Minimizing the error is yet one of the future tasks to tackle. Still, we can see that our Stochastic Collocation technique is able to cope
5.3. Results for RE=100

Figure 5.8.: Monte Carlo Uncertainty Quantification (left) and sparse grid Stochastic Collocation (right) applied on the computational fluid scenario described in 5.1 for the drag coefficient $c_D$ for different sample sizes 10, 50 and 100, different levels 1, 2, 3, 4 and $RE = 100$.

Figure 5.9.: Sparse grid Stochastic Collocation applied on the computational fluid scenario described in 5.1 for the drag coefficient $c_D$ for different levels 1, 2, 3, 4 and $RE = 100$. Zoomed into a section of figure 5.8.
5. Results

with unsteady flow as well, and that we can expect the flow to show the same behavior for input with underlying uncertainty as for deterministic input and not to turn chaotic.

5.3.2. Lift coefficient

In contrast to the drag coefficient, for the lift coefficient we can see that all Stochastic Collocation methods approach the deterministic solution quite well. We will again look at the result for a deterministic simulation run compared to the Monte Carlo simulation and our sparse grid Stochastic Collocation method. In figure 5.10, the results for the Monte Carlo methods (left) and sparse grid method (right) are illustrated. Opposed to figure 5.8 we see no outliers and all lines show the same profile as the deterministic run. If we enlarge a section of the plot, as in figure 5.11, we see that for a higher level, the Stochastic Collocation method approaches the deterministic solution better. Furthermore, for the same level, the Monte Carlo quadrature resembles the deterministic run more than the corresponding sparse grid quadrature. Again, this confirms the result of the drag coefficient. Nonetheless, both methods show a non-chaotic behavior for an unsteady flow.

5.4. Distribution of the Quantities of Interest

Drag coefficient \( c_D \)

We will now look at the distribution of the drag coefficient \( c_D \) in the last time step of the simulation. We therefore used 10000 interpolation points in the Monte Carlo quadrature of the sparse grid Stochastic Collocation method. Afterwards, we used a histogram, a common tool to visualize a distribution. Since we expected \( c_D \) to be normally distributed, we used Freedman-Diaconis’ rule \( 29 \) to define the bin width \( h \). The bin width is thus defined as follows:

\[
h = 2 \frac{IQR(x)}{n^{\frac{1}{3}}},
\]

(5.5)
5.4. Distribution of the Quantities of Interest

Figure 5.11.: Sparse grid Stochastic Collocation applied on the computational fluid scenario described in 5.1 for the lift coefficient $c_L$ for different levels 1, 2, 3, 4 and $RE = 100$. Zoomed into a section of figure 5.10.

where $x$ are the samples and $n$ is the number of samples and $IQR$ is the interquartile range. Furthermore, we calculated the number of bins as

$$k = \left\lceil \frac{\max(x) - \min(x)}{h} \right\rceil.$$  \hfill (5.6)

Figure 5.12 shows two histograms for the drag coefficient $c_D$ in the last time step of the simulation, calculated on different levels $l$ and different dimensions $d$. On the left, $l = 1$ and $d = 1$, whereas on the right $l = 3$ and $d = 3$. That means, that for level 3 we added a standard deviation not only to the inlet velocity but also to the density and the viscosity.
5. Results

As one can see, the histogram for \( l = 3 \) and \( d = 3 \) looks quite similar to a gaussian distribution, while for \( l = 1 \) and \( d = 1 \) there is a high spike in the middle. This is probably due to the fact that a one-dimensional grid of level one is very coarse. Additionally, for a higher number of random parameters, the distribution comes better to light, whereas for only one random parameter it is still more deterministic, visualized by the spike. Nevertheless, we can expect the drag coefficient to be gaussian distributed and to not show a chaotic behavior for small variations in the input data. In appendix C you can find a visualization of all histograms from level 1 to level 3 and dimension 1 to 3.

**Lift coefficient** \( c_L \)

Regarding the distribution of \( c_L \) with the help of a histogram, we can also expect a normal distribution, which gets clearer for a higher grid level \( l \) and a higher number of random input parameters \( d \). Since the results are similar to subsection 5.4, we refer to appendix D where we visualized the histograms for level 1 to 3 and dimension 1 to 3. Again, we can expect \( c_L \) to underlie a probabilistic distribution and to not behave chaotically with small random disturbances in the input.

### 5.5. Runtime comparison

To compare the runtime, we additionally implemented a full grid Stochastic Collocation method to visualize the curse of dimensionality. First, however, we will compare the runtime of our two sparse grid methods with the Monte Carlo method in figure 5.13 on the left. As can be seen, the Stochastic Collocation methods have a shorter runtime compared to Monte Carlo for small samples, that means a low grid level. For an increasing number of grid points, the runtime of Monte Carlo increases significantly faster than the Stochastic Collocation methods. In figure 5.13 on the right, we compare the runtime between the full grid Stochastic Collocation and sparse grid Stochastic Collocation methods with sparse grid and Monte Carlo quadrature for different grid levels.

![Runtime comparison between the different Uncertainty Quantification methods](image)

Figure 5.13.: Runtime comparison between the different Uncertainty Quantification methods, Monte Carlo, and sparse grid Stochastic Collocation with sparse grid and Monte Carlo quadrature based on the number of samples (left). Runtime comparison between the full grid Stochastic Collocation and sparse grid Stochastic Collocation methods with sparse grid and Monte Carlo quadrature for different grid levels (right).
of samples and grid points, the Monte Carlo method is faster. That is due to the quadrature overhead in the sparse grid Stochastic Collocation methods, which is not necessary in Monte Carlo.

The other figure on the right shows the curse of dimensionality quite impressively. For this comparison, we used a three-dimensional input variable, that means $\sigma_{u_n} > 0$, $\sigma_{\rho_0} > 0$ and $\sigma_{\eta_0} > 0$, in our implementations. Due to the fact that the number of grid points in the full grid increases exponentially with its number of dimensions and levels, we see an exponential growth in its runtime. The sparse grid methods, on the other hand, almost have a linear growth in their runtime for an increasing level, underlining the performance advantages compared to the full grid. The two sparse grid methods have almost the same runtime since they only differ in the type of quadrature being used.

We can also confirm these results if we look at the number of grid points for a dimension $d$ and level $l$ of the full and sparse grid. The numbers are shown in table 5.1. We can see that the number of samples in Monte Carlo is independent of the number of random parameters. The number of grid points for the Stochastic Collocation methods using grids increases with the number of dimensions and the grid level. With $d = 2$ we can already recognize a difference. The number of grid points on the full grid is more than tripling the number of grid points on the sparse grid on level 4. The difference gets even bigger for dimension 3, where on level 3 the number of grid points of the sparse grids is already about one sixth of the points on the full grid. On level 4, the advantage of the sparse grid gets even more obvious, where 297 points on the sparse grid stand in no relation to 4913 points on the full grid.

Here we can already see the effects of the curse of dimensionality for only three dimensions, which will be worse for even higher ones. Therefore, we can show that the sparse grid Stochastic Collocation method is a powerful approach to cope with a high number of random parameters, which are quite common in engineering.

<table>
<thead>
<tr>
<th></th>
<th>MonteCarlo</th>
<th>SC FullGrid</th>
<th>SC SparseGrid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NumberOfSamples</td>
<td>l=1</td>
<td>l=2</td>
</tr>
<tr>
<td>d=1</td>
<td>10 50 100</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>d=2</td>
<td>10 50 100</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>d=3</td>
<td>10 50 100</td>
<td>27</td>
<td>125</td>
</tr>
</tbody>
</table>

Table 5.1.: Number of sample points and grid points used for the Monte Carlo, full grid and sparse grid Stochastic Collocation method respectively. Thereby, $d$ describes the dimension, i.e. the number of random input parameters and $l$ is the level of the grid.
5. Results

5.6. L2Error

In this section, we regard the L2Error between the deterministic solution \( y_D \) and the Uncertainty Quantification solutions \( y_{UQ} \). We compute the Mean-Squared Error (MSE) in the following form

\[
MSE(y_D, y_{UQ}) = \frac{1}{n} \sum_{i=1}^{n} (y_{Di} - y_{UQi})^2
\]  

(5.7)

where in our case \( n \) is the number of time steps.

We can see in figure 5.14 that the L2Error for the Monte Carlo method decreases for a rising number of samples for \( c_D \) as well as for \( c_L \). For the sparse grid Stochastic Collocation method with sparse grid quadrature on the other hand, the error stays the same for an increasing level and even ascends for \( c_L \). For the Monte Carlo quadrature, we see a decrease on level 2 and 4 but a random climb on level 3. The results show that an increase on the grid level does not have that high of an impact on the result for the stationary flow with \( RE = 20 \).

![Figure 5.14: L2Error for the Uncertainty Quantification methods compared to the deterministic solution for \( RE = 20 \) for \( c_D \) (left) and \( c_L \) (right)](image)

For \( RE = 100 \), however, we received different results. As illustrated in figure 5.15 on the right, while the error for \( c_D \) does only decreases slightly for a higher grid level, we can see an effect of the sparse grid level on the L2Error of the lift coefficient \( c_L \). For both methods - the sparse grid Stochastic Collocation with sparse grid quadrature, as well as with Monte Carlo quadrature - the error decrease with higher grid level, which shows that for an unsteady flow the use of a higher level is effective to better approximate the solution.

Regarding the Monte Carlo Uncertainty Quantification method on the left, however, we see the smallest error for 10 samples for \( c_D \) as well as \( c_L \). Since Monte Carlo draws these samples randomly this is probably due to lucky draws close to the mean of \( u_{in} \). Nevertheless, we can see an error decrease from 50 to 100 samples which is more representative and shows the validity of the method. Still, in future work, we will compare our method to Monte Carlo runs with higher sample sizes. The runtime of Monte Carlo with 100 samples
for $RE = 100$ was albeit about five days and longer runs were not realizable for this work.

In this chapter, we presented the results of our sparse grid Stochastic Collocation methods. We showed that a simulation influenced by small uncertainties still shows similar behavior than a deterministic run. Furthermore, our results indicated, that the accuracy of the Stochastic Collocation methods for higher Reynolds numbers is dependent on the used grid level. We did not make this observation for a lower Reynolds number. Our methods showed the same results as the Monte Carlo Uncertainty Quantification, which displayed its validity. They are able to show the impact of uncertainty on the solution and to approximate the distribution of the quantity of interest. Furthermore, the sparse grid Stochastic Collocation method can cope with high dimensions, limiting the curse of dimensionality, thereby preserving the accuracy of Monte Carlo.
5. Results
6. Conclusion and Outlook

This thesis presented two Stochastic Collocation method using sparse grids with Monte Carlo and sparse grid quadrature respectively in a common computational fluid dynamics benchmark scenario. We showed the usage of the SG++ library in Matlab as grid generator for these method. Additionally, we adapted the computational fluid dynamics simulator quickfluid to use it for the simulation runs. The results were then presented for a cylinder channel scenario with up to three random input parameters and compared to a Monte Carlo Uncertainty Quantification method. We showed the methods' ability to reach the same and even better approximation results for less number of samples than Monte Carlo. Furthermore, their efficiency was also displayed by comparing the sparse grid Uncertainty Quantification to Monte Carlo and a full grid Stochastic Collocation scheme. We proved their ability to cope with the curse of dimensionality for a higher number of random parameters opposed to using full grid. Finally, their strength of needing less samples than Monte Carlo make them more performant than the standard sample approach as well.

There is still a lot of research going on in the field of Uncertainty Quantification and Stochastic Collocation using sparse grids in particular. Other works that should be mentioned concerning the topic of sparse grids in Uncertainty Quantification, are [7], [10] as well as [17]. While the first two use global basis for example with Lagrangian interpolating polynomials for the grid construction, [17] is the only international work known to us which uses hierarchical bases as we do. Their work even goes one step further, by adding adaptivity to the grid using the hierarchical surpluses as error indicator. The idea of our work, however, was to take a first step into Uncertainty Quantification, connecting UQ, sparse grids and computational fluid dynamics.

Nonetheless, adding sparse grid adaptivity could be one of multiple possible tasks for future research. It could come hand in hand with an error analysis which examines the interpolation error made by the sparse grid interpolation and the sparse grid quadrature. Due to the high runtime of each deterministic sample run, a parallelization of the methods should be considered as well. Since every run is independent by itself, parallelizing each sample should be quite efficient and straightforward to implement. Additionally, varying standard deviations, higher random dimensions and bigger sample sizes should be tested to further broaden the influence of uncertainty on the simulation. Furthermore, the method should be tested on various other scenarios and a Stochastic Galerkin method or Stochastic Collocation using Polynomial Chaos Expansions could be implemented as well to compare its results with our method. The field of Uncertainty Quantification is still a rather new area of research and there is yet a lot we do not know.
6. Conclusion and Outlook
Appendix
function [ result, noGridPoints ] = interfaceJava( dim, level, funct, boundaries )

% INTERFACEJAVA Summary of this function goes here
% Detailed explanation goes here

%import all packages
%import sgpp.*;

% or, better, include only the ones needed
import sgpp.DataVector;
import sgpp.GridGenerator;
import sgpp.GridStorage;
import sgpp.Grid;
import sgpp.GridIndex;
import sgpp.jsgpp;
import sgpp.OperationEval;

sgpp.LoadJSGPPLib.loadJSGPPLib();

% create a dim dimensional grid, either with or without boundarypoints
if (boundaries==0)
    grid = Grid.createLinearGrid(dim);
else
    grid = Grid.createLinearBoundaryGrid(dim);
end
gridStorage = grid.getStorage();
dimensionality = gridStorage.dim();

% create regular grid, level level
gridGen = grid.createGridGenerator();
gridGen.regular(level);
Gridlevel = level
noGridPoints = gridStorage.size()

alpha = DataVector(noGridPoints);
alpha.setAll(0);

% set function values in alpha
alpha_size = alpha.getSize();
alpha_array = nan(1, alpha_size);
alpha_array_hier = nan(1, alpha_size);
tmp_point = nan(1, dim);
for i = 0 : noGridPoints-1
    gp = gridStorage.get(i)
    for j = 0 : dim-1
        tmp_point(j+1) = gp.abs(j);
    end
    if (dim==1)
        res = funct(tmp_point(1));
    end
end
%Saving grid points in x_array
x_array = nan(1, noGridPoints*dim);
for i = 0 : noGridPoints−1
    gp = gridStorage.get(i);
    for j = 0 : dim−1
        x_array(1,1+i+noGridPoints*j) = gp.abs(j);
    end
end

%Hierarchize
operationHierarchisation = jsgpp.createOperationHierarchisation(grid);
operationHierarchisation.doHierarchisation(alpha);

for i = 0 : gridStorage.size()−1
    alpha_array.hier(1,1+i)=alpha.get(i);
end

%evaluate
p = DataVector(dim);
p.set(0,0.52);
p.set(1,0.73);
opEval = jsgpp.createOperationEval(grid);
disp("u(0.52,0.73) = " + opEval.eval(alpha,p));
result = nan(1,noGridPoints*(dim+2));
%x_array
result(1,1:noGridPoints*dim)=x_array;
%alpha_array
result(1,(noGridPoints*dim)+1:noGridPoints*(dim+1))=alpha_array;
%alpha_array.hier
result(1,(noGridPoints*(dim+1))+1:noGridPoints*(dim+2))=alpha_array.hier;
#include "mex.h"
#include "math.h"
// All SG++ headers
#include "sgpp_base.hpp"
// Or, include only those that are required
//# include "data/DataVector.hpp"
//# include "grid/Grid.hpp"
//# include "grid/GridStorage.hpp"
//# include "grid/generation/GridGenerator.hpp"
//# include "operation/common/OperationEval.hpp"

using namespace std;
using namespace sg;
using namespace base;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double a;
    if (nrhs <= 1 || nrhs > 4) mexErrMsgTxt("Not enough or too many arguments");
    if (!mxIsClass(prhs[2], "function_handle")) mexErrMsgTxt("Third input argument is not a function_handle!");

    int dim = (int)*mxGetPr(prhs[0]);
    mxArray *lhs = *rhs[dim+1];
    int level = (int)*mxGetPr(prhs[1]);
    rhs[0] = const_cast<mxArray *>(prhs[2]);
    int boundaries = (int)*mxGetPr(prhs[3]);

    // create a two-dimensional piecewise bi-linear grid
    Grid* grid;
    if (boundaries==0)
        grid = Grid::createLinearGrid(dim);
    else
        grid = Grid::createLinearBoundaryGrid(dim);

    GridStorage* gridStorage = grid->getStorage();
    cout << "dimensionality: " << gridStorage->dim() << endl;

    // create regular grid, level
    GridGenerator* gridGen = grid->createGridGenerator();
    gridGen->regular(level);
    double noGridPoints = (double)gridStorage->size();
    cout << "GridLevel: " << level << endl;
    cout << "number of grid points: " << noGridPoints << endl;

    // create coefficient vector
    DataVector alpha(gridStorage->size());
    alpha.setAll(0.0);
cout << "length of alpha-vector: " << alpha.getSize() << endl;

// set function values in alpha
double alpha_array[alpha.getSize()];
double alpha_array_hier[alpha.getSize()];
GridIndex* gp;
for (int i=0; i < gridStorage->size(); i++) {
  gp = gridStorage->get(i);
  for(int j = 0; j < dim; j++) {
    a = gp->abs(j);
    rhs[j+1]=mxCreateDoubleScalar(a);
  }
  mexCallMATLAB(1,&lhs,dim+1,rhs,"feval");
  alpha[i] = (*mxGetPr(lhs));
}

//Save grid points to x_array
int j = 0;
int k = 0;
double x_array[gridStorage->size()*dim];
for (int i = 0; i < gridStorage->size(); i++) {
  gp = gridStorage->get(i);
  for (int j = 0; j < dim; j++) {
    x_array[i+gridStorage->size()*j]=gp->abs(j);
  }
}

// hierarchize
for (int i = 0; i < gridStorage->size(); i++) {
  alpha_array[i] = (double) alpha[i];
}

op_factory::createOperationHierarchisation(*grid)->doHierarchisation(alpha);
for (int i = 0; i < gridStorage->size(); i++) {
  alpha_array_hier[i]=(double)alpha[i];
}

// evaluate
DataVector p(dim);
p[0] = 0.52;
p[1] = 0.73;

OperationEval* opEval = op_factory::createOperationEval(*grid);
cout << " u(0.52, 0.73) = " << opEval->eval(alpha, p) << endl;

plhs[0] = mxCreateDoubleMatrix(1, alpha.getSize()*(dim+2), mxREAL);
double* result = mxGetPr(plhs[0]);
plhs[1] = mxCreateDoubleMatrix(1,1,mxREAL);
double* noGridPoints = mxGetPr(plhs[1]);

for (int i = 0; i < gridStorage->size()*(dim+2); i++) {
  if (i < gridStorage->size() * dim) {
    result[i] = x_array[i];
  } else if (i < gridStorage->size() * (dim + 1)) {
    result[i] = alpha_array[i - gridStorage->size() * dim];
  } else {
    result[i] = alpha_array_hier[i - gridStorage->size() * (dim + 1)];
  }
Listing B.1: interface.cpp source code

```cpp
  }
  delete grid;
}
```
B. Source Code: interface.cpp
C. Histograms for drag coefficient $c_D$

Figure C.1.: Histogram to show distribution of drag coefficient $c_D$ on level 1 from dimension 1 to 3 from left to right.

Figure C.2.: Histogram to show distribution of drag coefficient $c_D$ on level 2 from dimension 1 to 3 from left to right.

Figure C.3.: Histogram to show distribution of drag coefficient $c_D$ on level 3 from dimension 1 to 3 from left to right.
C. Histograms for drag coefficient $c_D$
D. Histograms for lift coefficient $c_L$

Figure D.1.: Histogram to show distribution of lift coefficient $c_L$ on level 1 from dimension 1 to 3 from left to right.

Figure D.2.: Histogram to show distribution of lift coefficient $c_L$ on level 2 from dimension 1 to 3 from left to right.

Figure D.3.: Histogram to show distribution of lift coefficient $c_L$ on level 3 from dimension 1 to 3 from left to right.
Bibliography


