Interpolants, Error Bounds, and Mathematical Software for Modeling and Predicting Variability in Computer Systems

Thomas C. H. Lux

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Layne Watson, Chair
Yili Hong, Co-chair
Kirk Cameron
Bert Huang
Gang Wang
Young Cao

August 14, 2020
Blacksburg, Virginia

Keywords: Approximation Theory, Numerical Analysis, High Performance Computing,
Computer Security, Nonparametric Statistics, Mathematical Software

Interpolants, Error Bounds, and Mathematical Software for Modeling and Predicting Variability in Computer Systems

Thomas C. H. Lux

(ABSTRACT)

Function approximation is an important problem. This work presents applications of interpolants to modeling random variables. Specifically, this work studies the prediction of distributions of random variables applied to computer system throughput variability. Existing approximation methods including multivariate adaptive regression splines, support vector regressors, multilayer perceptrons, Shepard variants, and the Delaunay mesh are investigated in the context of computer variability modeling. New methods of approximation using Box splines, Voronoi cells, and Delaunay for interpolating distributions of data with moderately high dimension are presented and compared with existing approaches. Novel theoretical error bounds are constructed for piecewise linear interpolants over functions with a Lipschitz continuous gradient. Finally, a mathematical software that constructs monotone quintic spline interpolants for distribution approximation from data samples is proposed.

Interpolants, Error Bounds, and Mathematical Software for Modeling and Predicting Variability in Computer Systems

Thomas C. H. Lux

(GENERAL AUDIENCE ABSTRACT)

It is common for scientists to collect data on something they are studying. Often scientists want to create a (predictive) model of that phenomenon based on the data, but the choice of *how* to model the data is a difficult one to answer. This work proposes methods for modeling data that operate under very few assumptions that are broadly applicable across science. Finally, a software package is proposed that would allow scientists to better understand the *true* distribution of their data given relatively few observations.

Dedication

For Alex Faustie Grant, my dream teammate.

Acknowledgments

Finishing a Ph.D. is hard work. It's not always fun. And sometimes it's stressful. But the people I've met along the way, the things I've learned, and the people from all paths of life that have supported me throughout this journey made all the difference. My gratitudes are endless, but I'll do my best to enumerate some here.

First and foremost I want to thank my advisor Dr. Watson for his enduring push for deep scientific rigor and always reminding me of my most important job, to "think!" While reading that without context might fall short, Dr. Watson's incredible effort, earnest care, and impressive repertoire of knowledge afforded me many great lessons throughout my graduate tenure that I value dearly. His genuine feedback has helped me become a better researcher and he's someone I won't hesitate to consult in the future.

I want to thank my coadvisor Dr. Hong for deepening my knowledge of statistics and introducing me to countless relevant and interesting lines of research. His support and direction drove the statistical significance of my work. In addition I want to thank all of my committee members for their care and contributions, Dr. Cameron for asking important practical questions and encouraging a bigger picture cohesion throughout my work, Dr. Huang and Dr. Wang for ensuring theoretical groundedness and relevance to modern research as well as being great teachers, and Dr. Cao for motivating many connections between my work and other areas of applied computational science.

I was uniquely lucky in that the entirety of my work at Virginia Tech was supported by the VarSys project. Those team members, both students and faculty, guided my growth as a researcher and my research direction as a whole. I thank them all for the many conversations about fascinating and diverse topics in computer science, as well as constant feedback and collaboration that broadened my interests and drove me to produce higher quality results.

Where would I be without my closest friends through graduate school. I sincerely thank all of Grad Council, those who helped found the group and see it to its great success as a leading computer science graduate council in the nation. I am especially grateful for my lab mates Tyler Chang and Ayaan Kazerouni for the lunches around Blacksburg, encouraging conversations, and general support that made graduate life fun and memorable in ways that it would never have been otherwise.

Another thanks goes to my family for their continued love and support, particularly my parents Karen and John Lux, brothers Dylan and Austen, sisters Amy, and Arial, and grandmothers Gooma and Mor Mor. They have made all the difference and I am unbelievably grateful for them all.

Lastly I would like to thank my fiancé Alex Grant for her enduring love and support over the last four years. I couldn't imagine getting this far without our fun trips to Kentucky, summers together, and Skype dinner and movie dates that made the distance between us from Blacksburg to St. Louis bearable.

This work was supported in part by the National Science Foundation under Grant Numbers CNS-1565314 and CNS-1838271. Any opinions, findings, and conclusions or recommendations expressed in this material are my own and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

The Importance and Applications of Variability

Computational variability presents itself in a variety of forms. Processes that are apparently identical in a cloud computing or high performance computing (HPC) environment may take different amounts of time to complete the same job. This variability can cause unintentional violations of service level agreements in cloud computing applications or indicate suboptimal performance in HPC applications. The sources of variability, however, are distributed throughout the system stack and often difficult to identify. The methodology presented in this work is applicable to modeling the expected variability of useful computer system performance measures without any prior knowledge of system architecture. Some examples of interesting performance measures that could be modeled with the techniques in this work include computational throughput, power consumption, processor idle time, number of context switches, and RAM usage, as well as any other numeric measure of performance.

Predicting performance variability in a computer system is a challenging problem that has primarily been attempted in one of two ways: (1) build a statistical model of the performance data collected by running experiments on the system at select settings, or (2) run artificial experiments using a simplified simulation of the target system to estimate architecture and application bottlenecks. In this work, modeling techniques rest in the first category and represent a notable increase in the ability to model precise characteristics of variability.

Many previous works attempting to model system performance have used simulated environments [39, 79, 80]. Grobelny et al. [39] refer to statistical models as being oversimplified and not capable of capturing the true complexity of the underlying system. Historically, statistical models have been limited to modelling at most one or two system parameters and have therefore not been capable of modeling the complexity of the underlying system [4, 5, 75, 84]. These limited statistical models have provided satisfactory performance in narrow application settings. However these techniques require additional code annotations, hardware abstractions, or additional assumptions about the behavior of the application in order to generate models. In contrast, the approaches that are presented here require no modifications of applications, no architectural abstractions, nor any structural descriptions of the input data being modeled. The techniques used in this work only require performance data as input.

Most existing work on performance variability has focused on operating system (OS) induced variability [8, 23]. Yet, system I/O variability has been particularly difficult for statistical models

to capture [4]. The prior work attempting to model I/O variability has been similarly limited to one or two system parameters [53].

Chapters 3 and 5 present and evaluate applications of multivariate interpolation to the domain of computer system I/O throughput. Interpolants and regressors are used to predict the different measures of variability (e.g., mean, variance, cumulative distribution functions) for the expected I/O throughput on a system with previously unseen configurations. Beyond these I/O case studies, the techniques discussed can tractably model tens of interacting system parameters with tens of thousands of known configurations. A major contribution of this work is a modeling framework that uses multivariate interpolation to capture precise characteristics (via cumulative distribution functions) of arbitrary performance measure on any type of computer system.

1.1 Broader Applications of Approximation

Regression and interpolation are problems of considerable importance that find applications across many fields of science. Pollution and air quality analysis [26], energy consumption management [49], and student performance prediction [19, 54] are a few of many interdisciplinary applications of multivariate regression for predictive analysis. As discussed later, these techniques can also be applied to prediction problems related to forest fire risk assessment [18], Parkinson's patient clinical evaluations [77], local rainfall and weather [82], credit card transactions [22], and high performance computing (HPC) file input/output (I/O) [56].

Regression and interpolation have a considerable theoretical base in one dimension [15]. Splines in particular are well understood as an interpolation technique in one dimension [24], particularly B-splines. Tensor products of B-splines [38] or other basis functions have an unfortunate exponential scaling with increasing dimension. Exponential scaling prohibits tensor products from being reasonably applied beyond three-dimensional data. In order to address this dimensional scaling challenge, C. de Boor and others proposed box splines [25], of which one of the approximation techniques in Chapter 4 of this work is composed [57].

The theoretical foundation of low dimensional interpolation allows the construction of strong error bounds that are absent from high dimensional problems. Chapter 6 extends some known results regarding the secant method [27] to construct an interpolation error bound for problems of arbitrary dimension. These error bounds are useful, considering the same cannot be said for regression algorithms in general. The maximum complexity of an interpolant is bounded by the amount of data available, while the maximum complexity of a regressor is bounded by both the amount of data and the chosen parametric form. For this reason, generic uniform bounds are largely unobtainable for regression techniques on arbitrary approximation problems, even when the approximation domain is bounded. These generic bounds imply that interpolants may be suitable for a broader class of approximation problems than (heuristically chosen) regression techniques.

Aside from theoretical motivation for the use of interpolants, there are often computational advan-

tages as well. Interpolants do not have the need for *fitting* data, or minimizing error with respect to model parameters. In applications where the amount of data is large and the relative number of predictions that need to be made for a given collection of data is small, the direct application of an interpolant is much less computationally expensive.

In this work, multivariate interpolation is defined given a closed convex subset Y of a metrizable topological vector space with metric s, some function $f : \mathbb{R}^d \to Y$ and a set of points $X = \left\{x^{(1)}, \ldots, x^{(n)}\right\} \subset \mathbb{R}^d$, along with associated function values $f\left(x^{(i)}\right)$. The goal is to construct an approximation $\hat{f} : \mathbb{R}^d \to Y$ such that $\hat{f}\left(x^{(i)}\right) = f\left(x^{(i)}\right)$ for all $i = 1, \ldots, n$. It is often the case that the form of the true underlying function f is unknown, however it is still desirable to construct an approximation $\hat{f}$ with small approximation error at $y \notin X$. The two metric spaces that will be discussed in this work are the real numbers with metric $s(x,y) = |x-y|$, and the set of cumulative distribution functions (CDFs) with the Kolmogorov-Smirnov (KS) statistic [52] as metric.

Multivariate regression is often used when the underlying function is presumed to be stochastic, or stochastic error is introduced in the evaluation of f. Hence, multivariate regression relaxes the conditions of interpolation by choosing parameters P defining $\hat{f}(x;P)$ to minimize the error vector $\left(\left|\hat{f}\left(x^{(1)};P\right) - f\left(x^{(1)}\right)\right|, \ldots, \left|\hat{f}\left(x^{(n)};P\right) - f\left(x^{(n)}\right)\right|\right)$ in some norm. The difficult question in the case of regression is often what parametric form to adopt for any given application.

The most challenging problem when scaling in dimension is that the number of possible interactions between dimensions grows exponentially. Quantifying all possible interactions becomes intractable, and hence beyond three-dimensional data mostly linear models are used. That is not to say nonlinear models are absent, but nonlinearities are often either preconceived or model pairwise interactions between dimensions at most. Even globally nonlinear approximations such as neural networks are constructed from compositions of summed low-interaction functions [17].

Provided the theoretical and practical motivations for exploring interpolants, this work aims to study the empirical performance differences between a set of scalable (moderately) interpolation techniques and a set of common regression techniques. These techniques are applied to a collection of moderately high dimensional problems ($5 \leq d \leq 30$) and the empirical results are discussed.

1.2 An Initial Application of Approximation

Performance tuning is often an experimentally complex and time intensive chore necessary for configuring High Performance Computing (HPC) systems. The procedures for this tuning vary largely from system to system and are often subjectively guided by the system engineer(s). Once a desired level of performance is achieved, an HPC system may only be incrementally reconfigured as required by updates or specific jobs. In the case that a system has changing workloads or nonstationary performance objectives that range from maximizing computational throughput to minimizing power consumption and system variability, it becomes clear that a more effective and automated tool is needed for configuring systems. This scenario presents a challenging and

important application of multivariate approximation and interpolation techniques.

Among the statistical models presented in prior works for modeling computer systems, [4] specifically mention that it is difficult for simplified models to capture variability introduced by I/O. System variability in general has become a problem of increasing interest to the HPC and systems communities, however most of the work has focused on operating system (OS) induced variability [8, 23]. The work that has focused on managing I/O variability does not use any sophisticated modeling techniques [53]. Hence, Chapter 3 presents a case study applying advanced mathematical and statistical modeling techniques to the domain of HPC I/O characteristics. The models are used to predict the mean throughput of a system and the variance in throughput of a system. The discussion section that follows outlines how the techniques presented can be applied to any performance metric and any system.

1.3 Organization of this Document

This thesis is comprised of work from multiple conference and journal papers. Chapters 1 and 2 share the broader context for this work and describe relevant existing algorithms for approximation, these chapters derive from multiple published papers [55, 58, 59]. Chapter 3 presents a first attempt at modeling variability and appears in [55]. Chapter 4 appears in [57] and presents multiple novel approximation techniques. Chapter 5 applies the novel approximation techniques to distribution prediction and appears in [56]. Chapter 6 presents a novel error bound and is to appear in [58]. Chapter 7 proposes a mathematical software package for monotone quintic spline interpolation, was partially presented in [60], and is to be submitted in full to ACM Transactions on Mathematical Software (TOMS). Finally Chapter 8 concludes, presenting the major takeaways of this work as well as promising future research directions.

Chapter 2

Algorithms for Constructing Approximations

2.1 Multivariate Regression

Multivariate regressors are capable of accurately modeling a complex dependence of a response (in Y) on multiple variables (represented as a points in $\mathbb{R}^d$). The approximations to some (unknown) underlying function $f : \mathbb{R}^d \to Y$ are chosen to minimize some error measure related to data samples $f\left(x^{(i)}\right)$. For example, least squares regression uses the sum of squared differences between modeled function values and true function values as an error measure. In this section and the next, some techniques are limited to approximating real valued functions ($Y \subset \mathbb{R}$). These techniques can be extended to real vector-valued ranges by repeating the construction for each component of the vector output. Throughout the following, x denotes a d-tuple, x_i the ith component of x, and $x^{(i)}$ the ith d-tuple data point. Different symbols are used to represent the approximation function $\hat{f}$.

2.1.1 Multivariate Adaptive Regression Splines

This approximation was introduced in [31] and subsequently improved to its current version in [32], called fast multivariate adaptive regression splines (Fast MARS). In Fast MARS, a least squares fit model is iteratively built by beginning with a single constant valued function and adding two new basis functions at each iteration of the form

$$B_{2j-1}(x) = B_l(x)\left(x_i - x_i^{(p)}\right)_+,$$
$$B_{2j}(x) = B_k(x)\left(x_i - x_i^{(p)}\right)_-,$$

where $j \leq m$ is the iteration number, m is the maximum number of underlying basis functions, $1 \leq p \leq n$, and $B_l(x)$, $B_k(x)$ are basis functions from the previous iteration,

$$w_+ = \begin{cases} w, & w \geq 0 \\ 0, & w < 0 \end{cases},$$

and $w_- = (-w)_+$. After iteratively constructing a model, MARS then iteratively removes basis functions that do not contribute to goodness of fit. In effect, MARS creates a locally component-wise tensor product approximation of the data. The overall computational complexity of Fast

MARS is $\mathcal{O}(ndm^3)$. This work uses an implementation of Fast MARS [72] with $m = 200$ throughout all experiments.

2.1.2 Support Vector Regressor

Support vector machines are a common method used in machine learning classification tasks that can be adapted for the purpose of regression [7]. How to build a support vector regressor (SVR) is beyond the scope of this summary, but the resulting functional fit $p : \mathbb{R}^d \to \mathbb{R}$ has the form

$$p(x) = \sum_{i=1}^{n} a_i K\left(x, x^{(i)}\right) + b,$$

where K is the selected kernel function, $a_i \in \mathbb{R}^n$, $b \in \mathbb{R}$ are coefficients to be solved for simultaneously. The computational complexity of the SVR is $\mathcal{O}(n^2dm)$, with m being determined by the minimization convergence criterion. This work uses the scikit-learn SVR [66] with a radial basis function kernel.

2.1.3 Multilayer Perceptron Regressor

The neural network is a well studied and widely used method for both regression and classification tasks [45, 73]. When using the rectified linear unit (ReLU) activation function [21] and fitting the model with stochastic gradient descent (SGD) or BFGS minimization techniques [35, 61, 71], the model built by a multilayer perceptron uses layers $l : \mathbb{R}^i \to \mathbb{R}^j$ defined by

$$l(u) = \left(u^t W_l + c_l\right)_+,$$

where $c_l \in \mathbb{R}^j$ and W_l is the $i \times j$ weight matrix for layer l. In this form, the multilayer perceptron (MLP) produces a piecewise linear model of the input data. The computational complexity of training a multilayer perceptron is $\mathcal{O}(ndm)$, where m is determined by the sizes of the layers of the network and the stopping criterion of the minimization used for finding weights. This work uses an MLP built from Keras and Tensorflow to perform regression [1, 16] with ten hidden layers each having thirty two nodes (a total of approximately ten thousand parameters), ReLU activation, and one hundred thousand steps of SGD for training. It should be noted that the choice of neural network architecture affects approximation performance, but no architecture tuning is performed here.

2.2 Multivariate Interpolation

The following interpolation techniques demonstrate a reasonable variety of approaches to interpolation. All of the presented interpolants produce approximations that are continuous in value, which is often a desirable property in applied approximation problems.

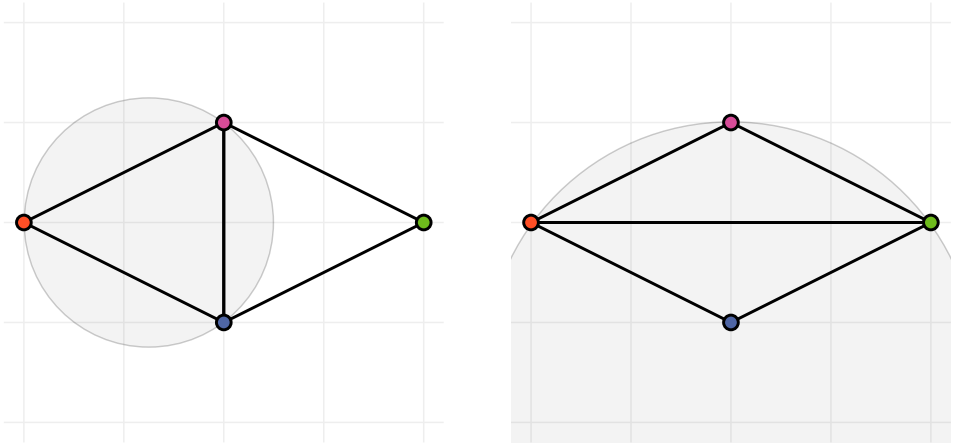

Figure 2.1: On the left above is a depiction of a Delaunay triangulation over four points, notice that the circumball (shaded circle) for the left simplex does not contain the fourth point. On the right above, a non-Delaunay mesh is depicted. Notice that the circumball for the top simplex (shaded circle, clipped at bottom edge of the visual) contains the fourth point which violates the Delaunay condition for a simplex.

2.2.1 Delaunay

The Delaunay triangulation is a well-studied geometric technique for producing an interpolant [50]. The Delaunay triangulation of a set of points into simplices is such that there are no points inside the sphere defined by the vertices of each simplex. For a d-simplex S with vertices $x^{(0)}$, $x^{(1)}$, ..., $x^{(d)}$, and function values $f\left(x^{(i)}\right)$, $i = 0, \ldots, d$, $y \in S$ is a unique convex combination of the vertices,

$$y = \sum_{i=0}^{d} w_i x^{(i)}, \quad \sum_{i=0}^{d} w_i = 1, \quad w_i \geq 0, \quad i = 0, \ldots, d,$$

and the Delaunay interpolant $\hat{f}(y)$ at y is given by

$$\hat{f}(y) = \sum_{i=0}^{d} w_i f\left(x^{(i)}\right).$$

The computational complexity of Delaunay interpolation (for the implementation used [14]) is $\mathcal{O}(knd^3)$. Given pathological data the entire triangulation could be computed with $k = n^{\lceil d/2 \rceil}$, but the average case yields $k = d \log d$ and is capable of scaling reasonably to $d \leq 50$. In the present application, a Delaunay simplex S containing y is found, then the $d+1$ vertices (points in X) of S are used to assign weights to each vertex and produce the predicted function value for point y.

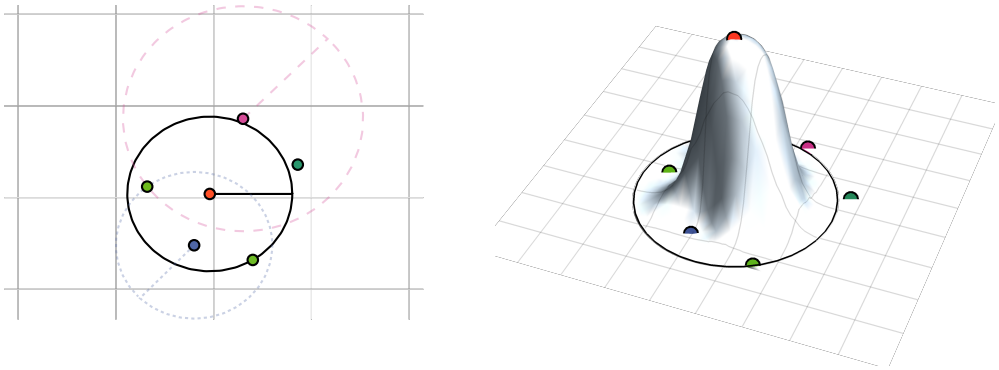

Figure 2.2: On the left above is a depiction of the radius of influence for three chosen points of a collection in two dimensions using the modified Shepard criteria. On the right a third axis shows the relative weight for the center most interpolation point $x^{(i)}$ with the solid line representing its radius of influence, where $W_i(x)$ is 0 for $\|x-x^{(i)}\|_2 \geq r_i$ and $W_i(x)/\sum_{k=1}^{n} W_k(x) \to 1$ as $x \to x^{(i)}$.

2.2.2 Modified Shepard

The modified Shepard method used here (ShepMod) generates a continuous approximation based on Euclidean distance and resembles a nearest neighbor interpolant [20]. This model is a type of *metric interpolation*, also called a Shepard method [36, 74]. The interpolant has the form

$$p(x) = \frac{\sum_{k=1}^{n} W_k(x) f\left(x^{(k)}\right)}{\sum_{k=1}^{n} W_k(x)},$$

where $p\left(x^{(k)}\right) = f\left(x^{(k)}\right)$, $W_k(x) = \left(\left(r_k - \left\|x-x^{(k)}\right\|_2\right)_+ / \left(r_k \left\|x-x^{(k)}\right\|_2\right)\right)^2$ for $x \neq x^{(k)}$, and $r_k \in \mathbb{R}$ is the smallest radius such that at least $d+1$ other points $x^{(j)}$, $j \neq k$, are inside the closed Euclidean ball of radius r_k about $x^{(k)}$. The interpolant $p(x)$ is continuous because the singularities at the $x^{(k)}$ are removable. The computational complexity of ShepMod is $\mathcal{O}(n^2 d)$. This work uses a Fortran 95 implementation of ShepMod derived from SHEPPACK [76].

2.2.3 Linear Shepard

The linear Shepard method (LSHEP) is a blending function using local linear interpolants, a special case of the general Shepard algorithm [76]. The interpolant has the form

$$p(x) = \frac{\sum_{k=1}^{n} W_k(x) P_k(x)}{\sum_{k=1}^{n} W_k(x)},$$

where $W_k(x)$ is the same as for the modified Shepard method and $P_k(x)$ is a local linear approximation to the data satisfying $P_k\left(x^{(k)}\right) = f\left(x^{(x)}\right)$. The computational complexity of LSHEP is $\mathcal{O}(n^2d^3)$. This work uses the Fortran 95 implementation of LSHEP in SHEPPACK [76].

Chapter 3

Naïve Approximations of Variability

This chapter compares five of the multivariate approximation techniques that operate on inputs in $\mathbb{R}^d$ (d-tuples of real numbers) and produce predicted responses in $\mathbb{R}$. Three of the chosen techniques are regression based and the remaining two are interpolants, providing reasonable coverage of the varied mathematical strategies that can be employed to solve continuous modeling problems.

3.1 I/O Data

In order to evaluate the viability of multivariate models for predicting system performance, this work presents a case study of a four-dimensional dataset produced by executing the IOzone benchmark from Norcott [63] on a homogeneous cluster of computers. Multiple I/O Zone data sets will be used throughout the work, however this chapter relies on this specific four-dimensional dataset. All experiments were performed on parallel shared-memory nodes common to HPC systems. Each system had a lone guest Linux operating system (Ubuntu 14.04 LTS//XEN 4.0) on a dedicated 2TB HDD on a 2 socket, 4 core (2 hyperthreads per core) Intel Xeon E5-2623 v3 (Haswell) platform with 32 GB DDR4. The system performance data was collected by executing IOzone 40 times for each of a select set of system configurations. A single IOzone execution reports the max I/O throughput seen for the selected test in kilobytes per second. The 40 executions for each system configuration are converted into the mean and variance, both values in $\mathbb{R}$ capable of being modeled individually by the multivariate approximation techniques presented in Chapter 2. The summary of data used in the experiments for this chapter can be seen in Table 3.1. Distributions of raw throughput values being modeled can be seen in Figure 3.1.

3.1.1 Dimensional Analysis

This analysis utilizes an extension to standard k-fold cross validation that allows for a more thorough investigation of the expected model performance in a variety of real-world situations. Alongside randomized splits, two extra components are considered: the amount of training data provided, and the dimension of the input data. It is important to consider that algorithms that perform well with less training input also require less experimentation. Although, the amount of training data required may change as a function of the dimension of the input and this needs to be studied as well. The framework used here will be referred to as a multidimensional analysis (MDA) of the

System Parameter	Values
File Size	64, 256, 1024
Record Size	32, 128, 512
Thread Count	1, 2, 4, 8, 16, 32, 64, 128, 256
Frequency	$\{12, 14, 15, 16, 18, 19, 20, 21, 23, 24, 25, 27, 28, 29, 30, 30.01\} \times 10^5$
Response Values	
Throughput Mean	$[2.6 \times 10^5, 5.9 \times 10^8]$
Throughput Variance	$[5.9 \times 10^{10}, 4.7 \times 10^{16}]$

Table 3.1: A description of the system parameters being considered in the IOzone tests. Record size must not be greater than file size and hence there are only six valid combinations of the two. In total there are $6 \times 9 \times 16 = 864$ unique system configurations.

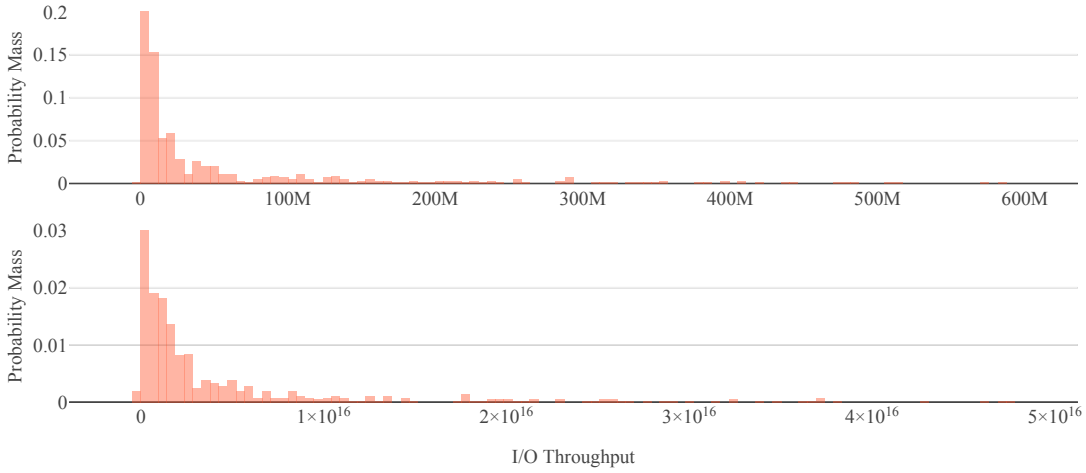

Figure 3.1: Histograms of 100-bin reductions of the PMF of I/O throughput mean (top) and I/O throughput variance (bottom). In the mean plot, the first 1% bin (truncated in plot) has a probability mass of .45. In the variance plot, the second 1% bin has a probability mass of .58. It can be seen that the distributions of throughputs are primarily of lower magnitude with occasional extreme outliers.

IOzone data.

Multidimensional Analysis

This procedure combines random selection of training and testing splits with changes in the input dimension and the ratio of training size to testing size. Given an input data matrix with n rows (points) and d columns (components), MDA proceeds as follows:

1. For all $k = 1, \ldots, d$ and for all nonempty subsets $F \subset \{1, 2, \ldots, d\}$, reduce the input data to points $(z, f_F(z))$ with $z \in \mathbb{R}^k$ and $f_F(z) = E\left[\left\{f\left(x^{(i)}\right) \mid \left(x_F^{(i)} = z\right)\right\}\right]$, where $E[\cdot]$ denotes the mean and $x_F^{(i)}$ is the subvector of $x^{(i)}$ indexed by F.
2. For all r in $\{5, 10, \ldots, 95\}$, generate N random splits $(train, test)$ of the reduced data with r percentage for training and $100 - r$ percentage for testing.
3. When generating each of N random $(train, test)$ splits, ensure that all points from $test$ are in the convex hull of points in $train$ (to prevent extrapolation); also ensure that the points in $train$ are well spaced.

In order to ensure that the testing points are in the convex hull of the training points, the convex hull vertices of each set of (reduced dimension) points are forcibly placed in the training set. In order to ensure that training points are well spaced, a statistical method for picking points from Amos et al. [3] is used:

1. Generate a sequence of all pairs of points $\left(z^{(i_1)}, z^{(j_1)}\right), \left(z^{(i_2)}, z^{(j_2)}\right), \ldots$ sorted by ascending pairwise Euclidean distance between points, so that $\left|\left|z^{(i_k)} - z^{(j_k)}\right|\right|_2 \leq \left|\left|z^{(i_{k+1})} - z^{(j_{k+1})}\right|\right|_2$.
2. Sequentially remove points from candidacy until only $|train|$ remain by randomly selecting one point from the pair $\left(z^{(i_m)}, z^{(j_m)}\right)$ for $m = 1, \ldots$ if both $z^{(i_m)}$ and $z^{(j_m)}$ are still candidates for removal.

Given the large number of constraints, level of reduction, and use of randomness in the MDA procedure, occasionally N unique training/testing splits may not be created or may not exist. In these cases, if there are fewer than N possible splits, then deterministically generated splits are used. Otherwise after $3N$ attempts, only the unique splits are kept for analysis. The MDA procedure has been implemented in Python#3 while most regression and interpolation methods are Fortran wrapped with Python. All randomness has been seeded for repeatability.

For any index subset F (of size k) and selected value of r, MDA will generate up to N multivariate models $f_F(z)$ and predictions $\hat{f}_F\left(z^{(i)}\right)$ for a point $z^{(i)} \in \mathbb{R}^k$. There may be fewer than N predictions made for any given point. Extreme points of the convex hull for the selected index subset will always be used for training, never for testing. Points that do not have any close neighbors will

often be used for training in order to ensure well-spacing. Finally, as mentioned before, some index subsets do not readily generate N unique training and testing splits. The summary results presented in this work use the median of the (N or fewer) values $\hat{f}_F(z)$ at each point as the model estimate for error analysis.

3.2 Naïve Variability Modeling Results

A naïve multivariate prediction technique such as nearest neighbor could experience relative errors in the range $[0, \left(\max_x f(x) - \min_x f(x)\right) / \min_x f(x)]$ when modeling a nonnegative function $f(x)$ from data. The IOzone mean data response values span three orders of magnitude (as can be seen in Table 3.1) while variance data response values span six orders of magnitude. It is expected therefore, that all studied multivariate models perform better than a naïve approach, achieving relative errors strictly less than 10^3 for throughput mean and 10^6 for throughput variance. Ideally, models will yield relative errors significantly smaller than 1. The time required to compute thousands of models involved in processing the IOzone data through MDA was approximately five hours on a CentOS workstation with an Intel i7-3770 CPU at 3.40GHz. In four dimensions for example, each of the models could be constructed and evaluated over hundreds of points in less than a few seconds.

3.2.1 I/O Throughput Mean

Almost all multivariate models analyzed make predictions with a relative error less than 1 for most system configurations when predicting the mean I/O throughput of a system given varying amounts of training data. The overall best of the multivariate models, Delaunay, consistently makes predictions with relative error less than .05 (5% error). In Figure 3.3 it can also be seen that the Delaunay model consistently makes good predictions even with as low as 5% training data (43 of the 864 system configurations) regardless of the dimension of the data.

3.2.2 I/O Throughput Variance

The prediction results for variance resemble those for predicting mean. Delaunay remains the best overall predictor (aggregated across training percentages and dimensions) with median relative error of .47 and LSHEP closely competes with Delaunay having a median signed relative error of -.92. Outliers in prediction error are much larger for all models. Delaunay produces relative errors as large as 78 and other models achieve relative errors around 10^3. The relative errors for many models scaled proportional to the increased orders of magnitude spanned by the variance response compared with mean response. As can be seen in Figure 3.4, all models are more sensitive to the amount of training data provided than their counterparts for predicting mean.

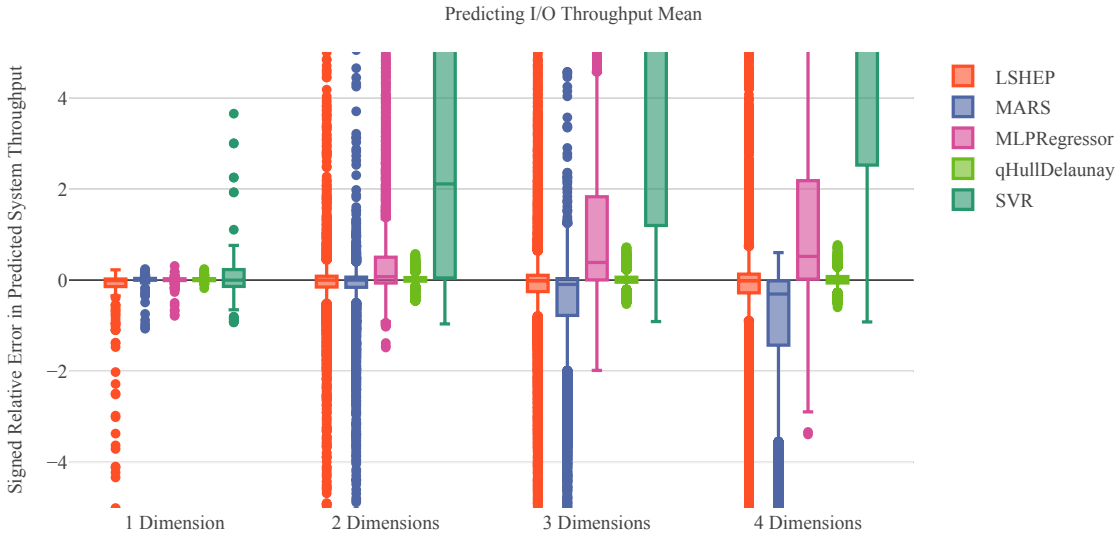

Figure 3.2: These box plots show the prediction error of mean with increasing dimension. The top box whisker for SVR is 40, 80, 90 for dimensions 2, 3, and 4, respectively. Notice that each model consistently experiences greater magnitude error with increasing dimension. Results for all training percentages are aggregated.

3.2.3 Increasing Dimension and Decreasing Training Data

As can be seen in Figure 3.2, all of the models suffer increasing error rates in higher dimension. This is expected, because the number of possible interactions to model grows exponentially. However, LSHEP and Delaunay maintain the slowest increase in relative error. The increase in error seen for Delaunay suggests that it is capable of making predictions with a range of relative errors that grows approximately linearly with increasing dimension input. This trend suggests that Delaunay would remain a viable technique for accurately modeling systems with 10's of parameters given only small amounts of training data. All models, with the exception of MARS, produce smaller errors given more training data. Increasing the amount of training data most notably reduces the number of prediction error outliers.

3.3 Discussion of Naïve Approximations

The results presented above demonstrate that a straightforward application of multivariate modeling techniques can be used to effectively predict HPC system performance. Some modeling effort on the part of a systems engineer combined with a significant amount of experimentation (days of CPU time for the IOzone data used here) can yield a model capable of accurately tuning an HPC system to the desired performance specification, although qualitatively correct predictions can be

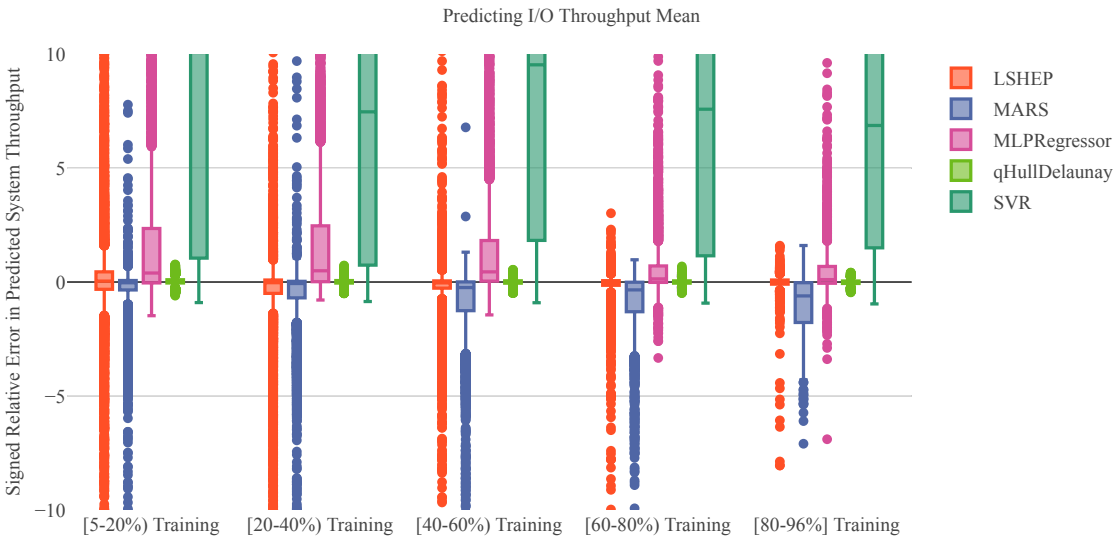

Figure 3.3: These box plots show the prediction error of mean with increasing amounts of training data provided to the models. Notice that MARS is the only model whose primary spread of performance increases with more training data. Recall that the response values being predicted span three orders of magnitude and hence relative errors should certainly remain within that range. For SVR the top box whisker goes from around 100 to 50 from left to right and is truncated in order to maintain focus on better models. Results for all dimensions are aggregated. Max training percentage is 96% due to rounding training set size.

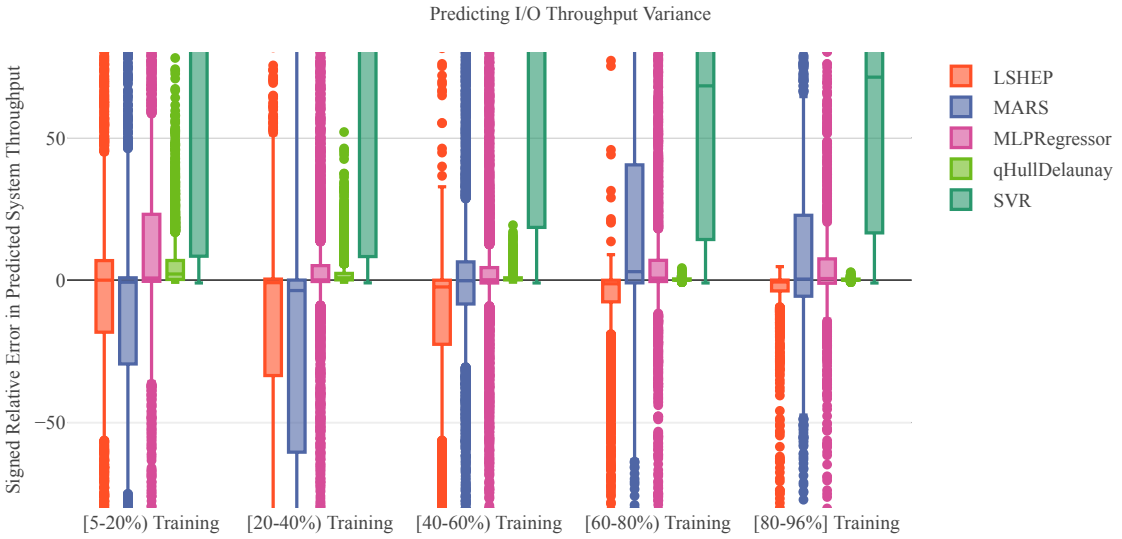

Figure 3.4: These box plots show the prediction error of variance with increasing amounts of training data provided to the models. The response values being predicted span six orders of magnitude. For SVR the top box whisker goes from around 6000 to 400 (decreasing by factors of 2) from left to right and is truncated in order to maintain focus on better models. Results for all dimensions are aggregated. Max training percentage is 96% due to rounding training set size.

achieved with much less (10%, say) effort.

3.3.1 Modeling the System

The modeling techniques generated estimates of drastically different quality when predicting I/O throughput mean and variance. A few observations: SVR has the largest range of errors for all selections of dimension and amounts of training data; MARS and LSHEP produce similar magnitude errors while the former consistently underestimates and the latter consistently overestimates; Delaunay has considerably fewer outliers than all other methods. SVR likely produces the poorest quality predictions because the underlying parametric representation is global and oversimplified (a single polynomial), making it unable to capture the complex local behaviors of system I/O. It is still unclear, however, what causes the behaviors of LSHEP, MARS, and Delaunay. An exploration of this topic is left to future work.

The Delaunay method appears to be the best predictor in the present IOzone case study. Particularly a piecewise linear interpolant like Delaunay appears well-suited for prediction when relatively small amounts of data are available to mode la function. It is important to note that the Delaunay computational complexity in the dimension of the input is worse than other techniques.

Finally, the ability of the models to predict variance was significantly worse than for the I/O mean.

The larger scale in variance responses alone do not account for the increase in relative errors witnessed. This suggests that system variability has a greater underlying functional complexity than the system mean and that latent factors are reducing prediction performance.

3.3.2 Extending the Analysis

System I/O throughput mean and variance are simple and useful system characteristics to model. The process presented in this chapter is equally applicable to predicting other useful performance characteristics of HPC systems such as: computational throughput, power consumption, processor idle time, context switches, RAM usage, or any other ordinal performance metric. For each of these there is the potential to model system variability as well. This chapter uses variance as a measure of variability, but the techniques are applied to more precise measures of variability (the entire distribution itself) in Chapter 5.

3.4 Takeaway From Naïve Approximation

Multivariate models of HPC system performance can effectively predict I/O throughput mean and variance. These multivariate techniques significantly expand the scope and portability of statistical models for predicting computer system performance over previous work. In the IOzone case study presented, the Delaunay method produces the best overall results making predictions for 821 system configurations with less than 5% error when trained on only 43 configurations. Analysis also suggests that the error in the Delaunay method will remain acceptable as the number of system parameters being modeled increases. These multivariate techniques should be applied to HPC systems with more than four tunable parameters in order to identify optimal system configurations that may not be discoverable via previous methods nor by manual performance tuning, which will be explored in later chapters.

Chapter 4

Box Splines: Uses, Constructions, and Applications

Chapter 3 demonstrated that the modeling and approximation techniques described in Chapter 2 can be used to model and predict numeric approximations of system variability. The case study was only for four-dimensional data, and the key weakness of the best predictor (Delaunay) is scaling with dimension. This Chapter discusses a modeling approach that may achieve better scaling with dimension, proposes a quasi-mesh construction, and analyzes the performance of this proposed technique.

4.1 Box Splines

A box spline in $\mathbb{R}^d$ is defined by its *direction vector set* A, composed of s d-vectors where $s \geq d$. Further, A will be written as a $d \times s$ matrix. The first m column vectors of A are denoted by A_m, $m \leq s$. A_d is required to be nonsingular. Consider the unit cube in s dimensions $Q_s = [0,1)^s$. $A_s(Q_s)$ is now the image (in d dimensions) of Q_s under the linear map A. This image is the region of support for the box spline defined by A_s in d dimensions. The box spline function in d dimensions for A_d is defined as

$$B(x \mid A_d) = \begin{cases} (\det(A_d))^{-1}, & x \in A_d(Q_d), \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

For A_s when $s > d$ the box spline is computed as

$$B(x \mid A_s) = \int_0^1 B(x - t v_s \mid A_{s-1}) dt, \tag{4.2}$$

where v_s is the sth direction vector of A.

The application of box splines presented in this chapter always utilizes the d-dimensional identity matrix as A_d. This simplifies the computation in Equation 4.1 to be the characteristic function for the unit cube. Composing A strictly out of k repetitions of the identity matrix forms the kth order B-spline with knots located at 0, 1, …, $k-1$, k along each axis (see Figure 4.1). Furthermore, while the number of subregions for the kth order d-dimensional box spline grows as k^d (see Figure

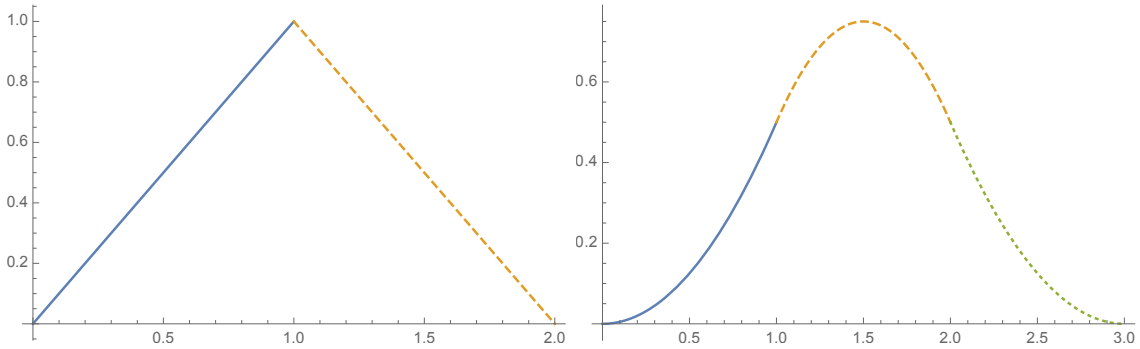

Figure 4.1: 1D linear (order 2) and quadratic (order 3) box splines with direction vector sets $\begin{pmatrix} 1 & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$ respectively. Notice that these direction vector sets form the B-Spline analogues, order 2 composed of two linear components and order 3 composed of 3 quadratic components (colored and styled in plot).

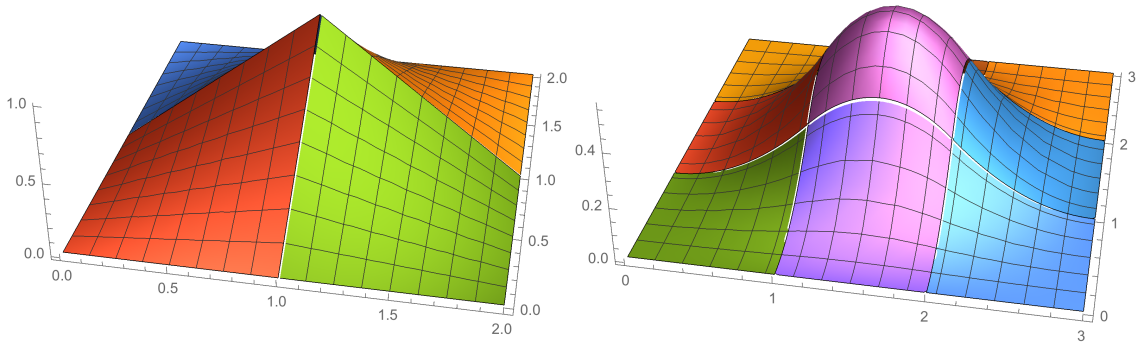

Figure 4.2: 2D linear (order 2) and quadratic (order 3) box splines with direction vector sets $\begin{pmatrix} I & I \end{pmatrix}$ and $\begin{pmatrix} I & I & I \end{pmatrix}$ respectively, where I is the identity matrix in two dimensions. Notice that these direction vector sets also produce boxes with order^2 subregions (colored in plot).

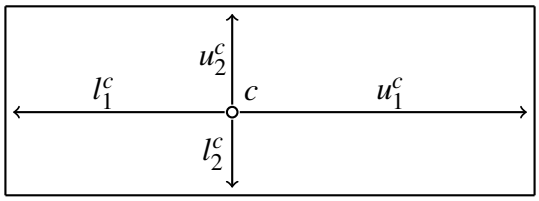

Figure 4.3: An example box in two dimensions with anchor c, upper widths u_1^c, u_2^c, and lower widths l_1^c, l_2^c. Notice that c is not required to be equidistant from opposing sides of the box, that is $u_i^c \neq l_i^c$ is allowed.

4.2), the symmetry provided by direction vector sets composed of repeated identity matrices allows the computation of box splines to be simplified. The value of a box spline at any location is then the product of all axis-aligned 1-dimensional kth order box splines.

The box splines as presented are viable basis functions. Each box spline can be shifted and scaled without modifying the underlying computation (similar to wavelets), yet the underlying computation is simple and scales linearly with dimension. For a more thorough introduction and exploration of box splines in their more general form, readers are referred to [25].

Throughout this section, the notation will be reused from Chapter 2. $X \subset \mathbb{R}^d$ is a finite set of points with known response values $f(x)$ for all $x \in X$. Also let $L, U \in \mathbb{R}^d$ define a bounding box for X such that $L < x < U$ for all $x \in X$.

Define a box $b^c = (c, l^c, u^c)$ in d dimensions with anchor $c \in \mathbb{R}^d$, lower width vector $l^c \in \mathbb{R}^d_+$, and upper width vector $u^c \in \mathbb{R}^d_+$ (where u_i^c refers to the ith component of u^c). A visual example of a box in two dimensions can be seen in Figure 4.3. Now, define a componentwise rescaling function $g : \mathbb{R}^d \to \mathbb{R}^d$ at point $x \in \mathbb{R}^d$ to be

$$\left(g^c(x)\right)_r = \frac{k}{2}\left(1 - \frac{(x_r - c_r)_-}{l_r^c} + \frac{(x_r - c_r)_+}{u_r^c}\right), \tag{4.3}$$

where $y_+ = \max\{y, 0\}$, $y_- = (-y)_+$, k is the order of the box spline as described in Section 4.1. Finally, each box spline in a box mesh can be evaluated as $B^c(x) = B(g^c(x) \mid A)$ presuming the order of approximation implies A. Both box meshes described in the following sections use box spline basis functions of this form.

A notable property of boxes defined with the linear rescaling function g^c, is that C^0 and C^1 continuity of the underlying box spline are maintained. C^0 continuity is maintained through scaling. C^1 continuity is maintained for all box splines with C^1 continuity (order ≥ 3) because the scaling discontinuity is located at c, where all box splines (of the presented form) have first derivative zero. All continuity beyond the first derivative is lost through the rescaling function g^c.

4.2 Max Box Mesh

The first of the three meshes produces a set of boxes around chosen control points and each box has maximal distance between the control point and the nearest side of the box. This centrality property is one mechanism for creating the largest reasonable regions of support for the underlying basis functions. The individual boxes are constructed via the following procedure given a set of control points $C \subseteq X$, $c^{(i)} \in C$,

1. Initialize a box $b^{c^{(1)}} = \left(c^{(1)}, (c^{(1)} - L), (U - c^{(1)})\right)$.
2. Identify $c^{(i)}$ over $\left\{ j \mid j \neq 1, B^{c^{(1)}}\left(c^{(j)}\right) \neq 0 \right\}$ that minimizes $\left\| c^{(j)} - c^{(1)} \right\|_\infty$.
3. Change the the box $b^{c^{(1)}}$ along the first dimension r such that $\left|\left| c^{(1)} - c^{(i)} \right|\right|_\infty = \left| c^{(1)} - c^{(i)} \right|_r$, to exclude $c^{(i)}$ from the support of $B^{c^{(1)}}$.
4. Repeat steps 2 and 3 until no point in C is in the support of $B^{c^{(1)}}$ (at most $2d$ times, once for each boundary of a box).

The same process is used to construct boxes around all control points in C. In order to improve the generality of the approximation, a set of control points is initially chosen to be well-spaced using a statistical method from Amos et al. [3]:

1. Generate a sequence of all pairs of points sorted by ascending pairwise Euclidean distance between points $\left(x^{(i_1)}, x^{(j_1)}\right)$, $\left(x^{(i_2)}, x^{(j_2)}\right)$, ... , so that $\left\| x^{(i_k)} - x^{(j_k)} \right\|_2 \leq \left\| x^{(i_{k+1})} - x^{(j_{k+1})} \right\|_2$.
2. Sequentially remove points from candidacy until only $|C|$ remain by randomly selecting a single point from each pair $\left(x^{(i_m)}, x^{(j_m)}\right)$ for $m = 1, \ldots$ if both $x^{(i_m)}$ and $x^{(j_m)}$ are still candidates for removal.

Once the boxes for a max box mesh have been constructed, the parameters can be identified via a least squares fit. The max box mesh (denoted MBM) is used to generate a $|X| \times |C|$ matrix M of box spline basis function evaluations at all points in X. The solution to the least squares problem $\min_P \left\| M\ P - f(X) \right\|_2$ is the parameterization of MBM. When $C = X$, M is the $|X| \times |X|$ identity matrix, making the max box mesh approximation $\hat{f}$ an interpolant.

While setting the number of boxes equal to the number of points causes the max box mesh to be an interpolant, the generality of the max box mesh approximation can often be improved by bootstrapping the selection of control points. Given a user-selected batch size $s \leq |X|$, start with s well-spaced control points. Next, measure the approximation error at $x \notin C$ and if the error is too large (determined by user), pick s points at which the magnitude of approximation error is largest, add those points to C, and recompute the max box mesh. The user is left to decide the batch size s and the error tolerance based on validation performance and computability. This work uses a batch size of one.

Definition 1 The hyperplane $x_r = c_r + u_r^c$ is the upper boundary of box b^c along dimension r, and similarly $x_r = c_r - l_r^c$ is the lower boundary of box b^c. When the anchor point y, for some box b^y, lies in the hyperplane (and facet of box b^c) defining either boundary along dimension r of b^c it is said that b^y *bounds* b^c in dimension r and is denoted $\left(b^c \,\middle|_r\, b^y\right)$.

Throughout all experiments and all repeated trials conducted for this study, all tested interpolation points were covered by at least one box in the *MBM*. However, it is possible for the *MBM* to not form a covering of $[L,U]$ when there are cyclic boundaries. Consider the following example in three dimensions:

$$\begin{aligned} C &= \{(0,0,0),(1,0,2/3),(1,1,4/3)\}, \\ b^{c^{(1)}} &= \big((0,0,0),\ (*,*,*),\ (1,*,4/3)\big), \\ b^{c^{(2)}} &= \big((1,0,2/3),\ (1,*,*),\ (*,1,*)\big), \\ b^{c^{(3)}} &= \big((1,1,4/3),\ (*,1,4/3),\ (*,*,*)\big). \end{aligned}$$

Asterisks are used to represent boxes that are not bounded by other boxes along some dimensions. The point $(2,2,-3)$ is not in any of the max boxes defined above. In this case, there is a cycle in box boundaries that looks like $\left(b^{c^{(1)}} \,\middle|_1\, b^{c^{(2)}} \,\middle|_2\, b^{c^{(3)}} \,\middle|_3\, b^{c^{(1)}}\right)$. This example demonstrates that it is geometrically possible for the max box mesh to fail to cover a space, however experiments demonstrate that it is empirically unlikely.

The max box mesh remains a viable strategy for computed approximations. Given a maximum of c control points in d dimensions with n points, the computational complexities are: $\mathcal{O}(c^2 d)$ for computing boxes, $\mathcal{O}(cd^2 + d^3)$ for a least squares fit, and $\mathcal{O}(n/s)$ for bootstrapping (which is multiplicative over the fitting complexities). Evaluating the max box mesh requires $\mathcal{O}(cd)$ computations.

4.3 Iterative Box Mesh

The iterative box mesh (*IBM*) comprises box-shaped regions that each contain exactly one control point in their interior just as in the *MBM*. However, the mesh is a covering for $[L,U]$ by construction and places boxes in a way that reduces apparent error. The boxes are constructed via the following procedure given a finite set of points $X \subset \mathbb{R}^d$, where $C \subseteq X$ is the (initially empty) set of control points.

1. Add the box that covers $[L,U]$ anchored at the most central point $x^{(k)} \in X$, add $x^{(k)}$ to C, and least squares fit the *IBM* model to all $x \in X$.
2. Add a new box $[L,U]$ anchored at $x^{(i)} \notin C$ such that $\left|IBM(x^{(i)}) - f(x^{(i)})\right| = \max_{x \in X \setminus C} \left|IBM(x) - f(x)\right|$, reshaping all boxes $b^{x^{(j)}}$ that contain $x^{(i)}$ by bounding the first dimension r such that

$\left|x_r^{(j)} - x_r^{(i)}\right| = \left\|x^{(j)} - x^{(i)}\right\|_\infty$ (also reshaping the box $b^{x^{(i)}}$ symmetrically), add $x^{(i)}$ to C, and then least squares fit the *IBM* model to all $x \in X$.

3. Repeat Step 2 until model approximation error is below tolerance t.

Just as for the *MBM*, the parameters can be identified via a least squares fit. The iterative box mesh is used to generate a $|X| \times |C|$ matrix M of box spline function evaluations at all points in X. Now the box spline coefficients are the solution to the least squares problem $\min_P \left\|M\,P - f(X)\right\|_2$. Also as for the *MBM*, $C = X$ causes M to equal the $|X| \times |X|$ identity, making the iterative box mesh approximation $\hat{f}$ an interpolant.

As opposed to the max box mesh, the bootstrapping procedure is built into the iterative box mesh. The user is left to decide the most appropriate error tolerance, however a decision mechanism and analysis is presented in Section 4.5.4. As mentioned earlier, the iterative box mesh is a covering for $[L,U]$ by construction and this can be proved by an inductive argument.

An *IBM* least squares fit $\hat{f}(z) = \sum_j P_j B^{c^{(i_j)}}(z)$ can generate approximations at new points $z \in Z \subset \mathbb{R}^d$ by evaluating $\hat{f}(z)$. The computational complexity for generating the mesh is $\mathcal{O}(c^2 nd)$ where c is the number of control points determined by the minimum error threshold and $n = |X|$. The computational complexity of evaluating the mesh at a single point is $\mathcal{O}(cd)$.

4.4 Voronoi Mesh

The final of the three meshes utilizes 2-norm distances to define boundaries rather than max norm distances. It also does not rely on box splines as the underlying basis function. A well-studied technique for classification and approximation is the nearest neighbor algorithm [20]. Nearest neighbor inherently utilizes the convex region $v^{x^{(i)}}$ (Voronoi cell [28]) consisting of all points closer to $x^{(i)}$ than any other point $x^{(j)}$. The Voronoi mesh smooths the nearest neighbor approximation by utilizing the Voronoi cells to define support via a generic basis function $V : \mathbb{R}^d \to \mathbb{R}_+$ given by

$$V^{x^{(i)}}(y) = \left(1 - \frac{\left\|y - x^{(i)}\right\|_2}{2\,d(y \mid x^{(i)})}\right)_+,$$

where $x^{(i)}$ is the center of the Voronoi cell, $y \in \mathbb{R}^d$ is an interpolation point, and $d(y \mid x^{(i)})$ is the distance between $x^{(i)}$ and the boundary of the Voronoi cell $v^{x^{(i)}}$ in the direction $y - x^{(i)}$. $V^{x^{(i)}}\left(x^{(j)}\right) = \delta_{ij}$ and $V^{x^{(i)}}$ has local support. While $V^{x^{(i)}}(x^{(i)}) = 1$, the 2 in the denominator causes all basis functions to go linearly to 0 at the boundary of the twice-expanded Voronoi cell. Note that this basis function is C^0 because the boundaries of the Voronoi cell are C^0. In the case that there is no boundary along the vector w, the basis function value is always 1.

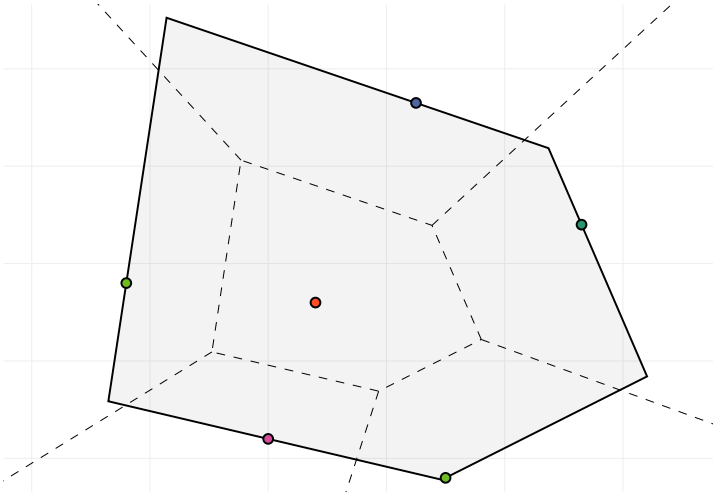

Figure 4.4: Above is a depiction of the Voronoi cell boundaries (dashed lines) about a set of interpolation points (dots) in two dimensions. In this example, the Voronoi mesh basis function about the center most point has nonzero weight in the shaded region and transitions from a value of one at the point to zero at the boundary of the twice expanded Voronoi cell (solid line).

While the cost of computing the exact Voronoi cells for any given set of points grows exponentially [29], the calculation of d is linear with respect to the number of control points and dimensions. Given any center $x^{(i)} \in \mathbb{R}^d$, set of control points $C \subseteq X$, and interpolation point $y \in \mathbb{R}^d$, $d\left(y \mid x^{(i)}\right)$ is the solution to

$$\max_{c \in C \setminus \{x^{(i)}\}} \frac{\left\|y - x^{(i)}\right\|_2}{2} \frac{y \cdot \left(c - x^{(i)}\right) - x^{(i)} \cdot \left(c - x^{(i)}\right)}{c \cdot \left(c - x^{(i)}\right) - x^{(i)} \cdot \left(c - x^{(i)}\right)}. \tag{4.4}$$

The parameters of the VM can now be computed exactly as for the MBM and IBM. The Voronoi mesh is used to generate a $|X| \times |C|$ matrix M of basis function evaluations at all points in X. Now the VM coefficients are the solution to the least squares problem $\min_P \left\|M\ P - f(X)\right\|_2$. When $X = C$, M is the identity making the mesh an interpolant. Bootstrapping can be performed with an identical procedure to that for the IBM.

1. Pick the most central point $x^{(k)} \in X$ to be the first control point in C and fit the VM model to all $x \in X$.
2. Identify a control point $x^{(i)} \notin C$ such that $\left|VM(x^{(i)}) - f(x^{(i)})\right| = \max_{x \in X \setminus C} \left|VM(x) - f(x)\right|$, add $x^{(i)}$ to C, and then fit the VM model to all $x \in X$.
3. Repeat Step 2 until approximation error is below tolerance t.

Any VM is naïvely a covering for $[L, U]$, since any possible interpolation point will have a nearest neighbor control point. The computational complexity of evaluating a parameterized Voronoi mesh

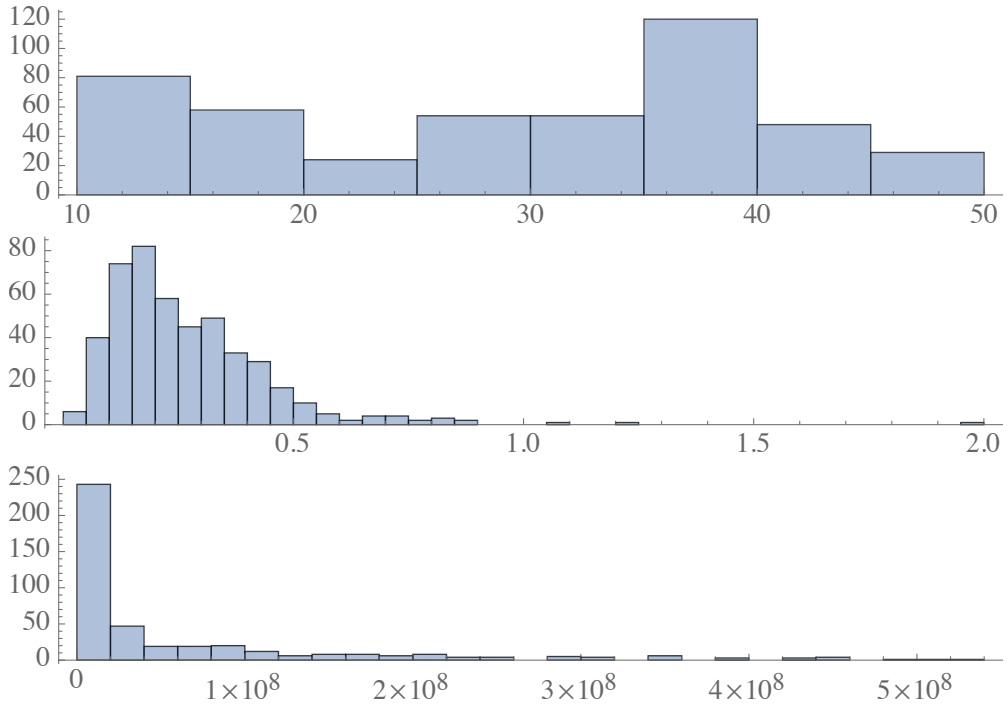

Figure 4.5: Histograms of Parkinsons (total UPDRS), forest fire (area), and HPC I/O (mean throughput) response values respectively. Notice that both the forest fire and HPC I/O data sets are heavily skewed.

with c control points is $\mathcal{O}(c^2 d)$. Bootstrapping the generation of a Voronoi mesh requires $\mathcal{O}(c^2 nd)$ computations for a maximum number of basis functions c determined by the error threshold.

4.5 Data and Analysis

Some data sets will be used to evaluate the interpolation and regression meshes proposed above. This chapter utilizes three data sets of varying dimension and application. In the following subsections the sources and targets of each data set are described, as well as challenges and limitations related to interpolating and approximating these data sets. The distributions of response values being modeled can be seen in Figure 4.5. The preprocessing and approximation processes are described in Section 4.5.4.

4.5.1 High Performance Computing I/O ($n = 532, d = 4$)

The first of three data sets is a four-dimensional data set produced by executing the IOzone benchmark from [63] on a homogeneous cluster of computers. The system performance data was collected by executing IOzone 40 times for each of a select set of system configurations. A single

IOzone execution reports the max I/O file-read throughput seen. The 40 executions for each system configuration are converted to their mean, which is capable of being modeled by each of the multivariate approximation techniques presented earlier in this chapter. The four dimensions being modeled to predict throughput mean are file size, record size, thread count, and CPU frequency.

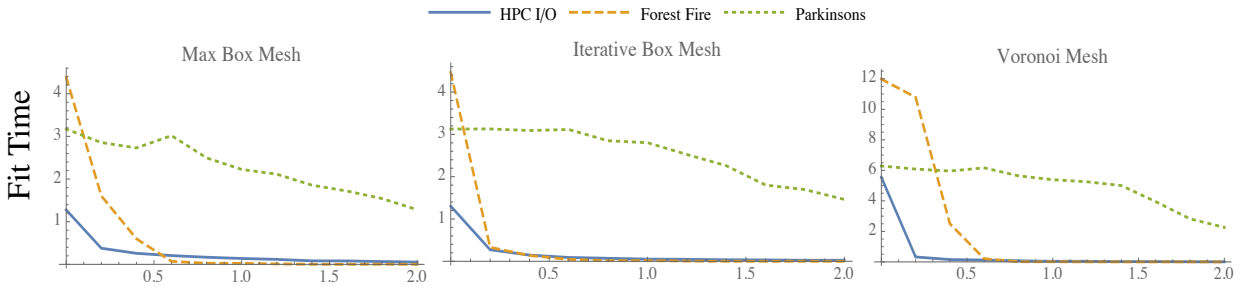

Figure 4.6: Time required to generate model fits for each technique with varying relative error tolerance during bootstrapping.

4.5.2 Forest Fire ($n = 517, d = 12$)

The forest fire data set [18] describes the area of Montesinho park burned on specific days and months of the year in terms of the environmental conditions. The twelve dimensions being used to model burn area are the x and y spatial coordinates of burns in the park, month and day of year, the FFMC, DMC, DC, and ISI indices (see source for details), the temperature in Celsius, relative humidity, wind speed, and outdoor rain. The original analysis of this data set demonstrated it to be difficult to model, likely due to the skew in response values.

4.5.3 Parkinson's Telemonitoring ($n = 468, d = 16$)

The final data set for evaluation [77] is derived from a speech monitoring study with the intent to automatically estimate Parkinson's disease symptom development in Parkinson's patients. The function to be predicted is a time-consuming clinical evaluation measure referred to as the UPDRS score. The total UPDRS score given by a clinical evaluation is estimated through 16 real numbers generated from biomedical voice measures of in-home sound recordings.

4.5.4 Performance Analysis

The performance of the approximation techniques varies considerably across the three evaluation data sets. Relative errors for the most naïve approximators such as nearest neighbor can range

Data Set	Technique	Tolerance	Average Error
HPC I/O	MBM	1.2	0.597
Forest Fire	MBM	1.8	3.517
Parkinson's	MBM	0.6	0.114
HPC I/O	IBM	0.4	0.419
Forest Fire	IBM	1.8	3.615
Parkinson's	IBM	1.8	0.121
HPC I/O	VM	0.2	0.382
Forest Fire	VM	1.0	4.783
Parkinson's	VM	2.0	1.824

Table 4.1: The optimal error tolerance bootstrapping parameters for each technique and each data set as well as the average absolute relative errors achieved by that tolerance. Notice that large relative error tolerances occasionally yield even lower evaluation errors, demonstrating the benefits of approximation over interpolation for noisy data sets.

from zero to $\left(\max_x f(x) - \min_x f(x)\right) / \min_x f(x)$ when modeling a positive function $f(x)$ from data. Each of the approximation techniques presented remain within these bounds and all errors are presented in signed relative form $(\hat{f}(x) - f(x))/f(x)$. Before the models are constructed all data values (components $x_r^{(i)}$ of $x^{(i)} \in X$) are shifted and scaled to be in the unit cube $[0,1]^d$, while the response values are taken in their original form. All models are evaluated with 10 random $80/20$ splits of the data.

Each of the approximation techniques presented incorporates bootstrapping based on an allowable error tolerance t. An analysis of the effects of bootstrapping error tolerances on validation accuracy can be seen in Figure 4.7. The approximation meshes perform best on the forest fire and Parkinson's data sets when the error tolerance used for fitting is large (smoothing rather than interpolating), while near-interpolation generally produces the most accurate models for HPC I/O. Another performance result of note is that the *MBM* and *IBM* have very similar basis functions with largely different outputs.

The selection of bootstrapping error tolerance also effects the computation time required to fit each of the models to data. Figure 4.6 presents the time required to construct approximations for each model and each data set with varying t. The rapid reduction in computation time required for the forest fire and HPC I/O data sets suggests that large reductions in error can be achieved with relatively few basis functions. The Parkinson's data set however presents a more noisy response, with increasing number of basis functions reducing error much less quickly.

The distributions of errors experienced by each approximation technique when the optimal bootstrapping relative error tolerance is selected can be seen in Figure 4.8. HPC I/O exhibits the most normal approximation errors, which suggests that the models are converging on the random noise of the response for the data set. The worst relative approximation errors are produced by the Voronoi mesh on the forest fire data set. The small magnitude true response values contribute to the larger relative errors. Regardless, the *VM* errors are unacceptably large.

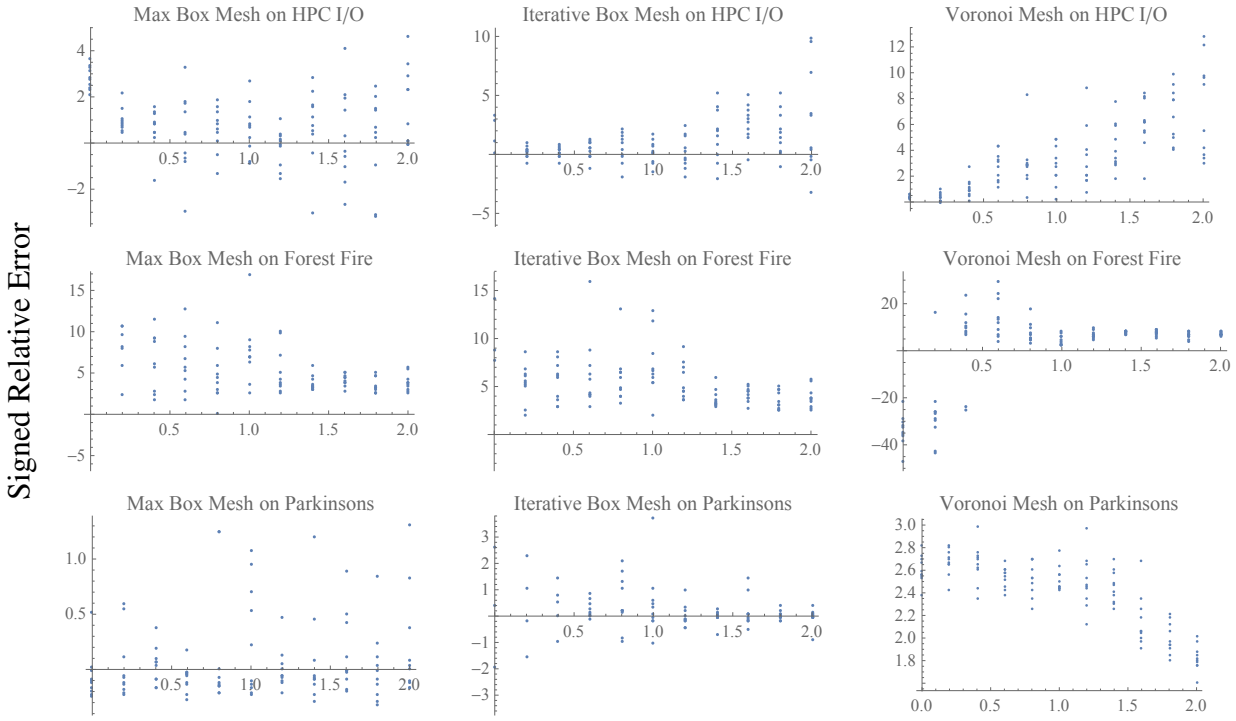

Figure 4.7: The performance of all three techniques with varied relative error tolerance for the bootstrapping parameter. The columns are for Max Box Mesh, Iterative Box Mesh, and Voronoi Mesh, respectively. The rows are for HPC I/O, Forest Fire, and Parkinson's respectively. Notice the techniques' behavior on the Parkinson's and Forest Fire data sets, performance increases with larger error tolerance.

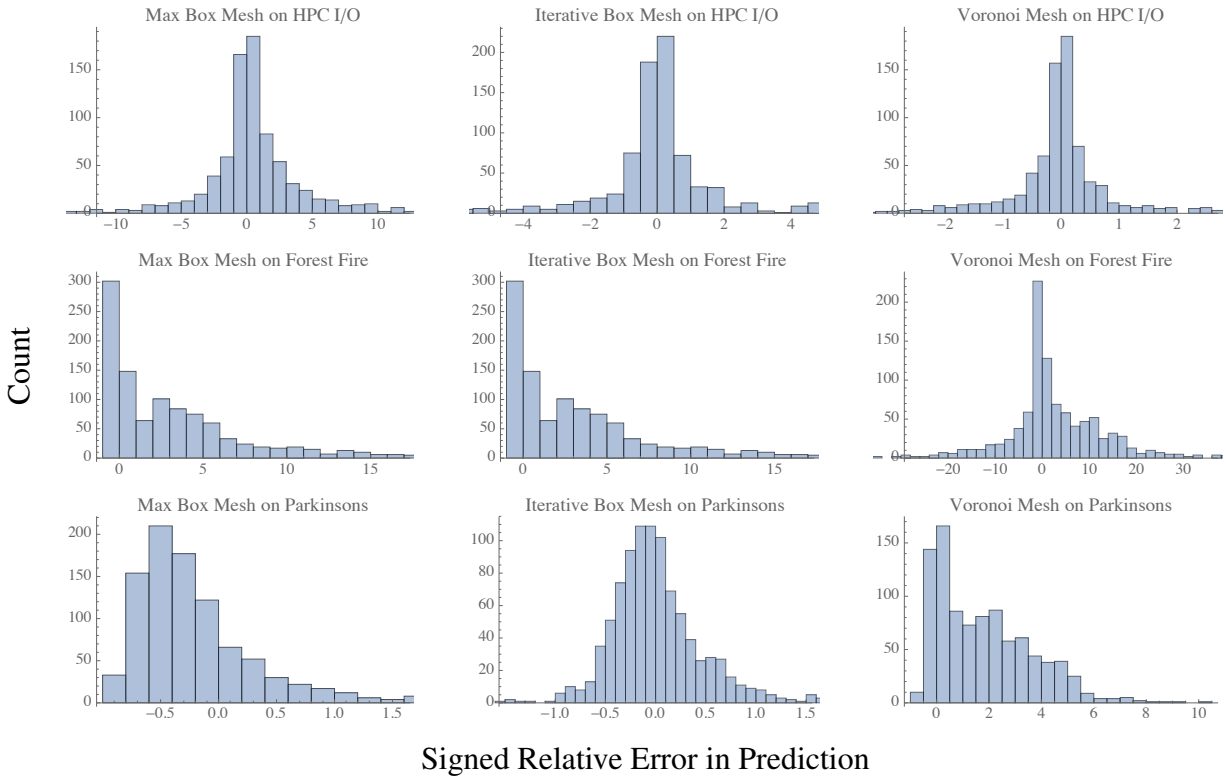

Figure 4.8: A sample of relative errors for all three techniques with optimal selections of error tolerance. The columns are for Max Box Mesh, Iterative Box Mesh, and Voronoi Mesh, respectively. The rows are for HPC I/O, Forest Fire, and Parkinson's respectively.

4.6 Discussion of Mesh Approximations

The bootstrapping procedure presented for each quasi-mesh approximation is very computationally expensive and does not provide much improvement over the interpolatory approach. Analysis suggests that the appropriate relative error tolerance needs to be discovered empirically for each application of a modeling technique. Further analytic studies could arrive at methods for determining optimal error tolerances at runtime, however increases in runtime complexity may not be afforded in many applications.

The box-shaped basis functions and the construction algorithms used for the *MBM* and *IBM* could become a source of error when d (the dimension of the data X) is comparable to n (the number of known points). The blending regions in which multiple basis functions overlap are always axis aligned and in applications such as image analysis, any single dimension may be unsuitable for approximating the true underlying function. The Voronoi mesh attempts to address this problem by utilizing boundaries between points in multiple dimensions simultaneously. However, it is empirically unclear whether the true benefits of the *VM* are seen in applications where $d \ll n$.

Each of the case studies presented have fewer than 1000 points. The complexity of the presented approximation techniques are suitable for large dimension, but the increased complexity associated with brute-force bootstrapping prohibits their use on larger data sets. The Voronoi mesh in particular has a large complexity with respect to n which could be significantly improved by dropping bootstrapping, as is done in Chapter 5. While each technique requires less than ten seconds on average to produce a fit in the presented case studies, the fit time required quickly grows into minutes around 1000 points. These results demonstrate the limits of expensive bootstrapping. However, the viability of each mesh encourages the further applications explored in later chapters.

4.7 Implications of Quasi-Mesh Results

The Max Box Mesh, Iterative Box Mesh, and Voronoi Mesh each provide novel strategies for effectively approximating multivariate phenomonon. The underlying constructions are theoretically straightforward. The computational complexities of each make them particularly suitable for applications in many dimensions, while the bootstrapping error tolerance parameter allows a balance between smoothing and interpolation to be explored empirically with each application. However, the expense of bootstrapping prohibits its use on larger data sets.

Chapter 5

Stronger Approximations of Variability

Performance and its variability can be summarized by a variety of statistics. Mean, range, standard deviation, variance, and interquartile range are a few summary statistics that describe performance and variability. However, the most precise characterization of any performance measure is the cumulative distribution function (CDF), or its derivative the probability density function (PDF). Previous techniques for predicting system performance have strictly modeled real-valued summary statistics because there exists a large base of mathematical techniques capable of approximating functions of the form $f : \mathbb{R}^d \to \mathbb{R}$. However, there is little systems work approximating functions $f : \mathbb{R}^d \to \{g \mid g : \mathbb{R} \to \mathbb{R}\}$.

5.1 Measuring Error

When the range of an approximation is the real numbers, error is reported with summary statistics including: min absolute error, max absolute error, and absolute error quartiles. When the range of an approximation is the space of cumulative distribution functions, the Kolmogorov-Smirnov statistic (max-norm difference between the functions) is used.

A hurdle when modeling function-valued outputs such as cumulative distribution functions (CDFs) or probability density functions (PDFs) is that certain properties must be maintained. It is necessary that a PDF $f : \mathbb{R} \to \mathbb{R}$ have the properties $f(x) \geq 0$ and $\int_{-\infty}^{\infty} f(x)dx = 1$. Instead, for a CDF $F : \mathbb{R} \to \mathbb{R}$ the properties are $F(x) \in [0,1]$ and $F(x)$ is absolutely continuous and nondecreasing. This work utilizes the fact that a convex combination of CDFs (or PDFs) results in a valid CDF (or PDF). Given $G(x) = \sum_i w_i F_i(x)$, $\sum_i w_i = 1$, $w_i \geq 0$, and each F_i is a valid CDF, G must also be a valid CDF. A demonstration of how this is applied can be seen in Figure 5.1. In this example the KS test null hypothesis is rejected at p-value 0.01, however it is not rejected at p-value 0.001.

The performance of approximation techniques that predict probability functions can be analyzed through a variety of summary statistics. This work uses the max absolute difference, also known as the Kolmogorov-Smirnov (KS) statistic [52] for its compatibility with the KS test.

The two-sample KS test is a useful nonparametric test for comparing two empirical CDFs while only assuming stationarity, finite mean, and finite variance. The null hypothesis (that two empirical

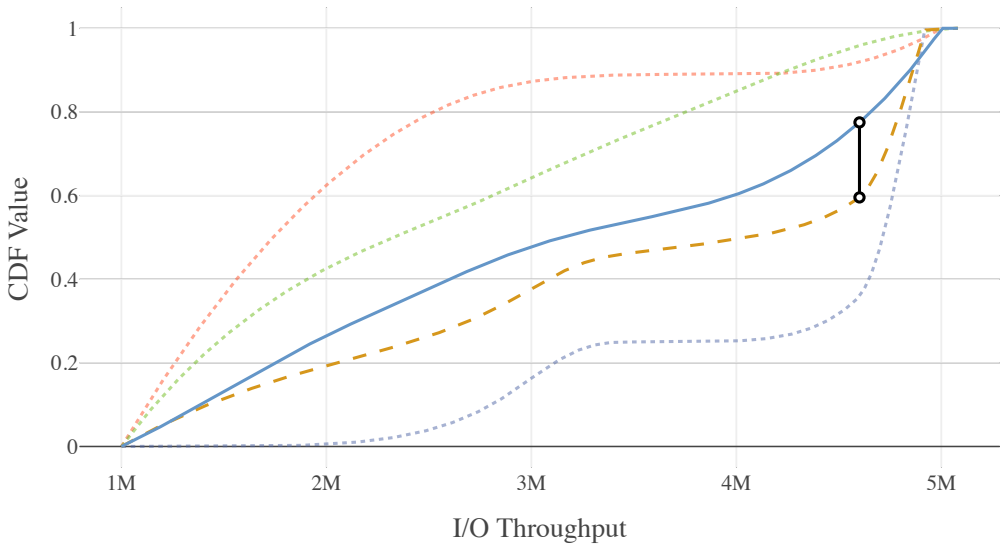

Figure 5.1: In this HPC I/O example, the general methodology for predicting a CDF and evaluating error can be seen, where M means $\times 10^6$. The Delaunay method chose three source distributions (dotted lines) and assigned weights $\{.1, .3, .6\}$ (top to bottom at middle). The weighted sum of the three known CDFs produces the predicted CDF (dashed line). The KS Statistic (vertical line) computed between the true CDF (solid line) and predicted CDF (dashed line) is 0.2 for this example. For this example the KS test null hypothesis is rejected at p-value 0.01, however it is not rejected at p-value 0.001.

System Parameter	Values
File Size (KB)	4, 16, 64, 256, 1024, 4096, 8192, 16384
Record Size (KB)	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384
Thread Count	1, 8, 16, 24, 32, 40, 48, 56, 64
Frequency (GHz)	1.2, 1.6, 2, 2.3, 2.8, 3.2, 3.5
Test Type	Readers, Rereaders, Random Readers, Initial Writers, Rewriters, Random Writers

Table 5.1: A description of system parameters considered for IOzone. Record size must be $\leq$ file size during execution.

CDFs come from the same underlying distribution) is rejected at level $p \in [0,1]$ when

$$KS > \sqrt{-\frac{1}{2}\ln\left(\frac{p}{2}\right)}\sqrt{\frac{1}{n_1}+\frac{1}{n_2}},$$

with distribution sample sizes $n_1, n_2 \in \mathcal{N}$. For all applications of the KS test presented in this work $n_1 = n_2$. An example of the process of generating predicted distributions from known distributions and the subsequent calculation of error can be seen in Figure 5.1. A brief listing of relevant statistical terms used throughout this work is provided in the Appendix (Section A).

5.1.1 Feature Weighting

It is well-known that an important procedure in any application of predictive methodologies is identifying those features of the data that are most relevant to making accurate predictions [40]. Selection strategies such as the floating searches studied in [68] or others compared in [30] can be too expensive for large approximation problems. Rather, this work poses feature selection as a continuous optimization problem. Let X be an $n \times d$ matrix of n known system configurations with d parameters each normalized to be in $[0,1]$. Define an error function that computes the error of a predictive model trained on X diag w, $w \in \mathbb{R}^d$, by performing ten random splits with 80% of the rows of X diag w for training and 20% for testing. A minimum of this error function could be considered an optimal weighting of the features of X. Minimization is performed using a zero order method in the absence of a readily computable gradient.

5.2 Variability Data

This chapter utilizes a variability modeling case study with a five-dimensional dataset produced by executing the IOzone benchmark [63] on a homogeneous cluster of computers. Each node contains

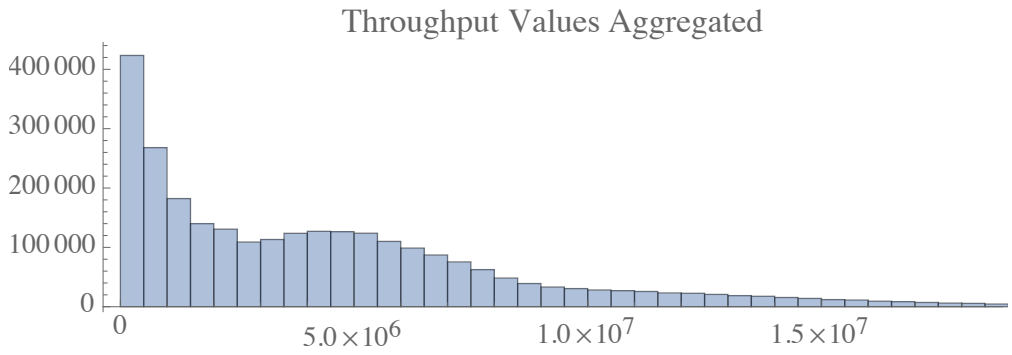

Figure 5.2: Histogram of the raw throughput values recorded during all IOzone tests across all system configurations. The distribution is skewed right, with few tests having significantly higher throughput than most others.

two Intel Xeon E5-2637 CPUs offering a total of 16 CPU cores with 16GB of DRAM. While the CPU frequency varies depending on the test configuration, the I/O from IOzone is performed by an ext4 filesystem sitting above an Intel SSDSC2BA20 SSD drive. At the time of data collection, Linux kernel Version 4.13.0 was used. The system performance data was collected over two weeks by executing IOzone 150 times for each of a select set of approximately 18K system configurations, for a total of approximately 2.7M executions of IOzone. A single IOzone execution reports the max I/O throughput in kilobytes per second seen for the selected test type. The summary of the data components in $x^{(i)}$ for the experiments for this chapter can be seen in Table 5.1. Distributions of raw throughput values being modeled can be seen in Figure 5.2.

Some mild preprocessing was necessary to prepare the data for modeling and analysis. All features were shifted by their minimum value and scaled by their range, mapping each feature independently into $[0,1]$. This normalization ensures each feature is treated equally by the interpolation techniques and should be performed on all data before building models and making predictions regardless of application. All 150 repeated trials for a system configuration were grouped with that configuration. The only nonordinal feature in this data is the test type. All test types were treated as different applications and were separated for modeling and analysis, i.e., predictions for the "readers" test type were made using only known configurations for the "readers" test type.

5.3 Distribution Prediction Results

All three interpolation techniques are used to predict the distribution of I/O throughput values at previously unseen system configurations. In order to improve robustness of the error analysis, ten random selections of 80% of the IOzone data are used to train each model and the remaining 20% provide approximation error for each model. The recorded errors are grouped by unique system configuration and then averaged within each group. The samples are identical for each interpolation technique, ensuring consistency in the training and testing sets.

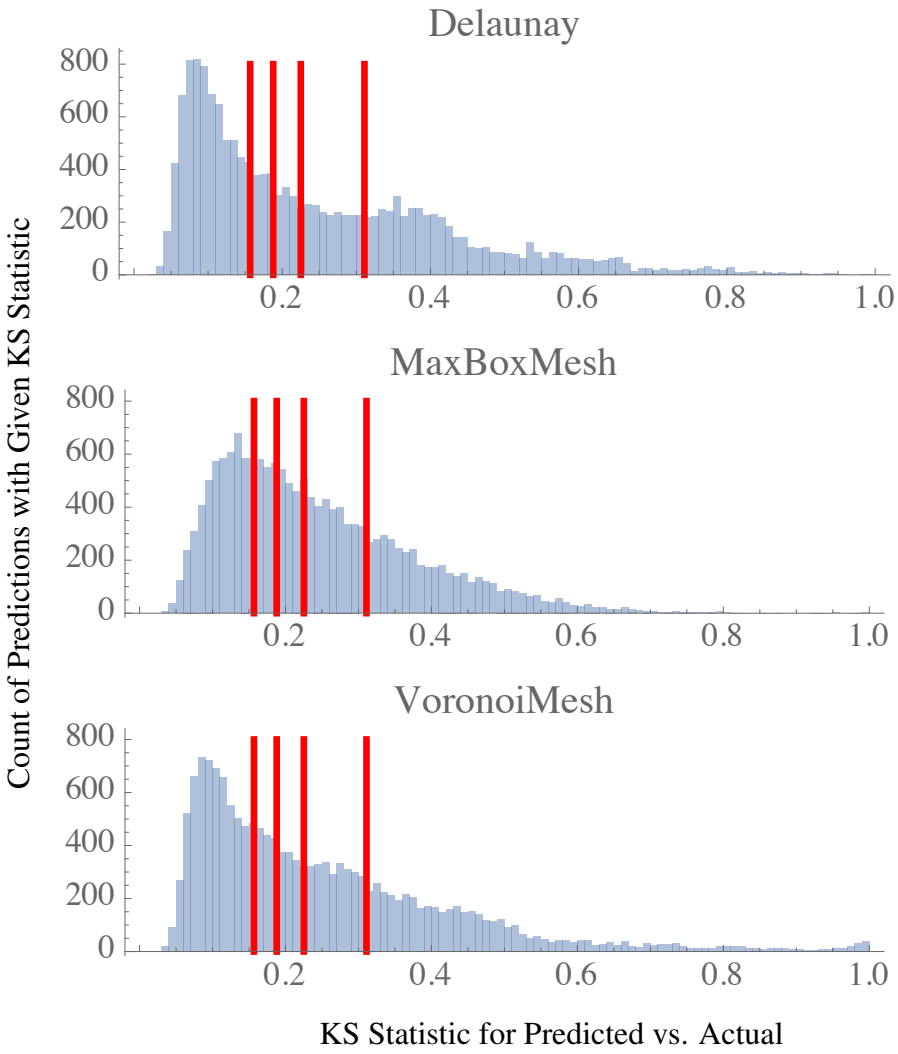

Figure 5.3: Histograms of the prediction error for each modeling algorithm from ten random splits when trained with 80% of the data aggregated over all different test types. The distributions show the KS statistics for the predicted throughput distribution versus the actual throughput distribution. The four vertical red lines represent commonly used p-values $\{0.05, 0.01, 0.001, 1.0\text{e-}6\}$ respectively. All predictions to the right of a red line represent CDF predictions that are significantly different (by respective p-value) from the actual distribution according to the KS test.

Algorithm	*P*-Value	% N.H. Rejections
Delaunay	.05	58.4
Max Box Mesh		69.3
Voronoi Mesh		61.9
Delaunay	.01	51.1
Max Box Mesh		58.4
Voronoi Mesh		53.4
Delaunay	.001	44.1
Max Box Mesh		46.9
Voronoi Mesh		45.1
Delaunay	1.0e-6	31.4
Max Box Mesh		26.6
Voronoi Mesh		28.7

Table 5.2: Percent of null hypothesis rejections rate by the KS-test when provided different selections of p-values. These accompany the percent of null hypothesis rejection results from Figure 5.3.

The aggregation of errors across all IOzone tests given 80% of the data as training can be seen in Figure 5.3. Agglomerate errors for each technique resemble a Gamma distribution. The percentages of significant prediction errors with varying p-values are on display in Table 5.2. The primary p-value used for analyses in this work is 0.001, chosen because close to 2K predictions are made for each test type. Also, applications executed in cloud and HPC systems that could benefit from statistical modeling will be executed at least thousands of times. In line with this knowledge, it is important to ensure that only a small fraction of interpretable results could occur solely under the influence of random chance. When considering the $p = 0.001$ results for each technique, a little under half of the predicted CDFs are significantly different from the measured (and presumed) correct CDFs. A rejection rate of 45% would seem a poor result, however in this situation the complexity of the problem warrants a slightly different interpretation. These predictions are a very *precise* characterization of performance variability, in fact the cumulative distribution function of a random variable is the strongest possible characterization of variability that can be predicted. Globally, only a little under half of the predictions fail to capture *all* of the characteristics of performance variability at new system configurations. It is also demonstrated later in this Section that this result can likely be improved.

While interpreting null hypothesis rejection rates for these interpolation techniques, it is important to consider how the rejection rate reduces with increasing amounts of training data. Figure 5.4 displays the change in $p = 0.001$ null hypothesis rejection rate with increasing density of training data up to the maximum density allowed by this set. Delaunay interpolation provides the best results with the least training data by about 5%, but these low density rejection rates are unacceptably high (90%). Figure 5.4 clearly shows that this data set and/or the system variables used in the models of performance variability is inadequate to capture the full variability map from system parameters

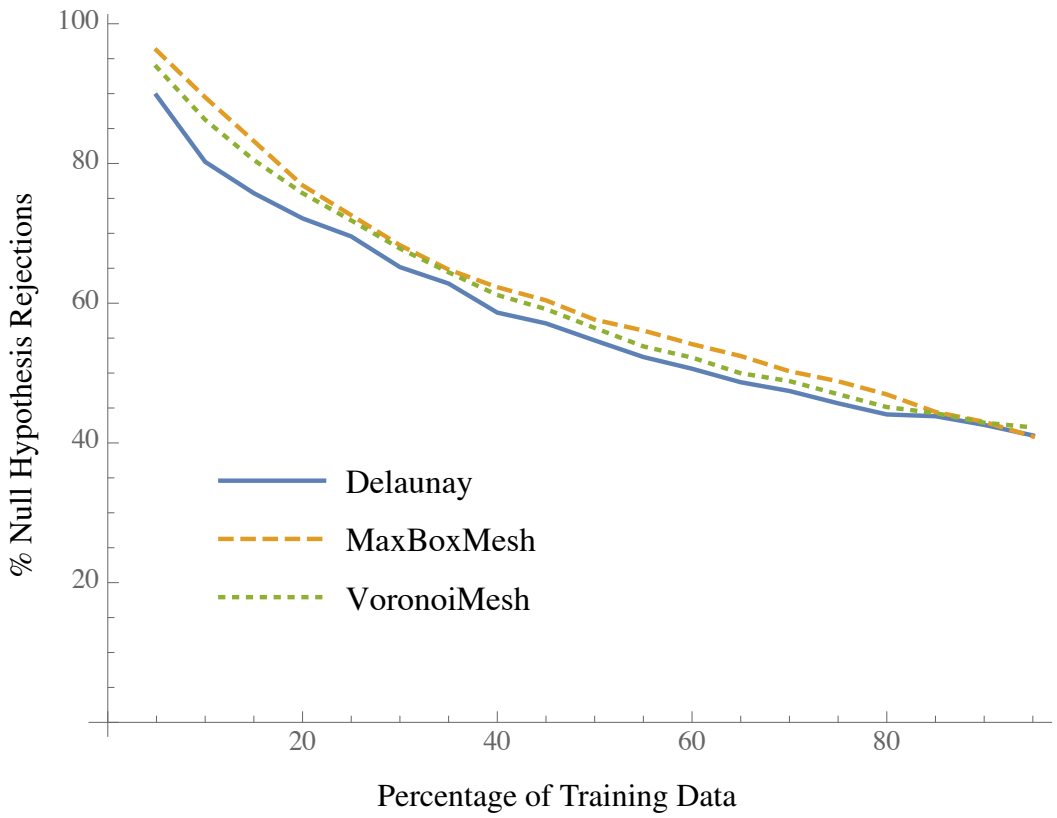

Figure 5.4: The performance of each algorithm on the KS test ($p = 0.001$) with increasing amounts of training data averaged over all IOzone test types and ten random splits of the data. The training percentages range from 5% to 95% in increments of 5%. Delaunay is the best performer until 95% of data is used for training, at which Max Box mesh becomes the best performer by a fraction of a percent.

to performance CDF. Which or both obtains is not clear. A few well chosen data points can significantly improve the interpolants, and thus a careful study of the rejection instances is warranted, besides enlarging the set of system variables being modeled.

It may be misleading to consider the global performance of each prediction technique across all test types, as some test types are more difficult than others to predict and have more apparent latent variables. In Figure 5.5, the relative difficulty of each IOzone test type can be compared. The I/O test types analyzing reads are typically approximated with lower error than those test types analyzing writes. Regardless of test type, in the aggregate results the KS statistics hover consistently around 0.15, demonstrating an impressively low KS statistic for predictions. In order to address the opacity of aggregate analysis, another case study and an application of the methodology from Section 5.1.1 is presented in Table 5.3.

The results presented in Table 5.3 are achieved by permitting each approximation technique 300 iterations of simulated annealing. In each iteration, the impact of potential weights on the average

Algorithm	*P*-Value	Unweighted % N.H. Rejection	Weighted % N.H. Rejection
Delaunay	.05	24.9	30.2
Max Box Mesh		21.3	21.2
Voronoi Mesh		18.7	11.3
Delaunay	.01	21.6	27.4
Max Box Mesh		16.4	16.4
Voronoi Mesh		14.9	7.0
Delaunay	.001	19.7	25.4
Max Box Mesh		13.1	13.1
Voronoi Mesh		12.3	4.6
Delaunay	1.0e-6	17.9	23.4
Max Box Mesh		11.3	11.3
Voronoi Mesh		8.5	2.3

Table 5.3: The null hypothesis rejection rates for various p-values with the KS-test. These results are strictly for the "readers" IOzone test type and show unweighted results as well as the results with weights tuned for minimum error (KS statistic) by 300 iterations of simulated annealing. Notice that the weights identified for the Delaunay model cause data dependent tuning, reducing performance. MaxBoxMesh performance is improved by a negligible amount. VoronoiMesh performance is notably improved.

KS statistic were considered. All weights were kept in the range [0,2], and were applied to the normalized features for frequency, file size, record size, and number of threads. All three approximation techniques had similar optimal weights achieved by simulated annealing of approximately $(.001, 2, 1.7, 1.5)$ for frequency, file size, record size, and number of threads, respectively. Recall that each interpolation technique uses small distances to denote large influences on predictions, meaning that frequency was the most important feature when predicting variability for the "readers" test type, followed not-so-closely by number of threads, then record size.

The "readers" test type results demonstrate that the underlying prediction techniques work and are capable of seeing rejection rates below 5% when tuned for a given application. It is important to emphasize that the roughly 95% of predictions for which the null hypothesis was not rejected are predicting the *precise* distribution of I/O throughput that will be witnessed at a previously unseen system configuration. To the authors' knowledge, there is no existing methodology that is generically applicable to any system performance measure, agnostic of system architecture, and capable of making such powerful predictions.

5.4 Discussion of Distribution Prediction

The results of the IOzone case study indicate that predicting the CDF of I/O throughput at previously unseen system configurations is a challenging problem. The KS statistic captures the worst part of any prediction and hence provides a conservatively large estimate of approximation error. The average absolute errors in the predicted CDFs are always lower than the KS statistics. However, the KS statistic was chosen because of the important statistical theory surrounding it as an error measure. Considering this circumstance, a nonnegligible volume of predictions provide impressively low levels of error. Powerful predictive tools such as those presented in this work allow for more in-depth analysis of system performance variability. For example, system configurations that are most difficult to predict in these tests are likely "outlier" configurations that do not resemble those configurations that share many similar parameters. Analysis of these configurations may provide valuable insight into effective application specific operation of computer systems.

As mentioned at the beginning of this chapter, no prior work has attempted to model an arbitrary performance measure for a system to such a high degree of precision. All previous statistical modeling attempts capture a few (< 3) ordinal performance measures. Generating models that have such high degrees of accuracy allows system engineers to identify previously unused configurations that present desired characteristics. Service level agreements (SLAs) in cloud computing environments are cause for capital competition that is affected heavily by system performance [65]. Users prefer SLAs that allow the most computing power per monetary unit, incentivizing service providers to guarantee the greatest possible performance. Overscheduling and irregular usage patterns force cloud service providers to occasionally overload machines, in which case precise models of system performance can be used to statistically minimize the probability of SLA violation. Similar targeted performance tuning techniques can be applied to HPC system configuration to maximize application throughput or minimize system power consumption.

A final application domain affected by this methodology is computer security. Collocated users on cloud systems have received attention recently [2]. If a malicious collocated user is capable of achieving specific insight into the configuration of the system, or the activity of other collocated users by executing performance evaluation programs (i.e., IOzone), a new attack vector may present itself. Malicious users could be capable of identifying common performance distributions of vulnerable system configurations and vulnerable active user jobs. This knowledge may allow targeted exploits to be executed. Light inspection of raw IOzone I/O throughputs provides substantial evidence that distinct performance distributions coincide closely with specific system configuration parameters. Conversely, a service provider may defend against such attacks by deliberately obfuscating the performance of the machine. Models such as those presented in this chapter could identify optimal staggering and time-delay whose introduction into the system would prevent malicious users from identifying system configurations and active jobs.

Results presented in Table 5.3 are particularly interesting, demonstrating that Delaunay appears most vulnerable to data dependent tuning, Max Box mesh is largely insensitive to such tuning, and Voronoi mesh benefits (for this data set) from the tuning.

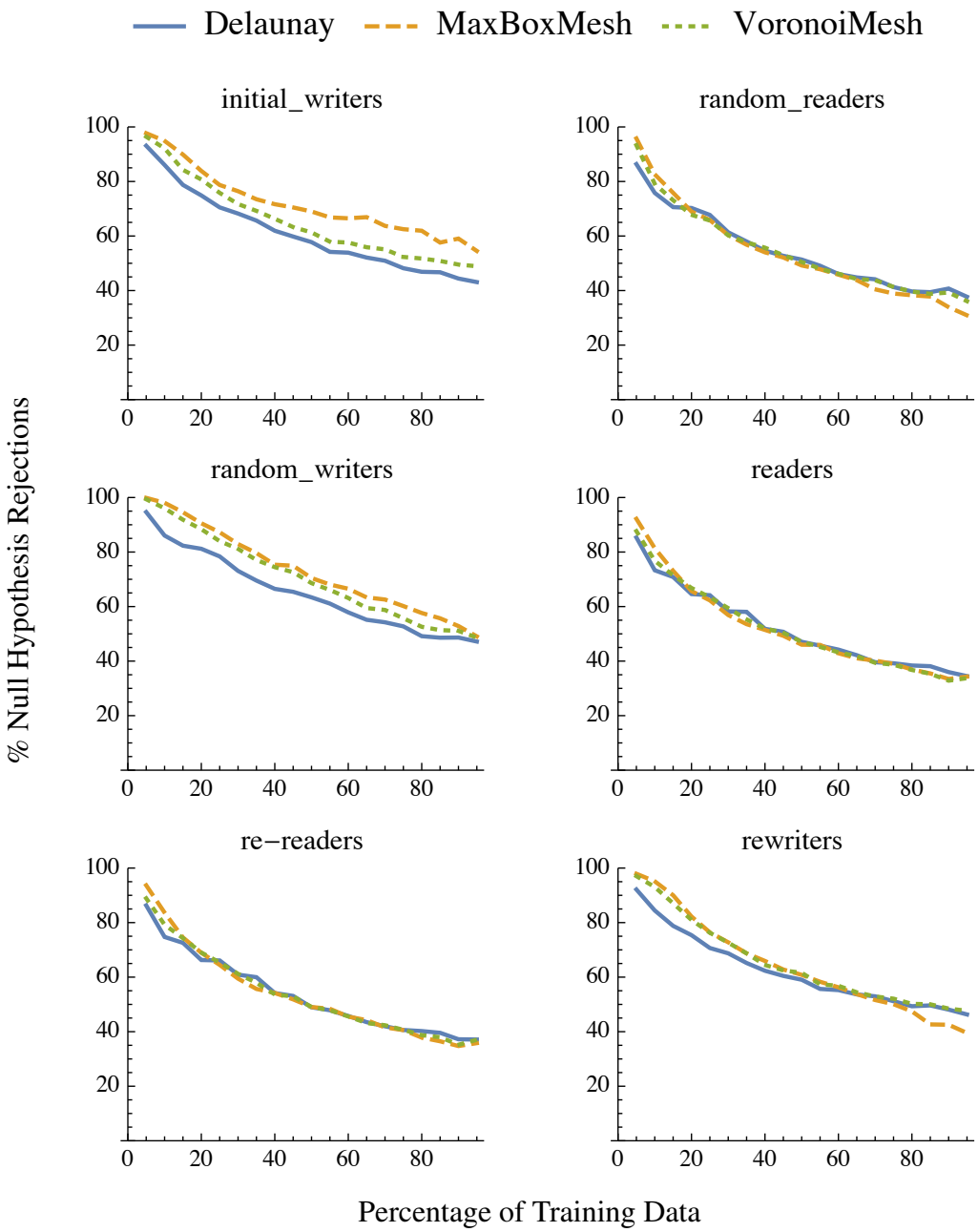

Figure 5.5: The percentage of null hypothesis rejections for predictions made by each algorithm on the KS test ($p = 0.001$) over different IOzone test types with increasing amounts of training data. Each percentage of null hypothesis rejections is an average over ten random splits of the data. The training percentages range from 5% to 95% in increments of 5%. The read test types tend to allow lower rejection rates than the write test types.

There are many avenues for extending this modeling methodology. One extension is to add categorical variables to the models. Presently the rejection rate of distribution predictions can only be reduced with large volumes of performance data, however the judicious choice (via experimental design, e.g.) of new data points may be able to effectively reduce the amount of training data required. Finally, more case studies need to be done to test the robustness of the present modeling techniques to changes in domain and performance measure.

5.5 The Power of Distribution Prediction

The methodology presented is capable of providing new insights, extending existing analyses, and improving the management of computational performance variability. Delaunay, Max Box mesh, and Voronoi mesh interpolation are viable techniques for constructing approximations of performance cumulative distribution functions. A case study on I/O throughput demonstrated that the models are capable of effectively predicting CDFs for most unseen system configurations for any of the available I/O test types. The present methodology represents a notable increase in the ability to statistically model arbitrary system performance measures involving the interaction of many ordinal system parameters.

Chapter 6

An Error Bound on Piecewise Linear Interpolation

This chapter presents theoretical results bounding the error of (piecewise) linear interpolation. The error analysis relies on linear interpolation for three reasons: (1) second order results can be obtained utilizing a Lipschitz constant on the gradient of a function, rather than standard Lipschitz bounds; (2) the results directly apply to Delaunay interpolation; and (3) multiple other interpolants in this work compute predictions as convex combinations of observed function values, which may allow for straightforward extensions of this error bound.

Lemma 1 Let $S \subset \mathbb{R}^d$ be open and convex, $f : S \to \mathbb{R}$, and let $\nabla f(x)$ be γ-Lipschitz continuous in the 2-norm. Then for all $x, y \in S$

$$\left|f(y) - f(x) - \langle \nabla f(x), y - x \rangle\right| \leq \frac{\gamma \|y - x\|_2^2}{2}.$$

Proof. Consider the function $g(t) = f\big((1-t)x + ty\big)$, $0 \leq t \leq 1$, whose derivative $g'(t) = \big\langle \nabla f\big((1-t)x + ty\big), y - x \big\rangle$ is the directional derivative of f in the direction $(y - x)$.

$$\begin{aligned}
\left|f(y) - f(x) - \langle \nabla f(x), y - x \rangle\right| &= \left|g(1) - g(0) - g'(0)\right| \\
&= \left| \int_0^1 g'(t) - g'(0)\, dt \right| \leq \int_0^1 \left|g'(t) - g'(0)\right| dt \\
&= \int_0^1 \left| \big\langle \nabla f\big((1-t)x + ty\big) - \nabla f(x), y - x \big\rangle \right| dt \\
&\leq \int_0^1 \left\| \nabla f\big((1-t)x + ty\big) - \nabla f(x) \right\|_2 \|y - x\|_2 \, dt \\
&\leq \int_0^1 \big(\gamma \|y - x\|_2\big) \, \big(\|y - x\|_2 t\big) \, dt = \frac{\gamma \|y - x\|_2^2}{2}.
\end{aligned}$$

□

Lemma 2 Let $x, y, v_i \in \mathbb{R}^d$, $c_i \in \mathbb{R}$, and $|\langle y-x, v_i\rangle| \le c_i$ for $i = 1, \ldots, d$. If $M = (v_1, \ldots, v_d)$ is nonsingular, then

$$\|y-x\|_2^2 \le \frac{1}{\sigma_d^2} \sum_{i=1}^{d} c_i^2,$$

where σ_d is the smallest singular value of M.

Proof. Using the facts that M and M^t have the same singular values, and $\|M^t w\|_2 \ge \sigma_d \|w\|_2$, gives

$$\begin{aligned}
\|y-x\|_2^2 &\le \frac{\|M^t(y-x)\|_2^2}{\sigma_d^2} \\
&= \frac{1}{\sigma_d^2} \sum_{i=1}^{d} \langle y-x, v_i\rangle^2 \\
&\le \frac{1}{\sigma_d^2} \sum_{i=1}^{d} c_i^2.
\end{aligned}$$

□

Lemma 3 Given f, γ, S as in *Lemma 1*, let $X = \{x_0, x_1, \ldots, x_d\} \subset S$ be the vertices of a d-simplex, and let $\hat{f}(x) = \langle c, x-x_0\rangle + f(x_0)$, $c \in \mathbb{R}^d$ be the linear function interpolating f on X. Let σ_d be the smallest singular value of the matrix $M = (x_1 - x_0, \ldots, x_d - x_0)$, and $k = \max_{1 \le j \le d} \|x_j - x_0\|_2$. Then

$$\left\|\nabla f(x_0) - c\right\|_2 \le \frac{\sqrt{d}\,\gamma k^2}{2\sigma_d}.$$

Proof. Consider $g(t) = f(x(t)) - \hat{f}(x(t))$ along the line segment $x(t) = (1-t)x_0 + tx_j$, $0 \le t \le 1$, from x_0 to x_j. Observe that $g(0) = f(x_0) - \hat{f}(x_0) = 0$, $g(1) = f(x_j) - \hat{f}(x_j) = 0$, and $g'(t)$ is γ_g-Lipschitz continuous with $\gamma_g = \gamma\|x_j - x_0\|_2^2$. The following is visualized in Figure 6.1.

Suppose $g'(0) > \gamma_g/2$. Then $|g'(0) - g'(t)| \le \gamma_g t \implies g'(0) - \gamma_g t \le g'(t)$, and the line $w = g'(0) - \gamma_g t$ intersects the t-axis at $\tilde{t} = g'(0)/\gamma_g > 1/2$. $\tilde{t} < 1$ necessarily, since by Rolle's Theorem there exists $0 < z < 1$ such that $g'(z) = 0$ and so $g'(0) - \gamma_g z \le g'(z) = 0$. Now by integrating

$$\frac{g'(0)^2}{2\gamma_g} = \frac{g'(0)\tilde{t}}{2} = \int_0^{\tilde{t}} (g'(0) - \gamma_g t)dt \le \int_0^{\tilde{t}} g'(t)dt = g(\tilde{t}),$$

and using $g'(0) > \gamma_g/2$

$$\begin{aligned}
\frac{-g'(0)^2}{2\gamma_g} < g'(0) - \frac{g'(0)^2}{2\gamma_g} - \gamma_g/2 = \frac{(1-\tilde{t})(g'(0)-\gamma_g)}{2} &= \int_{\tilde{t}}^{1} (g'(0) - \gamma_g t)dt \\
&\le \int_{\tilde{t}}^{1} g'(t)dt = -g(\tilde{t}),
\end{aligned}$$

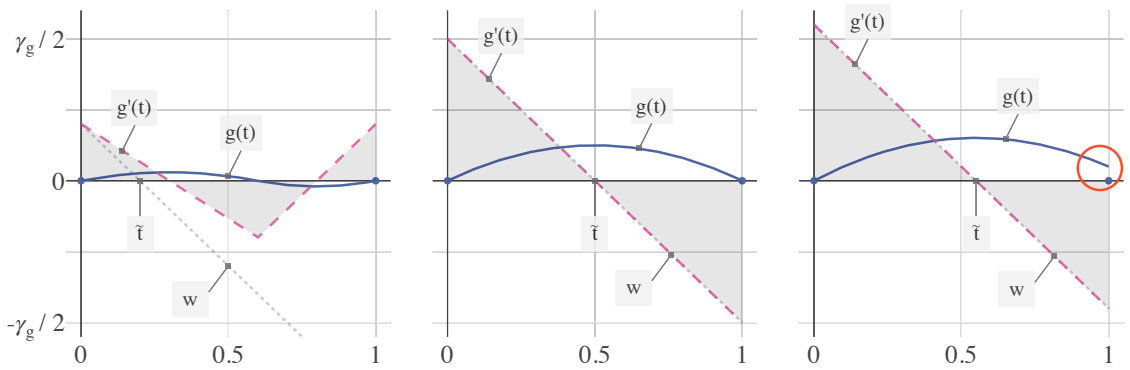

Figure 6.1: Three different scenarios visualizing *Lemma 3*, where $g(t)$ is the difference between a piecewise linear interpolant and the approximated function along a normalized line segment between interpolation points, $g'(t)$ is γ_g-Lipschitz continuous, and w and $\tilde{t}$ are defined in the proof. Leftmost is a randomly chosen permissible shape of g and g'. The middle is the only possible shape of g and g' given $g'(0) = \gamma_g/2$, establishing the case of equality in the lemma. Rightmost is the resulting contradiction when $g'(0) > \gamma_g/2$, notice it is impossible to ensure $g'(t)$ is γ_g-Lipschitz continuous and satisfy $g(1) = 0$ (highlighted with red circle on the right).

a contradiction for the value of $g(\tilde{t})$. A similar contradiction arises for $g'(0) < -\gamma_g/2$. Therefore $|g'(0)| \leq \gamma_g/2$. In terms of f,

$$\left|\langle \nabla f(x_0) - c, x_j - x_0 \rangle\right| = |g'(0)| \leq \gamma \|x_j - x_0\|_2^2/2 \leq \gamma k^2/2,$$

which holds for all $1 \leq j \leq d$. Finally, using *Lemma 2*,

$$\|\nabla f(x_0) - c\|_2^2 \leq \frac{d}{\sigma_d^2}(\gamma k^2/2)^2 \implies \|\nabla f(x_0) - c\|_2 \leq \frac{\sqrt{d}\,\gamma k^2}{2\sigma_d}.$$

□

Theorem Under the assumptions of *Lemma 1* and *Lemma 3*, for $z \in S$,

$$\left|f(z) - \hat{f}(z)\right| \leq \frac{\gamma \|z - x_0\|_2^2}{2} + \frac{\sqrt{d}\,\gamma k^2}{2\sigma_d}\|z - x_0\|_2.$$

Proof. Let $v = \nabla f(x_0) - c$, where $\|v\|_2 \leq \sqrt{d}\,\gamma k^2/(2\sigma_d)$ by *Lemma 3*. Now

$$\begin{aligned}
\left|f(z) - \hat{f}(z)\right| &= \left|f(z) - f(x_0) - \langle c, z - x_0\rangle\right| \\
&= \left|f(z) - f(x_0) - \langle \nabla f(x_0) - v, z - x_0\rangle\right| \\
&= \left|f(z) - f(x_0) - \langle \nabla f(x_0), z - x_0\rangle + \langle v, z - x_0\rangle\right| \\
&\leq \left|f(z) - f(x_0) - \langle \nabla f(x_0), z - x_0\rangle\right| + \left|\langle v, z - x_0\rangle\right| \\
&\leq \left|f(z) - f(x_0) - \langle \nabla f(x_0), z - x_0\rangle\right| + \|v\|_2 \|z - x_0\|_2 \\
&\leq \left|f(z) - f(x_0) - \langle \nabla f(x_0), z - x_0\rangle\right| + \left(\sqrt{d}\,\gamma k^2/(2\sigma_d)\right)\|z - x_0\|_2 \\
&\leq \frac{\gamma \|z - x_0\|_2^2}{2} + \frac{\sqrt{d}\,\gamma k^2}{2\sigma_d}\|z - x_0\|_2,
\end{aligned}$$

where the last inequality follows from *Lemma 1*. □

In summary, the approximation error of a linear (simplicial) interpolant tends quadratically towards zero when approaching observed data only when the diameter of the simplex is also reduced at a proportional rate. Only linear convergence to the true function is achieved in practice without the incorporation of additional observations, by moving closer to interpolation points. Notice that the approximation error is largely determined by the spacing of observed data. Predictions made by simplices whose vertices are not well-spaced (i.e., have large diameter, or are nearly contained in a hyperplane) have higher error.

That the theoretical error bound is sharp can be observed with the test function $q(x) = \|x\|_2^2, x \in \mathbb{R}^d$, and the simplex defined by vertices $X = \{0, e_1, \ldots, e_d\}$, where e_i is the i-th standard basis vector in $\mathbb{R}^d$, $e = (1, \ldots, 1) \in \mathbb{R}^d$, and 0 denotes the zero vector in any dimension. The constants relevant to the error bound are

$$\gamma = 2, \qquad \sigma_d = 1, \qquad k = 1, \qquad x_0 = 0, \qquad \hat{q}(x) = \langle e, x - 0\rangle + q(0).$$

Noting that $q(0) = 0$, the approximation error at $z = -(1/2)e$ is

$$\left|q(z) - \hat{q}(z)\right| = \left|\|z\|_2^2 - \langle e, z\rangle\right| = \left|d/4 + d/2\right| = 3d/4,$$

while the error bound from the theorem gives

$$\begin{aligned}
\left|q(z) - \hat{q}(z)\right| &\leq \frac{\gamma \|z - x_0\|_2^2}{2} + \frac{\sqrt{d}\,\gamma k}{2\sigma_d}\|z - x_0\|_2 \\
&= \|z\|_2^2 + \sqrt{d}\|z\|_2 \\
&= d/4 + d/2 = 3d/4.
\end{aligned}$$

Acknowledging that the error bound is sharp, it may be of interest to observe the error of piecewise linear approximation techniques on an analytic test function.

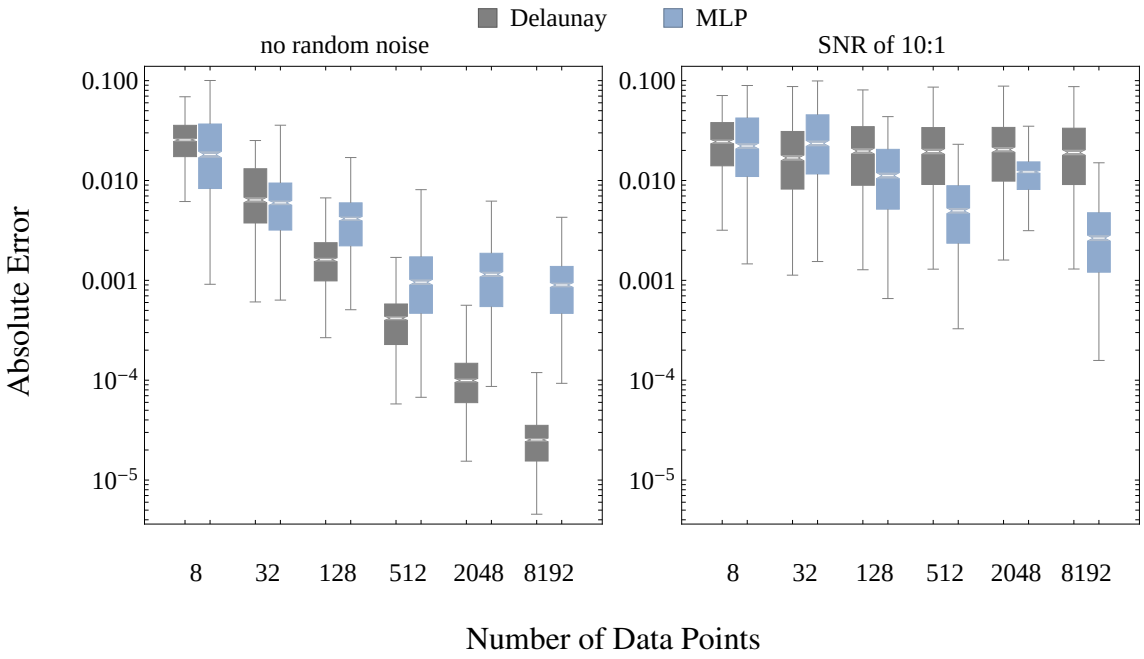

Figure 6.2: Delaunay and MLP approximations are constructed from Fekete points over the unit cube evaluating the test function $f(x) = \cos(\|x\|_2)$ for $x \in \mathbb{R}^2$. The figure shows the first/third quartiles at the box bottom/top, the second quartile (median) at the white bar, median 95% confidence interval (cones, barely visible in figure), and whiskers at 3/2 of the adjacent interquartile ranges, for the absolute prediction error for each model at 1000 random evaluation points. The left plot observes a perfect interpolation problem with exact evaluations of f. The right plot observes a regression problem with uniform random noise giving values in $[.9f(x),\ 1.1f(x)]$ for each x. Both axes are log scaled.

6.1 Demonstration on an Analytic Test Function

The theoretical results constructed in Chapter 6 for (piecewise) linear interpolation are promising and apply directly to Delaunay interpolation, however they are difficult to interpret in context with approximation algorithms that do not have similar known uniform error bounds. For that reason, an analytic function is used to measure the error of a piecewise linear interpolant (Delaunay) and a piecewise linear regressor (the MLP) when provided an increasing amount of data with varying levels of random noise. Only Delaunay and the MLP are considered in this demonstration because all other mentioned techniques are not strictly piecewise linear approximations.

The test function chosen here for analysis resembles the *oscillatory* function used by [6]. However, a slight modification is made to remove the simple dot product structure (which is favorable for the MLP). Let $f(x) = \cos(\|x\|_2)$ for $x \in \mathbb{R}^d$ where d is either 2 (Fig. 6.2) or 20 (Fig. 6.3). This function has a bounded change in gradient 2-norm and hence meets the necessary Lipschitz condition for the error bound. Data points and approximation points for this test will be within the unit cube $[0,1]^d$.

The error of a linear interpolant constructed from well spaced points is dominated by the distance to the nearest data point. In order to uniformly decrease the distance to the nearest data point across the unit cube, exponentially more data points must be available for approximation. For this experiment $N = 2(4^i)$, $i = 1, \ldots, 6$, chosen points are kept well spaced by computing approximate Fekete points. The following experiment was also run with a Latin hypercube design [64], which produced almost identical results that are not reported here. Fekete points have a history in potential theory [48] and are most generally defined as those points that maximize the absolute value of a Vandermonde determinant. Here, the QR method outlined in [11] is used to identify approximate Fekete points from a Vandermonde matrix. A multivariate polynomial basis of size $2N$ is constructed containing all polynomials of degree $\leq n$ for the largest n such that $\binom{d+n}{n} \leq 2N$ and $2N - \binom{d+n}{n}$ arbitrarily selected polynomials of degree $n+1$. The Vandermonde matrix for these $2N$ polynomials is used to select N approximate Fekete points.

Finally an additional aspect is added to this test problem by incorporating random uniform noise into the evaluations of f. For each test two experiments are executed, one with exact function evaluations (an interpolation problem) and one with a constant signal-to-noise ratio (SNR) of 10:1 (a regression problem).

The bound from the theorem suggests that by increasing the number of well-spaced data points, approximation error can be reliably decreased. The $d = 2$ test seen on the left half of Figure 6.2 shows a consistent decrease in error for Delaunay and also shows the eventual accuracy plateau obtained by a parametric regression form (the MLP, at roughly 500 points). On the right hand side of Figure 6.2 the random noise clearly prohibits Delaunay from converging to f, while the MLP is able to improve its approximation with more data points on average. The convergence result for a very low-dimensional problem like this is expected. However, intuition fails for higher dimensional problems.

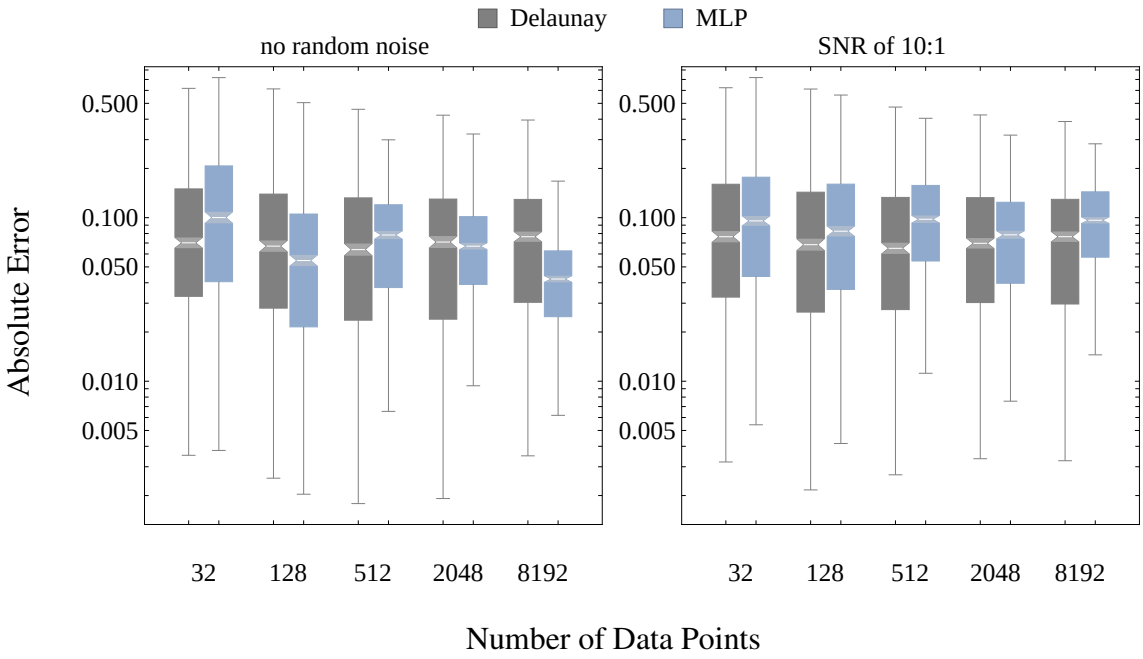

Figure 6.3: Delaunay and MLP approximations are constructed from Fekete points over the unit cube evaluating the test function $f(x) = \cos(\|x\|_2)$ for $x \in \mathbb{R}^{20}$. The details are the same as for Fig. 6.2.

Figure 6.3 shows the test function with $d = 20$ presents a significantly more challenging approximation problem than its counterpart in low dimension. The same increase in number of data points from 32 to 8192 causes no apparent improvement in approximation for either the noise-free or the noisy problems. Perhaps unexpectedly, the interpolation technique performs better than the regression technique on the noisy data (right) in Figure 6.3, and worse than the regression technique on the noise-free data (left). This result emphasizes the relevance of interpolation for problems in high dimension. It also reveals that the outcome of MLP regressions can get worse when adding more data points.

The results achieved for both $d = 2$ and $d = 20$ align with the theoretical error bound as can be seen in Figure 6.4. In low dimension, thousands of data points meaningfully reduce the sparsity of the approximation problem. However, in higher dimension it takes many more points to achieve a reduction in data sparsity while simultaneously being more difficult to produce well-conditioned simplices. This evidences the inherent challenge of data sparsity in high dimension approximation problems. The analytic results presented are not caused by the chosen test function, but rather the exponential increase in complexity that accompanies increased dimension.

In summary, the regime of signal-to-noise ratios that result in competitively accurate interpolants is greater for moderate to high dimensional problems due to increased sparsity. Acknowledging the viability of interpolation for problems of moderate dimension, the next section will consider real-world problems of similar proportion (thousands of examples in tens of dimensions).

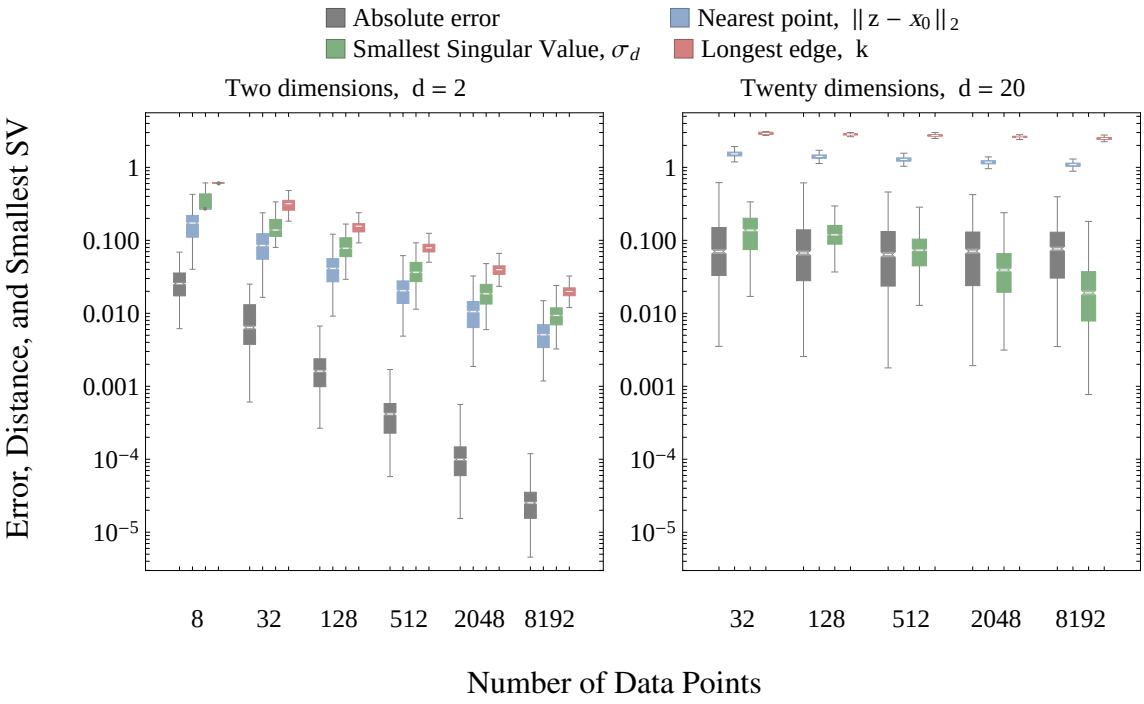

Figure 6.4: The distribution of absolute error, distance to the nearest data point, smallest singular value (SV) and the longest edge of the simplex containing each approximation point in the tests from Fig. 6.2 and Fig. 6.3 for Delaunay. In two dimensions it can be seen that $\|z - x_0\|_2$, σ_d, and k all shrink at the same rate for well-spaced approximation points, resulting in a faster rate of decrease for approximation error. Notice that in higher dimension the data remains sparse even with thousands of data points, and the decay in data spacing is more prominant. The relatively small reduction in k along with the decrease in σ_d explain the minimal reduction in error seen by Delaunay in Fig. 6.3.

6.2 Data and Empirical Analysis

This section extends the comparison of interpolation and regression algorithms to a sample of real-world problems. Five different data sets of varying dimension and application are utilized to construct approximations and compare the accuracy of different techniques.

In the following five subsections the sources and targets of each test data set are described, as well as challenges and limitations related to approximating these data. The distribution of response values being modeled is presented followed by the distribution of approximation errors for each algorithm. The plots for all five data sets have the same format.

All five data sets are rescaled such that the domain of approximation is the unit hypercube. The range of the first four data sets is the real numbers, while the range of the fifth data set is the space of cumulative distribution functions. All approximation techniques are applied to the first four data sets, while only those interpolants whose approximations are convex combinations of observed data are applied to the final data set.

All approximations are constructed using k-fold cross validation as described in [47] with $k = 10$. This approach randomly partitions data into k (nearly) equal sized sets. Each algorithm is then evaluated by constructing an approximation over each unique union of $k-1$ elements of the partition, making predictions for points in the remaining element. As a result, each observed data point is used in the construction of $k-1$ different approximations and is approximated exactly once. The k-fold cross validation method is data-efficient and provides an unbiased estimate of the expected prediction error [47], however it should be noted that neither this method nor others can provide a universally unbiased estimator for the variance of prediction error [9].

In addition to the figures displaying approximation results for each data set, scatter plots of predicted versus actual values and tables of accompanying numerical results including timings and quartiles of prediction error are located in the Appendix (Section A). All of the test data sets capture underlying functions that are almost certainly stochastic. As described in Chapter 2, regression techniques appear most appropriate for these problems. However, typically data grows exponentially more sparse with increasing dimension. Given that sparse data regressors tend towards interpolation and, as demonstrated in Section 6.1, interpolants produce similar (if not identical) results, it is presumed that interpolants are equally viable approximation techniques on these problems.

6.2.1 Forest Fire ($n = 504$, $d = 12$)

The forest fire data set [18] describes the area of Montesinho park burned over months of the year along with environmental conditions. The twelve dimensions being used to model burn area are the x and y spatial coordinates of burns in the park, month of year (mapped to x, y coordinates on a unit circle), the FFMC, DMC, DC, and ISI indices (see source for details), the temperature, relative humidity, wind speed, and outdoor rain. The original analysis of this data set demonstrated it to be difficult to model, likely due to the skew in response values.

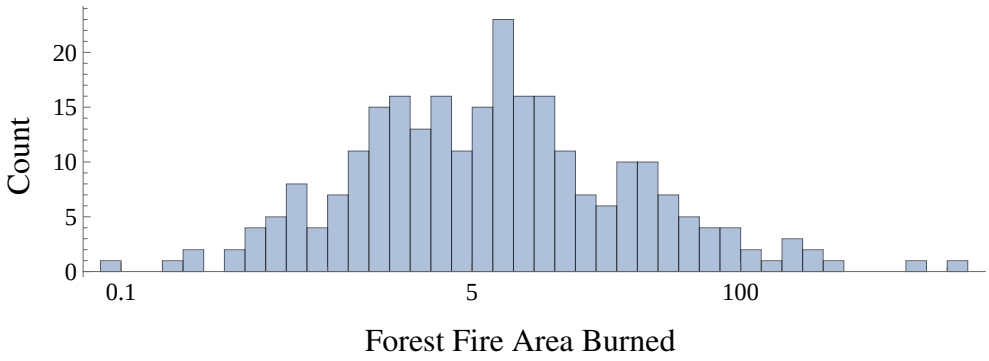

Figure 6.5: Histogram of forest fire area burned under recorded weather conditions. The data is presented on a ln scale because most values are small with exponentially fewer fires on record that burn large areas.

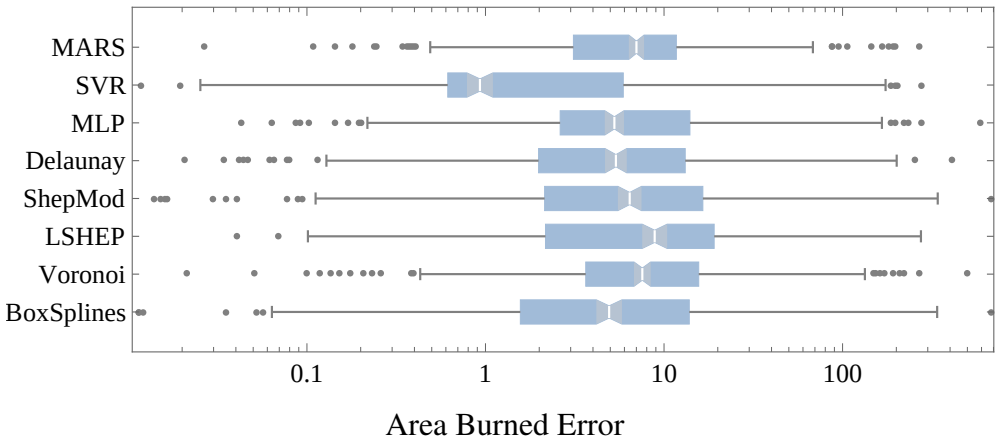

Figure 6.6: All models are applied to approximate the amount of area that would be burned given environment conditions. 10-fold cross validation as described in the beginning of Section 6.2 is used to evaluated each algorithm. This results in exactly one prediction from each algorithm for each data point. These boxes depict the median (middle bar), median 95% confidence interval (cones), quartiles (box edges), fences at $3/2$ interquartile range (whiskers), and outliers (dots) of absolute prediction error for each model. Similar to Figure 6.5, the errors are presented on a ln scale. The numerical data corresponding to this figure is provided in Table A.1 in the Appendix.

As suggested by Figure 6.6, the SVR has the lowest absolute prediction errors for 80% of the data, with MLP and Delaunay being the nearest overall competitors. The effectiveness of SVR on this data suggests the underlying function can be defined by relatively few parameters, as well as the importance of capturing the low-burn-area data points.

6.2.2 Parkinson's Telemonitoring ($n = 5875$, $d = 19$)

The second data set for evaluation [77] is derived from a speech monitoring study with the intent to automatically estimate Parkinson's disease symptom development in Parkinson's patients. The function to be approximated is a time-consuming clinical evaluation measure referred to as the UPDRS score. The total UPDRS score given by a clinical evaluation is estimated through 19 real numbers generated from biomedical voice measures of in-home sound recordings.

Figure 6.8 shows the ShepMod algorithm has the lowest minimum, first quartile, and median of absolute errors for this problem, while providing the best prediction 66% of the time. The MLP has the lowest third quartile and provides the best prediction for 32% of approximations. The dominance of ShepMod may be due in part to regular-interval total UPDRS scores provided by clinicians, favoring a nearest-neighbor strategy of prediction.

6.2.3 Australian Daily Rainfall Volume ($n = 2609$, $d = 23$)

The third data set for the total daily rainfall in Sydney, Australia [82] provides a slightly higher dimensional challenge for the interpolants and regressors. This data is composed of many meteorological readings from the region in a day including: min and max temperatures, sunshine, wind speed directions (converted to coordinates on a circle), wind speeds, and humidities throughout the day, day of the year (converted to coordinates on a circle), and the model must predict the amount of rainfall tomorrow.

While Figure 6.9 makes MARS look far better than other techniques, it only provides the best prediction for 11% of points. The MLP has the lowest absolute error for 56% of points and LSHEP is best for 28%. MARS likely achieves such a low first quartile due to the high occurrence of the value zero in the data.

6.2.4 Credit Card Transaction Amount ($n = 5562$, $d = 28$)

The fourth test data set, and the final with a real-valued range, is a collection of credit card transactions [22]. The provided data carries no direct real-world meaning, being the output of a principle component analysis on the original hidden source data. This obfuscation is done to protect the information of the credit card users. This data has the largest dimension of all considered, at 28.

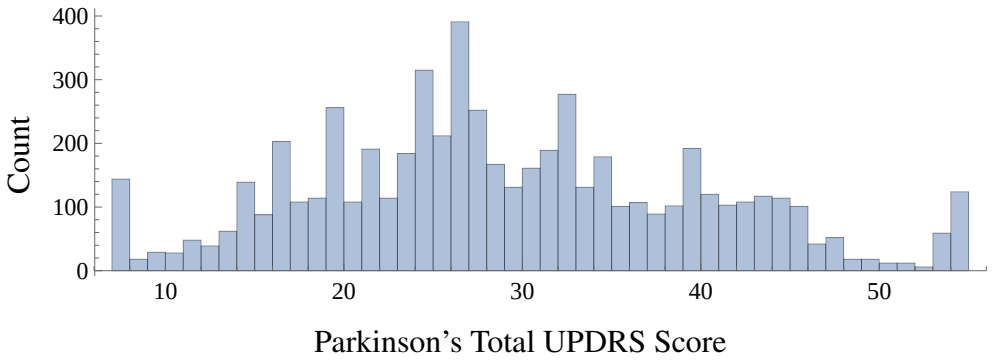

Figure 6.7: Histogram of the Parkinson's patient total UPDRS clinical scores that will be approximated by each algorithm.

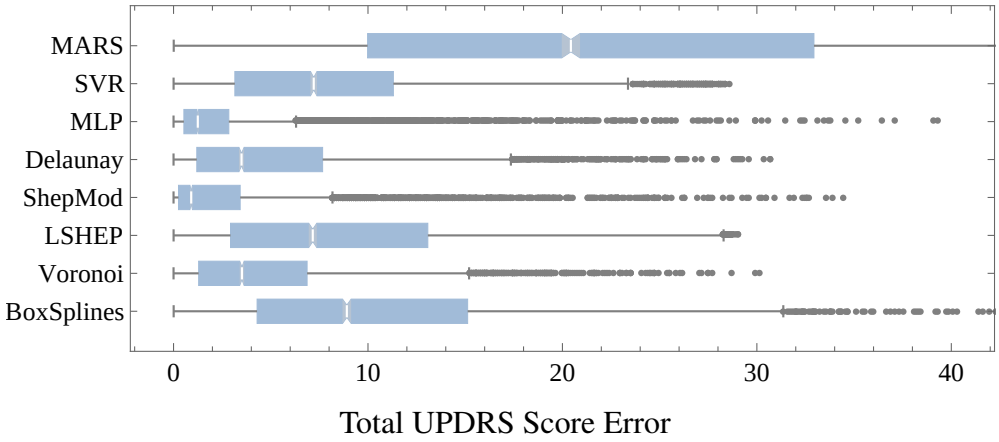

Figure 6.8: All models are applied to approximate the total UPDRS score given audio features from patients' home life, using 10-fold cross validation. These boxes depict the median (middle bar), median 95% confidence interval (cones), quartiles (box edges), fences at $3/2$ interquartile range (whiskers), and outliers (dots) of absolute prediction error for each model. The numerical data corresponding to this figure is provided in Table A.3 in the Appendix.

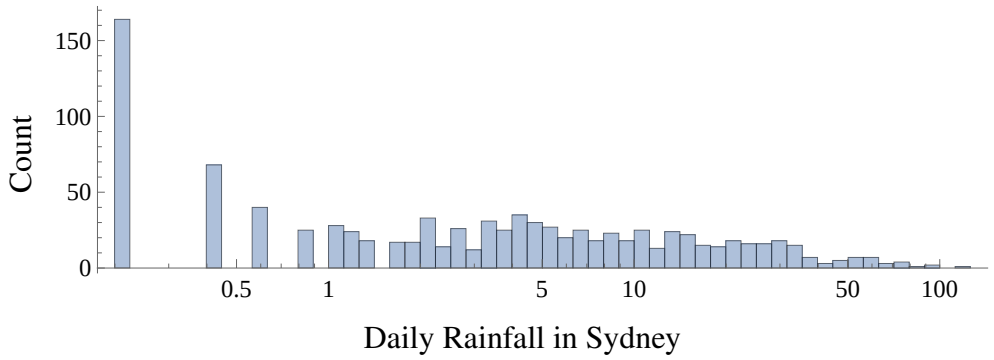

Figure 6.9: Histogram of daily rainfall in Sydney, Australia, presented on a ln scale because the frequency of larger amounts of rainfall is significantly less. There is a peak in occurrence of the value 0, which has a notable effect on the resulting model performance.

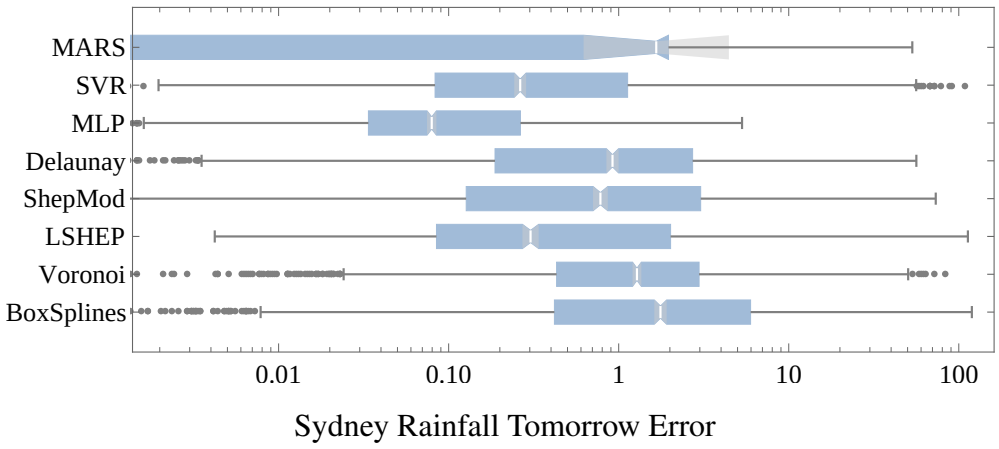

Figure 6.10: All models are applied to approximate the amount of rainfall expected on the next calendar day given various sources of local meteorological data, using 10-fold cross validation. These boxes depict the median (middle bar), median 95% confidence interval (cones), quartiles (box edges), fences at $3/2$ interquartile range (whiskers), and outliers (dots) of absolute prediction error for each model. The errors are presented on a ln scale, mimicking the presentation in Figure 6.9. The numerical data corresponding to this figure is provided in Table A.5 in the Appendix.

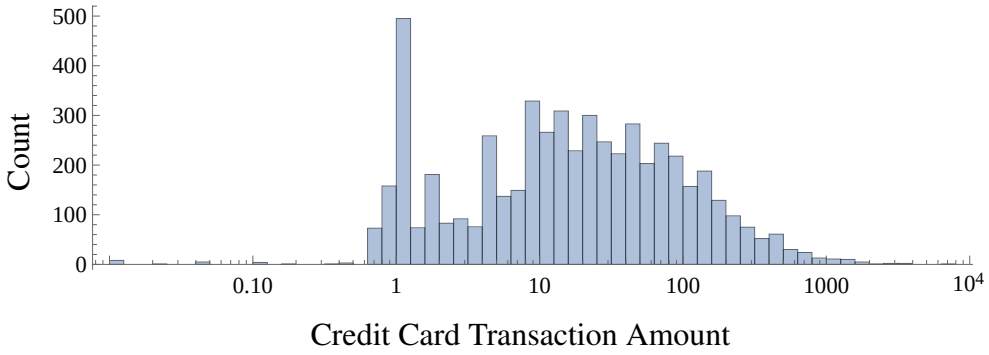

Figure 6.11: Histogram of credit card transaction amounts, presented on a ln scale. The data contains a notable frequency peak around $1 transactions. Fewer large purchases are made, but some large purchases are in excess of five orders of magnitude greater than the smallest purchases.

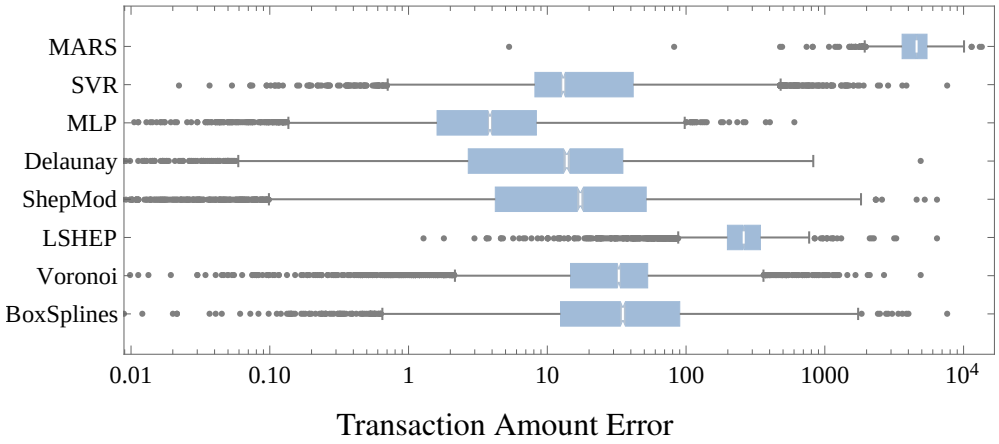

Figure 6.12: All models are applied to approximate the expected transaction amount given transformed (and obfuscated) vendor and customer-descriptive features, using 10-fold cross validation. These boxes depict the median (middle bar), median 95% confidence interval (cones), quartiles (box edges), fences at $3/2$ interquartile range (whiskers), and outliers (dots) of absolute prediction error for each model. The absolute errors in transaction amount predictions are presented on a ln scale, just as in Figure 6.11. The numerical data corresponding to this figure is provided in Table A.7 in the Appendix.

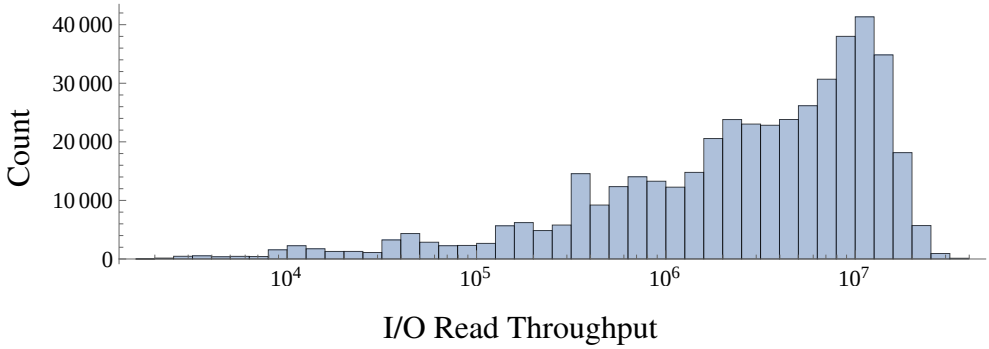

Figure 6.13: Histogram of the raw throughput values recorded during all IOzone tests across all system configurations. The distribution is skewed right, with few tests having significantly higher throughput than most others. The data is presented on a ln scale.

A model for this data predicts the transaction amount given the vector of principle component information.

As can be seen in Figure 6.12, the MLP outperforms all other algorithms at the first, second, third, and fourth quartiles. The MLP produces the lowest absolute error prediction for 80% of transactions, Delaunay bests the remaining 20%. It is likely that with less data, Delaunay would be the best performer.

6.2.5 High Performance Computing I/O ($n = 3016$, $d = 4$)

The final of five data sets is derived from [13], which provides four-dimensional distribution data by executing the IOzone benchmark [63] on a computer system and varying the system's file size, record size, thread count, and CPU frequency. At each configuration, IOzone samples the I/O file-read throughput (in bytes per second) 150 times. Empirical distribution function points are computed from each set of 150 executions, which are interpolated with a piecewise cubic Hermite interpolating polynomial [34] to approximate the CDF. All interpolation algorithms with the exception of LSHEP are used to approximate these CDFs from system configurations.

Delaunay achieves the lowest KS statistic (max norm difference) for 62% of approximations, while Voronoi is best for the remaining 38%. Figure 6.14 shows that while Delaunay may have more best predictions, the behavior of Voronoi may be preferable.

Figure 6.15 expands on the KS statistic results presented in Figure 6.14. Agglomerate errors for each algorithm resemble a Gamma distribution. The percentages of significant prediction errors with varying p-values are on display in Table 6.1. When considering the $p = 0.001$ results for each technique, a little over one third of the predicted CDFs are significantly different from the measured (and presumed) correct CDFs. However, it should be noted that with 150 samples, the error of an empirical distribution function (EDF) can reasonably be upwards of .1, which serves as a rough

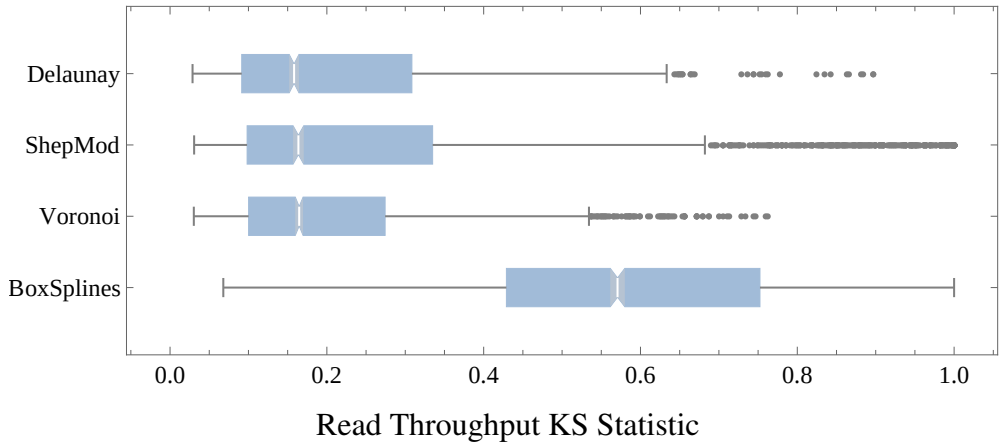

Figure 6.14: The models directly capable of predicting distributions are applied to predicting the expected CDF of I/O throughput at a previously unseen system configuration, using 10-fold cross validation. The KS statistic (max norm) between the observed distribution at each system configuration and the predicted distribution is recorded and presented above. Note that the above figure is *not* log-scaled like Figure 6.13. The numerical data corresponding to this figure is provided in Table A.9 in the Appendix.

	$p = .05$	$p = .01$	$p = .001$	$p = 10^{-6}$
Delaunay	**50.3**%	*43.5%*	*36.2%*	*24.7%*
ShepMod	*51.4%*	44.8%	38.1%	27.7%
Voronoi	52.6%	**43.4**%	**34.4**%	**19.1**%
BoxSplines	99.4%	98.5%	96.6%	89.3%

Table 6.1: Numerical counterpart of the histogram data presented in Figure 6.15. The columns display the percent of null hypothesis rejections by the KS-test when provided different selections of p-values for each algorithm. The algorithm with the lowest rejection rate at each p is boldface, while the second lowest is italicized.

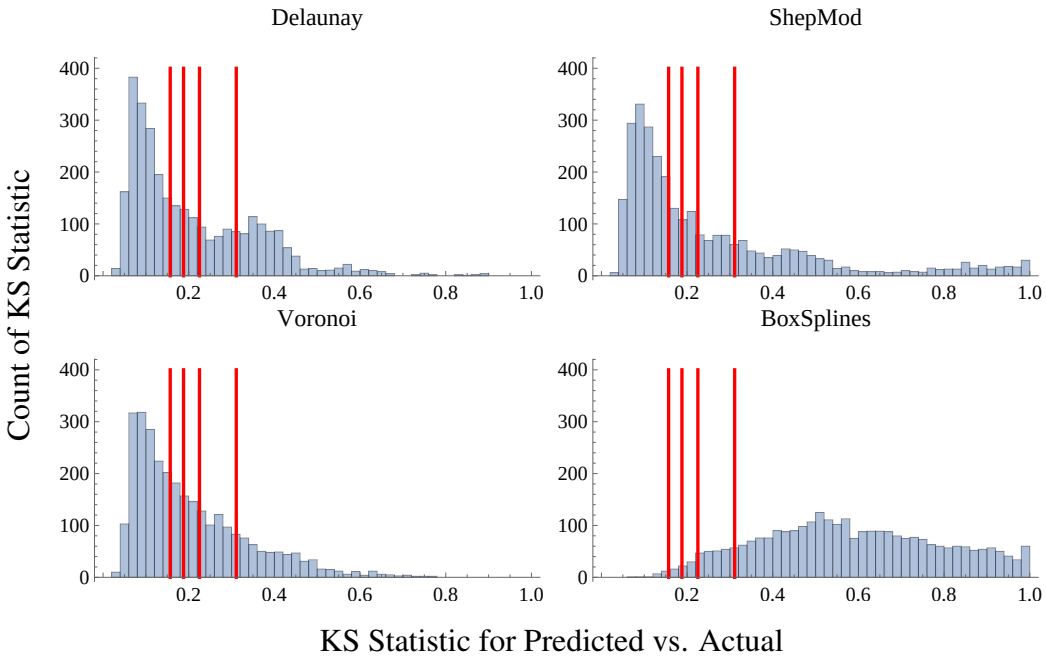

Figure 6.15: Histograms of the prediction error for each interpolant that produces predictions as convex combinations of observed data, using 10-fold cross validation. The histograms show the KS statistics for the predicted throughput distribution versus the actual throughput distribution. The four vertical lines represent cutoff KS statistics given 150 samples for commonly used p-values 0.05, 0.01, 0.001, 10^{-6}, respectively. All predictions to the right of a vertical line represent CDF predictions that are significantly different (by respective p-value) from the actual distribution according to the KS test. The numerical counterpart to this figure is presented in Table 6.1.

Algorithm	Avg. % Best	Avg. Fit or Prep. Time (s)	Avg. App. Time (s)
MARS	4.5	20.0s	0.001s
SVR	*19.5*	**0.5**s	**0.0001**s
MLP	**43.1**	200.0s	0.001s
Delaunay	5.2	1.0s	1.0s
ShepMod	18.0	*0.7*s	**0.0001**s
LSHEP	8.4	2.0s	**0.0001**s
Voronoi	0.5	1.0s	0.04s
BoxSplines	3.5	0.8s	*0.0005*s

Table 6.2: This average of Appendix Tables A.2, A.4, A.6, and A.8 provides a gross summary of overall results. The columns display (weighted equally by data set, *not* points) the average frequency with which any algorithm provided the lowest absolute error approximation, the average time to fit/prepare, and the average time required to approximate one point. The times have been rounded to one significant digit, as reasonably large fluctuations may be observed due to implementation hardware. Interpolants provide the lowest error approximation for nearly one third of all data, while regressors occupy the other two thirds. This result is obtained without any customized tuning or preprocessing to maximize the performance of any given algorithm. In practice, tuning and preprocessing may have large effects on approximation performance.

estimate for the lower limit of achievable error. Globally, only a third of Voronoi predictions fail to capture *all* of the characteristics of the CDFs at new system configurations.

6.3 Discussion of Empirical Results

Table 6.2 summarizes results across the four test data sets with real-valued ranges. The interpolants discussed in this work produce the *best* approximations roughly one third of the time, and produce competitive approximations for almost all data sets. These test problems are almost certainly *stochastic* in nature, but the high dimension leads to greater data sparsity and model construction cost, making interpolation more competitive.

The major advantages to interpolation lie in the near absence of *fit* time. Delaunay, LSHEP, and ShepMod all require pairwise distance calculations, for numerical robustness (Delaunay) and to determine the radii of influence for data points (LSHEP and ShepMod). At least hundreds, and sometimes hundreds of thousands of predictions can be made by the interpolants before the most widely used regressor (MLP) finishes fitting these relatively small data sets. However, the computational complexities of all interpolants presented are greater than linear in either dimension or number of points, whereas the regressors' nonlinear complexity in dimension generally comes from the model fitting optimization.

The new theoretical results presented at the beginning of this chapter directly apply to Delaunay interpolation, however the performance of Delaunay does not appear significantly better than other algorithms on these data sets. This observation may be due to the stochastic nature of the data, but it also speaks to the power of the approximations generated by the different interpolation methods. The strong performance of other interpolants suggests that theoretical results similar to those presented in this work can be achieved for the other interpolants under reasonable assumptions.

Finally, most of the interpolants presented in this work benefit from the ability to model *any* function over real d-tuples with a range that is closed under convex combinations. In general, error can be quantified by any measure (particularly of interest may be L^2, L^∞, etc.). The results of the distribution prediction case study indicate that interpolants can effectively predict CDFs. The error analysis for that work relies on the KS statistic, which captures the worst part of any prediction and hence provides a conservatively large estimate of approximation error. The average absolute errors in the predicted CDFs are always lower than the KS statistics. However, the KS statistic was chosen as a metric because of the important surrounding statistical theory. A nonnegligible volume of predictions provide impressively low levels of average absolute error in that final case study.

6.4 Takeaway from Empirical Results

The major contributions of this work are: 1) new uniform theoretical error bounds for piecewise linear interpolation in arbitrary dimension; 2) an empirical evaluation across real-world problems that demonstrates interpolants produce competitively accurate models of multivariate phenomenon when compared with common regressors for sparse, moderately high dimensional problems (Section 6.2); and 3) a demonstration that some interpolants generalize to interpolation in function spaces, preserving monotonicity (with CDFs, e.g.), neither of which common regressors can do.

The various interpolants discussed in this work have been demonstrated to effectively approximate multivariate phenomena up to 30 dimensions. The underlying constructions are theoretically straightforward, interpretable, and yield reasonably accurate predictions. Most of the interpolants' computational complexities make them particularly suitable for applications in even higher dimension. The major benefits of interpolation are seen when only a small number of approximations (≤ 1000) are made from data and when there are relatively few data points for the dimension (for empirical results presented, $\log_d n \leq 5$). These findings encourage broader application of interpolants to multivariate approximation in science.

Chapter 7

A Package for Monotone Quintic Spline Interpolation

7.1 Relevance of Quintic Splines

Many domains of science rely on smooth approximations to real-valued functions over a closed interval. Piecewise polynomial functions (splines) provide the smooth approximations for animation in graphics [43, 69], aesthetic structural support in architecture [12], efficient aerodynamic surfaces in automotive and aerospace engineering [12], prolonged effective operation of electric motors [10], and accurate nonparametric approximations in statistics [46]. While polynomial interpolants and regressors apply broadly, splines are often a good choice because they can approximate globally complex functions while minimizing the local complexity of an approximation.

It is often the case that the true underlying function or phenomenon being modeled has known properties like convexity, positivity, various levels of continuity, or monotonicity. Given a reasonable amount of data, it quickly becomes difficult to achieve desirable properties in a single polynomial function. In general, the maintenance of function properties through interpolation/regression is referred to as *shape preserving* [34, 37]. The specific properties the present algorithm will preserve in approximations are monotonicity and C^2 continuity. In addition to previously mentioned applications, these properties are crucially important in statistics to the approximation of a cumulative distribution function and subsequently the effective generation of random numbers from a specified distribution [70]. A spline function with these properties could approximate a cumulative distribution function to a high level of accuracy with relatively few intervals. A twice continuously differentiable approximation to a cumulative distribution function (CDF) would produce a corresponding probability density function (PDF) that is continuously differentiable, which is desirable.

The currently available software for monotone piecewise polynomial interpolation includes quadratic [42], cubic [34], and (with limited application) quartic [67, 81, 83] cases. In addition, a statistical method for bootstrapping the construction of an arbitrarily smooth monotone fit exists [51], but the method does not take advantage of known analytic properties of quintic polynomials. The code by Fritsch [33] for C^1 cubic spline interpolation is the predominantly utilized code for constructing monotone interpolants at present. Theory has been provided for the quintic case [44, 78] and that theory was recently utilized in a proposed algorithm [60] for monotone quintic spline construction, however no published mathematical software exists.

The importance of piecewise quintic interpolation over lower order approximations can be simply observed. In general, the order of a polynomial determines the number of function (and derivative) values it can interpolate, which in turn determines the growth rate of error away from interpolated values. C^2 quintic (order six) splines match the function value and two given derivatives at each breakpoint. This work provides a Fortran 2003 subroutine `MQSI` based on the necessary and sufficient conditions in Ulrich and Watson [78] for the construction of monotone quintic spline interpolants of monotone data. Precisely, the problem is, given a strictly increasing sequence $X_1 < X_2 < \cdots < X_n$ of breakpoints with corresponding monotone increasing function values $Y_1 \le Y_2 \le \cdots \le Y_n$, find a C^2 monotone increasing quintic spline $Q(x)$ with the same breakpoints satisfying $Q(X_i) = Y_i$ for $1 \le i \le n$. (`MQSI` actually does something slightly more general, producing $Q(x)$ that is monotone increasing (decreasing) wherever the data is monotone increasing (decreasing).)

The remainder of this chapter is structured as follows: Section 2 provides the algorithms for constructing a C^2 monotone quintic spline interpolant to monotone data, Section 3 outlines the method of spline representation (B-spline basis) and evaluation, Section 4 analyzes the complexity and sensitivity of the algorithms in `MQSI`, and Section 5 presents an empirical performance study and some graphs of constructed interpolants.

7.2 Monotone Quintic Interpolation

In order to construct a monotone quintic interpolating spline, two primary problems must be solved. First, reasonable derivative values at data points need to be estimated. Second, the estimated derivative values need to be modified to enforce monotonicity on all polynomial pieces.

Fritsch and Carlson [34] originally proposed the use of central differences to estimate derivatives, however this often leads to extra and unnecessary *wiggles* in the spline when used to approximate second derivatives. In an attempt to capture the local shape of the data, this package uses a facet model from image processing [41] to estimate first and second derivatives at breakpoints. Rather than picking a local linear or quadratic fit with minimal residual, this work uses a quadratic facet model that selects the local quadratic interpolant with minimum magnitude curvature.

Algorithm 1: `QUADRATIC_FACET`$\big(X(1{:}n), Y(1{:}n), i\big)$
where $X_j, Y_j \in \mathbb{R}$ for $j = 1, \ldots, n$, $1 \le i \le n$, and $n \ge 3$. Returns the slope and curvature at X_i of the local quadratic interpolant with minimum magnitude curvature.

```
if ((i ≠ 1 ∧ Y_i ≈ Y_{i-1}) or (i ≠ n ∧ Y_i ≈ Y_{i+1})) then return (0,0)
else if i = 1 then
  f_1 := interpolant to (X_1,Y_1), (X_2,Y_2), and (X_3,Y_3).
  if (Df_1(X_1)(Y_2 − Y_1) < 0) then return (0,0)
  else return (Df_1(X_1), D^2 f_1)
  endif
```

```
else if i = n then
  f_1 := interpolant to (X_{n-2}, Y_{n-2}), (X_{n-1}, Y_{n-1}), and (X_n, Y_n).
  if (Df_1(X_n)(Y_n - Y_{n-1}) < 0) then return (0, 0)
  else return (Df_1(X_n), D^2 f_1)
  endif
else if (1 < i < n ∧ (Y_{i+1} - Y_i)(Y_i - Y_{i-1}) < 0) then
  The point (X_i, Y_i) is an extreme point. The quadratic with minimum magnitude curvature that
  has slope zero at X_i will be the facet chosen.
  f_1 := interpolant to (X_{i-1}, Y_{i-1}), (X_i, Y_i), and Df_1(X_i) = 0.
  f_2 := interpolant to (X_i, Y_i), (X_{i+1}, Y_{i+1}), and Df_2(X_i) = 0.
  if (|D^2 f_1| ≤ |D^2 f_2|) then return (0, D^2 f_1)
  else return (0, D^2 f_2)
  endif
else
  The point (X_i, Y_i) is in a monotone segment of data. In the following, it is possible that f_1 or f_3
  do not exist because i ∈ {2, n-1}. In those cases, the minimum magnitude curvature among
  existing quadratics is chosen.
  f_1 := interpolant to (X_{i-2}, Y_{i-2}), (X_{i-1}, Y_{i-1}), and (X_i, Y_i).
  f_2 := interpolant to (X_{i-1}, Y_{i-1}), (X_i, Y_i), and (X_{i+1}, Y_{i+1}).
  f_3 := interpolant to (X_i, Y_i), (X_{i+1}, Y_{i+1}), and (X_{i+2}, Y_{i+2}).
  if (Df_1(X_i)(Y_i - Y_{i-1}) ≥ 0 ∧ |D^2 f_1| = min{|D^2 f_1|, |D^2 f_2|, |D^2 f_3|}) then
    return (Df_1(X_i), D^2 f_1)
  else if (Df_2(X_i)(Y_i - Y_{i-1}) ≥ 0 ∧ |D^2 f_2| = min{|D^2 f_1|, |D^2 f_2|, |D^2 f_3|}) then
    return (Df_2(X_i), D^2 f_2)
  else if (Df_3(X_i)(Y_{i+1} - Y_i) ≥ 0) then
    return (Df_3(X_i), D^2 f_3)
  else return (0, 0)
  endif
endif
```

The estimated derivative values by the quadratic facet model are not guaranteed to produce monotone quintic polynomial segments. Ulrich and Watson [78] established tight constraints on the monotonicity of a quintic polynomial piece, while deferring to Heß and Schmidt [44] for a relevant simplified case. The following algorithm implements a sharp check for monotonicity by considering the nondecreasing case. The nonincreasing case is handled similarly.

Algorithm 2:
IS_MONOTONE$(x_0, x_1, f(x_0), Df(x_0), D^2 f(x_0), f(x_1), Df(x_1), D^2 f(x_1))$
where $x_0, x_1 \in \mathbb{R}$, $x_0 < x_1$, and f is an order six polynomial defined by $f(x_0)$, $Df(x_0)$, $D^2 f(x_0)$, $f(x_1)$, $Df(x_1)$, $D^2 f(x_1)$. Returns TRUE if f is monotone increasing on $[x_0, x_1]$.

1. if $(f(x_0) \approx f(x_1))$ then

2. `return` $\big(0 = Df(x_0) = Df(x_1) = D^2f(x_0) = D^2f(x_1)\big)$
3. `endif`
4. `if` $\big(Df(x_0) < 0$ `or` $Df(x_1) < 0\big)$ `then return FALSE endif`
5. $w := x_1 - x_0$
6. $z := f(x_1) - f(x_0)$

   The necessity of Steps 1–4 follows directly from the fact that f is C^2. The following Steps 7–13 coincide with a simplified condition for quintic monotonicity that reduces to one of cubic positivity studied by Schmidt and Heß [1988]. Given α, β, γ, and δ as defined by Schmidt and Heß, monotonicity results when $\alpha \geq 0$, $\delta \geq 0$, $\beta \geq \alpha - 2\sqrt{\alpha\delta}$, and $\gamma \geq \delta - 2\sqrt{\alpha\delta}$. Step 4 checked for $\delta < 0$, Step 8 checks $\alpha < 0$, Step 10 checks $\beta < \alpha - 2\sqrt{\alpha\delta}$, and Step 11 checks $\gamma < \delta - 2\sqrt{\alpha\delta}$. If none of the monotonicity conditions are violated, then the order six piece is monotone and Step 12 concludes.
7. `if` $\big(Df(x_0) \approx 0$ `or` $Df(x_1) \approx 0\big)$ `then`
8. `if` $\big(D^2f(x_1)w > 4Df(x_1)$ `then return FALSE endif`
9. $t := 2\sqrt{Df(x_0)(4Df(x_1) - D^2f(x_1)w)}$
10. `if` $\big(t + 3Df(x_0) + D^2f(x_0)w < 0\big)$ `then return FALSE endif`
11. `if` $\big(60z - w\big(24Df(x_0) + 32Df(x_1) - 2t + w(3D^2f(x_0) - 5D^2f(x_1))\big) < 0\big)$
    `then return FALSE endif`
12. `return TRUE`
13. `endif`

    The following code considers the full quintic monotonicity case studied by Ulrich and Watson [78]. Given τ_1, α, β, and γ as defined by Ulrich and Watson, a quintic piece is proven to be monotone if and only if $\tau_1 > 0$, and $\alpha, \gamma > -(\beta+2)/2$ when $\beta \leq 6$, and $\alpha, \gamma > -2\sqrt{\beta - 2}$ when $\beta > 6$. Step 14 checks $\tau_1 \leq 0$, Steps 19 and 20 determine monotonicity based on α, β, and γ.
14. `if` $\big(w\big(2\sqrt{Df(x_0)\,Df(x_1)} - 3(Df(x_0) + Df(x_1))\big) - 24z \leq 0\big)$
    `then return FALSE endif`
15. $t := \big(Df(x_0)\,Df(x_1)\big)^{3/4}$
16. $\alpha := (4Df(x_1) - D^2f(x_1)w)\sqrt{Df(x_0)}/t$
17. $\gamma := (4Df(x_0) - D^2f(x_0)w)\sqrt{Df(x_1)}/t$
18. $\beta := \dfrac{60z/w + 3\big(w(D^2f(x_1) - D^2f(x_0)) - 8(Df(x_0) + Df(x_1))\big)}{2\sqrt{Df(x_0)\,Df(x_1)}}$
19. `if` $(\beta \leq 6)$ `then return` $\big(\min\{\alpha, \gamma\} > -(\beta+2)/2\big)$
20. `else return` $\big(\min\{\alpha, \gamma\} > -2\sqrt{\beta - 2}\big)$
21. `endif`

It is shown by Ulrich and Watson [78] that when $0 = DQ(X_i) = DQ(X_{i+1}) = D^2Q(X_i) = D^2Q(X_{i+1})$, the quintic polynomial over $[X_i, X_{i+1}]$ is guaranteed to be monotone. Using this fact, the following algorithm shrinks (in magnitude) initial derivative estimates until a monotone spline is achieved and outlines the core routine in the accompanying package.

Algorithm 3: `MQSI`$\big(X(1{:}n), Y(1{:}n)\big)$
where $(X_i, Y_i) \in \mathbb{R} \times \mathbb{R}$, $i = 1, \ldots, n$ are data points. Returns monotone quintic spline interpolant $Q(x)$ such that $Q(X_i) = Y_i$ and is monotone increasing (decreasing) on all intervals that Y_i is monotone increasing (decreasing).

Approximate first and second derivatives at X_i with `QUADRATIC_FACET`.
`for` $i := 1$ `step` 1 `until` n `do`
  $(u_i, v_i) :=$ `QUADRATIC_FACET`(X, Y, i)
`enddo`
Identify and store all intervals where Q is nonmonotone in a queue $\mathscr{Q}$.
`for` $i := 1$ `step` 1 `until` $n-1$ `do`
  `if not IS_MONOTONE`$(X_i, X_{i+1}, Y_i, u_i, v_i, Y_{i+1}, u_{i+1}, v_{i+1})$ `then`
    Add interval $\big(X_i, X_{i+1}\big)$ to queue $\mathscr{Q}$.
  `endif`
`enddo`
`do while` (queue $\mathscr{Q}$ of intervals is nonempty)
  Shrink (in magnitude) DQ (in u) and D^2Q (in v) that border intervals where Q is nonmonotone.
  Identify and store remaining intervals where Q is nonmonotone in queue $\mathscr{Q}$.
`enddo`
Construct and return a B-spline representation of $Q(x)$.

Since `IS_MONOTONE` can handle both nondecreasing and nonincreasing simultaneously by taking into account the sign of z, Algorithm 3 produces $Q(x)$ that is monotone increasing (decreasing) over exactly the same intervals that the data (X_i, Y_i) is monotone increasing (decreasing).

Given the minimum magnitude curvature nature of the initial derivative estimates, it is desirable to make the smallest necessary changes to the initial interpolating spline Q while enforcing monotonicity. In practice a binary search for the boundary of monotonicity is used in place of solely shrinking DQ and D^2Q at breakpoints adjoining *active* intervals, or intervals over which Q is nonmonotone at least once during the search. The binary search considers a Boolean function $B_i(s)$, for $0 \le s \le 1$, that is true if the order six polynomial $Q(x)$ on $[X_i, X_{i+1}]$ matching derivatives $Q(X_i) = Y_i$, $DQ(X_i) = s\,u_i$, $D^2Q(X_i) = s\,v_i$ at X_i, and derivatives $Q(X_{i+1}) = Y_{i+1}$, $DQ(X_{i+1}) = s\,u_{i+1}$, $D^2Q(X_{i+1}) = s\,v_{i+1}$ at X_{i+1} is monotone, and false otherwise. As is outlined in Algorithm 3, the binary search is only applied at those breakpoints adjoining intervals $[X_i, X_{i+1}]$ over which Q is nonmonotone and hence $B_i(1)$ is false. It is further assumed that there exists $0 \le s^* \le 1$ such that $B_i(s)$ is true for $0 \le s \le s^*$ and false for some $1 > s > s^*$. Since the derivative conditions at interior breakpoints are shared by intervals left and right of the breakpoint, the binary search is performed at all breakpoints simultaneously. Specifically, the monotonicity of Q is checked on all active intervals in each step of the binary search to determine the next derivative modification at each breakpoint. The goal of this search is to converge on the boundary of the monotone region in the $(\tau_1, \alpha, \beta, \gamma)$ space (described in Ulrich and Watson [78]) for all intervals. This multiple-interval binary search allows the value zero to be obtained for all (first and second) derivative values in a

fixed number of computations, hence has no effect on computational complexity order. This binary search algorithm is outlined below.

Algorithm 4: `BINARY_SEARCH`$\big(X(1{:}n), Y(1{:}n), u(1{:}n), v(1{:}n)\big)$
where $(X_i, Y_i) \in \mathbb{R} \times \mathbb{R}$, $i = 1, \ldots, n$ are data points, and $Q(x)$ is a quintic spline interpolant such that $Q(X_i) = Y_i$, $DQ(X_i) = u_i$, $D^2Q(X_i) = v_i$. Modifies derivative values (u and v) of Q at data points to ensure `IS_MONOTONE` is true for all intervals defined by adjacent data points, given a desired precision $\mu \in \mathbb{R}$. $\text{proj}\big(w, \text{int}(a,b)\big)$ denotes the projection of w onto the closed interval with endpoints a and b.

Initialize the step size s, make a copy of data defining Q, and construct three queues necessary for the multiple-interval binary search.
$s := 1$
$(\hat{u}, \hat{v}) := (u, v)$
`searching` := `TRUE`
`checking` := empty queue for holding left indices of *intervals*
`growing` := empty queue for holding indices of *data points*
`shrinking` := empty queue for holding indices of *data points*
`for` $i := 1$ `step` 1 `until` $n-1$ `do`
  `if not IS_MONOTONE`$\big(X_i, X_{i+1}, Y_i, u_i, v_i, Y_{i+1}, u_{i+1}, v_{i+1}\big)$ `then`
    Add data indices i and $i+1$ to queue `shrinking`.
  `endif`
`enddo`
`do while` (`searching or` (`shrinking` is nonempty))
  Compute the *step size* s for this iteration of the search.
  `if searching then` $s := \max\{\mu, s/2\}$ `else` $s := 3s/2$ `endif`
  `if` $(s = \mu)$ `then searching` := `FALSE`; clear queue `growing endif`
  Increase in magnitude u_i and v_i for all data indices i in `growing` such that the points X_i are strictly adjoining intervals over which Q is monotone.
  `for` ($i \in$ `growing`) `and` ($i \notin$ `shrinking`) `do`
    $u_i := \text{proj}\big(u_i + s\,\hat{u}_i, \text{int}(0, \hat{u}_i)\big)$
    $v_i := \text{proj}\big(v_i + s\,\hat{v}_i, \text{int}(0, \hat{v}_i)\big)$
    Add data indices $i-1$ (if not 0) and i (if not n) to queue `checking`.
  `enddo`
  Decrease in magnitude u_i and v_i for all data indices i in `shrinking` and ensure those data point indices are placed into `growing` when `searching`.
  `for` $i \in$ `shrinking do`
    If `searching`, then add index i to queue `growing` if not already present.
    $u_i := \text{proj}\big(u_i - s\,\hat{u}_i, \text{int}(0, \hat{u}_i)\big)$
    $v_i := \text{proj}\big(v_i - s\,\hat{v}_i, \text{int}(0, \hat{v}_i)\big)$
    Add data indices $i-1$ (if not 0) and i (if not n) to queue `checking`.
  `enddo`
  Empty queue `shrinking`, then check all intervals left-indexed in queue `checking` for

```
    monotonicity with IS_MONOTONE, placing data endpoint indices of intervals over which Q
    is nonmonotone into queue shrinking.
    Clear queue shrinking.
    for i ∈ checking do
      if not IS_MONOTONE(X_i, X_{i+1}, Y_i, u_i, v_i, Y_{i+1}, u_{i+1}, v_{i+1}) then
        Add data indices i and i + 1 to shrinking.
      endif
    enddo
    Clear queue checking.
  enddo
```

In the subroutine MQSI, $\mu = 2^{-26}$, which results in 26 guaranteed search steps for all intervals that are initially nonmonotone. An additional 43 steps could be required to reduce a derivative magnitude to zero with step size growth rate of $3/2$. This can only happen when Q becomes nonmonotone on an interval for the first time while the step size equals μ, but for which the only viable solution is a derivative value of zero. The maximum number of steps is due to the fact that $\sum_{i=0}^{42} \mu(3/2)^i > 1$. In total BINARY_SEARCH search could require 69 steps.

7.3 Spline Representation

The monotone quintic spline interpolant $Q(x)$ is represented in terms of a B-spline basis. The routine FIT_SPLINE in this package computes the B-spline coefficients α_i of $Q(x) = \sum_{i=1}^{3n} \alpha_i B_{i,6,t}(x)$ to match the piecewise quintic polynomial values and (first two) derivatives at the breakpoints X_i, where the spline order is six and the knot sequence t has the breakpoint multiplicities (6, 3, ..., 3, 6). The routine EVAL_SPLINE evaluates a spline represented in terms of a B-spline basis. A Fortran 2003 implementation EVAL_BSPLINE of the B-spline recurrence relation evaluation code by C. deBoor [24] for the value, derivatives, and integral of a B-spline is also provided.

7.4 Complexity and Sensitivity

Algorithms 1, 2, and 4 have $\mathcal{O}(1)$ runtime. Given a fixed schedule for shrinking derivative values, Algorithm 3 has a $\mathcal{O}(n)$ runtime for n data points. In execution, the majority of the time, still $\mathcal{O}(n)$, is spent solving the banded linear system of equations for the B-spline coefficients. Thus for n data points, the overall execution time is $\mathcal{O}(n)$. The quadratic facet model produces a unique sensitivity to input perturbation, as small changes in input may cause different quadratic facets to be associated with a breakpoint, and thus different initial derivative estimates can be produced. This phenomenon is depicted in Figure 7.1. Despite this sensitivity, the quadratic facet model is still preferred because it generally provides aesthetically pleasing (low *wiggle*) initial estimates

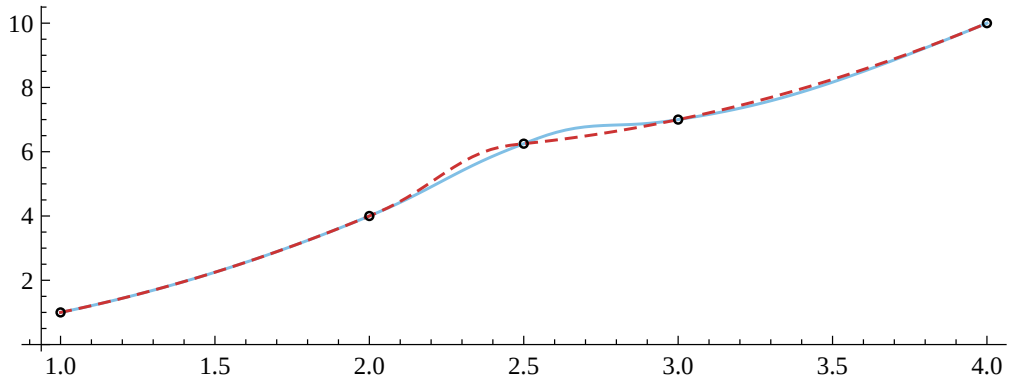

Figure 7.1: A demonstration of the quadratic facet model's sensitivity to small data perturbations. This example is composed of two quadratic functions $f_1(x) = x^2$ over points $\{1,\ 2,\ 5/2\}$, and $f_2(x) = (x-2)^2 + 6$ over points $\{5/2,\ 3,\ 4\}$. Notably, $f_1(5/2) = f_2(5/2)$ and f_1, f_2 have the same curvature. Given the exact five data points seen above, the quadratic facet model produces the slope seen in the solid blue line at $x = 5/2$. However, by subtracting the value of $f_3 = \varepsilon(x-2)^2$ from points at $x = 3,\ 4$, where ε is the machine precision (2^{-52} for an IEEE 64-bit real), the quadratic facet model produces the slope seen in the dashed red line at $x = 5/2$. This is the nature of a facet model and a side effect of associating data with local facets.

of the first and second derivatives at the breakpoints while perfectly capturing local low-order phenomena (e.g., a run of points on a straight line) in the data. The binary search for a point near the monotone boundary in $(\tau_1,\ \alpha,\ \beta,\ \gamma)$ space is preferred because it results in monotone quintic spline interpolants that are nearer to initial estimates than a policy that strictly shrinks derivative values.

7.5 Performance and Applications

This section contains graphs of sample `MQSI` results given various data configurations. Computation times for various problem sizes are also provided. The files accompanying the subroutine `MQSI` offer multiple usages, namely `sample_main.f90` that demonstrates Fortran 2003 usage and a command line interface `cli.f90` that produces `MQSI` estimates for points in batches from data files. Compilation instructions and the full package contents are specified in the `README` file.

Throughout, all visuals have points that are stylized by local monotonicity conditions. Blue circles denote extreme points, purple squares are in *flat* regions with no change in function value, red down triangles are monotone decreasing, and green up triangles are monotone increasing.

Figure 7.2 offers examples of the interpolating splines produced by the routine `MQSI` on various hand-crafted sets of data. These same data sets are used for testing local installations in the provided program `test_all.f90`. Notice that the quadratic facet model perfectly captures the local linear segments of data in the piecewise polynomial test for Figure 7.2. Figure 7.3 depicts an

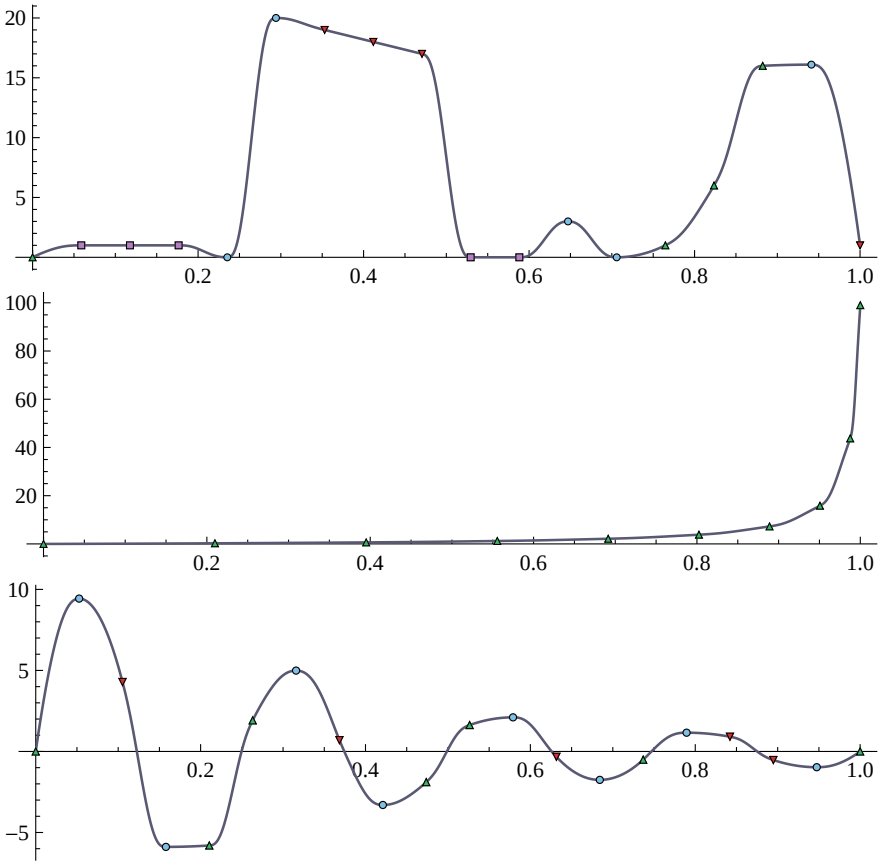

Figure 7.2: `MQSI` results for three of the functions in the included test suite. The *piecewise polynomial* function (top) shows the interpolant capturing local linear segments, local flats, and alternating extreme points. The *large tangent* (middle) problem demonstrates outcomes on rapidly changing segments of data. The *signal decay* (bottom) alternates between extreme values of steadily decreasing magnitude.

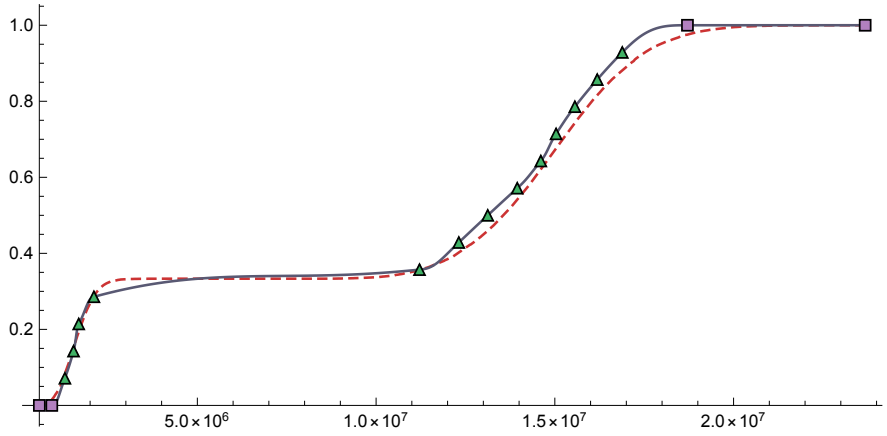

Figure 7.3: `MQSI` results when approximating the cumulative distribution function of system throughput (bytes per second) data for a computer with a 3.2 GHz CPU performing file read operations from Cameron et al. [13]. The empirical distribution of 30 thousand throughput values is shown in the red dashed line, while the solid line with stylized markers denotes the approximation made with MQSI given equally spaced empirical distribution points from a sample of size 100.

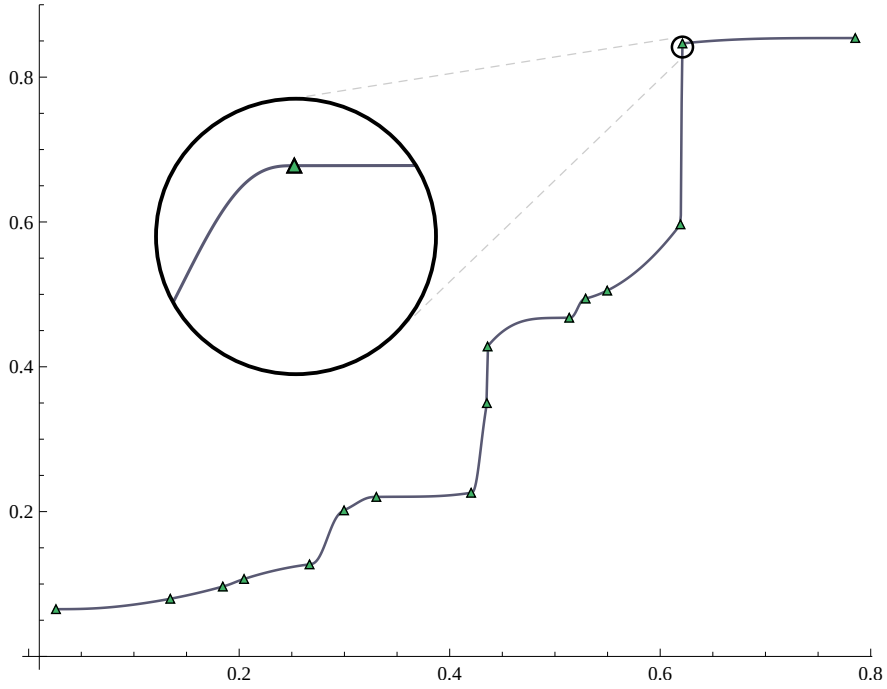

Figure 7.4: The *random monotone* test poses a particularly challenging problem with large variations in slope. Notice that despite drastic shifts in slope, the resulting monotone quintic spline interpolant provides smooth and reasonable estimates to function values between data.

approximation of a cumulative distribution function made by `MQSI` on a computer systems application by Cameron et al. [13] that studies the distribution of throughput (in bytes per second) when reading files from a storage medium. Figure 7.4 provides a particularly difficult monotone interpolation challenge using randomly generated monotone data.

On a computer running MacOS 10.15.5 with a 2 GHz Intel Core i5 CPU, the quadratic facet (Algorithm 1) takes roughly one microsecond (10^{-6} seconds) per breakpoint, while the binary search (Algorithm 4) takes roughly four microseconds per breakpoint; these times were generated from 100 repeated trials averaged over 14 different testing functions. The vast majority of execution time is spent solving the banded linear system of equations in the routine `FIT_SPLINE` for the B-spline coefficients. For large problems ($n > 100$) it would be faster to construct splines over intervals independently (each interval requiring a 6×6 linear system to be solved), however the single linear system is chosen here for the decreased redundancy in the spline description.

Chapter 8

Conclusions on Modeling and Approximating Variability in Computer Systems

This body of work reveals many interesting findings related to modeling variability in computer systems and approximation in general. While generic "off the shelf" techniques from machine learning are immediately capable of being applied to the prediction of variability summary statistics, a little extra effort can lend significant improvements in the accuracy and descriptiveness of the predictions being made. For example, interpolants like Delaunay yield lower error predictions with less data (improved accuracy), while also trivially extending to the prediction of full distributions (improved precision). The major takeaways from this work are threefold:

1. computer performance data is often *not* normally distributed and accurate descriptions of performance should rely on nonparametric methods or mixtures of parametric distributions;
2. interpolants such as Delaunay and Shepard methods methods tend to perform better than regressors when used to predict variability-related performance measures, especially when the amount of available data is limited; and
3. in medium to high dimension regressors and interpolants produce equally accurate approximations on analytic test functions, but interpolants favor applications where data is either limited or rapidly changing (i.e., real time systems), because they do not have the *fit* time required by regressors.

Acknowledging these takeaways, there is some room to speculate on the most promising directions for future research related to variability approximation. Specifically, consider systems that need to operate in real time, making approximations in less than a tenth of a millisecond with dynamically updating data. For problems in less than tens of dimensions and with at most thousands of points, it appears best to use Delaunay to make predictions of either means, variances, or full distributions. When the number of points or dimension goes up to tens / hundreds of dimensions with tens of thousands of points, Shepard methods remain particularly well suited to making variability predictions. Finally if there are hundreds / thousands or more dimensions and tens to hundreds of thousands of data points, then a fast k nearest neighbor lookup mixed with a Shepard method could give good results. Notably, all of these approximation methods are interpolants that (given

nondegenerate data) require no concept of a *fit*. In addition, all of these interpolants are readily capable of predicting distribution outcomes.

One of the most popular approximation methods in the research community at present is neural networks. While they show incredible promise in computer vision, reinforcement learning, and stream (text sequence) processing, they have failed to maintain superior performance when applied specifically to computer performance modeling. However, the most interesting line of research that ties together interpolants, real time systems, and neural networks is the application of a neural network as a preconditioner for interpolation methods. For instance, train a neural network on computer system data offline, then remove the final layer and instead use the transformed representation of data in the last internal layer as a new vector representation for standard interpolation. Fundamentally this takes advantage of the best components from each method, the structure processing / compositional nature of neural networks, and the ability to quickly make real time predictions from recently generated (or otherwise rapidly changing) runtime data.

Lastly, the potential improvements in distribution approximation provided by the `MQSI` software package are significant. The ability to capture second derivative information in a piecewise monotone spline interpolant can greatly reduce the amount of data required to estimate a distribution from a sample. Extensions of this work should consider statistical importance sampling, smoothing, and other methods to provide robust estimates of points on the distribution functions that can be used in conjunction with `MQSI` to provide the best possible distribution estimates.

This work provides a deep dive into the best methodologies for approximating variability in computer systems. By comparing known approximation techniques, proposing new approximations, extending variability modeling from real-valued (parametric) prediction to distribution-valued (nonparametric) prediction, proving a novel error bound for piecewise linear interpolants, and writing a mathematical software package for constructing monotone quintic spline interpolants, this work has significantly contributed to the study of computer system variability and approximation by interpolants. The body of this work was possible largely due to funding and support from the National Science Foundation, and support and guidance from the faculty and staff of Virginia Polytechnic Institute and State University.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos. Security in cloud computing: Opportunities and challenges. *Information Sciences*, 305:357–383, 2015.

[3] Brandon D. Amos, David R. Easterling, Layne T. Watson, William I. Thacker, Brent S. Castle, and Michael W. Trosset. Algorithm XXX: QNSTOP: quasi-Newton algorithm for stochastic optimization. *Technical Report 14-02, Dept. of Computer Science, VPI&SU, Blacksburg, VA*, 2014.

[4] David H. Bailey and Allan Snavely. Performance modeling: Understanding the past and predicting the future. In *European Conference on Parallel Processing*, pages 185–195. Springer, 2005.

[5] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and José Carlos Sancho. Using performance modeling to design large-scale systems. *Computer*, 42(11), 2009.

[6] Volker Barthelmann, Erich Novak, and Klaus Ritter. High dimensional polynomial interpolation on sparse grids. *Advances in Computational Mathematics*, 12(4):273–288, 2000.

[7] Debasish Basak, Srimanta Pal, and Dipak Chandra Patranabis. Support vector regression. *Neural Information Processing-Letters and Reviews*, 11(10):203–224, 2007.

[8] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[9] Yoshua Bengio and Yves Grandvalet. No unbiased estimator of the variance of k-fold cross-validation. *Journal of machine learning research*, 5(Sep):1089–1105, 2004.

[10] Tomas Berglund, Andrej Brodnik, Håkan Jonsson, Mats Staffanson, and Inge Soderkvist. Planning smooth and obstacle-avoiding B-spline paths for autonomous mining vehicles. *IEEE Transactions on Automation Science and Engineering*, 7(1):167–172, 2009.

[11] Len Bos, Stefano De Marchi, Alvise Sommariva, and Marco Vianello. Computing multivariate fekete and leja points by numerical linear algebra. *SIAM Journal on Numerical Analysis*, 48(5):1984–1999, 2010.

[12] AnnMarie Brennan. Measure, modulation and metadesign: NC fabrication in industrial design and architecture. *Journal of Design History*, 11 2020. ISSN 0952-4649. doi: 10.1093/jdh/epz042.

[13] Kirk W. Cameron, Ali Anwar, Yue Cheng, Li Xu, Bo Li, Uday Ananth, Jon Bernard, Chandler Jearls, Thomas C. H. Lux, Yili Hong, Layne T. Watson, and Ali R. Butt. Moana: Modeling and analyzing i/o variability in parallel system experimental design. *IEEE Transactions on Parallel and Distributed Systems*, 2019. ISSN 1045-9219. doi: 10.1109/TPDS.2019.2892129.

[14] Tyler H. Chang, Layne T. Watson, Thomas C. H. Lux, Bo Li, Li Xu, Ali R. Butt, Kirk W. Cameron, and Yili Hong. A polynomial time algorithm for multivariate interpolation in arbitrary dimension via the Delaunay triangulation. In *Proceedings of the ACMSE 2018 Conference*, ACMSE '18, pages 12:1–12:8, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5696-1. doi: 10.1145/3190645.3190680.

[15] Elliott Ward Cheney and William Allan Light. *A Course in Approximation Theory*, volume 101. American Mathematical Soc., 2009.

[16] François Chollet. Keras, 2015.

[17] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[18] Paulo Cortez and Aníbal de Jesus Raimundo Morais. A data mining approach to predict forest fires using meteorological data. *13th Portuguese Conference on Artificial Intelligence*, 2007.

[19] Paulo Cortez and Alice Maria Gonçalves Silva. Using data mining to predict secondary school student performance. *Proceedings of 5th Annual Future Business Technology Conference, Porto*, 2008.

[20] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[21] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013*, pages 8609–8613. IEEE, 2013.

[22] Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson, and Gianluca Bontempi. Calibrating probability with undersampling for unbalanced classification. In *IEEE Symposium Series on Computational Intelligence*, pages 159–166. IEEE, Dec 2015. doi: 10.1109/SSCI.2015.33. [Online; accessed 2019-01-25].

[23] Pradipta De, Ravi Kothari, and Vijay Mann. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *IEEE International Conference on Cluster Computing*, pages 331–340. IEEE, 2007.

[24] Carl de Boor. *A Practical Guide to Splines*, volume 27. Springer-Verlag New York, 1978.

[25] Carl de Boor, Klaus Höllig, and Sherman Riemenschneider. *Box splines*, volume 98. Springer Science & Business Media, 2013.

[26] Saverio De Vito, Ettore Massera, Marco Piga, Luca Martinotto, and Girolamo Di Francia. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. *Sensors and Actuators B: Chemical*, 129(2):750–757, 2008.

[27] John E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16. Siam, 1996.

[28] G. Lejeune Dirichlet. Über die reduction der positiven quadratischen formen mit drei unbestimmten ganzen zahlen. *Journal für die Reine und Angewandte Mathematik*, 40:209–227, 1850.

[29] Mathieu Dutour Sikirić, Achill Schürmann, and Frank Vallentin. Complexity and algorithms for computing Voronoi cells of lattices. *Mathematics of Computation*, 78(267):1713–1731, 2009.

[30] Francesc J. Ferri, Pavel Pudil, Mohamad Hatef, and Josef Kittler. Comparative study of techniques for large-scale feature selection. *Pattern Recognition in Practice IV*, 1994:403–413, 1994.

[31] Jerome H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, pages 1–67, 1991.

[32] Jerome H. Friedman and the Computational Statistics Laboritory of Stanford University. Fast MARS. Technical report, 1993.

[33] Frederick N. Fritsch. Piecewise cubic hermite interpolation package. final specifications. Technical report, Lawrence Livermore National Lab., CA (USA), 1982.

[34] Frederick N. Fritsch and Ralph E. Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246, 1980. doi: 10.1137/0717021.

[35] Gabriel Goh. Why momentum really works. *Distill*, 2017. doi: 10.23915/distill.00006.

[36] William J. Gordon and James A. Wixom. Shepard's method of "metric interpolation" to bivariate and multivariate interpolation. *Mathematics of Computation*, 32(141):253–264, 1978.

[37] John A. Gregory. Shape preserving spline interpolation. (19850020252), 1985.

[38] Gunther Greiner and Kai Hormann. Interpolating and approximating scattered 3D-data with hierarchical tensor product B-splines. In *Proceedings of Chamonix*, page 1, 1996.

[39] Eric Grobelny, David Bueno, Ian Troxel, Alan D. George, and Jeffrey S. Vetter. FASE: A framework for scalable performance prediction of HPC systems and applications. *Simulation*, 83(10):721–745, 2007.

[40] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3(Mar):1157–1182, 2003.

[41] Robert M. Haralick and Layne Watson. A facet model for image data. *Computer Graphics and Image Processing*, 15(2):113–129, 1981.

[42] Xuming He and Peide Shi. Monotone b-spline smoothing. *Journal of the American Statistical Association*, 93(442):643–650, 1998.

[43] Daniel Herman and Mark Oftedal. Techniques and workflows for computer graphics animation system, 2006. US Patent App. 11/406,050.

[44] Walter Heß and Jochen W. Schmidt. Positive quartic, monotone quintic C^2-spline interpolation in one and two dimensions. *Journal of Computational and Applied Mathematics*, 55(1): 51–67, 1994. doi: 10.1016/0377-0427(94)90184-8.

[45] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[46] Gary D. Knott. *Interpolating Cubic Splines*, volume 18. Springer Science & Business Media, 2012.

[47] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14(2), pages 1137–1145. Montreal, Canada, 1995.

[48] Thomas Kövari and Christian Pommerenke. On the distribution of fekete points. *Mathematika*, 15(1):70–75, 1968.

[49] Dimitris Lazos, Alistair B. Sproul, and Merlinde Kay. Optimisation of energy management in commercial buildings with weather forecasting inputs: A review. *Renewable and Sustainable Energy Reviews*, 39:587–603, 2014.

[50] Der-Tsai Lee and Bruce J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[51] Florian Leitenstorfer and Gerhard Tutz. Generalized monotonic regression based on b-splines with an application to air pollution data. *Biostatistics*, 8(3):654–673, 2007.

[52] Hubert W. Lilliefors. On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318):399–402, 1967.

[53] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the IO performance of petascale storage systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2010*, pages 1–12. IEEE, 2010.

[54] Thomas C. H. Lux, Randall Pittman, Maya Shende, and Anil Shende. Applications of supervised learning techniques on undergraduate admissions data. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 412–417. ACM, 2016.

[55] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Jon Bernard, Bo Li, Li Xu, Godmar Back, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Predictive modeling of i/o characteristics in high performance computing systems. In *Proceedings of the High Performance Computing Symposium*, HPC '18, San Diego, CA, USA, 2018. Society for Computer Simulation International. ISBN 9781510860162.

[56] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Jon Bernard, Bo Li, Xiaodong Yu, Li Xu, Godmar Back, Ali R Butt, Kirk W. Cameron, and Yili Hong. Nonparametric distribution models for predicting and managing computational performance variability. In *SoutheastCon 2018*, pages 1–7. IEEE, 2018.

[57] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Jon Bernard, Bo Li, Xiaodong Yu, Li Xu, Godmar Back, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Novel meshes for multivariate interpolation and approximation. In *Proceedings of the ACMSE 2018 Conference*, page 13. ACM, 2018.

[58] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Yili Hong, and Kirk W. Cameron. Interpolation of sparse high-dimensional data. *Numerical Algorithms*, 08 2020. Submitted April, 2018.

[59] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Li Xu, Yueyao Wang, Jon Bernard, Yili Hong, and Kirk W Cameron. Effective nonparametric distribution modeling for distribution approximation applications. In *SoutheastCon 2020*, pages 1–7. IEEE, 2020.

[60] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Li Xu, Yueyao Wang, and Yili Hong. An algorithm for constructing monotone quintic interpolating splines. In *Proceedings of the High Performance Computing Symposium*. Spring Simulation Multiconference, May 2020.

[61] Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

[62] William Cyrus. Navidi. *Statistics for Engineers and Scientists*. McGraw-Hill Education, 4 edition, 2015.

[63] William D. Norcott. IOzone filesystem benchmark, 2017.

[64] Jeong-Soo Park. Optimal latin-hypercube designs for computer experiments. *Journal of statistical planning and inference*, 39(1):95–111, 1994.

[65] Pankesh Patel, Ajith H. Ranabahu, and Amit P. Sheth. Service level agreement in cloud computing. *Wright State University CORE Scholar Repository*, 2009.

[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[67] Abd Rahni Mt Piah and Keith Unsworth. Improved sufficient conditions for monotonic piecewise rational quartic interpolation. *Sains Malaysiana*, 40(10):1173–1178, 2011.

[68] Pavel Pudil, Jana Novovičová, and Josef Kittler. Floating search methods in feature selection. *Pattern Recognition Letters*, 15(11):1119–1125, 1994.

[69] Antoine Quint. Scalable vector graphics. *IEEE MultiMedia*, 10(3):99–102, 2003.

[70] James O. Ramsay. Monotone regression splines in action. *Statistical Science*, 3(4):425–441, 1988.

[71] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[72] Jason Rudy and Mehdi Cherti. Py-Earth: A python implementation of multivariate adaptive regression splines, 2017.

[73] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[74] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524. ACM, 1968.

[75] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE Conference*, pages 21–21. IEEE, 2002.

[76] William I. Thacker, Jingwei Zhang, Layne T. Watson, Jeffrey B. Birch, Manjula A. Iyer, and Michael W. Berry. Algorithm 905: SHEPPACK: Modified shepard algorithm for interpolation of scattered multivariate data. *ACM Transactions on Mathematical Software (TOMS)*, 37 (3):34, 2010.

[77] Athanasios Tsanas, Max A. Little, Patrick E. McSharry, and Lorraine O. Ramig. Accurate telemonitoring of parkinson's disease progression by noninvasive speech tests. *IEEE Transactions on Biomedical Engineering*, 57(4):884–893, 2010.

[78] Gary Ulrich and Layne T. Watson. Positivity conditions for quartic polynomials. *SIAM Journal on Scientific Computing*, 15(3):528–544, 1994. doi: 10.1137/0915035.

[79] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09)*, pages 1–11. IEEE, 2009.

[80] Guanying Wang, Aleksandr Khasymski, K. R. Krish, and Ali R. Butt. Towards improving mapreduce task scheduling using online simulation based predictions. In *International Conference on Parallel and Distributed Systems (ICPADS), 2013*, pages 299–306. IEEE, 2013.

[81] Qiang Wang and Jieqing Tan. Rational quartic spline involving shape parameters. *Journal of Information and Computational Science*, 1(1):127–130, 2004.

[82] Graham J. Williams. Weather dataset rattle package. In *Rattle: A Data Mining GUI for R*, volume 1, pages 45–55. The R Journal, 2009. [Online; accessed 2019-01-25].

[83] Jin Yao and Karl E Nelson. An unconditionally monotone C^2 quartic spline method with nonoscillation derivatives. *Advances in Pure Mathematics*, 8(1):25–40, 2018.

[84] Kejiang Ye, Xiaohong Jiang, Siding Chen, Dawei Huang, and Bei Wang. Analyzing and modeling the performance in xen-based virtual cluster environment. In *12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 273–280. IEEE, 2010.

Appendices

Appendix A

Error Bound Empirical Test Results

Statistical Terminology. A random variable X is precisely defined by its *cumulative distribution function* (CDF) F_X and the derivative of the CDF, the *probability density function* (PDF) f_X. For any possible value x of X, the *percentile* of x is 100 $F_X(x)$, the percentage of values drawn from X that would be less than or equal to x as the number of samples tends towards infinity. The *quartiles* of X are its 25-th, 50-th (median), and 75-th percentiles. The absolute difference between the median and an adjacent quartile is an *interquartile range*. Given an independent and identically distributed sample from X and presuming that X has finite mean and variance, a *confidence interval* can be drawn about any percentile estimated from the sample. A *confidence interval* describes the probability that a value lies within an interval. The *null hypothesis* is a statement (derived from some test statistic) that the expected value of the observed statistic is equal to an assumed population statistic. The p value is the probability of observing a given statistic if the *null hypothesis* is true. For a more detailed introduction to statistics and related terminology, see the work of Navidi [62].

Raw Numerical Results. The tables that follow show the precise experimental results for all data sets presented in Section 5.2. The tests were all run serially on an otherwise idle machine with a CentOS 6.10 operating system and an Intel i7-3770 CPU operating at 3.4 GHz. The detailed performance results in the tables that follow are very much dependent on the problem and the algorithm implementation (e.g., some codes are TOMS software, some industry distributions, and others are from conference paper venues). Different typeface is used to show best performers, however not much significance should be attached to ranking algorithms based on small time (millisecond) differences. The results serve as a demonstration of conceptual validity.

Algorithm	Min	25^{th}	50^{th}	75^{th}	Max
MARS	0.00984	3.11	7.01	*11.7*	1090.0
SVR	0.0118	**0.615**	**0.931**	**5.89**	1090.0
MLP	0.0426	2.63	5.27	14.0	1090.0
Delaunay	**0.0**	1.98	5.37	13.1	*1080.0*
ShepMod	**0.0**	1.93	6.27	16.0	1090.0
LSHEP	0.04	2.17	8.87	19.1	**1070.0**
Voronoi	*0.00982*	3.65	7.56	15.6	1090.0
BoxSpline	**0.0**	*1.27*	*4.61*	12.6	1090.0

Table A.1: This numerical data accompanies the visual provided in Figure 6.6. The columns of absolute error percentiles correspond to the minimum, first quartile, median, third quartile, and maximum absolute errors respectively. The minimum of each column is boldface, while the second lowest value is italicized. All values are rounded to three significant digits.

Algorithm	% Best
MARS	7.3
SVR	**78.0**
MLP	0.0
Delaunay	0.2
ShepMod	2.0
LSHEP	5.1
Voronoi	0.0
BoxSpline	*9.7*

Fit/Prep. Time (s)	App. Time (s)	Total App. Time (s)
29.1	0.00137	0.0686
0.00584	*0.0000620*	*0.00310*
32.8	0.000871	0.0436
0.0151	0.0234	1.18
0.00634	0.0000644	0.00322
0.0275	**0.0000463**	**0.00231**
0.0152	0.000396	0.0198
0.00724	0.0000978	0.00489

Table A.2: The left above shows how often each algorithm had the lowest absolute error approximating forest fire data in Table A.1. On the right columns are median fit time of 454 points, median time for one approximation, and median time approximating 50 points.

Algorithm	Min	25^{th}	50^{th}	75^{th}	Max
MARS	0.00948	9.98	20.4	32.9	863.0
SVR	0.00233	3.15	7.21	11.3	**28.6**
MLP	0.0000239	*0.533*	*1.25*	**2.84**	39.3
Delaunay	3.72×10^{-12}	1.2	3.5	7.67	30.7
ShepMod	**0.0**	**0.255**	**0.908**	*3.43*	34.5
LSHEP	0.00254	2.93	7.16	13.1	*29.0*
Voronoi	**0.0**	1.29	3.52	6.87	30.1
BoxSpline	0.006	4.3	8.91	15.1	45.3

Table A.3: This numerical data accompanies the visual provided in Figure 6.8. The columns of absolute error percentiles correspond to the minimum, first quartile, median, third quartile, and maximum absolute errors respectively. The minimum of each column is boldface, while the second lowest value is italicized. All values are rounded to three significant digits.

Algorithm	% Best
MARS	0.0
SVR	0.1
MLP	*32.0*
Delaunay	0.0
ShepMod	**66.4**
LSHEP	0.0
Voronoi	1.6
BoxSpline	0.0

Fit/Prep. Time (s)	App. Time (s)	Total App. Time (s)
37.9	0.00253	1.48
0.859	*0.000181*	*0.106*
348.0	0.00111	0.653
2.47	1.22	720.0
1.13	0.000182	0.107
2.39	**0.000144**	**0.0845**
2.77	0.0274	16.1
1.26	0.000643	0.377

Table A.4: The left above shows how often each algorithm had the lowest absolute error approximating Parkinson's data in Table A.3. On the right columns are median fit time of 5288 points, median time for one approximation, and median time approximating 587 points.

Algorithm	Min	25^{th}	50^{th}	75^{th}	Max
MARS	6.45×10^{-15}	$\mathbf{2.70 \times 10^{-14}}$	1.66	1.96	*53.3*
SVR	0.0000915	0.0833	0.263	*1.13*	109.0
MLP	0.0000689	0.0337	**0.0795**	**0.264**	**5.31**
Delaunay	**0.0**	0.187	0.919	2.72	56.3
ShepMod	**0.0**	0.0957	0.685	2.9	73.2
LSHEP	**0.0**	*0.0153*	*0.106*	1.17	113.0
Voronoi	**0.0**	0.43	1.28	2.94	83.8
BoxSpline	**0.0**	0.342	1.59	5.53	119.0

Table A.5: This numerical data accompanies the visual provided in Figure 6.10. The columns of absolute error percentiles correspond to the minimum, first quartile, median, third quartile, and maximum absolute errors respectively. The minimum value of each column is boldface, while the second lowest is italicized. All values are rounded to three significant digits.

Algorithm	% Best
MARS	10.7
SVR	0.0
MLP	**60.9**
Delaunay	0.1
ShepMod	3.5
LSHEP	*28.4*
Voronoi	0.2
BoxSpline	4.1

Fit/Prep. Time (s)	App. Time (s)	Total App. Time (s)
0.151	0.000117	0.0304
0.133	**0.0000947**	**0.0246**
169.0	0.00137	0.356
0.664	0.886	230.0
0.265	0.000128	0.0333
0.874	*0.0000975*	*0.0254*
0.675	0.0270	7.01
0.330	0.000406	0.106

Table A.6: Left table shows how often each algorithm had the lowest absolute error approximating Sydney rainfall data in Table A.5. On the right columns are median fit time of 2349 points, median time for one approximation, and median time approximating 260 points.

Algorithm	Min	25^{th}	50^{th}	75^{th}	Max
MARS	5.36	3610.0	4580.0	5450.0	13400.0
SVR	0.00706	8.14	*13.0*	41.6	7690.0
MLP	0.00151	**1.6**	**3.86**	**8.34**	**604.0**
Delaunay	**0.0**	*2.69*	13.8	*35.0*	*4840.0*
ShepMod	**0.0**	4.21	17.3	51.6	6510.0
LSHEP	1.27	199.0	260.0	343.0	6530.0
Voronoi	2.89×10^{-10}	14.7	32.8	52.9	4860.0
BoxSpline	**0.0**	12.4	35.0	90.1	7690.0

Table A.7: This numerical data accompanies the visual provided in Figure 6.12. The columns of absolute error percentiles correspond to the minimum, first quartile, median, third quartile, and maximum absolute errors respectively. The minimum value of each column is boldface, while the second lowest is italicized. All values are rounded to three significant digits.

Algorithm	% Best
MARS	0.0
SVR	0.0
MLP	**79.5**
Delaunay	*20.5*
ShepMod	0.1
LSHEP	0.0
Voronoi	0.0
BoxSpline	0.0

Fit/Prep. Time (s)	App. Time (s)	Total App. Time (s)
22.0	0.00148	0.820
1.01	*0.000210*	*0.117*
290.0	0.000714	0.397
3.12	3.71	2070.0
1.45	0.000220	0.122
3.47	**0.000176**	**0.0981**
3.32	0.0950	52.8
1.66	0.000956	0.532

Table A.8: The left above shows how often each algorithm had the lowest absolute error approximating credit card transaction data in Table A.7. On the right columns are median fit time of 5006 points, median time for one approximation, and median time approximating 556 points.

Algorithm	Min	25^{th}	50^{th}	75^{th}	Max
Delaunay	**0.0287**	**0.0914**	**0.158**	*0.308*	*0.897*
ShepMod	0.0307	*0.0983*	*0.164*	0.335	1.0
Voronoi	*0.0303*	0.1	0.165	**0.274**	**0.762**
BoxSpline	0.0679	0.429	0.571	0.752	1.0

Table A.9: This numerical data accompanies the visual provided in Figure 6.14. The columns of absolute error percentiles correspond to the minimum, first quartile, median, third quartile, and maximum KS statistics respectively between truth and guess for models predicting the distribution of I/O throughput that will be observed at previously unseen system configurations. The minimum value of each column is boldface, while the second lowest is italicized. All values are rounded to three significant digits.

Algorithm	% Best
Delaunay	**62.0**
ShepMod	0.0
Voronoi	*38.0*
BoxSpline	0.0

Fit/Prep. Time (s)	App. Time (s)	Total App. Time (s)
0.344	0.00984	2.71
0.0884	**0.000145**	**0.0436**
0.360	0.00253	0.762
0.0972	*0.000210*	*0.0633*

Table A.10: The left above shows how often each algorithm had the lowest KS statistic on the I/O throughput distribution data in Table A.9. On the right columns are median fit time of 2715 points, median time for one approximation, and median time approximating 301 points.

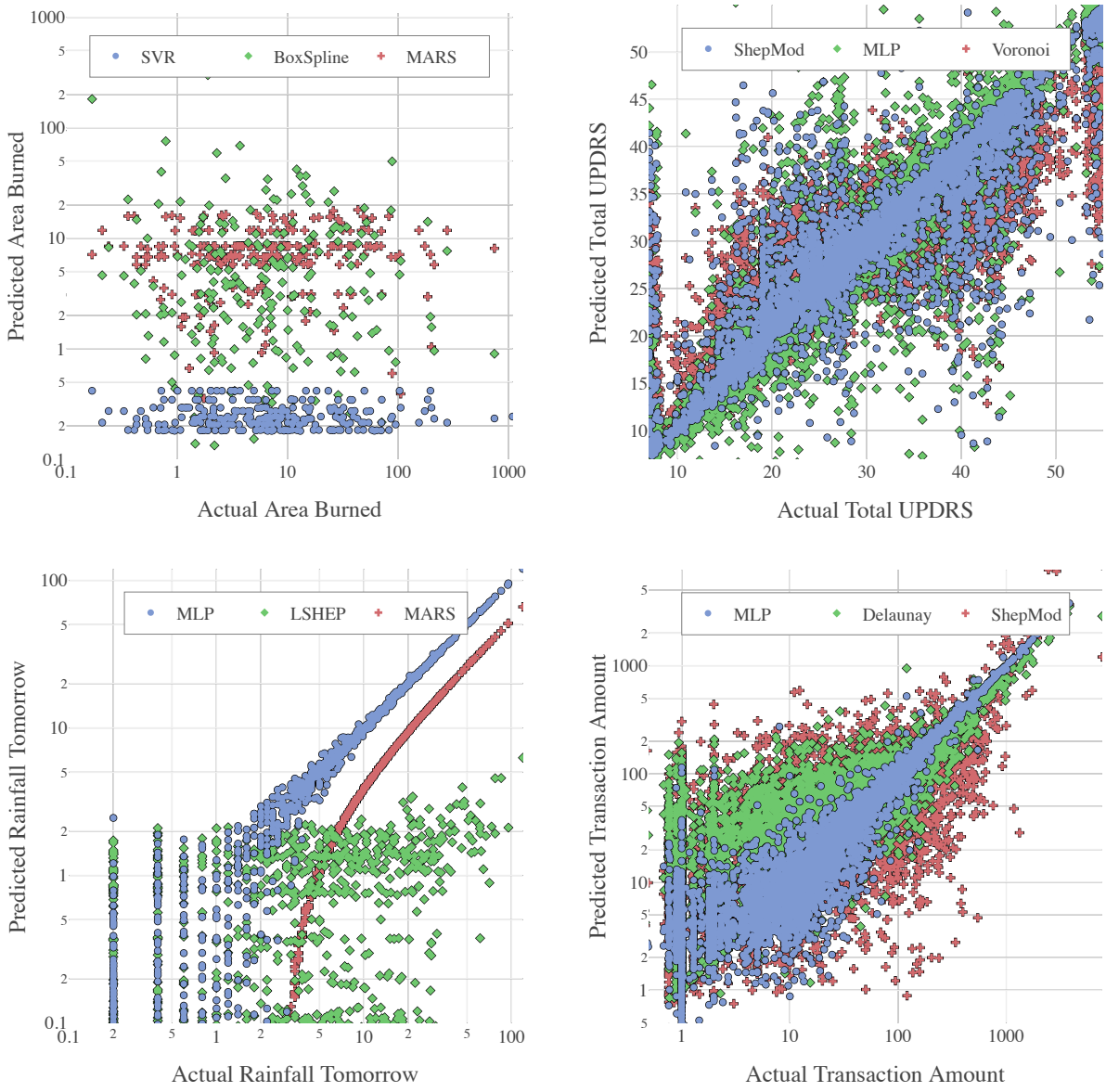

Figure A.1: Scatter plots for predicted versus actual values for the top three models on each of the four real valued approximation problems. Top left is forest fire data, top right is Parkinson's data, bottom left is rainfall data, and bottom right is credit card transaction data. The blue circles correspond to the best model, green diamonds to the second best, and red crosses to the third best model for each data set. There are a large of number of 0-valued entries in the forest fire and rainfall data sets that are not included in the visuals making the true ranking of the models appear to disagree with the observed outcomes.

Appendix B

MQSI Package

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                              README

          MQSI: Monotone Quintic Spline Interpolation

The package MQSI is a collection of Fortran 2003 routines for the
construction of a C^2 monotone quintic spline interpolant Q(X) to data
(X(I),F(X(I)), with the property that Q(X) is monotone increasing
(decreasing) over the same intervals [X(I), X(J)] as the data F(X(.))
is monotone increasing (decreasing). All routines use the module
REAL_PRECISION from HOMPACK90 (ACM TOMS Algorithm 777) for specifying
the real data type.

The physical organization into files is as follows.

 * The file 'REAL_PRECISION.f90' contains the module  REAL_PRECISION.

 * The file 'MQSI.f90' contains the subroutine MQSI, which computes a
   C^2 monotone quintic spline interpolant to data in terms of a
   B-spline basis, matching the monotonicity of the data.

 * The file 'SPLINE.f90' contains the subroutines FIT_SPLINE for
   constructing a B-spline basis representation of an osculatory
   spline Q(X) interpolating F(X), F'(X), ..., F^{(NCC-1)}(X) for NCC
   continuity conditions at each breakpoint X(K), and EVAL_SPLINE
   for computing the value, derivatives, and integral of Q(X).

 * The file 'EVAL_BSPLINE.f90' contains the subroutine EVAL_BSPLINE, a
   Fortran 2003 implementation of the B-spline recurrence relation
   evaluation code by C. deBoor for the value, derivatives, and
   integral of a B-spline.

 * The files 'blas.f' and 'lapack.f' contain just the necessary BLAS
   and LAPACK routines, in case the complete BLAS and LAPACK libraries
   are not available.
```

```
To compile and link, use the command:

 $F03 $OPTS REAL_PRECISION.f90 EVAL_BSPLINE.f90 SPLINE.f90 MQSI.f90 \
    $MAIN -o $EXEC_NAME $LIB

where '$F03' is the name of the Fortran 2003 compiler, '$OPTS' are
compiler options such as '-O3', '$MAIN' is the name of the main
program, '$EXEC_NAME' is the name of the executable, and '$LIB'
provides a flag to link BLAS and LAPACK. If the BLAS and LAPACK
libraries are not available on your system, then replace $LIB with the
filenames 'blas.f lapack.f'; these files contain the routines from the
BLAS and LAPACK libraries that are necessary for this package.

This package includes a sample main program in the file 'sample_main.f90'
illustrating the use of an input data file 'sample_main.dat', and an
extensive test suite in the file 'test_all.f90', all of which can be
compiled and linked as above.

-----------------------------------------------------------------------

Inquiries should be directed to Thomas C. H. Lux (tchlux@vt.edu).
```

```
! ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!                         REAL_PRECISION.f90
!
MODULE REAL_PRECISION  ! module for 64-bit real arithmetic
  INTEGER, PARAMETER:: R8=SELECTED_REAL_KIND(13)
END MODULE REAL_PRECISION
```

```
! ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!                          MQSI.f90
!
! DESCRIPTION:
!   This file defines a subroutine MQSI for constructing a monotone
!   quintic spline interpolant of data in terms of its B-spline basis.
!
! CONTAINS:
!   SUBROUTINE MQSI(X, Y, T, BCOEF, INFO, UV)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: X
!     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: Y
!     REAL(KIND=R8), INTENT(OUT),   DIMENSION(:) :: T, BCOEF
!     INTEGER, INTENT(OUT) :: INFO
!     REAL(KIND=R8), INTENT(OUT), DIMENSION(:,:), OPTIONAL :: UV
!   END SUBROUTINE MQSI
!
```

```
! DEPENDENCIES:
!   MODULE REAL_PRECISION
!     INTEGER, PARAMETER :: R8
!   END MODULE REAL_PRECISION
!
!   SUBROUTINE FIT_SPLINE(XI, FX, T, BCOEF, INFO)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),  DIMENSION(:)   :: XI
!     REAL(KIND=R8), INTENT(IN),  DIMENSION(:,:) :: FX
!     REAL(KIND=R8), INTENT(OUT), DIMENSION(:)   :: T, BCOEF
!     INTEGER, INTENT(OUT) :: INFO
!   END SUBROUTINE FIT_SPLINE
!
! CONTRIBUTORS:
!   Thomas C.H. Lux (tchlux@vt.edu)
!   Layne T. Watson (ltwatson@computer.org)
!   William I. Thacker (thackerw@winthrop.edu)
!
! VERSION HISTORY:
!   June 2020 -- (tchl) Created file, (ltw / wit) reviewed and revised.
!
SUBROUTINE MQSI(X, Y, T, BCOEF, INFO, UV)
! Computes a monotone quintic spline interpolant (MQSI), Q(X), to data
! in terms of spline coefficients BCOEF for a B-spline basis defined by
! knots T. Q(X) is theoretically guaranteed to be monotone increasing
! (decreasing) over exactly the same intervals [X(I), X(J)] that the
! data Y(.) is monotone increasing (decreasing).
!
! INPUT:
!   X(1:ND) -- A real array of ND increasing values.
!   Y(1:ND) -- A real array of ND function values Y(I) = F(X(I)).
!
! OUTPUT:
!   T(1:3*ND+6) -- The knots for the MQSI B-spline basis.
!   BCOEF(1:3*ND) -- The coefficients for the MQSI B-spline basis.
!   INFO -- Integer representing the subroutine return status.
!     0  Normal return.
!     1  There are fewer than three data points, so there is nothing to do.
!     2  X(:) and Y(:) have different sizes.
!     3  The size of T(:) must be at least the number of knots 3*ND+6.
!     4  The size of BCOEF(:) must be at least the spline space dimension 3*ND.
!     5  The values in X(:) are not increasing or not separated by at
!        least the square root of machine precision.
!     6  The magnitude or spacing of the data (X(:), Y(:)) could lead to
!        overflow. Some differences |Y(I+1) - Y(I)| or |X(I+1) - X(I)| exceed
```

```
!        10**38, which could lead to local Lipschitz constants exceeding 10**54.
!     7  The computed spline does not match the provided data in Y(:) and
!        this result should not be used. This arises when the scaling of
!        function values and derivative values causes the linear system used
!        to compute the final spline interpolant to have a prohibitively
!        large condition number.
!     8  The optional array UV must have size at least ND x 2.
!   >10  20 plus the info flag as returned by DGBSV from LAPACK when
!        computing the final spline interpolant.
!   UV(1:ND,1:2) -- First and second derivatives of Q(X) at the breakpoints
!     (optional argument).
!
USE REAL_PRECISION, ONLY: R8
IMPLICIT NONE
! Arguments.
REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: X
REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: Y
REAL(KIND=R8), INTENT(OUT),   DIMENSION(:) :: T, BCOEF
INTEGER, INTENT(OUT) :: INFO
REAL(KIND=R8), INTENT(OUT), DIMENSION(:,:), OPTIONAL :: UV
! Local variables.
!  Estimated first and second derivatives (columns) by quadratic
!  facet model at all data points (rows).
REAL(KIND=R8), DIMENSION(SIZE(X),2) :: FHATX
!  Spline values, first, and second derivatives (columns) at all points (rows).
REAL(KIND=R8), DIMENSION(SIZE(X),3) :: FX
!  Execution queues holding intervals to check for monotonicity
!  CHECKING, TO_CHECK, derivative values to grow (after shrinking)
!  GROWING, TO_GROW, and derivative values to shrink (because of
!  nonmonotonicity) SHRINKING, TO_SHRINK. The arrays GROWING and
!  SHRINKING are repurposed to identify locations of local maxima and
!  minima of provided Y values early in the routine.
LOGICAL, DIMENSION(SIZE(X)) :: CHECKING, GROWING, SHRINKING
INTEGER, DIMENSION(SIZE(X)) :: TO_CHECK, TO_GROW, TO_SHRINK
!  Coefficients for a quadratic interpolant A and B, the direction of
!  function change DIRECTION, a derivative value DX, the exponent
!  scale difference between X and Y values SCALE, and the current
!  bisection search (relative) step size STEP_SIZE.
REAL(KIND=R8) :: A, B, DIRECTION, DX, SCALE, STEP_SIZE
!  The smallest spacing of X and step size ACCURACY, the machine
!  precision at unit scale EPS, the largest allowed spacing of X or Y
!  values 10^38 TP38, and the placeholder large magnitude curvature
!  value 10^54 TP54.
REAL(KIND=R8), PARAMETER :: ACCURACY = SQRT(EPSILON(1.0_R8)), &
     EPS = EPSILON(1.0_R8), &
```

```
      TP38 = 10.0_R8**38, TP54 = 10.0_R8**54
!  Iteration indices I and J, number of data points ND, counters for
!  execution queues, checking, NC, growing, NG, and shrinking, NS.
INTEGER :: I, IM1, IP1, J, NC, ND, NG, NS
!  Boolean indicating whether the bisection search is in progress.
LOGICAL :: SEARCHING
INTERFACE
   SUBROUTINE FIT_SPLINE(XI, FX, T, BCOEF, INFO)
     USE REAL_PRECISION, ONLY: R8
     REAL(KIND=R8), INTENT(IN),  DIMENSION(:)   :: XI
     REAL(KIND=R8), INTENT(IN),  DIMENSION(:,:) :: FX
     REAL(KIND=R8), INTENT(OUT), DIMENSION(:)   :: T, BCOEF
     INTEGER, INTENT(OUT) :: INFO
   END SUBROUTINE FIT_SPLINE
END INTERFACE
ND = SIZE(X)
! Check the shape of incoming arrays.
IF      (ND .LT. 3)             THEN; INFO = 1; RETURN
ELSE IF (SIZE(Y) .NE. ND)       THEN; INFO = 2; RETURN
ELSE IF (SIZE(T) .LT. 3*ND + 6) THEN; INFO = 3; RETURN
ELSE IF (SIZE(BCOEF) .LT. 3*ND) THEN; INFO = 4; RETURN
END IF
! Verify that X values are increasing, and that (X,Y) data is not extreme.
DO I = 1, ND-1
   IP1 = I+1
   IF ((X(IP1) - X(I)) .LT. ACCURACY)    THEN; INFO = 5; RETURN
   ELSE IF (((X(IP1) - X(I)) .GT. TP38) .OR. &
         (ABS(Y(IP1) - Y(I)) .GT. TP38)) THEN; INFO = 6; RETURN
   END IF
END DO
IF (PRESENT(UV)) THEN
   IF ((SIZE(UV,DIM=1) .LT. ND) .OR. (SIZE(UV,DIM=2) .LT. 2)) THEN
      INFO = 8; RETURN; END IF
END IF
! Scale Y by an exact power of 2 to make Y and X commensurate, store
! scaled Y in FX and also back in Y.
J = INT(LOG((1.0_R8+MAXVAL(ABS(Y(:))))/MAXVAL(ABS(X(:)))) / LOG(2.0_R8))
SCALE = 2.0_R8**J
FX(:,1) = (1.0_R8/SCALE)*Y(:)
Y(:) = FX(:,1)

! ======================================================================
!          Algorithm 1: Estimate initial derivatives by
!         using a minimum curvature quadratic facet model.
!
```

```
! Identify local extreme points and flat points. Denote location
! of flats in GROWING and extrema in SHRINKING to save memory.
GROWING(1) = ABS(Y(1) - Y(2)) .LT. EPS*(1.0_R8+ABS(Y(1))+ABS(Y(2)))
GROWING(ND) = ABS(Y(ND-1) - Y(ND)) .LT. EPS*(1.0_R8+ABS(Y(ND-1))+ABS(Y(ND)))
SHRINKING(1) = .FALSE.
SHRINKING(ND) = .FALSE.
DO I = 2, ND-1
   IM1 = I-1
   IP1 = I+1
   IF ((ABS(Y(IM1)-Y(I)) .LT. EPS*(1.0_R8+ABS(Y(IM1))+ABS(Y(I)))) .OR. &
       (ABS(Y(I)-Y(IP1)) .LT. EPS*(1.0_R8+ABS(Y(I))+ABS(Y(IP1))))) THEN
      GROWING(I) = .TRUE.
      SHRINKING(I) = .FALSE.
   ELSE
      GROWING(I) = .FALSE.
      SHRINKING(I) = (Y(I) - Y(IM1)) * (Y(IP1) - Y(I)) .LT. 0.0_R8
   END IF
END DO
! Use local quadratic interpolants to estimate slopes and second
! derivatives at all points. Use zero-sloped quadratic interpolants
! at extrema with left/right neighbors to estimate curvature.
estimate_derivatives : DO I = 1, ND
   IM1 = I-1
   IP1 = I+1
   ! Initialize the curvature to be maximally large.
   FX(I,3) = TP54
   ! If this is locally flat, then first and second derivatives are zero valued.
   pick_quadratic : IF (GROWING(I)) THEN
      FX(I,2:3) = 0.0_R8
   ! If this is an extreme point (local minimum or maximum Y),
   ! construct quadratic interpolants that have zero slope here and
   ! hit left/right neighbors.
   ELSE IF (SHRINKING(I)) THEN
      ! Set the first derivative to zero.
      FX(I,2) = 0.0_R8
      ! Compute the coefficient A in  Ax^2+Bx+C  that interpolates at X(I-1).
      FX(I,3) = (Y(IM1) - Y(I)) / (X(IM1) - X(I))**2
      ! Compute the coefficient A in  Ax^2+Bx+C  that interpolates at X(I+1).
      A = (Y(IP1) - Y(I)) / (X(IP1) - X(I))**2
      IF (ABS(A) .LT. ABS(FX(I,3))) FX(I,3) = A
      ! Compute the actual second derivative (instead of coefficient A).
      FX(I,3) = 2.0_R8 * FX(I,3)
   ELSE
      ! Determine the direction of change at the point I.
      IF (I .EQ. 1) THEN
```

```
   IF      (Y(I) .LT. Y(IP1)) THEN; DIRECTION =  1.0_R8
   ELSE IF (Y(I) .GT. Y(IP1)) THEN; DIRECTION = -1.0_R8
   END IF
ELSE
   IF      (Y(IM1) .LT. Y(I)) THEN; DIRECTION =  1.0_R8
   ELSE IF (Y(IM1) .GT. Y(I)) THEN; DIRECTION = -1.0_R8
   END IF
END IF
! --------------------
! Quadratic left of I.
IF (IM1 .GT. 1) THEN
   ! If a zero derivative at left point, use its right interpolant.
   IF (SHRINKING(IM1) .OR. GROWING(IM1)) THEN
      A = (Y(I) - Y(IM1)) / (X(I) - X(IM1))**2
      B = -2.0_R8 * X(IM1) * A
   ! Otherwise use the standard quadratic on the left.
   ELSE; CALL QUADRATIC(X, Y, IM1, A, B)
   END IF
   DX = 2.0_R8*A*X(I) + B
   IF (DX*DIRECTION .GE. 0.0_R8) THEN
      FX(I,2) = DX
      FX(I,3) = A
   END IF
END IF
! ------------------------
! Quadratic centered at I (require that it has at least one
! neighbor that is not forced to zero slope).
IF ((I .GT. 1) .AND. (I .LT. ND)) THEN
   IF (.NOT. ((SHRINKING(IM1) .OR. GROWING(IM1)) .AND. &
        (SHRINKING(IP1) .OR. GROWING(IP1)))) THEN
      ! Construct quadratic interpolant through this point and neighbors.
      CALL QUADRATIC(X, Y, I, A, B)
      DX = 2.0_R8*A*X(I) + B
      ! Keep this new quadratic if it has less curvature.
      IF ((DX*DIRECTION .GE. 0.0_R8) .AND. (ABS(A) .LT. ABS(FX(I,3)))) THEN
         FX(I,2) = DX
         FX(I,3) = A
      END IF
   END IF
END IF
! ---------------------
! Quadratic right of I.
IF (IP1 .LT. ND) THEN
   ! If a zero derivative at right point, use its left interpolant.
   IF (SHRINKING(IP1) .OR. GROWING(IP1)) THEN
```

```
            A = (Y(I) - Y(IP1)) / (X(I) - X(IP1))**2
            B = -2.0_R8 * X(IP1) * A
         ! Otherwise use the standard quadratic on the right.
         ELSE; CALL QUADRATIC(X, Y, IP1, A, B)
         END IF
         DX = 2.0_R8*A*X(I) + B
         ! Keep this new quadratic if it has less curvature.
         IF ((DX*DIRECTION .GE. 0.0_R8) .AND. (ABS(A) .LT. ABS(FX(I,3)))) THEN
            FX(I,2) = DX
            FX(I,3) = A
         END IF
      END IF
      ! Set the final quadratic.
      IF (FX(I,3) .EQ. TP54) THEN
         FX(I,2:3) = 0.0_R8
      ! Compute curvature of quadratic from coefficient of x^2.
      ELSE
         FX(I,3) = 2.0_R8 * FX(I,3)
      END IF
   END IF pick_quadratic
END DO estimate_derivatives

! ======================================================================
!          Algorithm 3: Identify viable piecewise monotone
!        derivative values by doing a quasi-bisection search.
!
! Store the initially estimated first and second derivative values.
FHATX(:,1:2) = FX(:,2:3)
! Identify which spline segments are not monotone and need to be fixed.
CHECKING(:) = .FALSE.
SHRINKING(:) = .FALSE.
NC = 0; NG = 0; NS = 0
DO I = 1, ND-1
   IP1 = I+1
   ! Check for monotonicity on all segments that are not flat.
   IF ((.NOT. (GROWING(I) .AND. GROWING(IP1))) .AND. &
        (.NOT. IS_MONOTONE(X(I), X(IP1), FX(I,1), FX(IP1,1), &
        FX(I,2), FX(IP1,2), FX(I,3), FX(IP1,3)))) THEN
      ! Store points bounding nonmonotone segments in the TO_SHRINK queue.
      IF (.NOT. SHRINKING(I)) THEN
         SHRINKING(I) = .TRUE.
         NS = NS+1
         TO_SHRINK(NS) = I
      END IF
      IF (.NOT. SHRINKING(IP1)) THEN
```

```
         SHRINKING(IP1) = .TRUE.
         NS = NS+1
         TO_SHRINK(NS) = IP1
      END IF
   END IF
END DO
! Initialize step size to 1.0 (will be halved at beginning of loop).
STEP_SIZE = 1.0_R8
SEARCHING = .TRUE.
GROWING(:) = .FALSE.
! Loop until the accuracy is achieved and *all* intervals are monotone.
DO WHILE (SEARCHING .OR. (NS .GT. 0))
   ! Compute the step size for this iteration.
   IF (SEARCHING) THEN
      STEP_SIZE = STEP_SIZE / 2.0_R8
      IF (STEP_SIZE .LT. ACCURACY) THEN
         SEARCHING = .FALSE.
         STEP_SIZE = ACCURACY
         NG = 0
      END IF
   ! Grow the step size (at a slower rate than the step size reduction
   ! rate) if there are still intervals to fix.
   ELSE
      STEP_SIZE = STEP_SIZE * 1.5_R8
   END IF
   ! Grow all those first and second derivatives that were previously
   ! shrunk, and correspond to currently monotone spline pieces.
   grow_values : DO J = 1, NG
      I = TO_GROW(J)
      ! Do not grow values that are actively related to a nonmonotone
      ! spline segment.
      IF (SHRINKING(I)) CYCLE grow_values
      ! Otherwise, grow those values that have been modified previously.
      FX(I,2) = FX(I,2) + STEP_SIZE * FHATX(I,1)
      FX(I,3) = FX(I,3) + STEP_SIZE * FHATX(I,2)
      ! Make sure the first derivative does not exceed its original value.
      IF ((FHATX(I,1) .LT. 0.0_R8) .AND. (FX(I,2) .LT. FHATX(I,1))) THEN
         FX(I,2) = FHATX(I,1)
      ELSE IF ((FHATX(I,1) .GT. 0.0_R8) .AND. (FX(I,2) .GT. FHATX(I,1))) THEN
         FX(I,2) = FHATX(I,1)
      END IF
      ! Make sure the second derivative does not exceed its original value.
      IF ((FHATX(I,2) .LT. 0.0_R8) .AND. (FX(I,3) .LT. FHATX(I,2))) THEN
         FX(I,3) = FHATX(I,2)
      ELSE IF ((FHATX(I,2) .GT. 0.0_R8) .AND. (FX(I,3) .GT. FHATX(I,2))) THEN
```

```
            FX(I,3) = FHATX(I,2)
         END IF
         ! Set this point and its neighboring intervals to be checked for
         ! monotonicity. Use sequential IF statements for short-circuiting.
         IF (I .GT. 1) THEN; IF (.NOT. CHECKING(I-1)) THEN
            CHECKING(I-1) = .TRUE.
            NC = NC+1
            TO_CHECK(NC) = I-1
         END IF; END IF
         IF (I .LT. ND) THEN; IF (.NOT. CHECKING(I)) THEN
            CHECKING(I) = .TRUE.
            NC = NC+1
            TO_CHECK(NC) = I
         END IF; END IF
      END DO grow_values
      ! Shrink the first and second derivatives that cause nonmonotonicity.
      shrink_values : DO J = 1, NS
         I = TO_SHRINK(J)
         SHRINKING(I) = .FALSE.
         IF (SEARCHING .AND. (.NOT. GROWING(I))) THEN
            GROWING(I) = .TRUE.
            NG = NG+1
            TO_GROW(NG) = I
         END IF
         ! Shrink the values that are causing nonmonotonicity.
         FX(I,2) = FX(I,2) - STEP_SIZE * FHATX(I,1)
         FX(I,3) = FX(I,3) - STEP_SIZE * FHATX(I,2)
         ! Make sure the first derivative does not pass zero.
         IF ((FHATX(I,1) .LT. 0.0_R8) .AND. (FX(I,2) .GT. 0.0_R8)) THEN
            FX(I,2) = 0.0_R8
         ELSE IF ((FHATX(I,1) .GT. 0.0_R8) .AND. (FX(I,2) .LT. 0.0_R8)) THEN
            FX(I,2) = 0.0_R8
         END IF
         ! Make sure the second derivative does not pass zero.
         IF ((FHATX(I,2) .LT. 0.0_R8) .AND. (FX(I,3) .GT. 0.0_R8)) THEN
            FX(I,3) = 0.0_R8
         ELSE IF ((FHATX(I,2) .GT. 0.0_R8) .AND. (FX(I,3) .LT. 0.0_R8)) THEN
            FX(I,3) = 0.0_R8
         END IF
         ! Set this point and its neighboring intervals to be checked for
         ! monotonicity.
         IF ((I .GT. 1) .AND. (.NOT. CHECKING(I-1))) THEN
            CHECKING(I-1) = .TRUE.
            NC = NC+1
            TO_CHECK(NC) = I-1
```

```
      END IF
      IF ((I .LT. ND) .AND. (.NOT. CHECKING(I))) THEN
         CHECKING(I) = .TRUE.
         NC = NC+1
         TO_CHECK(NC) = I
      END IF
   END DO shrink_values
   ! Identify which spline segments are nonmonotone after the updates.
   NS = 0
   check_monotonicity : DO J = 1, NC
      I = TO_CHECK(J)
      IP1 = I+1
      CHECKING(I) = .FALSE.
      IF (.NOT. IS_MONOTONE(X(I), X(IP1), FX(I,1), FX(IP1,1), &
           FX(I,2), FX(IP1,2), FX(I,3), FX(IP1,3))) THEN
         IF (.NOT. SHRINKING(I)) THEN
            SHRINKING(I) = .TRUE.
            NS = NS+1
            TO_SHRINK(NS) = I
         END IF
         IF (.NOT. SHRINKING(IP1)) THEN
            SHRINKING(IP1) = .TRUE.
            NS = NS+1
            TO_SHRINK(NS) = IP1
         END IF
      END IF
   END DO check_monotonicity
   NC = 0
END DO
! ----------------------------------------------------------------------

! Use FIT_SPLINE to fit the final MQSI. For numerical stability and accuracy
! this routine requires the enforced separation of the values X(I).
CALL FIT_SPLINE(X, FX, T, BCOEF, INFO)

! Restore Y to its original value and unscale spline and derivative values.
BCOEF(1:3*ND) = SCALE * BCOEF(1:3*ND)
Y(:) = SCALE*Y(:) ! Restore original Y.
IF (PRESENT(UV)) THEN; UV(1:ND,1:2) = FX(1:ND,2:3); END IF ! Return first
! and second derivative values at breakpoints.

CONTAINS

! ======================================================================
!          Algorithm 2: Check for monotonicity using tight
```

```
!        theoretical constraints on a quintic polynomial piece.
!
FUNCTION IS_MONOTONE(U0, U1, F0, F1, DF0, DF1, DDF0, DDF1)
! Given an interval [U0, U1] and function values F0, F1, first
! derivatives DF0, DF1, and second derivatives DDF0, DDF1 at U0, U1,
! respectively, IS_MONOTONE = TRUE if the quintic polynomial matching
! these values is monotone over [U0, U1], and IS_MONOTONE = FALSE otherwise.
!
REAL(KIND=R8), INTENT(IN) :: U0, U1, F0, F1, DF0, DF1, DDF0, DDF1
LOGICAL :: IS_MONOTONE
! Local variables.
REAL(KIND=R8), PARAMETER :: EPS = EPSILON(1.0_R8)
REAL(KIND=R8) :: ALPHA, BETA, GAMMA, SIGN, TEMP, W

! When the function values are flat, everything *must* be 0.
IF (ABS(F1 - F0) .LT. EPS*(1.0_R8 + ABS(F1) + ABS(F0))) THEN
   IS_MONOTONE = (DF0  .EQ. 0.0_R8) .AND. (DF1  .EQ. 0.0_R8) .AND. &
                 (DDF0 .EQ. 0.0_R8) .AND. (DDF1 .EQ. 0.0_R8)
   RETURN
END IF
! Identify the direction of change of the function (increasing or decreasing).
IF (F1 .GT. F0) THEN
   SIGN =  1.0_R8
ELSE
   SIGN = -1.0_R8
END IF
W = U1 - U0
! Determine which set of monotonicity conditions to use based on the
! assigned first derivative values at either end of the interval.
IF ((ABS(DF0) .LT. EPS) .OR. (ABS(DF1) .LT. EPS)) THEN
   ! Simplified monotone case, which reduces to a test of cubic
   ! positivity studied in
   !
   ! J. W. Schmidt and W. He{\ss}, ``Positivity of cubic polynomials on
   ! intervals and positive spline interpolation'', {\sl BIT Numerical
   ! Mathematics}, 28 (1988) 340--352.
   !
   ! Notably, monotonicity results when the following conditions hold:
   !    alpha >= 0,
   !    delta >= 0,
   !    beta  >= alpha - 2 * sqrt{ alpha delta },
   !    gamma >= delta - 2 * sqrt{ alpha delta },
   !
   ! where alpha, beta, delta, and gamma are defined in the paper. The
   ! equations that follow are the result of algebraic simplifications
```

```
   ! of the terms as they are defined by Schmidt and He{\ss}.
   !
   ! The condition delta >= 0 was enforced when first estimating
   ! derivative values (with correct sign). Next check for alpha >= 0.
   IF (SIGN*DDF1*W .GT. SIGN*4.0_R8*DF1) THEN
      IS_MONOTONE = .FALSE.
   ELSE
      ! Compute a simplification of 2 * sqrt{ alpha delta }.
      TEMP = DF0 * (4*DF1 - DDF1*W)
      IF (TEMP .GT. 0.0_R8) TEMP = 2.0_R8 * SQRT(TEMP)
      ! Check for gamma >= delta - 2 * sqrt{ alpha delta }
      IF (TEMP + SIGN*(3.0_R8*DF0 + DDF0*W) .LT. 0.0_R8) THEN
         IS_MONOTONE = .FALSE.
      ! Check for beta >= alpha - 2 * sqrt{ alpha delta }
      ELSE IF (60.0_R8*(F1-F0)*SIGN - W*(SIGN*(24.0_R8*DF0 + 32.0_R8*DF1) &
           - 2.0_R8*TEMP + W*SIGN*(3.0_R8*DDF0 - 5.0_R8*DDF1)) .LT. 0.0_R8) THEN
         IS_MONOTONE = .FALSE.
      ELSE
         IS_MONOTONE = .TRUE.
      END IF
   END IF
ELSE
   ! Full quintic monotonicity case related to the theory in
   !
   ! G. Ulrich and L. T. Watson, ``Positivity conditions for quartic
   ! polynomials'', {\sl SIAM J. Sci. Comput.}, 15 (1994) 528--544.
   !
   ! Monotonicity results when the following conditions hold:
   !    tau_1 > 0,
   !    if (beta <= 6) then
   !       alpha > -(beta + 2) / 2,
   !       gamma > -(beta + 2) / 2,
   !    else
   !       alpha > -2 sqrt(beta - 2),
   !       gamma > -2 sqrt(beta - 2),
   !    end if
   !
   ! where alpha, beta, gamma, and tau_1 are defined in the paper. The
   ! following conditions are the result of algebraic simplifications
   ! of the terms as defined by Ulrich and Watson.
   !
   ! Check for tau_1 > 0.
   IF (W*(2.0_R8*SQRT(DF0*DF1) - SIGN * 3.0_R8*(DF0+DF1)) - &
        SIGN*24.0_R8*(F0-F1) .LE. 0.0_R8) THEN
      IS_MONOTONE = .FALSE.
```

```
   ELSE
      ! Compute alpha, gamma, beta from theorems to determine monotonicity.
      TEMP = (DF0*DF1)**(0.75_R8)
      ALPHA = SIGN * (4.0_R8*DF1 - DDF1*W) * SQRT(SIGN*DF0) / TEMP
      GAMMA = SIGN * (4.0_R8*DF0 + DDF0*W) * SQRT(SIGN*DF1) / TEMP
      BETA = SIGN * (3.0_R8 * ((DDF1-DDF0)*W - 8.0_R8*(DF0+DF1)) + &
           60.0_R8 * (F1-F0) / W) / (2.0_R8 * SQRT(DF0*DF1))
      IF (BETA .LE. 6.0_R8) THEN
         TEMP = -(BETA + 2.0_R8) / 2.0_R8
      ELSE
         TEMP = -2.0_R8 * SQRT(BETA - 2.0_R8)
      END IF
      IS_MONOTONE = (ALPHA .GT. TEMP) .AND. (GAMMA .GT. TEMP)
   END IF
END IF
END FUNCTION IS_MONOTONE

SUBROUTINE QUADRATIC(X, Y, I2, A, B)
! Given data X, Y, an index I2, compute the coefficients A of x^2 and B
! of x for the quadratic interpolating Y(I2-1:I2+1) at X(I2-1:I2+1).
REAL(KIND=R8), INTENT(IN),  DIMENSION(:) :: X, Y
INTEGER, INTENT(IN) :: I2
REAL(KIND=R8), INTENT(OUT) :: A, B
! Local variables.
REAL(KIND=R8) :: C1, C2, C3, & ! Intermediate terms for computation.
  D ! Denominator for computing A and B via Cramer's rule.
INTEGER :: I1, I3
I1 = I2-1
I3 = I2+1
! The earlier tests for extreme data (X,Y) keep the quantities below
! within floating point range. Compute the shared denominator.
D = (X(I1) - X(I2)) * (X(I1) - X(I3)) * (X(I2) - X(I3))
! Compute coefficients A and B in quadratic interpolant  Ax^2 + Bx + C.
C1 = X(I1) * (Y(I3) - Y(I2))
C2 = X(I2) * (Y(I1) - Y(I3))
C3 = X(I3) * (Y(I2) - Y(I1))
A = (C1 + C2 + C3) / D
B = - (X(I1)*C1 + X(I2)*C2 + X(I3)*C3) / D
END SUBROUTINE QUADRATIC

END SUBROUTINE MQSI

! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
!                           SPLINE.f90
!
```

```
! DESCRIPTION:
!   This file defines the subroutines FIT_SPLINE for computing the
!   coefficients of a B-spline basis necessary to reproduce given
!   function and derivative values, and EVAL_SPLINE for evaluating the
!   value, integral, and derivatives of a spline defined in terms of
!   its B-spline basis.
!
! CONTAINS:
!   SUBROUTINE FIT_SPLINE(XI, FX, T, BCOEF, INFO)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),  DIMENSION(:)   :: XI
!     REAL(KIND=R8), INTENT(IN),  DIMENSION(:,:) :: FX
!     REAL(KIND=R8), INTENT(OUT), DIMENSION(:)   :: T, BCOEF
!     INTEGER, INTENT(OUT) :: INFO
!   END SUBROUTINE FIT_SPLINE
!
!   SUBROUTINE EVAL_SPLINE(T, BCOEF, XY, INFO, D)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T, BCOEF
!     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
!     INTEGER, INTENT(OUT) :: INFO
!     INTEGER, INTENT(IN), OPTIONAL :: D
!   END SUBROUTINE EVAL_SPLINE
!
! EXTERNAL DEPENDENCIES:
!   MODULE REAL_PRECISION
!     INTEGER, PARAMETER :: R8
!   END MODULE REAL_PRECISION
!
!   SUBROUTINE EVAL_BSPLINE(T, XY, D)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T
!     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
!     INTEGER, INTENT(IN), OPTIONAL :: D
!   END SUBROUTINE EVAL_BSPLINE
!
! CONTRIBUTORS:
!   Thomas C.H. Lux (tchlux@vt.edu)
!   Layne T. Watson (ltwatson@computer.org)
!   William I. Thacker (thackerw@winthrop.edu)
!
! VERSION HISTORY:
!   June 2020 -- (tchl) Created file, (ltw / wit) reviewed and revised.
!
SUBROUTINE FIT_SPLINE(XI, FX, T, BCOEF, INFO)
```

```
! Subroutine for computing a linear combination of B-splines that
! interpolates the given function value (and function derivatives)
! at the given breakpoints.
!
! INPUT:
!   XI(1:NB) -- The increasing real-valued locations of the NB
!               breakpoints for the interpolating spline.
!   FX(1:NB,1:NCC) -- FX(I,J) contains the (J-1)st derivative at
!                     XI(I) to be interpolated, providing NCC
!                     continuity conditions at all NB breakpoints.
!
! OUTPUT:
!   T(1:NB*NCC+2*NCC) -- The nondecreasing real-valued locations
!                        of the knots for the B-spline basis.
!   BCOEF(1:NB*NCC) -- The coefficients for the B-splines that define
!                      the interpolating spline.
!   INFO -- Integer representing the subroutine execution status:
!     0    Successful execution.
!     1    SIZE(XI) is less than 3.
!     3    SIZE(FX,1) does not equal SIZE(XI).
!     4    SIZE(T) too small, should be at least NB*NCC + 2*NCC.
!     5    SIZE(BCOEF) too small, should be at least NB*NCC.
!     6    Elements of XI are not strictly increasing.
!     7    The computed spline does not match the provided FX
!          and this fit should be disregarded. This arises when
!          the scaling of function values and derivative values
!          causes the resulting linear system to have a
!          prohibitively large condition number.
!   >10    20 plus the info flag as returned by DGBSV from LAPACK.
!
!
! DESCRIPTION:
!   This subroutine computes the B-spline basis representation of the
!   spline interpolant to given function values (and derivative values)
!   at unique breakpoints. The osculatory interpolating spline of order
!   2*NCC is returned in terms of knots T and coefficients BCOEF, defining
!   the underlying B-splines and their linear combination that interpolates
!   the given data. This function uses the subroutine EVAL_BSPLINE to
!   evaluate the B-splines at all knots and the LAPACK routine DGBSV to
!   compute the B-spline coefficients. The difference between the provided
!   function (and derivative) values and the actual values produced by the
!   computed interpolant can vary depending on the spacing of the knots
!   and the magnitudes of the values provided. When the condition number
!   of the linear system defining BCOEF is large, the computed interpolant
!   may fail to accurately reproduce the data, indicated by INFO = 7.
```

```
!
USE REAL_PRECISION, ONLY: R8
IMPLICIT NONE
! Arguments.
REAL(KIND=R8), INTENT(IN),  DIMENSION(:)   :: XI
REAL(KIND=R8), INTENT(IN),  DIMENSION(:,:) :: FX
REAL(KIND=R8), INTENT(OUT), DIMENSION(:)   :: T, BCOEF
INTEGER, INTENT(OUT) :: INFO
! Local variables.
!  LAPACK pivot indices.
INTEGER, DIMENSION(SIZE(BCOEF)) :: IPIV
!  Storage for linear system that is solved to get B-spline coefficients.
REAL(KIND=R8), DIMENSION(1 + 3*(2*SIZE(FX,2)-1), SIZE(FX)) :: AB
!  Maximum allowed (relative) error in spline function values.
REAL(KIND=R8), PARAMETER :: MAX_ERROR = SQRT(SQRT(EPSILON(1.0_R8)))
INTEGER :: DEGREE, & ! Degree of B-spline.
     DERIV, & ! Derivative loop index.
     I, I1, I2, J, J1, J2, & ! Miscellaneous loop indices.
     K, &    ! Order of B-splines = 2*NCC.
     NB, &   ! Number of breakpoints.
     NCC, &  ! Number of continuity conditions.
     NK, &   ! Number of knots = NSPL + 2*NCC.
     NSPL    ! Dimension of spline space = number of B-spline
             ! coefficients = NB * NCC.
! LAPACK subroutine for solving banded linear systems.
EXTERNAL :: DGBSV
INTERFACE
   SUBROUTINE EVAL_BSPLINE(T, XY, D)
     USE REAL_PRECISION, ONLY: R8
     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T
     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
     INTEGER, INTENT(IN), OPTIONAL :: D
   END SUBROUTINE EVAL_BSPLINE
   SUBROUTINE EVAL_SPLINE(T, BCOEF, XY, INFO, D)
     USE REAL_PRECISION, ONLY: R8
     REAL(KIND=R8), INTENT(IN), DIMENSION(:) :: T, BCOEF
     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
     INTEGER, INTENT(OUT) :: INFO
     INTEGER, INTENT(IN), OPTIONAL :: D
   END SUBROUTINE EVAL_SPLINE
END INTERFACE

! Define some local variables for notational convenience.
NB = SIZE(XI)
NCC = SIZE(FX,2)
```

```
NSPL = SIZE(FX)
NK = NSPL + 2*NCC
K = 2*NCC
DEGREE = K - 1
INFO = 0

! Check the shape of incoming arrays.
IF      (NB .LT. 3)             THEN; INFO = 1; RETURN
ELSE IF (SIZE(FX,1) .NE. NB)    THEN; INFO = 3; RETURN
ELSE IF (SIZE(T) .LT. NK)       THEN; INFO = 4; RETURN
ELSE IF (SIZE(BCOEF) .LT. NSPL) THEN; INFO = 5; RETURN
END IF
! Verify that breakpoints are increasing.
DO I = 1, NB - 1
   IF (XI(I) .GE. XI(I+1)) THEN
      INFO = 6
      RETURN
   END IF
END DO

! Copy the knots that will define the B-spline representation.
! Each internal knot will be repeated NCC times to maintain the
! necessary level of continuity for this spline.
T(1:K) = XI(1)
DO I = 2, NB-1
   T(I*NCC+1 : (I+1)*NCC) = XI(I)
END DO
! Assign the last knot to exist a small step outside the supported
! interval to ensure the B-spline basis functions are nonzero at the
! rightmost breakpoint.
T(NK-DEGREE:NK) = MAX( XI(NB) + ABS(XI(NB))*SQRT(EPSILON(XI(NB))),  &
                       XI(NB) + SQRT(EPSILON(XI(NB))) )

! The next block of code evaluates each B-spline and it's derivatives
! at all breakpoints. The first and last elements of XI will be
! repeated K times and each internal breakpoint will be repeated NCC
! times. As a result, the support of each B-spline spans at most three
! breakpoints. The coefficients for the B-spline basis are determined
! by solving a linear system with NSPL columns (one column for each
! B-spline) and NB*NCC rows (one row for each value of the
! interpolating spline).
!
! For example, a C^1 interpolating spline over three breakpoints
! will match function value and first derivative at each breakpoint
! requiring six fourth order (third degree) B-splines each composed
```

```
! from five knots. Below, the six B-splines are numbered (first
! number, columns) and may be nonzero at the three breakpoints
! (middle letter, rows) for each function value (odd rows, terms end
! with 0) and first derivative (even rows, terms end with 1). The
! linear system will look like:
!
!       B-SPLINE VALUES AT BREAKPOINTS      SPLINE          VALUES
!        1st  2nd  3rd  4th  5th  6th    COEFFICIENTS
!      _                                _     _   _           _    _
!     |                                  |   |     |         |      |
!   B |  1a0  2a0  3a0  4a0              |   |  1  |         |  a0  |
!   R |  1a1  2a1  3a1  4a1              |   |  2  |         |  a1  |
!   E |  1b0  2b0  3b0  4b0  5b0  6b0    |   |  3  |   ===   |  b0  |
!   A |  1b1  2b1  3b1  4b1  5b1  6b1    |   |  4  |   ===   |  b1  |
!   K |            3c0  4c0  5c0  6c0    |   |  5  |         |  c0  |
!   S |            3c1  4c1  5c1  6c1    |   |  6  |         |  c1  |
!     |_                                _|   |_   _|         |_    _|
!
! Notice this matrix is banded with lower/upper bandwidths KL/KU equal
! to (one less than the maximum number of breakpoints for which a
! spline takes on a nonzero value) times (the number of continuity
! conditions) minus (one). In general KL = KU = DEGREE = K - 1.

! Initialize all values in AB to zero.
AB(:,:) = 0.0_R8
! Evaluate all B-splines at all breakpoints (walking through columns).
DO I = 1, NSPL
   ! Compute index of the last knot for the current B-spline.
   J = I + K
   ! Compute the row indices in the coefficient matrix A.
   I1 = ((I-1)/NCC - 1) * NCC + 1 ! First row.
   I2 = I1 + 3*NCC - 1            ! Last row.
   ! Only two breakpoints will be covered for the first NCC
   ! B-splines and the last NCC B-splines.
   IF (I .LE. NCC)      I1 = I1 + NCC
   IF (I+NCC .GT. NSPL) I2 = I2 - NCC
   ! Compute the indices of the involved breakpoints.
   J1 = I1 / NCC + 1 ! First breakpoint.
   J2 = I2 / NCC     ! Last breakpoint.
   ! Convert the i,j indices in A to the banded storage scheme in AB.
   ! The mapping looks like   A[i,j] --> AB[KL+KU+1+i-j,j] .
   I1 = 2*DEGREE+1 + I1 - I
   I2 = 2*DEGREE+1 + I2 - I
   ! Evaluate this B-spline, computing function value and derivatives.
   DO DERIV = 0, NCC-1
```

```
      ! Place the evaluations into a subsection of a column in AB,
      ! shift according to which derivative is being evaluated and use
      ! a stride determined by the number of continuity conditions.
      AB(I1+DERIV:I2:NCC,I) = XI(J1:J2)
      CALL EVAL_BSPLINE(T(I:J), AB(I1+DERIV:I2:NCC,I), D=DERIV)
   END DO
END DO
! Copy the FX into the BCOEF (output) variable.
DO I = 1, NCC ; BCOEF(I:NSPL:NCC) = FX(:,I) ; END DO
! Call the LAPACK subroutine to solve the banded linear system.
CALL DGBSV(NSPL, DEGREE, DEGREE, 1, AB, SIZE(AB,1), IPIV, BCOEF, NSPL, INFO)
! Check for errors in the execution of DGBSV, (this should not happen).
IF (INFO .NE. 0) THEN; INFO = INFO + 20; RETURN; END IF
! Check to see if the linear system was correctly solved by looking at
! the difference between produced B-spline values and provided values.
DO DERIV = 0, NCC-1
   DO I = 1, NB
      ! Compute the indices of the first and last B-splines that have a knot
      ! at this breakpoint.
      I1 = MAX(1, (I-2)*NCC+1)
      I2 = MIN(NB*NCC, (I+1)*NCC)
      ! Evaluate this spline at this breakpoint. Correct usage is
      ! enforced here, so it is expected that INFO=0 always.
      AB(1,1) = XI(I)
      CALL EVAL_SPLINE(T(I1:I2+2*NCC), BCOEF(I1:I2), AB(1,1:1), INFO, D=DERIV)
      ! Check the precision of the reproduced value.
      ! Return an error if the precision is too low.
      IF (ABS( (AB(1,1) - FX(I,DERIV+1)) / &
           (1.0_R8 + ABS(FX(I,DERIV+1))) ) .GT. MAX_ERROR) THEN
         INFO = 7; RETURN
      END IF
   END DO
END DO

END SUBROUTINE FIT_SPLINE

SUBROUTINE EVAL_SPLINE(T, BCOEF, XY, INFO, D)
! Evaluate a spline constructed with FIT_SPLINE.
!
! INPUT:
!   T(1:NSPL+2*NCC) -- The nondecreasing real-valued locations of the
!      knots for the underlying B-splines, where the spline has NCC-1
!      continuous derivatives and is composed of NSPL B-splines.
!   BCOEF(1:NSPL) -- The coefficients of the B-spline basis functions
!      defining this interpolating spline.
```

```
!
! INPUT/OUTPUT:
!   XY(1:M) -- On input, the locations at which the spline is to be
!      evaluated; on output, holds the value (or Dth derivative) of
!      the spline with knots T and coefficients BCOEF evaluated at the
!      given locations.
!
! OUTPUT:
!   INFO -- Integer representing subroutine execution status.
!     0  Successful execution.
!     1  The sizes of T and BCOEF are incompatible.
!     2  Given the sizes of T and BCOEF, T is an invalid knot sequence.
!
! OPTIONAL INPUT:
!   D --  The order of the derivative of the spline to take at the
!     points in XY. If omitted, D = 0. When D < 0, the spline integral
!     over [T(1), XY(.)] is returned in XY(.).
!
! DESCRIPTION:
!
!   This subroutine serves as a convenient wrapper to the underlying
!   calls to EVAL_BSPLINE to evaluate the full spline. Internally this
!   evaluates the spline at each provided point XY(.) by first using a
!   bisection search to identify which B-splines could be nonzero at
!   that point, then computing the B-splines and taking a linear
!   combination of them to produce the spline value. Optimizations are
!   incorporated that make the evaluation of successive points in increasing
!   order most efficient, hence it is recommended that the provided points
!   XY(:) be increasing.
!
USE REAL_PRECISION, ONLY: R8
IMPLICIT NONE
! Arguments.
REAL(KIND=R8), INTENT(IN), DIMENSION(:) :: T, BCOEF
REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
INTEGER, INTENT(OUT) :: INFO
INTEGER, INTENT(IN), OPTIONAL :: D
! Local variables.
INTEGER :: DERIV, I, I1, I2, ITEMP, J, K, NB, NCC, NK, NSPL
REAL(KIND=R8), DIMENSION(SIZE(T)-SIZE(BCOEF)) :: BIATX
INTERFACE
   SUBROUTINE EVAL_BSPLINE(T, XY, D)
     USE REAL_PRECISION, ONLY: R8
     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T
     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
```

```
      INTEGER, INTENT(IN), OPTIONAL :: D
   END SUBROUTINE EVAL_BSPLINE
END INTERFACE
NK = SIZE(T) ! Number of knots.
NSPL = SIZE(BCOEF) ! Number of spline basis functions (B-splines).
! Check for size-related errors.
IF ( ((NK - NSPL)/2 .LT. 1) .OR. (MOD(NK - NSPL, 2) .NE. 0) )&
  THEN; INFO = 1; RETURN; ENDIF
! Compute the order for each B-spline (number of knots per B-spline minus one).
K = NK - NSPL ! = 2*NCC, where NCC = number of continuity conditions at each
   ! breakpoint.
NCC = K/2
NB = NSPL/NCC ! Number of breakpoints.
! Check for nondecreasing knot sequence.
DO I = 1, NK-1
   IF (T(I) .GT. T(I+1)) THEN; INFO = 2; RETURN; END IF
END DO
! Check for valid knot sequence for splines of order K.
DO I = 1, NK-K
   IF (T(I) .EQ. T(I+K)) THEN; INFO = 2; RETURN; END IF
END DO
! Assign the local value of the optional derivative argument D.
IF (PRESENT(D)) THEN; DERIV = D;  ELSE; DERIV = 0; END IF

! In the following code, I1 (I2, respectively) is the smallest (largest,
! respectively) index of a B-spline B_{I1}(.) (B_{I2}(.), respectively)
! whose support contains XY(I).  I1 = I2 + 1 - K .
!
! Initialize the indices I1 and I2 before looping.
I1 = 1; I2 = K ! For evaluation points in first breakpoint interval.
! Evaluate all the B-splines that have support at each point XY(I) in XY(:).
evaluate_at_x : DO I = 1, SIZE(XY)
   ! Return zero for points that are outside the spline's support.
   IF ((XY(I) .LT. T(1)) .OR. (XY(I) .GE. T(NK))) THEN
      XY(I) = 0.0_R8
      CYCLE evaluate_at_x
   ELSE IF ( (T(I2) .LE. XY(I)) .AND. (XY(I) .LT. T(I2+1)) ) THEN
      CONTINUE
   ELSE IF ( (I2 .NE. NB*NCC) .AND. (T(I2+NCC) .LE. XY(I)) .AND. &
      (XY(I) .LT. T(I2+NCC+1)) ) THEN
      I1 = I1 + NCC; I2 = I2 + NCC
   ELSE ! Find breakpoint interval containing XY(I) using a bisection method
        ! on the breakpoint indices.
      I1 = 1; I2 = NB
      DO WHILE (I2 - I1 .GT. 1)
```

```
         J = (I1+I2)/2 ! Breakpoint J = knot T((J+1)*NCC).
         IF ( T((J+1)*NCC) .LE. XY(I) ) THEN; I1 = J; ELSE; I2 = J; END IF
      END DO ! Now I2 = I1 + 1, and XY(I) lies in the breakpoint interval
             ! [ T((I1+1)*NCC), T((I1+2)*NCC) ).
      I2 = (I1 + 1)*NCC ! Spline index = knot index.
      I1 = I2 + 1 - K ! The index range of B-splines with support containing
                      ! XY(I) is I1 to I2.
   END IF
   ! Store only the single X value that is relevant to this iteration.
   BIATX(1:K) = XY(I) ! K = I2-I1+1.
   ! Compute the values of selected B-splines.
   ITEMP = 0
   DO J = I1, I2
      ITEMP = ITEMP + 1
      CALL EVAL_BSPLINE(T(J:J+K), BIATX(ITEMP:ITEMP), D=DERIV)
   END DO
   ! Evaluate spline interpolant at XY(I) as a linear combination of B-spline
   ! values, returning values in XY(:).
   XY(I) = DOT_PRODUCT(BIATX(1:K), BCOEF(I1:I2))
END DO evaluate_at_x
INFO = 0  ! Set INFO to indicate successful execution.
END SUBROUTINE EVAL_SPLINE

! ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!                        EVAL_BSPLINE.f90
!
! DESCRIPTION:
!   This file defines a subroutine EVAL_BSPLINE for computing the
!   value, integral, or derivative(s) of a B-spline given its knot
!   sequence.
!
! CONTAINS:
!   SUBROUTINE EVAL_BSPLINE(T, XY, D)
!     USE REAL_PRECISION, ONLY: R8
!     REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T
!     REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
!     INTEGER, INTENT(IN), OPTIONAL :: D
!   END SUBROUTINE EVAL_BSPLINE
!
! EXTERNAL DEPENDENCIES:
!   MODULE REAL_PRECISION
!     INTEGER, PARAMETER :: R8
!   END MODULE REAL_PRECISION
!
! CONTRIBUTORS:
```

```
!   Thomas C.H. Lux (tchlux@vt.edu)
!   Layne T. Watson (ltwatson@computer.org)
!   William I. Thacker (thackerw@winthrop.edu)
!
! VERSION HISTORY:
!   June 2020 -- (tchl) Created file, (ltw / wit) reviewed and revised.
!
SUBROUTINE EVAL_BSPLINE(T, XY, D)
! Subroutine for evaluating a B-spline with provided knot sequence, T.
! W A R N I N G : This routine has NO ERROR CHECKING and assumes
! informed usage for speed. It has undefined behavior for input that
! doesn't match specifications.
!
! INPUT:
!   T(1:N) -- The nondecreasing sequence of N knots for the B-spline.
!
! INPUT / OUTPUT:
!   XY(1:M) -- On input, the locations at which the B-spline is evaluated;
!     on output, holds the value of the Dth derivative of the B-spline
!     with prescribed knots evaluated at the given locations, or the
!     integral of the B-spline over [T(1), XY(.)] in XY(.).
!
! OPTIONAL INPUT:
!   D --  The order of the derivative to take of the B-spline. If omitted,
!     D = 0. When D < 0, this subroutine integrates the B-spline over each
!     interval [T(1), XY(.)].
!
! DESCRIPTION:
!
!   This function uses the recurrence relation defining a B-spline:
!
!   B_{I,1}(X) = {1, if T(I) <= X < T(I+1),
!                {0, otherwise,
!
!   where I is the spline index, J = 2, ..., N-MAX{D,0}-1 is the order, and
!
!                    X-T(I)                        T(I+J)-X
!   B_{I,J}(X) = ------------- B_{I,J-1}(X) + ------------- B_{I+1,J-1}(X).
!                T(I+J-1)-T(I)                 T(I+J)-T(I+1)
!
!   All the intermediate steps (J) are stored in a single block of
!   memory that is reused for each step.
!
!   The computation of the integral of the B-spline proceeds from
!   the above formula one integration step at a time by adding a
```

```
!   duplicate of the last knot, raising the order of all
!   intermediate B-splines, summing their values, and rescaling
!   each sum by the width of the supported interval divided by the
!   degree plus the integration coefficient.
!
!   For the computation of the derivative of the B-spline, the divided
!   difference definition of B_{I,J}(X) is used, building from J = N-D,
!   ..., N-1 via
!
!                        (J-1) B_{I,J-1}(X)     (J-1) B_{I+1,J-1}(X)
!   D/DX[B_{I,J}(X)] =  ------------------  -  --------------------.
!                         T(I+J-1) - T(I)         T(I+J) - T(I+1)
!
!   The final B-spline is right continuous and has support over the
!   interval [T(1), T(N)).
!
USE REAL_PRECISION, ONLY: R8
IMPLICIT NONE
! Arguments.
REAL(KIND=R8), INTENT(IN),    DIMENSION(:) :: T
REAL(KIND=R8), INTENT(INOUT), DIMENSION(:) :: XY
INTEGER, INTENT(IN), OPTIONAL :: D
! Local variables.
!  Iteration variables.
INTEGER :: I, J
!  Derivative being evaluated, order of B-spline K, one less than the
!  order L, and number of knots defining the B-spline N.
INTEGER :: DERIV, K, L, N
!  Evaluations of T constituent B-splines (columns) at all points (rows).
REAL(KIND=R8), DIMENSION(SIZE(XY), SIZE(T)) :: BIATX
!  Temporary storage for compute denominators LEFT, RIGHT, and last knot TN.
REAL(KIND=R8) :: LEFT, RIGHT, TN
! Assign the local value of the optional derivative "D" argument.
IF (PRESENT(D)) THEN
   DERIV = D
ELSE
   DERIV = 0
END IF
! Set local variables that are used for notational convenience.
N = SIZE(T) ! Number of knots.
K = N - 1 ! Order of B-spline.
L = K - 1 ! One less than the order of the B-spline.
TN = T(N) ! Value of the last knot, T(N).

! If this is a large enough derivative, we know it is zero everywhere.
```

```
IF (DERIV+1 .GE. N) THEN
   XY(:) = 0.0_R8
   RETURN

! ---------------- Performing standard evaluation ------------------
! This is a standard B-spline with multiple unique knots, right continuous.
ELSE
   ! Initialize all values to 0.
   BIATX(:,:) = 0.0_R8
   ! Assign the first value for each knot index.
   first_b_spline : DO J = 1, K
      IF (T(J) .EQ. T(J+1)) CYCLE first_b_spline
      ! Compute all right continuous order 1 B-spline values.
      WHERE ( (T(J) .LE. XY(:)) .AND. (XY(:) .LT. T(J+1)) )
         BIATX(:,J) = 1.0_R8
      END WHERE
   END DO first_b_spline
END IF
! Compute the remainder of B-spline by building up from the order one B-splines.
! Omit the final steps of this computation for derivatives.
compute_spline : DO I = 2, K-MAX(DERIV,0)
   ! Cycle over each knot accumulating the values for the recurrence.
   DO J = 1, N - I
      ! Check divisors, intervals with 0 width add 0 value to the B-spline.
      LEFT = (T(J+I-1) - T(J))
      RIGHT = (T(J+I) - T(J+1))
      ! Compute the B-spline recurrence relation (cases based on divisor).
      IF (LEFT .GT. 0) THEN
         IF (RIGHT .GT. 0) THEN
            BIATX(:,J) = &
                 ((XY(:) - T(J)) / LEFT) * BIATX(:,J) + &
                 ((T(J+I) - XY(:)) / RIGHT) * BIATX(:,J+1)
         ELSE
            BIATX(:,J) = ((XY(:) - T(J)) / LEFT) * BIATX(:,J)
         END IF
      ELSE
         IF (RIGHT .GT. 0) THEN
            BIATX(:,J) = ((T(J+I) - XY(:)) / RIGHT) * BIATX(:,J+1)
         END IF
      END IF
   END DO
END DO compute_spline

! -------------------- Performing integration ----------------------
int_or_diff : IF (DERIV .LT. 0) THEN
```

```
! Integrals will be 1 for TN <= X(J) < infinity.
WHERE (TN .LE. XY(:))
   BIATX(:,N) = 1.0_R8
END WHERE
! Currently the first B-spline in BIATX(:,1) has full order and covers
! all knots, but each following B-spline spans fewer knots (having
! lower order). This loop starts at the back of BIATX at the far right
! (order 1) constituent B-spline, raising the order of all constituent
! B-splines to match the order of the first by using the standard
! forward evaluation.
raise_order : DO I = 1, L
   DO J = N-I, K
      LEFT = (TN - T(J))
      RIGHT = (TN - T(J+1))
      IF (LEFT .GT. 0) THEN
         IF (RIGHT .GT. 0) THEN
            BIATX(:,J) = &
                 ((XY(:) - T(J)) / LEFT) * BIATX(:,J) + &
                 ((TN - XY(:)) / RIGHT) * BIATX(:,J+1)
         ELSE
            BIATX(:,J) = ((XY(:) - T(J)) / LEFT) * BIATX(:,J)
         END IF
      ELSE
         IF (RIGHT .GT. 0) THEN
            BIATX(:,J) = ((TN - XY(:)) / RIGHT) * BIATX(:,J+1)
         END IF
      END IF
   END DO
END DO raise_order

! Compute the integral of the B-spline.
! Do a forward evaluation of all constituents.
DO J = 1, K
   LEFT = (TN - T(J))
   RIGHT = (TN - T(J+1))
   IF (LEFT .GT. 0) THEN
      IF (RIGHT .GT. 0) THEN
         BIATX(:,J) = &
              ((XY(:) - T(J)) / LEFT) * BIATX(:,J) + &
              ((TN - XY(:)) / RIGHT) * BIATX(:,J+1)
      ELSE
         BIATX(:,J) = ((XY(:) - T(J)) / LEFT) * BIATX(:,J)
      END IF
   ELSE
      IF (RIGHT .GT. 0) THEN
```

```
         BIATX(:,J) = ((TN - XY(:)) / RIGHT) * BIATX(:,J+1)
      END IF
   END IF
END DO
! Sum the constituent functions at each knot (from the back).
DO J = K, 1, -1
   BIATX(:,J) = (BIATX(:,J) + BIATX(:,J+1))
END DO
! Divide by the degree plus one.
BIATX(:,1) = BIATX(:,1) / REAL(L+1,R8)
! Rescale the integral by its width.
BIATX(:,1) = BIATX(:,1) * (TN - T(1))

! ------------------ Performing differentiation --------------------
ELSE IF (DERIV .GT. 0) THEN
! Compute a derivative of the B-spline (if D > 0).
compute_derivative : DO J = N-DERIV, K
   ! Cycle over each knot just as for standard evaluation, however
   ! instead of using the recurrence relation for evaluating the value
   ! of the B-spline, use the recurrence relation for computing the
   ! value of the derivative of the B-spline.
   DO I = 1, N-J
      ! Assure that the divisor will not cause invalid computations.
      LEFT = (T(I+J-1) - T(I))
      RIGHT = (T(I+J) - T(I+1))
      ! Compute the derivative recurrence relation.
      IF (LEFT .GT. 0) THEN
        IF (RIGHT .GT. 0) THEN
          BIATX(:,I) = REAL(J-1,R8)*(BIATX(:,I)/LEFT-BIATX(:,I+1)/RIGHT)
        ELSE
          BIATX(:,I) = REAL(J-1,R8)*(BIATX(:,I)/LEFT)
        END IF
      ELSE
         IF (RIGHT .GT. 0) THEN
            BIATX(:,I) = REAL(J-1,R8)*(-BIATX(:,I+1)/RIGHT)
         END IF
      END IF
   END DO
END DO compute_derivative
END IF int_or_diff

! Assign the values to the output XY.
XY(:) = BIATX(:,1)

END SUBROUTINE EVAL_BSPLINE
```

Appendix C

Box Spline Subroutine

```
! ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!                          boxspline.f90
!
! DESCRIPTION:
!   This file contains the subroutine BOXSPLEV that can be used
!   to evaluate a box-spline, defined by its associated direction
!   vector set, at given evaluation points.
!
! CONTRIBUTORS:
!   Thomas C.H. Lux (tchlux@vt.edu)
!   Layne T. Watson (ltwatson@computer.org)
!
! VERSION HISTORY:
!   July 2018 -- (tchl) Created file, (ltw) reviewed and revised.
!
! ========================================================================
SUBROUTINE BOXSPLEV(UNIQUE_DVECS, DVEC_MULTS, EVAL_PTS, BOX_EVALS, ERROR)
  ! BOXSPLEV evaluates a box-spline, defined by a direction vector set
  ! in dimension S, at given evaluation points.
  !
  ! This implementation uses the numerically consistent algorithm for
  ! evaluating box-splines originally presented in [1]. Most notably,
  ! the evaluation of the box-spline near the boundaries of polynomial
  ! pieces does not exhibit the random behavior that is seen in the
  ! naive recursive implementation. As described in [1], the
  ! computational complexity for direction vector sets with repeated
  ! direction vectors is reduced from the naive recursive
  ! implementation complexity
  !     O( 2^{n - s} {n! / s!} ),
  ! where "n" is the number of direction vectors and "s" is the
  ! dimension, to
  !     O( (k)^{n - s} ),
  ! where "k" is the number of unique direction vectors, "n" is the
  ! total number of direction vectors, and "s" is the dimension.
  !
  ! The memory complexity of the implementation provided here is
```

```
!     O( s k + n m s + n^2 ),
! where "s" is the dimension, "k" is the number of unique direction
! vectors, "n" is the number of direction vectors, and "m" is the
! number of evaluation points. This memory complexity is achieved by
! not precomputing the 2^k normal vectors, which require O( 2^k s )
! space in [1].
!
! [1] Kobbelt, Leif. "Stable Evaluation of Box-Splines."
!     Numerical Algorithms 14.4 (1997): 377-382.
!
!
! The subroutine BOXSPLEV utilizes LAPACK routines DGECON, DGELS,
! DGESVD, DGETRF, and BLAS routine DGEMM.
!
! On input:
!
! UNIQUE_DVECS(:,:)
!    is a real S x M array whose columns are the unique direction
!    vectors used in defining the box-spline.  S <= M, and
!    UNIQUE_DVECS(1:S,1:S) must be invertible.
!
! DVEC_MULTS(:)
!    is an integer array of length M containing the multiplicity
!    of each corresponding direction vector.
!
! EVAL_PTS(:,:)
!    is a real S x L array whose columns are the points at which
!    the box-spline is to be evaluated.
!
! On output:
!
! BOX_EVALS(:)
!    is a real array of length L containing the box-spline
!    values at the evaluation points.
!
! ERROR
!    is an integer error flag of the form
!              100*(LAPACK INFO flag) + 10*T + U.
!    ERROR .EQ. 0 is a normal return.  For ERROR .NE. 0, the
!    meanings of the tens digit T and the units digit U are:
!
!    Tens digit T:
!      0  Improper usage.
!      1  Error computing box-spline.
!      2  Error computing a minimum norm representation of direction vectors.
```

```
!      3  Error copying direction vectors with nonzero multiplicity.
!      4  Error computing an orthogonal vector.
!      5  Error computing the reciprocal condition number of matrix.
!      6  Error computing a matrix rank.
!      7  Error computing a matrix determinant.
!      8  Error preparing memory.
!
!    Units digit U, for T = 0:
!      1  Mismatched dimension, SIZE(DVEC_MULTS) .NE. SIZE(UNIQUE_DVECS,2).
!      2  Mismatched dimension, SIZE(EVAL_PTS,1) .NE. SIZE(UNIQUE_DVECS,1).
!      3  Mismatched dimension, SIZE(BOX_EVALS)  .NE. SIZE(EVAL_PTS,1).
!      4  One of the multiplicity values provided was < 1.
!      5  Column vectors of UNIQUE_DVECS are not unique.
!      6  M < S or UNIQUE_DVECS(1:S,1:S) is near singular.
!
!    Units digit U, for T /= 0:
!      0  Work array allocation failed.
!      1  Work array size query failed.
!      2  DGELS computation error, see 'INFO' for details.
!      3  DGESVD computation error, see 'INFO' for details.
!      4  DGETRF computation error, see 'INFO' for details.
!      5  DGECON computation error, see 'INFO' for details.
!
! ------------------------------------------------------------------
! The calling program should include the following interface to BOXSPLEV:
!
! INTERFACE
!   SUBROUTINE BOXSPLEV(UNIQUE_DVECS, DVEC_MULTS, EVAL_PTS, BOX_EVALS, ERROR)
!     USE REAL_PRECISION, ONLY: REAL64
!     REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:) :: UNIQUE_DVECS
!     INTEGER,           INTENT(IN),  DIMENSION(:)   :: DVEC_MULTS
!     REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:) :: EVAL_PTS
!     REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:)   :: BOX_EVALS
!     INTEGER,           INTENT(OUT)                 :: ERROR
!   END SUBROUTINE BOXSPLEV
! END INTERFACE
! ------------------------------------------------------------------
!
USE ISO_FORTRAN_ENV, ONLY : REAL64
IMPLICIT NONE
REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:) :: UNIQUE_DVECS
INTEGER,           INTENT(IN),  DIMENSION(:)   :: DVEC_MULTS
REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:) :: EVAL_PTS
REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:)   :: BOX_EVALS
INTEGER,           INTENT(OUT)                 :: ERROR
```

```
! Reusable recursion variables and global variables.
INTEGER :: D, NUM_DVECS, NUM_PTS, DEPTH, INFO
REAL(KIND=REAL64) :: POSITION
REAL(KIND=REAL64), PARAMETER :: SQRTEPS = SQRT(EPSILON(1.0_REAL64))
REAL(KIND=REAL64), PARAMETER :: ONE  = 1.0_REAL64
REAL(KIND=REAL64), PARAMETER :: ZERO = 0.0_REAL64
INTEGER,           PARAMETER :: BLOCK_SIZE = 32
! ----------------------------------------------------------------
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,2), &
     SIZE(UNIQUE_DVECS,2)) :: IDENTITY
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,1), &
     SIZE(UNIQUE_DVECS,1)) :: DET_WORK
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,2), &
     SIZE(UNIQUE_DVECS,2)) :: ORTHO_MIN_WORK
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1)-1,&
     SIZE(UNIQUE_DVECS,1)) :: TRANS_DVECS
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1),  &
     SIZE(EVAL_PTS,2)      :: TEMP_EVAL_PTS
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1)) :: PT_SHIFT
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,2)) :: PT_SHIFT_R
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1)) :: SING_VALS
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1)) :: NORMAL_VECTOR
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,2)) :: REMAINING_DVECS
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,2)) :: NONZERO_DVECS
REAL(KIND=REAL64), DIMENSION(&
     SIZE(UNIQUE_DVECS,1)*SIZE(UNIQUE_DVECS,2)*(1+BLOCK_SIZE)) :: LAPACK_WORK
! Unique recursion variables for evaluating box-spline. Last dimension is depth.
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,1),SIZE(UNIQUE_DVECS,2), &
     SUM(DVEC_MULTS)-SIZE(UNIQUE_DVECS,1)+2) :: DVECS
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,2),                      &
     SUM(DVEC_MULTS)-SIZE(UNIQUE_DVECS,1)+2) :: LOC
INTEGER,           DIMENSION(SIZE(UNIQUE_DVECS,2),                      &
     SUM(DVEC_MULTS)-SIZE(UNIQUE_DVECS,1)+1) :: MULTS
REAL(KIND=REAL64), DIMENSION(SIZE(EVAL_PTS,2),                          &
     SUM(DVEC_MULTS)-SIZE(UNIQUE_DVECS,1)+2) :: EVALS_AT_PTS
REAL(KIND=REAL64), DIMENSION(SIZE(UNIQUE_DVECS,2),SIZE(EVAL_PTS,2),     &
     SUM(DVEC_MULTS)-SIZE(UNIQUE_DVECS,1)+2) :: SHIFTED_EVAL_PTS
! ----------------------------------------------------------------
! Local variables for preparation.
INTEGER :: IDX_1, IDX_2

! Initialize error and info flags.
ERROR = 0; INFO = 0;
! Initialize parameters
! Store 'global' constants for box-spline evaluation.
```

```
D          = SIZE(UNIQUE_DVECS, 1)
NUM_DVECS = SIZE(UNIQUE_DVECS, 2)
NUM_PTS    = SIZE(EVAL_PTS, 2)
! Check for usage errors (dimension mismatches, invalid multiplicity)
IF (SIZE(DVEC_MULTS)   .NE. NUM_DVECS) THEN; ERROR = 1; RETURN; END IF
IF (SIZE(EVAL_PTS,1)   .NE. D)         THEN; ERROR = 2; RETURN; END IF
IF (SIZE(BOX_EVALS)    .NE. NUM_PTS)   THEN; ERROR = 3; RETURN; END IF
IF (MINVAL(DVEC_MULTS) .LT. 1)         THEN; ERROR = 4; RETURN; END IF
! Check uniqueness of DVECS columns.
DO IDX_1 = 1, NUM_DVECS-1
   DO IDX_2 = IDX_1+1, NUM_DVECS
      IF (SUM(ABS(UNIQUE_DVECS(:,IDX_1) - UNIQUE_DVECS(:,IDX_2))) &
           .LT. SQRTEPS) THEN; ERROR = 5; RETURN
      END IF
   END DO
END DO
! Check if NUM_DVECS < D or rank DVECS(1:D,1:D) < D.
IF (NUM_DVECS .LT. D) THEN
   ERROR = 6; RETURN
ELSE
   ! Compute condition number COND(DVECS(1:D,1:D)) and test for near
   ! rank deficiency: 1/COND(DVECS(1:D,1:D)) < SQRT(EPSILON(1.0_REAL64)).
   IF (MATRIX_CONDITION_INV(UNIQUE_DVECS(1:D, 1:D)) .LT. SQRTEPS) THEN
      ERROR = 6; RETURN
   ENDIF
ENDIF
! Allocate a work array large enough for all LAPACK calls.
IF (ERROR .NE. 0) THEN; RETURN; END IF
! Create an identity matrix (for easy matrix-vector multiplication).
IDENTITY = 0
FORALL (IDX_1 = 1:NUM_DVECS) IDENTITY(IDX_1,IDX_1) = 1
! Calculate the DVECS for the beginning of box-spline evaluation.
DEPTH = 0; DVECS(:,:,1) = UNIQUE_DVECS(:,:);
CALL MAKE_DVECS_MIN_NORM(DVECS(:,:,1))
IF (ERROR .NE. 0) THEN; RETURN; END IF
! Initialize the multiplicities and location.
MULTS(:,1) = DVEC_MULTS(:); LOC(:,1) = 0;
! Compute the shifted evaluation points.
CALL DGEMM('T', 'N', SIZE(DVECS,2), SIZE(EVAL_PTS,2), SIZE(DVECS,1), &
     ONE, DVECS(:,:,1), SIZE(DVECS,1), EVAL_PTS, SIZE(EVAL_PTS,1), &
     ZERO, SHIFTED_EVAL_PTS(:,:,1), SIZE(SHIFTED_EVAL_PTS,1))
! Recursively evaluate the box-spline.
CALL EVALUATE_BOX_SPLINE(&
     DVECS(:,:,1), LOC(:,1), MULTS(:,1), SHIFTED_EVAL_PTS(:,:,1), &
     EVALS_AT_PTS(:,1), &
```

```
      DVECS(:,:,2), LOC(:,2), MULTS(:,2), SHIFTED_EVAL_PTS(:,:,2), &
      EVALS_AT_PTS(:,2))
  IF (ERROR .NE. 0) THEN; RETURN; END IF
  BOX_EVALS(:) = EVALS_AT_PTS(:,1)
CONTAINS

  ! ================================================================
  RECURSIVE SUBROUTINE EVALUATE_BOX_SPLINE(&
      R_DVECS, R_LOC, R_MULTS, R_SHIFTED_EVAL_PTS, R_EVALS_AT_PTS, &
      R_NEXT_DVECS, R_NEXT_LOC, R_NEXT_MULTS, R_NEXT_SHIFTED_EVAL_PTS, &
      R_NEXT_EVALS_AT_PTS)
    ! 1) EVALUATE_BOX_SPLINE:
    !
    !   Evaluate the box-spline defined by "DVECS" recursively, where
    !   this iteration handles the remaining direction vectors, at all
    !   points in "SHIFTED_EVAL_PTS" and store box-spline evaluations
    !   in "EVALS_AT_PTS".
    !
    REAL(KIND=REAL64), INTENT(IN), DIMENSION(:,:) :: R_DVECS
    INTEGER,           INTENT(IN), DIMENSION(:)   :: R_LOC, R_MULTS
    REAL(KIND=REAL64), INTENT(IN), DIMENSION(:,:) :: R_SHIFTED_EVAL_PTS
    REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:,:) :: R_NEXT_DVECS
    INTEGER,           INTENT(OUT), DIMENSION(:)   :: R_NEXT_LOC, R_NEXT_MULTS
    REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:,:) :: R_NEXT_SHIFTED_EVAL_PTS
    REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:)   :: R_EVALS_AT_PTS
    REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:)   :: R_NEXT_EVALS_AT_PTS
    ! Local variables
    INTEGER :: IDX_1, IDX_2, IDX_3, TEMP_NUM_DVECS_1, TEMP_NUM_DVECS_2
    ! Adjust the global variable "DEPTH" to use appropriate memory slices.
    DEPTH = DEPTH + 1
    ! Recursion case ...
    IF (SUM(R_MULTS) > D) THEN
       R_EVALS_AT_PTS(:) = 0._REAL64
       IDX_2 = 1
       ! Sum over all direction vectors.
       DO IDX_1 = 1, SIZE(MULTS,1)
          ! Update multiplicity of directions and position in recursion tree.
          R_NEXT_MULTS(:) = R_MULTS(:) - IDENTITY(:,IDX_1) ! Reduce multiplicity.
          ! Make recursive calls.
          IF (R_MULTS(IDX_1) .GT. 1) THEN
             ! Copy LOC, DVECS, and SHIFTED_EVAL_PTS down one level
             ! for next recursion.
             R_NEXT_LOC(:) = R_LOC(:)
             ! Pass the DVECS down
             R_NEXT_DVECS(:,:NUM_DVECS) = R_DVECS(:,:NUM_DVECS)
```

```
      R_NEXT_SHIFTED_EVAL_PTS(:,:) = R_SHIFTED_EVAL_PTS(:,:)
      ! Perform recursion with only reduced multiplicity.
      CALL EVALUATE_BOX_SPLINE( &
           R_NEXT_DVECS, R_NEXT_LOC, R_NEXT_MULTS, &
           R_NEXT_SHIFTED_EVAL_PTS, R_NEXT_EVALS_AT_PTS, &
           DVECS(:,:,DEPTH+2), LOC(:,DEPTH+2), MULTS(:,DEPTH+2), &
           SHIFTED_EVAL_PTS(:,:,DEPTH+2), EVALS_AT_PTS(:,DEPTH+2))
      IF (ERROR .NE. 0) RETURN
      R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) + &
           R_NEXT_EVALS_AT_PTS(:) * R_SHIFTED_EVAL_PTS(IDX_2,:)
      ! Perform recursion with transformed set of direction
      ! vectors and evaluation points. (store at next 'DEPTH')
      PT_SHIFT_R(:) = MATMUL(UNIQUE_DVECS(:,IDX_1), R_DVECS(:,:))
      compute_shift_1 : DO IDX_3 = 1, NUM_PTS
         R_NEXT_SHIFTED_EVAL_PTS(:,IDX_3) = &
              R_SHIFTED_EVAL_PTS(:,IDX_3) - PT_SHIFT_R(:)
      END DO compute_shift_1
      ! Update location for next level of recursion.
      R_NEXT_LOC(:) = R_LOC(:) + IDENTITY(:,IDX_1)
      CALL EVALUATE_BOX_SPLINE( &
           R_NEXT_DVECS, R_NEXT_LOC, R_NEXT_MULTS, &
           R_NEXT_SHIFTED_EVAL_PTS, R_NEXT_EVALS_AT_PTS, &
           DVECS(:,:,DEPTH+2), LOC(:,DEPTH+2), MULTS(:,DEPTH+2), &
           SHIFTED_EVAL_PTS(:,:,DEPTH+2), EVALS_AT_PTS(:,DEPTH+2))
      IF (ERROR .NE. 0) RETURN
      R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) + &
           R_NEXT_EVALS_AT_PTS(:) * &
           (R_MULTS(IDX_1) - R_SHIFTED_EVAL_PTS(IDX_2,:))
      IDX_2 = IDX_2 + 1
   ELSE IF (R_MULTS(IDX_1) .GT. 0) THEN
      TEMP_NUM_DVECS_1 = NUM_DVECS
      ! Get the remaining direction vectors.
      CALL PACK_DVECS(R_NEXT_MULTS, UNIQUE_DVECS, R_NEXT_DVECS)
      IF (ERROR .NE. 0) RETURN
      ORTHO_MIN_WORK(:NUM_DVECS,:D) = TRANSPOSE(R_NEXT_DVECS(:,:NUM_DVECS))
      IF (MATRIX_RANK(ORTHO_MIN_WORK(:NUM_DVECS,:D)) .EQ. D) THEN
         IF (ERROR .NE. 0) RETURN
         ! Store the number of direction vectors at this level before
            recursion.
         TEMP_NUM_DVECS_2 = NUM_DVECS
         ! Assign location for next recursion level to be unchanged.
         R_NEXT_LOC(:) = R_LOC(:)
         ! Update Least norm representation of the direction vectors.
         CALL MAKE_DVECS_MIN_NORM(R_NEXT_DVECS(:,:NUM_DVECS))
         IF (ERROR .NE. 0) RETURN
```

```
      ! Perform recursion with only reduced multiplicity.
      PT_SHIFT(:) = MATMUL(UNIQUE_DVECS(:,:), R_LOC(:))
      compute_shift_2 : DO IDX_3 = 1, NUM_PTS
         TEMP_EVAL_PTS(:,IDX_3) = EVAL_PTS(:,IDX_3) - PT_SHIFT(:)
      END DO compute_shift_2
      ! Next dvecs x temp eval points
      CALL DGEMM('T', 'N', NUM_DVECS, SIZE(EVAL_PTS,2), SIZE(DVECS,1), &
           ONE, R_NEXT_DVECS(:,:NUM_DVECS), SIZE(DVECS,1), &
           TEMP_EVAL_PTS, SIZE(EVAL_PTS,1), ZERO, &
           R_NEXT_SHIFTED_EVAL_PTS(:NUM_DVECS,:), NUM_DVECS)
      ! Perform recursion with only reduced multiplicity.
      CALL EVALUATE_BOX_SPLINE( &
           R_NEXT_DVECS, R_NEXT_LOC, R_NEXT_MULTS, &
           R_NEXT_SHIFTED_EVAL_PTS, R_NEXT_EVALS_AT_PTS, &
           DVECS(:,:,DEPTH+2), LOC(:,DEPTH+2), MULTS(:,DEPTH+2), &
           SHIFTED_EVAL_PTS(:,:,DEPTH+2), EVALS_AT_PTS(:,DEPTH+2))
      IF (ERROR .NE. 0) RETURN
      R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) + &
           R_NEXT_EVALS_AT_PTS(:) * R_SHIFTED_EVAL_PTS(IDX_2,:)
      ! Update location for next level of recursion.
      R_NEXT_LOC(:) = R_LOC(:) + IDENTITY(:,IDX_1)
      PT_SHIFT(:) = MATMUL(UNIQUE_DVECS(:,:), R_NEXT_LOC(:))
      compute_shift_3 : DO IDX_3 = 1, NUM_PTS
         TEMP_EVAL_PTS(:,IDX_3) = EVAL_PTS(:,IDX_3) - PT_SHIFT(:)
      END DO compute_shift_3
      ! Recalculate "NUM_DVECS" since it was destroyed in recursion.
      NUM_DVECS = TEMP_NUM_DVECS_2
      ! Next dvecs x temp eval points.
      CALL DGEMM('T', 'N', NUM_DVECS, SIZE(EVAL_PTS,2), SIZE(DVECS,1), &
           ONE, R_NEXT_DVECS(:,:NUM_DVECS), SIZE(DVECS,1), TEMP_EVAL_PTS, &
           SIZE(EVAL_PTS,1), ZERO, R_NEXT_SHIFTED_EVAL_PTS(:NUM_DVECS,:), &
           NUM_DVECS)
      ! Perform recursion with transformed set of direction vectors.
      CALL EVALUATE_BOX_SPLINE( &
           R_NEXT_DVECS, R_NEXT_LOC, R_NEXT_MULTS, &
           R_NEXT_SHIFTED_EVAL_PTS, R_NEXT_EVALS_AT_PTS, &
           DVECS(:,:,DEPTH+2), LOC(:,DEPTH+2), MULTS(:,DEPTH+2), &
           SHIFTED_EVAL_PTS(:,:,DEPTH+2), EVALS_AT_PTS(:,DEPTH+2))
      IF (ERROR .NE. 0) RETURN
      R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) + &
           R_NEXT_EVALS_AT_PTS(:) * &
           (R_MULTS(IDX_1) - R_SHIFTED_EVAL_PTS(IDX_2,:))
   END IF
   ! Reset "NUM_DVECS" after executing the above steps.
   NUM_DVECS = TEMP_NUM_DVECS_1
```

```
         IDX_2 = IDX_2 + 1
      END IF
   END DO
   ! Normalize by number of direction vectors in computation.
   R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) / &
        REAL(SUM(R_MULTS(:)) - D, REAL64)
ELSE
   ! Base case ... compute characteristic function.
   R_EVALS_AT_PTS(:) = 1.0_REAL64
   ! Pack the unique direction vectors into the memory location
   ! for the current set of direction vectors (since the 'current'
   ! are not needed for base case evaluation).
   CALL PACK_DVECS(R_MULTS(:), UNIQUE_DVECS, R_NEXT_DVECS(:,:))
   IF (ERROR .NE. 0) RETURN
   ! Delayed translations (this is what makes the algorithm more stable).
   PT_SHIFT(:) = MATMUL(UNIQUE_DVECS(:,:), R_LOC(:))
   compute_shift_4 : DO IDX_1 = 1, NUM_PTS
      R_NEXT_SHIFTED_EVAL_PTS(:D,IDX_1) = EVAL_PTS(:,IDX_1) - PT_SHIFT(:)
   END DO compute_shift_4
   ! Check evaluation point locations against all remaining direction vectors.
   DO IDX_1 = 1, D
      ! Get the active set of direction vectors (exluding current vector).
      TRANS_DVECS(:,:) = TRANSPOSE(R_NEXT_DVECS(:,:D-1))
      IF (IDX_1 .LT. D) TRANS_DVECS(IDX_1,:) = R_NEXT_DVECS(:,D)
      ! Calculate the orthogonal vector to the remaining direction vectors.
      CALL COMPUTE_ORTHOGONAL(TRANS_DVECS, NORMAL_VECTOR)
      IF (ERROR .NE. 0) RETURN
      ! Compute shifted position (relative to normal vector).
      POSITION = DOT_PRODUCT(R_NEXT_DVECS(:,IDX_1), NORMAL_VECTOR(:))
      ! Compute shifted evaluation locations. (1, D) x (D, NUM_PTS)
      R_NEXT_EVALS_AT_PTS(:) = MATMUL(NORMAL_VECTOR, &
           R_NEXT_SHIFTED_EVAL_PTS(:D,:))
      ! Identify those points that are outside of this box (0-side).
      IF (POSITION .GT. 0) THEN
         WHERE (R_NEXT_EVALS_AT_PTS(:) .LT. 0) R_EVALS_AT_PTS(:) = 0._REAL64
      ELSE IF (POSITION .LT. 0) THEN
         WHERE (R_NEXT_EVALS_AT_PTS(:) .GE. 0) R_EVALS_AT_PTS(:) = 0._REAL64
      END IF
      ! Recompute shifted location (other side of box) based
      ! on selected direction vector.
      compute_shifted_point_2 : DO IDX_2 = 1, NUM_PTS
         R_NEXT_EVALS_AT_PTS(IDX_2) = DOT_PRODUCT( &
              EVAL_PTS(:,IDX_2) - PT_SHIFT(:) - R_NEXT_DVECS(:,IDX_1), &
              NORMAL_VECTOR(:))
      END DO compute_shifted_point_2
```

```
        ! Identify those shifted points that are outside of this box
        ! on REMAINING_DVEC(IDX_1)-side.
        IF (POSITION .GT. 0) THEN
           WHERE (R_NEXT_EVALS_AT_PTS(:) .GE. 0) R_EVALS_AT_PTS(:) = 0._REAL64
        ELSE IF (POSITION .LT. 0) THEN
           WHERE (R_NEXT_EVALS_AT_PTS(:) .LT. 0) R_EVALS_AT_PTS(:) = 0._REAL64
        END IF
      END DO
      ! Normalize evaluations by determinant of box.
      R_EVALS_AT_PTS(:) = R_EVALS_AT_PTS(:) / &
           ABS(MATRIX_DET(TRANSPOSE(R_NEXT_DVECS(:,:D))))
      IF (ERROR .NE. 0) RETURN
    END IF
    DEPTH = DEPTH - 1
  END SUBROUTINE EVALUATE_BOX_SPLINE

  !=====================================================================
  !             Supporting code for computing box-spline
  !=====================================================================

  ! ====================================================================
  SUBROUTINE MAKE_DVECS_MIN_NORM(DVECS)
    ! 2) MAKE_DVECS_MIN_NORM
    !
    !   Compute the minimum norm representation of 'MATRIX' and store
    !   it in 'MIN_NORM', use DGELS to find the least squares
    !   solution to the problem (AX = I). This is a more numerically
    !   stable solution to the linear system (A^T A) X = A^T.
    REAL(KIND=REAL64), INTENT(INOUT), DIMENSION(:,:) :: DVECS
    INTEGER :: IDX
    ! Make "ORTHO_MIN_WORK" the identity matrix
    ORTHO_MIN_WORK(:,:) = 0._REAL64
    FORALL (IDX = 1:NUM_DVECS) ORTHO_MIN_WORK(IDX,IDX) = 1.0_REAL64
    ! Call DGELS for actual solve.
    CALL DGELS('T', SIZE(DVECS,1), SIZE(DVECS,2), SIZE(UNIQUE_DVECS,2),&
         DVECS, SIZE(DVECS,1), ORTHO_MIN_WORK, &
         SIZE(ORTHO_MIN_WORK,1), LAPACK_WORK, SIZE(LAPACK_WORK), INFO)
    ! Check for error.
    IF (INFO .NE. 0) THEN; ERROR = 100*INFO + 22; RETURN; END IF
    ! Extract the minimum norm representation from the output of DGELS.
    DVECS(:,:) = ORTHO_MIN_WORK(:SIZE(DVECS,1),:SIZE(DVECS,2))
  END SUBROUTINE MAKE_DVECS_MIN_NORM

  ! ====================================================================
  SUBROUTINE PACK_DVECS(MULTS, SRCE_DVECS, DEST_DVECS)
```

```
  ! 3) PACK_DVECS
  !
  !   Given multiplicities and source direction vectors, pack all
  !   of the nonzero-multiplicity source direction vectors into the
  !   storage location provided for remaining direction vectors.
  !   Update global variable "NUM_DVECS" in the process.
  !
  ! Inputs:
  !   MULTS(:)        -- Integer array of multiplicities.
  !   SRCE_DVECS(:,:) -- Real dense matrix of (source) direction
  !                      vectors.
  !   DEST_DVECS(:,:) -- Real dense matrix for storing direction
  !                      vectors.
  !
  INTEGER,           INTENT(IN),  DIMENSION(:)   :: MULTS
  REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:) :: SRCE_DVECS
  REAL(KIND=REAL64), INTENT(OUT), DIMENSION(:,:) :: DEST_DVECS
  INTEGER :: IDX
  NUM_DVECS = 0
  DO IDX = 1, SIZE(MULTS)
     IF (MULTS(IDX) .GT. 0) THEN
        NUM_DVECS = NUM_DVECS + 1
        DEST_DVECS(:,NUM_DVECS) = SRCE_DVECS(:,IDX)
     END IF
  END DO
END SUBROUTINE PACK_DVECS

! ======================================================================
SUBROUTINE COMPUTE_ORTHOGONAL(A, ORTHOGONAL)
  ! 4) COMPUTE_ORTHOGONAL
  !
  !   Given a matrix A of row vectors, compute a vector orthogonal to
  !   the row vectors in A and store it in ORTHOGONAL using DGESVD.
  !   If there any near-zero singular values are identified, the
  !   vector associated with the smallest such singular values is
  !   returned.
  !
  ! Input:
  !   A(:,:) -- Real dense matrix.
  !
  ! Output:
  !   ORTHOGONAL(:) -- Real vector orthogonal to given matrix.
  !
  REAL(KIND=REAL64), INTENT(IN),  DIMENSION(:,:)       :: A
  REAL(KIND=REAL64), INTENT(OUT), DIMENSION(SIZE(A,2)) :: ORTHOGONAL
```

```
  ! Local variables.
  REAL(KIND=REAL64) :: TOL
  REAL(KIND=REAL64), DIMENSION(1) :: U
  ! Use the SVD to get the orthogonal vector(s).
  CALL DGESVD('N', 'A', SIZE(A,1), SIZE(A,2), A, SIZE(A,1), &
       SING_VALS, U, SIZE(U), ORTHO_MIN_WORK(:SIZE(A,2),:SIZE(A,2)), &
       SIZE(A,2), LAPACK_WORK, SIZE(LAPACK_WORK), INFO)
  IF (INFO .NE. 0) THEN; ERROR = 100*INFO + 43; RETURN; END IF
  ! Compute the appropriate tolerance based on calculations.
  TOL = EPSILON(1.0_REAL64) * MAXVAL(SHAPE(A)) * MAXVAL(SING_VALS)
  ORTHOGONAL(:) = 0._REAL64
  ! Check for a vector in the orthonormal basis for the null
  ! space of A (a vector with an extremely small singular value).
  IF (SING_VALS(SIZE(SING_VALS)-1) .LE. TOL) THEN
     ORTHOGONAL = ORTHO_MIN_WORK(SIZE(SING_VALS)-1,:SIZE(A,2))
  ! If no orthogonal vector was found and the matrix does not contain
  ! enough vectors to span the space, use the first vector in VT at
  ! (RANK(A) + 1).
  ELSE IF (SIZE(A,2) > SIZE(SING_VALS)-1) THEN
     ORTHOGONAL = ORTHO_MIN_WORK(SIZE(SING_VALS),:SIZE(A,2))
  END IF
END SUBROUTINE COMPUTE_ORTHOGONAL

! ======================================================================
FUNCTION MATRIX_CONDITION_INV(MATRIX) RESULT(RCOND)
  ! 5) MATRIX_CONDITION_INV
  !
  !   Compute the condition number (for testing near rank deficiency)
  !   using DGECON (which computes 1 / CONDITION).
  !
  ! Input:
  !   MATRIX(:,:) -- Real dense matrix.
  !
  ! Output:
  !   RCOND -- Real value corresponding to the reciprocal of the
  !            condition number of the provided matrix.
  REAL(KIND=REAL64), INTENT(IN), DIMENSION(:,:) :: MATRIX
  REAL(KIND=REAL64) :: RCOND
  ! Local arrays for work.
  REAL(KIND=REAL64), DIMENSION(4*SIZE(MATRIX,2)) :: WORK
  INTEGER,           DIMENSION(  SIZE(MATRIX,2)) :: IWORK
  ! Use LAPACK to compue recirpocal of matrix codition number.
  CALL DGECON('I', SIZE(MATRIX,2), MATRIX, SIZE(MATRIX,1),&
       SUM(ABS(MATRIX)), RCOND, WORK, IWORK, INFO)
  ! Check for errors during exeuction.
```

```
  IF (INFO .NE. 0) ERROR = 100 * INFO + 55
END FUNCTION MATRIX_CONDITION_INV

! ========================================================================
FUNCTION MATRIX_RANK(MATRIX)
  ! 6) MATRIX_RANK
  !
  !   Get the rank of the provided matrix using the SVD.
  !
  ! Input:
  !   MATRIX(:,:) -- Real dense matrix.
  !
  ! Output:
  !   MATRIX_RANK -- The integer rank of MATRIX.
  !
  REAL(KIND=REAL64), INTENT(IN), DIMENSION(:,:) :: MATRIX
  ! Local variables for computing the orthogonal vector.
  INTEGER :: IDX, MATRIX_RANK
  REAL(KIND=REAL64) :: TOL
  ! Unused DGESVD parameters.
  REAL(KIND=REAL64), DIMENSION(1) :: U, VT
  ! Early return if possible, when we know the rank must be low.
  IF (SIZE(MATRIX,1) .LT. SIZE(SING_VALS)) THEN; MATRIX_RANK = 0; RETURN; END IF
  ! Use the SVD to get the orthogonal vectors.
     CALL DGESVD('N', 'N', SIZE(MATRIX,1), SIZE(MATRIX,2), MATRIX, &
          SIZE(MATRIX,1), SING_VALS, U, SIZE(U), VT, SIZE(VT), &
          LAPACK_WORK, SIZE(LAPACK_WORK), INFO)
  IF (INFO .NE. 0) THEN; ERROR = 100*INFO + 63; RETURN; END IF
  ! Compute a reasonable singular value tolerance based
  ! on expected numerical error.
  TOL = EPSILON(1.0_REAL64) * MAXVAL(SHAPE(MATRIX)) * MAXVAL(SING_VALS)
  ! Find the first near-0 singular value starting from smallest
  ! value, assumes high rank is more likely than low rank.
  find_null : DO IDX = SIZE(SING_VALS), 1, -1
     IF (SING_VALS(IDX) .GT. TOL) THEN
        MATRIX_RANK = IDX
        EXIT find_null
     END IF
  END DO find_null
END FUNCTION MATRIX_RANK

! ========================================================================
FUNCTION MATRIX_DET(MATRIX)
  ! 7) MATRIX_DET
  !
```

```
  !   Compute the determinant of a matrix using the LU decomposition.
  !
  ! Input:
  !   MATRIX(:,:) -- Real dense matrix.
  !
  ! Output:
  !   MATRIX_DET -- The real-valued determinant of MATRIX.
  !
  REAL(KIND=REAL64), INTENT(IN), DIMENSION(:,:) :: MATRIX
  REAL(KIND=REAL64) :: MATRIX_DET
  INTEGER :: IDX
  ! Do the LU decomposition.
  CALL DGETRF(SIZE(MATRIX,1), SIZE(MATRIX,2), MATRIX, &
       SIZE(MATRIX,1), DET_WORK, INFO)
  ! Check for errors.
  IF (INFO .NE. 0) THEN
     MATRIX_DET = 1.0_REAL64 ! <- Set a value that will not break caller.
     ERROR = 100*INFO + 74
     RETURN
  END IF
  ! Compute the determinant (product of diagonal of U).
  MATRIX_DET = 1.
  DO IDX = 1, MIN(SIZE(MATRIX,1), SIZE(MATRIX,2))
     MATRIX_DET = MATRIX_DET * MATRIX(IDX,IDX)
  END DO
END FUNCTION MATRIX_DET

! ======================================================================
SUBROUTINE PRINT_OPTIMAL_BLOCK_SIZE()
  ! 8) PRINT_OPTIMAL_BLOCK_SIZE
  !
  !   Use the LAPACK routine ILAENV to print out the optimal block
  !   size for the current architecture (requires optimal LAPACK and
  !   BLAS installation).
  !
  INTEGER :: BLOCK_SIZE
  INTERFACE
     FUNCTION ILAENV(ISPEC, NAME, OPTS, N1, N2, N3, N4)
       INTEGER :: ISPEC
       CHARACTER :: NAME, OPTS
       INTEGER, OPTIONAL :: N1,N2,N3,N4
       INTEGER :: ILAENV
     END FUNCTION ILAENV
  END INTERFACE
  !
```

```
    ! Compute storage needed by each of the LAPACK routines.
    ! DGELS optimal work array size:
    !   D * NUM_DVECS * (1 +  BLOCK_SIZE)
    ! DGESVD optimal work array size:
    !   MAX(3*D + NUM_DVECS, 5*D)
    !
    ! BLOCK_SIZE = 65536 ! <- 64KB
    ! BLOCK_SIZE = 8192  ! <- 8KB
    !
    BLOCK_SIZE = ILAENV(1, 'DGELS', 'T', D, NUM_DVECS, NUM_DVECS)
    WRITE (*,'(A,I6)') "BLOCK SIZE:", BLOCK_SIZE
  END SUBROUTINE PRINT_OPTIMAL_BLOCK_SIZE

END SUBROUTINE BOXSPLEV
```