Aalto University
School of Science

Jonatan Lehtonen

Collocation method for solving stochastic elasticity problems with an uncertain domain

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology in the Degree Programme in Engineering Physics and Mathematics.

Espoo, February 10, 2015

Supervisor:	Associate Professor Nuutti Hyvönen
Instructor:	D.Sc. (Tech.) Harri Hakula

Aalto University
School of Science

Aalto University School of Science	ABSTRACT OF THE MASTER'S THESIS

Author: Jonatan Lehtonen

Title: Collocation method for solving stochastic elasticity problems with an uncertain domain

Degree Programme: Degree Programme in Engineering Physics and Mathematics

Major subject: Mathematics
Minor subject: Information and Computer Science

Chair (code): Mat-1

Supervisor: Associate Professor Nuutti Hyvönen
Instructor: D.Sc. (Tech.) Harri Hakula

Abstract: In this thesis, we formulate a method for determining how quantities such as stress in an elastic body change depending on its shape. This stochastic elasticity problem has important applications in structural analysis and design, such as determining how manufacturing flaws affect the properties of an object. We assume that the shape of the object depends on some stochastic parameters, and use a combination of multivariate interpolation and conformal mappings to solve the problem. The interpolation allows us to reduce the stochastic problem to a collection of deterministic elasticity problems, which are solved by using existing finite element analysis software, and the conformal mappings are used to accommodate the varying shape of the object. A sparse grid interpolation scheme is used to diminish the curse of dimensionality related to multivariate interpolation. We define model problems involving two stochastic parameters, for both 2D and 3D objects. The implementation of the method is described in detail, and numerical results are provided for the model problems. With as few as 29 deterministic problems, we reach the point where the interpolation accuracy cannot be improved due to the inherent inaccuracy of the finite element solutions.

Date: February 10, 2015	**Language:** English	**Number of pages:** v+53

Keywords: elasticity problem, stochastic domain, stochastic collocation, Smolyak construction, sparse grid interpolation, conformal mapping

Aalto-yliopisto
Perustieteiden
korkeakoulu

Aalto-yliopisto Perustieteiden korkeakoulu	DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Jonatan Lehtonen

Työn nimi: Kollokaatiomenetelmä stokastiselle elastisuustehtävälle epävarmassa alueessa
Työn nimi (engl.): Collocation method for solving stochastic elasticity problems with an uncertain domain

Tutkinto-ohjelma: Teknillisen fysiikan ja matematiikan tutkinto-ohjelma

Pääaine: Matematiikka
Sivuaine: Tietojenkäsittelytiede

Opetusyksikön (ent. professuuri) koodi: Mat-1

Työn valvoja: Professori Nuutti Hyvönen
Työn ohjaaja: TkT Harri Hakula

Tiivistelmä: Tässä diplomityössä esittelemme menetelmän, jolla voidaan määrittää kuinka jännityksen tai jonkin muun suureen arvo muuttuu elastisessa kappaleessa kun sen muoto vaihtelee. Tällä stokastisella elastisuustehtävällä on tärkeitä sovelluksia rakenteiden analyysissä ja suunnittelussa; sitä käyttäen voidaan esimerkiksi määrittää valmistusvirheiden vaikutus kappaleen ominaisuuksiin. Oletamme kappaleen muodon riippuvan joistakin stokastisista parametreista, ja käytämme usean muuttujan interpolointia sekä konformikuvauksia tehtävän ratkaisemiseksi. Interpoloinnin avulla stokastinen ongelma voidaan muuntaa kokoelmaksi deterministisiä tehtäviä, jotka ratkaistaan käyttäen elementtimenetelmää; konformikuvauksien avulla käsitellään stokastisuuden aiheuttama kappaleen muodon vaihtelu. Usean muuttujan interpolointiin liittyvän dimensionaalisuuden kirouksen lievittämiseksi käytämme Smolyakin konstruktioon perustuvaa harvan hilan interpolointimenetelmää. Mallitehtävinä käytämme kahdesta stokastisesta parametrista riippuvia kappaleita, tarkastellen sekä kaksi- että kolmiulotteista tapausta. Kuvailemme menetelmän toteutuksen yksityiskohtaisesti, ja esittelemme mallitehtävien numeeriset tulokset. Menetelmä saavuttaa jo 29 deterministisen tehtävän avulla pisteen, jonka jälkeen interpolaation tarkkuutta ei enää voida parantaa elementtimenetelmälle luontaisesta epätarkkuudesta johtuen.

Päivämäärä: 10.02.2015	**Kieli:** englanti	**Sivumäärä:** v+53

Avainsanat: elastisuustehtävä, stokastinen alue, stokastinen kollokaatio, Smolyakin konstruktio, harvan hilan interpolointi, konformikuvaus

Contents

Notation and abbreviations

Symbols

$\mathbb{R}$	set of real numbers
$\mathbb{N}$	set of natural numbers (non-negative integers)
$\mathbb{R}^2, \mathbb{R}^d$	set of plane vectors, set of d-dimensional vectors
$\mathbb{N}^d$	set of d-dimensional multi-indices (vectors of natural numbers)
$\boldsymbol{a}, \boldsymbol{x}$	vectors (lower-case bold letters)
$\boldsymbol{n}$	normal unit vector
$[a, b], I$	intervals of the set $\mathbb{R}$

Operators

$\dfrac{\partial}{\partial x_i}$	partial derivative with respect to x_i
$\nabla \boldsymbol{x}$	vector gradient
Δu	Laplace operator
$\boldsymbol{x} \cdot \boldsymbol{y}$	vector dot product
$A \subset B$	A is a subset of B
$x \in \mathbb{R}$	x is an element of $\mathbb{R}$
$\sum a_i$	sum
$\prod a_i$	product
$\bigotimes a_i$	tensor product (see Section 3.3)

Chapter 1

Introduction

Finite element analysis is widely used for solving problems of linear elasticity, such as studying the deformation of an elastic solid object when subjected to various loads, and computing the stresses caused by these loads. As these problems are crucial for structural analysis and design, the finite element method (FEM) has been the subject of extensive research. However, in classical FEM we need to explicitly know all properties of the object and the loads acting on it. Thus it is only suited for solving *deterministic* problems, where there is no random component involved. Naturally, this is seldom the case in real-world problems, where measurements cannot be exact and objects never have an ideal shape, due to manufacturing flaws and other such effects. These issues can often be ignored by assuming that their effect on the properties of the object is small, but if the variations in shape are large or if high accuracy is important, the shape of the object can no longer be considered deterministic and an alternate method is required.

This leads us to a *stochastic* elasticity problem. Suppose that we have an elastic body with a shape that is random to some extent. In order to make this computationally feasible, we assume that the randomness can be adequately described by some finite number of parameters; thus, these parameters can be thought of as random variables, and fixing their values yields a deterministic problem. Our goal is then to determine the distribution of, say, the stresses in the object as a function of these parameters. For simplicity, we focus on finding the stresses at a single measurement point, but our methods can easily be extended to several measurement points with only a small additional cost. Note that while we refer to the problem as stochastic, the same methodology could just as well be applied to, say, structural optimization. Strictly speaking, we only need the shape of the object to depend on some parameters; whether the parameters are random variables or not makes little difference from a practical point of view.

What we have described above is generally known as a problem of *uncertainty quantification*. Extensive research has been done in this field; good overviews of the related *spectral methods* used for solving such problems can be found, for example, in [11] and [7]. Unfortunately, most of the research in this field focuses on problems where the stochastic part lies in the governing equations rather than the geometry of the domain. To an extent, the same methods can still be applied when the geometry is stochastic, but some problems arise since the finite element mesh used by FEM needs to change with the geometry.

The most commonly used spectral methods are the Galerkin, collocation and Monte Carlo methods. Galerkin methods are known as *intrusive* methods, since they require purpose-built software; this leads to high development costs, which are further compounded by the changing mesh. By contrast, collocation and Monte Carlo are *non-intrusive* methods, which strive to construct the solution to the stochastic problem by solving a number of deterministic problems; that is, solving the problem for a collection of fixed values of the parameters which define the shape of the object. Non-intrusive methods are therefore much easier to implement, as we can use existing FEM software for the deterministic problems. Another key advantage with non-intrusive methods is that, as they are based on solving several deterministic problems, they can be trivially parallelized to an almost arbitrary number of computing cores; indeed, in our work we found that one of the greatest bottlenecks on our performance was not the availability of computing cores, but the number of software licenses available for Abaqus.[1]

Monte Carlo methods work by sampling the distribution at randomly selected points, which can be used to measure quantities such as the mean or variance of a distribution. In [8], Schenk and Schuëller applied such methods to a problem which is somewhat similar to ours. However, we want to compute the distribution itself, which means we need to use some kind of an interpolation scheme, and Monte Carlo methods do not naturally lend themselves to interpolation.

Instead, we turn our attention to collocation methods. The main difference compared to Monte Carlo methods is that the points where we sample the distribution are chosen deterministically. As the shape of the object depends on a number of parameters, and our goal is to get the stress (or some other quantity) at a given point as a function of these parameters, the problem essentially becomes one of interpolating a multivariate function. Intuitively, it could therefore make sense to use, say, a grid of equidistant nodes and interpolate between them. However, this is extremely inefficient due to the *curse of dimensionality*, as the number of nodes would increase exponentially with the number of parameters used to describe the shape of the object. Fortunately, this problem was solved by Smolyak in [9], where he proposed a sparse interpolation grid which significantly diminishes the curse of dimensionality while preserving the accuracy up to a logarithmic factor.

We emphasize that while the curse of dimensionality is diminished, it is certainly still present. For example, if we increase both the number of stochastic parameters and the order of interpolation by one, the number of FEM problems which need to be solved increases approximately by a factor of six. Naturally, the question of whether a given stochastic problem can be

[1]Abaqus is a software suite for finite element analysis; for additional information, see `http://www.3ds.com/products-services/simulia/products/abaqus`.

solved in a reasonable time depends on a multitude of factors, such as how much computation time a single FEM problem requires, how many can be solved in parallel, how high the order of interpolation has to be, and so on. It is therefore difficult to assess beforehand whether a given problem is feasible. We will discuss this subject further in Chapter 5.

In Chapter 2, we formalize the notion of the stochastic problem which we described earlier, and introduce the model problems which we will use to gauge the quality of our method. In Chapter 3, we provide an overview of all the steps required to implement our method and give a detailed explanation of the interpolation scheme which we mentioned earlier. We will also describe how we can use conformal mappings to overcome the problems caused by the random shape of the object. Details about the model problems and the implementation of our methods are provided in Chapter 4, with the intent of ensuring that anyone with some knowledge of FEM problems and programming has the necessary information to use our methods in practice. Finally, in Chapter 5, we give the results of our numerical experiments and briefly describe the factors that should be taken into account to ensure that the desired accuracy can be achieved in a reasonable amount of computation time, before wrapping things up in Chapter 6.

1.1 Prerequisites and definitions

This thesis has a broad target audience, and as such we make very few assumptions about what the reader already knows. The theoretical parts of this thesis should only require a basic understanding of multivariate calculus, which is part of all engineering-oriented university curriculums. Readers who wish to implement the methods described in this work should already be familiar with using some FEM solver (for this thesis, we used Abaqus); some experience with programming will also be helpful.

We will now introduce some of the notation and terminology used in this thesis. Note that the following definitions are very informal — they are intended to be understandable, not rigorous or general.

Firstly, we use $\mathbb{R}$ to denote the set of real numbers. If a variable x only takes values between two numbers a and b, i.e. it satisfies $a \leq x \leq b$, then we write $x \in [a, b]$. We call $[a, b]$ an *interval*, and denote it by I. Since I is a subset of $\mathbb{R}$, we write $I \subset \mathbb{R}$.

We call a function *univariate* if it only depends on one variable, and *multivariate* if it depends on several. In the univariate case, we may say that $f(x)$ is defined on an interval I, which means that the function is only defined for $x \in I$. In the multivariate case we will write the function as $f(\boldsymbol{x}) = f(x^1, \ldots, x^d)$, where $\boldsymbol{x} \in \mathbb{R}^d$ is a d-dimensional vector. Just like univariate functions may be restricted to intervals, a multivariate function $f(x^1, \ldots, x^d)$ may be restricted to a *hyperrectangle* $I_1 \times \cdots \times I_d$, which is equivalent to saying that each x^i is restricted to its corresponding interval I_i.

We will also need another version of functions, which we call *operators.* Simply put, the difference is that while functions map numbers to numbers, operators can also map e.g. functions to numbers or even functions to other functions. We will typically denote operators by U or V. For example, we could have an integral operator U defined as

$$Uf := \int_a^b f(x)dx.$$

We call U a linear operator if, for any two functions f and g and any two real numbers a and b, it satisfies

$$U(af + bg) = a(Uf) + b\,(Ug).$$

In particular, we will be interested in a special class of linear operators called univariate interpolation formulas, which we will describe in detail in Section 3.2.

When we define the multivariate interpolation scheme in Section 3.4, we will make use of *multi-index notation*. We denote by $\mathbb{N}$ the set of all non-negative integers, better known as the natural numbers. We call α a *multi-index* if $\alpha \in \mathbb{N}^d$, i.e. if $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ is a vector of natural numbers. The 1-*norm* of α is defined as

$$|\alpha| = \sum_{i=1}^{d} \alpha_i.$$

Chapter 2

Model problem

In this chapter, we describe the model problems which will be used as examples throughout this thesis. Sections 2.1 and 2.2 introduce a simple two-dimensional problem that we used for testing our method. In Section 2.3 we briefly discuss the general three-dimensional version of the problem, and in Section 2.4 we describe a model problem which we will use to demonstrate how the two-dimensional solution extends to three dimensions. The model problems will be described in very general terms in this chapter, with specific details provided later on in Chapter 4.

2.1 2D deterministic problem

Let us begin by considering the two-dimensional problem setting which served as a test case in our work. Let Ω be a domain representing a deformable medium subject to a surface traction $\boldsymbol{g}$. The 2D model problem is then to find the displacement field $\boldsymbol{u} = (u_1, u_2)$ and the symmetric stress tensor $\boldsymbol{\sigma} = (\sigma_{ij})_{i,j=1}^2$ satisfying

$$\begin{aligned}
\boldsymbol{\sigma}(\boldsymbol{u}) &= \lambda \operatorname{div}(\boldsymbol{u})\boldsymbol{I} + 2\mu\boldsymbol{\varepsilon} && \text{in } \Omega, && (2.1)\\
\operatorname{div}(\boldsymbol{\sigma}) &= \boldsymbol{0} && \text{in } \Omega, && (2.2)\\
\boldsymbol{u} &= \boldsymbol{0} && \text{on } \partial\Omega_D, && (2.3)\\
\boldsymbol{\sigma}\cdot\boldsymbol{n} &= \boldsymbol{g} && \text{on } \partial\Omega_N. && (2.4)
\end{aligned}$$

These equations describe the deformation of a two-dimensional elastic body under plane strain, although we have omitted the term for body forces. Plane strain is a state which typically occurs in cases where the length of an object is significantly greater than its width or height; thus, these equations can, for example, be used to find the stresses in the cross-section of a beam.

There are several definitions needed for the equations (2.1)–(2.4) above, so let us work through them from the top down. Firstly, the operator $\operatorname{div}(\boldsymbol{\sigma})$ is the vector-valued tensor divergence, defined as

$$\operatorname{div}(\boldsymbol{\sigma}) := \left(\sum_{j=1}^{2} \frac{\partial \sigma_{ij}}{\partial x_j} \right)_{i=1}^{2}. \qquad (2.5)$$

Note that we use the notation $(\cdot)_{i=1}^2$ to refer to the elements of a two-dimensional vector; this should not be confused with $(\cdot)^2$, which denotes the square of an expression.

The coefficients λ and μ appearing in (2.1) are the Lamé coefficients, defined through

$$\lambda := \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad \mu := \frac{E}{2(1+\nu)}, \tag{2.6}$$

where E is Young's modulus and ν is Poisson's ratio. In our work we used the values $E = 1000$ and $\nu = 0.3$. The value for Poisson's ratio was chosen to correspond to an object made of steel. As Young's modulus only has a linear effect on the solution, its value makes no difference in a model problem; therefore, we simply chose a value which yielded reasonable displacements in an object with unit width and height, subject to a unit force.

In (2.1) we also have the identity tensor $\boldsymbol{I}$ and the symmetric strain tensor

$$\boldsymbol{\varepsilon}(\boldsymbol{u}) = \frac{1}{2}\left(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^T\right). \tag{2.7}$$

Finally, $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ is the boundary of Ω, and $\boldsymbol{n}$ is its outward normal unit vector.

What we have described in this section is a relatively simple engineering problem, which can be solved for example using the finite element method (FEM). In our work, we used the new `NDSolve‘FEM`-package introduced in Mathematica 10, but any method of solving this problem is acceptable. We will assume that anyone implementing our method is familiar with solving deterministic elasticity problems, and thus we will not go into further details about how to solve them. Specifics such as the shape of the domain Ω will be given in Chapter 4, since they are not relevant to the theoretical aspects of our work.

2.2 2D stochastic problem

The deterministic model described in Section 2.1 is sufficient as long as the domain Ω is fixed. However, it may be the case that we do not know the exact shape of the domain; perhaps due to inaccuracies in manufacturing, or any number of other reasons. Whatever the reason may be, it is clear that simply solving the deterministic model may not be sufficient. This is precisely the type of problem that our method is designed to solve.

Suppose that instead of a fixed domain Ω, we have a stochastic domain $\Omega(s_1, s_2, \ldots, s_d)$ defined as a function of d parameters, with the parameters chosen in such a way that they describe the uncertainty of the model; for instance, if the radius of a circular object is not known exactly, it could be chosen as one of the parameters. Since we have d parameters, we adopt the shorthand notation $\Omega(\boldsymbol{s}) := \Omega(s_1, s_2, \ldots, s_d)$ for the domain, where $\boldsymbol{s} = (s_1, \ldots, s_d)$ is a vector in $\mathbb{R}^d$. Furthermore, we assume that each parameter s_i is constrained to a corresponding interval $I_i^s = [s_{i,\min}, s_{i,\max}]$; using the terminology of Section 1.1, this means that the vector $\boldsymbol{s}$ is constrained to the *hyperrectangle* $I_1^s \times \cdots \times I_d^s$.

Now, we can model our uncertainty about the domain by treating the parameters s_i as random variables. This turns the deterministic problem of Section 2.1 into a stochastic one, and gives rise to a stochastic space of domains $\Omega(\boldsymbol{s})$. Ideally we would like to solve the deterministic problem for every possible domain in this space.

In practice, of course, solving an infinite number of problems is somewhat difficult. We will tackle this issue by using a collocation method to discretize the stochastic part of the problem, turning a stochastic problem into a collection of deterministic ones. These will then be solved separately, and the results will be interpolated to obtain a solution to the stochastic problem. This procedure is explained in greater detail in Section 3.1.

2.3 General 3D problem

The method described in this thesis can be used for solving a very broad class of problems. In principle, the only conditions are that we have a well-defined stochastic space of domains such as in Section 2.2 and that we know how to solve the given problem for any element of this stochastic space. Furthermore, the governing equations should be deterministic; the stochastic part of the problem should be restricted to the geometry of the domain. Of course, there are ways of generalizing these methods to handle stochastic equations, but that is beyond the scope of this work.

As a disclaimer, we note that the difficulty of implementing this method may depend greatly on both the shape of the domain and the variation in that shape caused by the stochastic part. The reason for this is that, given two domains and a fixed point in one of them, we need a consistent way of finding the corresponding point in the other domain. In this work, we will describe a way of doing this for 3D problems assuming that, in some sense, their geometry varies in just two dimensions. This is not as restrictive as it may sound, since e.g. axisymmetric objects satisfy this condition. A prime example of this is the 3D model problem we describe in Section 2.4.

Of course, more complex geometries can be handled as well if a suitable mapping can be found, but the choice of such mappings is left to the reader as it is beyond the scope of this work. However, as long as this part can be managed, the rest of the method should be straightforward to implement regardless of the exact nature of the problem; aside from the choice of mapping, the method we describe is quite problem-independent.

In this thesis, we will constrain ourselves to problems of solid mechanics, such as the ones solved by the standard Abaqus framework. In Section 2.4, we describe one such problem, which we use to demonstrate how our solution of the 2D problem extends to three dimensions.

2.4 3D model problem

The model problem that we chose is based on the 2D model problem described earlier in Sections 2.1 and 2.2. We simply revolved the domain of the 2D problem 180 degrees around an axis to obtain a 3D object, which was modelled as a general elastic body. All the details about this problem will be given in Section 4.2.

Chapter 3

Methods

In this chapter, we introduce the various methods that will be necessary to solve the stochastic elasticity problems described in Chapter 2. Since the solution algorithm consists of many parts, we begin by briefly describing the entire algorithm in Section 3.1, in order to provide the reader with an overview of how all the pieces fit together. The following sections will then describe the theory behind each part in greater detail.

As we mentioned in Chapter 1, we are interested in solving a *stochastic* elasticity problem. Essentially this means that we have an elastic object for which we want to compute quantities such as stresses or displacements, but we do not know the exact shape of the object; rather, we know the probability distribution for its shape. Given a measurement point $\tilde{\boldsymbol{x}}$ in some reference object, our goal is to obtain the distribution of, say, the stresses at $\tilde{\boldsymbol{x}}$ as the shape of the object changes. Solving this problem involves using interpolation to deal with the the stochastic nature of the problem, reducing it to a series of deterministic problems which we know how to solve. Furthermore, since each deterministic problem involves an object of a slightly different shape, we need to map these solutions to the reference object to be able to compare and combine them.

In Sections 3.2 and 3.3 we introduce univariate interpolation and the tensor product, respectively. Section 3.4 uses these concepts to describe the multivariate interpolation scheme which is at the heart of our method. Finally, Section 3.5 describes how we used conformal mappings to deal with the changes in geometry caused by the stochastic part of the problem.

Since this chapter focuses on the theoretical aspects of the methods, it will also be the most mathematically demanding chapter in this thesis; having said that, we still only assume a basic understanding of multivariate calculus, which should suffice for understanding the concepts that follow.

Implementation details will be given in Chapter 4.

3.1 Outline of the algorithm

In this section, we provide a brief overview of the necessary steps for implementing the methods described in this thesis. We begin by making some assumptions and definitions, and at the end of this section we give an explicit list of the general steps involved in making our method work. We will then give detailed explanations of the more complex parts of the algorithm in Sections 3.2–3.5.

As described in Section 2.2, we assume that we have a stochastic domain $\Omega(\boldsymbol{s}) = \Omega(s_1, \dots, s_d)$ determined by d random parameters. If we fix all the parameters, we obtain a deterministic problem which we know how to solve; suppose that the solution is some distribution $\sigma(\boldsymbol{x})$, which represents the distribution of some quantity in the domain, such as stress in the case of our model problems. In general, it makes no difference what exactly is being measured, as long as the result is a distribution in the domain; that is, as long as the solution can be evaluated at any point $\boldsymbol{x}$ in the domain. Whatever this distribution may be, we denote it by $\sigma(\boldsymbol{x}; \boldsymbol{s})$ for consistency. Note that $\boldsymbol{s}$ is included in the notation since the domain and consequently the solution depends on $\boldsymbol{s}$.

The next step is to fix what we will call the *nominal domain*, denoted by $\widetilde{\Omega}$. In some sense, the domain $\widetilde{\Omega}$ defines our reference object; as such, in applications where the stochastic part of the problem represents some manufacturing errors, it may make sense to define the nominal domain $\widetilde{\Omega}$ to be the "ideal object" which has no such flaws. However, all we need to assume is that such a domain exists and is an element of the stochastic domain; in other words, there exists some vector $\tilde{\boldsymbol{s}}$ of parameter values $\tilde{s}_i$ such that $\widetilde{\Omega} = \Omega(\tilde{\boldsymbol{s}})$.

The reason why we need a nominal domain is quite simple; we need to be able to combine the solutions $\sigma(\boldsymbol{x}; \boldsymbol{s})$ from several domains $\Omega(\boldsymbol{s})$ into a single one, and we need to do this consistently. Imagine as an extreme example that we had a stochastic domain where some choice of the parameters s_i results in a cube domain, and another results in a ball. We can probably solve the deterministic problem in each of these domains, but how does one combine the solution in a cube with the one in a ball? There are points $\boldsymbol{x}$ which are contained in the cube but not in the ball, or vice versa; naturally, the same problem always arises with different geometries, even if their difference is not as extreme. We solve this problem by mapping all the solutions to the nominal domain $\widetilde{\Omega}$.

This leads us to our next assumption: for each domain $\Omega(\boldsymbol{s})$, assume that we have a mapping $\phi(\boldsymbol{x}; \boldsymbol{s})$ which maps each point of the nominal domain $\widetilde{\Omega}$ to the respective point in the domain $\Omega(\boldsymbol{s})$; furthermore, the mapping should be *bijective* and *consistent*. Bijectivity guarantees that for every point $\boldsymbol{x}'$ in the target domain $\Omega(\boldsymbol{s})$, there is *exactly* one point $\boldsymbol{x}$ in the nominal domain such that $\phi(\boldsymbol{x}; \boldsymbol{s}) = \boldsymbol{x}'$; this is why a bijection is sometimes called a 1–1 correspondence. A pleasant consequence of bijectivity is also that the mapping $\phi(\boldsymbol{x}; \boldsymbol{s})$ can be inverted to obtain a similar mapping from $\Omega(\boldsymbol{s})$ to the nominal domain $\widetilde{\Omega}$.

The condition of consistency is more vague, since the notion of consistency depends greatly on the problem itself, but the basic principle is that a small change in the domain $\Omega(\boldsymbol{s})$ should cause only small changes in the mapping

$\phi(\boldsymbol{x}; \boldsymbol{s})$. In Section 3.5 we will explicitly define the mapping we used for the model problems in this thesis.

Next, we need to choose the domains $\Omega(\boldsymbol{s})$ in which we will solve the deterministic problem. These domains are chosen according to the interpolation formula that has been chosen; in our work, we used multivariate Lagrangian polynomial interpolation, but in principle any multivariate interpolation formula can be used. Whatever the choice may be, it should give a set of domains $\Omega(\boldsymbol{s})$ for which we solve the deterministic problem.

Observe that in Sections 3.2–3.4 we interpolate an arbitrary d-dimensional function $f(\boldsymbol{x}) = f(x_1, \ldots, x_d)$ for the sake of generality. In our work, the measurement point $\tilde{\boldsymbol{x}}$ is fixed; it is the point where we want to obtain the distribution of $\sigma(\boldsymbol{x}; \boldsymbol{s})$ over all $\boldsymbol{s}$ in the stochastic space. Thus we actually interpolate over the parameters $\boldsymbol{s}$, and the function $f(\boldsymbol{x})$ is replaced by $f(\boldsymbol{s}) = \sigma(\phi(\tilde{\boldsymbol{x}}; \boldsymbol{s}); \boldsymbol{s})$. This function maps the fixed point $\tilde{\boldsymbol{x}}$ to the domain $\Omega(\boldsymbol{s})$ using the mapping $\phi(\boldsymbol{x}; \boldsymbol{s})$, and then evaluates the solution $\sigma(\boldsymbol{x}; \boldsymbol{s})$ of the deterministic problem at the point $\phi(\tilde{\boldsymbol{x}}; \boldsymbol{s})$. Since the mapping $\phi(\boldsymbol{x}; \boldsymbol{s})$ is chosen in such a way that $\phi(\boldsymbol{x}; \boldsymbol{s})$ is relatively close to $\boldsymbol{x}$, the function $\sigma(\phi(\tilde{\boldsymbol{x}}; \boldsymbol{s}); \boldsymbol{s})$ essentially gives us the distribution of σ at $\tilde{\boldsymbol{x}}$ as a function of $\boldsymbol{s}$, which is exactly what we wanted to compute.

To sum up, the general scheme is as follows:

1. Pick a problem that can be solved in any fixed domain Ω.
2. Define the stochastic space $\Omega(\boldsymbol{s})$, the nominal domain $\widetilde{\Omega}$ and the measurement point $\tilde{\boldsymbol{x}} \in \widetilde{\Omega}$.
3. Pick a multivariate interpolation formula; we used multivariate Lagrangian polynomial interpolation on a sparse grid.
4. Pick a mapping $\phi(\boldsymbol{x}; \boldsymbol{s})$ which maps the point $\boldsymbol{x}$ from $\widetilde{\Omega}$ to $\Omega(\boldsymbol{s})$ in a consistent manner.
5. Solve the deterministic problem in the domains $\Omega(\boldsymbol{s})$ to obtain the solutions $\omega(\boldsymbol{x}; \boldsymbol{s})$, where the parameter vectors $\boldsymbol{s}$ are dictated by the interpolation formula.
6. Construct the interpolating function $f(\boldsymbol{s}) = \sigma(\phi(\tilde{\boldsymbol{x}}; \boldsymbol{s}); \boldsymbol{s})$ from these solutions, using the mapping ϕ.

In the following sections, we will describe in detail the interpolation formula and mapping which we used for our method.

3.2 Univariate interpolation

Before we can introduce the multivariate interpolation scheme in Section 3.4, we need to discuss two simpler concepts: univariate interpolation and tensor products. Readers who are already familiar with these concepts may skip ahead to Section 3.4, but skimming through this and the following section is still recommended to become familiar with our notation and definitions.

Both this section and Section 3.4 are mainly based on Chapter 3 of [6], which gives a wonderful treatise on multivariate interpolation; we recommend it for anyone seeking more details on this subject.

Throughout Sections 3.2–3.4 we will consider the interpolation of some arbitrary function f, to ease the notation and keep things on a slightly more general level. We assume that evaluating f is relatively expensive; certainly this is the case in our work, where a single evaluation of f corresponds to solving an entire FEM problem.

The basic idea behind interpolation is that we know the value of the function f at specific points, and we want to construct another function that approximates f as accurately as possible *between* these points. In our work, we are dealing with the type of interpolation where we are able to choose the measurement points freely. The goal then becomes to pick the points in such a way that we can get a good approximation of f with as few points as possible. This is typically motivated by the cost of evaluating f, which we mentioned above.

As the name itself implies, interpolation is done between measurement points, so from now on we assume that f is defined in an interval I; for the multivariate case which we consider in later sections, the interval is replaced by a hyperrectangle $I_1 \times \cdots \times I_d$.

3.2.1 Piecewise linear interpolation

Let us first consider one of the simplest interpolation techniques available: piecewise linear interpolation. It is not ideal for accuracy, meaning that it will require more measurements to reach the same level of accuracy as other methods, but we mention it because it is very simple to implement. Once we move on to the multivariate case, we will not consider piecewise linear interpolation, but its extension to multivariate interpolation is trivial.

As the name implies, in piecewise linear interpolation the function is approximated by a line segment between measurement points. Thus, if we know the function values $f(x_i)$ at the points $x_0 = a < x_1 < \ldots < x_n = b$, then the interpolation formula U_1^h is given by

$$U_1^h(f)(x) := f(x_j) + \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}(x - x_j) \quad \text{for } x \in [x_j, x_{j+1}]. \tag{3.1}$$

Note that since the points x_i and function values $f(x_i)$ are constant, the only variable above is x and the interpolating function $U_1^h(f)$ is indeed linear. It is also easy to check that $f(x) = f(x_j)$ whenever $x = x_j$.

In (3.1), we used the notation U_1^h for the interpolation formula. Here, 1 refers to the degree of the estimate; similar notation will be used later on when we discuss a higher-order method. The superscript h indicates the largest length of an interval $[x_j, x_{j+1}]$ among the measurement points x_i; this is useful for estimating the interpolation error. Indeed, if we assume that f

is twice continuously differentiable in each interval $[x_j, x_{j+1}]$, which we write as $f \in C^2([x_j, x_{j+1}])$, then we have the convenient estimate

$$\left\| f - U_1^h(f) \right\|_\infty \leq \frac{h^2}{8} \max_{\xi \in [a,b]} |f''(\xi)|. \tag{3.2}$$

The norm on the left-hand side of (3.2) is the maximum norm, and is given by

$$\|f\|_\infty = \max_{\xi \in [a,b]} |f(\xi)|.$$

Looking at the right-hand side, we see that (3.2) gives us an estimate for the maximum norm of our error $f - U_1^h(f)$ which depends only on the maximum interval length h and the magnitude of the second derivative of f. Since f is a fixed function, the latter term is effectively constant, and we find that as we reduce the interval length h, the error decreases quadratically.

It goes without saying that in real-world applications we do not know the magnitude of f''; indeed, it might not even exist everywhere. However, the error estimate of (3.2) still gives us a good idea of what kind of accuracy we might expect to see as h decreases; it also suggests that the ideal distribution for the measurement points is to pick them equidistantly, i.e. such that $x_j - x_{j-1} = h$ for all $j = 1, \dots, n$.

Before we move on to higher-order methods, let us rewrite (3.1) in so-called Lagrangian form, since we will use the same form later on as well. To this end, define the basis functions a_j as

$$a_j(x) = \begin{cases} \frac{x - x_{j-1}}{x_j - x_{j-1}}, & \text{if } x \in [x_{j-1}, x_j], \quad j = 1, \dots, n, \\ \frac{x_{j+1} - x}{x_{j+1} - x_j}, & \text{if } x \in [x_j, x_{j+1}], \quad j = 0, \dots, n-1, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

These are so-called hat functions. Looking at the definition above, it is easy to see that the function a_j is linear on both $[x_{j-1}, x_j]$ and $[x_j, x_{j+1}]$, and we have $a_j(x_i) = 0$ whenever $i \neq j$, but $a_j(x_j) = 1$. This is the type of behaviour that we will see with all of our basis functions. Formally, this property can be written as $a_j(x_i) = \delta_{ij}$, where δ is the Kronecker delta, defined as follows:

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

Now, with some manipulation of formulas, it is easy to see that when $x \in [x_j, x_{j+1}]$, we have

$$\begin{aligned} f(x_j)a_j(x) + f(x_{j+1})a_{j+1}(x) &= f(x_j)\frac{x_{j+1} - x}{x_{j+1} - x_j} + f(x_{j+1})\frac{x - x_j}{x_{j+1} - x_j} \\ &= f(x_j) + \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}(x - x_j), \end{aligned}$$

which is exactly what we had in (3.1). This leads us to the Lagrangian form we mentioned earlier:

$$U_1^h(f)(x) = \sum_{j=0}^{n} f(x_j)a_j(x). \tag{3.4}$$

Finally, note that using our previous observation that $a_j(x_i) = \delta_{ij}$, it is immediately obvious that the form given by (3.4) is indeed exact at each measurement point:

$$U_1^h(f)(x_i) = \sum_{j=0}^{n} f(x_j)a_j(x_i) = \sum_{j=0}^{n} f(x_j)\delta_{ij} = f_(x_i).$$

3.2.2 Lagrangian polynomial interpolation

Now that we are familiar with interpolation and the Lagrangian form, we are ready to introduce the higher-order method. In the case of piecewise linear interpolation, we had $n+1$ measurement points but our estimate was only of degree 1. However, with $n+1$ distinct point values, it is always possible to find a polynomial which passes through each of these points. This polynomial is unique, and is known as the Lagrange polynomial. It is of degree at most n, and thus we denote the resulting interpolation formula by U_n.

As in the previous section, the interpolation formula U_n can be written in Lagrangian form. To this end, define the basis polynomials

$$l_j(x) = \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{x - x_k}{x_j - x_k}, \quad j = 0, \dots, n. \tag{3.5}$$

As before, we have assumed that $x_0 = a < x_1 < \dots < x_n = b$, which immediately implies that $x_j - x_k > 0$ for any $k \neq j$. Since the product skips the case $k = j$, this means that $l_j(x)$ is well-defined. Note also that the basis polynomials l_j exhibit the same property as the a_j's of the previous section: for any x_i, we have $l_j(x_i) = \delta_{ij}$. It also follows that l_j is always a polynomial of degree n, since it is non-zero and has n roots.

Given the basis functions l_j, we can again define U_n in Lagrangian form through

$$U_n(f)(x) = \sum_{j=0}^{n} f(x_j)l_j(x). \tag{3.6}$$

Our last result from the previous section holds here as well: for any measurement point x_i, we have

$$U_n(f)(x) = \sum_{j=0}^{n} f(x_j)l_j(x_i) = \sum_{j=0}^{n} f(x_j)\delta_{ij} = f_(x_i).$$

This proves that U_n is indeed a polynomial which passes through all of our $n+1$ measurement points.

In the previous section we mentioned that for piecewise linear interpolation, the ideal distribution of the measurement points is to pick them equidistantly; that is, choosing them so that $x_j - x_{j-1} = h$ for all $j = 1, \dots, n$. However, the same does not hold for Lagrangian polynomial interpolation. In fact, choosing the points equidistantly gives rise to the error estimate

$$\|f - U_n(f)\|_\infty \leq \frac{h^{n+1}}{4(n+1)} \max_{\xi \in [a,b]} |f^{(n+1)}(\xi)|, \tag{3.7}$$

where $f^{(n+1)}$ denotes the derivative of order $n+1$. At first glance, this may seem like a very good error bound, since h^{n+1} decreases very rapidly with h. However, note that the derivative term depends on n. Therefore, as n tends to infinity, the derivative term might explode, so we have no guarantee that the interpolating function $U_n(f)$ converges to f as n tends to infinity. This problem is known as Runge's phenomenon.

Thankfully, it is possible to choose a distribution of measurement points that does not exhibit the aforementioned problem; in fact, several such alternatives exist. We focus now on the Chebyshev–Gauss–Lobatto (CGL) points, which guarantee convergence when f is continuously differentiable. In fact, the weaker condition of Lipschitz continuity is sufficient; this is discussed in greater detail in Section 3.1 of [6].

The CGL points for a fixed $n > 0$ are given by the set

$$X_n := \left\{ x_j = \frac{a+b}{2} + \frac{a-b}{2} \cos(j\pi/n), \quad j = 0, \dots, n \right\}. \tag{3.8}$$

Observe that in the special case of $[a,b] = [-1,1]$ this reduces to $x_j = -\cos(j\pi/n)$, which should give a good idea of how the points are distributed; the cosine moves points towards the closest boundary, causing the distribution to have more points near a and b than in the middle.

Figure 3.1 demonstrates Runge's phenomenon, which we mentioned earlier. The figure shows the basis function corresponding to the midpoint of the interpolation interval $[-1,1]$ for $n = 8$ and $n = 32$, using equidistant and CGL nodes. The interpolation nodes lie at the zeros of the basis functions. As we can see, the basis function corresponding to the CGL nodes obtains its maximum at the midpoint of the interval, whereas the basis function for equidistant nodes blows up near the edges of the interval; note in particular the vertical scale in the top right figure. This is precisely why we use the CGL nodes for polynomial interpolation.

Finally, before moving on to tensor products, we give an error estimate for the Lagrangian polynomial interpolation formula U_n with Chebyshev–Gauss–Lobatto points: for any function f which is l times continuously differentiable in the interval $[-1,1]$ (in other words, $f \in C^l([-1,1])$), the interpolation error satisfies the estimate

$$\|f - U_n(f)\|_\infty \leq C_2 \|f^{(l)}\|_\infty n^{-l} \log(n), \tag{3.9}$$

where $f^{(l)}$ is the lth derivative of f and C_2 is a constant independent of l and n.

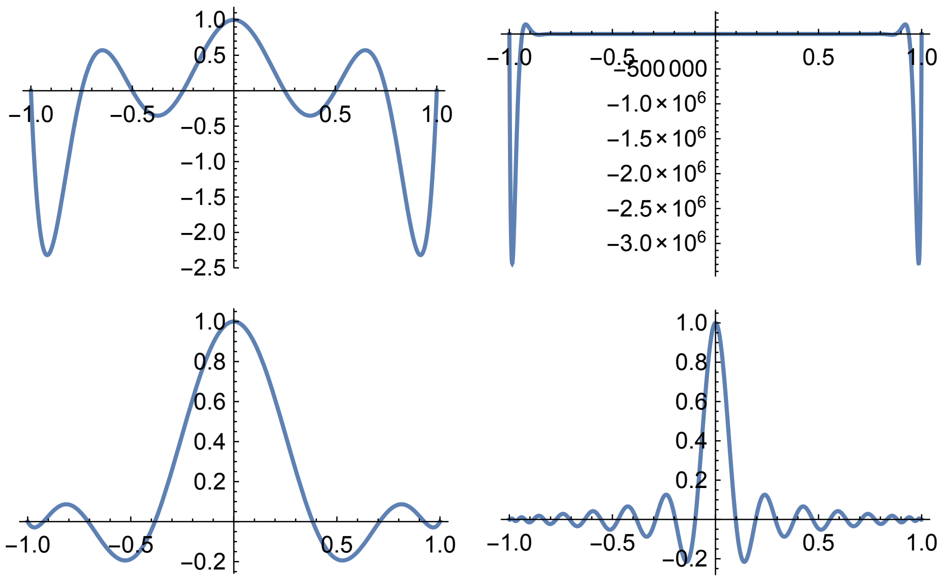

Figure 3.1: Demonstration of the pathological behavior of the polynomial basis functions on equidistant nodes. The top row uses equidistant nodes, the bottom row CGL nodes, with $n = 8$ on the left and $n = 32$ on the right.

3.3 Tensor product

Now that we can interpolate univariate functions, the next step is to extend univariate interpolation to multivariate functions. First, we need to make the distinction between *interpolating functions* and *interpolation formulas*. The difference between them is that interpolation formulas are applied to a function f to produce interpolating functions; in other words, U_n is an interpolation formula and $U_n(f)$ is the interpolating function that it produces. Observe that by the definitions we gave in Section 1.1, this means that U_n is, in fact, an *operator*. Definitions of this term vary, but in this work we use the word to describe an object that maps functions to functions; by contrast, we think of functions as mapping numbers to numbers.

Given these definitions, let us start thinking about interpolating multivariate functions, starting with a simple two-dimensional example. Suppose we have a function $g(x, y)$ that we wish to interpolate, but we only have a univariate interpolation formula U_n, given as a sum of terms of the form $l_j(x)f(x_j)$ where the points x_j are given by the set X_n defined in (3.8). This is the notation we used in the previous section for the Lagrangian interpolation formula; we will use it for the remainder of the theoretical sections since it is the formula we will use when implementing these methods.

Recall that the most important property for the basis functions $l_j(x)$ was that

$$l_j(x_i) = \delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{otherwise,} \end{cases}$$

for any point $x_i \in X_n$, where X_n is the set of interpolation points for the formula U_n. Our goal is now to construct two-dimensional basis functions with a similar property; fortunately, this is almost trivial, since the product of two basis functions gives exactly the kind of behaviour that we want. Indeed, if we define

$$l_{jk}(x,y) = l_j(x)l_k(y),$$

then for points $x_k, y_l \in X_n$ we have

$$l_{jk}(x_i,y_l) = l_j(x_i)l_k(y_l) = \delta_{ij}\delta_{kl} = \begin{cases} 1, & \text{if } i=j \text{ and } k=l, \\ 0, & \text{otherwise.} \end{cases}$$

Now, in the same way as we get a univariate interpolating function by summing the terms $l_j(x)f(x_j)$ over all $x_j \in X_n$, we get a two-dimensional interpolating function by summing $l_{jk}(x,y)f(x_j,y_k)$ over all *pairs* (x_j,y_k) where $x_j, y_k \in X_n$; that is, over $(n+1)^2$ different pairs (x_j,y_k). Writing this out as a sum yields

$$(U_n \otimes U_n)(g) = \sum_{k=0}^{n}\sum_{j=0}^{n} l_j(x)l_k(y)g(x_j,y_k). \tag{3.10}$$

The above is what we call the *tensor product* of two interpolation formulas, which is why we used the notation $U_n \otimes U_n$.

The next step is to extend the tensor product of (3.10) above to d dimensions. At the same time, we want to generalize it, so we now allow the individual interpolation formulas to be different. Since we want to give the tensor product explicitly, the notation unfortunately becomes a bit messy, but later on we will be able to avoid this.

Assume that we have a d-variate function $f(x^1,\dots,x^d)$ which we wish to interpolate in the hyperrectangle $I_1 \times \cdots I_d$; in other words, each variable x^i is constrained to a corresponding interval $I_i = [a_i,b_i]$. Furthermore, assume we have interpolation formulas U_{n_i} for each variable x^i, with corresponding sets X_{n_i} of interpolation points. Then the full d-dimensional tensor product formula is given by

$$(U_{n_1} \otimes \cdots \otimes U_{n_d})(f) = \sum_{j_1=0}^{n_1} \cdots \sum_{j_d=0}^{n_d} f(x^1_{j_1},\dots,x^d_{j_d}) \prod_{i=1}^{d} l^i_{j_i}(x^i), \tag{3.11}$$

where the last term denotes the product of basis functions $l^1_{j_1}(x_1),\dots,l^d_{j_d}(x_d)$.

At first glance it might seem that (3.11) is all we need for multivariate interpolation, but recall that by our assumption f is expensive to evaluate. Since the construction of the interpolating function $(U_{n_1} \otimes \cdots \otimes U_{n_d})(f)$ involves d nested sums of function evaluations $f(x^1_{j_1},\dots,x^d_{j_d})$, we need a total of $\prod_{i=1}^d(n_i+1)$ evaluations of f. Assuming that the n_i's are all equal to n, this means over n^d evaluations; clearly, this is unacceptable. The exponential growth of the number of evaluation points is known as the *curse of dimensionality.*

It turns out that the above tensor product construction is not particularly good in terms of accuracy. In fact, we can obtain similar levels of accuracy with much fewer evaluations if we use a more clever method. In the next section will introduce a sparse grid interpolation formula which allows us to greatly reduce the number of function evaluations while giving very good results in terms of accuracy. The tensor product formula is, however, far from useless, since it is crucial for constructing the sparse grid interpolation formula.

3.4 Multivariate sparse grid interpolation

Finally we have the necessary methods to introduce multivariate sparse grid interpolation, which lies at the heart of our method. As we mentioned at the end of the previous section, the problem with the tensor product interpolation formula is that it involves evaluating the function f far too many times. We call the resulting d-dimensional grid of interpolation points a *full grid*. By contrast, our goal is now to make do with a so-called *sparse grid*, which should have much fewer points than the full grid while still producing an interpolating function of comparable accuracy. This may seem like a very optimistic goal, but recall from Subsection 3.2.2 that changing the choice of interpolation points can greatly affect the properties of an interpolation formula; what we are doing now is similar, but rather than changing the interpolation points, we use a smaller number of them and change the construction of the interpolating function itself.

The reduction from a full grid to a sparse grid is based on an ingenious construction by S.A. Smolyak, whose original paper was published in 1963 [9]. The Smolyak construction is based on using a sequence of univariate interpolation formulas U^i of increasing accuracy, and combining them cleverly using tensor products. The key requirement for this sequence is that it should be *nested*: for each $i > 0$, we require that U^i uses all the interpolation points of U^{i-1}. Denoting by X^i the set of interpolation points for U^i, this condition can be written as $X^{i-1} \subset X^i$ for all $i > 0$.

As we mentioned in Section 3.2, this section is also based on [6]. However, we use somewhat different notation, so readers who are using both our thesis and [6] should do so carefully.

We construct the sequence U^i using the univariate Lagrangian interpolation formula with Chebyshev–Gauss–Lobatto points which we described in Subsection 3.2.2. Define $X^0 = \{\frac{a+b}{2}\}$, and for $i > 0$ let the set X^i be given by

$$X^i := \left\{ x_j = \frac{a+b}{2} + \frac{a-b}{2}\cos(j\pi/2^i), \quad j = 0, \ldots, 2^i \right\}. \tag{3.12}$$

This is precisely the set we had in (3.8), but we take $2^i + 1$ points instead of $n+1$. It is easy to check that the definition above produces a nested sequence of sets X^i; in fact, the points in X^{i-1} are obtained by taking every second point from the set X^i. The exception we mentioned above is that the set X^0 is defined to be the midpoint of the interval $[a, b]$, but the sequence is still nested since we have $X^1 = \{a, \frac{a+b}{2}, b\}$.

We now denote the basis functions for U^i corresponding to $x_j \in X^i$ by

$l_j^i(x)$, and define them as we did in (3.5):

$$l_j^i(x) := \prod_{x_k \in X^i} \frac{x - x_k}{x_j - x_k}. \tag{3.13}$$

As in the case of the sets X^i, we treat the case $i = 0$ separately, where we simply define $l_0^0(x) = 1$. Thus, according to (3.6), we have $U^0(f) = f(\frac{a+b}{2})$, and for $i > 0$ the formulas U^i are given by

$$U^i := \sum_{x_j \in X^i} l_j^i(x) f(x_j), \tag{3.14}$$

where X^i is given by (3.12) and l_j^i by (3.13).

Let $\alpha = (\alpha_1, \ldots, \alpha_d) \in \mathbb{N}^d$ be a d-dimensional *multi-index* as described in Section 1.1, and define $|\alpha|_1 = \alpha_1 + \ldots + \alpha_d$. Furthermore, define the *difference operators* Δ^i by $\Delta^0 = U^0$ and $\Delta^i = U^i - U^{i-1}$ for $i > 0$.

We are finally ready to present the multivariate sparse grid interpolation formula of order k:

$$\begin{aligned} A_d^k(f) &= \sum_{|\alpha| \leq k} (\Delta^{\alpha_1} \otimes \cdots \otimes \Delta^{\alpha_d})(f) \\ &= A_d^{k-1}(f) + \sum_{|\alpha| = k} (\Delta^{\alpha_1} \otimes \cdots \otimes \Delta^{\alpha_d})(f), \end{aligned} \tag{3.15}$$

where the sums are taken over all multi-indices α satisfying the given condition, and the term $A_d^{k-1}(f)$ vanishes when $k = 0$.

The interpolation formula A_d^k given by (3.15) is the sparse grid version of the tensor product interpolation formula $(U^k \otimes \cdots \otimes U^k)(f)$. It uses a fraction of the number of grid points, but maintains the accuracy of the tensor product version up to a logarithmic factor. The reason for the reduction in the number of grid points is the condition $|\alpha| \leq k$ in the sum of (3.15); since the difference operator Δ^i uses 2^i interpolation points (as many as U^i), the tensor products $\Delta^{\alpha_1} \otimes \cdots \otimes \Delta^{\alpha_d}$ use $\prod_{i=1}^d 2^{\alpha_i} = 2^{|\alpha|_1} \leq 2^k$ interpolation points. By contrast, the full grid of order k contains $2^{k \cdot d}$ interpolation points. Of course, there are many multi-indices α with $|\alpha|_1 \leq k$ and we are summing over all of them, but the reduction is still quite significant.

Observe that, as (3.15) indicates, the formula A_d^k inherits the nesting property of the sets X^i. This has two very convenient consequences: firstly, we can increase the order k without having to recompute everything; secondly, we can get an error estimate without needing any additional function evaluations by comparing the interpolating functions $A_d^k(f)$ and $A_d^{k-1}(f)$, since $A_d^k(f)$ contains $A_d^{k-1}(f)$.

It is also possible to rewrite (3.15) in terms of the univariate interpolation formulas U^i. In [10], Wasilkowski and Woźniakowski showed that the Smolyak interpolation operator $A_d^k(f)$ can be written as

$$A_d^k(f) = \sum_{k-d+1 \leq |\alpha| \leq k} (-1)^{k-|\alpha|} \binom{d-1}{k-|\alpha|} \left(U^{\alpha_1} \otimes \cdots \otimes U^{\alpha_d}\right)(f). \tag{3.16}$$

Note that $|\alpha| \geq 0$ since the elements of α are non-negative integers. The lower bound $|\alpha| \geq k - d + 1$ ensures that the binomial coefficient is well-defined.

The form given by (3.16) allows us to directly use the tensor product interpolation described in Section 3.3, which will prove useful when we describe the implementation of the sparse grid interpolation in Subsection 4.1.5. It also directly gives us the set of interpolation points $\eta(k, d)$, since we can replace the interpolation formulas U^i by their corresponding CGL node sets X^i and change (3.16) accordingly to obtain

$$\eta(k, d) = \bigcup_{|\alpha|=k} \left(X^{\alpha_1} \times \cdots \times X^{\alpha_d}\right), \tag{3.17}$$

where $X^{\alpha_1} \times \cdots \times X^{\alpha_d}$ is the Cartesian product of all X^{α_i}; that is, the set of all points $(x^1, \ldots, x^d)$ such that $x^i \in X^{\alpha_i}$ for all $1 \leq i \leq d$. Note that in (3.16) we only take unions over multi-indices α that satisfy $|\alpha| = k$, because the set of points corresponding to any multi-index β such that $|\beta| < k$ is contained in the set of points for some α satisfying $|\alpha| = k$; this is a consequence of the fact that the sets X^i are nested, as described in Section 3.4.

As an example of what the interpolation node sets $\eta(k, d)$ actually look like, Figure 3.2 shows the sets $\eta(k, 2)$ for $k = 1, \ldots, 6$; the set $\eta(0, 2)$ only contains the midpoint of the interpolation region, and is therefore omitted.

3.5 Conformal mapping

Recall from Section 3.1 that we have a nominal domain $\widetilde{\Omega}$ and a stochastic domain $\Omega(s_1, \ldots, s_n)$. We can discretize the stochastic domain by picking a set of realizations of the random parameters $(s_1, \ldots, s_n)$; these realizations are dictated by our choice of interpolation formula, as described in Section 3.1. Since fixing the parameters makes the problem deterministic, we can solve the problem separately for each of these realizations. The only open question is then how we can map the solutions back to the nominal domain Ω in a consistent way.

In our work, we chose to map the solutions using conformal mappings. A mapping between two domains is called conformal if it preserves angles locally. This means that if two lines intersect at, say, a 90° angle, the curves obtained by mapping those two lines conformally will also intersect at a 90° angle. Note that this property only applies locally — generally speaking, the lines will not be straight lines anymore after being mapped.

The existence of a conformal mapping is unfortunately not guaranteed between arbitrary three-dimensional domains; in fact, it generally does not exist at all. However, in two dimensions one can always find a conformal mapping between two simply connected domains (i.e. domains that do not have holes). At the end of this section we will describe a way of using these to obtain a mapping for certain types of three-dimensional problems as well, but for general three-dimensional problems some other kind of mapping may be needed.

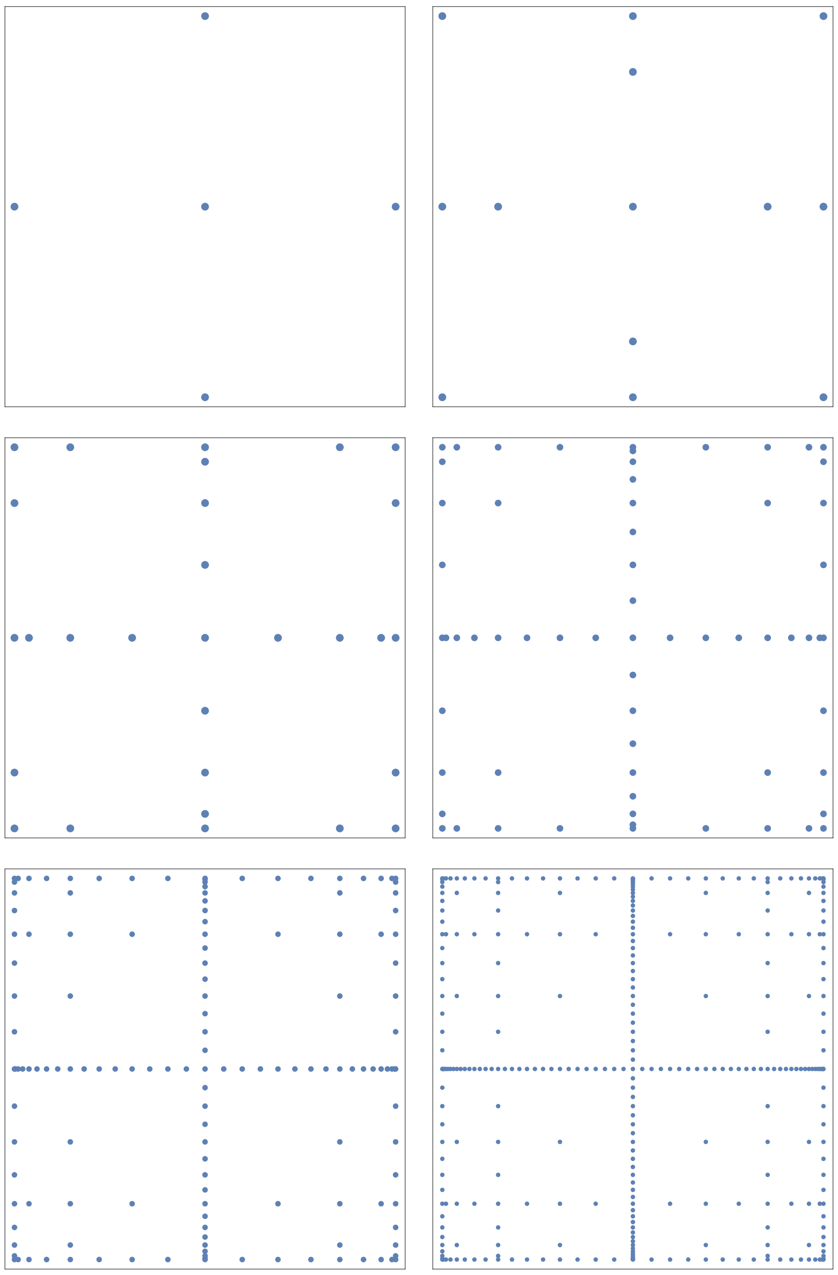
Figure 3.2: Interpolation node sets $\eta(k, d)$ given by (3.17), with $d = 2$ and $k = 1, \ldots, 6$. Note that the frames are merely for visual purposes; all nodes at the edges lie on the boundary of the interpolation range.

We will soon describe a simple way of finding a conformal mapping from any two-dimensional, simply connected domain to a rectangle. For a more detailed explanation, we refer the reader to [5]. Finding the mapping between two arbitrary domains is then a simple matter of mapping both domains to rectangles and inverting one of the mappings. There is a slight caveat, however: the exact dimensions of the rectangle depend on the domain, so there is no guarantee that the domains will be mapped to similar rectangles. In our work, this turns out not to be a problem, since it is not strictly necessary for the mapping to be conformal; conformal mappings are merely a convenient tool we are using to obtain mappings which are sufficiently consistent. As such, we chose to bypass this issue by simply scaling the rectangles to the same size. We do need to emphasize that this means the resulting mapping is not actually conformal, since scaling just one dimension is not a conformal transformation.

Disclaimers aside, let us see how exactly the conformal mappings can be found. Take some domain Ω, and denote by R_h the rectangle $[0,1] \times [0,h]$, where $h > 0$ is a positive constant depending only on the domain Ω. This constant will eventually vanish when we scale the rectangles.

Fix four distinct points $z_1, z_2, z_3, z_4 \in \partial\Omega$ on the boundary of Ω. These points may be chosen arbitrarily, as long as they are distinct and ordered counter-clockwise along the domain boundary $\partial\Omega$. The choice is significant, however, since these points will be mapped to the corners of the rectangle; in particular, the choice should be consistent with the similar points chosen in the nominal domain $\widetilde{\Omega}$, but we will get to that later on.

Fixing the four points as above effectively splits the boundary $\partial\Omega$ into four arcs, which we shall denote by (z_1, z_2), (z_2, z_3), and so on. The significance of these arcs will soon become apparent. Consider now the following (relatively simple) PDE problem: find a function u satisfying the equations

$$\begin{cases} \Delta u(\boldsymbol{x}) = 0 & \text{for all } \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = \psi_D(\boldsymbol{x}) & \text{for all } \boldsymbol{x} \in D \subset \partial\Omega, \\ \frac{\partial}{\partial \boldsymbol{n}} u(\boldsymbol{x}) = 0 & \text{for all } \boldsymbol{x} \in \partial\Omega \setminus D, \end{cases} \tag{3.18}$$

where D is a subset of the boundary $\partial\Omega$, ψ_D is a function defined on D, and $\Delta u(\boldsymbol{x})$ is the Laplacian of u, which is given in two dimensions by

$$\Delta u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}.$$

The term $\frac{\partial}{\partial \boldsymbol{n}} u(\boldsymbol{x})$ in the last equation gives the derivative of $u(\boldsymbol{x})$ in the direction of the outward unit normal $\boldsymbol{n}$, and the set $\partial\Omega \setminus D$ consists of all points on the boundary which do not belong to the set D.

The problem (3.18) is a simple PDE with boundary conditions, which is relatively easy to solve using a suitable piece of software; for our work, we used the new `NDSolve‘FEM` package introduced in Mathematica 10. The first equation of (3.18) is Laplace's equation, and the second equation is a Dirichlet boundary condition which fixes the values of u on the boundary.

The third equation is the homogeneous Neumann boundary condition; note that in finite elements, this boundary condition is implicitly applied whenever no other boundary condition is specified. The existence of a unique solution u for (3.18) is guaranteed as long as ψ_D is nice enough, which it will be in our application. For more discussion on the existence of solutions, we refer the reader to [2] and [4].

Now, set $D = (z_1, z_2) \cup (z_3, z_4)$, the union of two opposite arcs, and define

$$\psi_D(\boldsymbol{x}) := \begin{cases} 1, & \boldsymbol{x} \in (z_1, z_2), \\ 0, & \boldsymbol{x} \in (z_3, z_4). \end{cases} \tag{3.19}$$

With these definitions and a simply connected domain Ω, the Laplace problem (3.18) always admits a unique solution u. Similarly, if we set $D' = (z_2, z_3) \cup (z_4, z_1)$ and define

$$\psi_{D'}(\boldsymbol{x}) := \begin{cases} 1, & \boldsymbol{x} \in (z_2, z_3), \\ 0, & \boldsymbol{x} \in (z_4, z_1), \end{cases} \tag{3.20}$$

we obtain another unique solution v to (3.18). Combining these two solutions then gives us a conformal mapping $\varphi : \Omega \to R_h$ by setting

$$\varphi(\boldsymbol{x}) := (u(\boldsymbol{x}), hv(\boldsymbol{x})),$$

where h is a constant given by

$$h := \iint_\Omega |\nabla u|^2 \; dxdy. \tag{3.21}$$

Now that we know how to compute the conformal mappings, suppose that we have the mappings φ and $\widetilde{\varphi}$ with respective constant h and $\widetilde{h}$, such that φ maps the domain Ω to R_h and $\widetilde{\varphi}$ maps the nominal domain $\widetilde{\Omega}$ to $\widetilde{R}_h$. As a conformal mapping, φ is guaranteed to be invertible; thus we can obtain the mapping from $\widetilde{\Omega}$ to Ω by first using $\widetilde{\varphi}$ to map from $\widetilde{\Omega}$ to $\widetilde{R}_h$, then scaling the y-coordinate to map from $\widetilde{R}_h$ to R_h, and finally using φ^{-1} to map from R_h to Ω.

Finally, let us write down the entire mapping to a domain $\Omega(\boldsymbol{s})$. Denote the scaling function by T, defining it as

$$T(\boldsymbol{x}) = T(x, y) = \left(x, \frac{\widetilde{h}}{h} y \right).$$

Then the mapping from the nominal domain $\widetilde{\Omega}$ to $\Omega(\boldsymbol{s})$ is given by

$$\phi(\boldsymbol{x}; \boldsymbol{s}) = \varphi^{-1}(T(\widetilde{\varphi}(\boldsymbol{x}))), \text{ for } \boldsymbol{x} \in \widetilde{\Omega}. \tag{3.22}$$

We use the notation $\phi(\boldsymbol{x}; \boldsymbol{s})$ since both φ^{-1} and T above depend implicitly on $\boldsymbol{s}$. Note, however, that $\widetilde{\varphi}$ depends only on $\widetilde{\boldsymbol{s}}$, the parameter vector for the nominal region. Thus in applications $\widetilde{\varphi}$ only needs to be computed once.

Observe that φ^{-1} in (3.22) might not need to be computed explicitly. If the goal is to essentially obtain the entire solution $\sigma(\boldsymbol{x};\boldsymbol{s})$ for all parameter vectors $\boldsymbol{s}$, then inverting φ is indeed necessary. However, as we described at the start of this chapter, in our work we settle for taking the solutions at just one measurement point $\tilde{\boldsymbol{x}} \in \widetilde{\Omega}$; that is, finding the function $\sigma(\phi(\tilde{\boldsymbol{x}};\boldsymbol{s});\boldsymbol{s})$ mentioned at the end of Section 3.1. This means that we only need the mapping ϕ for the point $\tilde{\boldsymbol{x}}$ rather than the entire domain $\widetilde{\Omega}$, so it is sufficient to invert the mapping φ pointwise.

Recall that if $f^{-1}(\boldsymbol{x})$ is the inverse of $f(\boldsymbol{x})$, then it satisfies

$$f(f^{-1}(\boldsymbol{x})) = \boldsymbol{x}$$

for all $\boldsymbol{x}$ such that $f^{-1}(\boldsymbol{x})$ is defined. We can use this by applying the mapping φ to both sides of (3.22), which gives us

$$\varphi(\phi(\boldsymbol{x};\boldsymbol{s})) = T(\widetilde{\varphi}(\boldsymbol{x})). \tag{3.23}$$

Thus, $\phi(\boldsymbol{x};\boldsymbol{s})$ is the point in $\Omega(\boldsymbol{s})$ which satisfies equation (3.23) above, and since φ is invertible as a conformal mapping, this point is unique. Furthermore, if we denote the Laplace solutions defining φ and $\widetilde{\varphi}$ by (u, v) and $(\widetilde{u}, \widetilde{v})$, respectively, then we have

$$\begin{aligned}\varphi(\boldsymbol{x}) &= (u(\boldsymbol{x}), hv(\boldsymbol{x})),\\ \widetilde{\varphi}(\boldsymbol{x}) &= (\widetilde{u}(\boldsymbol{x}), \widetilde{h}\widetilde{v}(\boldsymbol{x})),\end{aligned}$$

and (3.23) yields

$$(u(\phi(\boldsymbol{x};\boldsymbol{s})), hv(\phi(\boldsymbol{x};\boldsymbol{s}))) = (\widetilde{u}(\boldsymbol{x}), h\widetilde{v}(\boldsymbol{x})), \tag{3.24}$$

where $\widetilde{h}$ has been replaced by h on the right-hand side due to the scaling function T.

Finally, since both sides of (3.24) above are two-dimensional vectors, we can split it into two equations, which gives us

$$u(\phi(\boldsymbol{x};\boldsymbol{s})) = \widetilde{u}(\boldsymbol{x}), \tag{3.25}$$
$$v(\phi(\boldsymbol{x};\boldsymbol{s})) = \widetilde{v}(\boldsymbol{x}). \tag{3.26}$$

Thus the mapping $\phi(\boldsymbol{x};\boldsymbol{s})$ of a point $\boldsymbol{x}$ is the unique point in $\Omega(\boldsymbol{s})$ which satisfies (3.25) and (3.26) above, and this point can be found without needing to compute φ^{-1}.

This pointwise approach might seem like a complicated way of doing things, but implementing it can be much easier than computing φ^{-1} explicitly. We used Mathematica to find the mapping, and while the `NDSolve‘FEM‘` package easily gives us interpolating functions for the mappings φ and $\widetilde{\varphi}$, inverting φ^{-1} can be very tricky. In fact, even though we know that φ is invertible, its interpolating function might not be. However, solving a pair of equations such as the one given by (3.25) and (3.26) is very easy in Mathematica; details of how this was done can be found in Chapter 4.

Finally, we describe a way of extending this to three dimensions for some types of problems; specifically, problems where we know that the point which we are mapping should only move on a two-dimensional surface. In such cases, mapping the point is a simple matter of applying the conformal mapping scheme on that surface. Of course, in general it can be quite difficult to know if we should constrain the mapping to two dimensions, but sometimes it is clear that we should. A prime example of this is the case where we have a two-dimensional domain depending on the parameters $\boldsymbol{s}$ which has been extruded to create a beam or revolved around an axis to create an axisymmetric object. In such cases, where the cross-section is identical throughout the entire object, it is clear that the point should only move within the cross-section. The same principle can be used for more general problems, but the quality of results may vary.

Chapter 4

Implementation

In this chapter, we describe how we implemented the methods described in the previous chapter. We begin by discussing the solution for the two-dimensional model problem that we defined in Sections 2.1 and 2.2, before moving on to the three-dimensional problem.

As we have mentioned before, we used Mathematica and Abaqus for solving our model problems; this chapter will focus on the details of how this was done. We will however include general remarks where appropriate, and much of the Mathematica/Abaqus-specific material should translate to other software with relative ease. The goal is to give the reader a solid understanding of how to implement this method regardless of the software used.

Naturally, the disclaimers from previous sections still apply. For instance, we assume that the reader is familiar with solving FEM problems, which is why we will not include a detailed explanation of how to create a model in Abaqus or any other FEM program. We will, however, describe how we modified the model to change the parameters; in other words, we assume the reader has a solution for the deterministic problem, and we only focus on how to use that solution to solve the stochastic problem.

As a further disclaimer, note that our work is mainly a proof of concept. As such, our implementation has not yet been optimized, and some of our solutions may even seem crude; our goal was simply to make a working prototype. In other words, while the methods we describe do work as advertised, there may be much room for improvement in terms of speed and convenience. We leave such improvements to the reader, although we will include any thoughts we have on improving the method.

4.1 2D problem

Let us consider first the two-dimensional model problem of Sections 2.1 and 2.2. This part of our work was done entirely with Mathematica 10. We used the `NDSolve‘FEM`-package both to solve the deterministic problem of Section 2.1 and to compute the conformal mappings described in Section 3.5.

Since Mathematica 10 was not yet available on the CSC supercomputers when the computations were done for this thesis, everything related to the

2D problem was run on a desktop computer. The machine we used had 8 GB of memory and an Intel Xeon X3450 quad-core processor running at 2.67 GHz with eight threads.

Since the entire solution process is somewhat lengthy, we have split the implementation details into five parts. We begin by defining the domain and its boundary conditions. After this we move on to constructing the mesh, finding the conformal mapping, solving the deterministic problem, and finally constructing the interpolating function.

4.1.1 Parameterized domain and boundary conditions

The first step was to define the domain in which we are solving the equations (2.1)–(2.4) that were given in Section 2.1. Consider the L-shaped domain obtained by taking the unit square $[0, 1] \times [0, 1]$ and removing the top-right quarter of it (i.e. the square $[\frac{1}{2}, 1] \times [\frac{1}{2}, 1]$). The domain we used was based on this L-shape, but we used a kind of fillet to smooth out the sharp inside corner of the domain.

Since we wanted to obtain a geometry which depends on several parameters, we replaced the corner with two 45° circle arcs and a line segment between them, as can be seen in Figure 4.1. The two circle arcs lay tangent to the line segments on either side of them and they were given the same radius r. The length of the line segment between them is denoted by d. In our computations, the parameters were constrained to the range $r, d \in [0.075, 0.125]$, and the nominal domain was given by setting $r = d = 0.1$.

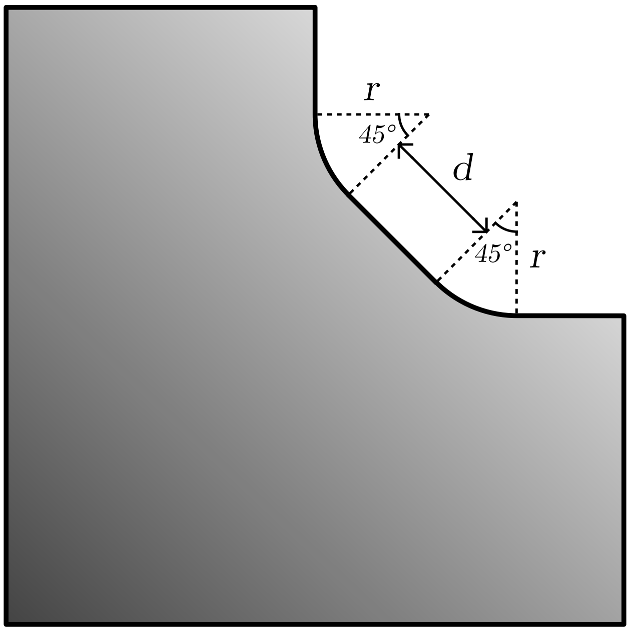

Figure 4.1: Geometry of the domain and its dependence on r and d.

Observe that the domain obtained by the construction above is symmetric, and it is uniquely determined by the values of the two parameters r and d, which will serve as the random variables in our model problem. It may seem odd that we have carefully described a multi-dimensional interpolation routine only to use a domain of just two parameters, but the reason for this is quite simple: if the method works with two parameters, it will work with several. In particular, the mappings we described in Section 3.5 do not use any information related to the number of parameters; thus it would be unnecessary to make the model problem any more complex. In Chapter 5 we will of course discuss how an increase in the number of parameters affects computation times and accuracy, but these are properties of the interpolation scheme and are effectively independent of the model problem.

Now that we have defined the domain for the 2D problem, the only things missing from the deterministic problem of Section 2.1 are the boundary conditions given by Equations (2.3) and (2.4). In our model problem, the left edge of the domain was held fixed in both x- and y-directions, which corresponds to the Dirichlet boundary condition (2.3). A uniform force was applied to the bottom edge, which in turn corresponds to the Neumann boundary condition (2.4). The force was directed upwards, and since the force was uniform, the load vector $\boldsymbol{g}$ was a constant vector in the positive y-direction. The magnitude of the load vector was chosen to be $|\boldsymbol{g}| = 1$. On the rest of the boundary, we applied the homogeneous Neumann boundary condition $\boldsymbol{g} = 0$.

4.1.2 Meshing the domain

The next step was to mesh the domain. The `NDSolve‘FEM`-package in Mathematica 10 is capable of autonomously meshing a domain. The only input it needs is a region, constructed either implicitly as a set of equations that points in the region must satisfy, or explicitly using simple geometric objects and Boolean operations such as unions and intersections. However, while this approach works quite well, problems sometimes arise when dealing with regions parameterized by floating point numbers.

Instead of letting Mathematica do all the work, we created the boundary mesh manually instead, and then let Mathematica fill in the rest of the mesh by itself. This is quite easy to do, all it takes is a list of the coordinates of every node on the boundary of the domain; with this input, the `ToBoundaryMesh`-function creates a boundary mesh in the format required by the FEM-package. Of course, this is somewhat cumbersome, but it has the benefit of giving the user complete control over the accuracy of the mesh.

We created a boundary mesh consisting of elements with a length of approximately 0.0023; as the domain Ω has a diameter of 1, this results in approximately 430 elements on each side of the domain. The length of the boundary elements was further divided by 3 for the inside corner of the domain, since that was the area where we wanted to measure the stresses; all in all, the boundary mesh consisted of 1850–1950 elements, depending on the values of r and d.

The boundary mesh was used as input for the `ToElementMesh`-function, which creates a triangle mesh with a maximum element size defined by the user. In this context the element size is the area of the element, which is why we chose the maximum size to be 0.00005. This results in a mesh where the largest element has a side length of approximately 0.014, which is six times as large as that of the boundary elements; the total number of elements in the mesh was approximately 37000. The mesh may well be much finer than actually necessary, we simply used the finest possible mesh given the available computational resources, mainly system memory. The computations were run in parallel on seven Mathematica kernels, and a finer mesh would have exhausted the 8 GB of system memory available.

4.1.3 Conformal mapping

Now that we have described the meshing process, we can move on to solving the two FEM-problems: the conformal mapping between domains, and the deterministic problem of Section 2.1. We begin with the former, since solving it is easier. The following few sections will include some Mathematica code, but we will not go into great detail about it; if the reader wishes to learn more about using FEM in Mathematica, there are a number of good tutorials on the Wolfram website.

Recall from Section 3.5 that we only need to solve two Laplace problems to obtain a conformal mapping from any domain Ω to a rectangle R_h. The first step is to pick four distinct points $z_1, z_2, z_3, z_4 \in \partial\Omega$, where $\partial\Omega$ is the boundary of the domain Ω. These are the points which will be mapped to the corners of the rectangle R_h, and the quality of the conformal mappings depends to some extent on how they are chosen. It may be a good idea to experiment with several choices and see how they work out, some examples of good choices can be found in the test cases used in [5].

In our simple model, the choice is quite natural, since the L-shaped region already resembles a rectangle that has been bent at the middle. We therefore chose the following four points:

$$z_1 = (1/2, 1), \qquad z_2 = (0, 1), \qquad z_3 = (1, 0), \qquad z_4 = (1, 1/2).$$

Note that these points have been chosen in counter-clockwise order around the boundary, as required in Section 3.5. This choice then gives rise to the following two Laplace problems:

$$\begin{cases} \Delta u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = 1, & \boldsymbol{x} \in (z_1, z_2), \\ u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in (z_3, z_4), \\ \frac{\partial}{\partial n} u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \partial\Omega \setminus D, \end{cases} \qquad \begin{cases} \Delta v(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \Omega, \\ v(\boldsymbol{x}) = 1, & \boldsymbol{x} \in (z_2, z_3), \\ v(\boldsymbol{x}) = 0, & \boldsymbol{x} \in (z_4, z_1), \\ \frac{\partial}{\partial n} u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \partial\Omega \setminus D'. \end{cases} \tag{4.1}$$

Above we have used the same notation as in Section 3.5: the sets (z_1, z_2) and so on are subsets of the boundary $\partial\Omega$, corresponding to the part of the boundary that lies between the respective points. Thus e.g. the set (z_1, z_2) is the line from $(1/2, 1)$ to $(0, 1)$ in our model problem; in other words, the set of points satisfying $y = 1$. Similarly, (z_2, z_3) is the set of points satisfying either $x = 0$ or $y = 0$, and so on. Finally, as in Section 3.5, the sets D

and D' denote the subset of the boundary where the Dirichlet boundary conditions are applied; in other words, the homogeneous Neumann boundary conditions in (4.1) are applied everywhere on the boundary $\partial\Omega$ where no Dirichlet condition is given.

Solving the Laplace problems of (4.1) is quite easy: we only need to encode the equation and boundary condition in a way which Mathematica can understand. This is quite easy with the `NDSolve‘FEM`-package; in fact, the solution to the first of these Laplace problems is given by the following lines of code:

```
op = Laplacian[u[x, y], {x, y}];
dirichlet = {DirichletCondition[u[x, y] == 0, x == 1],
             DirichletCondition[u[x, y] == 1, y == 1]};
NDSolveValue[{op == 0, dirichlet}, u, {x, y} \[Element] mesh}];
```

The last line of code above will output the solution to the Laplace problem. Since FEM problems are only solved at the nodes of the mesh, Mathematica interpolates the solution between the nodes.

The input for `NDSolveValue` consists of three parts: a set of the equations and boundary conditions, the function to be solved, and a list of the independent variables. `\[Element]` is the set membership symbol $\in$, and `mesh` is the output of the `ToElementMesh`-function described in the previous section; thus, the last part indicates that the point (x, y) is in the area defined by the mesh. The first line of code indicates that the operator `op` is the Laplacian of u with respect to x and y. The second and third line give the Dirichlet boundary conditions as described in Section 3.5: $u(\boldsymbol{x}) = 0$ on (z_3, z_4) and $u(\boldsymbol{x}) = 1$ on (z_1, z_2). Observe that the homogeneous Neumann condition in (4.1) does not need to be encoded, as it is implicitly applied on all parts of the boundary where no other boundary condition is given.

The solution to the second Laplace problem in (4.1) is obtained by simply applying the Dirichlet boundary conditions on (z_2, z_3) and (z_4, z_1) instead; the boundary conditions are thus replaced by the following:

```
{DirichletCondition[u[x, y] == 0, x >= 1/2 && y >= 1/2],
 DirichletCondition[u[x, y] == 1, x == 0 || y == 0]};
```

The rest of the code is identical. Note that the conditions given for x and y in `DirichletCondition` are only checked along the boundary of the domain; it makes no difference that the first line above has conditions which are also satisfied inside the domain. This makes giving the boundary conditions much easier, since we do not need to tell Mathematica anything about the shape of the inside corner.

Figure 4.2 demonstrates what the conformal mapping $\widetilde{\varphi}(\boldsymbol{x})$ from the nominal domain $\widetilde{\Omega}$ to the corresponding rectangle $R_{\widetilde{h}}$ looks like. The first image shows the shape of the nominal domain, and the second image gives a 20×20 rectangular grid in the rectangle $R_{\widetilde{h}}$. The third image in Figure 4.2 gives the preimage of the rectangular grid under $\widetilde{\varphi}(\boldsymbol{x})$; in other words, $\varphi(\boldsymbol{x})$ maps the curves in the third image to the grid displayed in the second image. We can see that the mapping is symmetric, which is to be expected with a symmetric domain and a symmetric choice of the points z_1, z_2, z_3 and z_4. Observe that the curves intersect at $90°$ as expected, since the conformal mapping preserves angles locally. The parameter $\widetilde{h}$ which determines the height of the rectangle $R_{\widetilde{h}}$ is given by (3.21), which yields $\widetilde{h} \approx 0.415$; however, this value is ignored when we map $\widetilde{\Omega}$ to another domain $\Omega(r,d)$.

Now we have the mapping from a domain Ω to a rectangle R_h, but what we really want is the mapping from the nominal domain $\widetilde{\Omega}$ to another domain $\Omega(r,d)$. Recall that we described a way of doing this pointwise at the end of Section 3.5. The goal is to obtain the mapping $\varphi(\boldsymbol{x};\boldsymbol{s}) = \varphi(\boldsymbol{x};r,d)$ by solving the following two equations:

$$\begin{aligned} u(\phi(\boldsymbol{x};\boldsymbol{s})) &= \widetilde{u}(\boldsymbol{x}), \\ v(\phi(\boldsymbol{x};\boldsymbol{s})) &= \widetilde{v}(\boldsymbol{x}). \end{aligned}$$

These are the equations (3.25) and (3.26) which were provided in Section 3.5.

Now, suppose that we have a point $\boldsymbol{x} = (x,y) \in \widetilde{\Omega}$ that we want to map to the domain $\Omega(r,d)$. The right-hand side of the equations above only requires the functions $\widetilde{u}$ and $\widetilde{v}$, which are obtained by solving the Laplace problems in the nominal region. Since we have already described how to solve them, we assume that we have the solutions $x_0 := \widetilde{u}(\boldsymbol{x})$ and $y_0 := \widetilde{v}(\boldsymbol{x})$. Similarly, we can solve the functions u and v corresponding to the domain $\Omega(r,d)$.

All that remains is then to find the point which u maps to x_0 and v maps to y_0; this point is unique and guaranteed to exist. Finding it is just a matter of solving two equations, and Mathematica has a number of ways for doing this. The way that we found to work well is to use the function `NMinimize`, which numerically finds the minimum of a given function. This can be used to solve equations by minimizing the norm of the difference between the two sides of the equation. The last bit of code needed to obtain the conformal mapping is then the following:

```
NMinimize[Norm[{x0 - u[x,y], y0 - v[x,y]}],
          {x,y} \[Element] MeshRegion[mesh]];
```

Note the additional `MeshRegion`-function; this is needed for `NMinimize` since the mesh is only used to specify the search region. We should also point out that the above code may require Mathematica 10.0.2 or a later version; in earlier versions, the following code can be used instead:

```
NMinimize[{Norm[{x0 - u[x,y], y0 - v[x,y]}],
          SignedRegionDistance[MeshRegion[mesh]][{x,y}] <= 0},
          {x,y}];
```

Observe that the functions $\widetilde{u}$ and $\widetilde{v}$ only need to be solved once, since the same functions can be used regardless of the target domain $\Omega(r,d)$. They also only need to be evaluated once for each mapped point.

Now that we have described the implementation of the conformal mapping, let us see how accurate it is. Recall from Subsection 4.1.2 that the

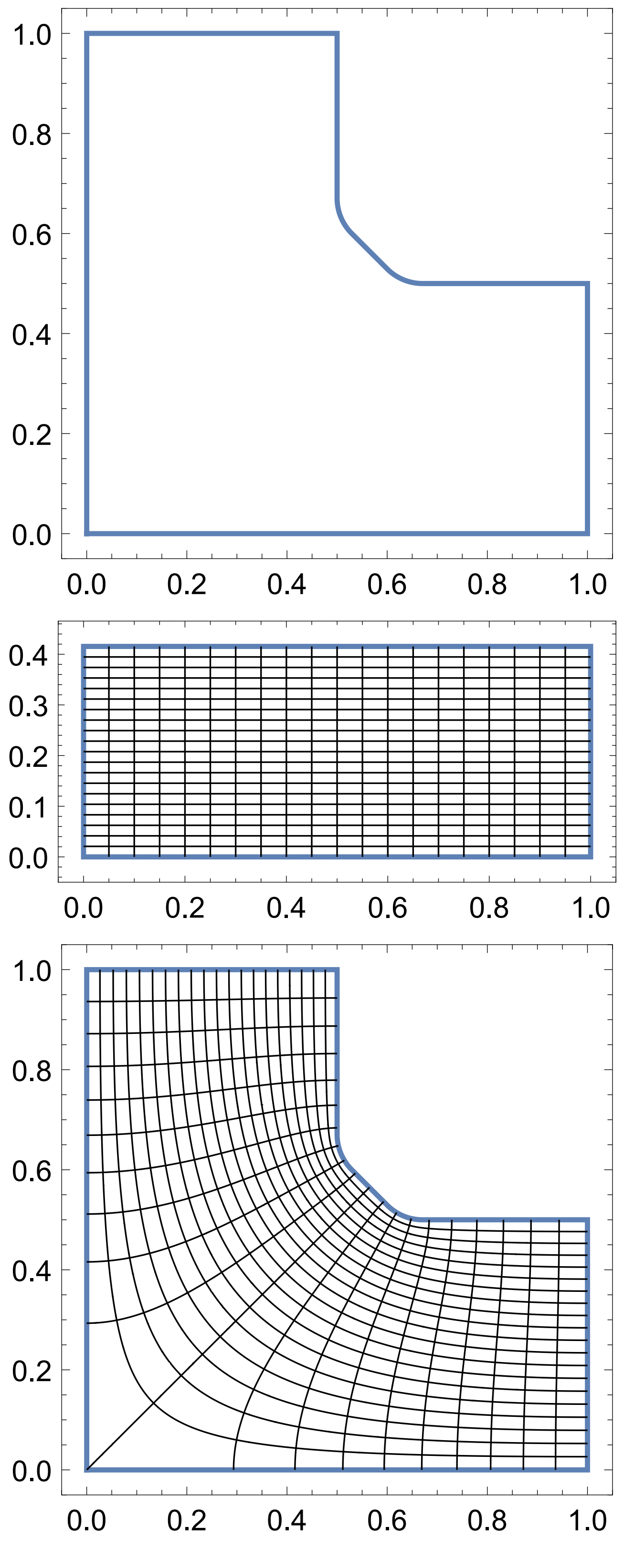

Figure 4.2: A demonstration of the conformal mapping $\widetilde{\varphi}(\boldsymbol{x})$ from the nominal domain $\widetilde{\Omega}$ to the rectangle $R_{\widetilde{h}}$. The first image shows the nominal domain, the second shows the rectangle and a 20×20 grid in it, and the third shows the curves in the nominal domain which $\widetilde{\varphi}(\boldsymbol{x})$ maps to the rectangular grid.

mesh we used consisted of 37 000 elements. In order to ensure that the conformal mapping was computed to a sufficient level of accuracy, we carried out a mesh convergence study to examine how the accuracy depends on the number of mesh elements. Scaling the values used in the construction of the mesh in Subsection 4.1.2, we created meshes with approximately $37000 \cdot 2^m$ elements, where $m = -10, \dots, 4$; the mesh corresponding to $m = 0$ was the one which we actually used in our work. For each of these meshes, we computed the conformal mapping of a point $\tilde{\boldsymbol{x}}$ from the nominal domain $\widetilde{\Omega}$ to another domain $\Omega(r, d)$; the exact location of the point $\tilde{\boldsymbol{x}}$ is given later on in Section 5.2, where we use it as the measurement point when we examine the results of our method.

The mesh convergence study was carried out for the four domains corresponding to the corners of the interpolation range, where r and d equal either 0.075 or 0.125. The worst accuracy occurred in the case where $r = d = 0.075$, which is not surprising since the inside corner of the domain $\Omega(r, d)$ is sharper when r and d are smaller. In Figure 4.3, we have plotted the accuracy of the conformal mapping against the number of mesh elements used. The mapping obtained using the densest mesh was used as a reference point, and the accuracy of all other meshes was computed as the absolute distance between the mapped point and the reference point; note that the width and height of the domain was 1. The accuracy of the mapping was approximately $4 \cdot 10^{-7}$ for the mesh which we described in Subsection 4.1.2; clearly, this accuracy is more than sufficient for our purposes.

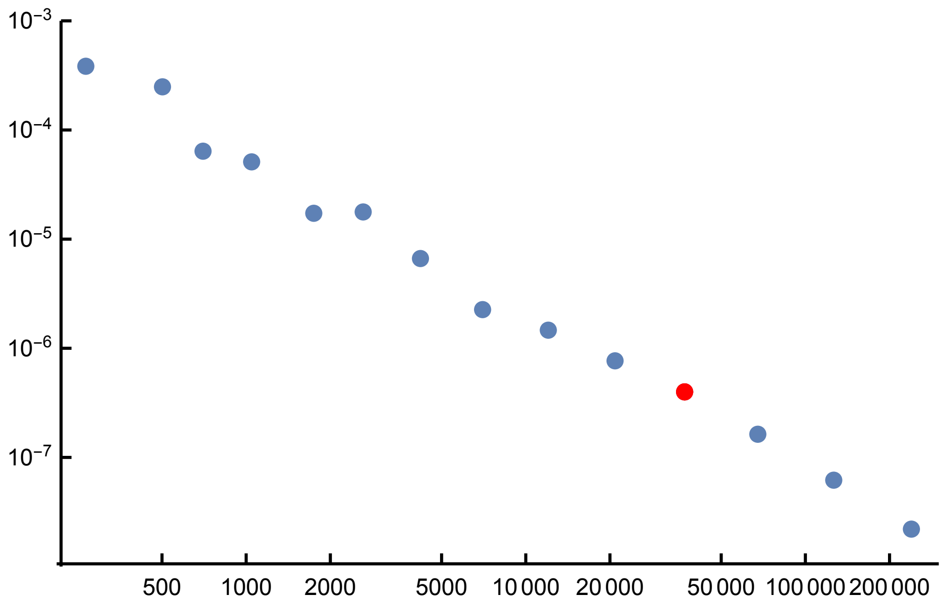

Figure 4.3: Semi-log plot of the accuracy of the conformal mapping as a function of the number of mesh elements. The accuracy was measured as the distance of the mapped point to the reference point which was computed using a mesh of almost 460 000 elements. The fourth point from the right has been marked in red to indicate the mesh which we used in our numerical experiments in Section 5.2.

In general, the conformal mappings can be computed quite accurately, as the only thing we need to do to obtain the mapping is solve a pair of Laplace problems. Of course, we also need to invert a conformal mapping pointwise to obtain the mapping from $\widetilde{\Omega}$ to $\Omega(r,d)$; this is what we used the `NMinimize`-function for earlier in this subsection. However, the inversion can be done with almost arbitrary accuracy, and so its effect on the accuracy of the mapping to $\Omega(r,d)$ is negligible. In our numerical experiments, we used a minimum accuracy of 10^{-14} for the inversion, as increasing the accuracy of `NMinimize` is much less costly than increasing the accuracy of the conformal mappings.

4.1.4 Solving the deterministic problem

The next step was to solve the deterministic problem which we introduced in Section 2.1. The procedure is very similar to how we solved the Laplace problems, but first let us manipulate the equations (2.1)–(2.2) into another form. The goal is to obtain an equation involving only the displacement vector $\boldsymbol{u}$. We will then solve the equation for $\boldsymbol{u}$ and use it to compute the stress component σ_{22}. The other stress components are excluded simply because we do not need them later on; the results were similar for all three stress components, which is why we will only demonstrate the results for one of them.

Substituting $\boldsymbol{\sigma}(\boldsymbol{u})$ from (2.1) into (2.2) and substituting $\boldsymbol{\varepsilon}(\boldsymbol{u})$ from (2.7) yields

$$\operatorname{div}\left(\lambda \operatorname{div}(\boldsymbol{u})\boldsymbol{I} + \mu\left(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^T\right)\right) = 0.$$

Above, $\operatorname{div}(\boldsymbol{u})$ is the vector divergence and $\nabla\boldsymbol{u}$ is the Jacobian matrix of $\boldsymbol{u}$. Denoting the components of $\boldsymbol{u}$ by u and v and writing the matrices out explicitly, we obtain

$$\operatorname{div}\left(\lambda\begin{pmatrix}\partial_x u + \partial_y v & 0\\ 0 & \partial_x u + \partial_y v\end{pmatrix} + \mu\begin{pmatrix}2\partial_x u & \partial_y u + \partial_x v\\ \partial_y u + \partial_x v & 2\partial_y v\end{pmatrix}\right) = 0, \quad (4.2)$$

where ∂_x denotes the partial derivative with respect to x.

Observe that the divergence now has a matrix as its argument. This is the vector-valued tensor divergence which we defined in (2.5). Applying this definition to (4.2) gives

$$\lambda\begin{pmatrix}\partial_{xx}u + \partial_{xy}v\\ \partial_{xy}u + \partial_{yy}v\end{pmatrix} + \mu\begin{pmatrix}2\partial_{xx}u + \partial_{yy}u + \partial_{xy}v\\ \partial_{xy}u + \partial_{xx}v + 2\partial_{yy}v\end{pmatrix} = 0. \quad (4.3)$$

Since a vector is zero if and only if each of its elements is zero, equation (4.3) above is equivalent to the following pair of equations:

$$\begin{cases}(\lambda+2\mu)\partial_{xx}u + (\lambda+\mu)\partial_{xy}v + \mu\partial_{yy}u = 0,\\ (\lambda+2\mu)\partial_{yy}v + (\lambda+\mu)\partial_{xy}u + \mu\partial_{xx}v = 0.\end{cases} \quad (4.4)$$

However, this cannot directly be used as input for Mathematica. Recall that the Neumann boundary condition (2.4) is given for the stress tensor $\boldsymbol{\sigma}$;

unfortunately, Mathematica has no way of knowing that this is the boundary condition we want. However, if we can give the equations (4.4) in terms of divergences, as in the original equations (2.2) and (2.4), then Mathematica will be able to interpret the Neumann boundary condition correctly.

Fortunately, writing (4.4) in terms of divergences is relatively simple. Indeed, the first equation can be written as

$$\nabla \cdot \left(\begin{pmatrix} \lambda + 2\mu & 0 \\ 0 & \mu \end{pmatrix} \cdot \nabla u + \begin{pmatrix} 0 & \lambda \\ \mu & 0 \end{pmatrix} \cdot \nabla v \right) = 0, \tag{4.5}$$

and the second equation admits a similar form.

Now that we have the necessary equations to find the displacement vector $\boldsymbol{u}$, all that remains is to obtain the stress component σ_{22} as a function of u and v. This can be done by directly solving from (2.1). Using the result already obtained in (4.2), we see that (2.1) can be written explicitly as

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{12} & \sigma_{22} \end{pmatrix} = \lambda \begin{pmatrix} \partial_x u + \partial_y v & 0 \\ 0 & \partial_x u + \partial_y v \end{pmatrix} + \mu \begin{pmatrix} 2\partial_x u & \partial_y u + \partial_x v \\ \partial_y u + \partial_x v & 2\partial_y v \end{pmatrix}.$$

This can be written as four separate equations, and the one involving σ_{22} yields

$$\sigma_{22} = (\lambda + 2\mu)\partial_y v + \lambda \partial_x u. \tag{4.6}$$

All in all, the plan is to solve the coupled PDEs of (4.4) using FEM, and use the partial derivatives of the solution to obtain σ_{22} from (4.6).

Now that we have all the equations we need, we can give the Mathematica code that was used for the actual computations. First, we need to obtain the displacements u and v. These are given by calling `NDSolveValue`:

```
{uif,vif} = NDSolveValue[{op=={0,neumann},dirichlet},{u,v},
                         {x,y} \[Element] mesh];
```

As before, `op` encodes the equation itself, while `neumann` and `dirichlet` give the boundary conditions. Note that the right-hand side of the equation is now a vector, since we are solving two PDEs. The Neumann condition is only applied to the second element of the vector, because the load vector $\boldsymbol{g}$ points in the positive y-direction and thus only has a y-component. Observe that the Neumann condition only needs to be specified when the load vector is non-zero; as we mentioned in Subsection 4.1.3, Mathematica implicitly assumes a homogeneous Neumann condition if no other boundary condition is provided.

In Subsection 4.1.1 we defined that the Neumann boundary condition is applied at the bottom edge where $y = 0$, and that the magnitude of $\boldsymbol{g}$ is 1. The boundary condition is then obtained in Mathematica by setting

```
neumann = NeumannValue[-1,y==0];
```

The first argument above is negative since the load vector $\boldsymbol{g}$ and the outward unit normal point in opposite directions.

The left edge of the domain was held fixed in both x- and y-directions; thus the Dirichlet boundary condition sets $u = v = 0$ at $x = 0$. The appropriate code in Mathematica for this is

```
dirichlet = DirichletCondition[{u[x,y]==0,v[x,y]==0},x==0];
```

Finally, we need to define the operator `op`. It is given by the following somewhat awkward piece of code:

```
op=Inactivate[{Div[{{λ+2μ,0},{0,μ}}.Grad[u[x,y],{x,y}],{x,y}]
               + Div[{{0,λ},{μ,0}}.Grad[v[x,y],{x,y}],{x,y}],
               Div[{{μ,0},{0,λ+2μ}}.Grad[v[x,y],{x,y}],{x,y}]
               + Div[{{0,μ},{λ,0}}.Grad[u[x,y],{x,y}],{x,y}]},
               Div|Grad];
```

Ignoring the `Inactivate`-function for a moment, the rest of the first four lines encode the equations given by (4.4). The equations have been input in the form given by (4.5); the functions `Div` and `Grad` give the divergence and gradient. The Lamé coefficients λ and μ were computed from (2.6) using the values $E = 1000$ and $\nu = 0.3$ which we defined in Section 2.1.

The reason we needed to include the `Inactivate`-function is that it prevents Mathematica from evaluating specific functions too early; in this case, the last row tells Mathematica to leave the `Div`- and `Grad`-functions inactive. If we did not use `Inactivate`, Mathematica would automatically simplify the form given by (4.5) back into that of (4.4), which would again result in the problem of `NDSolve` not understanding how we want the Neumann boundary condition to be applied.

This completes all the definitions that are needed before the actual call to `NDSolveValue` which we gave earlier. Now, we assume that we have the interpolating functions `uif` and `vif`, representing u and v respectively. Then all that remains is to obtain σ_{22} using (4.6). We skip the code since it is quite trivial. The partial derivatives can be obtained with the following code:

```
dxu = Derivative[1,0][uif][x0,y0];
dyv = Derivative[0,1][vif][x0,y0];
```

The arguments of `Derivative` indicate how many times the function should be differentiated with respect to each variable.

Naturally, at this point we can evaluate the stress component σ_{22} at as many points as we want without having to solve the problem again, since all we need to do is evaluate the derivatives at each point and apply (4.6). We should point out that in terms of accuracy, stresses require a denser mesh than displacements do, since solving them involves differentiating $\boldsymbol{u}$; this is the main reason why we constructed a mesh which was as dense as possible.

4.1.5 Interpolation

Up to this point we have focused on solving the entire problem for a single choice of the parameters r and d. Now that we have a solution for this, we can move on to interpolation.

As a quick recap, recall that we are using a sparse grid interpolation scheme which, given some function, aims to use as few function evaluations as possible to accurately interpolate the function over a given range of values. In our model problem, this function essentially does everything we just described in Subsections 4.1.1–4.1.4: given a point in the nominal domain and the variables r and d, the function maps the point to the domain given by r and d, solves the deterministic problem in the domain and returns the stresses at the mapped point. However, complex as this is, from an interpolation point-of-view it makes no difference; we can simply treat it as a regular function in two variables.

In our work, we used interpolation code which was kindly provided by our colleague Vesa Kaarnioja. However, the code is quite complex since it has been optimized for applications where the run time of the interpolation code itself has a significant impact on the overall speed. In our work this is of course not the case, since each function evaluation requires solving an entire FEM problem. We will therefore describe a simplified interpolation code, which is easier to implement but still fast enough for our purposes.

In order to describe an algorithm for sparse grid interpolation, we need to begin with an algorithm for the full grid interpolants $U^{\alpha_1} \otimes \cdots \otimes U^{\alpha_d}$ which appear in (3.16). Fortunately, Klimke has described such an algorithm in detail, so rather than repeat his work here, we refer the reader to pages 29–35 of [6]; the algorithm itself is on page 34.

Using the algorithm for full grid interpolation, (3.16) can then be implemented by simply summing over the terms. Algorithm 1 describes how this can be done. The algorithm was given by Gerstner in [3]; we have corrected a few misprints and modified it to match our notation. The full grid interpolation algorithm is denoted by FullGridInterpolation$[\boldsymbol{x}, \alpha]$, where $\boldsymbol{x}$ is the point which we want to interpolate and α is the multi-index from (3.16). In a real implementation Algorithm 1 would of course also require the interpolation range and function values at interpolation points as input, but since this information is only passed on to the full grid interpolation function, we have skipped such details.

Note that the binomial coefficient $\binom{n}{k}$ should not be computed using the definition

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

as it is prone to catastrophic overflow. If there is no pre-existing function available for computing it, an easy solution is to use the recurrence formula

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}, \qquad \binom{n}{0} = 1.$$

This can and should be implemented using floating point arithmetic, as using integers runs the risk of overflow.

Naturally, Algorithm 1 can also be used to obtain the set of interpolation points $\eta(k, d)$ given by (3.17) with very small modifications. The outer for-loop should be removed since we are only interested in the case where $l = k$. The variable p should be replaced by a set, or any other data structure which does not store duplicate values. Finally, the function fullGridInterp$[\boldsymbol{x}, \alpha]$

Algorithm 1 An algorithm for evaluating the sparse grid polynomial interpolant $A^d_k(f)$ using the form given by (3.16). This algorithm requires two functions: BinomialCoefficient$[n, k]$, which evaluates $\binom{n}{k}$, and FullGridInterpolation$[\boldsymbol{x}, \alpha]$, which evaluates the full grid interpolant $\left(U^{\alpha_1} \otimes \cdots \otimes U^{\alpha_d}\right)(f)(\boldsymbol{x})$. Note that the latter function will require the values of f at the interpolation nodes; this is not considered here, we only show how to evaluate the sum in (3.16).

Input:

$k \in \mathbb{N}$: Order of interpolation.

$d \in \mathbb{N}_+$: Dimension of interpolation.

$\boldsymbol{x}$: Point which we want to interpolate.

Output:

$p = A^k_d(f)(\boldsymbol{x})$: Interpolated value of f at $\boldsymbol{x}$.

```
 1: Let $p = 0$.
 2: **For** $\max(0, k-d+1) \le l \le k$ **do**
 3:     Let $\alpha = (\alpha_1, \dots, \alpha_j) = (0, \dots, 0)$.
 4:     Let $c = (-1)^{k-l} \cdot$ BinomialCoefficient$[d-1, k-l]$.
 5:     **If** $l = 0$ **then**
 6:         Let $p = p + c \cdot$ FullGridInterpolation$[\boldsymbol{x}, \alpha]$.
 7:         **Continue.**
 8:     **End If**
 9:     Let $\widehat{\alpha} = (\widehat{\alpha}_1, \dots, \widehat{\alpha}_j) = (l, \dots, l)$.
10:     Let $q = 1$.
11:     **while** $\alpha_d \le l$ **do**
12:         Let $\alpha_q = \alpha_q + 1$.
13:         **If** $\alpha_q > \widehat{\alpha}_q$ **then**
14:             **If** $q \ne d$ **then**
15:                 Let $\alpha_q = 0$.
16:                 Let $q = q + 1$.
17:             **End If**
18:         **Else**
19:             **For** $1 \le r \le q-1$ **do**
20:                 Let $\widehat{\alpha}_r = \widehat{\alpha}_q - \alpha_q$.
21:             **End For**
22:             Let $\alpha_1 = \widehat{\alpha}_1$.
23:             Let $p = p + c \cdot$ FullGridInterpolation$[\boldsymbol{x}, \alpha]$.
24:             Let $q = 1$.
25:         **End If**
26:     **End while**
27: **End For**
28: **Return** $p$.
```

should be replaced by a function which returns all the interpolation points in the full grid corresponding to the multi-index α.

The entire interpolation scheme can finally be implemented by taking the interpolation points given by the algorithm described in this subsection, solving the corresponding deterministic problems and mappings, and using the results as input for Algorithm 1.

4.2 3D problem

In this section we describe the implementation of the 3D model problem which was very briefly introduced in Section 2.4. The description will be shorter than the previous section, since most parts of the 2D implementation carry over to the three-dimensional case.

For the three-dimensional case, we still used Mathematica for computing the mappings and interpolating. However, Abaqus was used for solving the deterministic problems. Mathematica was still used on the same desktop computer that we described in Section 4.1, but for Abaqus we made use of the Taito supercluster, a computing server managed by CSC.

We will go into greater detail about computational aspects later on; for now, let us start from the beginning by filling in all the details for the model problem which were skipped in Section 2.4.

The three-dimensional domain $\Omega(r, d)$ was obtained by rotating the 2D domain, which we showed in Figure 4.1 of Subsection 4.1.1. In that subsection, we specified the coordinate system such that the bottom left corner was at the origin, with the x-axis pointing to the right and the y-axis upwards. For the three-dimensional case, let the bottom left corner of the 2D domain be located at $(0, 1)$ instead; otherwise, we use the same coordinate system. The three-dimensional object was then obtained by rotating the 2D domain 180 degrees around the y-axis; the resulting object is displayed in Figure 4.4. In the figure, the positive y-direction is to the left and the positive x-direction is upwards.

The object was modelled as a general elastic body, with the material properties of steel, given by $E = 2 \cdot 10^{11}$ and $\nu = 0.3$. The boundary conditions were similar to those of the 2D problem, with one face of the domain fixed and another subjected to an external load. Using Figure 4.4 for reference, the bottom side of the object was fixed using an encastre boundary condition, which prevents both movement and rotation in all coordinate directions. The external load was applied to the curved inner surface of the object, which corresponds to the bottom edge of the two-dimensional domain in Figure 4.1. Since we wanted to keep the boundary conditions analogous to those of the two-dimensional case, we modelled this external load as a uniform pressure on the surface, with its sign chosen such that the resulting force pushes the surface radially outward, away from the y-axis. The pressure was given a magnitude of 10^8.

The next step was to encode the deterministic problem described above in Abaqus. As we have stated earlier, we assume that anyone implementing our

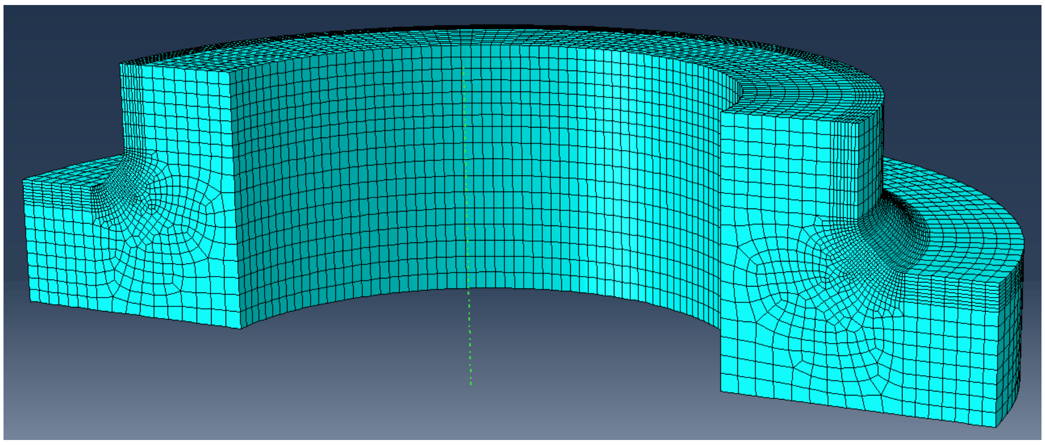

Figure 4.4: The 3D nominal domain and the mesh which was used for it. Observe that the mesh was made significantly denser in the inside corner, as this is where we wanted to measure the stresses.

work has the necessary experience to solve such deterministic problems using some software such as Abaqus, so we will focus on the additional challenges caused by needing to create several models for various values of r and d.

The easiest way to create an object such as the one in Figure 4.4 using Abaqus is to first sketch the 2D domain and then rotate it around an axis. This also made it easy to create a model depending on r and d, since we simply needed to create a sketch which depends on these two values. However, care must be taken to ensure that the sketch is correct for every choice of r and d in the chosen range, which in our case was $r, d \in [0.075, 0.125]$ as in the two-dimensional case. Fortunately Abaqus makes this quite easy, since its sketching tool includes constraints which can be used to, for example, force two line segments to be parallel or two arcs to have the same radius.

The fact that r and d can vary also makes meshing quite difficult. It goes without saying that the accuracy of the interpolation is limited by the accuracy of the FEM solution, so we had to make it as accurate as possible for all possible values of r and d. At the same time, we had to ensure that any single FEM problem could be solved quickly enough. The mesh that we used for the nominal domain can be seen in Figure 4.4, and it was constructed carefully to ensure that the mesh is similar regardless of the values chosen for r and d. The mesh was densest at the inside corner of the L-shape, since this is where we measured the stresses. The total number of elements in the mesh ranged from 58000 to 65000, depending on the values of r and d.

We should point out that the ranges we used for r and d were intentionally quite large; naturally, smaller variation in the parameters will also make mesh design easier. Regardless, we highly recommend checking the mesh quality for several values of the model parameters, in particular the extreme cases where parameters get their largest or smallest values.

We tackled the problem of creating multiple models by first creating a model for the nominal domain, and then writing a script which used that model as a template. In Abaqus, models can be edited or even created using

Python commands. Furthermore, when using the graphical user interface to design models, Abaqus simultaneously writes every action into a Python script, which makes it quite easy to script Abaqus actions without prior experience.

Naturally, creating or editing models using a script imposed a similar problem as meshing: extreme care had to be taken to ensure that the model was correct for any values of r and d. A particular problem that we encountered was that changing r and d could, in our model, change the surfaces where boundary conditions were applied. Such problems can be very rare; this one only happened five times in a set of over 700 models. We solved this problem by simply creating the boundary conditions with the script file instead of having them in the template model, but it serves well to highlight the kinds of problems which may arise. In most cases, these problems will of course stand out: either the calculations for the model fail, or the result clearly sticks out from the rest of the data.

The conformal mappings were implemented in the exact same way as in the two-dimensional case. As we explained at the end of Section 3.5, the mappings for a three-dimensional object can be constructed in two dimensions if the object has the same cross-section throughout. This is clearly the case here, since we have created the object by rotating a two-dimensional domain. Therefore, we used the same conformal mappings as in the two-dimensional case, making sure that the mapped point stays in the same cross-section as the original point; in other words, both points were required to lie in the same plane as the y-axis.

In the interpolation scheme, nothing changes when we add a spatial dimension. The interpolation needs no information about the underlying function, it only cares about the function values at specific points (r, d). However, since we were no longer working with a single software, or even on a single computer, our implementation did change slightly as a result. As we mentioned earlier, we used Abaqus on the Taito supercluster, while Mathematica was run on a desktop computer since the CSC machines did not yet have the latest version of Mathematica which introduced the FEM functionality we needed. Consequently, some parts of the solution process were impossible to automate. Thus, instead of solving the entire problem related to a point (r, d) before moving on to the next one, we instead solved a single step for all the points (r, d) given by the interpolation scheme before moving on to the next step.

Naturally, the first step was to extract a list of points (r, d) from the interpolation scheme. Next, we computed all the conformal mappings in Mathematica and solved all the deterministic problems in Abaqus; these two steps could be done at the same time since they are independent of each other. Finally, we extracted the stresses from the solutions to the deterministic problems at the points determined by the conformal mappings. This data was then used as input for the interpolation scheme.

Chapter 5

Numerical experiments and evaluation

Now that we have described the theory behind our method and how it was implemented, all that remains is to see how the method actually performs. In Section 5.1, we compare the sparse and full grid interpolation schemes which were described in Chapter 3. In Section 5.2, we present the results of our numerical experiments and analyze the accuracy of the interpolation. Finally, in Section 5.3 we discuss how various factors affect the speed and accuracy of our method, and provide the computation times in our model problem as a reference.

5.1 Sparse vs. full grid interpolation

One of the key parts of our method is the Smolyak sparse grid interpolation scheme. It is therefore natural to ask how much better it actually is compared to the naïve full grid interpolation. This is of course difficult to answer in general, since the answer depends greatly on the function which is being interpolated.

From a theoretical point of view, it can be shown that the sparse grid interpolant preserves the accuracy of a full grid interpolant of the same order up to a logarithmic factor. On the other hand, the full grid uses $(2^k + 1)^d$ grid points, which is exponential in d. Meanwhile, in [1], Bungartz and Griebel show that the sparse grid uses at most $\mathcal{O}(2^k k^{d-1})$ interpolation nodes; thus, the sparse grid greatly diminishes the curse of dimensionality which is predominant in full grid interpolation.

As a concrete example of just how many fewer points are needed by the sparse grid, we have computed the number of interpolation points required by both schemes for various values of k and d; the results are shown in Table 5.1. We can see that even with the simple two-dimensional case, the full grid of order 7 uses over 20 times as many points as the sparse grid does, and when we go up to eight dimensions, the ratio increases to over 10^{11}. The number of sparse grid points does seem to grow quite rapidly as well, but we should keep in mind that the underlying univariate quadrature rule uses

Table 5.1: A comparison of the number of nodes required by the full grid and sparse grid interpolation schemes at various values of d and k. The numbers are given by $N_{\text{full}} = (2^k + 1)^d$ and $N_{\text{sparse}} = |\eta(k, d)|$.

	$d = 2$		$d = 4$		$d = 8$	
k	N_{full}	N_{sparse}	N_{full}	N_{sparse}	N_{full}	N_{sparse}
0	1	1	1	1	1	1
1	9	5	81	9	6561	17
2	25	13	625	41	$3.9 \cdot 10^5$	145
3	81	29	6561	137	$4.3 \cdot 10^7$	849
4	289	65	83521	401	$7.0 \cdot 10^9$	3937
5	1089	145	$1.2 \cdot 10^6$	1105	$1.4 \cdot 10^{12}$	15713
6	4225	321	$1.8 \cdot 10^7$	2929	$3.2 \cdot 10^{14}$	56737
7	16641	705	$2.8 \cdot 10^8$	7537	$7.7 \cdot 10^{16}$	$1.9 \cdot 10^5$

$2^k + 1$ interpolation points, so incrementing k by one should always at least double the number of points used.

Naturally, we have to keep in mind that the full grid interpolant of order k does still give better results than the sparse grid version of the same order. Therefore, Table 5.1 does not actually tell the whole truth, but it merely serves to indicate how quickly the full grid approach becomes unfeasible; keep in mind that we need to solve a FEM problem for each grid point, and solving tens of thousands of them can be quite problematic.

All of this makes one thing clear: the sparse grid approach is definitely the way to go when k and d are large. Unfortunately, this is not necessarily the case in our work, since the expense of solving FEM problems makes it preferable to keep k and d as small as possible.

When the theoretical approach fails, the natural solution is to try both methods in practice. Luckily, A. Klimke has already done this in his wonderful PhD thesis [6], the same work which we used as a basis for the interpolation sections of Chapter 3. Interested readers should take a look at his results, which can be found on pages 55–57 of the thesis; the relevant figures are the ones which show the CGL interpolant, which is the same one that we have used in our work for the sparse grid interpolation. We will now briefly describe the results, but we do not include the graphs or specific details about the test functions used.

In his work, Klimke computed the accuracy of the sparse and full grid interpolants as a function of the number of grid points required. Five different test functions were used, and the test was done for dimensions $d = 2$, $d = 5$ and $d = 10$. The results which Klimke obtained clearly show that the sparse grid approach starts pulling ahead as the number of dimensions increases, which is precisely what we would expect. Even in the case of $d = 2$, it provided better results on average.

The full grid approach did prove to work better for two functions in the two-dimensional case and one function in the five-dimensional case, but these functions had relatively sharp peaks which is not the kind of behaviour that we would expect from the functions we are interpolating here. The conclusion is that while there do exist cases where the full grid interpolant may outperform its sparse counterpart, the results provided by [6] clearly favor the sparse grid approach.

5.2 Results

In this section, we examine how well the method worked for our two model problems. For both cases, we computed the interpolating functions of order up to $k = 7$. In order to gauge the accuracy of our method, we first define the maximum error for a function f through

$$e_k(f) := \left\| A_2^k(f) - A_2^7(f) \right\|_\infty, \tag{5.1}$$

where $A_2^k(f)$ is the two-dimensional interpolant of order k for f, and the interpolant of order 7 is used as a reference since the function f is not known explicitly. The norm $\|\cdot\|_\infty$ above gives the maximal absolute value of its argument; in other words, for any point (r, d) the difference in the function values predicted by the interpolants of order k and order 7 is at most $e_k(f)$.

Now, the error estimate which we will use in this section is a normalized version of (5.1), given by

$$e_k^{\mathrm{n}}(f) := \frac{e_k(f)}{e_0(f)} = \frac{\left\| A_2^k(f) - A_2^7(f) \right\|_\infty}{\left\| A_2^0(f) - A_2^7(f) \right\|_\infty}, \tag{5.2}$$

In other words, we normalize the errors such that $e_0^{\mathrm{n}}(f) = 1$. The main reason for this is to make it easier to compare the results of the two- and three-dimensional model problem.

Recall from Section 3.1 that a measurement point $\tilde{\boldsymbol{x}}$ is chosen in the nominal domain $\widetilde{\Omega}$. We experimented with a number of measurement points, but since none of the results particularly stood out, we present all our results using a single measurement point. In the 2D nominal domain of Figure 4.1, the point was located at the middle of the topmost arc, at a depth of 0.001 inside the domain; recall that the width and height of the domain is 1. For the 3D domain, the measurement point was chosen from the cross-section at the center of the domain. Since the cross-section is identical to the 2D domain, the same point was chosen within the cross-section.

For both cases, we computed the interpolating functions of order up to $k = 7$. As can be seen from Table 5.1, this means we solved a total of 705 FEM problems for each case. We measured the stress component σ_{22} in both the two-dimensional and three-dimensional case.

Figure 5.1 shows the function values at the interpolation points with order $k = 7$, and Figure 5.2 shows the interpolating function itself. The result looks quite smooth, which is not entirely unexpected given the simple nature of the model problem. It does however raise the question of whether it is necessary to use an order 7 interpolant.

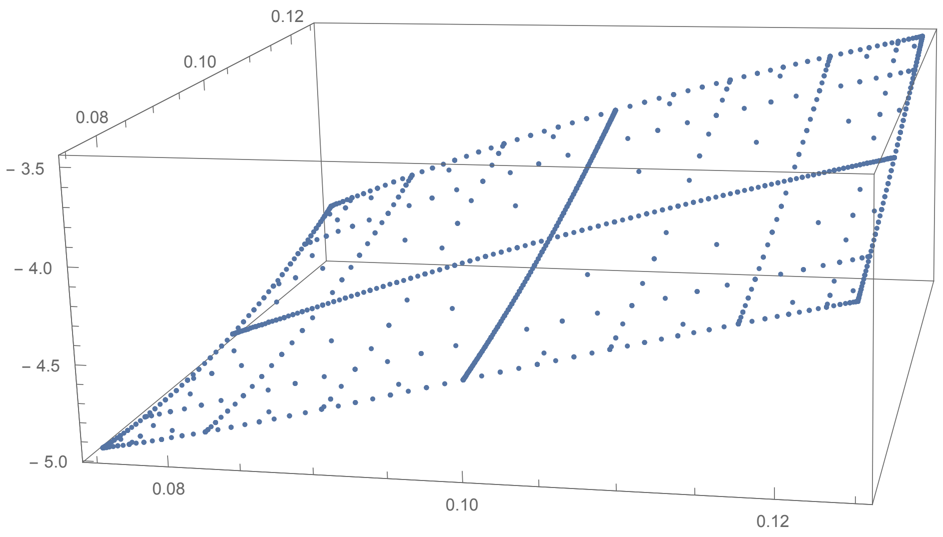

Figure 5.1: Function values at the interpolation points for the 2D model problem, with $k = 7$. The bottom axis corresponds to r, the diagonal axis to d and the vertical one to σ_{22}.

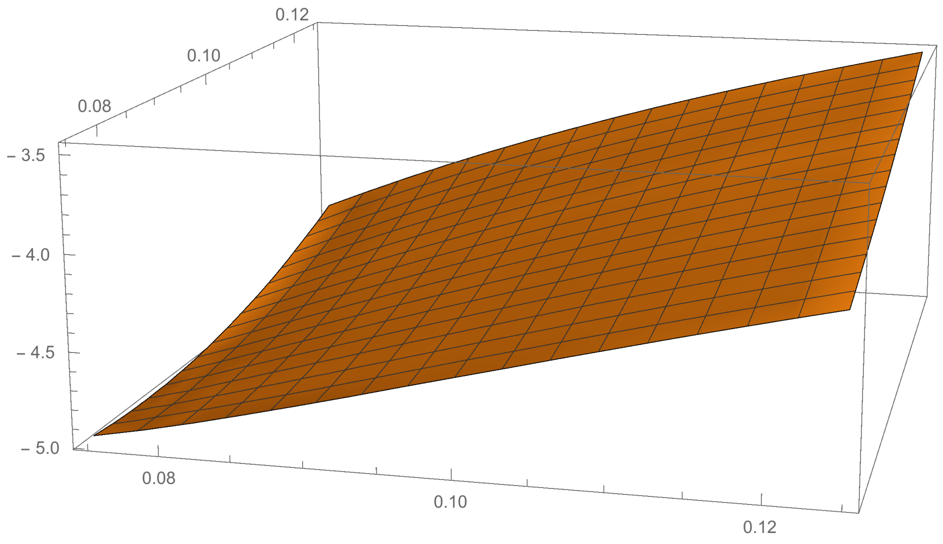

Figure 5.2: Interpolating function of σ_{22} for the 2D model problem, with $k = 7$, using the values and axes shown in Figure 5.1.

As it turns out, for the 2D model problem we would have obtained an interpolating function of sufficient accuracy even with an interpolant of order 2 or 3. Figure 5.3 displays the error of the interpolation as a function of k, using the normalized maximum error e_k^{n} defined in (5.2). The first two errors are quite large as expected, given that they correspond to a constant and linear interpolant. However, the interpolants of order 2–6 all have approximately the same maximum error. The reason for this lies in the inherent inaccuracy of the FEM solution for σ_{22}, which prevents us from getting any better accuracy regardless of the order of interpolation; naturally, the interpolating function cannot be more accurate than the function values which it is interpolating.

The noise in the data is almost impossible to notice in Figure 5.1, but if we first subtract the values predicted by a low-order interpolant, the effect becomes quite clear. In Figure 5.4, we have taken the interpolation points lying on the d-axis, subtracted the values predicted by the interpolant of order 1 at the same points, and then divided the values by $e_0 \approx 0.83$; this scaling was done to make it easier to see how the values might affect the accuracy of the method, as the same scaling was used in the normalized error given by (5.2). As we can see, the noise in the data is quite significant. For example, approximately at $d = 0.106$, there is an instance where two consecutive points differ by over 0.002, even though from the general trend we would expect the difference to be smaller by an order of magnitude. Since every second point in Figure 5.4 is only available for the interpolant of order 7, there is no way that a lower order interpolant could have a normalized error e_k^{n} smaller than 0.001. In this context, the error of 0.005 which we saw in Figure 5.3 is quite acceptable.

The above analysis demonstrates a key factor in our method: the accuracy of the FEM solution plays a large part in determining how accurate our interpolation can be. Naturally, in most applications we may not need to reach higher accuracies, but this effect can also be used as a good indicator for when to stop increasing the order of interpolation k. Of course, it is a good idea to verify that the accuracy is indeed sufficient by taking a random sample of points and comparing the real values at those points to the ones predicted by the interpolation.

Next, let us look at the 3D model problem. As can be seen in Figure 5.5, the result is essentially the same as in the 2D model problem, which is not entirely surprising since the three-dimensional problem is an analogue of the two-dimensional one; however, it does demonstrate that the method extends to three dimensions without any significant problems. Naturally, the three-dimensional analogue of Figure 5.1 also looks like its two-dimensional counterpart, which is why we chose not to include it here.

Figure 5.6 shows that the normalized errors e_k^{n} in the 3D model problem are quite similar to those of the 2D problem; however, the errors plateau at a level of 0.015–0.02, as opposed to the error of 0.005 which we had in the two-dimensional case. Still, this accuracy is certainly good enough considering that we are dealing with FEM problems.

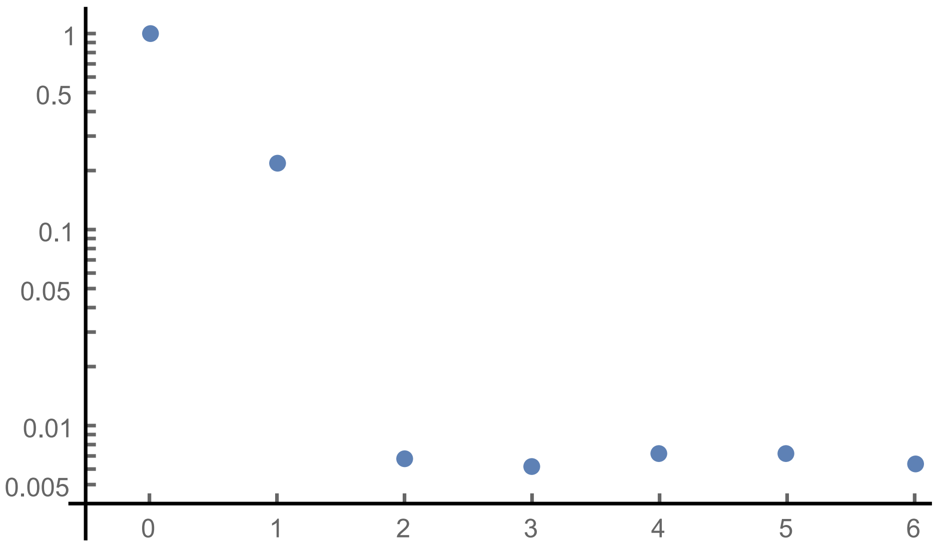

Figure 5.3: Semi-log plot of the normalized errors e_k^n for $k = 0, \dots, 6$ in the 2D model problem.

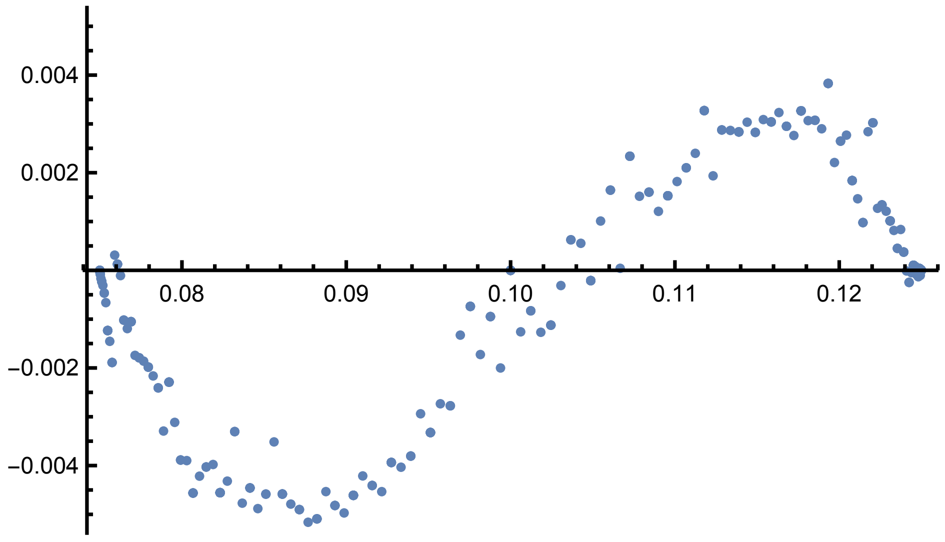

Figure 5.4: Function values at the interpolation points lying on the d-axis in Figure 5.1, after subtracting values predicted by the interpolant of order 1 and scaling by the error term $e_0 \approx 0.82$.

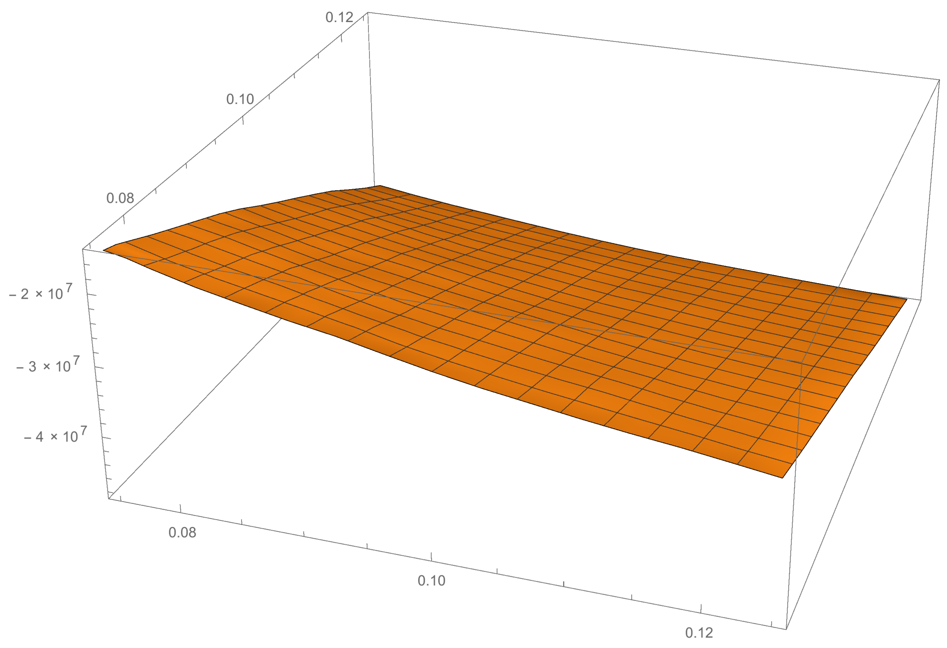

Figure 5.5: Interpolating function for the 3D model problem, with $k = 7$. The bottom axis corresponds to r, the diagonal axis to d and the vertical one to σ_{22}.

Figure 5.7 shows the only real difference which we observed between the results of the 2D and 3D problems. This is the analogue of Figure 5.4; the values have again been scaled by the error term e_0, which in the three-dimensional case had a value of approximately $2.0 \cdot 10^7$. While Figure 5.4 clearly showed how noisy the data was, what we see in Figure 5.7 does not at first glance look like numerical noise. However, there is certainly no physical explanation for why the graph should behave like this, since the domain $\Omega(r, d)$ changes smoothly with respect to r and d. It is therefore highly unlikely that Figure 5.4 represents any real problem with our method; rather, the noise in the data simply presents itself in a different way.

There are a number of things which could cause the behavior which we see in Figure 5.7. Our first thought was that the problem might be in the meshing process, but there was no discernible difference in the meshes corresponding to values of d at either side of any of the peaks in Figure 5.7. The problem may also be related to the way that Abaqus solves the FEM equations or the way it interpolates stress values between mesh nodes, or any number of other reasons. Needless to say, software like Abaqus is mainly designed for solving individual FEM problems rather than an entire collection of them, so surprising things are bound to happen. As we pointed out earlier, any doubts about the accuracy of the interpolation can easily be put to rest by simply comparing the interpolated values against a random collection of FEM problems.

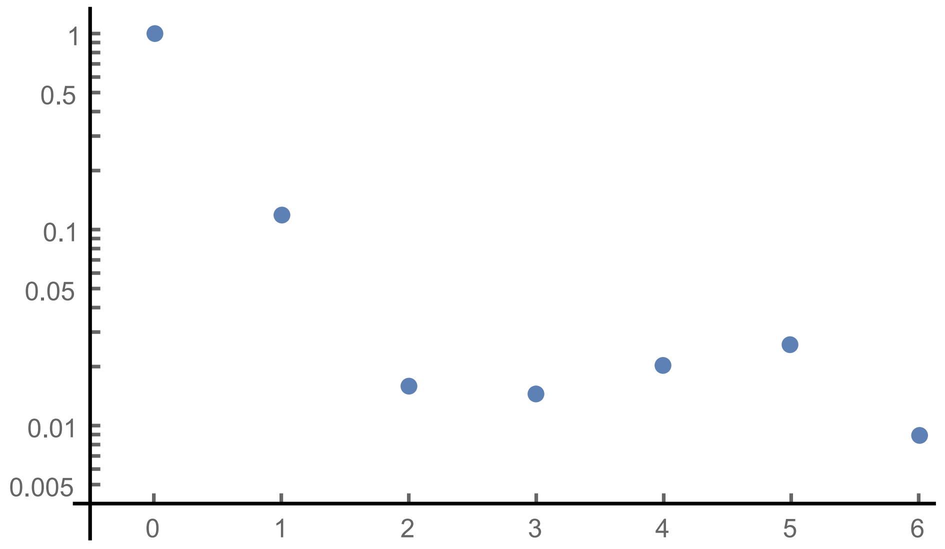

Figure 5.6: Semi-log plot of the normalized errors e_k^{n} for $k = 0, \dots, 6$ in the 3D model problem.

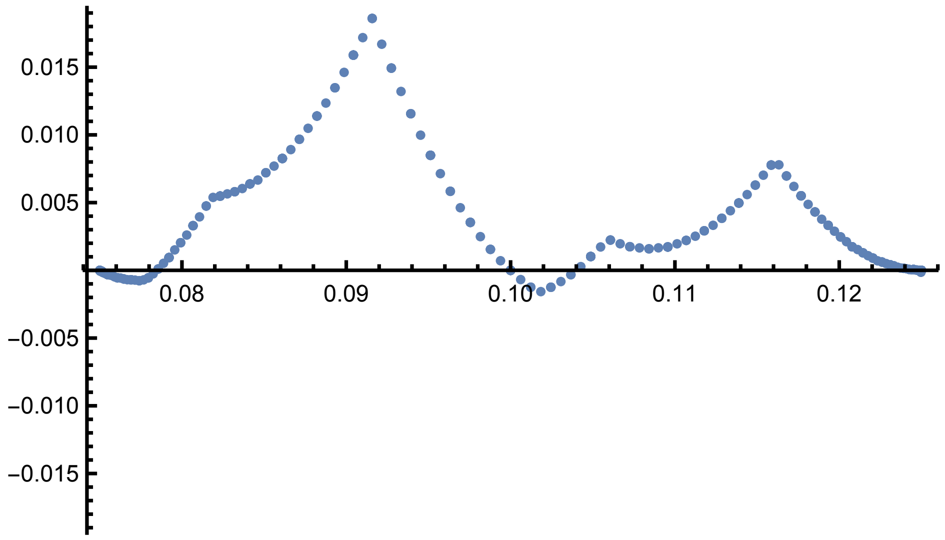

Figure 5.7: Function values at the points lying on the d-axis in the 3D model problem, after subtracting values predicted by the interpolant of order 1 and scaling by the error term $e_0 \approx 2.0 \cdot 10^7$.

5.3 Speed/accuracy tradeoff

Thus far, we have mainly focused on the accuracy of our method. However, speed is just as important, since a high-accuracy method is almost worthless if its running takes weeks. In this section, we list how much CPU and real time was required for solving each part of the model problems, before moving on to discuss the tradeoff between speed and accuracy in more general terms.

In the two-dimensional model problem, using Mathematica 10.0.1 with the desktop computer described in Section 4.1, solving the conformal mapping for a point (r, d) took approximately 6 seconds, and solving the deterministic problem took 7 seconds. These are CPU times, and since we used seven parallel kernels for the computations, both parts took approximately one second to solve. This adds up to approximately 22 minutes for computing all the values required by the order 7 interpolant. The interpolation computations were negligible by comparison: computing the coordinates for all the interpolation nodes took less than a second, and the interpolating function could be evaluated approximately 1500 times per second.

In the three-dimensional case, all other computation times naturally stay the same, but the deterministic problem took significantly longer to solve. Using the mesh described in Section 4.2, a single problem took an average of 5 minutes to solve using four cores; in other words, 20 minutes of CPU time was required per Abaqus model, and a total of 235 hours of CPU time was used to generate the data for the order 7 interpolant. This may of course seem like a huge amount of time, but such is the nature of 3D FEM problems; it should also be noted that the mesh we used was probably denser than truly necessary.

However, since the computations were done on a supercluster, we had potentially thousands of CPU cores at our disposal, and the computations were trivial to parallelize. Unfortunately we had to restrict ourselves to solving only 20 models at a time due to the number of Abaqus licenses available from CSC, but this still meant using 80 cores at the same time, which brought the previously mentioned CPU time down to 3 hours in real time.

All of this goes to show that several factors need to be taken into account when implementing our method. As we saw in Section 5.2, it makes no difference how high the order of the interpolating function is if the data itself is too noisy; on the other hand, if the interpolation uses too many dimensions or if a single deterministic problem takes too long to solve, it is possible that the point where the accuracy stops improving cannot be reached in a reasonable amount of time. Consequently, it is a good idea to begin by determining an approximate upper limit on how many interpolation points may be used.

The upper limit really depends on three factors: how many problems can be solved in parallel, how long it takes on average to solve a single problem, and how much time is available in total. Typically, the number of problems which can be solved in parallel will be limited by the availability of computing cores, memory, or software licenses. The average time it takes to solve a problem is mostly determined by the number of mesh elements and CPU cores used by a single FEM problem. The total time limitation can be in either real or CPU time; for instance, it may be desirable to complete the computations overnight, or there may be a CPU time quota which may not be exceeded.

When an upper limit has been determined, it is quite easy to check the highest order of interpolation k which can be reached for a given dimension d. In most cases, the goal is to reach $k = d$ at the very least, since this is the lowest order k which includes the corners of the interpolation region. The reason why lower-order interpolants do not use the corners of the region is that, as long as $k < d$, all d-dimensional multi-indices α satisfying $|\alpha| = k$ contain at least one zero. As can be seen from the definition of the collection of interpolation points $\eta(k, d)$ in (3.17), this means that the terms $X^{\alpha_1} \times \cdots \times X^{\alpha_d}$ necessarily include the set X^0, which only contains the midpoint of its corresponding interval; however, the corners of the entire interpolation range are only obtained if all sets X^{α_i} include both ends of their corresponding intervals, which only happens when $\alpha_i \geq 1$ for all $1 \leq i \leq d$.

Chapter 6

Conclusions

In this work, we have demonstrated an efficient way of using interpolation to solve FEM problems with stochastic domains. The accuracy of our method was examined in Section 5.2, to the conclusion that the interpolation matched the accuracy of the FEM problem already in the case of an interpolant of second order; that is, with a quadratic interpolant. While this may in part be due to the simplicity of our model problem, it still shows that even a low-order interpolant can be sufficient.

In Section 5.1, we explained why sparse grid interpolation should be preferred over its full grid counterpart, and referred to past experiments by Klimke in [6] to motivate this statement. While it is true that the full grid interpolant may perform better for certain low-dimensional functions, the average case strongly favored the sparse grid interpolation scheme. Of course, the sparse grid interpolation is somewhat harder to implement, but we feel that the pros far outweigh the cons even in low dimensions.

We did not specifically analyse how good conformal mappings are at controlling the consistency of our method compared to other options, as this was beyond the scope of our work. There may well be better options, but the conformal mapping has the significant advantage of working for a large category of problems, as its only requirement is that the mapping needs to be constructed in two dimensions.

In some applications, our method may be improved by using a dimension-adaptive sparse grid construction such as the one described in Section 3.4 of [6]. We did not consider this in our work, since the method of interpolation is essentially disconnected from the rest of our method, and the dimension-adaptive routine makes it impossible to compute the set of interpolation nodes before-hand. However, in higher-dimensional problems, a dimension-adaptive approach could yield significant improvements.

References

[1] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 5 2004.

[2] R. Dautray and J. Lions. *Mathematical Analysis and Numerical Methods for Science and Technology: Volume 2 Functional and Variational Methods.* Springer Berlin, 1988.

[3] T. Gerstner. Sparse grid quadrature methods for computational finance. *Habilitation, University of Bonn*, 2007.

[4] D. Gilbarg and N. Trudinger. *Elliptic Partial Differential Equations of Second Order*, volume 224 of *Grundlehren der mathematischen Wissenschaften.* Springer Berlin, 1977.

[5] H. Hakula, T. Quach, and A. Rasila. Conjugate function method for numerical conformal mappings. *Journal of Computational and Applied Mathematics*, 2011.

[6] W. A. Klimke. *Uncertainty Modeling using Fuzzy Arithmetic and Sparse Grids.* PhD thesis, Universität Stuttgart, 2006.

[7] O. P. Le Maître and O. M. Knio. *Spectral Methods for Uncertainty Quantification: With Applications to Computational Fluid Dynamics.* Scientific Computation. Springer, Dordrecht, 2010.

[8] C. Schenk and G. Schuëller. *Uncertainty Assessment of Large Finite Element Systems.* Lecture Notes in Applied and Computational Mechanics. Springer-Verlag Berlin Heidelberg, 2005.

[9] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics, Doklady*, 4:240–243, 1963.

[10] G. Wasilkowski and H. Woźniakowski. Explicit cost bounds of algorithms for multivariate tensor product problems. *Journal of Complexity*, 11(1):1–56, 1995.

[11] D. Xiu. *Numerical methods for stochastic computations: a spectral method approach.* Princeton University Press, 2010.