

ABSTRACT

Title of dissertation: FAST ALGORITHMS FOR THE SOLUTION
OF STOCHASTIC PARTIAL
DIFFERENTIAL EQUATIONS

Christopher W. Miller, Doctor of Philosophy, 2012

Dissertation directed by: Professor Howard Elman
Department of Computer Science
Institute for Advanced Computer Studies

We explore the performance of several algorithms for the solution of stochastic partial differential equations including the stochastic Galerkin method and the stochastic sparse grid collocation method. We also introduce a new method called the adaptive kernel density estimation (KDE) collocation method, which addresses some of the deficiencies present in other stochastic PDE solution methods. This method combines an adaptive sparse grid collocation method with KDE to optimally allocate stochastic degrees of freedom.

Several components of this method can be computationally expensive, such as automatic bandwidth selection for the kernel density estimate, evaluation of the kernel density estimate, and computation of the coefficients of the approximate solution. Fortunately all of these operations are easily parallelizable. We present an implementation of adaptive KDE collocation that makes use of NVIDIA's complete unified device architecture (CUDA) to perform the computations in parallel on graphics processing units (GPUs).

FAST ALGORITHMS FOR THE SOLUTION OF
STOCHASTIC PARTIAL DIFFERENTIAL
EQUATIONS

by

Christopher William Miller

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Howard Elman, Chair/Advisor
Professor Ramani Duraiswami
Professor Ricardo Nochetto
Professor Tobias von Petersdorff
Professor Alan Sussman, Dean's Representative

© Copyright by
Christopher William Miller
2012

Acknowledgments

A great many people have contributed to this research, and while thanking them in a manner commensurate with their contribution is an impossible task, I would like to show my gratitude to a few of the people who have made this work possible.

First and foremost I would like to thank my thesis advisor, Howard Elman, for his careful and patient guidance during this earliest phase of my research career. I am grateful for his willingness to devote large portions of his time to my education, and for allowing me a great deal of freedom to ‘wander in the wilderness’ while my research was developing. I am also immensely grateful for his financial support during my graduate career. I would also like to thank him for introducing me to many of his interesting colleagues. In particular I would like to thank Raymond Tuminaro and Eric Phipps both of Sandia National Laboratories, for providing me with several opportunities to work with Sandia, proofreading papers, and making numerous other contributions to this research.

I would like to thank Tobias von Petersdorff, Ricardo Nochetto, Ramani Duraiswami, and Alan Sussman for taking the time to serve on my committee.

I also want to thank the Center for Scientific Computation and Mathematical Modeling at Maryland for granting me access to GPU workstations. In particular I want to thank Jeff Henrikson for quickly addressing any IT problems that I encountered.

I am also indebted to my friends in the Mathematics and Applied Mathematics

departments, and at Sandia National Laboratories, with whom I have shared many fruitful mathematical discussions which aided our academic progress, and many non-mathematical discussions which maintained our sanity.

I owe my family for their constant and unbounded love and support, and for instilling in me the determination, work ethic, and curiosity required to undertake this task.

Lastly I am grateful to my wife Elizabeth, whose support with the sometimes curious demands of graduate school has been immeasurable, and whose love adds meaning to every day.

Table of Contents

List of Tables	vi
List of Figures	vii
1 Introduction	1
2 Description of the Problem and Survey of Existing Approaches	13
2.1 Derivation of the Karhunen-Loève expansion	14
2.1.1 Optimality of the Karhunen-Loève Expansion	18
2.2 Regularity of the Solution of the Truncated Diffusion Equation	22
2.3 Derivation of the Stochastic Galerkin Method	28
2.3.1 Error Analysis of the Stochastic Galerkin Method	31
2.4 Sparse Grid Collocation	33
2.4.1 Error Analysis of the Sparse Grid Collocation Method	35
3 Comparison of the Stochastic Finite Element Method and the Stochastic Collocation Method	37
3.1 Stochastic Galerkin Method With Finite Differences	39
3.2 Sparse Grid Collocation	45
3.3 Modeling Computational Costs	48
3.4 Experimental Results and Model Validation	53
3.4.1 Behavior of the Preconditioned Conjugate Gradient Algorithm	53
3.4.2 Computational Cost Comparison	59
3.5 Conclusion	62
4 The Adaptive KDE Collocation Method	68
4.1 Problem Statement	69
4.2 The Adaptive Collocation Method	73
4.3 Kernel Density Estimation	79
4.4 Adaptive Collocation With KDE Driven Grid Refinement	83
4.4.1 Error analysis of adaptive KDE collocation	84
4.4.2 Estimation of Solution Statistics	86
4.5 Numerical Experiments	90
4.5.1 Interpolation of a Highly Oscillatory Function	90
4.5.2 Two-parameter stochastic diffusion equation	96
4.5.3 Function with steep gradients and non-independently distributed random parameters	102
4.5.4 High-dimensional stochastic diffusion	107
4.6 Conclusions	109

5	Performance Analysis of the Adaptive KDE Collocation Method Using CUDA	111
5.1	Summary of Kernel Density Estimation	113
5.2	Summary of Adaptive KDE Collocation	115
5.3	Brief Description of the CUDA Architecture	119
5.4	Implementation of Kernel Density Estimation in CUDA C	126
5.4.1	The CUDA KDE kernel	126
5.4.2	The KernelDensityEstimator class	128
5.5	Implementation of the adaptive KDE collocation method in CUDA C	131
5.5.1	The CUDA adaptive collocation kernel	132
5.5.2	Implementation of the adaptive KDE collocation host code . .	134
5.6	Benchmarks	138
5.6.1	MLCV bandwidth selection benchmarks	139
5.6.2	Kernel density estimation benchmarks	140
5.6.3	Adaptive KDE collocation benchmarks	143
5.7	Conclusions	148
6	Summary and Conclusions	150
A	Source code listings for implementaion of adaptive KDE collocation in CUDA	153
A.1	CUDA kernel density estimation code	153
A.2	Cuda adaptive collocation code	174
	Bibliography	190

List of Tables

3.1	Degrees of freedom for various methods	50
3.2	Mean error in the discrete l_∞ -norm for the stochastic collocation and stochastic Galerkin methods.	56
3.3	Iterations for the stochastic collocation (left) and stochastic Galerkin methods (right)	57
3.4	Approximate values of τ for model problems	60
3.5	Solution (preconditioning) time in seconds for second model problem	66
4.1	Monte Carlo error bound, $ \frac{1}{N} \sum_{i=1}^N u(\boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(\boldsymbol{\xi}^{(i)}) $, and number of collocation points (in parentheses)	106
4.2	Monte Carlo error (left) and $\ \frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\ _{l^2(D)}$, 4 parameter problem	110
4.3	Monte Carlo error (left) and $\ \frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\ _{l^2(D)}$, 10 parameter problem	110
4.4	Monte Carlo error (left) and $\ \frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\ _{l^2(D)}$, 20 parameter problem	110
5.1	Time (in seconds) required to compute the optimal bandwidth using MLCV	140
5.2	Time (in seconds) required to compute the KDE for a 5-dimensional parameter space	143
5.3	Time (in seconds) required to compute the KDE for a 10-dimensional parameter space	143
5.4	Time (in seconds) required to compute the KDE for a 15-dimensional parameter space	143
5.5	Time (in seconds) required to compute the KDE for a 20-dimensional parameter space	144
5.6	Time (in seconds) required to compute the interpolant using adaptive KDE collocation	146
5.7	Collocation points required to compute the interpolant using adaptive KDE collocation	146
5.8	Time (in seconds) required to compute the interpolant using adaptive KDE collocation with varying refinement criterion τ	146

List of Figures

3.1	Block structure of stochastic Galerkin stiffness matrix for $M = 4$ and $p = 1, 2, 3, 4$	43
3.2	Errors vs stochastic DOF for $M = 4$. Uniform random variables (top), truncated Gaussian random variables (bottom)	63
3.3	Solution time vs. error for $M = 3, 4, 5$. Uniform random variables. . .	64
3.4	Solution time vs. error for $M = 3, 4, 5$. Truncated Gaussian random variables.	65
4.1	The hierarchical basis functions for $i = 1, 2, 3$	75
4.2	Under-smoothed kernel density estimate (left) and over-smoothed (right).	80
4.3	$u(\xi) = \xi \sin(1/\xi)$	92
4.4	$\ (u(\xi) - \mathcal{A}_k(u)(\xi))\rho(\xi)\ _\infty$ versus the number of collocation points . .	93
4.5	$u(\xi)$ and the collocation points used in constructing approximate solution	94
4.6	The tolerance τ vs the number of collocation points	95
4.7	Collocation points for various values of the error tolerance τ	99
4.8	Kernel density estimates for varying numbers of samples.	100
4.9	$\ (u(\mathbf{x}, \boldsymbol{\xi}) - \mathcal{A}(u)(\mathbf{x}, \boldsymbol{\xi}))\rho(\boldsymbol{\xi})\ _{l^2(D) \times l^\infty(\Gamma)}$ versus the number of collocation points	101
4.10	Function with two line singularities in parameter space	103
4.11	Samples (top) and kernel density estimate (bottom) for the distribution of two dependent random variables	104
4.12	Collocation points for refinement criterion $\tau = 1 \times 10^{-3}$ and $\tau = 1 \times 10^{-4}$	105
5.1	Floating-point operations per second and memory bandwidth for various CPU and GPU architectures [37]	112
5.2	Organization of CUDA thread blocks and CUDA threads [37]	121
5.3	CUDA memory hierarchy [37]	125
5.4	Time (in seconds) required to compute the optimal bandwidth using MLCV	141
5.5	Time (in seconds) required to compute the kernel density estimate for varying dimension and number of targets	142
5.6	Time required to construct the hierarchical interpolant vs the number of collocation points	147
5.7	Time required to construct the hierarchical interpolant vs the refinement tolerance τ	148

Chapter 1

Introduction

Mathematical models of physical systems arising in science and engineering are often specified in terms of coefficient functions and a set of parameters. These coefficients and parameters represent physical quantities (e.g. viscosity in fluid flow problems) which determine the behavior of the system. The coefficient functions may also be expanded in terms of a finite set of parameters, such as when the coefficient functions are decomposed using principal components analysis or some other spectral representation method. In many cases the solution to the model is very sensitive with respect to perturbations of these parameters. Traditional approaches of evaluating model behavior assign fixed values to the parameters and coefficients even though their true values are often unknown. This uncertainty in the specification of the model is due to the fact that the model inputs are either inherently unknowable (e.g. the velocity of a single gas molecule in molecular gas dynamics), or because insufficient measurement data is available (e.g. diffusion coefficients in a groundwater flow problem).

Rather than examining the model behavior for a single parameter value or range of parameter values, one may choose to express the uncertainty in the model inputs by treating the unknown parameters as random variables and the unknown coefficient functions as random fields, functions that depend on both the spatial

location and on the value of a random event. The uncertainty in the model inputs then produces uncertainty in the model output, which can then also be described as a random field. One of the central goals of uncertainty quantification is to derive algorithms that compute statistical descriptions of the model output when the model inputs are posed with uncertainty. From these statistical descriptions of the output it is possible to approximate the mean, variance, and higher order moments of the solution, as well as probability distributions associated with the solution.

The most well-known method for approximating the mean of an unknown random quantity is the Monte Carlo method [33]. The Monte Carlo method approximates the expected value of the solution by evaluating the model at a finite number of samples, independently drawn from the distribution of the input operator, and then computing the sample mean of the output. The Monte Carlo method is known to be very robust in that convergence in mean square error is guaranteed if the random quantity has finite variance. Furthermore, the convergence rate is independent of the dimension of the parameter space. However, the mean square error is proportional to the inverse square root of the number of samples. Thus, halving the error requires quadruple the number of samples. Since the model must be evaluated at each new sample, this approach can be prohibitively expensive if high accuracy is required or if model evaluations are computationally expensive. It should also be noted that the convergence is probabilistic and that any finite sample may result in a poor approximation to the true expected value particularly when the variance is large.

Recently, many new methods have been developed to efficiently find the so-

lution of stochastic partial differential equations (SPDEs), partial differential equations (PDEs) where either the coefficients, source terms, or boundary conditions are posed with uncertainty. The coefficients, source terms, and boundary conditions specifying a SPDE can all be expressed as random fields. For a single realization of these random fields the SPDE is a deterministic PDE. A solution to a SPDE is defined to be a random field that satisfies the associated deterministic PDE for almost every event. Once the solution to the SPDE is known, solution statistics can be obtained without evaluating the model at a large number of sample points as is required by the Monte Carlo method.

Computing the solution to a SPDE involves the following steps. First, in order to work numerically with the SPDE, the coefficients, source terms, and boundary conditions must be expressed in terms of a finite number of random variables, rather than as members of an abstract probability space. Second, both the spatial and stochastic portion of this reduced model must be discretized. Third, the approximate solution to the model must be computed as a function of both the spatial location and the random parameters. Lastly, any desired quantities, such as moments or probability distributions associated with the solution, need to be computed from the approximate solution.

Procedures for modeling random fields in terms of a finite number of random variables is an active area of research. Often it is assumed that the mean and covariance function of the random field is known. In this case the random fields are often expressed as a truncated Karhunen-Loève (KL) expansion [29]. This transforms the random field into a function of finitely many parameters. While the KL expansion

provides a functional form for a random field in terms of a finite number of parameters, the distributions of these parameters depend on higher order statistics and need to be specified in some way. In certain cases probability distributions of each of the random variables may be chosen based on a priori knowledge of the problem. However, this only specifies the marginal densities of the random variables. While the random variables appearing in a KL expansion are known to be uncorrelated they are not necessarily independent. Thus, it is not possible to infer the joint density function from assumptions about the marginal densities. Another possibility is that information about the distribution of the random parameters is available only from a finite number of independent experimental observations and that we do not have access to an explicit form of either the joint density or the marginal densities.

The way in which the model inputs are represented, the manner in which the densities of the random variables are expressed, and the post-processing of the solution all depend on the assumptions that are made to parameterize the model. For example, assuming that the joint density of the random parameters has a specific form is a much stronger assumption than assuming that one only has access to a finite amount of experimental data. In order for the algorithms discussed below to be used efficiently it is important to take full advantage of whatever is known about the model. In addition, solution techniques need to be flexible enough to handle a wide range of assumptions regarding the description of the random inputs.

The spatial discretization of the SPDE is typically accomplished using conventional methods for the numerical solution of deterministic PDEs, such as finite differences, finite elements, or finite volume methods. (The mathematical theory is

the most well developed in the case of finite elements.) Time-stepping may also be required if the problem is time dependent; in this thesis we focus on steady-state problems.

Many methods have been developed for the discretization of the stochastic portion of the problem. These methods are divided roughly into sampling and non-sampling methods. Sampling methods work by evaluating the model at a finite number of values of the random parameters. The solutions of the model at each of these points are then used to construct an approximation to the solution at different values of the random parameter. Methods of this type include the stochastic collocation method [2], the stochastic sparse grid collocation method [35, 50], the anisotropic stochastic collocation method [34], and the adaptive sparse grid collocation method [30]. These methods all compute an approximation to the solution to the SPED at every value of the parameter, rather than only computing the solution of the SPDE at a finite number of sample points as in the Monte Carlo method. This functional form for the solution to the SPDE can be postprocessed to compute approximations to the statistics of the solution as will be discussed later.

Sampling methods are useful because they require little modification to existing PDE codes. Each sample point gives rise to a separate deterministic problem that can be handled by existing software. Sampling methods are also trivially parallelizable. Since each sample point can be treated as a separate deterministic problem, the SPDE solution at each sample point can be computed by a separate instance of a deterministic PDE code. The main disadvantage of these sampling methods is that often the associated deterministic partial differential equation must be solved

at a very large number of sample points to obtain an acceptable level of accuracy. This is particularly true as the dimension of the parameter space grows. Reducing the number of PDE evaluations required has been a central motivating factor in the development of alternative methods. Model reduction techniques have also been used to reduce the size of the parameter space in order to reduce the number of PDE evaluations required by these sampling methods [31].

The primary non-sampling method is the stochastic finite element method, which is also often referred to as the stochastic Galerkin method [3, 9, 19]. This method is an extension of the standard finite element method in the sense that the method looks for an approximation to the solution of the SPDE that lies in a finite-dimensional subspace of some appropriate Hilbert space. Typically, the Hilbert space in which the problem is posed is the direct product of a Hilbert space of functions defined on the spatial domain with the space of functions of the random variables that are square integrable with respect to the probability measure. A standard finite element discretization is performed on the spatial domain. The finite-dimensional subspace of the stochastic function space is often taken to be a space of fixed degree multivariate polynomials in the random variables. The approximate solution is defined by a Galerkin orthogonality condition to lie in the direct product of the finite-dimensional space spanned by the spatial finite element basis with this space of multivariate polynomials.

Computing the coefficients of this approximation involves solving a single very large linear or non-linear system (in this thesis we will examine only the linear case). The size of this system is the product of the dimension of the spatial finite element

space and the dimension of the stochastic finite element space. Fortunately, in many cases there exist bases that are orthogonal with respect to the inner product induced by the probability measure. Therefore, while large, the stochastic Galerkin stiffness matrix derived from imposing the Galerkin orthogonality condition on this basis is very sparse and contains a great deal of structure. The global stiffness matrix can be assembled as a block matrix where each non-zero block has the same sparsity pattern as a stiffness matrix associated with the finite element discretization of a deterministic PDE. Additionally, most of the blocks are the zero matrix since the basis functions that span the stochastic finite element space are orthogonal. By taking advantage of the structure of this matrix it is possible to develop efficient solvers using iterative methods built from Krylov subspace methods and multigrid methods.

While it may seem advantageous to work with many smaller linear systems by using a sampling method, rather than one large system, the total number of stochastic degrees of freedom associated with the solution to a stochastic finite element problem can be much smaller than the number of samples required by a sampling method with a similar order of approximation. While sampling methods are without a doubt easier to implement than non-sampling methods, it is an open question which of the two requires more computational effort to obtain a solution of a prescribed accuracy.

Once the approximate solution to the SPDE is computed, one may wish to compute the moments of the solution. Often this is accomplished by integrating the approximate solution with respect to the joint density of the random variables

using numerical quadrature. To facilitate this, it is often assumed that the random variables are independent, which allows for the construction of an efficient multi-dimensional quadrature rule to evaluate these integrals. However, the assumption of independence is very strong and may not be valid in practice. Also it may be unreasonable to assume that we have access to either the joint density of the random variables or even the marginal densities of each random variable. As alluded to above, it may be the case that we only have access to the distribution of the random variables through a finite amount of experimental data.

If the joint density of the random variables is not available, then the Monte Carlo method can be combined with an approximate solution to the SPDE obtained by either a sampling or a non-sampling method. As in the Monte Carlo method, a large number of samples from the random parameter space is generated. Then, rather than solving a deterministic PDE at each of the sample points, the approximate solution is evaluated at each of the sample points and the sample mean of the approximate solution is computed. If the approximate solution is a good approximation to the true solution at each sample point, then the sample mean of the approximate solution will be a good approximation to the sample mean of the SPDE. We call this method of performing the Monte Carlo method on an approximate solution the surrogate-based Monte Carlo method.

As discussed previously, a chief limitation of the Monte Carlo method is the fact that a deterministic PDE must be solved at each sample from the parameter space. This can be very expensive if a fine discretization is used. The approximate solution to a SPDE computed by any of the methods discussed above is a piecewise

polynomial with respect to the random variables. Therefore, once the approximate solution is constructed, performing the surrogate Monte Carlo method is very efficient because evaluating a piecewise polynomial at a collection of sample points is typically much less expensive than solving a PDE at each sample point.

Replacing the standard Monte-Carlo estimate with the surrogate Monte Carlo estimate introduces bias into the estimation of the mean. This bias is due to the fact that the surrogate-based method is computing the sample mean of an approximation to the SPDE solution and not the sample mean of the SPDE solution itself. The bias is directly proportional to the approximation error resulting from the stochastic discretization of the problem. However, if the bias is small and computing the approximation is less expensive than evaluating the model at each sample point, then the surrogate-based method can be substantially more efficient than standard Monte Carlo.

The efficiency of surrogate-based Monte Carlo depends on minimizing the bias associated with the approximation error. The bias only depends on the approximation error at the sample data points, and not on the error on the entire stochastic domain. Thus, when computing an approximate solution to a SPDE we should attempt to minimize the error at the sample points. However, most of the research into the numerical solution of SPDEs has focused on discretization methods that are global with respect to the set of stochastic parameters [2, 3, 19]. This is due to the fact that in many cases the solution to the SPDE can be shown to be analytic with respect to each of the random parameters. This makes the use of approximation spaces consisting of globally defined polynomials appropriate. However, these meth-

ods can converge slowly or fail to converge if the solution contains steep gradients or discontinuities.

Interest in approximating the solution to SPDEs containing steep gradients or discontinuities led to the development of the adaptive sparse grid collocation method [30]. This method uses a hierarchical basis of locally supported linear basis functions to approximate the solution to a SPDE. The basis can be refined locally to achieve higher resolution in areas of the domain where the solution exhibits discontinuities or steep gradients. In addition, the coefficients of the approximation serve as a local error indicator that is used to guide the refinement procedure. A modified local refinement procedure is used in this thesis to ensure that the approximation error is small at the sample points.

The deficiency with the adaptive sparse grid collocation method that led to this modification is that the error indicator used to drive the refinement of the adaptive collocation grid only estimates the local interpolation error and does not take into consideration the statistics of the random parameters. This is an issue because the approximation error at a point in the stochastic domain is of little consequence if the probability density function is small in the neighborhood of that point. For example, if the solution to a SPDE is discontinuous in some region of the stochastic domain but the probability density function inside of that region is small, then it is a waste of computational effort to attempt to resolve the discontinuity since it has no effect on the statistics of the SPDE solution. A natural solution to this problem, proposed in this thesis, is to weight the refinement criterion proposed in [30] by the probability density of the random inputs.

Since we only have access to a finite set of samples of the random parameters, it is also necessary to construct an approximation to the joint density function. Density estimation techniques are divided into two classes: parametric estimators and non-parametric estimators. Parametric estimators make certain assumptions about the shape of the underlying distribution (i.e. assuming the distribution is Gaussian with known covariance). For this reason they are not well suited to this study since we are only assuming that we have access to a finite number of observations and make no assumptions about the underlying distribution. Non-parametric estimators make no assumptions on the shape of the underlying distribution and depend only on the sample data set.

In this thesis we will estimate the unknown probability density using kernel density estimation [44]. Kernel density estimation is a non-parametric technique that can be considered as a generalization of the histogram estimator. The estimate itself is the scaled sum of unit-normed “bump” functions of a prescribed width centered at each of the data points. The width of the bump functions is referred to in the literature as the bandwidth of the estimator. Finding the bandwidth that gives the optimal estimate is an open problem, particularly in high-dimensional parameter spaces. Here we will use a technique called maximum-likelihood cross-validation (MLCV) to estimate the optimal bandwidth. The combination of the adaptive collocation method with kernel density estimation will be referred to as adaptive KDE collocation.

Cost effective computation is a consideration in all of these algorithms. In addition to exploring the behavior of many of these algorithms we will present

implementations written for high performance computing environments. Implementations of the stochastic finite element method and the stochastic sparse grid collocation method were built using the *Trilinos* software package [24] developed at Sandia National Laboratories. This software is designed to work on many-core supercomputing clusters using the widely available Message Passing Interface (MPI) [45]. We also present an implementation of the adaptive sparse grid collocation method with kernel density estimation that takes advantage of NVIDIA’s Complete Unified Device Architecture (CUDA) [37]. CUDA presents a set of extensions to the C/C++ programming languages that allow for general purpose computation on graphics processing units (GPU). GPUs are very well suited for handling tasks that are ‘data parallel’, meaning that the same instructions can be executed in parallel on many data elements. Collocation methods and kernel density estimation are both well suited to this type of programming model.

The remainder of this thesis proceeds as follows. Chapter 2 gives a description of the class of problems we consider in the thesis and presents an overview of the current state of solution methods. Chapter 3 presents the results of an experimental study comparing the stochastic finite element method and the stochastic sparse grid collocation method, applied to the linear stochastic diffusion equation. Chapter 4 presents the adaptive KDE collocation method as well as the results of numerical experiments with this method and the surrogate-based Monte Carlo method. Chapter 5 presents an implementation of the adaptive KDE collocation method using the CUDA architecture. Chapter 6 presents a summary of the results and draws some conclusions.

Chapter 2

Description of the Problem and Survey of Existing Approaches

Let (Ω, Σ, P) be a complete probability space with event space Ω , σ -algebra $\Sigma \subset 2^\Omega$, and probability measure $P : \Sigma \rightarrow [0, 1]$. Let $D \subset \mathbb{R}^d$ be a d -dimensional bounded domain with smooth boundary ∂D ; for most applications $d = 1, 2$, or 3 . A stochastic partial differential equation is an equation of the form

$$\begin{aligned}\mathcal{L}(\mathbf{x}, \omega; u) &= f(\mathbf{x}, \omega), & \forall \mathbf{x} \in D, \omega \in \Omega, \\ \mathcal{B}(\mathbf{x}, \omega; u) &= g(\mathbf{x}, \omega), & \forall \mathbf{x} \in \partial D, \omega \in \Omega,\end{aligned}\tag{2.1}$$

where \mathcal{L} is a differential operator, f is the source function, \mathcal{B} is a boundary operator and g is the boundary value function. In the most general case, all of the coefficients, source terms, and boundary terms may be random fields that depend both on the spatial location \mathbf{x} and on the value of ω from the event space. As a consequence of the Doob-Dynkin lemma it follows that the solution u is also a random field [40].

Throughout this thesis we will focus on the stochastic linear diffusion equation with Dirichlet boundary condition given by

$$\begin{aligned}-\nabla \cdot [a(\mathbf{x}, \omega) \nabla u(\mathbf{x}, \omega)] &= f(\mathbf{x}, \omega), & \forall \mathbf{x} \in D, \omega \in \Omega, \\ u(\mathbf{x}, \omega) &= 0, & \forall \mathbf{x} \in \partial D, \omega \in \Omega.\end{aligned}\tag{2.2}$$

In order to treat the equation (2.2) numerically the problem must first be reformulated in terms of a finite number of real valued random variables. Two approaches for accomplishing this are the Karhunen-Loève expansion [29] and the polynomial chaos expansion [49]. The Karhunen-Loève expansion has several desirable qualities which justify its use in this application. The following section presents a derivation of the Karhunen-Loève expansion and outlines these properties. The discussion in the following section mostly follows a similar discussion in [19].

2.1 Derivation of the Karhunen-Loève expansion

Let the mean of the diffusion coefficient a be denoted by $a_0(\mathbf{x}) = \mathbb{E}[a(\mathbf{x}, \cdot)] = \int_{\Omega} a(\mathbf{x}, \omega) dP$. Then the covariance of a is given by

$$C_a(\mathbf{x}_1, \mathbf{x}_2) = \int_{\Omega} (a(\mathbf{x}_1, \omega) - a_0(\mathbf{x}_1))(a(\mathbf{x}_2, \omega) - a_0(\mathbf{x}_2)) dP. \quad (2.3)$$

The covariance $C_a : D \times D \rightarrow \mathbb{R}$ is a symmetric positive semi-definite function, meaning that for any finite set $\{\mathbf{x}^{(i)}\}_{i=1}^N$, the matrix $C_{ij} \equiv C_a(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ is positive semi-definite. If the covariance function is also continuous, then by Mercer's theorem there exists an orthonormal basis $\{a_i(\mathbf{x})\}_{i=1}^{\infty}$ of $L^2(D)$ and a corresponding sequence of scalars $\{\lambda_i\}_{i=1}^{\infty}$ such that

$$C_a(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i=1}^{\infty} \lambda_i a_i(\mathbf{x}_1) a_i(\mathbf{x}_2). \quad (2.4)$$

It can be shown that the series (2.4) converges uniformly and absolutely. We can therefore assume that the scalars $\{\lambda_i\}_{i=1}^{\infty}$ are indexed in non-increasing order $\lambda_1 \geq \lambda_2 \geq \dots$. Multiplying both sides of (2.4) by $a_j(\mathbf{x}_2)$, integrating with respect to \mathbf{x}_2 , and using the fact that the functions $\{a_i\}_{i=1}^{\infty}$ are orthonormal we have that,

$$\int_D C_a(\mathbf{x}_1, \mathbf{x}_2) a_j(\mathbf{x}_2) d\mathbf{x}_2 = \lambda_j a_j(\mathbf{x}_1). \quad (2.5)$$

This shows that the basis functions $\{a_i\}_{i=1}^{\infty}$ and scalars $\{\lambda_i\}_{i=1}^{\infty}$ are the solution to the integral eigenvalue problem (2.5).

If for a specific event ω $a(\mathbf{x}, \omega) - a_0(\mathbf{x}) \in L^2(D)$, then $a(\mathbf{x}, \omega) - a_0(\mathbf{x})$ can be expressed as a linear combination of the basis functions $\{a_i\}_{i=1}^{\infty}$. That is

$$a(\mathbf{x}, \omega) = a_0(\mathbf{x}) + \sum_{i=1}^{\infty} a_i(\mathbf{x}) \xi_i. \quad (2.6)$$

The scalars, ξ_i , can be found by taking the $L^2(D)$ projection of $a - a_0$ with each basis function a_i , that is

$$\xi_i = \int_D (a(\mathbf{x}, \omega) - a_0(\mathbf{x})) a_i(\mathbf{x}) d\mathbf{x}. \quad (2.7)$$

Under the assumption that $a(\mathbf{x}, \omega) \in L^2(D)$ for any value of ω , then (2.7) holds for any value of ω . Therefore the coefficients of the expansion are random variables $\xi_i = \xi_i(\omega)$.

Since we have that

$$a_0(\mathbf{x}) = \mathbb{E}[a(\mathbf{x}, \omega)] = \mathbb{E}[a_0(\mathbf{x})] + \mathbb{E}\left[\sum_{i=1}^{\infty} a_i(\mathbf{x})\xi_i\right], \quad (2.8)$$

the series $\sum_{i=1}^{\infty} a_i(\mathbf{x})\xi_i$ has mean zero. Multiplying both sides of (2.8) by $a_k(\mathbf{x})$, integrating with respect to \mathbf{x} , and using orthonormality we have that

$$0 = \int_D a_k(\mathbf{x}) \sum_{i=1}^{\infty} a_i(\mathbf{x}) \int_{\Omega} \xi_i dP d\mathbf{x} = \int_{\Omega} \xi_k dP = \mathbb{E}[\xi_k]. \quad (2.9)$$

Thus each of the random variables appearing in (2.6) have mean zero. Substituting (2.6) into (2.3) we have that

$$\begin{aligned} C(\mathbf{x}_1, \mathbf{x}_1) &= \int_{\Omega} \left(\sum_{i=1}^{\infty} a_i(\mathbf{x})\xi_i \right) \left(\sum_{j=1}^{\infty} a_j(\mathbf{x}_2)\xi_j \right) dP \\ &= \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} a_i(\mathbf{x}_1)a_j(\mathbf{x}_2) \int_{\Omega} \xi_i\xi_j dP. \end{aligned} \quad (2.10)$$

Multiplying both sides by $a_k(\mathbf{x}_2)$ and integrating with respect to \mathbf{x}_2 , by orthogonality and (2.5) we have that

$$\int_D a_k(\mathbf{x}_2)C(\mathbf{x}_1, \mathbf{x}_2) d\mathbf{x}_2 = \sum_{i=1}^{\infty} a_i(\mathbf{x}_1) \int_{\Omega} \xi_i\xi_k dP = \lambda_k a_k(\mathbf{x}_1). \quad (2.11)$$

Therefore, in addition to being mean zero, each of the random variables ξ_i is uncorrelated and has second moment $\sqrt{\lambda_i}$, that is

$$\int_{\Omega} \xi_i \xi_j dP = \lambda_i \delta_{ij}. \quad (2.12)$$

It is important to note that while the random variables are uncorrelated they are not necessarily independent. Typically each term in the expansion of $a(\mathbf{x}, \omega) - a_0(\mathbf{x})$ is multiplied by $\sqrt{\lambda_i}$. In this case the Karhunen-Loève expansion of $a(\mathbf{x}, \omega)$ is given by

$$a(\mathbf{x}, \omega) = a_0(\mathbf{x}) + \sum_{i=1}^{\infty} \sqrt{\lambda_i} a_i(\mathbf{x}) \xi_i(\omega). \quad (2.13)$$

Normalizing the expansion in this manner makes the random variables appearing in the expansion orthonormal and their values are given by

$$\xi_i(\omega) = \frac{1}{\sqrt{\lambda_i}} \int_D (a(\mathbf{x}, \omega) - a_0(\mathbf{x})) d\mathbf{x}. \quad (2.14)$$

The KL expansion for f can be found in the same manner provided that $f(\mathbf{x}, \omega) - f_0(\mathbf{x}) \in L^2(D)$ for almost every ω . Let the KL expansion for f be given by,

$$f(\mathbf{x}, \omega) = f_0(\mathbf{x}) + \sum_{i=1}^{\infty} \sqrt{\gamma_i} f_i(\mathbf{x}) \eta_i(\omega), \quad (2.15)$$

where f_0 is the mean of f , $\{(f_i, \gamma_i)\}_{i=1}^{\infty}$ are the eigenpairs associated with the covariance of f , and $\{\eta_i(\omega)\}_{i=1}^{\infty}$ is a set of orthonormal mean-zero random variables.

2.1.1 Optimality of the Karhunen-Loève Expansion

Since the Karhunen-Loève expansion involves an infinite number of terms it is necessary to truncate the expansion for use in numerical computations. If $a(\mathbf{x}, \omega)$ is a random field, then we denote the M -term truncated KL expansion by

$$\hat{a}_M(\mathbf{x}, \omega) = a_0(\mathbf{x}) + \sum_{i=1}^M \sqrt{\lambda_i} a_i(\mathbf{x}) \xi_i(\omega). \quad (2.16)$$

In the case where a truncated KL expansion is used to represent the coefficients of a SPDE, the truncation of the expansion introduces a consistency error since the coefficients are not represented exactly. It is possible to construct an expansion similar to (2.16) using any orthonormal basis for $L^2(D)$. In order to minimize the consistency error associated with truncating the KL expansion, the expansion should be accurate using as few terms as possible. An important property of the KL expansion is that the orthonormal basis used in the KL expansion minimizes the truncation error over all choices of orthonormal bases for $L^2(D)$.

To see this, assume that $\{b_i(\mathbf{x})\}_{i=1}^{\infty}$ is an arbitrary orthonormal basis for $L^2(D)$ and that we have expanded the coefficient field $a(\mathbf{x}, \omega)$ in this basis

$$a(\mathbf{x}, \omega) = a_0(\mathbf{x}) + \sum_{i=1}^{\infty} \sqrt{\beta_i} b_i(\mathbf{x}) \xi_i(\omega), \quad (2.17)$$

where the scalars $\{\beta_i\}_{i=1}^{\infty}$ are chosen so that $\mathbb{E}[\xi_i^2] = 1$. If this series is truncated after M terms, then the mean square truncation error is given by

$$\begin{aligned}\mathbb{E}[\epsilon^2] &= \int_{\Omega} (a(\mathbf{x}, \omega) - \hat{a}_M(\mathbf{x}, \omega))^2 dP \\ &= \int_{\Omega} \left(\sum_{i=M+1}^{\infty} \sqrt{\beta_i} b_i(\mathbf{x}) \xi_i(\omega) \right) \left(\sum_{j=M+1}^{\infty} \sqrt{\beta_j} b_j(\mathbf{x}) \xi_j(\omega) \right) dP.\end{aligned}\tag{2.18}$$

Substituting (2.14) into (2.18) we obtain

$$\mathbb{E}[\epsilon^2] = \sum_{i=M+1}^{\infty} \sum_{j=M+1}^{\infty} b_i(\mathbf{x}) b_j(\mathbf{x}) \int_D \int_D C_a(\mathbf{x}_1, \mathbf{x}_2) b_i(\mathbf{x}_1) b_j(\mathbf{x}_2) d\mathbf{x}_1 d\mathbf{x}_2.\tag{2.19}$$

Integrating both sides over D and using the orthonormality of $\{b_i(\mathbf{x})\}_{i=1}^{\infty}$ we obtain

$$\int_D \mathbb{E}[\epsilon^2] d\mathbf{x} = \sum_{i=M+1}^{\infty} \int_D \int_D C(\mathbf{x}_1, \mathbf{x}_2) b_i(\mathbf{x}_1) b_i(\mathbf{x}_2) d\mathbf{x}_1 d\mathbf{x}_2.\tag{2.20}$$

The problem is to find a orthogonal set of functions $\{b_i\}$ that minimizes (2.20) subject to the constraint

$$\int_D b_i(\mathbf{x})^2 d\mathbf{x} = 1,\tag{2.21}$$

that the basis functions be orthonormal. We introduce the Lagrange multipliers $\{\nu_i\}_{i=M+1}^{\infty}$ to enforce these constraints. Minimizing (2.20) is thus equivalent to

finding the critical points of the Lagrangian

$$F[\{b_i(\mathbf{x})\}_{i=M+1}^{\infty}] = \sum_{i=M+1}^{\infty} \int_D \int_D C(\mathbf{x}_1, \mathbf{x}_2) b_i(\mathbf{x}_1) b_i(\mathbf{x}_2) d\mathbf{x}_1 d\mathbf{x}_2 \quad (2.22)$$

$$- \nu_i \left(\int_D b_i(\mathbf{x})^2 d\mathbf{x} - 1 \right).$$

Setting the derivative of (2.22) with respect to $b_i(\mathbf{x}_1)$ equal to zero we obtain

$$0 = \int_D \left(\int_D C(\mathbf{x}_1, \mathbf{x}_2) b_i(\mathbf{x}_2) d\mathbf{x}_2 - 2\nu_i b_i(\mathbf{x}_1) \right) d\mathbf{x}_1. \quad (2.23)$$

Thus the critical points of the Lagrangian (2.22) satisfy the integral eigenvalue equation (2.5) and therefore the basis spanned by the KL eigenfunctions is the basis that minimizes the mean square truncation error.

Replacing a and f in (2.2) with truncated KL expansions \hat{a}_{M_a} and \hat{f}_{M_f} gives

$$-\nabla \cdot \left[\left(a_0(\mathbf{x}) + \sum_{i=1}^{M_a} \sqrt{\lambda_i} a_i(\mathbf{x}) \xi_i(\omega) \right) \nabla u(\mathbf{x}, \boldsymbol{\xi}, \boldsymbol{\eta}) \right] = f_o(\mathbf{x}) + \sum_{i=1}^{M_f} \sqrt{\gamma_i} f_i(\mathbf{x}) \eta_i(\omega), \quad (2.24)$$

where $\boldsymbol{\xi} = [\xi_1, \dots, \xi_{M_a}]^t$ and $\boldsymbol{\eta} = [\eta_1, \dots, \eta_{M_f}]^t$. Note that $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ are M_a -dimensional and M_f -dimensional continuous real random vectors respectively. To simplify notation define $M = M_a + M_f$ and define a third random vector $\mathbf{Y} = [\boldsymbol{\xi}^t, \boldsymbol{\eta}^t]^t$

with

$$Y_i = \begin{cases} \xi_i, & \text{if } 1 \leq i \leq M_a, \\ \eta_{i-M_a} & \text{if } M_a + 1 \leq i \leq M_a + M_f. \end{cases} \quad (2.25)$$

Let

$$\Gamma_i = \begin{cases} \text{Image}(\xi_i) & \text{if } 1 \leq i \leq M_a, \\ \text{Image}(\eta_{i-M_a}) & \text{if } M_a + 1 \leq i \leq M_a + M_f, \end{cases} \quad (2.26)$$

$$\Gamma = \text{Image}(\mathbf{Y}) \subset \prod_{i=1}^M \Gamma_i.$$

Let $\rho(\mathbf{Y})$ be the joint probability distribution of the random vector \mathbf{Y} , and let $\rho_i(Y_i)$ be the marginal density of the i^{th} component of \mathbf{Y} .

Using a truncated KL-expansion ensures that the consistency error is minimized over all finite term expansions of a and f . This is a valuable property since the computational work required by the solution algorithms described in subsequent chapters increases as the size of the problem's parameter space grows. Expanding the random fields in a truncated KL expansion ensures that the minimum number of parameters are used to attain a prescribed accuracy. It should be noted that computing the KL expansion numerically involves computing the largest eigenvalues of a (possibly large) dense matrix. In the case where the covariance function is stationary, that is where $C_a(\mathbf{x}_1, \mathbf{x}_2) = g_a(\mathbf{x}_1 - \mathbf{x}_2)$, with $g_a(z)$ admitting an analytic extension outside of $z = 0$, it is possible to compute the terms in the *KL*-expansion efficiently using the generalized fast multipole method and an iterative eigenvalue

solver based on the Lanczos or Arnoldi methods [43]. However, in other cases this may be prohibitively expensive if a large number of terms are required to obtain desired accuracy. It is often assumed that the covariance function is known and has a form that admits analytic expressions of the eigenfunctions and eigenvalues. In practical applications it is also possible that the SPDE is parameterized using some other method (i.e. modeling possibly correlated physical parameters as random variables). The methods discussed in chapter 4 and chapter 5 are capable of recovering solution statistics under any of these parameterization methods. The choice of a particular parameterization method is a modeling problem and is largely outside the scope of this thesis.

2.2 Regularity of the Solution of the Truncated Diffusion Equation

Before discussing numerical algorithms for the solution of (2.24) it is useful to derive some regularity properties of the solution u . First we will state some relevant properties of the associated deterministic elliptic PDE

$$\begin{aligned} -\nabla \cdot (a(\mathbf{x})u(\mathbf{x})) &= f(\mathbf{x}) & \forall \mathbf{x} \in D, & \quad (2.27) \\ u(\mathbf{x}) &= 0 & \forall \mathbf{x} \in \partial D. & \end{aligned}$$

A more detailed discussion of these points can be found in [6, 15, 28]. First define the Hilbert space $H^1(D)$ to be the space of $L^2(D)$ functions with weak first derivatives

in $L^2(D)$. This space is endowed with the norm

$$\|u\|_{H^1(D)}^2 = \|u\|_{L^2(D)}^2 + \|(\nabla u)^t \cdot \nabla u\|_{L^2(D)}^2. \quad (2.28)$$

Also define $H_0^1(D) \subset H^1(D)$ to be the space of $H^1(D)$ functions that have zero trace. It is assumed that there exists a constants a_{min}, a_{max} such that $0 < a_{min} \leq a(\mathbf{x}) \leq a_{max} < \infty$ and that $f \in L^2(D)$. Under these assumptions the bilinear form

$$\alpha(u, v) = \int_D \nabla u \cdot \nabla v d\mathbf{x} \quad (2.29)$$

is bounded and coercive in $H_0^1(D)$,

$$\begin{aligned} \alpha(u, v) &\leq a_{max} \|u\|_{H^1(D)} \|v\|_{H^1(D)} & \forall u, v \in H^1(D), \\ \alpha(u, u) &\geq \frac{a_{min}}{C} \|u\|_{H^1(D)}^2 & \forall u \in H_0^1(D), \end{aligned} \quad (2.30)$$

where C is a positive constant depending on D . Additionally the linear functional

$$l(v) = \int_D f v d\mathbf{x} \leq \|f\|_{L^2(D)} \|v\|_{H^1(D)} \quad (2.31)$$

is bounded on $H^1(D)$. An application of the Lax-Milgram lemma shows that there exists a unique $u(\mathbf{x}) \in H_0^1(D)$ such that u satisfies a weak form of (2.27)

$$\int_D a(\mathbf{x}) \nabla u \cdot \nabla v d\mathbf{x} = \int_D f v d\mathbf{x} \quad \forall v \in H^1(D). \quad (2.32)$$

Since $u \in H^1(D)$ by (2.30) and (2.31) we have the stability estimate with respect to f ,

$$\|u\|_{H^1(D)}^2 \leq \frac{C}{a_{min}} \alpha(u, u) = \frac{C}{a_{min}} l(u) \leq \frac{C}{a_{min}} \|f\|_{L^2(D)} \|u\|_{H^1(D)}. \quad (2.33)$$

Also u is continuous with respect to the coefficients a , that is, if a_1, a_2 are smooth functions such that $a_1, a_2 \geq a_{min} > 0$ and $u_1, u_2 \in H_0^1(D)$ satisfy the variational problems

$$\begin{aligned} \int_D a_1 \nabla u_1 \cdot \nabla v d\mathbf{x} &= \int_D f v d\mathbf{x}, \\ \int_D a_2 \nabla u_2 \cdot \nabla v d\mathbf{x} &= \int_D f v d\mathbf{x}, \end{aligned} \quad (2.34)$$

for all $v \in H^1(D)$, then

$$\|u_1 - u_2\|_{H^1(D)} \leq \frac{C}{a_{min}^2} \|a_1 - a_2\|_{C(D)} \|f\|_{L^2(D)}. \quad (2.35)$$

We are now ready to discuss the case when the diffusion coefficient and source term are written in terms of truncated KL expansions. Since in this case the solution u varies with the random vector \mathbf{Y} it is convenient for the analysis to define the random inputs $\hat{a}_{M_a}(\mathbf{x}, \boldsymbol{\xi})$ and $\hat{f}_{M_f}(\mathbf{x}, \boldsymbol{\eta})$ as functions on Γ . This is accomplished in

the obvious way by extending both functions by a constant, that is for $Y = [\boldsymbol{\xi}^t, \boldsymbol{\eta}^t]^t$,

$$\hat{a}_{M_a}(\mathbf{x}, \mathbf{Y}) = \hat{a}_{M_a}(\mathbf{x}, \boldsymbol{\xi}), \quad (2.36)$$

$$\hat{f}_{M_f}(\mathbf{x}, \mathbf{Y}) = \hat{f}_{M_f}(\mathbf{x}, \boldsymbol{\eta}).$$

In order to ensure the well posedness of (2.24) we will assume that \hat{a}_{M_a} is uniformly bounded and elliptic, that is, there exist constants a_{min}, a_{max} such that

$$0 < a_{min} \leq \hat{a}_{M_a}(\mathbf{x}, Y) \leq a_{max} < \infty \quad \forall \mathbf{x} \in D \text{ and } \mathbf{Y} \in \Gamma. \quad (2.37)$$

The primary result that motivated the development of solution methods for (2.24) is that under a wide variety of circumstances, the solution u is analytic with respect to each random parameter [2, 35].

Define a weight function¹ $\sigma(\mathbf{Y}) = \prod_{i=1}^M \sigma_i(Y_i) \leq 1$ where

$$\sigma_i = \begin{cases} 1 & \text{if } \Gamma_i \text{ is bounded,} \\ \exp(-\alpha_i |Y_i|) \text{ for some } \alpha_i > 0 & \text{if } \Gamma_i \text{ is unbounded.} \end{cases} \quad (2.38)$$

Define the function space

$$C_\sigma^0(\Gamma; V) = \left\{ v : \Gamma \rightarrow V, v \text{ continuous in } \mathbf{Y}, \max_{\mathbf{Y} \in \Gamma} \|\sigma(\mathbf{Y})v(\mathbf{Y})\|_V < \infty \right\}, \quad (2.39)$$

¹Note that if a is expanded in a truncated KL expansion then each Γ_i must be bounded in order to satisfy (2.37). Other finite representations of random fields may make use of unbounded random variables so we include the case where Γ_i is unbounded in (2.38) for completeness.

where V is a Banach space of functions defined on D . Define

$$C_{loc}^0(\Gamma; L^\infty(D)) = \{v : \Gamma \rightarrow L^\infty(D), v \text{ continuous on } \Gamma_{loc} \forall \Gamma_{loc} \subset \subset \Gamma, \quad (2.40)$$

$$\max_{\mathbf{Y} \in \Gamma_{loc}} \|v(\mathbf{Y})\|_{L^\infty(D)} < \infty\}.$$

We will assume that $\hat{f}_{M_f} \in C_\sigma^0(\Gamma; L^2(D))$ and that the joint probability density $\rho(\mathbf{Y})$ satisfies

$$\rho(Y) \leq C_\rho \exp\left(-\sum_{i=1}^M (\delta_i Y_i)^2\right) \quad \forall Y \in \Gamma, \quad (2.41)$$

where $C_\rho > 0$ and $\delta_i > 0$.

Lemma 1. (*Babuška, Nobile, Tempone [3]*) *If $f \in C_\sigma^0(\Gamma; L^2(D))$ and $a \in C_{loc}^0(\Gamma; L^\infty(D))$ then $u \in C_\sigma^0(\Gamma; H_0^1(D))$.*

Proof. This statement follows immediately from (2.33), (2.35) and the uniform coercivity of a . □

The analyticity of u with respect to Y is proved by examining a single component Y_i at a time. For a fixed value of Y_i define

$$\Gamma_i^* = \text{Image}([Y_1, \dots, Y_{i-1}, Y_{i+1}, \dots, Y_M]^t) \subset \prod_{\substack{j=1 \\ j \neq i}}^M \Gamma_j \quad (2.42)$$

$$\sigma_i^* = \prod_{\substack{j=1 \\ j \neq i}}^M \sigma_j,$$

and let an element of Γ_i^* be denoted by \mathbf{Y}^* .

Theorem 1. (Babuška, Nobile, Tempone [35]) Assume that, for every $\mathbf{Y} = [Y_i, \mathbf{Y}^*] \in \Gamma$ that there exists $\gamma_i < \infty$ such that

$$\left\| \frac{\partial_{Y_i}^k a(\mathbf{x}, \mathbf{Y})}{a(\mathbf{x}, \mathbf{Y})} \right\|_{L^\infty(D)} \leq \gamma_i^k k! \quad \text{and} \quad \frac{\|\partial_{Y_i}^k f(\mathbf{x}, \mathbf{Y})\|_{L^2(D)}}{1 + \|f(\mathbf{x}, \mathbf{Y})\|_{L^2(D)}} \leq \gamma_i^k k!. \quad (2.43)$$

Then the solution $u(\mathbf{x}, Y_i, \mathbf{Y}^*)$ as a function of Y_i , $u : \Gamma_n \rightarrow C_{\sigma_i^*}(\Gamma^*; H_0^1(D))$ admits an analytic extension $u(\mathbf{x}, z, \mathbf{Y}^*)$, $z \in \mathbb{C}$, in the region of the complex plane

$$\Sigma(\Gamma_i; \tau_i) = \{z \in \mathbb{C}, \text{dist}(z, \Gamma_i) \leq \tau_i\}, \quad (2.44)$$

with $0 < \tau_i < 1/(2\gamma_i)$.

If the diffusion coefficients and forcing terms in (2.2) are replaced with truncated KL expansions, then we have the reduced problem

$$\begin{aligned} -\nabla \cdot [\hat{a}_{M_a}(\mathbf{x}, \mathbf{Y}) \nabla u(\mathbf{x}, \mathbf{y})] &= \hat{f}_{M_f}(\mathbf{x}, \mathbf{Y}) & \forall (\mathbf{x}, \mathbf{Y}) \in D \times \Omega, \\ u(\mathbf{x}, \mathbf{Y}) &= 0 & \forall (\mathbf{x}, \mathbf{Y}) \in \partial D \times \Omega. \end{aligned} \quad (2.45)$$

It is shown in [2] that the coefficients and forcing terms in (2.45) satisfy the assumptions of Theorem 1, and therefore the solution u to (2.45) is analytic with respect to each random variable Y_i .

2.3 Derivation of the Stochastic Galerkin Method

The stochastic Galerkin method was first presented in [3, 9, 19] and is an extension of the standard finite element method to SPDEs. The derivation of the stochastic Galerkin method depends on having access to the joint probability density $\rho(\mathbf{Y})$. In particular, it is necessary to construct a basis consisting of multi-variate polynomials that are orthogonal with respect to the measure $\rho(\mathbf{Y})d\mathbf{Y}$. The construction of multi-variate orthogonal polynomials is an active area of research [17]. For a general positive measure defined on \mathbb{R}^M it is impossible to uniquely define a set of orthogonal polynomials. Because of this fact it is generally assumed when discussing stochastic Galerkin methods that the measure $\rho(\mathbf{Y})d\mathbf{Y}$ is a product measure, that is

$$\rho(\mathbf{Y})d\mathbf{Y} = \prod_{i=1}^M \rho_i(Y_i)dY_i. \quad (2.46)$$

This condition is equivalent to assuming that the random variables Y_i are independent. With this condition, a basis of orthogonal polynomials can be constructed as the tensor product of univariate polynomials orthogonal with respect to the weight $\rho_i(Y_i)$.

Define the space

$$H^1(D) \otimes L^2_\rho(\Gamma) = \left\{ v : D \times \Gamma \rightarrow \mathbb{R} \mid \int_\Gamma \|v(\mathbf{x}, \mathbf{Y})\|_{H^1(D)}^2 \rho(\mathbf{Y}) d\mathbf{Y} < \infty \right\}. \quad (2.47)$$

This is a Hilbert space with inner product

$$\langle u, v \rangle_{H^1(D) \otimes L^2_\rho(\Gamma)} = \int_\Gamma \int_D (uv + \nabla u^t \cdot \nabla v) d\mathbf{x} \rho(\mathbf{Y}) d\mathbf{Y}, \quad (2.48)$$

and induced norm

$$\|u\|_{H^1(D) \otimes L^2_\rho(\Gamma)}^2 = \int_\Gamma \int_D \|u\|_{H^1(D)}^2 d\mathbf{x} \rho(\mathbf{Y}) d\mathbf{Y}. \quad (2.49)$$

Also define the space $H_0^1(D) \otimes L^2_\rho(\Gamma) \subset H^1(D) \otimes L^2_\rho(\Gamma)$ to be the subspace consisting of random fields with zero trace on the boundary of D for each realization of \mathbf{Y} .

If we multiply both sides of (2.45) by $v \in H^1(D) \otimes L^2_\rho(\Gamma)$, integrate by parts in the spatial domain and take the expectation, we can define a variational form of (2.45) by: find $u(\mathbf{x}, \mathbf{Y}) \in H_0^1(D) \otimes L^2_\rho(\Gamma)$ such that

$$\int_\Gamma \int_D \hat{a}_{M_a}(\mathbf{x}, \mathbf{Y}) \nabla u \cdot \nabla v d\mathbf{x} \rho d\mathbf{Y} = \int_\Gamma \int_D \hat{f}_{M_f}(\mathbf{x}, \mathbf{Y}) v d\mathbf{x} \rho d\mathbf{Y}, \quad (2.50)$$

for all $v \in H^1(D) \otimes L^2_\rho(\Gamma)$. If \hat{a}_{M_a} satisfies (2.37) and $\hat{f}_{M_f} \in C^0_\sigma(\Gamma; L^2(D))$, then the Lax-Milgram lemma guarantees the existence of a unique solution to (2.50).

Equation (2.50) is often discretized by projecting the variational problem into a finite-dimensional subspace of $L^2_\rho(\Gamma) \otimes H_0^1(D)$, $S_p \otimes V_h$, where $S_p \subset L^2_\rho(\Gamma)$ and $V_h \subset H_0^1(D)$. If $\{\Psi_i(\mathbf{Y})\}_{i=1}^{N_Y}$ is a basis for S_p and $\{\Phi_i(\mathbf{x})\}_{i=1}^{N_x}$ is a basis for V_h then the fully discrete problem can be stated as find $u_{h,p} = \sum_{i=1}^{N_Y} \sum_{j=1}^{N_x} \Psi_i(\mathbf{Y}) \Phi_j(\mathbf{x})$ such

that

$$\int_{\Gamma} \int_D \hat{a}_{M_a} \nabla u_{h,p} \nabla v d\mathbf{x} \rho d\mathbf{Y} = \int_{\Gamma} \int_D \hat{f}_{M_a} v d\mathbf{x} \rho d\mathbf{Y} \quad (2.51)$$

for all $v \in S_p \otimes V_h$. The space V_h is often taken to be a standard finite element space with mesh discretization parameter h (e.g. the Q_1 or Q_2 finite element spaces for continuous piecewise linear or quadratic functions defined on a square mesh [15]). The space of functions S_p is often defined to be a space of globally defined multivariate polynomials in \mathbf{Y} . In [3, 9, 11] S_p is defined to be the space of tensor product polynomials in \mathbf{Y} of degree at most $\mathbf{p} = [p_1, \dots, p_M]^t$,

$$S_p = \left\{ \Psi(\mathbf{Y}) = \prod_{i=1}^M \psi_i(Y_i) : \psi_i(Y_i) \text{ is a polynomial with } \deg(\psi_i(Y_i)) \leq p_i \right\}. \quad (2.52)$$

The case where $p_1 = p_2 = \dots = p_M \equiv p$ is called isotropic. Tensor product polynomial spaces with anisotropic degree (i.e. $p_1 \neq p_2 \neq \dots \neq p_M$) are also discussed in [3]. In [12, 14, 19, 32] the space S_p is defined to be the space of tensor-product polynomials of total degree p ,

$$S_p = \left\{ \Psi(\mathbf{Y}) = \prod_{i=1}^M \psi_i(Y_i) : \psi_i(Y_i) \text{ is a polynomial and } \sum_{i=1}^M \deg(\psi_i(Y_i)) \leq p \right\}. \quad (2.53)$$

This space has dimension $N_Y = \frac{(M+p)!}{M!p!}$.

2.3.1 Error Analysis of the Stochastic Galerkin Method

The error analysis of the stochastic finite element method proceeds along the same lines as the analysis of the standard finite element method. We will present an outline of the error estimates here. A more complete discussion can be found in [3]. First, the finite element approximation u_{hp} is quasi-optimal in the $L^2_\rho(\Gamma) \otimes H^1(D)$ norm.

Lemma 2 (Cea's Lemma). *The error $u - u_{hp}$ is bounded by*

$$\|u - u_{hp}\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \leq C \inf_{V_h \otimes S_p} \|u - u_{hp}\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \quad (2.54)$$

where $C > 0$ depends on the domain, the probability density function ρ and the diffusion coefficient \hat{a}_{M_a} .

Proof. The proof uses standard techniques from the analysis of finite elements (see [6]) and follows from the fact that the bilinear form

$$\alpha(u, v) = \int_\Gamma \int_D \hat{a}_{M_a} \nabla u \nabla v d\mathbf{x} \rho d\mathbf{Y} \quad (2.55)$$

is coercive and bounded on $L^2_\rho(\Gamma) \otimes H_0^1(D)$, and that the discrete solution u_{hp} satisfies (2.51). □

Furthermore, the term on the right hand side of (2.54) separates into two

terms. We have that

$$\|u - u_{hp}\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \leq C \underbrace{\left(\inf_{v \in S_p \otimes H^1_0(D)} \|u - v\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \right)}_{(I)} + \underbrace{\left(\inf_{v \in L^2_\rho(\Gamma) \otimes V_h} \|u - v\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \right)}_{(II)}. \quad (2.56)$$

The term (I) describes the error resulting from the discretization of $L^2_\rho(\Gamma)$. The term (II) describes the error resulting from the discretization of $H^1(D)$. The second term can be bounded by using standard finite element error estimates [6, 28]. This error typically decays as $\mathcal{O}(h^r)$ where r depends on the finite element space chosen to discretize $H^1(D)$ and the regularity of the solution.

If the space S_p is defined to be the space of tensor product polynomials of degree at most $\mathbf{p} = [p_1, \dots, p_{M_a+M_f}]^t$, then it is possible to show that

$$\inf_{v \in S_p \otimes H^1(D)} \|u - v\|_{L^2_\rho(\Gamma) \otimes H^1(D)} \leq \sum_{i=1}^M \zeta_i^{p_i+1} \frac{\|\partial_{Y_i}^{p_i+1} u\|_{L^2(\Gamma) \otimes H^1(D)}}{(p_i + 1)!} \quad (2.57)$$

where $\zeta_i < 1$ [3]. In the isotropic case we have that

$$\inf_{v \in S_p \otimes H^1(D)} \|u - v\|_{L^2(\Gamma) \otimes H^1(D)} \leq C(p, M) \max_{1 \leq i \leq M} (\zeta_i)^{p+1}. \quad (2.58)$$

Therefore the first term in (2.56) decays exponentially with respect to the stochastic discretization parameter p .

2.4 Sparse Grid Collocation

An alternative to the stochastic Galerkin method is the class of stochastic collocation methods, which sample the input operator at a predetermined set of points $\Theta = \{\mathbf{Y}^{(i)}, \dots, \mathbf{Y}^{(N_Y)}\}$ and construct a high-order polynomial approximation to the solution function using discrete solutions to the deterministic PDEs

$$-\nabla \cdot (\hat{a}_{M_a}(\mathbf{x}, \mathbf{Y}^{(i)}) \nabla u(\mathbf{x}, \mathbf{Y}^{(i)})) = \hat{f}_{M_f}(\mathbf{x}, \mathbf{Y}^{(i)}), \quad (2.59)$$

where the diffusion coefficients and forcing term are evaluated at the sample points. If $u : \Gamma \rightarrow V$ is a function of the random parameter space then the sparse grid approximation to this function is denoted $\mathcal{A}(p, M)(u) : \Gamma \rightarrow V$, where p is a discretization parameter referred to as the *grid level* and M is the number of parameters specifying the SPDE. In general, increasing p leads to higher order approximations and to larger collocation point sets. Once the polynomial approximation to u is constructed, statistical information can be obtained at low cost [50], as for the stochastic Galerkin method. Collocation techniques for solving SPDEs were first developed by [2, 35, 50].

There are a large variety of collocation methods. The primary difference between each of them is how the collocation points are chosen and how the approximate solution is constructed from the PDE solution evaluated at these points. The first collocation methods use grids consisting of tensor products of one-dimensional point sets that are based on either the Clenshaw-Curtis or Gaussian abscissas, to construct

a global polynomial approximation [2]. In this method the set of points was the same in each dimension, this is referred to isotropic tensor collocation. In [2] anisotropic tensor grids, where the number of points in each dimension was varied, were also explored. These methods produce a global polynomial approximation where the polynomial degree in each direction is equal to the number collocation points used in that direction. These methods suffer from the “curse of dimensionality” in that the number of collocation points grows rapidly with increasing dimension. Later work used isotropic and anisotropic Smolyak sparse grids to construct collocation point sets [35, 34, 50]. Sparse grid methods are notable in that they attain similar approximation properties to tensor collocation methods while requiring the evaluation of the SPDE at many fewer collocation points.

It was shown in [4] that there is an equivalence between stochastic Galerkin techniques and stochastic collocation techniques. Approximations based on isotropic and anisotropic tensor and sparse grids were each shown to lie in a space of globally defined multivariate polynomials. The approximation error for each of these collocation methods was shown to be equivalent, up to a constant, to the approximation derived by a Galerkin projection onto this polynomial space. In particular it was shown that the approximation error of a solution constructed from collocation on an isotropic level- p Smolyak sparse grid would produce errors on the same order as a Galerkin approximation in the space of multivariate polynomials of total degree p defined in (2.53). In many of these cases the total number of collocation points is greater than the dimension of the associated polynomials space.

2.4.1 Error Analysis of the Sparse Grid Collocation Method

A complete description of the sparse grid collocation method will follow in chapter 3. Here we will present the error analysis of the sparse grid collocation method applied to the stochastic diffusion equation. These results were first established in [35] and show that the error associated with the sparse grid collocation method decays exponentially provided that the solution u is sufficiently smooth. We will assume that each deterministic PDE (2.59) is discretized by finite elements with V_h being a discrete subspace of $H^1(D)$. Define $u_h(\mathbf{x}, \mathbf{Y}) : \Gamma \rightarrow \mathbb{R}^{N_x}$ to be the discrete solution to (2.59) at the point \mathbf{Y} , where N_x is the number of spatial degrees of freedom. Let $\mathcal{A}(p, M)(u_h)$ be the sparse grid approximation to u_h . Similar to the analysis of the stochastic Galerkin method, the error associated with collocation methods separates into two parts, the error associated with the discretization in space and sparse grid interpolation error,

$$\begin{aligned} \|u - \mathcal{A}(p, M)(u_h)\|_{L^2_\rho(\Gamma) \otimes H^1(D)} &\leq \underbrace{\|u_h - \mathcal{A}(p, M)(u_h)\|_{L^2_\rho(\Gamma) \otimes H^1(D)}}_{(I)} + \quad (2.60) \\ &\quad \underbrace{\|u - u_h\|_{L^2_\rho(\Gamma) \otimes H^1(D)}}_{(II)}. \end{aligned}$$

As in (2.56), (II) is the spatial discretization error and can be bounded using techniques from finite element theory. The first term is the error resulting from the discretization of the stochastic portion of the problem by the sparse grid collocation method. It is shown in [35] that the stochastic error (I) decays exponentially in the sparse grid level.

Lemma 3 (Nobile, Tempone, Webster [35]). *Given a function $u \in C^0(\Gamma) \otimes H^1(D)$ that satisfies the assumptions of theorem 1, the Smolyak formula (3.18) based on Gaussian abscissas satisfies:*

$$\|u - \mathcal{A}(p, M)(u)\|_{L^2(\Gamma) \otimes H^1(D)} \leq CF^M e^{-\sigma p}, \quad (2.61)$$

Where σ , C , and F are positive constants that depend on Γ and the radius of convergence τ from Lemma 1.

The sparse grid will have on the order of 2^p more points than there are stochastic degrees of freedom in the Galerkin scheme, $|\Theta| \approx 2^p N_Y$ for $M \gg 1$ [50].

Chapter 3

Comparison of the Stochastic Finite Element Method and the Stochastic Collocation Method

In this chapter we present a comparison of the stochastic Galerkin and stochastic sparse grid collocation methods applied to the stochastic linear elliptic diffusion equation with zero Dirichlet boundary conditions. Here we will assume that only the diffusion coefficient is uncertain. This can be written as

$$\begin{aligned} -\nabla \cdot (a(\mathbf{x}, \omega) \nabla u(\mathbf{x}, \omega)) &= f(\mathbf{x}) & (\mathbf{x}, \omega) \in D \times \Omega, \\ u(x, \omega) &= 0 & (\mathbf{x}, \omega) \in \partial D \times \Omega. \end{aligned} \quad (3.1)$$

The random input field will be represented as a truncated KL expansion given by

$$a(\mathbf{x}, \omega) = \hat{a}_M(\mathbf{x}, \boldsymbol{\xi}(\omega)) = a_0(\mathbf{x}) + \sum_{k=1}^M \sqrt{\lambda_k} \xi_k(\omega) a_k(\mathbf{x}), \quad (3.2)$$

where (λ_i, a_i) are solutions to the integral eigenvalue equation (2.5) with the covariance of a denoted as $C(\mathbf{x}_1, \mathbf{x}_2) : D \times D \rightarrow \mathbb{R}$. By (2.9), (2.12), and (2.14), the random variables appearing in (3.2) are uncorrelated, mean zero, and are given by

$$\xi_k(\omega) = \frac{1}{\sqrt{\lambda_k}} \int_D (a(\mathbf{x}, \omega) - a_0(\mathbf{x})) a_k(\mathbf{x}) \, d\mathbf{x}. \quad (3.3)$$

We make the further modeling assumption that the random variables $\{\xi_k\}$ are independent and admit a joint probability density of the form $\rho(\boldsymbol{\xi}) = \prod_{k=1}^M \rho_k(\xi_k)$. The covariance function is positive definite and its eigenvalues can be ordered so that $\lambda_1 \geq \lambda_2 \geq \dots \geq 0$. To ensure the existence of a unique solution to (3.1) it is necessary to assume that the diffusion coefficient is uniformly bounded away from zero; we assume that there exist constants a_{min} and a_{max} such that

$$0 < a_{min} \leq \hat{a}_M(\mathbf{x}, \boldsymbol{\xi}) \leq a_{max} < \infty, \quad (3.4)$$

almost everywhere P -almost surely, $\hat{a}_M(\cdot, \boldsymbol{\xi}) \in L_2(D)$ P -almost surely, and $\hat{f}_M \in C_\sigma^0(\Gamma; L^2(D))$.

In this chapter we present a model of the computational costs and compare the performance of the stochastic Galerkin method [3, 9, 19, 42, 51, 52] and the sparse grid collocation method [2, 35, 50] for computing the solution of (3.1). See [4] for related work. Section 3.1 outlines a modification of the stochastic Galerkin method that uses finite differences to discretize in space rather than finite elements. Section 3.2 outlines the sparse grid collocation method. Section 3.3 presents our model of the computational costs of the two methods. Section 3.4 explores the performance of the methods applied to several numerical examples using the *Trilinos* software package [24]. Finally in Section 3.5 we draw some conclusions.

3.1 Stochastic Galerkin Method With Finite Differences

The conventional stochastic Galerkin method, as described in chapter 2, uses finite elements in both space and the stochastic domain to discretize (3.1). It is also possible to use finite differences to discretize in space and finite elements to discretize the stochastic domain. If a uniform mesh is used with a finite difference discretization in space then the linear system obtained will be spectrally equivalent to the system obtained by the traditional stochastic Galerkin method on a uniform spatial mesh. Details of this method are as follows.

Define $\Gamma = \times_{k=1}^M \Gamma_k = \times_{k=1}^M \text{Image}(\xi_k)$ and let

$$\langle u, v \rangle_{L^2_\rho(\Gamma)} = \int_{\Gamma} u(\boldsymbol{\xi})v(\boldsymbol{\xi})\rho(\boldsymbol{\xi}) d\boldsymbol{\xi} = \int_{\Omega} u(\boldsymbol{\xi}(\omega))v(\boldsymbol{\xi}(\omega)) dP. \quad (3.5)$$

be the inner product over the space $L^2_\rho(\Gamma) = \{v(\boldsymbol{\xi}) : \|v\|_{L^2_\rho(\Gamma)}^2 = \langle v^2 \rangle < \infty\}$. We can define a variational form of (3.1) in the stochastic domain by: for all $\mathbf{x} \in D$, find $u(\mathbf{x}, \boldsymbol{\xi}) \in L_2(\Gamma)$ such that

$$-\langle \nabla \cdot (a \nabla u), v \rangle = \langle f, v \rangle \quad (3.6)$$

for all $v \in L_2(\Gamma)$. This leads to a set of coupled second-order linear partial differential equations in the spatial dimension. Define S_p to be the space of multivariate polynomials in $\boldsymbol{\xi}$ of total degree at most p . This space has dimension $N_\xi = \frac{(M+p)!}{M!p!}$. Let $\{\Psi_k\}_{k=0}^{N_\xi-1}$ be a basis for S_p orthonormal with respect to the inner product (3.5).

Substituting the KL-expansion for $a(\mathbf{x}, \omega)$ and restricting (3.6) to $v \in S_p$ gives

$$-\int_{\Gamma} \nabla \cdot \left(\hat{a}_M(\mathbf{x}, \boldsymbol{\xi}) \left(\sum_{i=0}^{N_{\xi}-1} \nabla u_i(\mathbf{x}) \Psi_i \right) \right) \Psi_j d\boldsymbol{\xi} = \int_{\Gamma} f \Psi_j d\boldsymbol{\xi} \quad \forall j = 0 : N_{\xi} - 1. \quad (3.7)$$

This is a set of coupled second-order differential equations for the unknown functions $u_i(\mathbf{x})$ defined on D , which can then be discretized using finite differences. This gives rise to a global linear system of the form

$$A\vec{u} = \vec{f}. \quad (3.8)$$

With orderings of \vec{u} and \vec{f} (equivalently, the columns and rows of A , respectively) corresponding to a blocking by spatial degrees of freedom, $\vec{u}^T = [u_1^T, \dots, u_{N_{\xi}}^T]$, the coefficient matrix has the tensor product structure

$$A = \sum_{k=0}^M G_k \otimes A_k. \quad (3.9)$$

The matrices $\{G_k\}$ depend only on the stochastic basis,

$$G_0(i, j) = \langle \Psi_i \Psi_j \rangle, \quad (3.10)$$

$$G_k(i, j) = \langle \xi_k \Psi_i \Psi_j \rangle,$$

The matrices $\{A_k\}$ correspond to a standard five-point operator for $-\nabla \cdot (a_k \nabla u)$ and $\{f_k\}$ are the associated right hand side vectors. In the two-dimensional examples we

explore below, we use a uniform mesh of width h . The discrete difference operators are formed by using the following five-point stencil

$$\begin{bmatrix} & & a_k(x, y + \frac{h}{2}) & & \\ a_k(x - \frac{h}{2}, y) & a_k(x, y) & a_k(x + \frac{h}{2}, y) & & \\ & & a_k(x, y - \frac{h}{2}) & & \end{bmatrix}. \quad (3.11)$$

The matrix A_k is symmetric for all k and A positive definite by (3.4). Since the random variables appearing in (3.3) are mean-zero, it also follows from (3.4) that A_0 is positive-definite.

The matrix A is of order $N_x N_\xi$ where N_x is the number of degrees of freedom used in the spatial discretization. It is also sparse in the block sense due to the orthogonality of the stochastic basis functions. Specifically, since the random variables $\{\xi_k\}$ are assumed to be independent, we can construct the stochastic basis functions $\{\Psi_i\}$ by taking tensor products of univariate polynomials satisfying the orthogonality condition

$$\langle \psi_i(\xi_k), \psi_j(\xi_k) \rangle_k \equiv \int_{\Gamma_k} \psi_i(\xi_k) \psi_j(\xi_k) \rho_k(\xi_k) d\xi_k = \delta_{ij}. \quad (3.12)$$

This basis is referred to as the generalized polynomial chaos of order p . The use of this basis for representing random fields is discussed extensively in [19] and [51]. The univariate polynomials appearing in the tensor product can be expressed via

the familiar three-term recurrence

$$\psi_{i+1}(\xi_k) = (\xi_k - \alpha_i)\psi_i(\xi_k) - \beta_i\psi_{i-1}(\xi_k), \quad (3.13)$$

where $\psi_0 = 1$, $\psi_{-1} = 0$. It follows that

$$G_0(i, j) = \langle \Psi_i, \Psi_j \rangle = \prod_{k=1}^M \langle \psi_{i_k}(\xi_k), \psi_{j_k}(\xi_k) \rangle_k = \prod_{k=1}^M \delta_{i_k j_k} = \delta_{ij}, \quad (3.14)$$

and for $k > 0$ the entries in G_k are

$$\begin{aligned} G_k(i, j) &= \langle \xi_k \Psi_i, \Psi_j \rangle \\ &= \langle \xi_k \psi_{i_k}, \psi_{j_k} \rangle_k \prod_{l=1, l \neq k}^M \langle \psi_{i_l}, \psi_{j_l} \rangle_l \\ &= (\langle \psi_{i_{k+1}}, \psi_{j_k} \rangle_k + \alpha_{i_k} \langle \psi_{i_k}, \psi_{j_k} \rangle_k + \beta_{i_k} \langle \psi_{i_{k-1}}, \psi_{j_k} \rangle_k) \prod_{l=1, l \neq k}^M \langle \psi_{i_l}, \psi_{j_l} \rangle_l. \end{aligned} \quad (3.15)$$

Thus G_0 is diagonal and G_k has at most three entries per row for $k > 0$. Furthermore, if the density functions ρ_k are symmetric with respect to the origin, i.e. $\rho_k(\xi_k) = \rho_k(-\xi_k)$, then the coefficients α_i in the three-term recurrence are all zero and G_k then has at most two non-zeros per row. The sparsity of the matrices $\{G_k\}$ causes the full system A to be sparse in the block sense. Furthermore, each non-zero block of A inherits the sparsity structure associated with the five-point finite difference matrix (i.e. it is tri-diagonal in the one dimensional case, penta-diagonal in the two dimensional case, etc.). Examples of the block sparsity structure of the matrix A is shown in Figure 3.1.

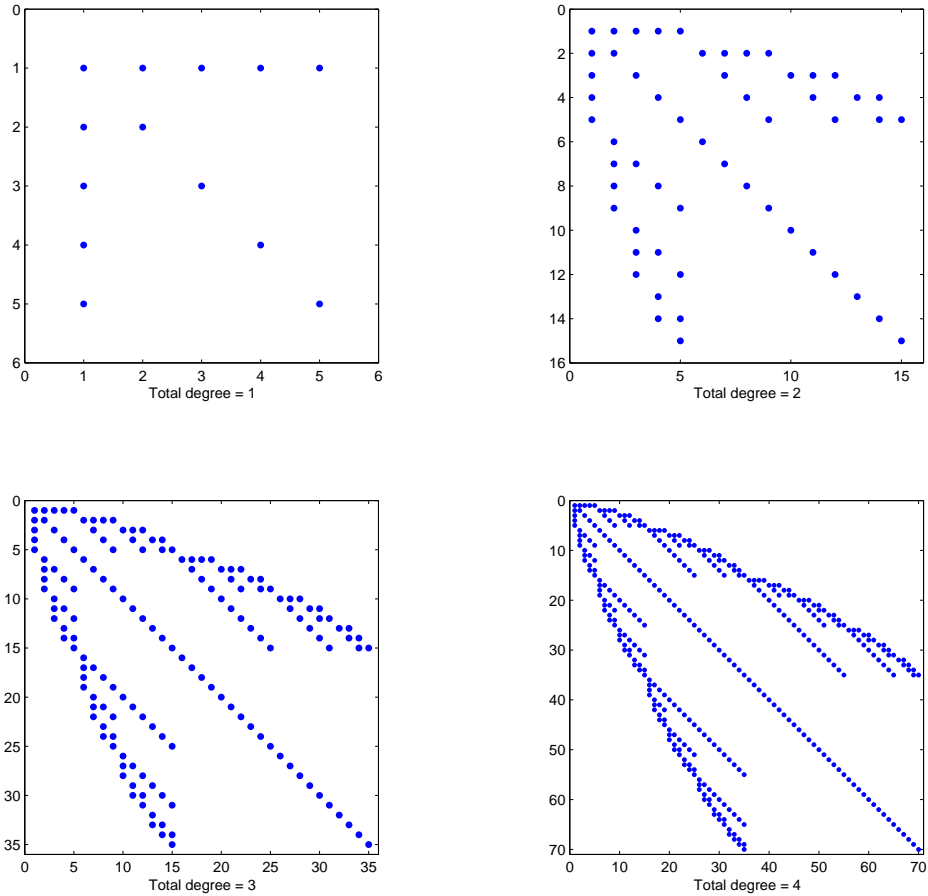


Figure 3.1: Block structure of stochastic Galerkin stiffness matrix for $M = 4$ and $p = 1, 2, 3, 4$.

The stochastic Galerkin method requires the solution to the large linear system (3.8). Once the solution to (3.8) is obtained, statistical quantities such as moments or a probability distribution associated with the solution process can be obtained cheaply [19]. Although the Galerkin linear system is large, there are techniques available by which the solution can be computed efficiently.

We elect to directly solve the large symmetric and positive-definite Galerkin system using the conjugate gradient (CG) method. CG only requires the evalua-

tion of matrix-vector products, so that it is not necessary to store the assembled matrix A . The matrix-vector products can be performed implicitly following a procedure described in [39]. Each matrix A_k is constructed and the terms $\langle \xi_k \Psi_i \Psi_j \rangle$ are precomputed. When the global stiffness matrix A is to be multiplied by a vector u , the matrix-vector product is expressed as $(Au)_j = \sum_{i=0}^{N_\xi-1} \sum_{k=0}^M \langle \xi_k \Psi_i \Psi_j \rangle (A_k u_i)$. The terms $A_k u_i$ are precomputed and then scaled by the terms $\langle \xi_k \Psi_i \Psi_j \rangle$ as needed. This approach is efficient since most of the terms $\langle \xi_k \Psi_i \Psi_j \rangle$ are zero. The cost of performing the matrix-vector product in this manner is essentially determined by the computation of $A_k u_i$ for $0 \leq k \leq M$ and $0 \leq i \leq N_\xi - 1$, which entails $(M+1)N_\xi$ sparse matrix-vector products by matrices $\{A_k\}$ of order N_x . The implicit matrix-vector product also only requires the assembly of $M+1$ order- N_x stiffness matrices and the assembly of the components $\langle \xi_k \Psi_i \Psi_j \rangle$ of $\{G_k\}$. Alternatively one could assemble the entire Galerkin matrix and perform the block matrix-vector product in the obvious way. This is obviously less efficient in terms of memory usage since it requires the assembly and storage of many matrices of the form $\langle \xi_k \Psi_i \Psi_j \rangle (A_k u_i)$. It is also shown in [39] that performing the matrix-vector products in this way is less efficient in terms of memory bandwidth.

To obtain fast convergence, we will also use a preconditioner. In particular, it has been shown in [14] that an effective choice is an approximation to $A_0^{-1} \otimes G_0^{-1}$, where A_0 is the mean stiffness matrix. Since the stochastic basis functions are orthonormal, G_0 is the identity matrix. The preconditioner then entails the approximate action of N_ξ uncoupled copies of A_0^{-1} . For this, we will use a single iteration of an algebraic multigrid solver provided by [18].

3.2 Sparse Grid Collocation

As discussed in chapter 2, an alternative to the Galerkin scheme is the collocation method, which samples the input operator at a predetermined set of points $\Theta = \{\boldsymbol{\xi}^{(1)}, \dots, \boldsymbol{\xi}^{(n)}\}$ and constructs a high-order polynomial approximation to the solution function using discrete solutions to the deterministic PDE. In this section we will present a complete description of the stochastic isotropic sparse grid collocation method.

For simplicity of presentation we first discuss a collocation method using the full tensor product of one-dimensional point sets. Let $\{\psi_i\}$ be the set of polynomials orthogonal with respect to the measure ρ_k . Let $\theta_i = \{\xi : \psi_i(\xi) = 0\} := \{\xi_{i,k}^{(j)}\}_{j=1}^i$ for $i = 1, 2, \dots$, and $j = 1, 2, \dots, i$. These are the abscissas for an (i) -point Gauss quadrature rule with respect to the measure ρ_k . A one-dimensional (i) -point interpolation operator is given by

$$U^i(u)(\xi) = \sum_{j=1}^i u(\xi_i^{(j)}) l_i^{(j)}(\xi), \quad l_i^{(j)}(\xi) = \prod_{n=1, n \neq j}^i \frac{\xi - \xi_i^{(n)}}{\xi_i^{(j)} - \xi_i^{(n)}}. \quad (3.16)$$

These can be used to construct an approximation to the M -dimensional random function $u(\mathbf{x}, \boldsymbol{\xi})$ by defining a tensor interpolation operator

$$U^{i_1} \otimes \dots \otimes U^{i_M}(u)(\boldsymbol{\xi}) = \sum_{j_1=1}^{i_1} \dots \sum_{j_M=1}^{i_M} u(\xi_{i_1}^{(j_1)}, \dots, \xi_{i_M}^{(j_M)}) (l_{i_1}^{(j_1)} \otimes \dots \otimes l_{i_M}^{(j_M)}). \quad (3.17)$$

The evaluation of this operator requires the solution of a collection of deterministic PDEs (2.59), one for each sample point in $\Theta_{tensor} = \times_{j=1}^M \theta_{i_j}$.

This method suffers from the so-called curse of dimensionality since the number of sample points $|\Theta_{tensor}| = \prod_{j=1}^M |\theta_{i_j}| = \prod_{j=1}^M (i_j)$. In the case where $i_1 = i_2 = \dots = i_M \equiv i$ we have that $|\Theta_{tensor}| = i^M$, which grows exponentially with the dimension of the problem. This makes tensor-product collocation inappropriate for problems where the stochastic dimension is moderate or large. This cost can be significantly reduced using sparse grid methods [50].

Sparse grid collocation methods are based on the Smolyak approximation formula. The Smolyak operator $\mathcal{A}(p, M)$ is a linear combination of the product formulas in (3.17). Let $Y(p, M) = \{\mathbf{i} \in \mathbb{N}^M : p + 1 \leq |\mathbf{i}|_1 \leq p + M\}$. Then the Smolyak formula is given by

$$\mathcal{A}(p, M)(u) = \sum_{\mathbf{i} \in Y(p, M)} (-1)^{p+M-|\mathbf{i}|_1} \binom{M-1}{p+M-|\mathbf{i}|_1} (U^{i_1} \otimes \dots \otimes U^{i_M}). \quad (3.18)$$

The evaluation of the Smolyak formula requires the solution of deterministic PDEs (2.59) for $\boldsymbol{\xi}^{(l)}$ in the set of points

$$\Theta_{p, M} = \bigcup_{\mathbf{i} \in Y(p, M)} (\theta_{i_1} \times \dots \times \theta_{i_M}). \quad (3.19)$$

For moderate or large values of M , $|\Theta_{p, M}| \ll |\Theta_{tensor}|$.

If Gaussian abscissas with respect to the marginal density ρ_i are used in the definition of θ_i and if u is an M -variate polynomial of total degree p in $\boldsymbol{\xi}$, then $u = \mathcal{A}(p, M)u$ [4]; that is, the Smolyak interpolant exactly reproduces such polyno-

mials.¹ The parameter p in $\mathcal{A}(p, M)$ is referred to as the sparse grid level. It is shown in [35] that sampling the differential operator on the sparse grid $\Theta_{p,M}$ will produce $\mathcal{A}(p, M)(u) = u_p$ where u_p is an approximate solution to (3.1) of similar accuracy to the solution obtained using an order p stochastic Galerkin scheme. The sparse grid will have on the order of a factor of 2^p more points than there are stochastic degrees of freedom in the Galerkin scheme, $|\Theta| \approx 2^p N_\xi$ for $M \gg 1$ [50].

For a fully non-intrusive collocation method, the diffusion coefficients of (2.59) would be sampled at the points in the sparse grid and for each sample the deterministic stiffness matrix would be constructed for the PDE

$$-\nabla \cdot (\hat{a}_M(\mathbf{x}, \boldsymbol{\xi}^{(l)}) \nabla u(\mathbf{x}, \boldsymbol{\xi}^{(l)})) = \hat{f}_M(\mathbf{x}, \boldsymbol{\xi}^{(l)}). \quad (3.20)$$

This repeated assembly can be very expensive. We elect in our implementations to take advantage of the fact that the stiffness matrix at a given value of the random variable is a scaled sum of the stiffness matrices A_k appearing in (3.9). For a given value of $\boldsymbol{\xi}$ the deterministic stiffness matrix can be expressed as

$$A(\boldsymbol{\xi}) = A_0 + \sum_{k=1}^M \xi_k A_k. \quad (3.21)$$

In our implementation we assemble the matrices $\{A_k\}$ first by applying the finite difference stencil (3.11) to the functions a_k and then compute the scaled sum (3.21)

¹An alternative choice of sparse grid points is to use the Clenshaw-Curtis abscissas with $|\theta_1| = 1$ and $|\theta_i| = 2^{i-1} + 1$ for $i > 1$, which produces nested sparse grids [21, 35, 50]. The choice used here, non-nested Gaussian abscissas with a linear growth rate, $|\theta_i| = i$, produces grid sets of cardinalities comparable to those for the nested Clenshaw-Curtis grids, i.e. $|\Theta_{p,M}^{Gaussian}| \approx |\Theta_{p,M}^{Clenshaw-Curtis}|$.

of these matrices at each collocation point. This requires the assembly of $M+1$ order N_x stiffness matrices and then at each collocation point, to assemble $A(\boldsymbol{\xi})$, $M+1$ order N_x matrix additions are required. An often cited advantage of collocation methods is that they do not require modification to the underlying deterministic PDE code. The method proposed here is intrusive in that it requires modification of the deterministic PDE solver for the diffusion equation; however it greatly reduces the amount of time required to perform assembly in the collocation method.

One could construct a separate multigrid preconditioner for each of the deterministic systems. This can become very expensive as the cost of constructing an algebraic multigrid preconditioner can often be of the same order as the iterative solution. This repeated cost can be eliminated if one simply builds an algebraic preconditioner for the mean problem A_0^{-1} and applies this preconditioner to all of the deterministic systems. If the variance of the operator is small then the mean-based AMG preconditioner is nearly as effective as doing AMG on each sub-problem and saves time in setup costs. Other techniques for developing preconditioners balancing performance with the cost of repeated construction are considered in [21].

3.3 Modeling Computational Costs

From an implementation perspective, collocation is quite advantageous in that it requires only a modest modification to existing deterministic PDE applications. Collocation samples the stochastic domain at a discrete set of points and requires the solution of uncoupled deterministic problems. This can be accomplished by

repeatedly invoking a deterministic application with different input parameters determined by the collocation point-sampling method. A Galerkin method, on the other hand, is much more intrusive as it requires the solution of a system of equations with a large coefficient matrix that has been discretized in both spatial and stochastic dimensions. To better understand the relationship between these two methods, we develop a model for the computational costs.

We begin by stating in more detail some of the computational differences between the two methods. The Galerkin method requires the computation of the matrices $G_0 = \langle \Psi_i \Psi_j \rangle$ and $G_k = \langle \xi_k \Psi_i \Psi_j \rangle$ associated with the stochastic basis functions, the assembly of the right-hand side vector and the spatial stiffness matrices $\{A_k\}$, and finally the solution to the large coupled system of equations. Collocation requires the construction of a sparse grid and the derivation of an associated sparse grid quadrature rule, and the assembly/solution of a series of deterministic subproblems. Further, as observed above, the number of sample points needed for collocation tends to be much larger than the dimension of the Galerkin system required to achieve comparable accuracy.

In this thesis we only examine methods that are isotropic in the stochastic dimension, allocating an equal number of degrees of freedom to each stochastic direction. Anisotropic versions of both the sparse grid collocation method and the stochastic Galerkin could be implemented by weighting the maximum degree of the approximation space in each direction. This has been explored in the case of sparse grid collocation [34]. We expect a cost comparison for an anisotropic stochastic Galerkin method and the anisotropic sparse grid collocation method to be compara-

ble to that of their isotropic counterparts. Additional modifications to the stochastic collocation for adaptively dealing with very high-dimensional problems are considered in [30] and [31]. The method considered in [30] is presented and extended in chapter 4.

For a fixed M, p , let Z_G be the number of preconditioned conjugate gradi-

	Level p Sparse Grid (Gaussian) $ \Theta $	Galerkin N_ξ	Non-Zero Blocks per row in Galerkin Matrix	Tensor Grid
$M = 2$				
$p = 1$	5	3	2.33	4
$p = 2$	13	6	3.00	9
$p = 3$	29	10	3.40	16
$p = 4$	53	15	3.67	25
$M = 10$				
$p = 1$	21	11	2.82	1024
$p = 2$	221	66	4.33	59049
$p = 3$	1581	286	5.62	1048576
$p = 4$	8761	1001	6.71	9765625
$M = 20$				
$p = 1$	41	21	2.90	1.04×10^6
$p = 2$	841	231	4.64	3.49×10^9
$p = 3$	11561	1771	6.22	1.10×10^{12}

Table 3.1: Degrees of freedom for various methods

ent (PCG) iterations required to solve the Galerkin system, let $N_\xi\alpha$ be the cost of applying the mean-based preconditioner during a single iteration of the stochastic Galerkin method, and let $N_\xi\gamma$ be the cost of a single matrix-vector product for (3.8), where α and γ are constants. Note in particular that α is constant because of the optimality of the multigrid computation. Then the total cost of the Galerkin

method can be modeled by

$$\text{Galerkin cost} = N_\xi Z_G (\alpha + \gamma) \quad (3.22)$$

The parameter γ can be thought of as the number of order- N_x matrix-vector products required per block row in the stochastic Galerkin matrix. When implementing the implicit matrix-vector product, γ is equal to $M + 1$.

We can model the costs of the collocation method with the mean-based multi-grid preconditioner by

$$\text{collocation cost} = Z_C 2^p N_\xi (\alpha + 1) \quad (3.23)$$

where p is the Smolyak grid level, N_ξ is the number of degrees of freedom needed by an order p Galerkin system, Z_C is the average number of PCG iterations needed to solve a single deterministic system, and $\alpha + 1$ is the cost of the preconditioning operation and a single order- N_x matrix-vector product. The factor of 2^p derives from the relation between the number of degrees of freedom for the stochastic Galerkin and sparse grid collocation methods for large M .

The relative costs of the two methods depend on the parameter values. In particular,

$$\frac{\text{Collocation cost}}{\text{Galerkin cost}} = \left(\frac{Z_C}{Z_G} \right) 2^p \frac{(\alpha + \gamma)}{(\alpha + 1)} \quad (3.24)$$

If, for example, the ratio of iteration counts (Z_G/Z_C) is close to 1 and the preconditioning costs dominate the matrix vector costs (i.e. $\alpha \gg \gamma$), then, we can expect the stochastic Galerkin method to outperform the sparse grid collocation method because of the factor 2^p . Alternatively, if γ is comparable compared to α , the preconditioning cost, then collocation is more attractive. The cost of the two methods is identical when (3.23) and (3.22) are equal. After canceling terms this gives $2^p\alpha \approx (Z_{SG}/Z_C)(\alpha+\gamma)$. Table 3.1 gives values of N_ξ , and $|\Theta|$ for various values of M and p . One can observe that the estimate $2^p N_\xi \approx |\Theta|$ is a slight overestimate but improves as M grows larger. For reference, the number of points used by a full tensor product grid is also shown.

In our application, we fix the multigrid parameters as follows: One V-cycle is performed at each iteration and within each V-cycle one symmetric Gauss-Seidel iteration is used for both presmoothing and postsmoothing. The coarsest grid is assumed coarse enough so that a direct solver can be used without affecting the cost per iteration; in our implementations we use a 1×1 grid. These parameters were chosen to optimize the run time of a single deterministic solve. The cost to apply a single multigrid iteration is equivalent to approximately 5-6 matrix products (2 matrix-vector products for fine level presmoothing, another 2 for fine level postsmoothing, and 1 matrix-vector product for a fine level residual calculation). Thus α can be assumed to be 5 or 6 after accounting for computational overhead.

In the remainder of this chapter, we explore the model and assess the validity of assumptions. In particular, we compare the accuracy of a level- p Smolyak grid with a degree- p polynomial approximation in the Galerkin approach. We also investigate

the cost of matrix-vector products, and the convergence behavior of mean-based preconditioning.

3.4 Experimental Results and Model Validation

In this section we present the results of numerical experiments with the stochastic Galerkin and collocation methods, with the aims of comparing their accuracy and solution costs and validating the model developed in the previous section. First, we investigate a problem with a known solution to verify that both methods are converging to the correct solution and to examine the convergence of the PCG iteration. Second, we examine two problems where the diffusion coefficient is defined using a known covariance function, and we measure the computational effort required by each method.

3.4.1 Behavior of the Preconditioned Conjugate Gradient Algorithm

For well-posed Poisson problems, PCG with a multigrid preconditioner converges rapidly. Since collocation entails the solution of multiple deterministic systems, we expect multigrid to behave well. For Galerkin systems, the performance of mean-based preconditioning is more complicated. To understand this we investigate the problem

$$-\nabla \cdot (a(\mathbf{x}, \xi)u(\mathbf{x}, \xi)) = f(\mathbf{x}, \xi) \tag{3.25}$$

in the domain $[-.5, .5]^2$ with zero Dirichlet boundary conditions, where the diffusion coefficient given as a one-term KL expansion,

$$a(\mathbf{x}, \xi) = 1 + \sigma \frac{1}{\pi^2} \xi \cos\left(\frac{\pi}{2}(x^2 + y^2)\right). \quad (3.26)$$

We choose the function

$$u = \exp(-|\xi|^2) 16(x^2 - .25)(y^2 - .25) \quad (3.27)$$

as the exact solution, and the forcing term f is defined by applying (3.25) to u .

The diffusion coefficient must remain positive for the problem to remain well-posed. This is the case provided

$$\left| \sigma \frac{1}{\pi^2} \xi \cos\left(\frac{\pi}{2} r^2\right) \right| < 1, \quad (3.28)$$

which holds when $|\xi| < \frac{\pi^2}{\sigma}$. As a consequence of this, well-posedness cannot be guaranteed when ξ is unbounded. There are various ways this can be addressed.

We assume here that the random variable in (3.26) has a *truncated Gaussian density*

$$\rho(\xi) = \frac{1}{\int_{-c}^c \exp\left(-\frac{\xi^2}{2}\right) d\xi} \exp\left(-\frac{\xi^2}{2}\right) \mathbf{1}_{[-c,c]}, \quad (3.29)$$

which corresponds to taking the diffusion coefficient from a screened sample where the screening value c is chosen to enforce the conditions (3.28) for ellipticity and boundedness. The cutoff parameter c is chosen to be equal to 2.575. For this cutoff

the area under a standard normal distribution between $\pm c$ is equal to .99. For this value of c , $|\xi| < 2.575$ and the problem is guaranteed to remain well posed provided that $\sigma < \frac{\pi^2}{\max(|\xi|)} = 3.8329$.

Polynomials orthogonal to a truncated Gaussian measure are referred to as Rys polynomials [17]. As the parameter c is increased, the measure approaches the standard Gaussian measure and the Rys polynomials are observed to approach the behavior of the Hermite polynomials. For our implementation of collocation, the sparse grids are based on the zeros of the Rys polynomials. This leads to an efficient multidimensional quadrature rule for performing integration with respect to the measure (3.29), using the Gaussian weights and abscissas.

The recurrence coefficients for orthogonal polynomials can be expressed explicitly as

$$\alpha_i = \frac{\int_{\Gamma} \xi \psi_i(\xi)^2 \rho(\xi) d\xi}{\int_{\Gamma} \psi_i(\xi)^2 \rho(\xi) d\xi}, \quad \beta_i = \frac{\int_{\Gamma} \psi_i(\xi)^2 \rho(\xi) d\xi}{\int_{\Gamma} \psi_{i-1}(\xi)^2 \rho(\xi) d\xi}. \quad (3.30)$$

In the case of Hermite polynomials there exist closed forms for the recurrence coefficients $\{\alpha_i, \beta_i\}$. No such closed form is known in general for the Rys polynomials so a numerical method must be employed. The generation of orthogonal polynomials by numerical methods is discussed extensively in [17] and the use of generalized polynomial chaos bases in the stochastic Galerkin method is discussed in [51]. We compute the coefficients $\{\alpha_i\}$ and $\{\beta_i\}$ via the discretized Stieltjes procedure [41] where integrals in (3.30) are approximated by quadrature.

Testing for both the sparse grid collocation method and the stochastic Galerkin

method was performed using the truncated Gaussian PDF and Rys polynomials for several values of σ . The linear solver in all cases was stopped when $\frac{\|r_k\|_2}{\|b\|_2} < 10^{-12}$, where $r_k = b - Ax_k$ is the linear residual and A and b are the coefficient matrix and right-hand side, respectively. We constructed the sparse grids using the *Dakota* software package [10].

Table 3.2 reports $\|\langle e_p \rangle\|_{l_\infty}$, the discrete l_∞ -norm of the mean error $\langle e_p \rangle$ evaluated on the spatial grid points. For problems in one random variable, such as this one, the stochastic collocation and stochastic Galerkin methods produce identical results. Table 3.3 shows the average number of iterations required by each deterministic sub-problem as a function of grid level and σ . Problems to the right of the double line do not satisfy (3.28) and some of the associated systems will be indefinite for a high enough grid level as some of the collocation points will be placed in the region of ill-posedness. If the solver failed to converge for any of the individual sub-problems, the method is reported as having failed using “DNC”.

Level/p	σ				
	1	2	3	4	5
1	0.1856	0.1971	0.2175	0.2466	0.2807
2	0.0737	0.0811	.0932	0.1095	0.1207
3	0.0245	.0279	.0331	0.0389	0.1195
4	0.0070	.0082	.0099	0.0121	DNC
5	0.0017	0.0021	.0026	0.0029	DNC
6	3.7199e-4	4.6301e-4	5.7900e-4	6.7702e-4	DNC
7	7.2002e-5	9.1970e-5	1.1605e-4	4.1598e-4	DNC

Table 3.2: Mean error in the discrete l_∞ -norm for the stochastic collocation and stochastic Galerkin methods.

Level	σ					p	σ				
	1	2	3	4	5		1	2	3	4	5
1	10	10	10.5	11	11	1	13	15	16	18	21
2	10	10.33	10.67	11.33	12.67	2	13	17	22	28	38
3	10	10.5	11	12.25	22	3	14	19	26	39	140
4	10	10.6	11.2	13	DNC	4	14	20	29	53	DNC
5	10.17	10.5	11.33	13.83	DNC	5	14	21	31	69	DNC
6	10.14	10.43	11.43	15	DNC	6	15	21	33	94	DNC
7	10.13	10.63	11.38	16.75	DNC	7	15	21	34	136	DNC

Table 3.3: Iterations for the stochastic collocation (left) and stochastic Galerkin methods (right)

Table 3.3 shows the PCG iteration counts for both methods. Again, problems to the right of the double line are ill-posed and the Galerkin linear system as well as a subset of the individual collocation systems are guaranteed to become indefinite as the degree of polynomial approximation p (for stochastic Galerkin) or sparse grid level (for collocation) increases [14]. Table 3.3 shows that the iteration counts are fairly well behaved when mean-based preconditioning is used. In general, iterations grow as the degree of polynomial approximation increases.

It is well known that bounds on convergence of the conjugate gradient method are determined by the condition number of the matrix. It is shown in [14] that if the diffusion coefficient is given by a stationary field, as in (3.26), then the eigenvalues of the preconditioned stochastic Galerkin system lie in the interval $[1 - \tau, 1 + \tau]$

where

$$\tau = C_{p+1}^{max} \frac{\sigma}{\mu} \left(\sum_{k=1}^M \sqrt{\lambda_k} \|a_k(\mathbf{x})\|_{L_\infty} \right) \quad (3.31)$$

and C_{p+1}^{max} is the magnitude of the largest zero of the degree $p+1$ orthogonal polynomial. Therefore the condition number is bounded by $\kappa(A) \leq \frac{1+\tau}{1-\tau}$. It is possible to bound the eigenvalues of a single system arising in collocation in a similar manner using the relation (3.21). The eigenvalues of the system arising from sampling (3.21) at $\boldsymbol{\xi}$ lie in the bounded interval $[1 - \tilde{\tau}(\boldsymbol{\xi}), 1 + \tilde{\tau}(\boldsymbol{\xi})]$ where

$$\tilde{\tau}(\boldsymbol{\xi}) = \frac{\sigma}{\mu} \left(\sum_{k=1}^M \sqrt{\lambda_k} \|a_l(\mathbf{x})\|_{L_\infty} |\xi_k| \right). \quad (3.32)$$

Likewise the condition number for a given preconditioned collocation system can be bounded by $\kappa(A(\boldsymbol{\xi})) \leq \frac{1+\tilde{\tau}}{1-\tilde{\tau}}$. For both methods, as σ increases relative to μ the associated systems may become ill-conditioned and will eventually become indefinite. Likewise as p or the sparse grid level increases, C_{p+1}^{max} and $\max_{\Theta_{p,M}} |\boldsymbol{\xi}|$ increase and the problems may again become indefinite. However if Γ is bounded then both C_{p+1}^{max} and $\max_{\Theta_{p,M}} |\boldsymbol{\xi}|$ are bounded for all choices of p and the sparse grid level and the systems are guaranteed to remain positive definite provided σ is not too large.

The effect of these bounds can be seen in the above examples since as σ increases the iteration counts for both methods increase until finally for large choices of σ and large p or grid level the PCG iteration fails to converge. It should also be noted that by (3.31) and (3.32), the condition number of the entire stochastic

Galerkin system depends on the largest zero of the degree $p + 1$ orthogonal polynomial, whereas the condition number of each collocation system depends only on the particular collocation point. This accounts for the larger iteration counts for the stochastic Galerkin method since for most collocation systems the condition number bounded closer to one. However, for smaller values of σ the PCG iteration converges in a reasonable number of iterations for all tested values of p and grid level.

3.4.2 Computational Cost Comparison

In this section we compare the performance of the two methods using both the model developed above and the implementations in Trilinos. For our numerical examples, we consider a problem where only the covariance of the diffusion field is given. We consider two problems of the form

$$-\nabla \cdot [(\mu + \sigma \sum_{k=1}^M \sqrt{\lambda_k} \xi_k f_k(x)) \nabla u] = 1 \quad (3.33)$$

where values of M between 3 and 15 are explored and $\{\lambda_k, f_k\}$ are the eigenpairs associated with the covariance kernel

$$C(\mathbf{x}_1, \mathbf{x}_2) = \exp(-|x_1 - x_2| - |y_1 - y_2|). \quad (3.34)$$

The KL-expansion of this kernel is investigated extensively in [19]. For the first problem, the random variables $\{\xi_k\}$ are chosen to be identically independently distributed uniform random variables on $[-1, 1]$. For the second problem, the random

variables $\{\xi_k\}$ are chosen to be identically independently distributed truncated Gaussian random variables as in the previous section. For the first, problem $\mu = .2$ and $\sigma = .1$. For the second problem, $\mu = 1$ and $\sigma = .25$. These parameters were chosen to ensure that the problem remains well posed. Table 3.4 shows approximate values for τ for both of the above problems. In the second case, where truncated Gaussian random variables are used, $1 - \tau$ becomes close to zero as the stochastic dimension of the problem increases. Thus this problem could be said to be nearly ill-posed. In terms of computational effort this should favor the sparse grid collocation method since, as was seen in the previous section, iteration counts for the stochastic Galerkin method increased faster than those for the collocation method as the problem approaches ill-posedness. The spatial domain is discretized by a uniform mesh with discretization parameter $h = \frac{1}{32}$. Note that the mean-based preconditioning eliminates the dependence on h of the conditioning of the problem [14] so we consider just a single value of the spatial mesh parameter.

M	Uniform Random Variables $\Gamma_i = [-1, 1], \sigma = .1, \mu = .2$	Truncated Gaussian Random Variables $\Gamma_i = [-2.576, 2.576], \sigma = .25, \mu = 1$
3	0.533	0.686
4	0.549	0.708
5	0.566	0.729

Table 3.4: Approximate values of τ for model problems

Approximate solutions are used to measure the error since there is no analytic expression for the exact solution to either of the above problems. To measure the error for the Galerkin method the exact solution is approximated by a high order

($p = 10$) Galerkin scheme. For the collocation method we take the solution from a level-10 sparse grid approximation as an approximation to the exact solution. These two “nearly exact” solutions differed by an amount on the order of the machine precision. The error in the stochastic space is then estimated by computing the mean and variance of the approximate solutions and comparing it to the mean and variance of the order-10 (level-10) approximations. The linear solves for both methods stop when $\frac{\|r_k\|_2}{\|b\|_2} < 10^{-12}$. In measuring the time, setup costs are ignored. The times reported are non-dimensionalized by the time required to perform a single deterministic matrix vector product and compared with the model developed above.

Figure 3.2 explores the accuracy obtained for the two discretizations for $M = 4$; the behavior was the same for $M = 3$ and $M = 5$. In particular, it can be seen that for both sample problems, the same value of p (corresponding to the polynomial space for the Galerkin method and the sparse grid level for the collocation method) the two methods produce solutions of comparable accuracy. Thus the Galerkin method gives higher accuracy per stochastic degree of freedom. Since the unknowns in the Galerkin scheme are coupled, the cost per degree of freedom will be higher. In terms of computational effort then the question is whether or not the additional accuracy per degree of freedom will be worth the additional cost.

Figures 3.3 and 3.4 compare the costs incurred by the two methods, measured in CPU time, for obtaining solutions of comparable accuracy. The timings reflect time spent to execute the methods on an Intel Core 2 Duo machine running at 3.66GHz with 6Gb of RAM. In the figures these timings are non-dimensionalized by dividing by the cost of a single sparse matrix-vector product with the (five-diagonal)

nonzero structure of $\{A_k\}$. This cost is measured by dividing the total time used by the collocation method for matrix-vector products by the total number of CG iterations performed in the collocation method. This allows the times to be compared to the cost model (3.22) and (3.23), which in turn helps ensure that the implementations are of comparable efficiency. The model is somewhat less accurate for the collocation method, because for these relatively low-dimensional models the approximation $|\Theta_{p,M}| = 2^p N_\xi$ is an overestimate. For the values of M used for these results ($M = 3, 4$, and 5), it can be seen that the Galerkin method requires less CPU time than the collocation method to compute solutions of comparable accuracy, and that the gap widens as the dimension of the space of random variables increases. Also, it is seen in Figure 3.3 and Figure 3.4 that the performance of each method is largely independent of the density functions used in defining the random variables ξ_k .

Table 3.5 expands on these results for larger values of M , based on our expectation that the same value of p (again, corresponding to the polynomial space for the Galerkin method or the level for the collocation method) yields solutions of comparable accuracy. The trends are comparable for all M and show that as the size of the approximation space increases, the overhead for collocation associated with the increased number of degrees of freedom becomes more significant.

3.5 Conclusion

In this chapter we have examined the costs of solving the linear systems of equations arising when either the stochastic Galerkin method or the stochastic col-

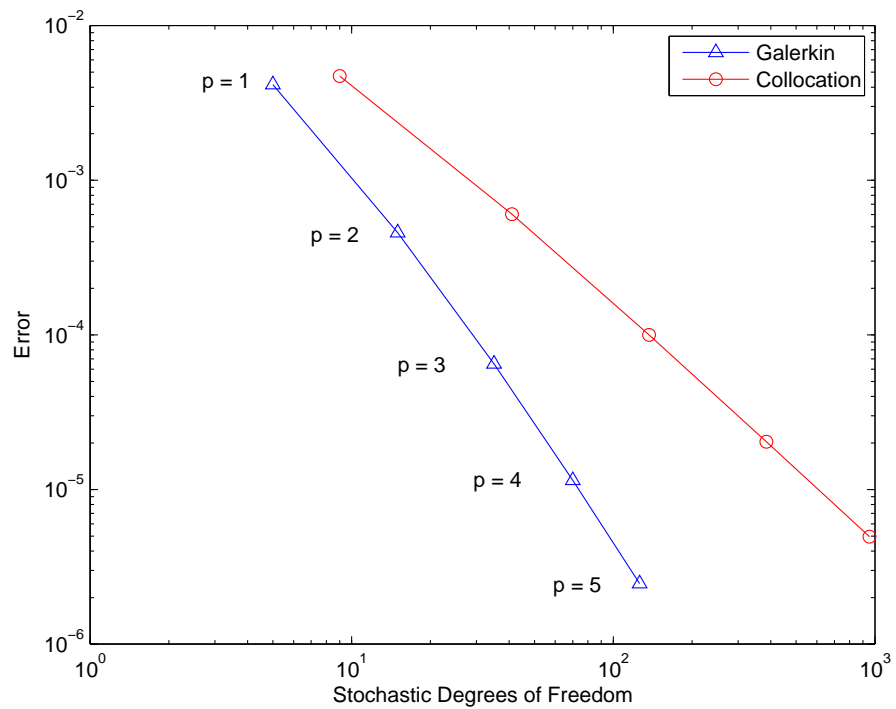
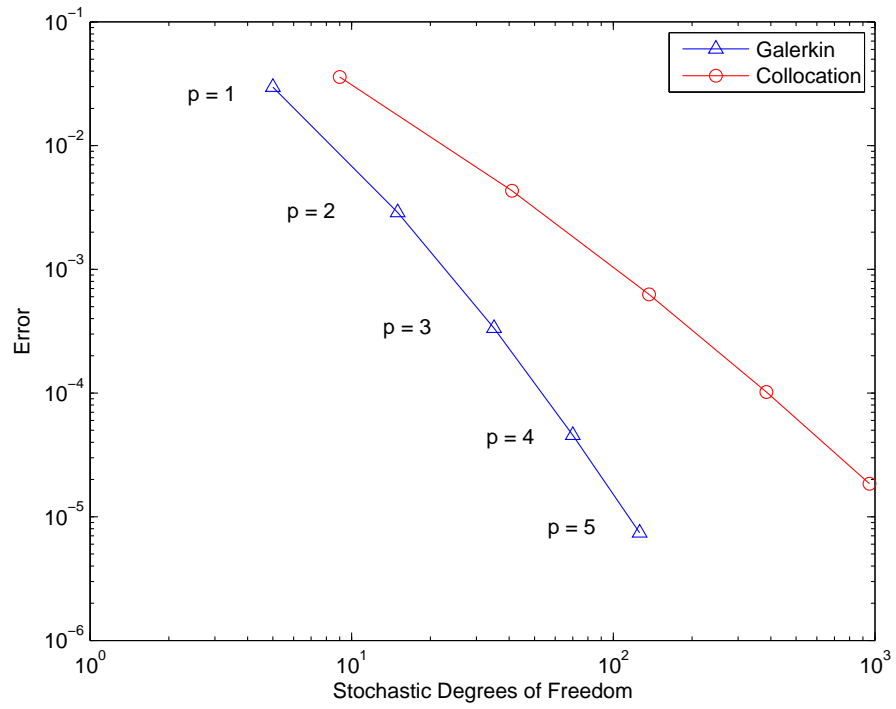


Figure 3.2: Errors vs stochastic DOF for $M = 4$. Uniform random variables (top), truncated Gaussian random variables (bottom)

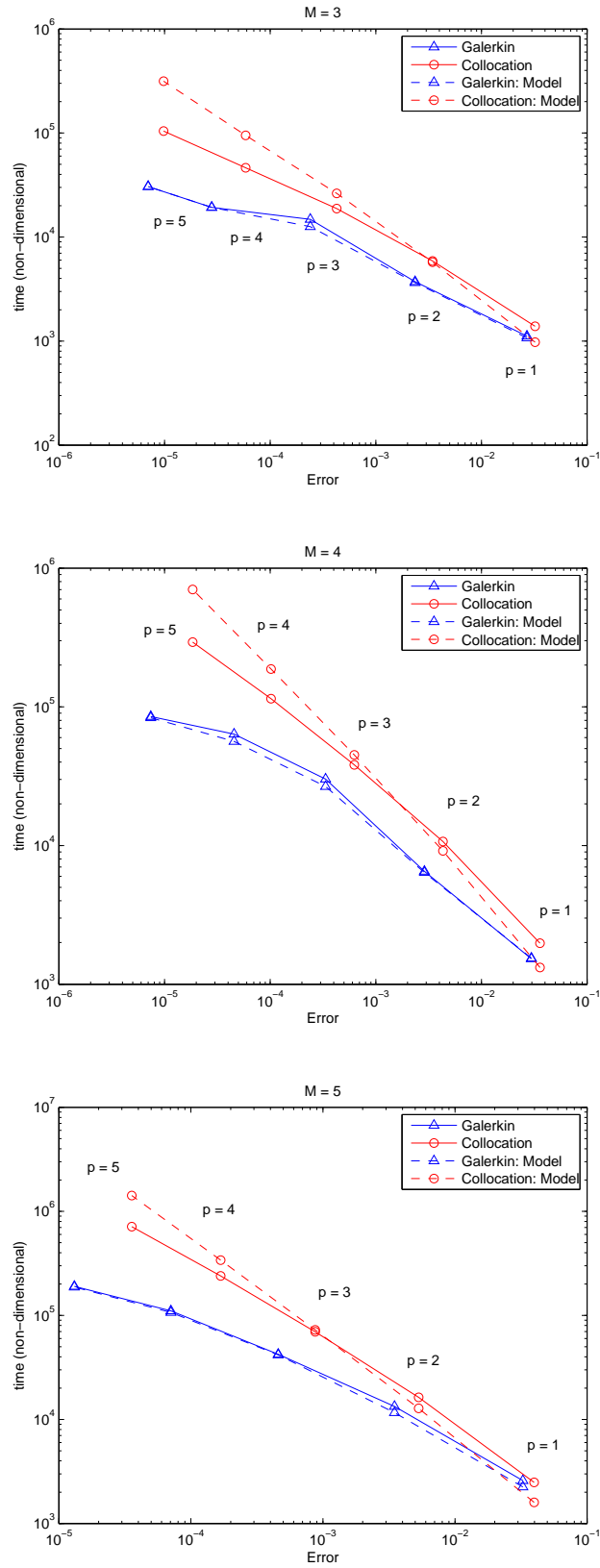


Figure 3.3: Solution time vs. error for $M = 3, 4, 5$. Uniform random variables.

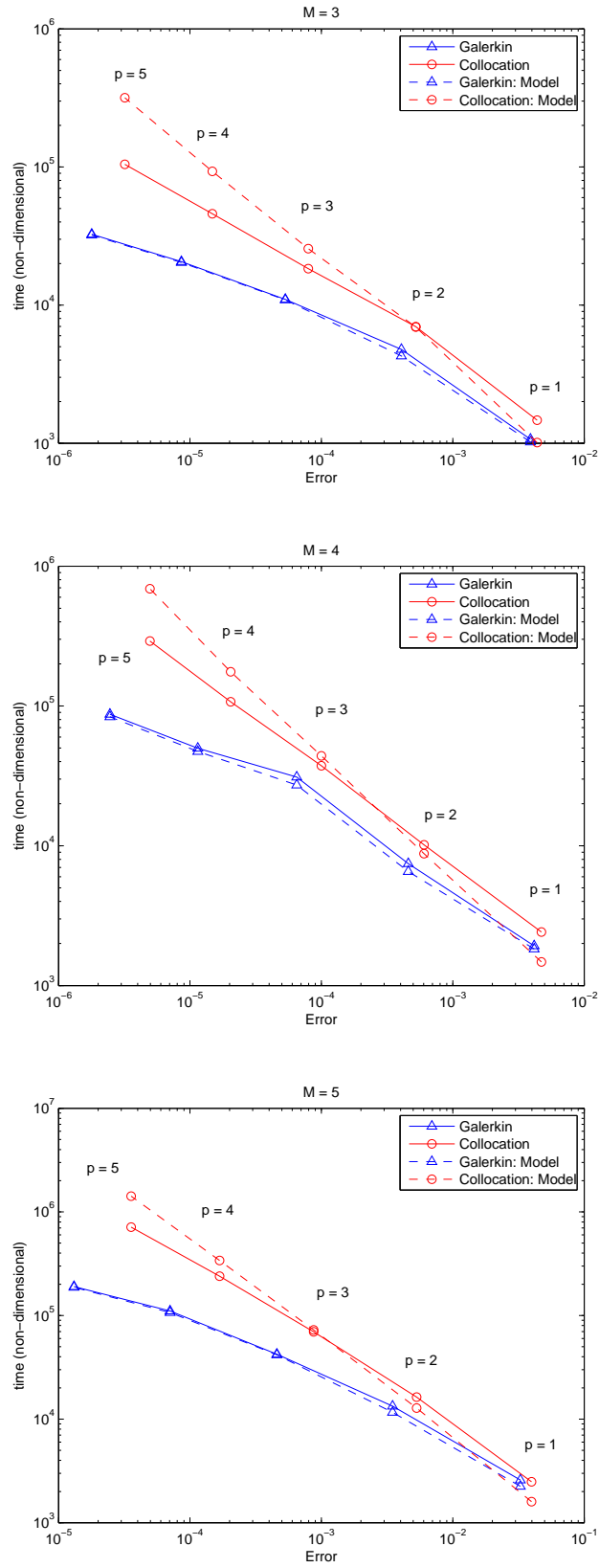


Figure 3.4: Solution time vs. error for $M = 3, 4, 5$. Truncated Gaussian random variables.

	Stochastic Galerkin		
	$M = 5$	$M = 10$	$M = 15$
$p = 1$	0.058139 (0.026912)	0.147306 (0.051521)	0.320443 (0.085775)
2	0.269301 (0.119066)	1.20465 (0.0385744)	3.80461 (1.04111)
3	1.20353 (0.372013)	13.1382 (2.57246)	51.448 (7.40171)
4	3.50061 (1.1846)	53.786 (10.1633)	168.112 (41.325)
5	6.510255 (2.89493)	117.729 (36.2012)	
	Sparse Grid Collocation		
$Level = 1$	0.068934 (0.036288)	0.163258 (0.078107)	0.285779 (0.123893)
2	0.532407 (0.275829)	2.13126 (0.98289)	5.07825 (2.1247)
3	2.41468 (1.20969)	16.9871 (7.54744)	57.9837 (23.1414)
4	8.31068 (4.14521)	102.595 (44.0484)	493.042 (193.199)
5	24.5645 (12.0362)	515.751 (221.546)	

Table 3.5: Solution (preconditioning) time in seconds for second model problem

location method is used to discretize the diffusion equation in which the diffusion coefficient is a random field modeled by (3.2). The results indicate that when mean-based preconditioners are coupled with the conjugate gradient method to solve the systems that arise, the stochastic Galerkin method is quite competitive with collocation. Indeed, the costs of the Galerkin method are typically lower than for collocation, and this differential becomes more pronounced as the number of terms

in the truncated KL expansion increases. We have also developed a cost model for both methods that closely mirrors the complexity of the algorithms.

Chapter 4

The Adaptive KDE Collocation Method

The methods discussed in chapter 3 all exhibit asymptotic exponential convergence with respect to the stochastic discretization parameter. This is possible because Lemma 1 guarantees that the solution to the stochastic diffusion equation is analytic with respect to the parameters when the diffusion coefficient is expanded in a truncated KL expansion. It is possible in the general case however for the solution u to be discontinuous with respect to the parameters, or to have steep gradients. In fact even when the solution is analytic it may have local features that make global approximation impossible without using very high order polynomial spaces or very high level sparse grids.

An additional weakness of the methods discussed in chapter 2 derives from the assumption that the joint density of the random parameters is known and that the parameters are independent. Both of these assumptions are very strong and may not be true in practice. In particular it may be the case that one only has access to a finite sample set from the parameter space and that the individual parameters are not independent of one another. In this chapter we will present an extension of an adaptive collocation method developed in [30] which is capable of approximating the solution to parameter-dependent functions that are discontinuous or have steep gradients. The extension of this method uses KDE to approximate the unknown dis-

tribution of the parameters and can efficiently recover the statistics of the solution when only a sample from the parameter space is available.

4.1 Problem Statement

In this chapter we will concern ourselves with the general case of approximating the moments of a function $u : D \times \Omega \rightarrow \mathbb{R}$ where D is the spatial domain and Ω is an abstract event space and u is the solution to an SPDE of the form (2.1). As discussed previously, it is necessary to first represent u in terms of a finite number of random variables $\boldsymbol{\xi} = [\xi_1, \xi_2, \dots, \xi_M]^T$. In this chapter we only make the assumption that the model has been parametrized by some reasonable process, not necessarily using a KL-expansion. If we denote $\Gamma = \text{Image}(\boldsymbol{\xi})$, then we can write (2.1) as

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \boldsymbol{\xi}; u) &= f(\mathbf{x}, \boldsymbol{\xi}), & \forall \mathbf{x} \in D, \boldsymbol{\xi} \in \Gamma \\ \mathcal{B}(\mathbf{x}, \boldsymbol{\xi}; u) &= g(\mathbf{x}, \boldsymbol{\xi}), & \forall \mathbf{x} \in \partial D, \boldsymbol{\xi} \in \Gamma. \end{aligned} \tag{4.1}$$

We will assume that for a given realization of the random vector $\boldsymbol{\xi} \in \Gamma$, the system (4.1) is a well-posed deterministic partial differential equation that can be solved using a deterministic solver.

One is typically interested in methods that allow statistical properties of u to be computed. If $\rho(\boldsymbol{\xi})$ denotes the joint probability density function of the random

vector $\boldsymbol{\xi}$, then the k^{th} moment of the solution u is defined as

$$\mathbb{E}(u^k) = \int_{\Gamma} u^k \rho(\boldsymbol{\xi}) d\boldsymbol{\xi}. \quad (4.2)$$

One may also be interested in computing probability distributions associated with u , for example $P(u(\mathbf{x}, \boldsymbol{\xi}) \geq c)$.

Several methods have been developed for computing approximations to the random field u and the associated statistical quantities. The most widely known is the Monte Carlo method, where the desired statistics are obtained by repeatedly sampling the distribution of $\boldsymbol{\xi}$, solving each of the resulting deterministic PDEs, and then estimating the desired quantities by averaging. In chapter 3 we discussed the stochastic Galerkin and stochastic sparse grid collocation methods. These methods typically approximate the solution u as a high-degree multivariate polynomial in $\boldsymbol{\xi}$. If this approximation is denoted $u_p(\mathbf{x}, \boldsymbol{\xi})$, then the error $u - u_p$ can be measured in terms of an augmented Sobolev norm

$$\|\cdot\|_{L_P^2 \otimes V} = \left(\int_{\Omega} \|\cdot\|_V^2 dP \right)^{\frac{1}{2}}. \quad (4.3)$$

Here V is an appropriate Sobolev space that depends on the spatial component of the problem and $\|\cdot\|_V$ is the norm over this space. Once u_p has been constructed, if the joint density function is known, then the k^{th} moment of the solution can be

computed as

$$\mathbb{E}(u^k) \approx \int_{\Gamma} u_p^k \rho \, d\boldsymbol{\xi}, \quad (4.4)$$

by using quadrature.

From (2.57) and Lemma 3 we know that as the total degree of the polynomial approximation is increased, the error in the above norm, $\|u - u_p\|_{L_p^2; V}$, decays very rapidly provided that the solution u is sufficiently smooth in $\boldsymbol{\xi}$. If u is not sufficiently smooth then the convergence of these methods can stall or they may not converge at all [30]. Several methods have been proposed for treating problems that are discontinuous in the stochastic space. One approach partitions the stochastic space into elements and approximates the solution locally within elements by polynomials, continuous on the domain [3, 47]. This is analogous to hp -methods from the theory of finite elements. Another approach is to use a hierarchical basis method developed in [27], which approximates u using a hierarchical basis of piecewise linear functions defined on a sparse grid. This idea was used with stochastic collocation in [30] where the sparse grid is refined adaptively using an a posteriori error estimator.

If the truncated Karhunen-Loève expansion is used to express \mathcal{L} and \mathcal{B} , then the random variables $\xi_1, \xi_2, \dots, \xi_M$ have zero mean and are uncorrelated. It is frequently assumed that the random variables are independent and that their marginal density functions $\rho_i(\xi_i)$ are known explicitly. In this case the joint density function is simply the product of the marginal densities $\rho(\boldsymbol{\xi}) = \prod_{k=1}^M \rho_k(\xi_k)$. This assumption simplifies the evaluation of the moments of the solution since the multidimensional

integral in (4.2) can be written as the product of one-dimensional integrals. It is not the case, however, that uncorrelated random variables are necessarily independent, and in the worst case the support of the product of the marginal densities may contain points that are not in the support of the true joint density. Thus, it may not be appropriate to define the joint density function as the product of the marginal density functions. See [22] for further discussion of this point.

In this chapter we explore a method for approximating the statistics of the solution u when an explicit form of the joint distribution is not available and we only have access to a finite number of samples of the random vector $\boldsymbol{\xi}$. In particular, we are able to treat the case where information on the parameters of the problem is only available in the form of experimental data. The method works by constructing an approximation $\hat{\rho}(\boldsymbol{\xi})$ to the joint probability distribution $\rho(\boldsymbol{\xi})$ using kernel density estimations [44]. This construction is then combined with an adaptive collocation strategy similar to the one derived in [30] to compute an approximation to the random field u . This technique ensures that the approximation error is small near the sample points. Moments can then be efficiently evaluated by performing surrogate Monte Carlo on this approximation to the solution. That is, if the approximate solution is denoted $\mathcal{A}(u)$, and $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$ is a set of samples independently drawn from the distribution of $\boldsymbol{\xi}$, then the expected value is approximated by

$$\mathbb{E}[u] \approx \frac{1}{N} \sum_{i=1}^N \mathcal{A}(u)(\boldsymbol{\xi}^{(i)}). \quad (4.5)$$

The remainder of this chapter proceeds as follows. Section 4.2 discusses the adaptive collocation method in [30]. Section 4.3 presents an overview of the kernel density estimation technique used for approximating the unknown distribution of $\boldsymbol{\xi}$. Section 4.4 presents the method developed in this paper for approximating solutions to problems of the form (4.1). An error bound for the method is given in Section 4.4.1, and Section 4.4.2 presents techniques for extracting solution statistics. Section 4.5 presents the results of numerical experiments showing the performance of the new method and comparing this performance with that of the Monte Carlo method. Finally in Section 4.6 we draw some conclusions.

4.2 The Adaptive Collocation Method

Collocation methods work by solving the equation (4.1) for a finite number of pre-determined parameters $\{\boldsymbol{\xi}^{(1)}, \dots, \boldsymbol{\xi}^{(N_c)}\}$ using a suitable deterministic solver. The solutions at each sample point are then used to construct an interpolant to the solution for arbitrary choices of the random vector $\boldsymbol{\xi}$. We denote such an approximation generally as $\mathcal{A}(u)(\boldsymbol{\xi})$. In the collocation methods discussed in chapters 2 and 3, the solution random field is approximated globally by a multivariate polynomial in the random vector $\boldsymbol{\xi}$. These methods are therefore only useful when the random field u is sufficiently regular in $\boldsymbol{\xi}$.

An adaptive collocation method was developed in [30]. This method is designed to compute approximations of random fields that possess discontinuities or

strong gradients, and for which the image set Γ is bounded.¹ In the following, we present an overview of this method and our proposed modifications. To simplify the presentation we describe the case of a function u defined by a single random parameter whose image is a subset of $[0, 1]$. This can be generalized in a straightforward manner to a function defined by M parameters with image contained in any M -dimensional hypercube. Define

$$m_i = \begin{cases} 1 & \text{if } i = 1, \\ 2^{i-1} + 1 & \text{if } i > 1, \end{cases} \quad (4.6)$$

$$\xi_j^i = \begin{cases} \frac{j-1}{m_i-1} & \text{for } j = 1, \dots, m_i, \text{ if } m_i > 1, \\ 0.5 & \text{for } j = 1, \text{ if } m_i = 1. \end{cases} \quad (4.7)$$

For $i = 1, 2, \dots$, we have that $\theta^i = \{\xi_j^i\}_{j=1}^{m_i}$ consists of m_i distinct equally spaced points on $[0, 1]$. We also have that $\theta^i \subset \theta^{i+1}$. Since these points are equidistant, the use of global polynomial interpolation as in [50] is not appropriate due to the Runge phenomenon. We make the assumption that the solution u is almost surely Lipschitz continuous with respect to the random parameters. This is a substantially weaker assumption than assuming that the solution is analytic. For example, the solution may contain singularities that global polynomial approximations will not resolve. To address these issues, a hierarchical basis of piecewise linear functions is used to construct the interpolant. Define $\theta^0 = \emptyset$ and $\Delta\theta^i = \theta^i \setminus \theta^{i-1}$. Note that $|\Delta\theta^i| = m_i - m_{i-1}$. Let the members of $\Delta\theta^i$ be denoted $\{\xi_j^{\Delta i}\}_{j=0}^{|\Delta\theta^i|-1}$. The

¹For unbounded Γ , interpolation is carried out on a bounded subset of Γ , see e.g. [48].

hierarchical basis is defined on the interval $[0, 1]$ as

$$a_0^1(\xi) = 1 \tag{4.8}$$

$$a_j^i(\xi) = \begin{cases} 1 - (m_i - 1)|\xi - \xi_j^{\Delta^i}| & \text{if } |\xi - \xi_j^{\Delta^i}| < 1/(m_i - 1), \\ 0 & \text{otherwise,} \end{cases} \tag{4.9}$$

for $i > 1$ and $j = 0, \dots, |\Delta\theta^i| - 1$. These functions are piecewise linear and have

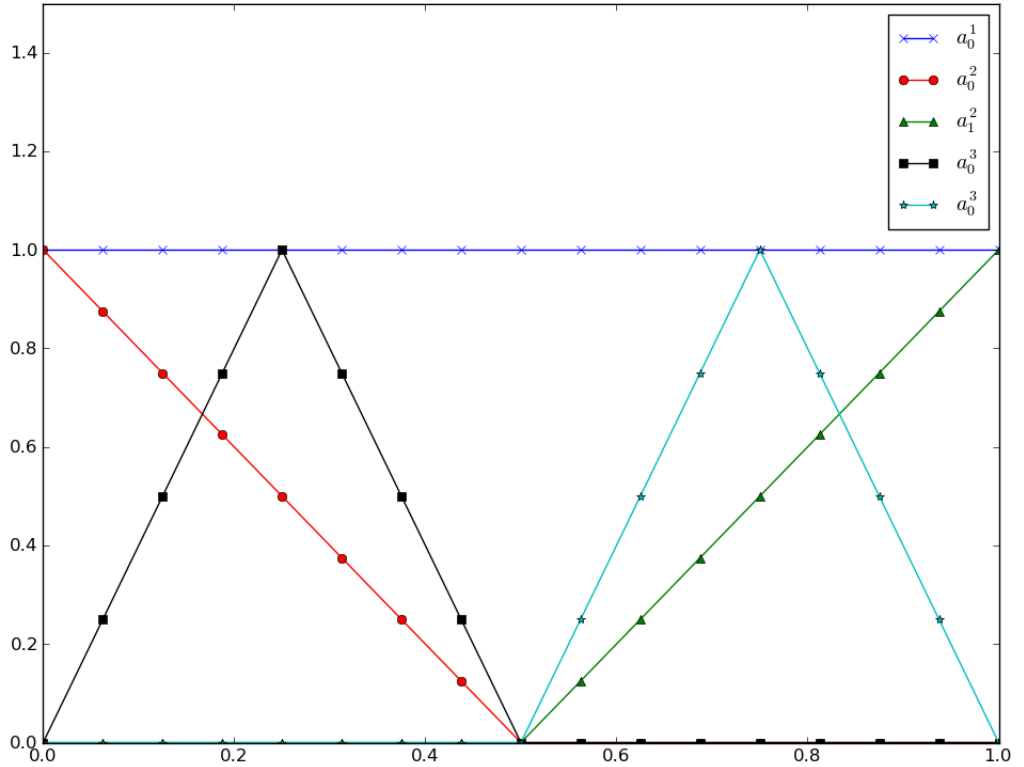


Figure 4.1: The hierarchical basis functions for $i = 1, 2, 3$.

the property that $a_j^i(\xi_k^{\Delta^i}) = \delta_{jk}$, and $a_j^i(\xi_k^s) = 0$ for all $s < i$. Note that there is a binary tree structure on the nodes in θ^i . That is, we can define the set of children

of a point $\xi_j^{\Delta^i}$ as

$$child(\xi_j^{\Delta^i}) = \begin{cases} \{\xi_j^{\Delta^{i+1}}\} & \text{if } i = 2 \\ \{\xi_{2j}^{\Delta^{i+1}}, \xi_{2j+1}^{\Delta^{i+1}}\} & \text{otherwise.} \end{cases} \quad (4.10)$$

We also denote the parent of a point in this tree as $par(\xi_j^{\Delta^i})$.

Algorithm 1 defines an interpolation scheme using the hierarchical basis functions. The quantities $\{w_j^k\}$ are referred to as the hierarchical surplus. They represent

Algorithm 1 Interpolation With Hierarchical Basis Functions

Define $\mathcal{A}_0(u)(\xi) = 0$.

Define $k = 1$

repeat

 Construct $\Delta\theta^k$

 Evaluate $u(\xi_j^{\Delta^k}) \forall \xi_j^{\Delta^k} \in \Delta\theta^k$

$w_j^k = u(\xi_j^{\Delta^k}) - \mathcal{A}_{k-1}(u)(\xi_j^{\Delta^k}) \forall \xi_j^{\Delta^k} \in \Delta\theta^k$

 Define $\mathcal{A}_k(u)(\xi) = \sum_{i=1}^k \sum_{j=0}^{|\Delta\theta^i|-1} w_j^i a_j^i(\xi)$.

$k = k + 1$

until $max(|w_j^{k-1}|) < \tau$

the correction to the interpolant $\mathcal{A}_{k-1}(u)$ at the points in $\Delta\theta_k$. For functions with values that vary dramatically at neighboring points, the hierarchical surpluses $\{w_j^i\}$ remain large for several iterations. This provides us with a natural error indicator as well as a convergence criterion for the method, whereby we require that the largest hierarchical surplus be smaller than a given tolerance. The hierarchical surpluses also provide a mechanism to implement adaptive grid refinement. The grid is adaptively refined at points with large hierarchical surpluses. For such a point, its children are added to the next level of the grid. Algorithm 2 defines such an adaptive interpolation algorithm that is similar to the one appearing in [30]. The

Algorithm 2 Adaptive Interpolation With Hierarchical Basis Functions

Define $\mathcal{A}_0(u)(\xi) = 0$.
 Define $k = 1$
 Initialize $\Delta\theta_{adaptive}^1 = \theta^1$.
repeat
 $\Delta\theta_{adaptive}^{k+1} = \emptyset$
 for $\xi_j^{\Delta k} \in \Delta\theta_{adaptive}^k$ **do**
 Evaluate $u(\xi_j^{\Delta k})$
 $w_j^k = u(\xi_j^{\Delta k}) - \mathcal{A}_{k-1}(u)(\xi_j^{\Delta k})$
 if $\|w_j^k\| > \tau$ **then**
 $\Delta\theta_{adaptive}^{k+1} = \Delta\theta_{adaptive}^{k+1} \cup \text{child}(\xi_j^{\Delta k})$
 end if
 end for
 Define $\mathcal{A}_k(u)(\xi) = \sum_{i=1}^k \sum_j w_j^i a_j^i(\xi)$.
 $k = k + 1$
until $\max(\|w_j^{k-1}\|) < \tau$

interpolation error associated with this method is shown by numerical experiments in [30] to be significantly smaller than the bound $\mathcal{O}(|\theta^k|^{-2} \log(|\theta^k|^{3(M+1)}))$ presented in [26] for both smooth functions and examples that contain step gradients or discontinuities.

This method can be generalized in a straightforward way to functions defined on $[0, 1]^M$. All that is needed is to define a multidimensional hierarchical basis set and a method for generating the children of a given grid point. The multidimensional hierarchical basis consists of tensor products of the one-dimensional hierarchical basis functions. Given $\mathbf{i} = [i_1, \dots, i_M] \in \mathbb{N}^M$ and $\mathbf{j} = [j_1, \dots, j_M] \in \mathbb{N}^M$, let

$$a_{\mathbf{j}}^{\mathbf{i}}(\boldsymbol{\xi}) = a_{j_1}^{i_1}(\xi_1) \otimes \cdots \otimes a_{j_M}^{i_M}(\xi_M). \quad (4.11)$$

We can define the multidimensional interpolation grids by

$$\theta_1 = [0.5, 0.5, \dots, 0.5]$$

$$child(\boldsymbol{\xi}_j^{\Delta i}) = \{\boldsymbol{\xi} | \exists ! j \in 1, \dots, M \text{ s.t. } [\xi_1, \dots, \xi_{j-1}, par(\xi_j), \xi_{j+1}, \dots, \xi_M] = \boldsymbol{\xi}_j^{\Delta i}\}. \quad (4.12)$$

From this we can see that each grid point has at most $2M$ children.

This method can be used to approximate the solutions to (4.1) by applying a suitable deterministic solver to the equations at collocation points $\boldsymbol{\xi}_j^{\Delta i}$. We can then construct an interpolant of u , $\mathcal{A}_k(u)$ using the formula in Algorithm 2. In principle, the expected value of u can be approximated by

$$\mathbb{E}(u) \approx \int_{\Gamma} \mathcal{A}_k(u) \rho(\boldsymbol{\xi}) d\boldsymbol{\xi} = \sum_{\mathbf{i}} \sum_{\mathbf{j}} w_{\mathbf{j}}^{\mathbf{i}} \int_{\Gamma} a_{\mathbf{j}}^{\mathbf{i}}(\boldsymbol{\xi}) \rho(\boldsymbol{\xi}) d\boldsymbol{\xi}, \quad (4.13)$$

although in the cases under discussion ρ will not be known explicitly. Even in the case where ρ is known explicitly and can be expressed as the product of univariate functions, the integral in (4.13) can still be difficult to calculate when it is of high dimension.

4.3 Kernel Density Estimation

Let $K(\boldsymbol{\xi})$ be a function satisfying the following conditions:

$$\begin{aligned}\int_{\mathbb{R}^M} K(\boldsymbol{\xi}) d\boldsymbol{\xi} &= 1, \\ \int_{\mathbb{R}^M} K(\boldsymbol{\xi}) \boldsymbol{\xi} d\boldsymbol{\xi} &= 0, \\ \int_{\mathbb{R}^M} K(\boldsymbol{\xi}) \|\boldsymbol{\xi}\|^2 d\boldsymbol{\xi} &= k_2 < \infty, \\ K(\boldsymbol{\xi}) &\geq 0,\end{aligned}\tag{4.14}$$

where $\|\boldsymbol{\xi}\|$ is the Euclidean norm of the M -dimensional vector $\boldsymbol{\xi}$. Let $\boldsymbol{\xi}^{(1)}, \boldsymbol{\xi}^{(2)}, \dots, \boldsymbol{\xi}^{(N)}$ be N independent realizations of the random vector $\boldsymbol{\xi}$. The kernel density approximation to the joint distribution of $\boldsymbol{\xi}$ is given by

$$\hat{\rho}(\boldsymbol{\xi}) = \frac{1}{Nh^M} \sum_{k=1}^N K\left(\frac{\boldsymbol{\xi} - \boldsymbol{\xi}^{(i)}}{h}\right),\tag{4.15}$$

where h is a user-defined parameter called the bandwidth. It is straightforward to verify that the function $\hat{\rho}$ defined above satisfies the conditions for being a probability density function. The main challenge here lies in the selection of an appropriate value for h . If h is chosen to be too large then the resulting estimate is said to be oversmoothed and important features of the data may be obscured. If h is chosen to be too small then the resulting estimate is said to be undersmoothed and the approximation may contain many spurious features not present in the true distribution. Figure 4.2 shows kernel density estimates of a bimodal distribution for a small

and large value of h . The oversmoothed estimate does not detect the bimodality of the data whereas the undersmoothed estimate introduces spurious oscillations into the estimate.

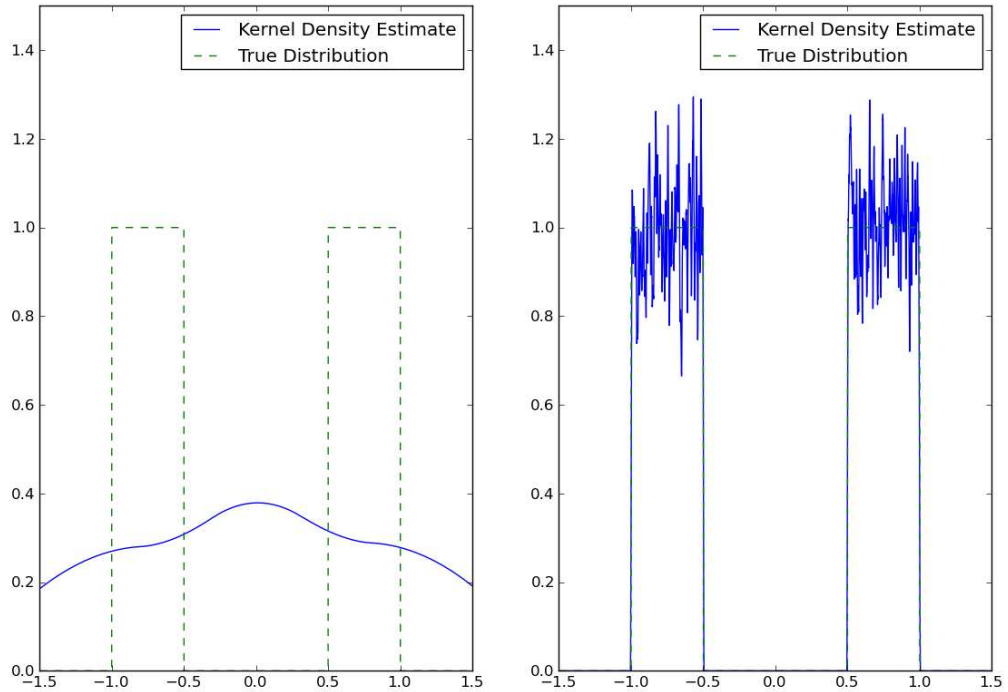


Figure 4.2: Under-smoothed kernel density estimate (left) and over-smoothed (right).

One method for specifying h is to choose the value that minimizes the approximate mean integrated square error (*AMISE*). For a given value of h , the *AMISE* is given by

$$AMISE(h, N) = \frac{1}{4}h^4\alpha^2 \int_{\mathbb{R}^M} (\Delta\rho(\boldsymbol{\xi}))^2 d\boldsymbol{\xi} + N^{-1}h^{-M}\beta, \quad (4.16)$$

where

$$\alpha = \int_{\mathbb{R}^M} \|\boldsymbol{\xi}\|_1^2 K(\boldsymbol{\xi}) d\boldsymbol{\xi}, \quad \beta = \int_{\mathbb{R}^M} K(\boldsymbol{\xi})^2 d\boldsymbol{\xi}, \quad (4.17)$$

and Δ here denotes the Laplace operator [44]. From this expression the optimal value of h can be derived as

$$h_{opt}^{M+4} = M\beta\alpha^{-2} \left\{ \int (\Delta\rho(\boldsymbol{\xi}))^2 d\boldsymbol{\xi} \right\}^{-1} N^{-1}. \quad (4.18)$$

It can be shown that the optimal bandwidth is of magnitude $\mathcal{O}(N^{-1/(M+4)})$ as the number of samples N increases. If the optimal value of h is used it can also be shown that the *AMISE* decays like $\mathcal{O}(N^{-\frac{4}{4+M}})$.

For numerical computations, choosing h to minimize the *AMISE* is impractical since it requires a priori knowledge of the exact distribution. Many techniques have been proposed for choosing the smoothing parameter h without a priori knowledge of the underlying distribution, including least-squares cross-validation and maximum likelihood cross-validation [44]. In the numerical experiments below we employ maximum likelihood cross-validation (MLCV). This method proceeds as follows. Given a finite set of samples, $\boldsymbol{\xi}^{(1)}, \boldsymbol{\xi}^{(2)}, \dots, \boldsymbol{\xi}^{(N)}$, of the random vector $\boldsymbol{\xi}$, define

$$\hat{\rho}_{-i}(\boldsymbol{\xi}) = \frac{1}{Nh^M} \sum_{k=1, k \neq i}^N K\left(\frac{\boldsymbol{\xi} - \boldsymbol{\xi}^{(k)}}{h}\right) \quad (4.19)$$

to be the kernel density estimate constructed by omitting the i^{th} sample. The maximum likelihood cross-validation method is to choose h that maximizes

$$CV(h) \equiv \frac{1}{N} \sum_{i=1}^N \log(\hat{\rho}_{-i}(\boldsymbol{\xi}^{(i)})). \quad (4.20)$$

Note that this value of h only depends on the data. The intuition behind this method is that if we are given an approximation to the true density based on $N - 1$ samples and we draw another sample, then the approximate density should be large at this new sample point. In the numerical experiments described below, we solved this optimization problem using the constrained optimization by linear approximation (COBYLA) method found in the *nlopt* software library [25]. The asymptotic cost of evaluating (4.20) is $\mathcal{O}(N^2)$. Thus as the number of samples grows large this method can become costly. In this case one typically only uses a randomly selected subset of the samples to evaluate (4.20) [23]. In the numerical experiments described below, we observed that for the sample sizes used, the cost of this optimization was significantly lower than the cost of repeatedly solving the algebraic systems of equations that arise from the spatial discretization of the PDE (4.1).

In [44] it is shown that the choice of kernel does not have a strong effect on the error associated with kernel density estimation. In our experiments we use the multivariate Epanechnikov kernel

$$K(\boldsymbol{\xi}) = \left(\frac{3}{4}\right)^M \prod_{i=1}^M (1 - \xi_i^2) \mathbf{1}_{\{-1 \leq \xi_i \leq 1\}}. \quad (4.21)$$

This kernel is frequently used in the case of univariate data as it minimizes the asymptotic mean integrated square error over all choices of kernels satisfying (4.14). It also has the advantage that it is compactly supported. This causes the approximate density function $\hat{\rho}$ to be compactly supported, which is important in assuring the well-posedness of some stochastic partial differential equations.

4.4 Adaptive Collocation With KDE Driven Grid Refinement

The interpolation method in [30] distributes interpolation nodes so that discontinuities and steep gradients in the solution function are resolved; however the method does not take into account how significant a given interpolation node is to the statistics of the solution function since the refinement process does not depend on ρ . The kernel density estimate described above can also be used to drive refinement of the adaptive sparse grid in Algorithm 2. The algorithm we propose is as follows. First construct an estimate $\hat{\rho}$ to the true density ρ using a finite number of samples $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$. Second, replace the refinement criterion in Algorithm 2 with

$$|w_j^k| \hat{\rho}(\boldsymbol{\xi}_j^{\Delta^k}) > \tau. \quad (4.22)$$

A similar approach is used in [31] to drive the refinement. However in that study it is again assumed that one has access to an explicit form of the joint density function. With the refinement criterion (4.22), the grid is only adaptively refined at points near the data $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$ since the kernel density estimate is only supported near the samples. In the sequel we refer to this proposed method, i.e., Algorithm

2 with refinement criterion (4.22), as *adaptive KDE collocation*. The remainder of this section is divided into two parts. In Section 4.4.1 we present interpolation error estimates associated with adaptive KDE collocation and in Section 4.4.2 we present methods for approximating the solution statistics of the random field u . Note that throughout this discussion we can ignore the spatial component of the problem.

4.4.1 Error analysis of adaptive KDE collocation

For simplicity we present the results for the case where the problem only depends on a single parameter and that interpolation is carried out on $[0, 1]$. Extension of the argument to multi-parameter problems defined on an arbitrary hypercube is straightforward. Also we ignore the spatial component of the problem as it has no effect on the discussion of the errors resulting from the discretization of the stochastic portion of the problem. Assume that $\mathcal{A}_k(u)$ is an interpolant generated using adaptive KDE collocation with tolerance τ . Let $\hat{\rho}$ be the kernel density estimate used in computing \mathcal{A}_k and let $\hat{\Gamma}$ be the support of $\hat{\rho}$. Let $\mathcal{A}_k^{complete}(u)$ be the interpolant constructed by Algorithm 1 with grid points $\Delta\theta^k = \{\xi_j^{\Delta i}\}$ and set of hierarchical surpluses $\{w_j^i\}$ at those grid points. By definition, $\Delta\theta_{adaptive}^k \subset \Delta\theta^k$. Define $\Delta\theta_{remaining}^k = \Delta\theta^k \setminus \Delta\theta_{adaptive}^k$. Then if $\xi_j^{\Delta i} \in \Delta\theta_{remaining}^k$, it follows from (4.22) that $|w_j^i \hat{\rho}(\xi_j^{\Delta i})| \leq \tau$. We can bound the difference between u and $\mathcal{A}_k(u)$ on $\hat{\Gamma}$

as

$$\begin{aligned} \|(u - \mathcal{A}_k(u))\rho\|_{L_\infty(\hat{\Gamma})} &\leq \left\| \frac{\rho}{\hat{\rho}} \right\|_{L_\infty(\hat{\Gamma})} \underbrace{\|(u - \mathcal{A}_k^{complete}(u))\hat{\rho}\|_{L_\infty(\hat{\Gamma})}}_{\epsilon_1} + \\ &\quad \underbrace{\|(\mathcal{A}_k^{complete}(u) - \mathcal{A}_k(u))\hat{\rho}\|_{L_\infty(\hat{\Gamma})}}_{\epsilon_2}. \end{aligned} \quad (4.23)$$

The term ϵ_1 is the interpolation error associated with piecewise multilinear approximation on a full grid. This case is studied in [26]. The interpolation error is bounded by

$$\|u - \mathcal{A}_k^{complete}(u)\|_{L_\infty(\Gamma)} = \mathcal{O}(|\Delta\theta^k|^{-2} |\log_2(|\Delta\theta^k|)|^{3(M-1)}) \quad (4.24)$$

Since $\hat{\rho}$ is bounded it follows that the bound on ϵ_1 decays at the same rate.

Bounding ϵ_2 depends on counting the points in $\Delta\theta_{remaining}^k$ and using the fact that at those points $|w_j^i \hat{\rho}| \leq \tau$. We have that

$$\|(\mathcal{A}_k^{complete}(u) - \mathcal{A}_k(u))\hat{\rho}\|_{L_\infty(\Gamma)} \leq \sum_{\Delta\theta_{remaining}^k} |w_j^i| \|a_j^i(\xi)\hat{\rho}(\xi)\|_{L_\infty(\Gamma)}. \quad (4.25)$$

Expanding $\hat{\rho}$ in a Taylor series around $\xi_j^{\Delta i}$ and noting that $a_j^i(\xi)\hat{\rho}(\xi)$ is only supported on an interval of size $\frac{1}{2^i}$ gives

$$\begin{aligned} \|(\mathcal{A}_k^{complete}(u) - \mathcal{A}_k(u))\hat{\rho}\|_{L^\infty(\Gamma)} &\leq \sum_{\Delta\theta_{remaining}^k} |w_j^i \hat{\rho}(\xi_j^{\Delta i})| + |w_j^i| \frac{\|\hat{\rho}'\|_{L^\infty(\Gamma)}}{2^i} \\ &\leq \tau |\Delta\theta_{remaining}^k| + \sum_{\Delta\theta_{remaining}^k} |w_j^i| \frac{\|\hat{\rho}'\|_{L^\infty(\Gamma)}}{2^i}. \end{aligned} \tag{4.26}$$

The sums here are over all \mathbf{i}, \mathbf{j} such that $\xi_j^{\Delta i} \in \Delta\Theta_{remaining}^k$. For decreasing τ , the number of points in $\Delta\theta_{remaining}^k$ decreases, since more points are locally refined and those points that remain in $\Delta\theta_{remaining}^k$ for large k correspond to basis functions with very small support. If τ is chosen to be small and k is allowed to grow so that the refinement criterion (4.22) is satisfied at every leaf node, the term ϵ_2 will converge to zero.

4.4.2 Estimation of Solution Statistics

Computation of the moments of the solution via the methods presented in [2, 3, 19, 30, 35, 50] all require that the joint density function ρ be explicitly available in order to evaluate the integral $\int_{\Gamma} \hat{u}(\mathbf{x}, \boldsymbol{\xi}) \rho(\boldsymbol{\xi}) d\boldsymbol{\xi}$ where \hat{u} is an approximation to u computed by either the stochastic Galerkin method [3, 19] or by the stochastic collocation method [2, 30, 35, 50]. In practice this may be an unrealistic assumption since we often only have access to a finite sample from the distribution of $\boldsymbol{\xi}$. This section describes two ways of approximating the solution statistics when only a

random sample from the distribution of $\boldsymbol{\xi}$ is available. The first is the well-known Monte Carlo method [33]; the second is a variant of the Monte Carlo predictor method presented in [48], which we have named surrogate based Monte Carlo.

Given a random field $u(x, \boldsymbol{\xi})$ and a finite number of samples $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$, the Monte Carlo method approximates the mean of u by the sample mean

$$\mathbb{E}(u)(\mathbf{x}) \approx \frac{1}{N} \sum_{i=1}^N u(\mathbf{x}, \boldsymbol{\xi}^{(i)}) \equiv \bar{u}(\mathbf{x}). \quad (4.27)$$

This method has the advantage that the convergence is independent of the dimension of the random parameter. The error in the expected value can be approximated by first noting that the estimate is unbiased,

$$Bias_{MC} = \mathbb{E}(u)(\mathbf{x}) - \mathbb{E} \left(\frac{1}{N} \sum_{i=1}^N u(\mathbf{x}, \boldsymbol{\xi}^{(i)}) \right) = 0, \quad (4.28)$$

and that

$$Var(\bar{u}(\mathbf{x})) = \frac{Var(u(\mathbf{x}, \boldsymbol{\xi}))}{N}, \quad (4.29)$$

where $Var(\bar{u}(\mathbf{x}))$ is the variance of the sample mean. An application of Chebyshev's inequality then gives a standard probabilistic estimate, that for $a > 0$,

$$P \left(\left| \mathbb{E}(u)(\mathbf{x}) - \frac{1}{N} \sum_{i=1}^N u(\mathbf{x}, \boldsymbol{\xi}^{(i)}) \right| \geq a \right) \leq \frac{Var(u)}{Na^2}. \quad (4.30)$$

Note that a factor of 2 error reduction requires an increase of the sample size by a factor of 4. This slow rate of convergence is often cited as the chief difficulty in using the Monte Carlo method [2, 19]. It is also important to note that this bound is probabilistic in nature and that it is possible for the Monte Carlo method to perform much worse (or much better) than expected. For a fixed choice of the quantity on the left hand side of (4.30), which we call P here, say $P = .05$, we have that

$$a \leq \sqrt{\frac{\text{Var}(u)}{.05N}}, \quad (4.31)$$

and from this we can conclude with 95% percent confidence that the Monte Carlo estimate is bounded by $\sqrt{\frac{\text{Var}(u)}{.05N}}$. Smaller values of P lead to looser bounds but greater confidence in those bounds.

The method presented in [48] is to construct an approximation \hat{u} of the solution function in the stochastic space using conventional sparse grid collocation and then, given a finite number of samples $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$, to approximate the expected value by

$$\mathbb{E}(u)(x) \approx \frac{1}{N} \sum_{i=1}^N \hat{u}(\mathbf{x}, \boldsymbol{\xi}^{(i)}). \quad (4.32)$$

Instead of using conventional sparse grid collocation, we construct an approximation \hat{u} using the adaptive KDE collocation method. Assuming that one has already constructed the interpolant, computation of the expected value can be carried out very quickly this way since the interpolant is simple to evaluate. Note also that while the standard Monte Carlo method was used to evaluate (4.32), adaptive KDE

collocation is also compatible with other sampling methods such as quasi-Monte Carlo [7] and multilevel Monte Carlo [5, 8]. In the case of quasi-Monte Carlo, the sample points used in (4.32) are simply chosen to be the quasi-Monte Carlo sample points, and in the case of multilevel Monte Carlo an expression similar to (4.32) is computed at each level of the computation. We expect that combining adaptive KDE collocation with either of these alternative sampling strategies would yield combined benefits; we do not explore this issue here.

The error associated with this method separates into two terms as follows,

$$\begin{aligned}
|\epsilon_{sparse}| &= |\mathbb{E}(u)(\mathbf{x}) - \frac{1}{N} \sum_{i=1}^N \mathcal{A}(u)(\mathbf{x}, \boldsymbol{\xi}^{(i)})| \\
&\leq |\mathbb{E}(u)(\mathbf{x}) - \frac{1}{N} \sum_{i=1}^N u(\mathbf{x}, \boldsymbol{\xi}^{(i)})| + |\frac{1}{N} \sum_{i=1}^N (u(\mathbf{x}, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(\mathbf{x}, \boldsymbol{\xi}^{(i)}))| \\
&= \epsilon_{MC} + \epsilon_{interp}.
\end{aligned}
\tag{4.33}$$

The first term is statistical error and depends only on the number of samples taken and the variance of u , and decays according to (4.30). The second term is the interpolation error and is bounded since the infinity norm of the interpolation error is bounded in the neighborhood of the sample points using (4.23).

Given N samples of $\boldsymbol{\xi}$, evaluation of (4.27) requires N evaluations of the random field u . In the case where u is defined by a system such as (4.1), this requires N solutions of a discrete PDE. In contrast, evaluation of (4.32) requires N_{interp} evaluations of u to construct $\mathcal{A}(u)$ and then it requires N evaluations of

$\mathcal{A}(u)$. The relative computational efficiency of (4.32) then depends on two factors: first, whether an accurate interpolant $\mathcal{A}(u)$ can be constructed using $N_{interp} \ll N$ function evaluations, and second, whether the cost of evaluating $\mathcal{A}(u)$ is significantly less than the cost of evaluating u . The first condition, as shown by (4.24), depends on the dimension of the problem as well as the number of samples we have access to. For most problems of interest the second condition is satisfied in that it is much less expensive to evaluate a piecewise polynomial than it is to solve a discrete algebraic system associated with a complex physical model. Note that in order for ϵ_{interp} to be small the interpolation error only needs to be small near the sample points. For adaptive KDE collocation the kernel density estimate is designed to make the interpolant more accurate in the neighborhoods of these points by indicating where large clusters of points are located.

4.5 Numerical Experiments

In this section we assess the performance of adaptive KDE collocation applied to several test problems. We aim to measure quantitatively the two terms in the estimate (4.23) and to compare the computational efficiency of our method with the Monte Carlo method.

4.5.1 Interpolation of a Highly Oscillatory Function

Before exploring our main concern, the solution of PDEs with stochastic coefficients, we first examine the utility of adaptive collocation for performing a simpler

task, to interpolate a scalar-valued function whose argument is a random vector.

We use adaptive KDE collocation to construct an approximation to the function

$$u(\boldsymbol{\xi}) = \begin{cases} \prod_{k=1}^M |\xi_k| \sin(1/\xi_k) & \text{if } \xi_k \neq 0 \\ 0 & \text{otherwise,} \end{cases} \quad (4.34)$$

where $\boldsymbol{\xi}$ is a random variable uniformly distributed over the set $[-1, -0.5]^M \cup [0.5, 1]^M$. Figure 4.3 shows a plot of the function $u(\boldsymbol{\xi})$ for the single parameter case. The density of $\boldsymbol{\xi}$ is given explicitly by

$$\rho(\boldsymbol{\xi}) = 2^{M-1} \mathbb{1}_{[-1, -0.5]^M \cup [0.5, 1]^M}. \quad (4.35)$$

The function u is everywhere continuous but infinitely oscillatory along each axis of $\boldsymbol{\xi}$. The axes however are not contained in the support of ρ so the oscillations do not have any effect on the statistics of u with respect to the measure on $\boldsymbol{\xi}$. Algorithm 2 with the refinement criterion used in [30] would place many collocation points near the origin in an attempt to resolve the oscillatory behavior. Provided that the approximate density $\hat{\rho}$ is a good approximation to the true density, adaptive KDE collocation will only place collocation points near the support of ρ .

In our experiments, the density estimate for each choice of M will be constructed from 5,000 samples of $\boldsymbol{\xi}$ with the bandwidth h chosen by maximum likelihood cross validation. For a given value of $\boldsymbol{\xi}$ let $|(u(\boldsymbol{\xi}) - \mathcal{A}_k(u)(\boldsymbol{\xi}))\rho(\boldsymbol{\xi})|$ be the interpolation error scaled by ρ . First we measure the scaled interpolation error at 500 equally spaced points on $[-1.5, 1.5]$ and use the maximum observed error

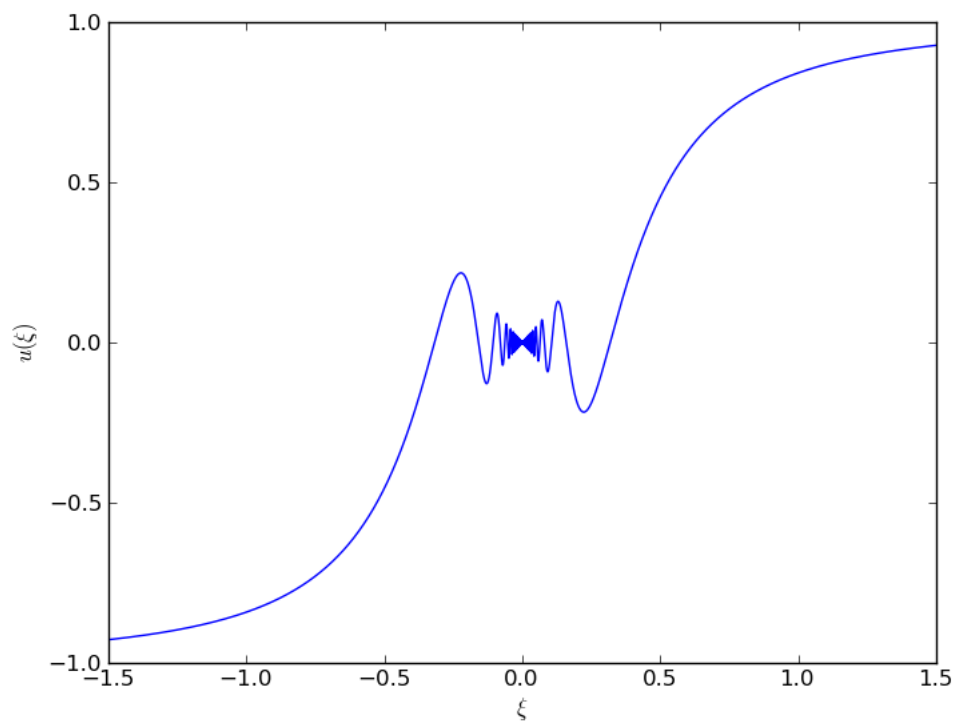


Figure 4.3: $u(\xi) = |\xi| \sin(1/\xi)$.

as an estimate for the infinity norm of the error $\|(u(\xi) - \mathcal{A}_k(u)(\xi))\rho(\xi)\|_{L_\infty(\Gamma)}$ for the one-parameter (i.e. $M = 1$ in (4.34)) problem. We denote this estimate by $\|(u(\xi) - \mathcal{A}_k(u)(\xi))\rho(\xi)\|_{l_\infty}$. Figure 4.4 shows the interpolation error in the mesh-

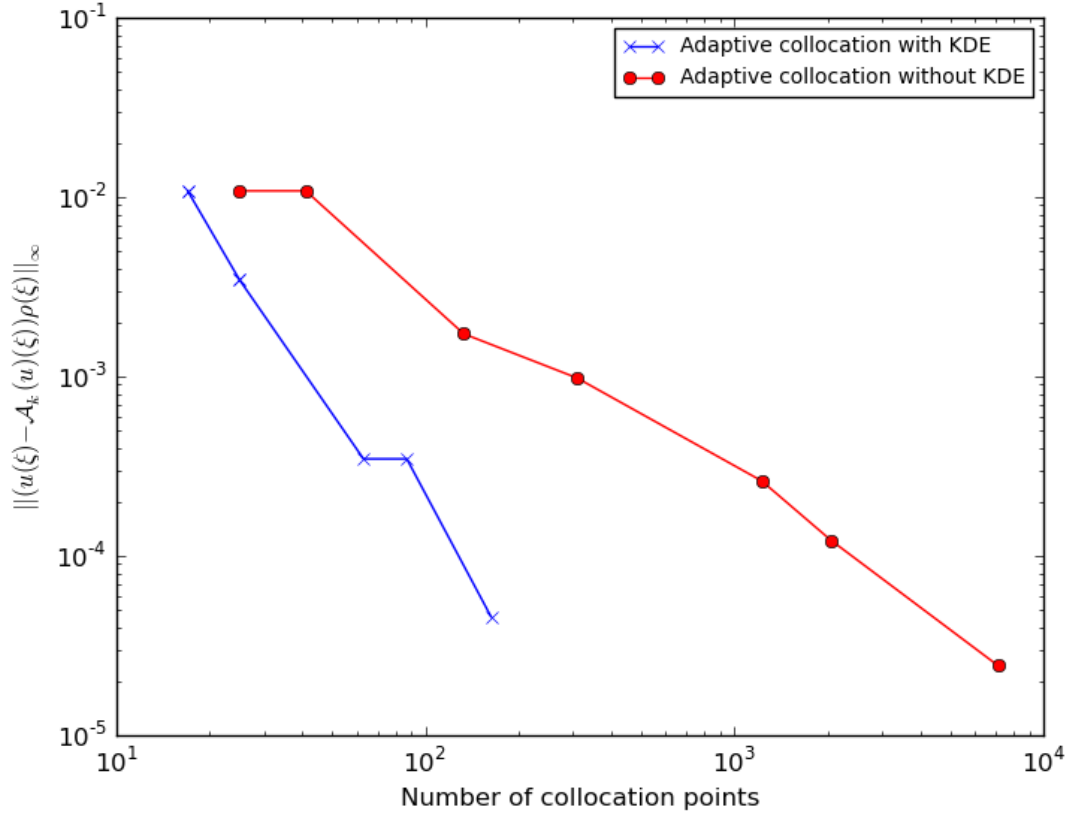


Figure 4.4: $\|(u(\xi) - \mathcal{A}_k(u)(\xi))\rho(\xi)\|_\infty$ versus the number of collocation points

norm $\|\cdot\|_{l_\infty}$. This norm only indicates the error on the support of ρ . Figure 4.4 shows that the interpolation error decays rapidly where the random variable ξ is supported. Figure 4.4 shows that adaptive KDE collocation converges significantly faster than Algorithm 2. The reason is that Algorithm 2 places many points near the origin, attempting to resolve the oscillations. After a few initial global refinements of the grid the new method concentrates all of the new collocation points inside the

support of ξ .² Figure 4.5 shows the collocation nodes used by the adaptive method with KDE driven refinement.

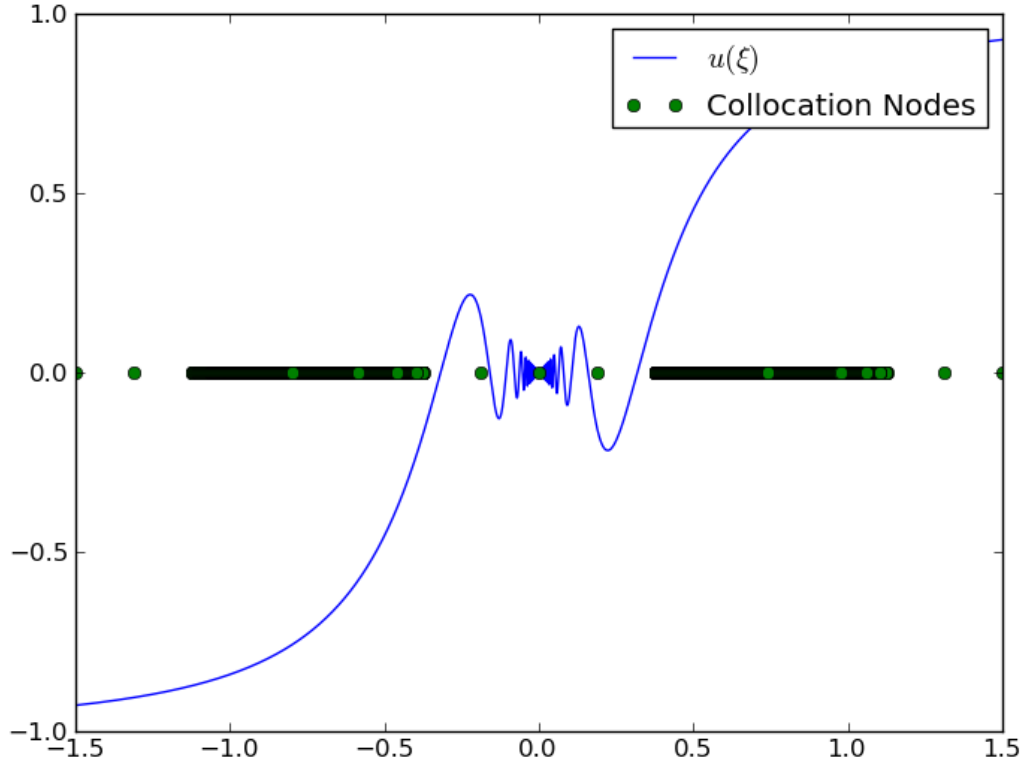


Figure 4.5: $u(\xi)$ and the collocation points used in constructing approximate solution

Now we examine the performance for the same task when u depends on multiple parameters in (4.34). Figure 4.6 shows the number of collocation points required as a function of the convergence criterion τ and the number of parameters. The figure shows that as the number of parameters is increased, the efficiency of the proposed method slows. This is due to the factor $\log_2(|\Delta\theta^k|)^{3(M-1)}$ appearing

²Algorithm 2 with the refinement criterion (4.22) indicates that a node is not refined if $\hat{\rho}||w_j^k||$ is small. In practice however it is necessary to perform some initial global grid refinements to achieve a minimum level of resolution.

in the estimate (4.24). Note however that for any fixed value of M , the asymptotic interpolation error bound (4.24) decays faster than the Monte Carlo error bound (4.30). The results in Section 4.5.4 indicate that the asymptotic bound (4.24) may be pessimistic for problems of interest.

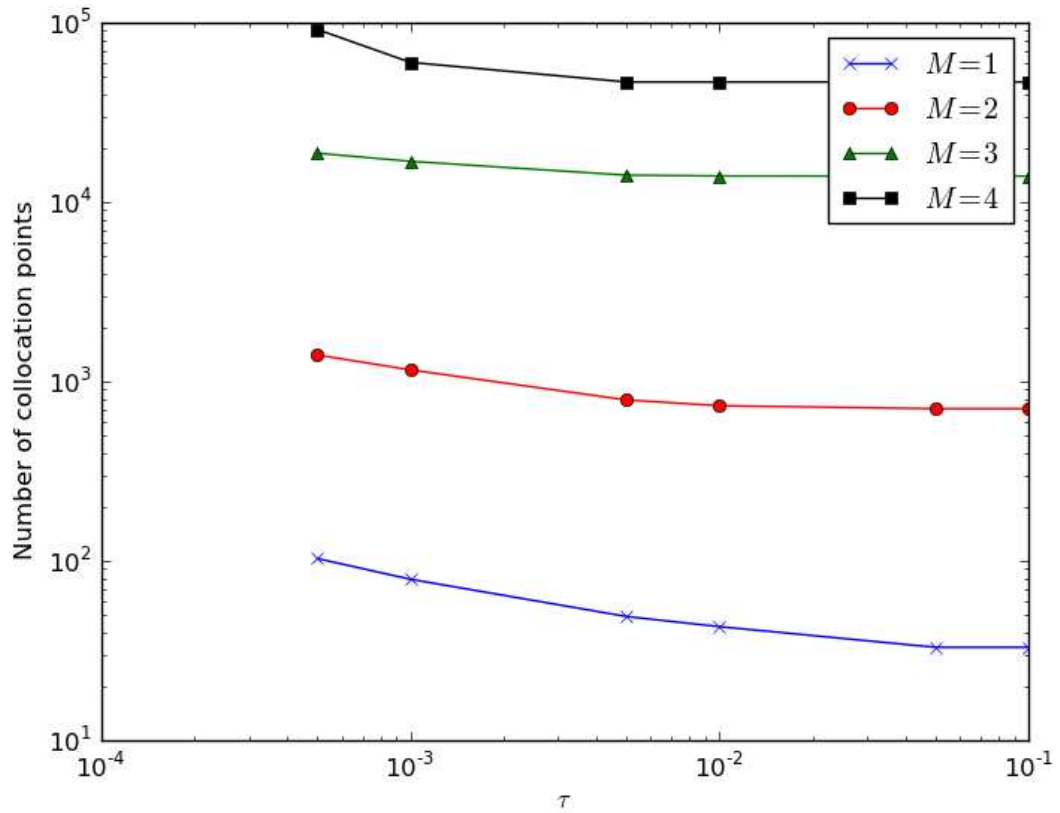


Figure 4.6: The tolerance τ vs the number of collocation points

4.5.2 Two-parameter stochastic diffusion equation

Next, we use the method derived in section 4.4 to compute statistics associated with the solution to the stochastic diffusion equation

$$-\nabla \cdot (a(\mathbf{x}, \xi_1, \xi_2) \nabla u(\mathbf{x}, \xi_1, \xi_2)) = 1, \quad \forall \mathbf{x} \in D \quad (4.36)$$

$$u(\mathbf{x}, \xi_1, \xi_2) = 0, \quad \forall \mathbf{x} \in \partial D \quad (4.37)$$

where $D = [0, 1]^2$. The diffusion coefficient a is defined for this example as follows. Define the set $LL = \{\mathbf{x} : 0 < x_1, x_2 \leq 0.5\}$ and the set $UR = \{\mathbf{x} : 0.5 < x_1, x_2 < 1.0\}$. Let $\mathbb{1}_{LL}(\mathbf{x})$ and $\mathbb{1}_{UR}(\mathbf{x})$ be the indicator functions on LL and UR respectively. The diffusion coefficient is piecewise constant and is given by

$$a(\mathbf{x}, \xi_1, \xi_2) = 1 + \mathbb{1}_{LL}(\mathbf{x})\xi_1 + \mathbb{1}_{UR}(\mathbf{x})\xi_2. \quad (4.38)$$

Here ξ_1 and ξ_2 are assumed to be independently distributed log-normal random variables. The PDF of ξ_i for $i = 1, 2$ is given by

$$\rho_i(\xi_i) = \frac{1}{\xi_i \sqrt{2\pi\sigma^2}} e^{-\frac{(\log(\xi_i) - \mu)^2}{2\sigma^2}}, \quad (4.39)$$

with $\sigma = 1$ and $\mu = 2$. Since ξ_1 and ξ_2 are assumed to be independent, their joint distribution is given by

$$\rho(\xi_1, \xi_2) = \frac{1}{2\pi\xi_1\xi_2} e^{-\frac{-(\log(\xi_1) - 2)^2 - (\log(\xi_2) - 2)^2}{2}}. \quad (4.40)$$

Note that ξ_1 and ξ_2 take on values in the range $(0, \infty)$. This, combined with the definition of the diffusion coefficient in (4.38) ensures that the diffusion coefficient will be positive at all points in D for all possible values of the random variables ξ_1 and ξ_2 . This is sufficient to ensure the well-posedness of (4.36) [2]. In the numerical experiments, interpolation was carried out on the domain $[1 \times 10^{-6}, 6]^2$. This computational domain contained all of the samples of (ξ_1, ξ_2) generated by the log-normal random number generator.

The method described above generates a set of collocation points in the stochastic space. At each of these points (4.36) must be solved by using a suitable deterministic solver. In this example the spatial discretization is accomplished using finite differences on a uniform 32×32 mesh. The discrete difference operators are formed using the five point stencil

$$\begin{bmatrix} & a(x, y + \frac{h_D}{2}, \xi_1, \xi_2) & \\ a(x - \frac{h_D}{2}, y, \xi_1, \xi_2) & a(x, y, \xi_1, \xi_2) & a(x + \frac{h_D}{2}, y, \xi_1, \xi_2) \\ & a(x, y - \frac{h_D}{2}, \xi_1, \xi_2) & \end{bmatrix}, \quad (4.41)$$

for $\mathbf{x} = [x, y]^T \in D$, and where h_D is the spatial discretization parameter. For this example the resulting linear systems are solved using a direct solver, although an iterative solver may also be used as in [16]. Although the spatial discretization of the problem introduces an additional source of error, it is known that the error resulting from the spatial discretization of the problem separates from the error associated with discretization of the stochastic component [2, 3]. Thus we can focus solely on

the error introduced by interpolating in the stochastic space and by approximating the true joint density by a kernel density estimate.

First we proceed as in Section 4.5.1 and evaluate the interpolation error. Since the exact solution is not known we compute $\mathcal{A}(u)$ with a very tight error tolerance $\tau = 10^{-9}$. We treat this as an accurate solution and observe the decay in error for interpolants obtained using a looser error tolerance. For each interpolant, the kernel density estimate is derived from 5,000 samples of $\boldsymbol{\xi} = [\xi_1, \xi_2]$ where ξ_1 and ξ_2 are independently distributed log-normal random variables as described above. The bandwidth for the kernel density estimates is chosen using the maximum likelihood cross-validation method described in section 4.3.

Figure 4.7 shows the collocation points used for several values of the error tolerance τ . Comparing these with the contour plot of the true joint density function in Figure 4.8, it can be seen that the method is concentrating collocation points in regions where the estimated joint PDF is large. Thus the method is only devoting resources towards computing an accurate interpolant in regions that are significant to the statistics of u . Figure 4.9 shows the interpolation error as a function of the number of collocation points. Since an exact solution to (4.36) is not available we treat the solution obtained by using the method with $\tau = 10^{-10}$ as an exact solution. As opposed to the first example, the solution u here depends on both the spatial location and the value of the random parameter. We report the error in the discrete norm $\|\cdot\|_{l^2(D) \times l^\infty(\Gamma)}$, where the space $l^2(D)$ consists of square summable mesh-functions defined on the spatial grid and $l^\infty(\Gamma)$ consists of bounded mesh-functions defined on a 500×500 uniform grid on Γ . Figure 4.9 shows that the interpolation

error decays quickly for the two parameter problem. The apparent slowdown in convergence rate is attributable to the fact that the exact solution is not available and the error is being measured with respect to an approximate solution.

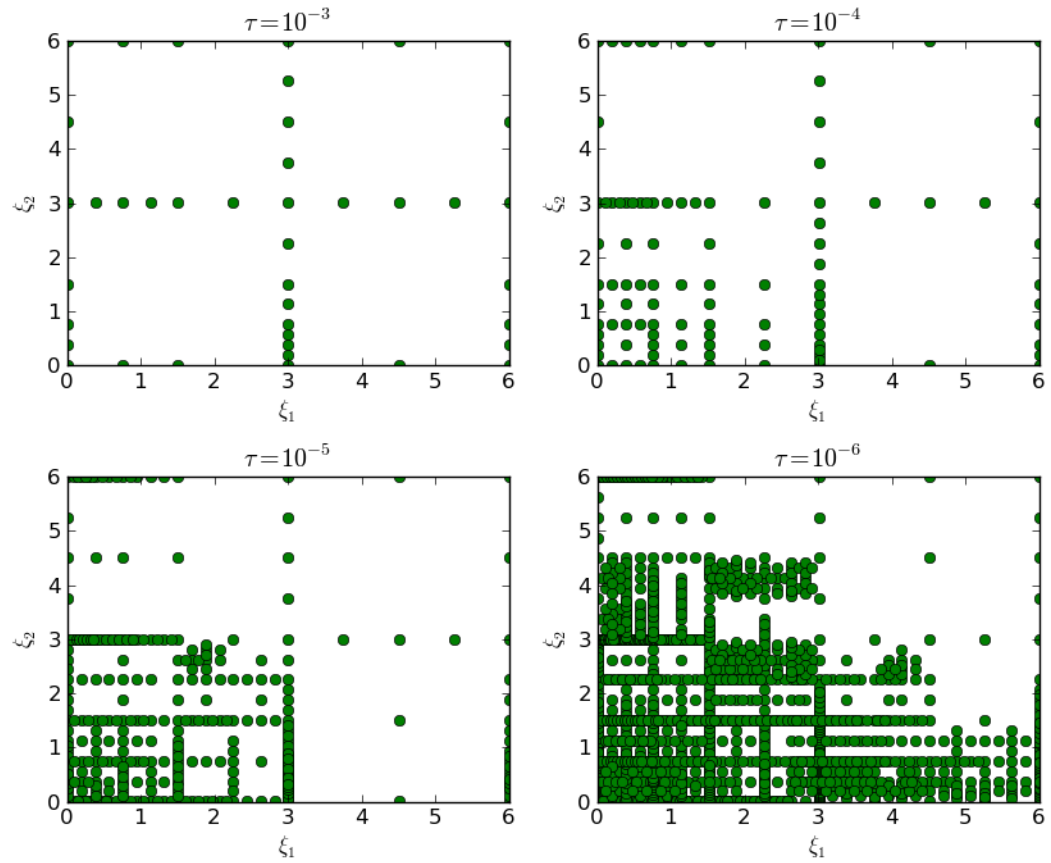


Figure 4.7: Collocation points for various values of the error tolerance τ

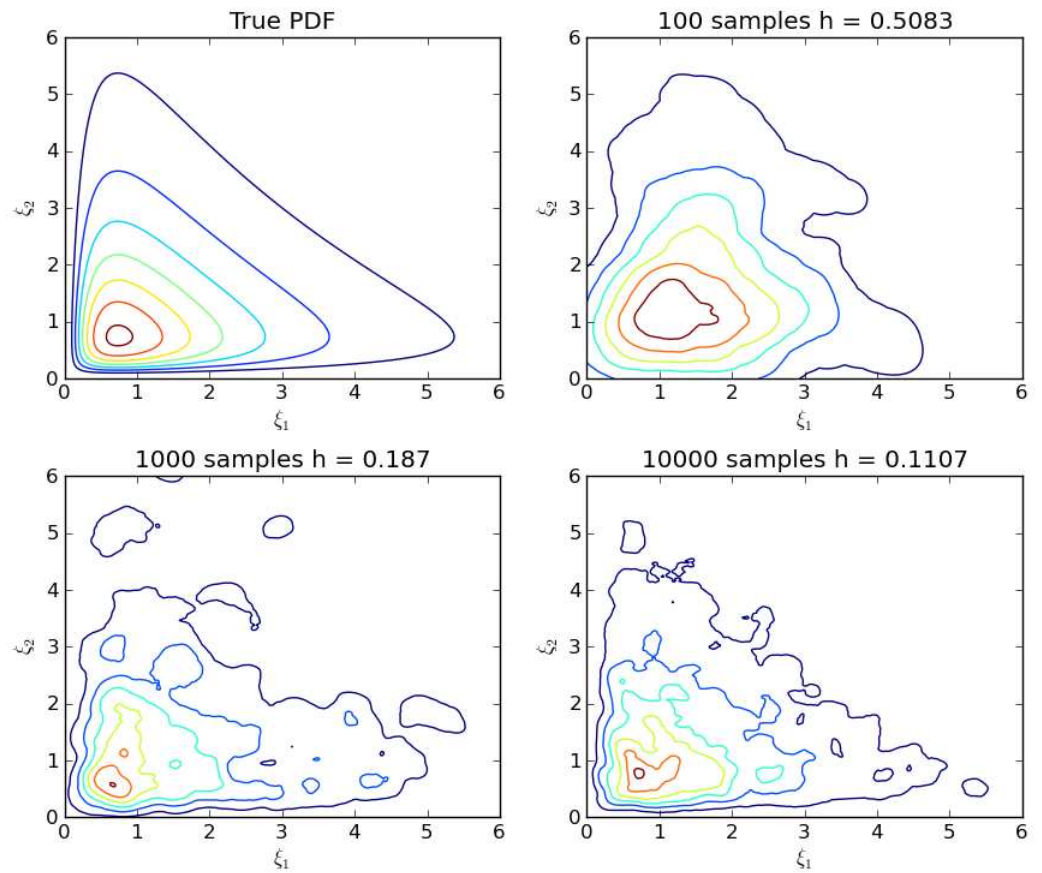


Figure 4.8: Kernel density estimates for varying numbers of samples.

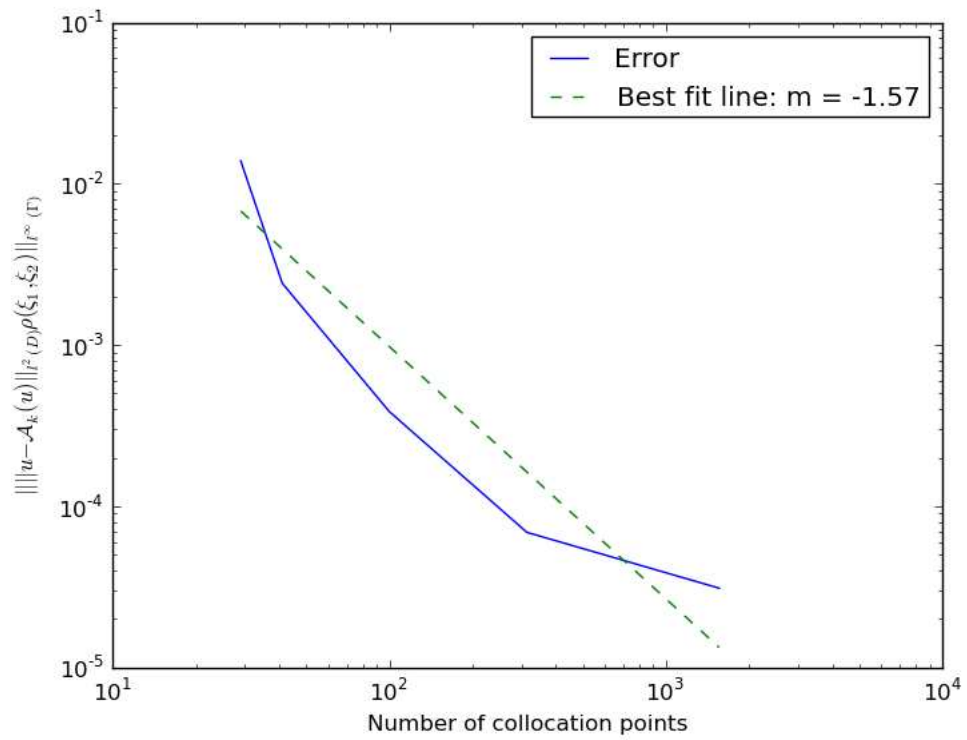


Figure 4.9: $\|(u(\mathbf{x}, \boldsymbol{\xi}) - \mathcal{A}(u)(\mathbf{x}, \boldsymbol{\xi}))\rho(\boldsymbol{\xi})\|_{l^2(D) \times l^\infty(\Gamma)}$ versus the number of collocation points

4.5.3 Function with steep gradients and non-independently distributed random parameters

We now use the adaptive KDE collocation method to compute the statistics associated with the function

$$\begin{aligned}
 u(\xi_1, \xi_2) &= \xi_1 \xi_2 + 10e^{-\frac{(1-r)^2}{.1}} + 10e^{-\frac{(1-\hat{r})^2}{.1}} & (4.42) \\
 r(\xi_1, \xi_2) &= \sqrt{\xi_1^2 + \xi_2^2} \\
 \hat{r}(\xi_1, \xi_2) &= \sqrt{(\xi_1 - 5)^2 + (\xi_2 - 5)^2},
 \end{aligned}$$

where ξ_1 is a log-normal random variable with marginal density function given by

$$\rho_1(\xi_1) = \frac{1}{\xi_1 \sqrt{2\pi(.4^2)}} e^{-\frac{\log(\xi_1)^2}{2(.4^2)}}, \quad (4.43)$$

and ξ_2 is given by

$$\xi_2 = \xi_1 + \eta, \quad (4.44)$$

where η is a uniformly distributed random variable on $[0, 1]$. This function has two line singularities, one along the circle of radius one centered at the origin, and another along the circle of radius one centered at $[5, 5]^t$. A surface plot of this function is shown in Figure 4.10

The random variables ξ_1 and ξ_2 are obviously dependent. A sample set $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$ consisting of $N = 10000$ samples from the distribution of $\boldsymbol{\xi} = [\xi_1, \xi_2]^t$

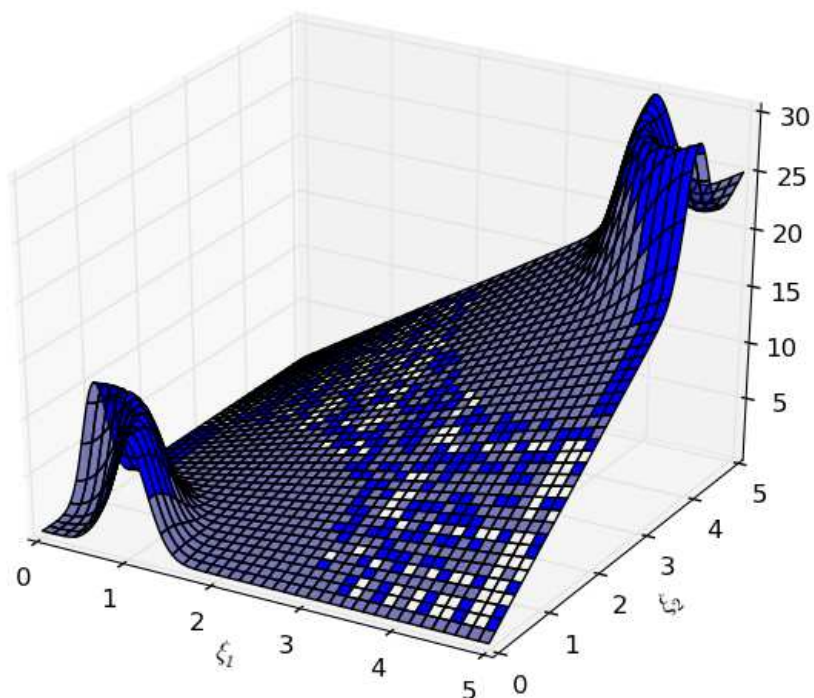


Figure 4.10: Function with two line singularities in parameter space

was generated. From this data a kernel density estimate was constructed using the optimal bandwidth obtained via MLCV. The data points and a contour plot of the associated KDE are shown in Figure 4.11. Since the distribution of data points is more dense on the left side of the parameter domain it follows that the line singularity centered at the origin will have a greater effect on the solution statistics than the line singularity centered at $[5, 5]^t$. Because of this, in order to recover accurate statistics, more effort should be spent resolving the singularity centered at the origin than the singularity centered at $[5, 5]^t$.

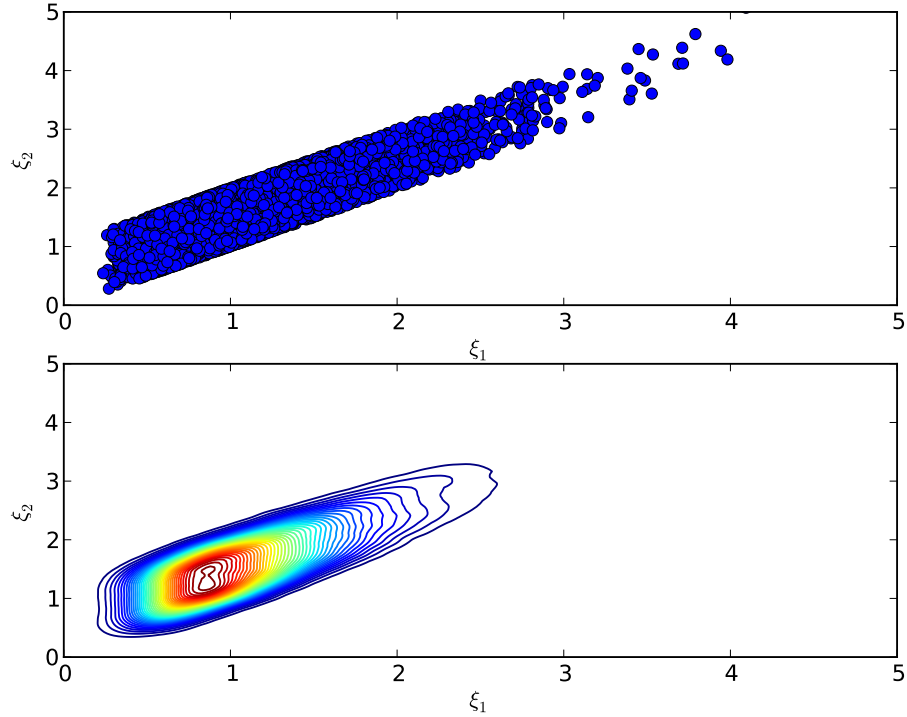


Figure 4.11: Samples (top) and kernel density estimate (bottom) for the distribution of two dependent random variables

Using the set of 10000 samples we performed a Monte Carlo simulation, approximating the expected value as $\mathbb{E}(u) \approx \frac{1}{N} \sum_{i=1}^N u(\boldsymbol{\xi}^{(i)})$. From the function values at each of these 10000 samples we also computed the sample variance of the solution,

$$\text{var}[u(\boldsymbol{\xi})] \approx \frac{1}{N-1} \sum_{i=1}^N \left(u(\boldsymbol{\xi}^{(i)}) - \frac{1}{N} \sum_{i=1}^N u(\boldsymbol{\xi}^{(i)}) \right)^2. \quad (4.45)$$

Using this as an estimate of the true variance, equation (4.31) was used to compute a 95% confidence bound of the Monte Carlo error. We also performed the adaptive KDE collocation method for several values of refinement criterion τ . The adaptive KDE collocation method was required to perform global refinements until at least

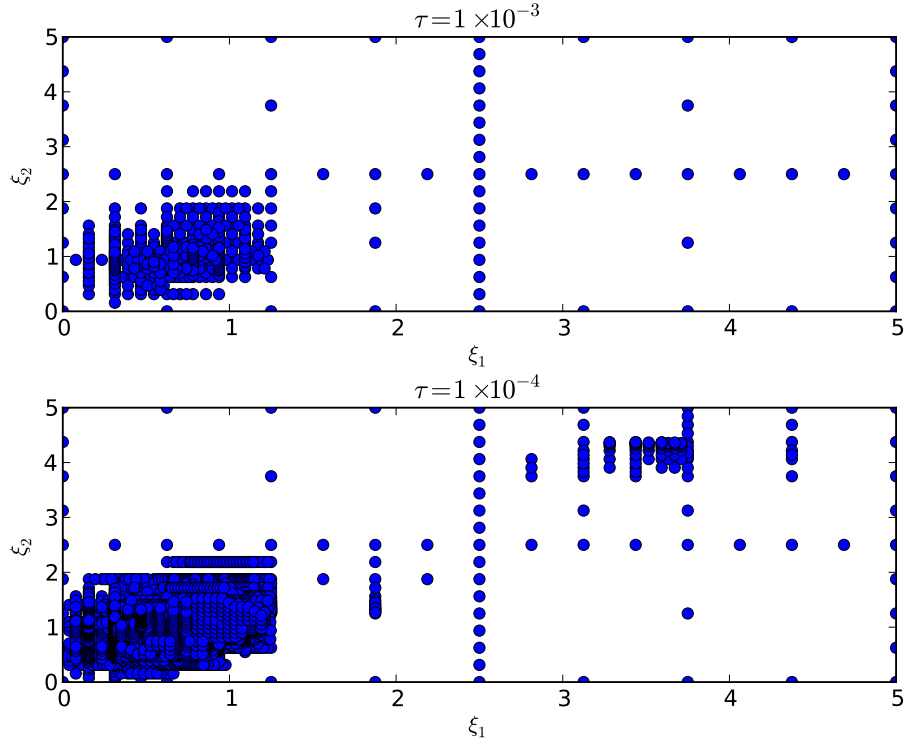


Figure 4.12: Collocation points for refinement criterion $\tau = 1 \times 10^{-3}$ and $\tau = 1 \times 10^{-4}$

the fifth level. This provided the minimum resolution to detect the presence of the line singularities. After the full fifth level grid was constructed, the refinement procedure was continued by using the grid refinement criterion (4.22). Once the hierarchical interpolant was constructed we performed the Monte Carlo method on the interpolant to approximate the mean, that is we approximated $\mathbb{E}(u) \approx \frac{1}{N} \sum_{i=1}^N \mathcal{A}(u)(\boldsymbol{\xi}^{(i)})$. This result was compared to the true sample mean $\mathbb{E}(u) \approx \frac{1}{N} \sum_{i=1}^N u(\boldsymbol{\xi}^{(i)})$. Figure 4.12 shows the distribution of collocation points for two values of τ . From this figure we see that the adaptive KDE collocation method is detecting the line singularities but is allocating more collocation points to resolve the singularity centered at the origin since the density of the data points is higher there.

Table 4.1 shows the confidence bound on the Monte Carlo error as well as the difference between the true sample mean and the sample mean of the hierarchical interpolant. The number of collocation points required to construct the interpolant is displayed in parentheses. Table 4.1 shows that the adaptive KDE collocation method can construct an approximation to the solution such that the interpolation error at the sample data points is substantially smaller than the bound on the Monte Carlo error. Thus, using the hierarchical interpolant in place of the true solution introduces a negligible extra error into the computation of the expected value. Furthermore many fewer function evaluations were required to compute the hierarchical interpolant than would be required to perform the Monte Carlo method. Here we are testing the method on a function given by an analytic formula however, in many practical situations function evaluations may be very expensive. Table 4.1 shows that by using the adaptive KDE collocation method we can obtain comparable results to the Monte Carlo method while using many fewer function evaluations, which are often the primary cost associated with sampling methods. Furthermore this method performs well even when the solution function has steep gradients and when there are dependencies between the parameters.

Monte Carlo	τ			
Error Bound	5×10^{-1}	3×10^{-1}	1×10^{-1}	1×10^{-4}
1.133×10^{-1}	7.083×10^{-1} (73)	8.482×10^{-2} (101)	2.508×10^{-2} (188)	6.121×10^{-3} (3984)

Table 4.1: Monte Carlo error bound, $|\frac{1}{N} \sum_{i=1}^N u(\boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(\boldsymbol{\xi}^{(i)})|$, and number of collocation points (in parentheses)

4.5.4 High-dimensional stochastic diffusion

We now examine the performance of adaptive KDE collocation for evaluating the statistics of a random field that depends on a large number of parameters. The problem is given by

$$-\frac{d}{dx}(a_M(x, \boldsymbol{\xi}) \frac{d}{dx}u(x, \boldsymbol{\xi})) = 1, \quad \forall x \in (0, 1) \quad (4.46)$$

$$u(0, \boldsymbol{\xi}) = u(1, \boldsymbol{\xi}) = 0. \quad (4.47)$$

The diffusion coefficient a_M is defined for even M by

$$a_M = \mu + \sum_{k=0}^{M/2-1} \lambda_k (\xi_{2k} \cos(2\pi kx) + \xi_{2k+1} \sin(2\pi kx)), \quad (4.48)$$

where $\lambda_k = \exp(-k)$, $\mu = 3$ and ξ_k is uniformly distributed on $[0, 1]$. The problem (4.46) is well posed on the image of $\boldsymbol{\xi}$. Experimental results for these problems are shown in Tables 4.2 (for $M = 4$ random variables), 4.3 ($M = 10$), and 4.4 ($M = 20$). We assess the performance of the method in a similar manner to the test problem from section 4.5.3. The contents of the tables are as follows.

First, for each M , we performed a Monte Carlo simulation with several choices of number of samples N . This sample size is shown in the first column of the tables. In addition, for each value of M , $\text{var}[u(\mathbf{x}, \boldsymbol{\xi})]$ was estimated at the spatial grid points using 20,000 samples. Equation (4.31) can then be used to compute a 95% confidence bound of the Monte Carlo error. This estimate is shown in the first column of Tables 4.2, 4.3, and 4.4 beneath the number of samples used to construct

the Monte Carlo estimate.

The other columns of the tables contain results for adaptive KDE collocation where the kernel density estimates are generated using the same set of sample points used for the Monte Carlo simulation. The total error for this method is bounded by (4.33). The term $\|\epsilon_{MC}\|_{l^2(D)}$ is estimated by the 95% confidence bound in the first column of the tables, as discussed in the previous paragraph. The other quantities in the table are the $l^2(D)$ -norm of the sample mean interpolation error, $\|\epsilon_{interp}\|_{l^2(D)}$, in the top of each box, together with (in parentheses) the number of collocation points N_{interp} used to construct $\mathcal{A}(u)$. For example, the second from left entry in the bottom row of Table 4.4 shows that for the 20-parameter problem and the 20,000 sample set, $\mathcal{A}(u)$ was constructed using 3,108 collocation points and $\|\epsilon_{interp}\|_{l^2(D)} = 6.52 \times 10^{-4}$.

The costs of the two methods are essentially determined by the number of PDE solves required, N for the Monte Carlo simulation and N_{interp} for adaptive KDE collocation. In the tables, the number of collocation points N_{interp} in parentheses are shown in bold typeface when they are smaller than the number of samples. For such cases, if $\|\epsilon_{interp}\|_{l^2(D)}$ is significantly smaller than $\|\epsilon_{MC}\|_{l^2(D)}$, then adaptive KDE collocation is less expensive than Monte Carlo simulation. It can be seen from the results that the savings can be significant when the number of samples increases. For example, the second from left entry in the bottom row of Table 4.4 shows that (by (4.33)) the error in mean for the adaptive collocation method is bounded by $\|\epsilon_{interp}\|_{l^2(D)} + \|\epsilon_{MC}\|_{l^2(D)} = 7.11 \times 10^{-3}$ while only requiring 3,108 PDE solves, an error comparable in magnitude to that obtained with the Monte Carlo method (6.46×10^{-3}) with 20,000 solves.

We also note that these results suggest that the factor $\log_2(|\Delta\theta^k|)^{3(M-1)}$ in the estimate (4.24) may be pessimistic for many problems of interest. Care must be taken when using the predictor method not to over-resolve the interpolant when one only has access to only a small amount of data. Doing so results in an interpolant that is too accurate given the number of samples available and results in wasted computation. This is the case in the right-hand columns of the tables where the interpolant is being resolved to a much higher level of accuracy than the associated Monte Carlo error bound.

4.6 Conclusions

We have presented a new adaptive sparse grid collocation method based on the method proposed in [30] that can be used when the joint PDF of the stochastic parameters is not available and all one has access to is a finite set of samples from that distribution. It is shown that in this case a kernel density estimate can provide a mechanism for driving the refinement of an adaptive sparse grid collocation strategy. Numerical experiments show that in cases involving a large number of samples it can be economical to construct a surrogate to the unknown function using fewer function evaluations and then to perform the Monte Carlo method on that surrogate. This method has the additional advantage that it performs well even in the case when there are dependencies between the parameters that define the problem.

N	τ				
	5×10^{-2}	1×10^{-3}	5×10^{-4}	1×10^{-4}	5×10^{-5}
100 8.43×10^{-2}	5.25×10^{-3} (28)	2.23×10^{-4} (212)	1.18×10^{-4} (301)	9.42×10^{-6} (813)	9.42×10^{-7} (1169)
500 3.78×10^{-2}	5.47×10^{-3} (28)	2.71×10^{-4} (211)	9.84×10^{-5} (315)	1.12×10^{-5} (777)	1.76×10^{-6} (1210)
1000 2.67×10^{-2}	4.29×10^{-3} (33)	2.36×10^{-4} (200)	1.24×10^{-4} (297)	9.78×10^{-6} (762)	2.61×10^{-6} (1207)
5000 1.19×10^{-2}	4.36×10^{-3} (33)	3.88×10^{-4} (172)	1.36×10^{-4} (286)	1.67×10^{-5} (745)	4.73×10^{-6} (1104)
20000 5.96×10^{-3}	4.32×10^{-3} (33)	2.73×10^{-4} (180)	1.30×10^{-4} (294)	1.09×10^{-5} (780)	3.58×10^{-6} (1107)

Table 4.2: Monte Carlo error (left) and $\|\frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\|_{l^2(D)}$, 4 parameter problem

N	τ				
	5×10^{-2}	1×10^{-3}	5×10^{-4}	1×10^{-4}	5×10^{-5}
100 9.08×10^{-2}	7.66×10^{-3} (76)	8.86×10^{-4} (1026)	4.41×10^{-4} (1655)	4.48×10^{-5} (5026)	8.28×10^{-6} (8111)
500 4.06×10^{-2}	7.13×10^{-3} (92)	6.08×10^{-4} (1170)	3.36×10^{-4} (1189)	2.34×10^{-5} (5773)	1.01×10^{-5} (9404)
1000 2.87×10^{-2}	9.19×10^{-3} (59)	6.03×10^{-4} (1216)	2.65×10^{-4} (1989)	1.95×10^{-5} (5996)	1.77×10^{-5} (9664)
5000 1.28×10^{-2}	7.16×10^{-3} (93)	6.62×10^{-4} (1120)	3.03×10^{-4} (2041)	2.04×10^{-5} (6095)	1.02×10^{-5} (9787)
20000 6.42×10^{-3}	7.25×10^{-3} (93)	6.27×10^{-4} (1187)	2.66×10^{-4} (2127)	1.96×10^{-5} (6050)	5.67×10^{-6} (9942)

Table 4.3: Monte Carlo error (left) and $\|\frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\|_{l^2(D)}$, 10 parameter problem

N	τ				
	5×10^{-2}	1×10^{-3}	5×10^{-4}	1×10^{-4}	5×10^{-5}
100 9.14×10^{-2}	1.64×10^{-2} (41)	1.65×10^{-3} (878)	2.15×10^{-3} (1299)	5.81×10^{-4} (4126)	2.39×10^{-4} (6958)
500 4.09×10^{-2}	1.45×10^{-2} (41)	2.77×10^{-3} (1045)	1.38×10^{-3} (1738)	3.75×10^{-4} (5545)	1.67×10^{-4} (9106)
1000 2.89×10^{-2}	8.45×10^{-3} (119)	1.46×10^{-3} (1618)	9.02×10^{-4} (2622)	1.66×10^{-4} (8580)	7.13×10^{-5} (14012)
5000 1.29×10^{-2}	8.70×10^{-3} (156)	9.58×10^{-4} (2459)	4.99×10^{-4} (4169)	7.88×10^{-5} (13389)	2.59×10^{-5} (22276)
20000 6.46×10^{-3}	7.25×10^{-3} (193)	6.52×10^{-4} (3108)	3.38×10^{-4} (4991)	3.48×10^{-5} (15963)	2.35×10^{-5} (26081)

Table 4.4: Monte Carlo error (left) and $\|\frac{1}{N} \sum_{i=1}^N u(x, \boldsymbol{\xi}^{(i)}) - \mathcal{A}(u)(x, \boldsymbol{\xi}^{(i)})\|_{l^2(D)}$, 20 parameter problem

Chapter 5

Performance Analysis of the Adaptive KDE Collocation Method

Using CUDA

Recently much attention has been paid to implementing numerical algorithms on graphics processing units (GPUs). GPUs are well suited for performing many numerical algorithms due to the large number of cores available on a single device and because GPUs allocate a higher proportion of computing resources to floating point operations than most commodity CPUs [37]. Figure 5.1 shows that increases in the theoretical performance of GPUs have vastly outpaced CPUs in single precision floating point computations and that the same trend is beginning to emerge in double precision computations. NVIDIA's CUDA (short for Complete Unified Device Architecture) is a parallel computing architecture that enables programmers to perform general purpose computations on the GPU using the CUDA-C application programming interface (API), which provides a set of extensions to the C programming language.

In this chapter we will present an implementation of the adaptive KDE collocation method developed in chapter 4 for solving systems of the form (4.1), which uses an adaptive strategy for choosing collocation points and kernel density estimation to approximate the joint density function $\rho(\boldsymbol{\xi})$. To handle some of the most computationally intensive portions of this method, we use CUDA C/C++ with wrappers

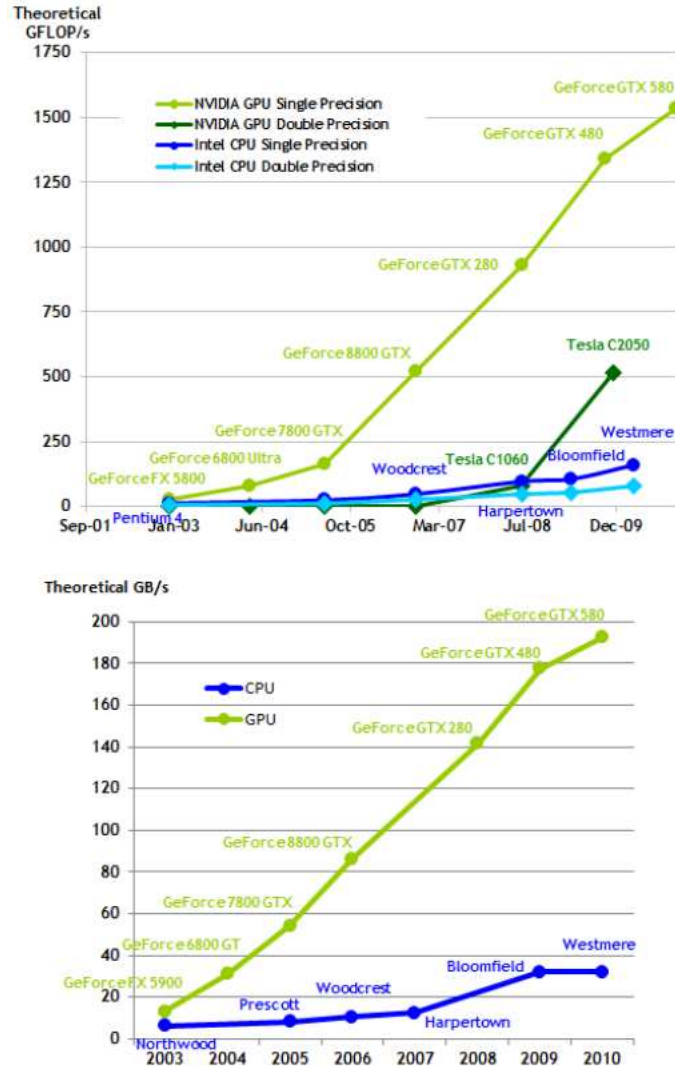


Figure 5.1: Floating-point operations per second and memory bandwidth for various CPU and GPU architectures [37]

that enable a Python interpreter to call the CUDA functions.

The remainder of the chapter proceeds as follows. Section 5.1 presents an overview of kernel density estimation. Section 5.2 presents an overview of the adaptive KDE collocation method. Section 5.3 presents an overview of the CUDA architecture. Section 5.4 presents CUDA C implementations of kernel density estimation. Section 5.5 presents CUDA C implementations of the adaptive KDE collocation

method. Section 5.6 presents benchmarks of the implementations. In section 5.7 we draw some conclusions.

5.1 Summary of Kernel Density Estimation

Recall that given N samples $\{\boldsymbol{\xi}^{(i)}\}_{i=1}^N$ of an M -dimensional random vector, the kernel density estimate is given by (4.15). In the adaptive KDE collocation method presented in chapter 4 we require the value of $\hat{\rho}$ computed at T targets denoted $\{\mathbf{x}^{(j)}\}_{j=1}^T$. In the programs presented below we use the multiplicative Epanechnikov kernel given by

$$K(\mathbf{u}) = \prod_{i=1}^M \frac{3}{4} (1 - u_i^2) \mathbb{1}_{\{|u_i| \leq 1\}}. \quad (5.1)$$

It is trivial to verify that the Epanechnikov kernel satisfies the assumptions of (4.14). An additional important property of the Epanechnikov kernel, in the context of stochastic partial differential equations, is that it is compactly supported.

Computing $\hat{\rho}$ is accomplished in two separate steps. First the kernel K is computed at each pair of samples and targets, that is, compute $K\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\xi}^{(j)}}{h}\right)$ for $1 \leq i \leq T$ and $1 \leq j \leq N$. If \mathbf{K} is the matrix with entries defined by

$$\mathbf{K}_{ij} = K\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\xi}^{(j)}}{h}\right), \quad (5.2)$$

and $X = [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}]$, then $\hat{\rho}(\mathbf{X}) = [\hat{\rho}(\mathbf{x}^{(1)}), \dots, \hat{\rho}(\mathbf{x}^{(T)})]^t$ is $\frac{1}{Nh^M}$ times the vector formed by summing the rows of \mathbf{K} . Once each entry of \mathbf{K} is computed then the

rows are summed and the resulting vector is scaled by the normalizing factor to obtain $\hat{\rho}$. Note that the asymptotic cost associated with this algorithm is $\mathcal{O}(MNT)$.

If the Gaussian kernel $K(u) = \frac{1}{2\pi} \exp(-u^2)$ is used then there is an alternative method for evaluating (4.15) is the fast Gauss transform. This method can evaluate $\hat{\rho}$ in $\mathcal{O}(T + N)$ time rather than in $\mathcal{O}(MN)$ time as described above [53]. However in that method, the asymptotic constant grows rapidly with increasing M , so it may not be tractable for problems posed in very high-dimensional parameter spaces. Also, the Gaussian kernel is unbounded, which can lead to problems since the PDE one is attempting to solve may become ill-posed for large parameter values.

In order to use kernel density estimation effectively it is necessary to choose an appropriate value of the bandwidth parameter h . Ideally a procedure for selecting the bandwidth h should depend only on the data and not require any a priori assumptions about the shape of the true density.

Recall that the maximum likelihood cross validation method defines the optimal value of h as

$$\operatorname{argmax}_h[CV(h)] = \operatorname{argmax}_h \left(\frac{1}{N} \sum_{i=1}^N \log(\hat{\rho}_{-i}(\boldsymbol{\xi}^{(i)})) \right), \quad (5.3)$$

where

$$\hat{\rho}_{-i}(\mathbf{x}) = \frac{1}{(N-1)h^M} \sum_{\substack{k=1 \\ k \neq i}}^N K \left(\frac{\mathbf{x} - \boldsymbol{\xi}^{(k)}}{h} \right). \quad (5.4)$$

Note that the objective function in (5.3) is equivalent to

$$CV(h) = \frac{1}{N} \sum_{i=1}^N \log \left(\frac{N}{N-1} \hat{\rho}(\boldsymbol{\xi}^{(i)}) - \frac{1}{(N-1)h^M} K(\mathbf{0}) \right), \quad (5.5)$$

where $\mathbf{0}$ is the M -dimensional zero vector. Each evaluation of the objective function therefore requires $\mathcal{O}(MN^2)$ flops. Note that evaluating both $\hat{\rho}$ and $CV(h)$ is ideally suited for implementation on GPUs since the task of evaluating the kernel K at each sample and target is inherently data parallel.

5.2 Summary of Adaptive KDE Collocation

Let $\boldsymbol{\xi}$ be an M -dimensional, continuous, real valued, random vector with joint probability density function $\rho(\boldsymbol{\xi})$. Let $\Gamma = \text{Image}(\boldsymbol{\xi})$ and let $u : \Gamma \rightarrow \mathbb{R}^s$. The solution $u(\boldsymbol{\xi})$ is often taken to be the vector of coefficients from a discrete approximation to the solution of a stochastic PDE such as the stochastic diffusion equation

$$-\nabla \cdot (a(\mathbf{x}, \boldsymbol{\xi}) \nabla u(\mathbf{x}, \boldsymbol{\xi})) = f(\mathbf{x}, \boldsymbol{\xi}). \quad (5.6)$$

Thus evaluating u for a specific value of $\boldsymbol{\xi}$ is equivalent to solving a deterministic partial differential equation using a discretization with s spatial degrees of freedom.

Recall from chapter 4 that the level- k adaptive KDE interpolant is evaluated as

$$\mathcal{A}_k(u)(\boldsymbol{\xi}) = \sum_{i=1}^k \sum_{\boldsymbol{\xi}_j^i \in \Delta \theta^i} w_j^{\Delta i} a_j^{\Delta i}(\boldsymbol{\xi}), \quad (5.7)$$

where $\Delta\theta_{adapt}^i$ is the set of points in the i^{th} adaptive grid level. Note that each hierarchical surplus $w_j^{\Delta^i}$ is a vector in \mathbb{R}^s . Note also that for the case of a single random variable, each grid point is uniquely defined via (4.6) by its approximation level i and the index j of the point within the i^{th} approximation level. For functions of multiple random variables, each grid point can be uniquely identified by a multi-index \mathbf{i} that specifies the interpolation level of that point in each dimension, and by a multi-index \mathbf{j} that specifies the index of the point within each level. Thus from a data structure perspective, once the endpoints of the interpolation domain are specified, the hierarchical grid can be defined as a collection of multi-indices that specify the level and index of each point in the grid.

For the purposes of discussing the algorithm, it is more convenient to index the collocation points and basis elements with a single index, that is,

$$\mathcal{A}_k(u)(\xi) = \sum_{i=1}^{N_\xi} w_i a_i(\xi), \quad (5.8)$$

where N_ξ is the total number of collocation points. If we want to evaluate the interpolant at T targets $[\boldsymbol{\xi}_1, \dots, \boldsymbol{\xi}_T] \in \mathbb{R}^{M \times T}$, then the result can be written as

$$Y = WA \quad (5.9)$$

where $Y \in \mathbb{R}^{s \times T}$ is the matrix whose i^{th} column contains the value of $\mathcal{A}(u)(\boldsymbol{\xi}_i)$, $W \in \mathbb{R}^{s \times N_\xi}$ is the matrix whose i^{th} column is the i^{th} hierarchical surplus, and

$A \in \mathbb{R}^{N_\xi \times T}$ is defined as

$$A = \begin{bmatrix} a_1(\boldsymbol{\xi}_1) & a_1(\boldsymbol{\xi}_2) & \cdots & a_1(\boldsymbol{\xi}_T) \\ a_2(\boldsymbol{\xi}_1) & \ddots & & \\ \vdots & & & \\ a_{N_\xi}(\boldsymbol{\xi}_1) & \cdots & & a_M(\boldsymbol{\xi}_T) \end{bmatrix}. \quad (5.10)$$

The hierarchical surpluses can be computed in a similar way. If there are T new collocation points in the $(i+1)^{st}$ grid level, denoted $[\boldsymbol{\xi}_{N_\xi+1}, \dots, \boldsymbol{\xi}_{N_\xi+T}]$, then we use the partitioned matrix

$$W = [W^*|U] = [w_1, \dots, w_{N_\xi} | u(\boldsymbol{\xi}_{N_\xi+1}), \dots, u(\boldsymbol{\xi}_{N_\xi+T})], \quad (5.11)$$

and the matrix

$$A = \begin{bmatrix} a_1(\boldsymbol{\xi}_{N_\xi+1}) & a_1(\boldsymbol{\xi}_{N_\xi+2}) & \cdots & a_1(\boldsymbol{\xi}_{N_\xi+T}) \\ a_2(\boldsymbol{\xi}_{N_\xi+1}) & \ddots & & \\ \vdots & & & \\ a_{N_\xi}(\boldsymbol{\xi}_{N_\xi+1}) & \cdots & & a_{N_\xi}(\boldsymbol{\xi}_{N_\xi+T}) \end{bmatrix}. \quad (5.12)$$

The new hierarchical surpluses can then be computed by the update

$$U = U - W^*A. \quad (5.13)$$

Thus, both computing the value of the interpolant and computing a new set of hierarchical surpluses involves the computation of many basis elements at many target points, and the computation of a matrix-matrix product. Performing the computation in this manner ensures that the computation is rich in matrix-matrix multiplication, which is generally desirable from the standpoint of computational efficiency [13, 20]. Also, phrasing the algorithm in terms of matrix algebra allows us to use the CUBLAS library [38], which contains high performance linear algebra routines optimized to run in parallel on CUDA GPUs.

From the discussion in this and the previous section it is evident that the construction of the adaptive KDE collocation interpolant contains several sub-tasks that may be computationally expensive. In most cases of interest it is safe to assume that the dominant cost of the method will be that of evaluating u at every collocation point. The method's other costs, that of performing MLCV, evaluating the KDE, computing the coefficients of the expansion, and evaluating the approximation, may also be high. MLCV involves repeated evaluations of an objective function (5.5) that scales quadratically in complexity as the number of sample data points increases. Evaluating the KDE scales similarly to MLCV when the number of samples and targets are of the same order. Evaluating the expansion coefficients for the hierarchical interpolant scales as the product of the number of collocation points from the previous grid level with the number of points from the new level. If the ability to construct an adaptive KDE collocation interpolant with large sample sets at a large number of collocation points is needed, then it is necessary to have efficient implementations of all of these tasks.

5.3 Brief Description of the CUDA Architecture

GPUs are ideal for certain numerical algorithms because they devote a much larger proportion of computing resources to performing floating point arithmetic than CPUs and they are capable of executing thousands of threads in parallel. In particular, algorithms that are data parallel, where the same instructions are carried out on multiple data elements, and where the ratio of arithmetic operations to memory accesses is high, are ideal candidates for GPU execution [37]. A GPU typically consists of a number of multiprocessors (processors capable of executing many threads simultaneously) each of which contains many cores and is capable of executing a very large number of threads¹ in parallel. Code executed on a CUDA device is referred to as a *computational kernel*.² The GPU is viewed as a separate computational engine from the host CPU that is designed to work in tandem with the CPU on floating point intensive tasks, not to replace it entirely. The general control flow for a program that uses CUDA is: data is copied from the host's main memory to the GPU; the host specifies an *execution configuration* for the CUDA kernel, and the kernel executes on the data elements in parallel; finally the result is copied from the GPU back to the host. In order to understand algorithms written for CUDA it is necessary to understand some of CUDA's key abstractions.

The first of these abstractions is the thread hierarchy. A CUDA kernel being executed on a GPU consists of some number of *thread blocks*. Each thread block itself

¹A thread is the smallest unit of processing that can be scheduled by an operating system.

²In order to avoid confusion with our other use of the word "kernel," in kernel density estimation, we will refer to computational kernels as *CUDA kernels* throughout the text. The word kernel by itself refers to the mathematical kernel used in kernel density estimation.

contains some number of individual threads. Each thread block is assigned by the GPU's scheduler to one of the device's multiprocessors. Individual threads within the thread block are assigned to individual cores within the multiprocessor. Each multiprocessor schedules the execution of individual threads in groups of 32 threads called a *warp*. Maximum performance can be attained by ensuring that each thread in a warp executes the same instructions and accesses consecutive memory addresses [36]. Once all of the threads in a block have finished execution, the scheduler allows a new block to begin executing. The order in which thread blocks are executed is undefined and communication between thread blocks is limited. Thread blocks are thus expected to be able to execute independently and in any order. The threads within a block can be synchronized by calling the `__syncthreads()` API function. A thread that reaches this instruction will wait until all of the other threads within its block reach the `__syncthreads()` call before continuing execution.

When a CUDA kernel is called by the host, the host must specify the execution configuration for the CUDA kernel. The execution configuration specifies the number of thread blocks that will be run as well as the number of threads in each thread block. The thread blocks can be arranged in a one-dimensional array, where the index of a particular block is given by `blockIdx.x`, or in a two-dimensional array where the index of a block is given by the pair `[blockIdx.x,blockIdx.y]`. The array of thread blocks is referred to as a *grid*. For a two-dimensional array of thread blocks the array dimensions are given by `gridDim.x` and `gridDim.y`. Within each block, the individual threads can also be arranged in a one, two, or three-dimensional array. In the two-dimensional case the index of a particular thread

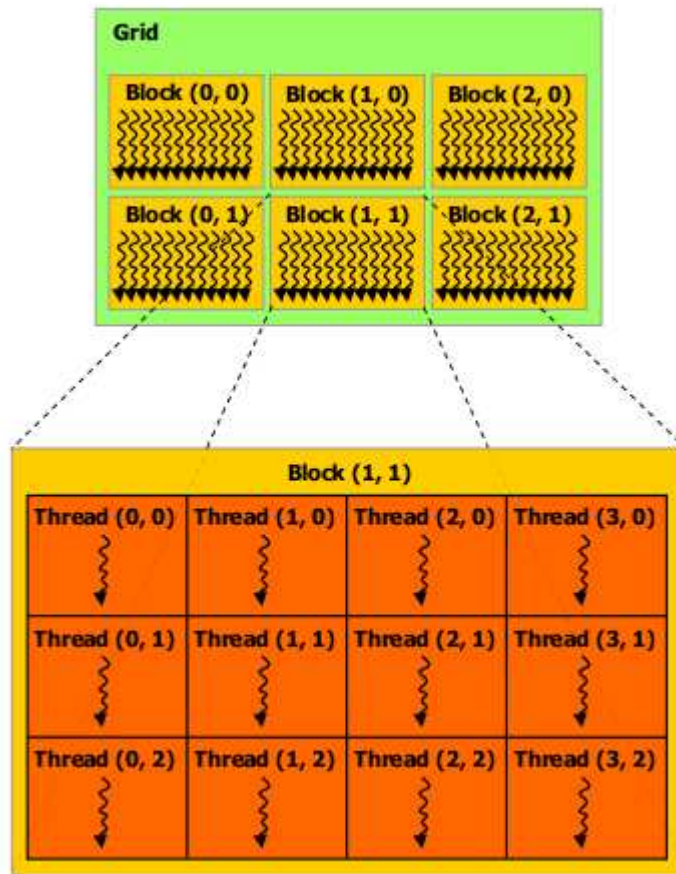


Figure 5.2: Organization of CUDA thread blocks and CUDA threads [37]

within a thread block is given by the pair `[threadIdx.x, threadIdx.y]`. Figure 5.2 shows the organization of the thread blocks on the grid and the threads within each thread block.

A problem to be solved using a CUDA device should first be partitioned into coarse sub-problems. Each of these sub-problems corresponds to a thread block within the grid. The task of solving each of the sub-problems is again subdivided among the individual threads within the block. The CUDA architecture follows the ‘single instruction multiple data’ (SIMD) paradigm and does not perform any predictive branching. A consequence of this is that threads within a warp that diverge at a conditional branch will serialize, that is, the threads that take the branch will execute first followed by those that do not take the branch (or vice versa, the order is undefined). This serialization of divergent threads at a conditional branch is referred to as a divergent warp. As a result, tasks that require many control statements will not exhibit significant gains in performance on GPUs, since after taking many branches the threads in a warp will be effectively serialized.

The second key abstraction is the memory hierarchy. There are several memory spaces on the CUDA device, each of which serves a different purpose. For the purposes of this discussion it is sufficient to focus on the *global*, *shared*, and *local* memory spaces. Every thread executing on the device has access to the device’s global memory. Global memory can be allocated on the CUDA device by a call to the `cudaMalloc()` function for one-dimensional arrays or by to `cudaMallocPitch()` for two-dimensional arrays. Global memory serves as a staging area for data between the host and the device. Data is copied from the host’s main memory to

the device's global memory and back by calls to `cudaMemcpy()` or `cudaMemcpy2d()` for one or two-dimensional arrays respectively. Applications gain the most benefit from running on the GPU if a very large number of threads can be executed in parallel. However, if a CUDA kernel executes with a large number of threads, then the per-thread bandwidth of the global memory space is very low. Thus, for good performance, frequent global memory accesses need to be avoided.

In addition to the global memory space, each thread block contains a modest amount of shared memory that can only be accessed by threads residing inside that block. Shared memory is very fast, as it is physically adjacent to each multiprocessor and functions as an explicitly managed cache. However, shared memory is not accessible to any threads in other thread blocks. When specifying the execution configuration, the host must also specify the amount of dynamically allocated shared memory that will be used by each thread block. Each individual thread also uses a small amount of local memory which is not accessible by any other thread. This local thread memory can be used for storing primitives, loop counters, and data pointers needed by the individual threads. Figure 5.3 shows a diagram of the CUDA memory hierarchy.

The transfer of data between the host and the GPU takes place along the *PCIe bus*. Since the PCIe bus has relatively small bandwidth, in order to obtain a performance benefit by using the GPU, this data transfer must be masked by a large number of floating point operations. Ideally, the ratio of floating point operations to data transferred should grow as the problem size increases [37]. Effective memory management in CUDA thus consists primarily of two considerations. First, since the

PCIe bus is slow, data transfers between the host computer and the CUDA device need to be minimized. Second, thread blocks need to make effective use of shared memory and minimize access to the device's global memory address space.

An additional factor which can affect performance is the coalescing of memory accesses. Data can be obtained from the GPU's global memory in blocks of B bytes, where the value of B depends on the device. Thus when data is being read from the global memory space, significant performance gains can be realized if the memory accesses can be aligned to B bytes. For example if $B = 32$ and an m by n two dimensional array A stored in row major order occupies a contiguous block of memory then the (i, j) entry is located in $A[n*(i-1) + j-1]$, using the convention that indexing starts with zero. Assuming that each entry of A uses 4 bytes in memory and $n = 5$, the second row of A begins 20 bytes after the start of A . If a thread requires the second row of A (which contains bytes 20 through 40) then the hardware must perform two reads to obtain this data, one for bytes 0 to 31 of A and another for bytes 32 to 63. If instead the end of each row is padded with 12 bytes, then each row can be accessed in a single read. This process of padding can be done using the CUDA API function `CudaMallocPitch()`, which automatically allocates memory for two-dimensional arrays so that the start of each row is properly aligned. `CudaMallocPitch()` returns the length in bytes of each padded row as well as a pointer to the allocated memory. Arrays allocated in this way are referred to as *pitched arrays*.

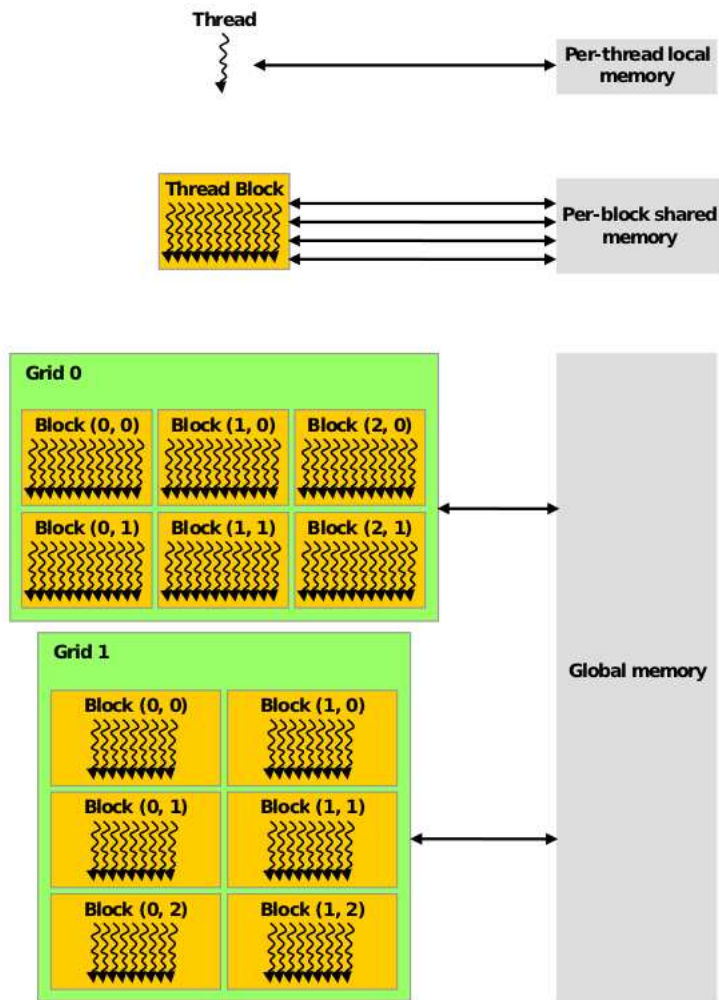


Figure 5.3: CUDA memory hierarchy [37]

5.4 Implementation of Kernel Density Estimation in CUDA C

The implementation of kernel density estimation on the GPU consists of two parts: the host code, which manages the movement of data between the host and the device, sets up the execution configuration, and provides a front end for the CUDA kernel; and the CUDA kernel, which computes the kernel density estimate in parallel on the GPU. In this section, we will describe the implementation of the CUDA kernel that evaluates the KDE, and then we will give a description of the host code.

5.4.1 The CUDA KDE kernel

The declaration of the CUDA KDE kernel for KDE is given by

```
--global-- void KDE_cuda_kernel(float const* targets ,  
                                const unsigned int dimension ,  
                                const unsigned int num_targets ,  
                                const size_t targets_pitch ,  
                                const float* samples ,  
                                const unsigned int num_samples ,  
                                const size_t samples_pitch ,  
                                const float bandwidth ,  
                                float* result ) .
```

The pointer `targets` points to a pitched array on the GPU that contains `num_targets` targets, each of which is a vector of size specified by the parameter `dimension`. The

pointer `samples` points to a pitched array on the GPU that contains `num_samples` samples of a random vector also of size `dimension`. The KDE defined by these samples and the `bandwidth` parameter is evaluated by this kernel and stored in the device array `result`. This CUDA kernel is used within the code both to evaluate the KDE and to evaluate the objective function for MLCV by using (5.5). Pseudo-code for this CUDA kernel is given by algorithm 3. The approach used here is similar to the approach used in [46].

Let the matrix of targets be denoted by $X = [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}]$ and the matrix of samples be denoted $\bar{\boldsymbol{\xi}} = [\boldsymbol{\xi}^{(1)}, \dots, \boldsymbol{\xi}^{(N)}]$. The threads are organized into two-dimensional thread blocks of size `BLOCK_SIZE`, where `BLOCK_SIZE` is a parameter defined in a header file. Since threads are executed in groups of 32 it is beneficial if the number of threads in each block is divisible by 32. In the tests shown below, `BLOCK_SIZE` was defined to be 16 so each thread block contained 256 threads. Each thread executing the kernel computes $K\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\xi}^{(j)}}{h}\right)$ for a single value of i and j . The thread blocks are arranged as a two-dimensional grid of size `ceil(num_targets/BLOCK_SIZE)` by `ceil(num_samples/BLOCK_SIZE)`. Each thread determines its sample and target by accessing the `blockIdx.x`, `blockIdx.y`, `threadIdx.x`, and `threadIdx.y` parameters.

Each block requires access to `BLOCK_SIZE` targets and `BLOCK_SIZE` samples. In order to minimize accesses to global memory, rather than letting each thread fetch its sample and target from global memory, the threads on the diagonal of each block (e.g. those with `threadIdx.x` equal to `threadIdx.y`) first copy a sample and target into the block's shared memory. The data in shared memory can then be accessed

quickly by all of the threads in the block. In this way the array of samples is read from global memory `num_targets/BLOCK_SIZE` times and the array of targets is read `num_samples/BLOCK_SIZE` times. Each thread in the block is forced to wait until all of the samples and targets have been copied to shared memory before resuming the computation by calling `__syncthreads()`.

Once the required samples and targets have been copied into shared memory, each thread computes $K\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\xi}^{(j)}}{h}\right)$ for its target and sample. The results from each thread then need to be accumulated. The most recent version of the CUDA architecture supports atomic addition operations³ for floating point values [37]. The results from each thread are first summed into a shared memory location. Once all of the results from the block are accumulated, a single thread for each target in the block atomically adds the corresponding local accumulation to the global `result` array which can then be copied to the host.

5.4.2 The KernelDensityEstimator class

The host code encapsulates all of the data required for computing the kernel density estimate in the C++ class `KernelDensityEstimator`. The `KernelDensityEstimator` constructor prototype is given by

```
KernelDensityEstimator::
```

³Atomic addition is a process where the operation $a = a + b$ is carried out in a single machine instruction. Without atomic addition the update $a = a + b$ can take a few operations to complete. This is problematic in multithreaded code since two threads may simultaneously attempt to perform the update, which can result in undefined behavior. Without atomic addition it is necessary for each thread to set a lock that prevents other threads from attempting to perform the update until the thread that set the lock is complete.

Algorithm 3 Pseudo-code for the CUDA KDE kernel

```
mySampleBlock  $\equiv$  blockIdx.x
myTargetBlock  $\equiv$  blockIdx.y
mySample  $\equiv$  threadIdx.x
myTarget  $\equiv$  threadIdx.y
sampleIndex  $\equiv$  BLOCK_SIZE*mySampleBlock + mySample
targetIndex  $\equiv$  BLOCK_SIZE*myTargetBlock + myTarget
Allocate sharedSamples a BLOCK_SIZE by dimension array in shared memory.
Allocate sharedResult a BLOCK_SIZE array in shared memory.
Allocate sharedTargets a BLOCK_SIZE by dimension array in shared memory.
if threadIdx.x==threadIdx.y then
    sharedSamples[mySample,:] = samples[sampleIndex,:]
    sharedTargets[myTarget,:] = targets[targetIndex,:]
end if
__syncthreads()
 $u = (\text{sharedTargets}[\text{myTarget},:] - \text{sharedSamples}[\text{mySample},:]) / \text{bandwidth}$ 
Atomically perform sharedResult[myTarget] +=  $K(u)$ 
__syncthreads()
if threadIdx.x == 0 then
    Atomically perform result[targetIndex] += sharedResult[myTarget]
end if
```

```
KernelDensityEstimator(float const* samples,
                        const unsigned int num_samples_,
                        const unsigned int dimension_,
                        const float bandwidth_,
                        const unsigned int num_expected_targets)
```

The array `samples` is an array of `num_samples_` samples drawn from the distribution of a `dimension_`-dimensional random vector. The parameter `num_expected_targets` tells the constructor how much memory to allocate on the CUDA device for the array of targets. If the KDE needs to be evaluated at additional targets, a reallocation of CUDA device memory will occur. Pseudo-code for the `KernelDensityEstimator` class constructor is given in algorithm 4.

Algorithm 4 Pseudo-code for the `KernelDensityEstimator` class constructor

```
bandwidth = bandwidth_  
Allocate a pitched array d_samples to store the samples on the GPU.  
Copy samples to d_samples  
Allocate a pitched array d_targets to store the targets on the GPU.  
Allocate an array d_result to store the result of the computation.
```

When the constructor is called, it first allocates some pitched memory on the CUDA device to store the samples, targets, and results of the KDE evaluation. The `KernelDensityEstimator` object maintains pointers to these device arrays. The samples are also transferred to the device where they reside until the object is deleted.

Once a `KernelDensityEstimator` is constructed, the KDE can be evaluated by calling the `evaluateKDE` member function

```
void KernelDensityEstimator::  
evaluateKDE(float const * targets ,  
            const unsigned int num_targets ,  
            float* result ).
```

This function copies the array of targets from the host to the device, sets up the CUDA kernel execution configuration, calls the CUDA kernel and copies the result back to the host. As discussed above, the computational grid is a two-dimensional array of thread blocks of size $\lceil \text{num_samples}/\text{BLOCK_SIZE} \rceil$ by $\lceil \text{num_targets}/\text{BLOCK_SIZE} \rceil$. Also, each block requires dynamically allocated shared memory to store its targets, samples, and the result of the local accumulations. Pseudo-code for the `evaluateKDE()` function is given in algorithm 5.

Algorithm 5 Pseudo-code for the evaluateKDE()

```
Copy targets to d_targets
Set up the execution configuration:
  dimBlock = [BLOCK_SIZE, BLOCK_SIZE]
  dimGrid = [ceil(num_samples/BLOCK_SIZE),
             ceil(num_targets/BLOCK_SIZE)]
  sharedMemSize = BLOCK_SIZE*samples_pitch +
                 BLOCK_SIZE*targets_pitch + BLOCK_SIZE
Invoke KDE_cuda_kernel with d_samples, d_targets
Return result to d_result
Copy d_result to result
```

5.5 Implementation of the adaptive KDE collocation method in CUDA

C

Similar to the implementation of the kernel density estimation discussed above, the implementation of adaptive KDE collocation consists of two parts, the host code, and the CUDA kernels. The host code is contained within the `CudaApproximation-Grid` class, which manages the movement of data between the host and the CUDA device and sets the execution configuration for the CUDA kernels. The CUDA kernels evaluate the basis functions associated collocation points in parallel. This is a necessary step in both the evaluation of the interpolant and the computation of hierarchical surpluses.

A CUDA kernel accomplishes the first task and routines from the CUBLAS library [38] accomplish the second.

5.5.1 The CUDA adaptive collocation kernel

CUDA kernels were written to evaluate the basis functions associated with a hierarchical grid at many target locations in parallel. One CUDA kernel computes the basis function at an arbitrary set of points. The second computes the basis function at a new level of collocation points defined by their multi-indices. Combining these with a matrix-matrix product routine from the CUBLAS library enables fast computation of hierarchical interpolants and hierarchical surpluses associated with new grid points, as described by (5.9) and (5.13). The two CUDA kernels are very similar so we only present an overview here of the CUDA kernel that evaluates the basis functions at arbitrary grid points.

A prototype of this CUDA kernel is given below.

```
__global__ void cuda_compute_basis_vals
(
    float const * x, const size_t x_pitch,
    const unsigned int num_evals,
    const unsigned int dimension,
    unsigned int const* levels,
    const size_t levels_pitch,
    unsigned int const* indexes,
    const size_t indexes_pitch,
    const unsigned int total_num_pts,
    float const * endpoints,
    const size_t value_pitch,

```

```
float * value)
```

The CUDA kernel takes `x`, a pitched array of floats stored on the CUDA device. This array specifies the locations where the basis functions are to be evaluated. The parameters `num_evals` and `dimension` specify the number of targets and the dimension of each of the targets. Thus `x` must have size at least `num_evals*dimension`. The grid information is passed from a `CudaApproximationGrid` object through the pitched arrays `levels` and `indexes` that store the multi-indices that define the points in the adaptive grid. The parameter `total_num_pts` is the number of collocation points in the approximation. The `CudaApproximationGrid` also passes the domain of interpolation through the `endpoints` parameter. The matrix A containing the value of the basis functions at the target points is returned through the pitched array `value`. Pseudo-code for the `cuda_compute_basis_vals()` CUDA kernel is given in algorithm 6.

Similar to the CUDA kernels for evaluating the kernel density estimate, the thread blocks in `cuda_compute_basis_vals()` are arranged in a two-dimensional grid of size `ceil(total_num_pts/BLOCK_SIZE)` by `ceil(num_evals/BLOCK_SIZE)`, where each block is responsible for computing a subset of the basis elements at a subset of target points. The threads in each thread block are arranged into a two-dimensional array of size `BLOCK_SIZE` by `BLOCK_SIZE` where each thread computes the value of a single basis element at a single target. In order to minimize access to global memory, the threads in each block with `threadIdx.x == threadIdx.y` load their target and collocation point into shared memory. Each thread then loops over

Algorithm 6 Pseudo-code for the CUDA hierarchical collocation kernel

```
myCollocBlock  $\equiv$  blockIdx.x
myTargetBlock  $\equiv$  blockIdx.y
myCollocPt  $\equiv$  threadIdx.x
myTarget  $\equiv$  threadIdx.y
collocIndex  $\equiv$  BLOCK_SIZE*myCollocBlock + myCollocPt
targetIndex  $\equiv$  BLOCK_SIZE*myTargetBlock + myTarget
Allocate sharedCollocLvl a BLOCK_SIZE by dimension array in shared memory.
Allocate sharedCollocIndex a BLOCK_SIZE by dimension array in shared mem-
ory.
Allocate sharedTargets a BLOCK_SIZE by dimension array in shared memory.
if threadIdx.x==threadIdx.y then
    sharedCollocLvl[myCollocPt,:] = levels[collocIndex,:]
    sharedCollocIndex[myCollocPt,:] = indexes[collocIndex,:]
    sharedTargets[myTarget,:] = targets[targetIndex,:]
end if
__syncthreads()
x = sharedTargets[myTarget,:]
i = sharedCollocLvl[myCollocPt,:]
j = sharedCollocIndex[myCollocPt,:]
value[collocIndex,targetIndex] =  $a_j^i(\mathbf{x})$ 
```

the number of input variables and computes the one-dimensional components of its basis element at its target while keeping a running product. The value from this computation is then passed back into the `value` array.

5.5.2 Implementation of the adaptive KDE collocation host code

The purpose of the `CudaApproximationGrid` class is to encapsulate the data needed by the CUDA device to evaluate hierarchical interpolants and to compute the coefficients of the hierarchical approximation given a set of new data. The `CudaApproximationGrid` class handles all of the required data movement between the host and the device. The logic that drives the grid refinement is implemented as a high level CPU code in Python since this operation is not a primary bottleneck.

The prototype of the `CudaApproximationGrid` constructor is shown below

```
CudaApproximationGrid::
```

```
CudaApproximationGrid(const unsigned int input_dim_,  
                      const unsigned int output_dim_,  
                      const unsigned int max_num_points_,  
                      float const * endpoints):
```

The constructor takes as arguments an unsigned integer `input_dim_` that specifies the number of parameters used to define u ; an unsigned integer `output_dim_` that specifies the dimension of the output of u , e.g. $u : \Gamma \rightarrow \mathbb{R}^{\text{output_dimension}}$ with $\Gamma \subset \mathbb{R}^{\text{input_dimension}}$; the maximum number of collocation points to be used in constructing the interpolant; and a float array of size `input_dimension` by 2 that defines the endpoints of the hypercube where the interpolant is defined. Pseudo-code for the `CudaApproximationGrid` constructor is given in algorithm 7. When the constructor is called it allocates memory on the CUDA device to store the hierarchical surpluses and the level and index sets that define the sparse grid. The host object keeps copies of these pointers.

Algorithm 7 Pseudo-code for the `KernelDensityEstimator` class constructor

Allocate array `d_endpoints` of size `input_dim_` by 2 on the device

Allocate pitched array `d_surpluses` of size `max_num_points_` by `output_dim_` on the device

Allocate pitched array `d_levels` of size `max_num_points_` by `input_dim_` on the device

Allocate pitched array `d_indexes` of size `max_num_points_` by `input_dim_` on the device

In order to add new points to a `CudaApproximationGrid` the `addLevel()` function is called. The prototype for this function is given below.

```
void CudaApproximationGrid::
addLevel(const unsigned int num_new_points,
         unsigned int const* new_levels,
         unsigned int const* new_indexes,
         float const* new_values)
```

The user must pass the number of new points to be added to the grid, arrays of size `input_dimension` by `num_new_points` that specify the levels and indexes of these new points, and an array of size `num_new_points` by `output_dimension` that specifies the function values at those points. If there is available space on the CUDA device to store the new points the `addLevel()` function transfers the new levels, indexes and values to the device and calls the function `computeNewSurpluses()` to compute the new hierarchical surpluses from the function values.

New hierarchical surpluses for a level added to the grid are computed by the `computeNewSurpluses()` function. The computation is accomplished as follows, before the function is called the `addLevel()` function adds the new function values to the end of the array storing the current set of hierarchical surpluses. That is, the matrix of surpluses is partitioned as $W = [W^*|U]$ as in (5.13). This function calls the CUDA kernel to compute the values of the basis functions from the old approximation level at the new points and the CUBLAS function to compute the matrix-matrix update. Pseudo-code for the `computeNewSurpluses()` routine is

given by algorithm 8.

The code for computing the value of the interpolant is similar to the code

Algorithm 8 Pseudo-code for the computeNewSurpluses() function

```
Define W to be the matrix that contains the hierarchical surpluses
Define U to be the matrix that contains the function values at the new collocation
points
Add U to the end of W
Allocate array A of size num_old_points by num_new_points on the device
Set up the execution configuration:
    dimBlock = [BLOCK_SIZE,BLOCK_SIZE]
    dimGrid = [ceil(num_old_points/BLOCK_SIZE),
               ceil(num_new_points/BLOCK_SIZE)]
    sharedMemSize = 2*input_sim*BLOCK_SIZE + input_dim*BLOCK_SIZE
Call the CUDA kernel cuda_compute_basis_vals to compute the basis functions
at the new collocation points and store the result in A
Call cublasSgemm to perform the update  $U = U - WA$ 
```

for computing hierarchical surpluses. The getInterpolantValue() function calls the CUDA kernel cuda_compute_basis_vals() to compute the values of the basis functions at the evaluation points, and the CUBLAS routine cublasSgemm() for computing the matrix-matrix product (5.9). The prototype for the getInterpolantValue function is given below.

```
void CudaApproximationGrid::
    getInterpolantValue(float const * x,
        const unsigned int num_evals,
        float* y)
```

The function takes as arguments an array of floats `x` of size `num_evals` by `input_dimension` that contains the locations where the interpolant is to be evaluated, and an array of floats `y` of size `num_evals` by `output_dimension` to store the result.

Pseudo-code for `getInterpolantValue()` is given in algorithm 9.

Algorithm 9 Pseudo-code for the `getInterpolantValue()` function

Define W to be the matrix that stores the hierarchical surpluses on the device.
Allocate the array \mathbf{d}_x on the device
Copy the target locations \mathbf{x} to \mathbf{d}_x
Allocate array \mathbf{A} of size `num_old_points` by `num_targets` on the device
Set up the execution configuration:
 `dimBlock = [BLOCK_SIZE,BLOCK_SIZE]`
 `dimGrid = [ceil(num_old_points/BLOCK_SIZE),`
 `ceil(num_targets/BLOCK_SIZE)]`
 `sharedMemSize = 2*input_sim*BLOCK_SIZE + input_dim*BLOCK_SIZE`
Call the CUDA kernel `cuda_compute_basis_vals` to compute the basis functions
at the targets points and store the result in \mathbf{A}
Call `cublasSgemm` to calculate the result $\mathcal{A}(u)(\mathbf{x}) = \mathbf{W}\mathbf{A}$

5.6 Benchmarks

In this section we present benchmarks of the CUDA implementations of KDE, MLCV, and adaptive KDE collocation against a serial implementation of these algorithms written in C and python. The GPU implementations were performed on an Intel Xeon x5550 server with a 2.66Ghz CPU and a NVIDIA Tesla C2050 GPU. The Tesla GPU has 14 multiprocessors each clocked at 1.15Ghz. Each multiprocessor has 32 cores for a total of 448 cores on the device. The GPU has 2.5 GB of global memory and 50 MB of shared memory available per multiprocessor. The serial implementations were run on an desktop machine with an Intel Core 2 Duo E6750 CPU clocked at 2.66Ghz and 8GB of memory.

5.6.1 MLCV bandwidth selection benchmarks

First we report the timings for computing the optimal bandwidth using MLCV with the two (CPU and GPU) implementations. Sample sets were chosen from the distribution of a 20 dimensional random vector $\boldsymbol{\xi} = [\xi_1, \dots, \xi_{20}]^t$ where each ξ_i is uniformly and independently distributed over $[0, 1]$. The sample sets generated were of size $N = 100, 1000, 20000, 50000, 100000$. Recall that each evaluation of the objective function (5.5) requires $\mathcal{O}(MN^2)$ flops. In addition, the implementation in CUDA requires the sample set to be transferred from main memory to the GPU over the PCIe bus, which introduces an additional cost for the GPU implementation. The timings reported for the GPU implementation include the cost of this data transfer. Also, only the objective function is computed using CUDA. The optimization routine for the both CUDA implementation and the serial implementation is performed in serial using the constrained optimization by linear approximation (COBYLA) method found in the *nlopt* software library [25]. This is a derivative-free optimization routine that approximates both the objective and constraint functions by linear functions.

Table 5.1 shows the time required by the serial algorithm and the GPU algorithm. The speedup displayed in Table 5.1 is defined to be the time required by the serial algorithm divided by the time required by the GPU algorithm. Figure 5.4 shows the same data as Table 5.1. From the figure it can be seen that both algorithms experience quadratic growth in complexity as the number of samples grows larger. This is in agreement with the asymptotic cost of evaluating the objective

function (5.5). The flat region of on the plot of the GPU timings represents the area where the processors on the GPU are not being fully utilized. Once the resources on the GPU are fully saturated the region of quadratic growth begins. Note that for the smallest sample set ($N = 100$), the serial algorithm runs faster than the GPU algorithm. This is due to the fact that there is some overhead associated with CUDA kernel invocation and the memory transfer between the host and device. In both cases however, the computation is essentially instantaneous. The execution of the 100000 sample case for the serial algorithm did not terminate in a reasonable time and was terminated.

N	Serial	GPU	Speedup
100	9.38×10^{-2}	1.85×10^{-1}	0.50
1000	1.02×10^1	1.42×10^{-1}	72
20000	4.31×10^3	6.25×10^1	69
50000	2.47×10^4	4.84×10^2	56
100000	--	1.63×10^3	--

Table 5.1: Time (in seconds) required to compute the optimal bandwidth using MLCV

5.6.2 Kernel density estimation benchmarks

Next we compared the serial and GPU implementation of kernel density estimation. In this example the number of samples is fixed, $N = 10000$ while the dimension M and the number of targets T is varied. The sample set consists of 10000 samples of the random vector $\boldsymbol{\xi} = [\xi_1, \dots, \xi_M]$ where each ξ_i is independently distributed over $[0, 1]$. The implementations were tested with $M = 5, 10, 15, 20$. For

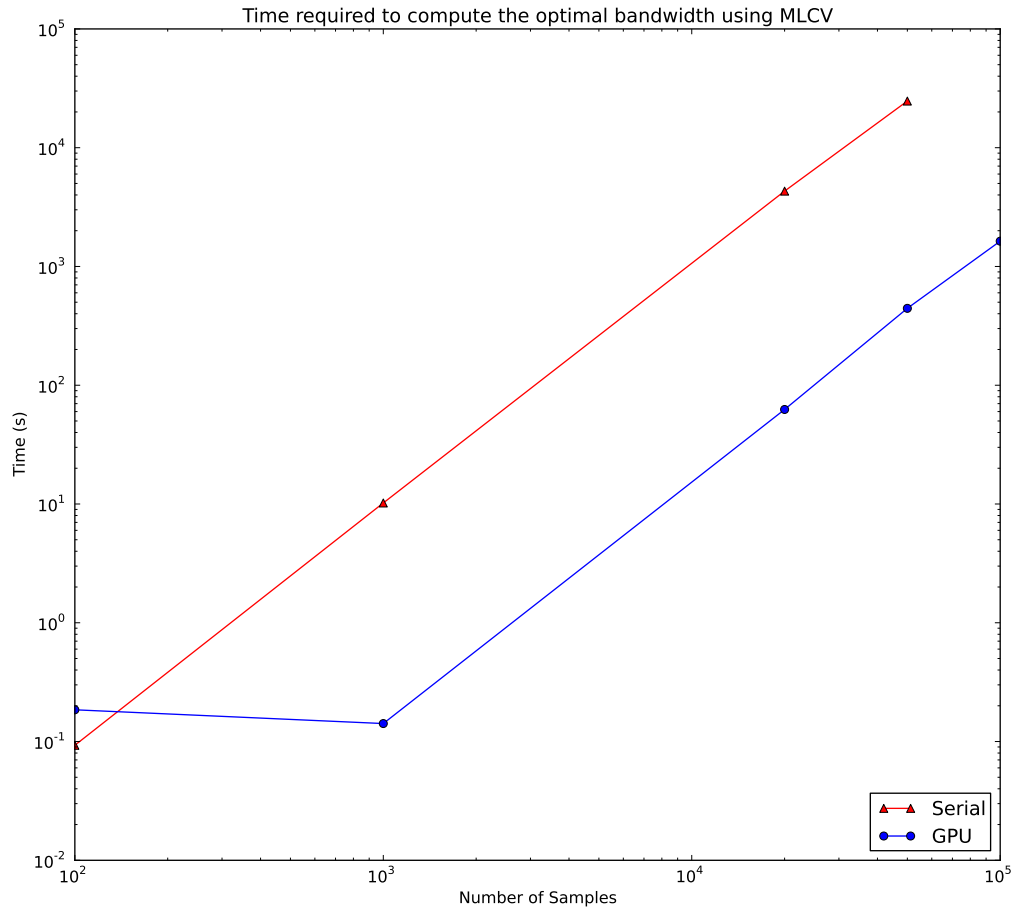


Figure 5.4: Time (in seconds) required to compute the optimal bandwidth using MLCV

each sample set the optimal bandwidth computed by MLCV was used. The cost associated with computing the optimal bandwidth is not included in the results measuring KDE performance. The time required to transfer the samples from the host to the GPU is also not included in this example since that cost would have already been incurred when computing the optimal bandwidth. The time required to transfer the target locations from the host to the GPU are included as well as the time required to transfer the result from the GPU back to the host. The tar-

get arrays are taken to be random samples on $[0, 1]^M$ of varying size. Figure 5.5, Tables 5.2, 5.3, 5.4, and 5.5 show the time required by the serial algorithm and the GPU algorithm to compute the kernel density estimate along with the associated speedup factor. The algorithm complexity grows linearly in the number of targets and the GPU implementation performs between one and two orders of magnitude faster than the serial implementation.

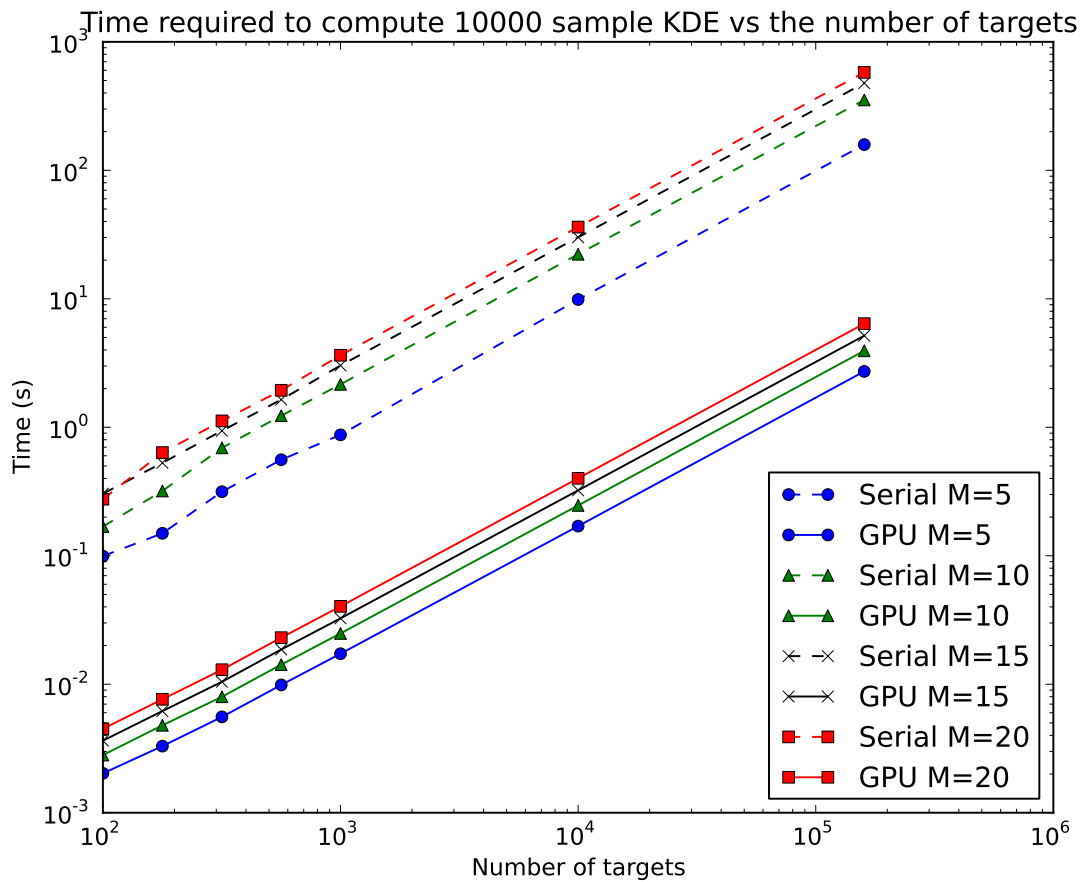


Figure 5.5: Time (in seconds) required to compute the kernel density estimate for varying dimension and number of targets

Number of Targets	Serial	GPU	Speedup
1.0×10^2	9.93×10^{-2}	2.02×10^{-3}	49
1.0×10^3	8.73×10^{-1}	1.72×10^{-2}	51
1.0×10^4	9.87×10^0	1.70×10^{-1}	58
1.6×10^5	1.59×10^2	2.72×10^0	58

Table 5.2: Time (in seconds) required to compute the KDE for a 5-dimensional parameter space

Number of Targets	Serial	GPU	Speedup
1.0×10^2	1.69×10^{-1}	2.81×10^{-3}	60
1.0×10^3	2.16×10^0	2.49×10^{-2}	87
1.0×10^4	2.22×10^1	2.46×10^{-1}	90
1.6×10^5	3.53×10^2	3.94×10^0	90

Table 5.3: Time (in seconds) required to compute the KDE for a 10-dimensional parameter space

Number of Targets	Serial	GPU	Speedup
1.0×10^2	3.04×10^{-1}	3.64×10^{-3}	84
1.0×10^3	3.02×10^0	3.26×10^{-2}	92
1.0×10^4	3.01×10^1	3.23×10^{-1}	93
1.6×10^5	4.77×10^2	5.16×10^0	92

Table 5.4: Time (in seconds) required to compute the KDE for a 15-dimensional parameter space

5.6.3 Adaptive KDE collocation benchmarks

In the following two examples we compare the performance of the GPU and serial algorithms applied to adaptive collocation with kernel density estimation. The

Number of Targets	Serial	GPU	Speedup
1.0×10^2	2.76×10^{-1}	4.50×10^{-3}	61
1.0×10^3	3.63×10^0	4.04×10^{-2}	90
1.0×10^4	3.62×10^1	4.01×10^{-1}	90
1.6×10^5	5.79×10^2	6.41×10^0	90

Table 5.5: Time (in seconds) required to compute the KDE for a 20-dimensional parameter space

test problem is given by

$$-\frac{d}{dx}(a_M(x, \boldsymbol{\xi}) \frac{d}{dx}u(x, \boldsymbol{\xi})) = 1, \quad \forall x \in (0, 1) \quad (5.14)$$

$$u(0, \boldsymbol{\xi}) = u(1, \boldsymbol{\xi}) = 0. \quad (5.15)$$

The diffusion coefficient a_M is defined for even M by

$$a_M = \mu + \sum_{k=0}^{M/2-1} \lambda_k (\xi_{2k} \cos(2\pi kx) + \xi_{2k+1} \sin(2\pi kx)), \quad (5.16)$$

where $\lambda_k = \exp(-k)$, $\mu = 3$ and ξ_k is uniformly distributed on $[0, 1]$.

In the first example the adaptive collocation procedure is performed for $M = 4, 10, 20$. The sample sets for this test were constructed to be of size $N = 1000, 20000$. The grid refinement criterion τ from (4.22) was defined to be $\tau = 1 \times 10^{-4}$. At each collocation point, (5.14) was discretized using finite differences on a uniform grid of size 200. Evaluating u at a collocation point therefore requires the solution of a 200 by 200 linear system. For both the GPU implementation and the serial implementation, the solution of this system was accomplished using the serial direct solver

included in numpy [1]. The total cost of the method can be divided into two parts, time spent evaluating the model $u(\boldsymbol{\xi})$ and time associated with the adaptive KDE collocation method. Since our focus is on attaining maximum performance in those parts of the algorithm related to the collocation method, in the timings presented below, the cost of solving the linear systems is not included. Therefore all that is being measured is the time required to compute the hierarchical surpluses, compute the kernel density estimates, and any overhead associated with the collocation grid data structures.

For both the serial and GPU implementations, the data structure used to store the collocation grid and the methods that refined the grid were handled in serial by Python. For the GPU algorithm, when a new level was added to the grid, Python passed the multi-indexes that described the location of the new collocation points, along with the function values at those points to CUDA. CUDA was then responsible for computing the new hierarchical surpluses, the norms of the new hierarchical surpluses, and the kernel density estimate at the new grid points. The norm of the hierarchical surpluses and the values from the kernel density estimate were passed back to python so that the grid could be adaptively refined. In both cases the kernel density estimate used to drive the refinement procedure used the optimal bandwidth obtained by MLCV. In the timings presented in Table 5.6, the cost of computing the MLCV bandwidth is not included.

Table 5.6 shows the time required by the two algorithms and Table 5.7 shows the number of collocation points required for each choice of M and N . These tables show that the algorithm in CUDA is capable of computing the hierarchical surpluses

of the interpolant significantly faster than a corresponding serial algorithm.

N						
	1000			20000		
M	Serial	CUDA	Speedup	Serial	CUDA	Speedup
4	0.745	0.036	21	1.228	0.03	41
10	52.826	1.089	49	67.46	1.325	51
20	61.383	1.285	48	361.385	4.607	78

Table 5.6: Time (in seconds) required to compute the interpolant using adaptive KDE collocation

	N	
M	1000	20000
4	756	685
10	5921	5689
20	6199	13425

Table 5.7: Collocation points required to compute the interpolant using adaptive KDE collocation

τ	Number of Collocation Points	Serial Time	GPU Time	Speedup
1×10^{-3}	2656	2.71×10^1	1.06×10^0	63
5×10^{-4}	4439	6.27×10^1	1.06×10^0	59
1×10^{-4}	14396	4.17×10^2	4.94×10^0	84
5×10^{-5}	23785	1.21×10^3	9.18×10^0	132

Table 5.8: Time (in seconds) required to compute the interpolant using adaptive KDE collocation with varying refinement criterion τ

In the next example, the sample set is fixed to consist of 20000 samples from a uniform distribution on $[0, 1]^{20}$ and the refinement tolerance τ is varied. As τ is decreased, the adaptive KDE collocation method requires more collocation points.

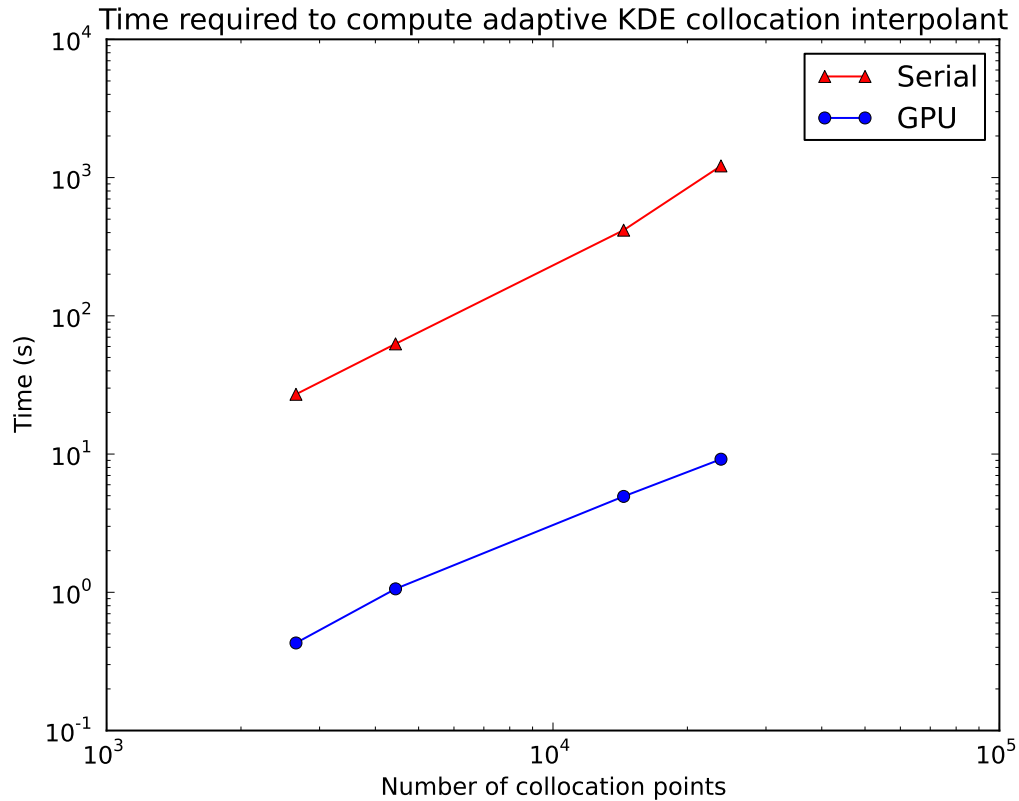


Figure 5.6: Time required to construct the hierarchical interpolant vs the number of collocation points

Thus as τ is decreased, the work required to construct the hierarchical interpolant increases. Figure 5.6 shows the time required to construct the hierarchical interpolant as a function of the number of collocation points. Figure 5.7 and shows the time required to construct the hierarchical interpolant as a function of the refinement tolerance τ . The data for both of these figures is also given in Table 5.8 along with the associated speedup factors. Again in this experiment the time required to compute u at each collocation point was not included in the timings. This experiment again shows that the algorithm run on the GPU scales favorably compared to the same algorithm run in serial.

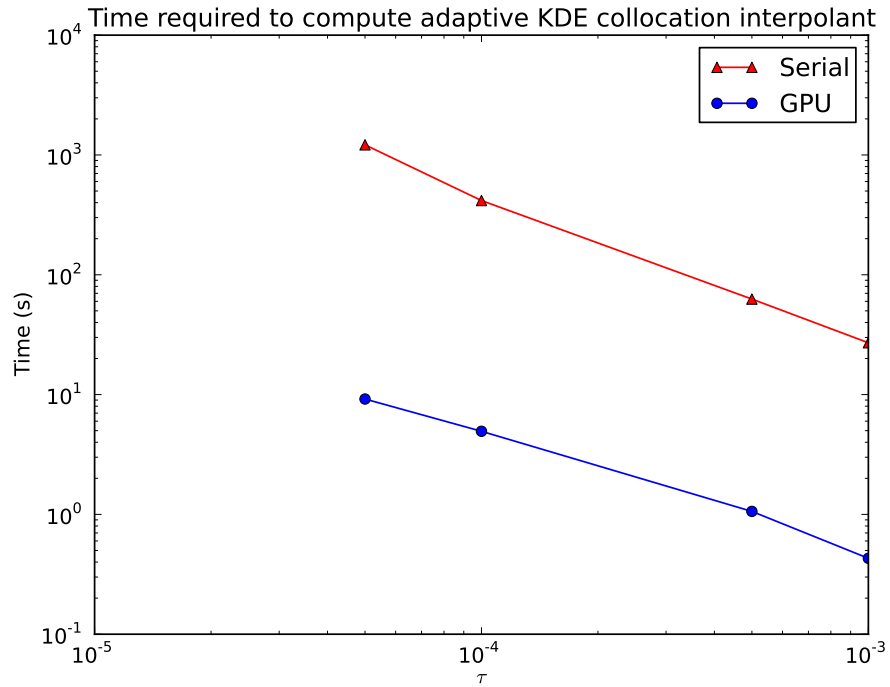


Figure 5.7: Time required to construct the hierarchical interpolant vs the refinement tolerance τ

5.7 Conclusions

In this chapter we presented an implementation of an adaptive KDE collocation method using NVIDIA’s CUDA platform. Several components of this method were parallelized to attain high performance. In particular parallel implementations of kernel density estimation and automatic bandwidth selection using MLCV showed significant gains over corresponding serial implementations. These two methods have wide applicability in addition to their use in the collocation method. In addition the computation of the expansion coefficients for the adaptive interpolant also experienced significant speedups when performed in parallel. The parallel implementations of these methods allow for approximations with many collocation

points to be constructed without significant overhead associated with constructing very fine hierarchical interpolants.

Chapter 6

Summary and Conclusions

In this thesis we explored the performance of a variety of methods for computing the solution to stochastic partial differential equations. These equations arise in science and engineering contexts when there is insufficient data to fully specify the model of some physical system. In such scenarios it is desired to describe the inputs as uncertain quantities and then to propagate this uncertainty through the model, in order to quantify the uncertainty in the model output. We also proposed and analyzed a new method, the adaptive KDE collocation method, and presented fast implementations of this method using NVIDIA's CUDA architecture for GPU computing.

In chapter 3 we compared the performance of the stochastic Galerkin and stochastic sparse grid collocation techniques for solving the stochastic diffusion equation. These techniques produce approximate solutions that lie in a similar approximation space and thus attain similar accuracy. The stochastic Galerkin method requires the solution to a single large linear system whereas the stochastic sparse grid collocation method requires the solution to many uncoupled smaller linear systems. It was shown that when a preconditioner based on the mean of the diffusion operator was used for the large system, stochastic Galerkin method was more computationally efficient than the stochastic sparse grid collocation method, and that

the advantage of the Galerkin method grew as the dimension of the input operator grew.

The use of the methods presented in chapter 3 depend on two very strong assumptions, first, that the solution is analytic with respect to the random input parameters, and second, that the random parameters that specify the problem are independent and that their probability distributions are known. In many cases the solution may exhibit discontinuities, steep gradients, or other strongly local features, that preclude the use of the global approximation techniques presented in chapter 3. It is also generally not the case that the random parameters that specify the problem are independent and in cases where the parameter values are only available from a finite set of independent data the joint probability density is unknown.

In chapter 4 we presented an adaptive KDE collocation method, which can compute the solution statistics of a SPDE solution under very general assumptions. In particular, the solution is only required to be almost everywhere Lipschitz continuous and the statistics of the random parameters were only available from the values of a finite random sample. The adaptive KDE collocation method was shown to allocate collocation points so as to resolve the solution behavior in areas of the parameter space where the solution displayed irregularities that were also near the sample points of the data. This makes the computed solution to be more accurate near the sample points. When the Monte-Carlo method was performed on the computed solution, estimates of the solution statistics were obtained whose bias was an order of magnitude smaller than the confidence bound on the Monte-Carlo error. This was achieved with many fewer PDE solves than there were samples of the pa-

rameters, thus obtaining a significant savings in computational work in comparison to the standard Monte-Carlo methods.

The method presented in chapter 4 contains several sub-tasks that are computationally expensive. In chapter 5 we presented implementations of MLCV, KDE, and adaptive KDE collocation that executed using NVIDIA's CUDA architecture. The computational tasks involved in adaptive KDE collocation are all easily parallelism and well suited for running on GPU's. It was shown that the algorithms programmed in parallel for CUDA could significantly outperform the associated serial algorithms even when the overhead associated with memory traffic between the host CPU and the GPU was taken into account.

Appendix A

Source code listings for implementaion of adaptive KDE collocation in CUDA

In the appendix we provide the full source code for the CUDA C implementations of kernel density estimation with maximum likelihood cross validation and hierarchical interpolation. Updated versions of this software will be available for download from the author's website at <http://www.math.umd.edu/~cmiller>. If this webpage becomes unavailable please contact the author to obtain a copy of the software. This software requires a computer with an NVIDIA GPU of compute capability at least 2.0.

A.1 CUDA kernel density estimation code

The CUDA C code that performs the kernel density estimation and maximum likelihood cross validation is contained in the files `KDE.hpp` and `KDE.cu`.

```
/**
 * @file KDE.hpp
 * @author Christopher Miller <cmiller@math.umd.edu>
 * @version 1.0
 * @section LICENSE
 * CudaKDE: Cuda code for the evaluation of kernel density
 * estimates
 * Copyright (c) 2011, Christopher Miller.
 * This software is distributed under the GNU Lesser General
 * Public License V3. For more information, see the README
 * file in the top CudaKDE directory.
```

```

*
* @brief Class definitions for the KernelDensityEstimator
*   class.
* @section DESCRIPTION
*
* The KernelDensityEstimator class computes kernel density
* estimates for a given data set and a given target set.
* The class can also compute the sample mean and sample
* covariance matrix of a sample set.
*/

#ifndef KDE_HPP
#define KDE_HPP

#include <assert.h>
#include <math.h>
#include <nlopt.h>
#include <iostream>
#include "cublas.h"

///Define the thread block size.
#define BLOCK_SIZE 16

/** @brief Class for evaluating kernel density estimates for
    a given data set.

    See B.W. Silverman, 'Density Estimation for
    Statistics and Data Analysis', Chapman and Hall, 1986.
*/

namespace cudaKDE{

class KernelDensityEstimator
{
public:
    //
    //- Heading: Constructor and destructor
    //

    /// Default constructor.
    /** @param samples A num_samples by dimension array of
        floats stored in row major format, i.e. samples[i][j]
        = samples[j + dimension*i]. The (i,j) entry of
        samples contains the jth coordinate of the ith sample
        data point.
    */

```

```

    @param num_samples_ The number of samples in the data
        set.
    @param dimension_ The dimension of each data point.
    @param bandwidth_ The bandwidth of the kernel density
        estimate. Currently only scalar bandwidths are
        supported.
    @param num_expected_targets The maximum number of
        targets expected during a single evaluation of the
        KDE. This tells the CUDA device how much memory to
        allocate for the targets and the result.
*/
KernelDensityEstimator(float const* samples,
                       const unsigned int num_samples_,
                       const unsigned int dimension_,
                       const float bandwidth_,
                       const unsigned int num_expected_targets = 1);

/// Default deconstructor
/** Frees arrays residing on CUDA device.
*/
~KernelDensityEstimator();

/** @brief Method evaluates the KDE at the targets and
    returns the result.

    @param targets A num_targets by dimension array of the
        target points stored in row major order. targets[j +
        dimension*i] contains the jth coordinate of the ith
        target.
    @param num_targets The number of targets. If
        num_targets is greater than max_num_targets then memory
        is reallocated on the device.
    @param result On start, result is a array of size
        num_targets. On finish, result[i] contains the kernel
        density estimate evaluated at the ith target.
*/
void evaluateKDE(const float * targets,
                 const unsigned int num_targets,
                 float* result);

/// Set the bandwidth of the approximation.
/**
    @param bandwidth_ The new bandwidth of the kernel
        density estimate.
*/

```

```

void setBandwidth(const float bandwidth_);

///  

/**  

    @param result On start, result is a array of size  

    num_samples. On finish, result[i] contains the kernel  

    density estimate evaluated at the ith sample.  

*/  

void evaluateKDE(float* result);

/** @brief Attempt to set the bandwidth by using maximum  

    likelihood cross-validation.  

    For a description of maximum likelihood cross-validation  

    See B.W. Silverman, 'Density Estimation for Statistics  

    and Data Analysis', Chapman and Hall, 1986.  

    @return The new bandwidth.  

*/  

float setBandwidthMLCV();

///  

/**  

    @return The dimension of the sample data points.  

*/  

unsigned int getDimension();

///  

/**  

    @return The number of sample data points.  

*/  

unsigned int getNumSamples();

///  

/**  

    @param result On start, if returnResult is true then  

    result is a float array of size at least dimension.  

    If return result is false then result is not  

    referenced and may be NULL. On exit, if return  

    result is true then result[i] contains the sample  

    mean of the ith random variable.  

    @param returnResult If true the computed sample mean  

    is copied from the device into result. If false then  

    result is not referenced.  

*/

```

```

void calculateSampleMean(float* result , bool returnResult);

/** @brief Calculates the sample covariance matrix of the
    sample set.

    @param result On start , if returnResult is true then
    result is a float array of size at least
    0.5*(dimension)*(dimension+1). If returnResult is
    false then result is not referenced and may be NULL.
    On finish , if returnResult is true , result contains
    the upper tringular part of the sample covariance
    matrix stored in row major format. That is
    @f$C(i,j) = result [(i-1)*dimension - i*(i-1)/2 + j]@f$
    for @f$j\geq i@f$.

    @param returnResult If true the sample covariance
    matrix is copied from the device into result .
    If false then result is not referenced.
*/
void calculateSampleCovariance(float* result ,
                               const bool returnResult);

/// Returns the MLCV score for a given value of bandwidth.
/** The value of x that minimizes this function is the
    maximum likelihood cross-validation bandwidth. This
    function is called by the nlopt optimizer.
    @param n The dimension of the input. Needed by
    Nlopt but currently always equal to 1.
    @param x x[0] is the argument to the objective
    function .
    @param grad Gradient of the objective function.
    Currently not used and always set to NULL.
    @param my_func_data A pointer to the
    KernelDensityEstimator object for which we're
    attempting to find the MLCV bandwidth.
*/
static double MLCVScore(unsigned int n,
                        const double *x,
                        double* grad ,
                        void * my_func_data);

private:

///Sample data set on the device.
float* d_samples;

```

```

    ///The number of samples.
    unsigned int num_samples;

    ///The dimension of each sample.
    unsigned int dimension;

    ///The pitch size in bites for the sample array.
    size_t samples_pitch;

    ///The bandwidth of the estimator.
    float bandwidth;

    ///Array allocated to contain the targets.
    float* d_targets;

    /** @brief The number of targets currently allocated for
        the d_targets array.
    */
    unsigned int max_num_targets;

    ///The pitch size in bites for the target array.
    size_t targets_pitch;

    /** @brief Allocated array for passing back the results
        from the device.
    */
    float* d_result;

    ///Allocated array for storing the sample mean.
    float* d_mean;

    ///Bool that checks if the sample mean has been computed.
    bool sampleMeanComputed;

    /**@brief Array that stores the upper triangle of the
        covariance matrix in row major order.
    */
    float* d_cov;

    /** @brief Bool that checks if the sample covariance has
        been computed.
    */
    bool sampleCovComputed;
};

```

```

} //namespace cudaKDE

/// CUDA kernel for evaluating the kernel density estimate.
/** @param targets A device pointer to a num_targets by
    dimension pitched array
    containing the locations where the KDE will be
    evaluated. targets[j + i*targets_pitch/sizeof(float)]
    is the @f$j^{th}@f$ coordinate of the @f$i^{th}@f$
    target.

    @param dimension The dimension of the targets and
    samples.
    @param num_targets The number of targets.
    @param targets_pitch The pitch size of the targets array
    in bytes. The leading dimension of targets is
    targets_pitch/sizeof(float).
    @param samples A device pointer to a num_samples by
    dimension pitched array containing the sample set.
    samples[j + i*samples_pitch/sizeof(float)] is the
    @f$j^{th}@f$ coordinate of the @f$i^{th}@f$ sample.
    @param samples_pitch The pitch size of the samples
    array in bytes. The leading dimension of samples is
    samples_pitch/sizeof(float).
    @param bandwidth The bandwidth of the kernel density
    estimate.
    @param result On start, a device pointer to an array
    of size num_targets. On finish, result[i] contains
    the kernel density estimate evaluated at the
    @f$i^{th}@f$ target
*/
__global__ void KDE_cuda_kernel(const float * targets,
                                const unsigned int dimension,
                                const unsigned int num_targets,
                                const size_t targets_pitch,
                                const float * samples,
                                const unsigned int num_samples,
                                const size_t samples_pitch,
                                const float bandwidth,
                                float* result);

#endif

```

```

/**
 * @file KDE.cu
 * @author Christopher Miller <cmiller@math.umd.edu>
 * @version 1.0
 * @section LICENSE
 * CudaKDE: Cuda code for the evaluation of kernel density
 * estimates
 * Copyright (c) 2011, Christopher Miller.
 * This software is distributed under the GNU Lesser General
 * Public License V3. For more information, see the README
 * file in the top CudaKDE directory.
 *
 * @brief Implementation of the KernelDensityEstimator class
 * and the CUDA kernel that performs the density estimation.
 * @section DESCRIPTION
 *
 * Implementation of the KernelDensityEstimator class
 */

#include "KDE.hpp"

using namespace cudaKDE;

KernelDensityEstimator::
KernelDensityEstimator(const float * samples,
                       const unsigned int num_samples_,
                       const unsigned int dimension_,
                       const float bandwidth_,
                       const unsigned int num_expected_targets)
{
    num_samples = num_samples_;
    dimension = dimension_;

    //Allocate memory for the samples on the device.
    cudaError_t error_catcher =
        cudaMallocPitch((void**) &d_samples,
                       &samples_pitch,
                       dimension*sizeof(float),
                       num_samples);

    if ( error_catcher != cudaSuccess ) {
        std::cout << "Error: could not allocate memory for"
            << " samples on device." << std::endl;
        throw error_catcher;
    }
}

```

```

}

//Copy the samples from the host to the device.
error_catcher = cudaMemcpy2D(d_samples,
                             samples_pitch,
                             samples,
                             dimension*sizeof(float),
                             dimension*sizeof(float),
                             num_samples,
                             cudaMemcpyHostToDevice);

if (error_catcher != cudaSuccess) {
    std::cout << "Error: could not copy samples from host"
    <<" to device" << std::endl;
    throw error_catcher;
}

//Allocate memory for the targets on the device.
max_num_targets = num_expected_targets;
error_catcher = cudaMallocPitch((void *)&d_targets,
                                &targets_pitch,
                                dimension*sizeof(float),
                                max_num_targets);

if ( error_catcher != cudaSuccess) {
    std::cout << "Error: could not allocate memory for "
    << "targets on device." << std::endl;
    throw error_catcher;
}

//Allocate space for the result on the device.
cudaMalloc((void**) &d_result,
           max_num_targets*sizeof(float));

bandwidth = bandwidth_;
sampleMeanComputed = false;
sampleCovComputed = false;
}

KernelDensityEstimator::~KernelDensityEstimator()
{
    //Free all arrays allocated on the cuda device.

```

```

    cudaFree(d_samples);
    cudaFree(d_targets);
    cudaFree(d_result);
    if( sampleMeanComputed ) cudaFree(d_mean);
    if( sampleCovComputed ) cudaFree(d_cov);
}

void KernelDensityEstimator::
setBandwidth(const float bandwidth_)
{
    bandwidth = bandwidth_;
}

void KernelDensityEstimator::
evaluateKDE(const float * targets ,
            const unsigned int num_targets ,
            float* result)
{
    cudaError_t error_catcher;

    /*Check if we need to reallocate the d_targets and
    d_results arrays. */
    if ( num_targets > max_num_targets ) {
        max_num_targets = num_targets;
        cudaFree(d_targets);
        error_catcher = cudaMallocPitch((void **) &d_targets ,
                                       &targets_pitch ,
                                       dimension*sizeof(float) ,
                                       num_targets);

        if ( error_catcher != cudaSuccess) {
            std::cout << "Error: could not allocate memory for "
                << "targets on device." << std::endl;
            throw error_catcher;
        }

        cudaFree(d_result);
        error_catcher =
            cudaMalloc((void **) &d_result , num_targets*
                sizeof(float));

        if ( error_catcher != cudaSuccess) {

```

```

        std::cout << "Error: could not allocate memory for "
        << "results on device." << std::endl;
        throw error_catcher;
    }
}

//Transfer the targets from the host to the device.
error_catcher = cudaMemcpy2D(d_targets ,
                             targets_pitch ,
                             targets ,
                             dimension*sizeof(float) ,
                             dimension*sizeof(float) ,
                             num_targets ,
                             cudaMemcpyHostToDevice);

if ( error_catcher != cudaSuccess) {
    std::cout << "Error: could not copy targets from host "
    << "to device." << std::endl;
    throw error_catcher;
}

/*Set up the grid.  Each block handles BLOCK_SIZE targets
and BLOCK_SIZE samples.
*/
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
dim3 dimGrid(ceil(num_samples/(float) BLOCK_SIZE) ,
             ceil(num_targets/(float) BLOCK_SIZE));
size_t shared_mem_size = (BLOCK_SIZE*samples_pitch +
                          BLOCK_SIZE*targets_pitch +
                          BLOCK_SIZE*sizeof(float));

//Zero the result
float f = 0.0f;

/*Writes 0 to every byte in result.  Don't currently have a
floating point version of this function
*/
cudaMemset(d_result ,
           reinterpret_cast<int*>(f) ,
           sizeof(float)*num_targets);

//Invoke the CUDA kernel to compute the kernel density
//estimate.
KDE_cuda_kernel<<<<dimGrid , dimBlock , shared_mem_size>>>
    (d_targets , dimension , num_targets , targets_pitch ,

```

```

        d_samples , num_samples , samples_pitch , bandwidth ,
        d_result );

//Copy results from the device back to the host.
cudaMemcpy(result ,
            d_result ,
            num_targets*sizeof(float) ,
            cudaMemcpyDeviceToHost);

}

void KernelDensityEstimator::evaluateKDE(float* result)
{
    cudaError_t error_catcher;

    //Make room for the result on the device
    if ( num_samples > max_num_targets ) {
        cudaFree(d_result);
        error_catcher =
            cudaMalloc((void **) &d_result , num_samples *
                      sizeof(float));

        if ( error_catcher != cudaSuccess) {
            std::cout << "Error: could not allocate memory for "
                      << "results on device." << std::endl;
            throw error_catcher;
        }
    }

    //Set up the grid
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
    dim3 dimGrid(ceil(num_samples/(float) BLOCK_SIZE) ,
                ceil(num_samples/(float) BLOCK_SIZE));
    size_t shared_mem_size = (BLOCK_SIZE*dimension +
                              BLOCK_SIZE*dimension +
                              BLOCK_SIZE)*sizeof(float);

    //Zero the result.
    float f = 0.0f;
    cudaMemset(d_result , reinterpret_cast<int&>(f) ,
              sizeof(float)*num_samples);

    //RELEASE THE KRACKEN!!!!!!!!!!!!
    KDE_cuda_kernel<<<<dimGrid , dimBlock , shared_mem_size>>>
        (d_samples , dimension , num_samples , samples_pitch ,

```

```

        d_samples , num_samples , samples_pitch , bandwidth ,
        d_result );

//Copy the result from the device to the host.
cudaMemcpy(result ,
            d_result ,
            num_samples*sizeof(float) ,
            cudaMemcpyDeviceToHost);

}

float KernelDensityEstimator::setBandwidthMLCV()
{
    /*This function sets up a derivative free optimizer
    to optimize MLCVScore().
    */
    const float old_bandwidth = bandwidth;
    double lb[1] = {0}; //The lower bound for optimization

    //create a new nlopt_opt object.
    nlopt_opt opt;
    opt = nlopt_create(NLOPT_LN_COBYLA, 1);

    //Constrain that the optimized bandwidth must be positive.
    nlopt_set_lower_bounds(opt, lb);

    /*Set nlopt to minimize MLCVScore.
    The function pointer cannot be a member function
    so MLCVScore is defined to be static and this
    is passed as an argument to MLCVScore.
    */
    nlopt_set_min_objective(opt, MLCVScore, this);

    //Set the convergence tolerance.
    nlopt_set_xtol_rel(opt, 1e-4);

    //Set an initial guess.
    double x[1] = {1};

    double minf; //minf catches the minimum function value.

    //run the optimizer. argmin(MLCVScore()) is stored in x.
    int code = nlopt_optimize(opt,x,&minf);

```

```

if (code < 0) {
    std::cout << "nlopt failed!" << std::endl;
    bandwidth = old_bandwidth;
} else {
    std::cout << "nlopt success! New bandwidth = " << x[0]
    << std::endl;
    bandwidth = x[0];
    std::cout << "nlopt code = " << code << std::endl;
}
return bandwidth;
}

unsigned int KernelDensityEstimator::getDimension()
{
    return dimension;
}

unsigned int KernelDensityEstimator::getNumSamples()
{
    return num_samples;
}

double KernelDensityEstimator::MLCVScore(unsigned int n,
                                           const double *x,
                                           double* grad,
                                           void * my_func_data)
{
    /*Arguments to the objective function are passed through
    my_func_data. We pass a KernelDensityEstimator to the
    objective function.
    */
    KernelDensityEstimator* my_estimator =
        (KernelDensityEstimator*) my_func_data;
    my_estimator->setBandwidth( (float) x[0] );
    const int num_samples = my_estimator->getNumSamples();
    const int dimension = my_estimator->getDimension();

    //Evaluate the KDE at it's own samples
    float* result = (float *) malloc(sizeof(float) *
                                     num_samples);
    my_estimator->evaluateKDE(result);

    /*The MLCV score is the average log likelihood of the
    KDE's with the ith sample removed, evaluated at the ith
    sample. We evaluate the KDE at all of it's samples and

```

```

    subtract K(0) which is equivalent.
*/
float score = 0;
for ( unsigned int idx = 0; idx < num_samples; ++idx ) {
    score -= log ((num_samples/(num_samples-1))*result[idx] -
                 pow(.75, dimension)/((num_samples-1)*
                 pow(x[0], dimension)));
}
score /= num_samples;
free(result);
return (double) score;
}

```

```

void KernelDensityEstimator::
calculateSampleMean(float* result, bool returnResult)
{
    /*We only compute the sample mean once and store the result
    on the device. The sample mean is computed by taking the
    samples arranged in column major order and taking the
    matrix-vector product with a vector of all ones and
    normalizing the result.

    If the samples are stored column major in a matrix X then
    the sample mean is computed by  $\bar{x} = (1/N)X*1$ , where
    1 is the vector of length dimension with a one in each
    entry.
    */

    if (!sampleMeanComputed) {
        //There must be a better way to do this.
        float * ones = (float *) malloc( num_samples*
                                         sizeof(float) );
        for ( unsigned int idx = 0; idx < num_samples; ++idx ) {
            ones[idx] = 1;
        }
        float * d_ones;
        cudaMalloc( (void **) &d_ones, num_samples*
                  sizeof(float) );
        cudaMemcpy(d_ones,
                  ones,
                  num_samples*sizeof(float),
                  cudaMemcpyHostToDevice);

        cudaMalloc( (void **) &d_mean, dimension*sizeof(float) );
    }
}

```



```

//See the cuda blas documentation. This does the MV
//product discussed above.
cublasSgemv( 'N',
            dimension ,
            num_samples ,
            1/(float) num_samples ,
            d_samples ,
            samples_pitch/sizeof(float) ,
            d_ones ,
            1 ,
            0.0 ,
            d_mean ,
            1);

sampleMeanComputed = true;
free(ones);
cudaFree(d_ones);
}

//Sometimes you just need the mean on the device and don't
//need it passed back.
//Set returnResult to false if you don't need the result
//to be passed back to the host.
if( returnResult ) {
    cudaMemcpy( result ,
              d_mean ,
              dimension*sizeof(float) ,
              cudaMemcpyDeviceToHost );
}

}

void KernelDensityEstimator::
calculateSampleCovariance(float* result ,
                        const bool returnResult)
{

/*We only compute the sample covariance once and store the
result on the device.

```

If the samples are stored column major in a matrix X then the sample covariance is computed in three steps. First: The data is mean normalized. This is accomplished

with the rank one update $X = X - \bar{x} * 1^t$, where \bar{x} is the sample mean vector and 1^t is a vector of all ones. Second: The covariance is computed as a rank-k update $C = (1/\text{num_samples}-1)X * X^t$. Third: The data is un-normalized.

```

    X = X + \bar{x} * 1^t
*/

if ( !sampleCovComputed ) {
    float * ones = (float *) malloc( num_samples *
                                     sizeof(float) );
    for ( unsigned int idx = 0; idx < num_samples; ++idx ) {
        ones[idx] = 1;
    }

    //First we need to mean normalize the data.  If the data
    // is stored column major, mean normalizing is the rank
    // one update samples = samples - sample_mean*ones.

    float * d_ones;
    cudaMalloc( (void **) &d_ones, num_samples *
                sizeof(float) );
    cudaMemcpy( d_ones,
                ones,
                num_samples * sizeof(float),
                cudaMemcpyHostToDevice);

    // If we haven't done the sample mean yet compute it and
    // leave the result on the device.
    if( !sampleMeanComputed ) {
        calculateSampleMean(NULL, false);
    }

    // Mean normalize the data.  Rank-1 update.
    cublasSger( dimension,
                 num_samples,
                 -1.0,
                 d_mean,
                 1,
                 d_ones,
                 1,
                 d_samples,
                 samples_pitch / sizeof(float));

    //The covariance matrix is

```

```

//1/(n-1)*\sum (samples - mean)*(samples-mean)^T
//Which is just a rank k update.
float * C;
size_t cov_pitch;
cudaMallocPitch((void**) &C,
                &cov_pitch,
                dimension*sizeof(float),
                dimension);

cublasSsyrk('l',
            'N',
            dimension,
            num_samples,
            1.0/(num_samples - 1),
            d_samples,
            samples_pitch/sizeof(float),
            0.0,
            C,
            cov_pitch/sizeof(float));

//Unnormalize the data.
cublasSger(dimension,
           num_samples,
           1.0,
           d_mean,
           1,
           d_ones,
           1,
           d_samples,
           samples_pitch/sizeof(float));

//We store the symmetric C as a vector only storing the
//Upper triangle as a flattened array in row major order.

cudaMalloc( (void**) &d_cov,
            sizeof(float)*(dimension)*(dimension+1)/2 );
unsigned int location = 0;
for ( unsigned int i = 0; i < dimension; ++i ) {
    cudaMemcpy( &d_cov[location],
                &C[i*cov_pitch/sizeof(float) + i],
                sizeof(float)*(dimension-i),
                cudaMemcpyDeviceToDevice);
    location += dimension-i;
}
free(ones);

```

```

    cudaFree(d_ones);
    cudaFree(C);
}
if ( returnResult ) {
    cudaMemcpy(result ,
               d_cov ,
               sizeof(float)*(dimension)*(dimension+1)/2,
               cudaMemcpyDeviceToHost);
}
}
}

__global__ void KDE_cuda_kernel(const float* targets ,
                                const unsigned int dimension ,
                                const unsigned int num_targets ,
                                const size_t targets_pitch ,
                                const float* samples ,
                                const unsigned int num_samples ,
                                const size_t samples_pitch ,
                                const float bandwidth ,
                                float* result)
{
    /**Shared array for storing the samples needed by a single
        thread block.
    */
    extern __shared__ float sampleSub[];

    //Get the block indices and thread indices
    const int mySampleBlock = blockIdx.x;
    const int myTargetBlock = blockIdx.y;
    const int mySample = threadIdx.x;
    const int myTarget = threadIdx.y;

    //The global index of the sample and target for this
    //thread.
    const int global_sample_num = BLOCK_SIZE*mySampleBlock +
                                   mySample;
    const int global_target_num = BLOCK_SIZE*myTargetBlock +
                                   myTarget;

    int num_samples_this_block;
    int num_targets_this_block;

```

```

//Check to see if we're dealing with a full block of
//samples and a full block of targets. The last block in
//each dimension may not be full if num_samples or
//num_targets doesn't divide BLOCK_SIZE.
if ( blockIdx.x ==
    ceilf(num_samples/(float) BLOCK_SIZE) - 1 ) {
    num_samples_this_block =
        ((num_samples % BLOCK_SIZE) == 0 ) ?
        BLOCK_SIZE : num_samples%BLOCK_SIZE;
} else num_samples_this_block = BLOCK_SIZE;

if ( blockIdx.y ==
    ceilf(num_targets/(float) BLOCK_SIZE) - 1 ) {
    num_targets_this_block =
        ((num_targets % BLOCK_SIZE) == 0 ) ?
        BLOCK_SIZE : num_targets%BLOCK_SIZE;
} else num_targets_this_block = BLOCK_SIZE;

//First we want to download the samples and targets needed
// by this block from global memory to shared memory.
float* targetSub =
    (float*) &sampleSub[num_samples_this_block*dimension];
float* resultShared =
    (float*) &targetSub[num_targets_this_block*dimension];

//The diagonal threads in each block get their sample and
//target from global memory and place them in shared
//memory.
if ( mySample == myTarget ) { //Load samples and targets
    //into shared memory.
    resultShared[myTarget] = 0;
    if ( global_sample_num < num_samples ) {
        const float * sample = samples +
            global_sample_num*
            samples_pitch/sizeof(float);
        for ( int idx = 0; idx < dimension; ++idx ) {
            sampleSub[dimension*mySample + idx] = sample[idx];
        }
    }
}

if ( global_target_num < num_targets ) {
    const float * target = targets +
        global_target_num*
        targets_pitch/sizeof(float);
}

```

```

        for ( int idx = 0; idx < dimension; ++idx ) {
            targetSub[dimension*myTarget + idx] = target[idx];
        }
    }
}

//Make sure everything is in shared memory before moving
//forward.
__syncthreads();

//Each thread computes K(target-sample/bandwidth
//If we're in the last block, past the
//end of the samples or targets, don't
//do anything.
if (global_sample_num < num_samples &&
    global_target_num < num_targets) {

    float local_result = 1;

    float * sample = &(sampleSub[mySample*dimension]);
    float * target = &(targetSub[myTarget*dimension]);
    for (int idx = 0; idx < dimension; ++idx) {
        float u = (target[idx] - sample[idx])/bandwidth;
        local_result *= (fabsf(u) < 1) ? .75*(1-u*u) : 0;
    }

    local_result *=
        1/(num_samples*powf(bandwidth, (float)dimension));

    //Add this result to a local accumulation.
    atomicAdd(&resultShared[myTarget], local_result);
    __syncthreads();

    //Once everyone is finished add the results from this
    //block to the result.
    if ( threadIdx.x == 0) {
        atomicAdd(&result[global_target_num],
            resultShared[myTarget]);
    }
    //End Kernel
}
}
}

```

A.2 Cuda adaptive collocation code

The CUDA C code that performs adaptive collocation is contained in the files `LocallyRefinableDriver.hpp` and `LocallyRefinableDriver.cu`.

```
/**
 * @file LocalRefinableDriver.hpp
 * @author Christopher Miller <cmiller@math.umd.edu>
 * @version 1.0
 * @section LICENSE
 * CudaKDE: Cuda code for the evaluation of kernel density
 * estimates
 * Copyright (c) 2011, Christopher Miller.
 * This software is distributed under the GNU Lesser General
 * Public License V3. For more information, see the README
 * file in the top CudaKDE directory.
 *
 * @brief Class definitions for the LocalRefinableDriver
 * class.
 * @section DESCRIPTION
 *
 * The LocalRefinableDriver class computes the hierarchical
 * interpolant described in X. Ma and N. Zabaras. "An adaptive
 * hierarchical sparse grid collocation algorithm for the
 * solution of stochastic differential equations".
 */

#ifndef LOCALREFINABLEDRIVER_HPP
#define LOCALREFINABLEDRIVER_HPP

#include "cublas.h"
#include <iostream>
#define BLOCK_SIZE 16

/// Class for evaluating hierarchical interpolants.
namespace cudaColloc{
    class CudaApproximationGrid{
    public:

        /// Default constructor.
        /**
```

```

    @param input_dim_ The dimension of the parameter
        space.
    @param output_dim_ The dimension of the solution.
    @param max_num_points_ The maximum number of points to
        be used in constructing the interpolant.
    @param endpoints A array of length 2*input_dim.
        Interpolation is performed on
        [endpoints[0],endpoints[1]] X
        [endpoints[2],endpoints[3]]...
**/
    CudaApproximationGrid(const unsigned int input_dim_,
                          const unsigned int output_dim_,
                          const unsigned int max_num_points_,
                          float const * endpoints);

/// Destructor
~CudaApproximationGrid();

/// Adds a new level to the hierarchical interpolant.
/** Transfers a set of new function values and multi
    indices to the CUDA device. Calls
    computeNewSurpluses() to update the hierarchical
    surplus array.
    @param num_new_points The number of new collocation
        points
    @param new_levels An array of size input_dim*
        num_new_points that specifies the level
        multi-indices for the new points.
    @param new_indexes An array of size input_dim*
        num_new_points that specifies the index
        multi-indices for the new points.
    @param new_values An array of size num_new_points*
        output_dimension that contains the function value
        at the new collocation points.
*/
void addLevel(const unsigned int num_new_points,
             unsigned int const* new_levels,
             unsigned int const* new_indexes,
             float const* new_values);

///Computes the value of the hierarchical interpolant.
/** @param x An array of size num_evals*input_dim.
    @param num_evals The number of points where the
        interpolant is evaluated.
    @param y An array of size num_evals*output_dim to

```



```

        store the result.
*/
void getInterpolantValue(float const* x,
                        const unsigned int num_evals,
                        float* y);

///Returns the output_dim.
unsigned int getOutputDim();

///Computes the norm of the hierarchical surpluses.
/** @param Norms an array of size stop-start+1 to store
    the norms.
    @param start the index of the first collocation point
    to compute the norm of.
    @param stop the index of the last collocation point
    to compute the norm of.
*/
void normSurpluses(float* norms,
                  const unsigned int start,
                  const unsigned int stop);

private:

///Computes a new set of hierarchical surpluses.
/** @param num_new_points The number of points added to
    the grid by addLevel().
*/
void computeNewSurpluses(const unsigned int
                        num_new_points);

unsigned int input_dim;
unsigned int output_dim;
unsigned int max_num_points;
int current_level;
unsigned int num_current_points;

///Device array to store the endpoints of interpolation.
float * d_endpoints;

///Pitched device array to store the hierarchical
///surpluses.
float * d_surpluses;
size_t surplus_pitch;

///Pitched device array to store the level multi-indexes.

```

```

    unsigned int * d_levels;
    size_t levels_pitch;

    ///Pitched device array to store the index multi-indexes.
    unsigned int * d_indexes;
    size_t indexes_pitch;

};
}

///Kernel for computing the hierarchical basis at an
///arbitrary point set.
/** @param x An array of size input_dim*num_evals.
    @param x_pitch The pitch size in bytes of x
    @param num_evals The number of evaluation points
    @param dimension Equal to input_dim
    @param levels An array of size total_num_pts*dimension.
    @param levels_pitch The pitch size in bytes of levels.
    @param indexes An array of size total_num_pts*dimension.
    @param indexes_pitch The pitch size in bytes of indexes.
    @param total_num_pts The number of collocation points.
    @param endpoints An array of size dimension*2
    @param value_pitch The pitch size in bytes of value.
    @param value An array of size total_num_pts*num_evals.
*/
__global__ void cuda_compute_basis_vals
(float const * x, const size_t x_pitch,
 const unsigned int num_evals, const unsigned int dimension,
 unsigned int const* levels, const size_t levels_pitch,
 unsigned int const* indexes, const size_t indexes_pitch,
 const unsigned int total_num_pts, float const * endpoints,
 const size_t value_pitch, float * value);

///Kernel for computing the hierarchical basis at a new set
///of collocation points.
/** Computes the hierarchical basis functions associated with
    the points defined by
    [levels [0:num_old_points-1,:],
     indexes [0:num_old_points-1,:]]
    at the new collocation points defined by
    [levels [num_old_points:num_old_points+num_new_points-1,:],
     indexes [num_old_points:num_old_points+num_new_points-1,:]].
    /** @param num_new_points The number of new collocation
        points.
        @param num_old_points The number of old collocation

```

```

        points.
        @param dimension Equal to input_dim
        @param levels An array of size total_num_pts*dimension.
        @param levels_pitch The pitch size in bytes of levels.
        @param indexes An array of size total_num_pts*dimension.
        @param indexes_pitch The pitch size in bytes of indexes.
        @param total_num_pts The number of collocation points.
        @param endpoints An array of size dimension*2
        @param value_pitch The pitch size in bytes of value.
        @param value An array of size total_num_pts*num_evals.
*/
__global__ void cuda_compute_basis_vals
(
    const unsigned int num_new_pts,
    const unsigned int num_old_pts,
    const unsigned int dimension,
    unsigned int const* levels, const size_t levels_pitch,
    unsigned int const* indexes, const size_t indexes_pitch,
    float const * endpoints, const size_t value_pitch,
    float * value);

///  

//Device function that computes  $a^{i-j}(x)$ .
__device__ float cuda_1D_basis_evaluation
(
    const float x, const unsigned int level,
    const unsigned int index, const float left,
    const float right, const bool useDerivs);

#endif



---



/**
 * @file LocalRefinableDriver.cpp
 * @author Christopher Miller <cmiller@math.umd.edu>
 * @version 1.0
 * @section LICENSE
 * CudaKDE: Cuda code for the evaluation of kernel density
 * estimates
 * Copyright (c) 2011, Christopher Miller.
 * This software is distributed under the GNU Lesser General
 * Public License V3. For more information, see the README
 * file in the top CudaKDE directory.
 *
 * @brief Implementation for the LocalRefinableDriver class.
 * @section DESCRIPTION
 *
 */

```

```

#include "LocalRefinableDriver.hpp"

using namespace cudaColloc;

CudaApproximationGrid::
CudaApproximationGrid(const unsigned int input_dim_,
                      const unsigned int output_dim_,
                      const unsigned int max_num_points_,
                      float const * endpoints):
    input_dim(input_dim_),
    output_dim(output_dim_),
    max_num_points(max_num_points_),
    current_level(0),
    num_current_points(0)
{
    cudaMalloc( (void**) &d_endpoints ,
                2*input_dim*sizeof(float) );
    cudaMemcpy(d_endpoints , endpoints ,
                2*input_dim*sizeof(float) ,
                cudaMemcpyHostToDevice);

    cudaMallocPitch( (void**) &d_surpluses ,
                    &surplus_pitch ,
                    output_dim*sizeof(float*) ,
                    max_num_points );

    cudaMallocPitch( (void**) &d_levels , &levels_pitch ,
                    input_dim*sizeof(unsigned int*) ,
                    max_num_points );

    cudaMallocPitch( (void**) &d_indexes , &indexes_pitch ,
                    input_dim*sizeof(unsigned int*) ,
                    max_num_points );
}

CudaApproximationGrid::~
~CudaApproximationGrid()
{
    cudaFree(d_endpoints);
    cudaFree(d_surpluses);
    cudaFree(d_levels);
    cudaFree(d_indexes);
}

```

```

}

void CudaApproximationGrid::
addLevel(const unsigned int num_new_points,
         unsigned int const* new_levels,
         unsigned int const* new_indexes,
         float const* new_values)
{
    if ( num_current_points + num_new_points > max_num_points){
        throw std::exception();
    } else {
        cudaError_t my_error =
            cudaMemcpy2D(&(d_surpluses [ num_current_points*
                                         surplus_pitch /
                                         sizeof(float)]),
                       surplus_pitch,
                       new_values,
                       output_dim*sizeof(float),
                       output_dim*sizeof(float),
                       num_new_points,
                       cudaMemcpyHostToDevice);
        cudaMemcpy2D(&(d_levels [ num_current_points*
                                   levels_pitch /
                                   sizeof(unsigned int)]),
                    levels_pitch,
                    new_levels,
                    input_dim*sizeof(float),
                    input_dim*sizeof(float),
                    num_new_points,
                    cudaMemcpyHostToDevice);
        cudaMemcpy2D(&(d_indexes [ num_current_points*
                                    indexes_pitch /
                                    sizeof(unsigned int)]),
                    indexes_pitch,
                    new_indexes,
                    input_dim*sizeof(float),
                    input_dim*sizeof(float),
                    num_new_points,
                    cudaMemcpyHostToDevice);
        this->computeNewSurpluses(num_new_points);
        num_current_points += num_new_points;
        ++current_level;
    } //endif
}

```

```

unsigned int CudaApproximationGrid::getOutputDim()
{
    return output_dim;
}

void CudaApproximationGrid::
computeNewSurpluses(const unsigned int num_new_points)
{
    if (num_current_points != 0){
        //Allocate memory for the matrix containing basis
        //function evaluations.
        float * phi;
        size_t phi_pitch;
        cudaMallocPitch((void**) &phi,
                        &phi_pitch,
                        num_current_points*sizeof(float),
                        num_new_points);

        dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
        dim3 dimGrid(ceil(num_current_points/(float) BLOCK_SIZE),
                    ceil(num_new_points/(float) BLOCK_SIZE));
        size_t shared_mem_size =
            (sizeof(unsigned int)*(2*input_dim*BLOCK_SIZE) +
             sizeof(float)*input_dim*BLOCK_SIZE);

        //Call the CUDA kernel to evaluate the old basis
        //functions at the new points
        cuda_compute_basis_vals
            <<<dimGrid,dimBlock,shared_mem_size+10000>>>
            (num_new_points,num_current_points,input_dim,
             d_levels,levels_pitch,d_indexes,indexes_pitch,
             d_endpoints,phi_pitch,phi);

        //Call the CUDAbblas function to update the matrix of surpluses.
        cublasSgemm('n',
                    'n',
                    output_dim,
                    num_new_points,
                    num_current_points,
                    -1.0,
                    d_surpluses,
                    surplus_pitch/sizeof(float),
                    phi,
                    phi_pitch/sizeof(float),
                    1.0,

```

```

        &(d_surpluses[num_current_points*
                    surplus_pitch/sizeof(float)]),
        surplus_pitch/sizeof(float));
    cudaFree(phi);
} //endif
} //end computeNewSurpluses

void CudaApproximationGrid::
getInterpolantValue(float const * x,
                   const unsigned int num_evals,
                   float* y)
{
    //Allocate memory for the matrix containing basis function
    //evaluations.
    float * phi;
    size_t phi_pitch;
    cudaMallocPitch((void**) &phi,
                  &phi_pitch,
                  num_current_points*sizeof(float),
                  num_evals);

    //Transfer the targets from the host to the device
    float * d_x;
    size_t x_pitch;
    cudaMallocPitch((void**) &d_x,
                  &x_pitch,
                  input_dim*sizeof(float),
                  num_evals);
    cudaMemcpy2D(d_x,
                x_pitch,
                x,
                input_dim*sizeof(float),
                input_dim*sizeof(float),
                num_evals,
                cudaMemcpyHostToDevice);

    // Run the CUDA kernel to compute the basis values at the
    // requested locations.
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
    dim3 dimGrid(ceil(num_current_points/(float) BLOCK_SIZE),
                ceil(num_evals/(float) BLOCK_SIZE));
    size_t shared_mem_size = (sizeof(unsigned int)*
                            (2*input_dim*BLOCK_SIZE) +
                            sizeof(float)*
                            (input_dim*BLOCK_SIZE));

```

```

cuda_compute_basis_vals
    <<<dimGrid , dimBlock , shared_mem_size+10000>>>(d_x ,
                                                    x_pitch ,
                                                    num_evals ,
                                                    input_dim ,
                                                    d_levels ,
                                                    levels_pitch ,
                                                    d_indexes ,
                                                    indexes_pitch ,
                                                    num_current_points ,
                                                    d_endpoints ,
                                                    phi_pitch ,
                                                    phi);

float* d_value;
size_t value_pitch;
cudaMallocPitch(&d_value ,
               &value_pitch ,
               output_dim*sizeof(float) ,
               num_evals);

//Compute the interpolant values.
cublasSgemm('n' ,
            'n' ,
            output_dim ,
            num_evals ,
            num_current_points ,
            1.0 ,
            d_surpluses ,
            surplus_pitch/sizeof(float) ,
            phi ,
            phi_pitch/sizeof(float) ,
            0.0 ,
            d_value ,
            value_pitch/sizeof(float));

//Return the result to the user.
cudaMemcpy2D(y ,
             output_dim*sizeof(float) ,
             d_value ,
             value_pitch ,
             output_dim*sizeof(float) ,
             num_evals ,
             cudaMemcpyDeviceToHost);

```



```

    cudaFree(phi);
} //end getInterpolantValue

void CudaApproximationGrid::
normSurpluses(float* norms, const unsigned int start,
              const unsigned int stop)
{
    for (unsigned int idx = start; idx <= stop; ++idx) {
        norms[idx-start] =
            cublasSnorm2(output_dim,
                        &(d_surpluses[idx*surplus_pitch/
                                      sizeof(float)]), 1);
    }
}

__global__ void cuda_compute_basis_vals
(const unsigned int num_new_pts,
 const unsigned int num_old_pts, const unsigned int dimension,
 unsigned int const* levels, const size_t levels_pitch,
 unsigned int const* indexes, const size_t indexes_pitch,
 float const * endpoints, const size_t value_pitch,
 float * value)
{
    extern __shared__ unsigned int collocLvlShared[];
    const int myCollocBlock = blockIdx.x;
    const int myTargetBlock = blockIdx.y;
    const int myCollocPt = threadIdx.x;
    const int myTarget = threadIdx.y;

    const unsigned int global_colloc_idx =
        myCollocPt + myCollocBlock*BLOCK_SIZE;
    const unsigned int global_target_idx =
        myTarget + myTargetBlock*BLOCK_SIZE;

    const int num_collocPts_this_block =
        ( myCollocBlock ==
          ceilf(num_old_pts/(float) BLOCK_SIZE) - 1 )?
        (((num_old_pts % BLOCK_SIZE) == 0 ) ? BLOCK_SIZE :
         num_old_pts%BLOCK_SIZE) :
        BLOCK_SIZE;

    const int num_tgts_this_block =
        ( myTargetBlock ==
          ceilf(num_new_pts/(float) BLOCK_SIZE) - 1 )?

```

```

(((num_new_pts % BLOCK_SIZE) == 0 ) ? BLOCK_SIZE :
    num_new_pts%BLOCK_SIZE) :
    BLOCK_SIZE;

//Get the required points level, index, and targets into
//shared.
unsigned int * collocIndexShared =
    (unsigned int *) &collocLvlShared[dimension*
        num_collocPts_this_block];
float * targetsShared =
    (float *) &collocIndexShared[dimension*
        num_collocPts_this_block];
if ( myTarget == myCollocPt ) {
    if ( global_colloc_idx < num_old_pts ) {
        for ( int idx = 0; idx < dimension; ++idx ) {
            collocLvlShared[dimension*myCollocPt + idx] =
                levels[global_colloc_idx*levels_pitch/
                    sizeof(unsigned int) + idx];
            collocIndexShared[dimension*myCollocPt + idx] =
                indexes[global_colloc_idx*indexes_pitch/
                    sizeof(unsigned int) + idx];
        }
    }

    if ( global_target_idx < num_new_pts ) {
        const unsigned int this_tgt_index =
            num_old_pts + global_target_idx;
        for (int idx = 0; idx < dimension; ++idx ) {
            const unsigned int this_dim_level =
                levels[this_tgt_index*levels_pitch/
                    sizeof(unsigned int) + idx];
            const unsigned int this_dim_idx =
                indexes[this_tgt_index*indexes_pitch/
                    sizeof(unsigned int) + idx];
            if ( this_dim_level == 1) {
                targetsShared[myTarget*dimension + idx] =
                    .5*(endpoints[2*idx] + endpoints[2*idx+1]);
            } else if (this_dim_level == 2) {
                if (this_dim_idx == 0)
                    targetsShared[myTarget*dimension + idx] =
                        endpoints[2*idx];
                else targetsShared[myTarget*dimension+idx] =
                    endpoints[2*idx + 1];
            } else {
                const float offset =

```

```

        ((endpoints[2*idx] + endpoints[2*idx + 1])/2.0 -
         endpoints[2*idx])/
        exp2f( (float)this_dim_level - 2.0);
    targetsShared[myTarget*dimension+idx] =
        endpoints[2*idx] + offset + 2*offset*this_dim_idx;
    }
}
}
}

__syncthreads();

if ( global_colloc_idx < num_old_pts &&
     global_target_idx < num_new_pts ) {
    float local_result = 1;
    float const* target = &targetsShared[myTarget*dimension];
    unsigned int const* level =
        &collocLvlShared[myCollocPt*dimension];
    unsigned int const* index =
        &collocIndexShared[myCollocPt*dimension];
    for ( int idx = 0; idx < dimension; ++idx ) {
        local_result *=
            cuda_1D_basis_evaluation( target[idx],
                                     level[idx],
                                     index[idx],
                                     endpoints[2*idx],
                                     endpoints[2*idx + 1],
                                     false);
    }
    value[global_target_idx*value_pitch/sizeof(float) +
         global_colloc_idx] = local_result;
}
}

__global__ void
cuda_compute_basis_vals
(float const * x, const size_t x_pitch,
 const unsigned int num_evals, const unsigned int dimension,
 unsigned int const* levels, const size_t levels_pitch,
 unsigned int const* indexes, const size_t indexes_pitch,
 const unsigned int total_num_pts, float const * endpoints,
 const size_t value_pitch, float * value)
{
    const int myCollocBlock = blockIdx.x;

```

```

const int myTargetBlock = blockIdx.y;
const int myCollocPt = threadIdx.x;
const int myTarget = threadIdx.y;

const unsigned int global_colloc_idx =
    myCollocPt + myCollocBlock*BLOCK_SIZE;
const unsigned int global_target_idx =
    myTarget + myTargetBlock*BLOCK_SIZE;

const int num_collocPts_this_block =
    ( myCollocBlock ==
      ceilf(total_num_pts/(float) BLOCK_SIZE) - 1 )?
    (((total_num_pts % BLOCK_SIZE) == 0 ) ? BLOCK_SIZE :
      total_num_pts%BLOCK_SIZE) :
    BLOCK_SIZE;

const int num_tgts_this_block =
    ( myTargetBlock ==
      ceilf(num_evals/(float) BLOCK_SIZE) - 1 )?
    (((num_evals % BLOCK_SIZE) == 0 ) ? BLOCK_SIZE :
      num_evals%BLOCK_SIZE) :
    BLOCK_SIZE;

//Get the required points level, index, and targets into
//shared.
extern __shared__ unsigned int collocLvlShared[];
unsigned int * collocIndexShared =
    (unsigned int *) &collocLvlShared[dimension*
                                      num_collocPts_this_block];
float * targetsShared =
    (float *) &collocIndexShared[dimension*
                                   num_collocPts_this_block];
if ( myTarget == myCollocPt ) {
    if ( global_colloc_idx < total_num_pts ) {
        for ( int idx = 0; idx < dimension; ++idx ){
            collocLvlShared[dimension*myCollocPt + idx] =
                levels[global_colloc_idx*levels_pitch/
                      sizeof(unsigned int) + idx];
            collocIndexShared[dimension*myCollocPt + idx] =
                indexes[global_colloc_idx*indexes_pitch/
                      sizeof(unsigned int) + idx];
        }
    }
}

if ( global_target_idx < num_evals ) {

```

```

        for (int idx = 0; idx < dimension; ++idx ) {
            targetsShared[myTarget*dimension + idx] =
                x[global_target_idx*x_pitch/sizeof(float) + idx];
        }
    }
}

__syncthreads();

if ( global_colloc_idx < total_num_pts &&
    global_target_idx < num_evals ) {
    float local_result = 1;
    float const* target = &targetsShared[myTarget*dimension];
    unsigned int const* level =
        &collocLvlShared[myCollocPt*dimension];
    unsigned int const* index =
        &collocIndexShared[myCollocPt*dimension];
    for ( int idx = 0; idx < dimension; ++idx ) {
        local_result *=
            cuda_1D_basis_evaluation(target[idx],
                                    level[idx],
                                    index[idx],
                                    endpoints[2*idx],
                                    endpoints[2*idx + 1],
                                    false);
    }
    value[global_target_idx*value_pitch/sizeof(float) +
        global_colloc_idx] = local_result;
} //endif
}

__device__ float cuda_1D_basis_evaluation
(const float x,const unsigned int level ,
 const unsigned int index,const float left ,const float right ,
 const bool useDerivs)
{
    float midpoint = (left + right)/2.0;
    if( !useDerivs ) {
        if(level == 0) {
            float ihateyou = 1/fabsf(0);
        }
        else if( level == 1 ) {
            if( x >= left && x<= right ) return 1.0;
            else return 0.0;
        }
    }
}

```

```

else if( level == 2 ) {
    if( index == 0 ) {
        if( x >= left && x < midpoint )
            return (midpoint- x)/(midpoint - left);
        else return 0.0;
    } else {
        if( x> midpoint && x<= right) {
            return (x - midpoint)/(right-midpoint);
        } else return 0;
    }
}

else{
    const float offset = ( (left + right)/ 2.0 - left ) /
        exp2f( (float)level - 2.0);
    const float location = left + offset + 2*offset*index;
    const float left_support = location - offset;
    const float right_support = location + offset;
    if( x > left_support && x <= location ) {
        return (x - left_support)/(location - left_support);
    } else if ( x > location && x < right_support ) {
        return (right_support - x)/
            (right_support - location);
    } else return 0.0;
}
}
return 0.0;
}

```

Bibliography

- [1] David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition, 1999.
- [2] I. Babuška, F. Nobile, and R. Tempone. A stochastic collocation method for elliptic partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 45:1005–1034, 2007.
- [3] I. Babuška, R. Tempone, and G. E. Zouraris. Galerkin finite element approximations of stochastic elliptic partial differential equations. *SIAM Journal on Numerical Analysis*, 42:800–825, 2004.
- [4] J. Bäck, F. Nobile, L Tamellini, and R. Tempone. Stochastic galerkin and collocation methods for pdes with random coefficients: A numerical comparison. Technical Report 09-33, Institute for Computational Engineering and Sciences, University of Texas at Austin, 2009.
- [5] A. Barth, C. Schwab, and N. Zollinger. Multi-level monte carlo finite element method for elliptic pdes with stochastic coefficients. *Numerische Mathematik*, 119:123–161, 2011.
- [6] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, Cambridge, 3rd edition, 2007.
- [7] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta Numerica*, 7:1–49, 1998.
- [8] K. A. Cliffe, M. B. Giles, R. Scheichl, and A. L. Teckentrup. Multilevel monte carlo methods and applications to elliptic pdes with random coefficients. Accepted for publication in *Computing and Visualization in Science*, 2011.
- [9] M. Deb, I. Babuška, and J.T. Oden. Solution of stochastic parital differential equations using galerkin finite element techniques. *Computer Methods in Applied Mechanics and Engineering*, 190:6359–6372, 2001.
- [10] M.S. Eldred, A.A. Giunta, B.G. van Bloemen Waanders, S.F. Wojtkiewicz, W.E. Hart, and M.P. Alleva. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.0 users manual. Technical Report SAND2006-6337, Sandia National Laboratories, October 2006.
- [11] H. Elman, O. Ernst, D. O’Leary, and M. Stewart. Efficient iterative algorithms for the stochastic finite element method with application to acoustic scattering. *Computer Methods in Applied Mechanics and Engineering*, 194:1037–1055, 2005.

- [12] H. Elman and D. Furnival. Solving the stochastic steady-state diffusion problem using multigrid. *IMA Journal of Numerical Analysis*, 27:675–688, 2007.
- [13] H. Elman and D. Lee. Use of linear algebra kernels to build an efficient finite element solver. *Parallel Computing*, 21:161–173, 1995.
- [14] H. Elman and C. Powell. Block-diagonal precondition for spectral stochastic finite-element systems. *IMA Journal of Numerical Analysis*, 29:350–375, 2009.
- [15] H. Elman, D. Silvester, and A. Wathen. *Finite Elements and Fast Iterative Solvers*. Oxford University Press, Oxford, 2005.
- [16] H. C. Elman, C. W. Miller, E. T. Phipps, and R. S. Tuminaro. Assessment of collocation and galerkin approaches to linear diffusion equations with random data. *International Journal for Uncertainty Quantifications*, 1:19–33, 2011.
- [17] W. Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, Oxford, 2004.
- [18] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 smoothed aggregation user’s guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.
- [19] R. Ghanem and P. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer-Verlag, New York, 1991.
- [20] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 3rd edition, 1996.
- [21] A. Gordon and C. Powell. Solving stochastic collocation systems with algebraic multigrid. In G. Kreiss, P. Lötstedt, A. Målqvist, and M. Neytcheva, editors, *Numerical Mathematics and Advanced Applications*. Proceedings of ENU-MATH, 2009.
- [22] M. Grigoriu. Probabilistic models for stochastic partial differential equations. *Journal of Computational Physics*, 229:8406–8429, 2010.
- [23] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2000.
- [24] M. Heroux, R. Bartlett, V. Hoekstra, J. Hu, T Kolda, R Lehoucq, K Long, R Pawlowski, E Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
- [25] Steven G. Johnson. The nlopt nonlinear-optimization package. <http://ab-initio.mit.edu/nlopt>. Version 2.2.4.
- [26] A. Klimke. *Uncertainty Modeling Using Fuzzy Arithmetic and Sparse Grids*. PhD thesis, Universität Stuttgart, 2006.

- [27] A. Klimke and B. Wohlmuth. Algorithm 847: spinterp: piecewise multilinear hierarchical sparse grid interpolation in MATLAB. *ACM Transactions on Mathematical Software*, 31(4):561–579, 2005.
- [28] S. Larsson and V. Thomée. *Partial Differential Equations with Numerical Methods*. Springer-Verlag, Berlin, 2005.
- [29] M. Loeve. *Probability Theory*, volume 2. Springer-Verlag, New York, 4th edition, 1978.
- [30] X. Ma and N. Zabaras. An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. *Journal of Computational Physics*, 228:3084–3113, 2009.
- [31] X. Ma and N. Zabaras. High-dimensional stochastic model representation technique for the solution of stochastic pdes. *Journal of Computational Physics*, 229(10):3884–3915, 2010.
- [32] O. Le Maitre, O. Knio, B. Debusschere, H. Najm, and R. Ghanem. A multigrid solver for two-dimensional stochastic diffusion equations. *Computer Methods in Applied Mechanical Engineering*, 192:4723–4744, 2003.
- [33] N. Metropolis and S. Ulam. The Monte-Carlo method. *Journal of the American Statistical Association*, 44:335–341, 1949.
- [34] F. Nobile, R. Tempone, and C.G. Webster. An anisotropic sparse grid collocation algorithm for the solution of stochastic differential equations. *SIAM Journal on Numerical Analysis*, 46:2411–2442, 2008.
- [35] F. Nobile, R. Tempone, and C.G. Webster. A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 45:2309–2345, 2008.
- [36] NVIDIA Corporation. Whitepaper: NVIDIA’s Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. Version 1.1.
- [37] NVIDIA Corporation. NVIDIA CUDA C programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2010. Version 3.2.
- [38] NVIDIA Corporation. CUDA Toolkit 4.0: CUBLAS Library. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf, 2011. Version 0.1.
- [39] M. Pellisetti and R. Ghanem. Iterative solution of systems of linear equations arising in the context of stochastic finite elements. *Advances in Engineering Software*, 31:607–616, 2000.

- [40] M. Rao and R. Smith. *Probability Theory with Applications*. Springer-Verlag, New York, 2nd edition, 2006.
- [41] R. Sagar and V. Smith. On the calculation of Rys polynomials and quadratures. *International Journal of Quantum Chemistry*, 43:827–836, 1992.
- [42] A. Sarkar and R. Ghanem. Mid-frequency structural dynamics with parameter uncertainty. *Computer Methods in Applied Mechanical Engineering*, 191:5499–5513, 2002.
- [43] C. Schwab and R. Todor. Karhunen-loève approximation of random fields by generalized fast multipole methods. *Journal of Computational Physics In Uncertainty Quantification in Simulation Science*, 217:100–122, 2007.
- [44] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.
- [45] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995.
- [46] BV. Srinivasan and R. Duraiswami. GPUML: Graphical processors for speeding up kernel machines. In *Workshop on High Performance Analytics- Algorithms, Implementations, and Applications*. SIAM Conference on Data Mining, 2010.
- [47] X. Wan and G. E. Karniadakis. An adaptive multi-element generalized polynomial chaos method for stochastic differential equations. *Journal of Computational Physics*, 209:617–642, 2005.
- [48] X. Wan and G. E. karniadakis. Solving elliptic problems with non-Gaussian spatially-dependent random coefficients. *Computer Methods in Applied Mechanics and Engineering*, 198:1985–1995, 2009.
- [49] N. Weiner. The homogeneous chaos. *American Journal of Mathematics*, 60(4):897–936, 1938.
- [50] D. Xiu and J. Hesthaven. High-order collocation methods for differential equations with random inputs. *SIAM Journal on Scientific Computing*, 27:1118–1139, 2005.
- [51] D. Xiu and G. Karniadakis. Modeling uncertainty of elliptic parital differential equations via generalized polynomial chaos. *Proceedings of the 15th ASCE Engineering Mechanics Conference*, 2002.
- [52] D. Xiu and J. Shen. Efficient stochastic galerkin methods for random diffusion equations. *Journal of Computational Physics*, 228:266–281, 2009.
- [53] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast gauss transform and efficient kernel density estimation. *Computer Vision*, 1:664–671, 2003.