UCRL-TR-235618

LAWRENCE LIVERMORE NATIONAL LABORATORY

Computer Experiments for Function Approximations

A. Chang, I. Izmailov, S. Rizzo, S. Wynter, O. Alexandrov, C. Tong

October 17, 2007

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

UCRL-TR-235618
Unlimited Release
October 2007

Computer Experiments for Function Approximations [1]

Allison Chang [2]

Ilia Izmailov [3]

Shemra Rizzo [4]

Sharolyn Wynter [5]

Oleg Alexandrov [6]

Charles Tong [7]

[1]This project was jointly sponsored by the National Science Foundation Division of Math Sciences Award #0439872 and the U.S. Department of Energy Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344.

[2]Massachusetts Institute of Technology, aachang@mit.edu

[3]Carnegie Mellon University, izmailov@math.princeton.edu

[4]ITESM, shemra@gmail.com

[5]North Carolina State University, swynter84@@hotmail.com

[6]UCLA, aoleg@math.ucla.edu

[7]Lawrence Livermore National Laboratory, chtong@llnl.gov

Abstract

This research project falls in the domain of *response surface methodology*, which seeks cost-effective ways to accurately fit an approximate function to experimental data. Modeling and computer simulation are essential tools in modern science and engineering. A computer simulation can be viewed as a function that receives input from a given parameter space and produces an output. Running the simulation repeatedly amounts to an equivalent number of function evaluations, and for complex models, such function evaluations can be very time-consuming. It is then of paramount importance to intelligently choose a relatively small set of sample points in the parameter space at which to evaluate the given function, and then use this information to construct a surrogate function that is close to the original function and takes little time to evaluate.

This study was divided into two parts. The first part consisted of comparing four sampling methods and two function approximation methods in terms of efficiency and accuracy for simple test functions. The sampling methods used were Monte Carlo, Quasi-Random LP$_\tau$, Maximin Latin Hypercubes, and Orthogonal-Array-Based Latin Hypercubes. The function approximation methods utilized were Multivariate Adaptive Regression Splines (MARS) and Support Vector Machines (SVM). The second part of the study concerned adaptive sampling methods with a focus on creating useful sets of sample points specifically for monotonic functions, functions with a single minimum and functions with a bounded first derivative.

Contents

List of Figures

List of Tables

Acknowledgments

We would like to recognize Dr. Michael Raugh at UCLA IPAM (Institute for Pure and Applied Mathematics) for overseeing the RIPS (Research in Industrial Projects for Students) program and offering constant encouragement. We also wish to thank the IPAM staff for their support. Finally, we would like to acknowledge Dr. Charles Tong and Dr. Oleg Alexandrov for their supervision and guidance throughout this project.

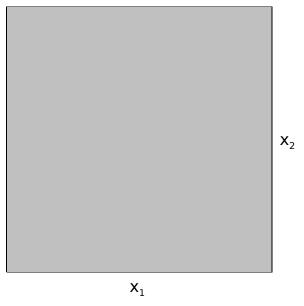
Chapter 1

Introduction

Modeling and computer simulation are essential tools in modern science because many systems of interest are too complex or too costly to build in real life. Simulations typically take in a set of inputs and return some output, so we can think of them as functions $F : X \to Y$. Here, X represents the parameter space from which we choose the input values, and Y represents the space of all possible output values. Models must undergo critical tests for verification and validation before they are actually deemed useful, and these tests generally require a large number of simulation runs. This translates into a large number of function evaluations, which may take an inordinate amount of time. One solution to this problem is the use of computer experiments to construct a new function, called a *response surface*, that closely approximates the original function and is inexpensive to evaluate.

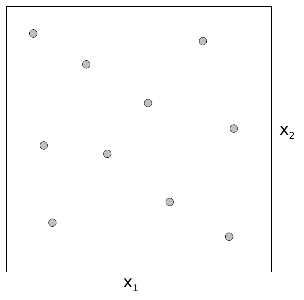
This project involved research in the area of *response surface (also called surrogate or emulator) surface methodology*, the aim of which is to develop efficient and accurate methods to fit an approximate function to experimental data. There are two major steps to constructing an approximation for a particular function. The first step is *experimental design*, or choosing a limited number of points in the parameter space at which to evaluate the original function. The selected sample points and corresponding output constitute a *training data set*. During this step, it is important that the training data capture the maximum amount of information with respect to the input-output relationships of the original function. The second step is to generate an approximate function based on the training data. Figures 1.1–1.4 provide a two-dimensional illustration of the process of constructing a response surface.

The first half of this project focused on existing function approximation methods, namely *Multivariate Adaptive Regression Splines* (MARS) and *Support Vector Machines* (SVM), which were both assessed in terms of efficiency and accuracy. Another objective for the first part was to establish whether there is any significant difference in using four different experimental design methods: *Monte Carlo*, *Quasi-Random* LP_τ, *Maximin Latin Hypercubes*, and *Orthogonal-Array-Based Latin Hypercubes.* These sampling methods are classified as *passive learning* techniques, that is, they select the sample points at which to evaluate the original function regardless of the nature of the function. Software for all the methods was either obtained from an outside source or implemented by the RIPS team, and the methods were applied to a number



htbp

Figure 1.1: Input parameter space of a 2-dimensional function, $f(x_1, x_2)$.

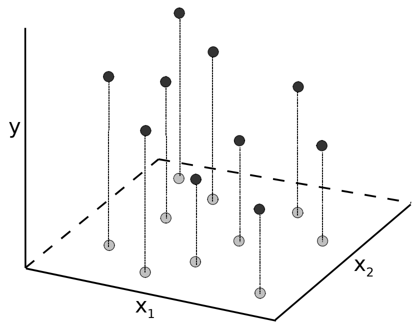


htbp

Figure 1.2: Well-selected sample points.

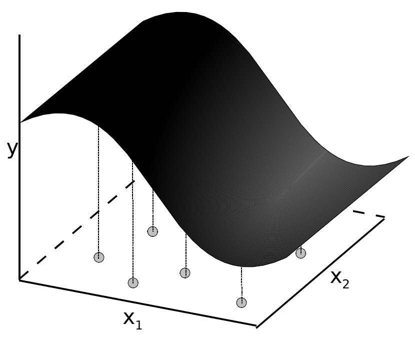
of test functions.

The second half of this project was devoted to *active learning* experimental designs, or *adaptive sampling*. The challenge was, knowing the values of the function for a set of sample points, to determine points at which the function should be further evaluated to facilitate an improved approximation. An important consideration for the second part of the project was how *a priori* information about a function could guide the sampling and function approximation methods. Users of function approximation techniques usually know some characteristics of their functions, such as monotonicity or bounded first derivative, even if the functions are not exactly defined. The ultimate goal of this project was to discover a means of using such information to develop a robust and cost-effective methodology for approximating functions.



htbp

Figure 1.3: Input-output pairs generated during sampling.



htbp

Figure 1.4: Response surface, or function approximation.

Chapter 2

Experimental Design

The first step in constructing a response surface is *experimental design*, also called sampling. In choosing points at which to evaluate the original function, we want the selected sample to be large enough so that the resulting approximation is reasonably accurate, but not too large since the original function is expensive to evaluate.

Ideally, regions in which the target function varies greatly would be sampled on a fine grid, and regions in which it varies little would be sampled on a coarser grid. For example, the left side of the function shown in Figure 2.1 changes dramatically while the right side is rather flat. Thus, it would be preferable to have many sample points in the left region of the parameter space and fewer points in the right region. Methods that implement this sort of reasoning fall in the area of *adaptive sampling*, which will be further discussed in Chapter 5.

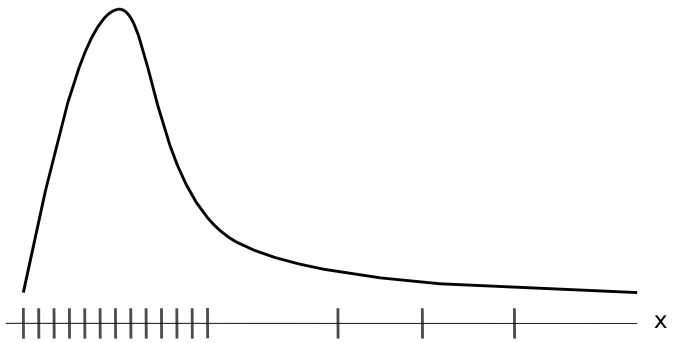


Figure 2.1: Ideal sampling.

However, in creating a response surface, the shape of the original function is generally unknown. So it is appropriate to use *passive sampling* methods that are "space-filling." In other words, these methods generate a set of sample of points that are more or less spread evenly over the entire parameter space. We used four different passive learning sampling methods: Monte Carlo, LP$_\tau$, Latin Hypercube and Orthogonal-Array-Based Latin Hypercube. All four methods return a set of n sample points within the unit hypercube $[0,1]^p$, where p is the dimension of the parameter space. The set of sample points is then scaled to cover the domain of the original function. In what follows, we describe the four methods in detail.

2.1 Monte Carlo

Monte Carlo methods are frequently used to solve various physical and mathematical problems. The primary characteristic of these methods is that they are stochastic.

The Monte Carlo method for this project was implemented in the C programming language using the `rand`() function, which outputs a random integer in the range from 0 to `RAND_MAX` inclusive, where `RAND_MAX` is the maximum value that can be returned by `rand`(). To generate a single sample point in p dimensions, the C program calls the `rand`() function p times. Each of the p components of the point is subsequently divided by `RAND_MAX` so that the point falls within the unit hypercube.

Figures 2.2–2.4 show two-dimensional Monte Carlo samples of n points, with $n = 100$, $n = 1000$, and $n = 5000$ respectively.

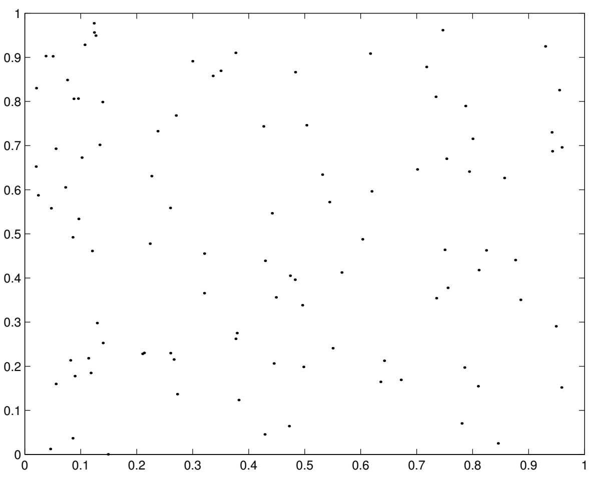


Figure 2.2: Monte Carlo sample in 2D, $n = 100$.

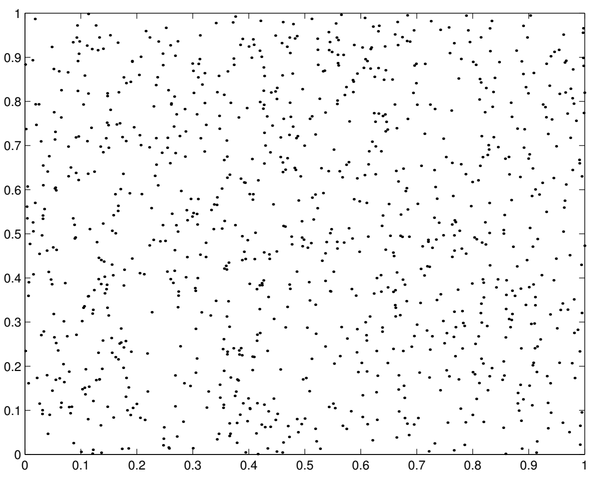


Figure 2.3: Monte Carlo sample in 2D, $n = 1000$.

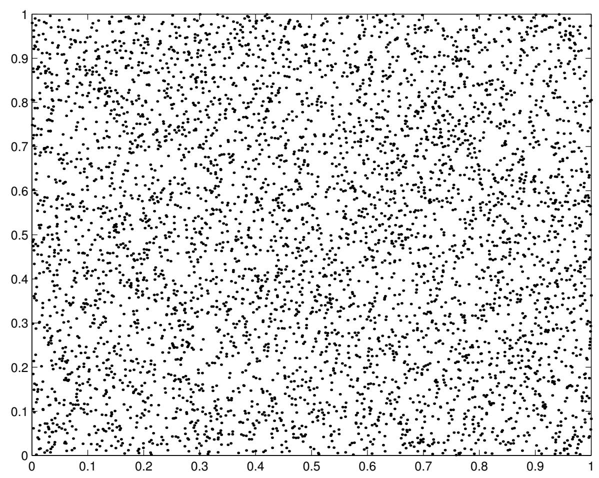


Figure 2.4: Monte Carlo sample in 2D, $n = 5000$.

2.2 LP$_\tau$

The LP$_\tau$ method is based on Sobol's quasi-random sequence generator [1] and returns a set of well-spaced points in a unit hypercube of any dimension up to 51. A key feature of this method is that the points follow a deterministic sequence, which generates $2^{30} - 1$ different points before repeating itself. Because the sequence is deterministic, a large sample includes all of the points in a smaller sample. In other words, if one creates a sample using LP$_\tau$ and then wishes to add more points to it, the method does not change the points already there; it just adds more points in a "space-filling" way.

To introduce the mathematics behind the LP$_\tau$ method, suppose a user desires a sample of n one-dimensional points $x_1, x_2, \ldots, x_n$. The LP$_\tau$ equation for generating these points is:

$$x_i = b_1 v_1 \oplus b_2 v_2 \oplus \ldots. \tag{2.1}$$

Here, $\oplus$ denotes the bit-by-bit exclusive-or operation, and $b_k b_{k-1} \ldots b_2 b_1$ is the binary representation of n. For example, the binary representation of 13 is 1101, so if $n = 13$, then $b_4 = 1, b_3 = 1, b_2 = 0, b_1 = 1$. Lastly, the v_i are *direction numbers*, or binary fractions that may be computed using the formula $v_i = m_i/2^i$, where the m_i are odd integers, $0 < m_i < 2^i$. The numbers v_i are calculated using a recurrence defined by the coefficients of a primitive polynomial in the field $\mathbb{Z}_2$. A deeper explanation of the Sobol generator is beyond the scope of this report but may be found in [1].

Figures 2.5–2.7 show two-dimensional LP$_\tau$ samples, with $n = 100$, $n = 1000$, and $n = 5000$, respectively.

In comparing samples generated by the Monte Carlo and LP$_\tau$ methods, it is evident that LP$_\tau$ points follow a regular pattern, while the purely random Monte Carlo points do not. In particular, the LP$_\tau$ sample covers the entire square in a well-spaced rhombus-like configuration and also concentrates several points in the center of each rhombus for more sensitive analysis of the data. Test results, presented in Chapter 4, appear to indicate that these characteristics make LP$_\tau$ a more effective

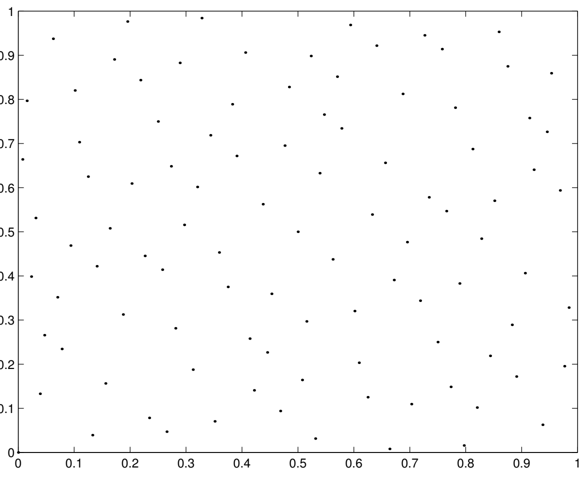


Figure 2.5: LP$_\tau$ sample in 2D, $n = 100$.

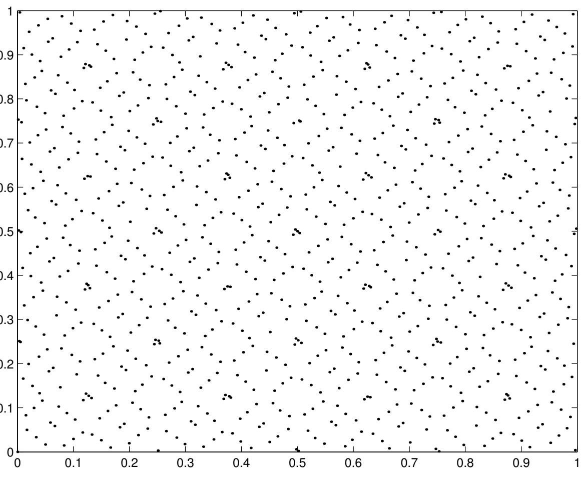


Figure 2.6: LP$_\tau$ sample in 2D, $n = 1000$.

sampling tool than Monte Carlo for the tested functions.

2.3 Latin Hypercubes

An $n \times n$ *Latin square* is a square of natural numbers in which each row and column contains a different permutation of the numbers $1, 2, \ldots n$ (or $0, 1, \ldots, n-1$), so that each of these numbers appears exactly once in every row and column. In Figure 2.3 below, we have a 5×5 Latin square.

Given an $n \times n$ Latin square, one can take the n cells at which the value is a fixed number to use as sample points. For example, Figure 2.3 shows how the coordinates of five sample points may derive from the locations of the number 1 in a Latin square. We can write these coordinates in an $n \times 2$ matrix, such as

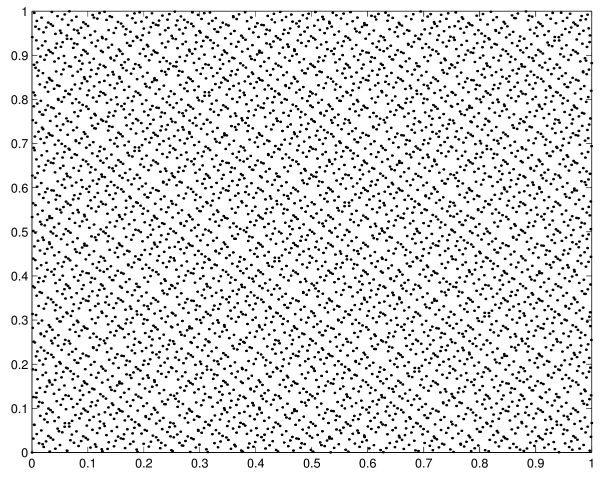


Figure 2.7: LP$_\tau$ sample in 2D, $n = 5000$.

Latin Square

1	2	3	4	5
3	5	1	2	4
2	4	5	3	1
5	3	4	1	2
4	1	2	5	3

Sample Points

1				
		1		
				1
			1	
	1			

Figure 2.8: A 5×5 Latin square and the corresponding sample points.

$$\begin{pmatrix} 1 & 1 \\ 2 & 3 \\ 3 & 5 \\ 4 & 4 \\ 5 & 2 \end{pmatrix}.$$

Note that each column of this matrix is a permutation π_j of $1, 2, \ldots, n$, where j is the number of the column. Since there are n rows and each entry of the matrix is a natural number between 1 and n inclusive, each column must represent all numbers in that range—otherwise there would be a row or column in the square with at least two sample points. The matrix of sample points can be scaled down to lie in the interior of $[0, 1]^2$ via the formula

$$X_{ij} = \frac{\pi_j(i) - 0.5}{n}, \tag{2.2}$$

where i is the number of the row. In our example, the result comes out to be

$$X = \begin{pmatrix} 0.1 & 0.1 \\ 0.3 & 0.5 \\ 0.5 & 0.9 \\ 0.7 & 0.7 \\ 0.9 & 0.3 \end{pmatrix}.$$

Figures 2.9–2.12 show Latin square samples of n points, using $n = 100$, $n = 500$, $n = 1000$, and $n = 5000$.

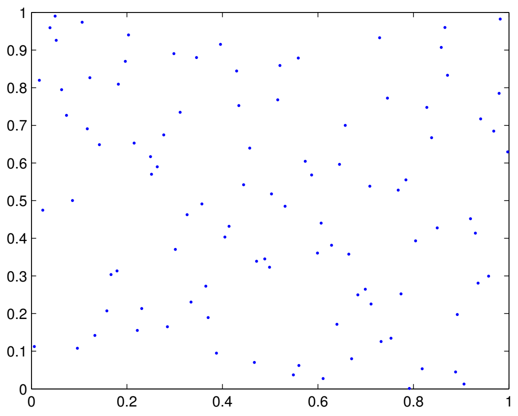


Figure 2.9: Latin hypersquare sample, $n = 100$.

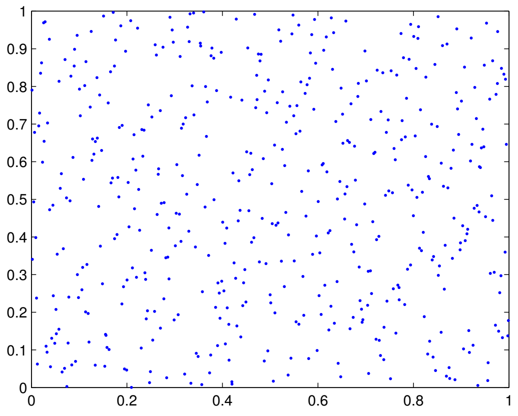


Figure 2.10: Latin hypersquare sample, $n = 500$.

A *Latin hypercube* is a generalization of a Latin square to p-dimensional space. Again, each row is a permutation of $1, 2, \ldots, n$, and all n^{p-1} points with a fixed value are chosen as sample points. The $n^{p-1} \times p$ array is scaled down to fit in $[0, 1]^p$ via basically the same formula as in the two-dimensional case (Equation 2.2). The only difference is that for two dimensions, $j \in \{1, 2\}$ and for p dimensions, $j \in \{1, 2, \ldots, p\}$. Figure 2.13 depicts an example of a cube ($p = 3$) with five sample points.

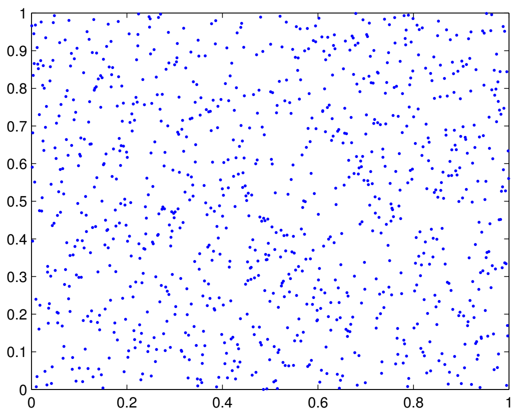


Figure 2.11: Latin hypersquare sample, $n = 1000$.

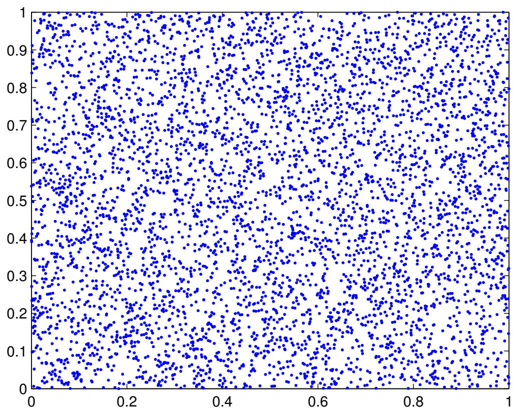


Figure 2.12: Latin hypersquare sample, $n = 5000$.

Note that the definition of a Latin square at the beginning of this section does not exclude the possibility of having the same number along the diagonal of the square. For instance, an equally valid alternative to Figure 2.8 could be a square in which all five 1's are located on the diagonal. Clearly this would not correspond to an ideal set of sample points. To counteract this problem, which generalizes to higher dimensions, this project uses a *maximin* Latin hypercube algorithm, which generates multiple Latin hypercube samples and then selects the one for which the minimum distance between the points is largest.

2.4 Orthogonal-Array-Based Latin Hypercubes

Although Latin hypercubes give rise to good samples with one point representing every row, *orthogonal arrays* (OAs) may be used to generate even more well-spaced samples. An orthogonal array A is an $N \times k$ matrix, each of whose entries is a

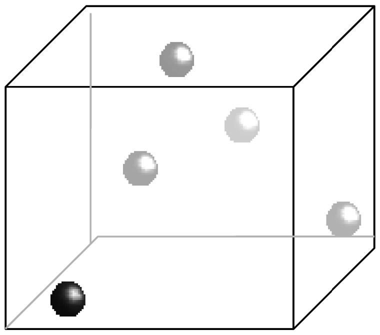


Figure 2.13: Sample points in a Latin cube.

number in $0, 1, \ldots, s-1$ for a predefined $s \in \mathbb{N}$, referred to as the *number of levels*. The *strength* of such an array, often denoted t, is the maximum number satisfying the condition that any subarray of A containing t of the columns and the full set of rows would have each of the possible rows represented an equal number of times. This number, λ, is called the *index* of the array and can be found by the formula $\lambda = \frac{N}{s^t}$. A common and convenient notation to describe an array with the above-mentioned parameters is $OA(N, k, s, t)$, which shall be used throughout this report. An example of an $OA(8, 7, 2, 2)$ array is shown below. Note that this array is not *orthogonal* in the typical linear algebra sense.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

For instance, if we take columns 1 and 5, we note that, as expected from the definition of an orthogonal array, each of the obtained rows of length 2 shows up $\frac{8}{2^2} = 2$ times:

$$\begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

It is worth noting that, given an arbitrary set of parameters (N, k, s, t), neither the existence nor the uniqueness of $OA(N, k, s, t)$ is guaranteed. However, there exists a method, called *Bush's construction*, for creating an orthogonal array $OA(s^t, s+1, s, t)$ for any $s, t \in \mathbb{N}$, with s being a power of a prime. Note that in the case at hand,

$$\lambda = \frac{s^t}{s^t} = 1,$$

so every row must be unique, a property that will be necessary later on. Since one can remove columns from an orthogonal array without destroying its orthogonality, we have means of creating an orthogonal array $OA(s^t, k, s, t)$ for any k such that $t \leq k \leq s$.

Given an orthogonal array A with p factors and s levels, it is possible to use it to create an array X of sample points (rows) by the formula

$$X_{ij} = \frac{\pi_j(A_{ij}) + 0.5}{s}. \tag{2.3}$$

We permute the elements of each column for added randomness. This sampling method is known as *Orthogonal-Array-Based Latin Hypercubes* (OALH) and is stronger than its non-OA-based counterpart for which, in essence, the sample points are derived from orthogonal arrays of strength 1.

Figures 2.14–2.17 show two-dimensional orthogonal-array-based Latin hypercube samples of different sample sizes n. Note how the points are distributed more evenly than in the case of an ordinary Latin hypercube (see Figures 2.9–2.12).

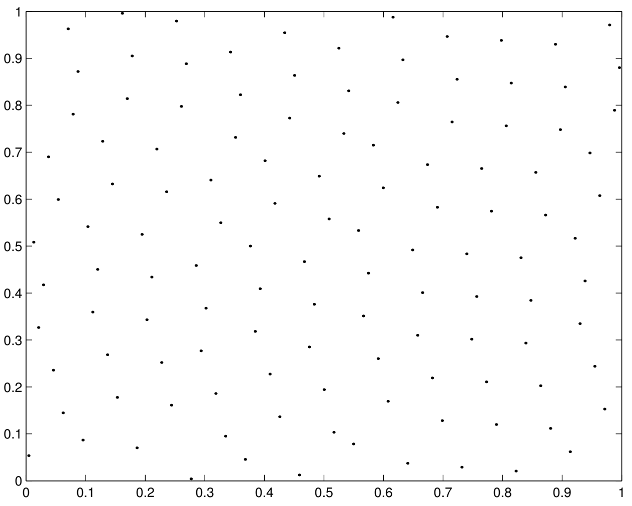


Figure 2.14: OALH sample in 2D, $n = 121$.

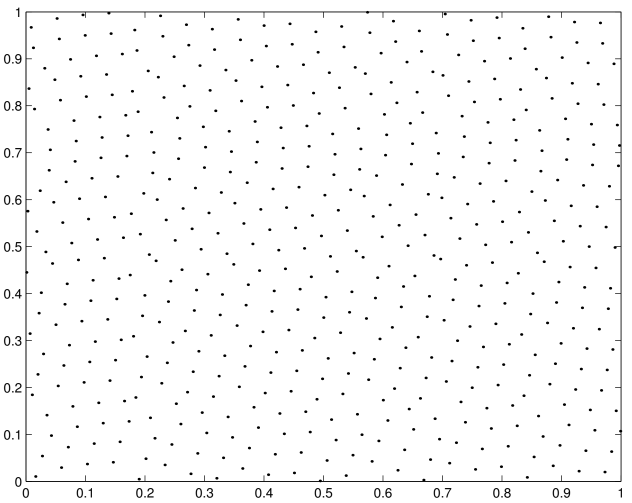


Figure 2.15: OALH sample in 2D, $n = 529$.

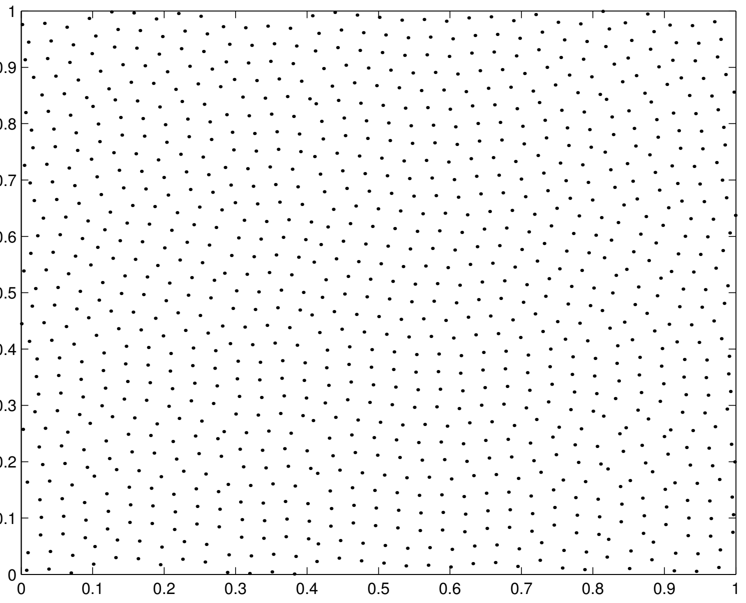


Figure 2.16: OALH sample in 2D, $n = 1,024$.

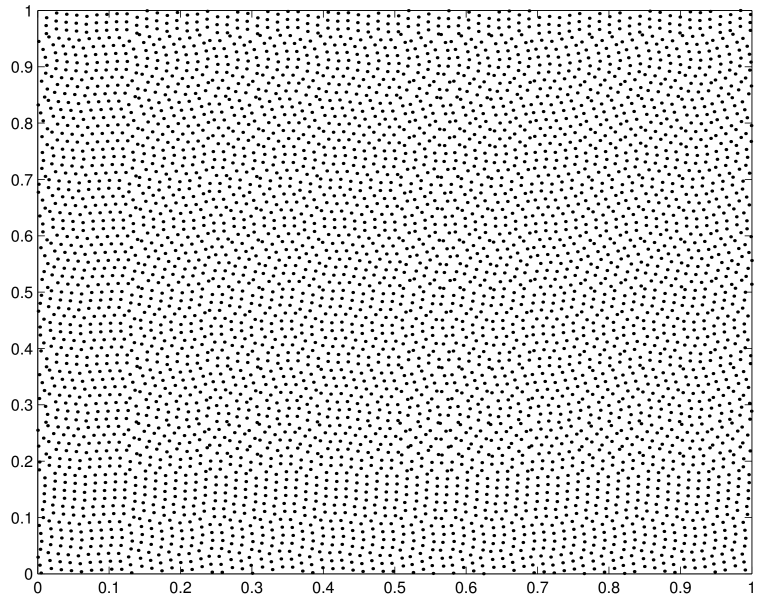


Figure 2.17: OALH sample in 2D, $n = 5,041$.

Chapter 3

Function Approximation

We obtained software packages that implement two function approximation methods, called MARS and SVM. Both packages build an approximate function based on a training data set that contains a set of sample points and the corresponding output values. This chapter first describes how we evaluated the error in the approximations and then discusses the two different methods in depth.

3.1 Error Metrics

We analyzed the accuracy of the approximations generated by MARS and SVM using two different error measures. Since we have explicit formulas for the test functions, we can compute the exact value of a function at certain points. Note that even though the motivation behind function approximation methods is the existence of computationally expensive functions, the test functions for this project are all simple enough that they can be easily evaluated for many points. Thus, it is possible to calculate a root mean squared (RMS) error between the exact and approximate values. If $\mathbf{g}$ and $\mathbf{h}$ are m-vectors, then the RMS error between them is defined by the formula

$$RMS = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(g_i - h_i)^2}. \tag{3.1}$$

We will call this first error measure the *prediction error.*

The second error measure involves a process called *cross-validation.* With this approach, we remove one point from the training data set and generate a function approximation to fit the remaining points in the set. We use this approximation to predict the function value at the removed point. In other words, we "cross-validate" an exact function value at a sample point in the training data set against the value that would be predicted at that point based on an approximation generated to fit the rest of the points in the data set. After performing cross-validation at a specified number of sample points, we compute an RMS error between the exact and predicted function values at those points. This error will be called the *cross-validation error.* Unlike the prediction error, the cross-validation error does not require knowledge of the original function to be calculated. This is an advantage because in practice, formulas for the original functions may not be known.

3.2 Multivariate Adaptive Regression Splines

Multivariate Adaptive Regression Splines (MARS) is a regression procedure that makes no assumption about the underlying functional relationship between the dependent and independent variables. The MARS algorithm has two steps, the first of which is a forward stepwise regression procedure that implements a recursive partitioning process. In a sense, the method is based on the "divide and conquer" strategy, which partitions the input space into regions, each with its own regression equation. MARS uses *basis functions* in each region to generate an approximation to the corresponding part of the original function. The actual number of basis functions is chosen by the user before running the algorithm. The final function approximation is a weighted sum of the individual basis functions.

More specifically, the response surfaces generated by MARS are *splines*, which are formed by joining polynomials of the same degree together at the *knots*, or division points between regions (see Figure 3.1). At each knot, MARS requires that the two relevant polynomials of degree k join *smoothly*, i.e., their derivatives must be equal at the knot up to order $k-1$. Note that in Figure 3.1, $k-1=0$, so the smoothness criterion requires only that the spline be continuous. The MARS software allows the user to choose between two kinds of splines — piecewise-linear ($k=1$) and piecewise-cubic ($k=3$).

The MARS algorithm analyzes the entire space of inputs and outputs, as well as the interactions between variables in the function, in order to decide how to split the function domain into regions and to select the optimal basis functions for each region. During this analysis, basis functions are added to the model to maximize an overall least squares goodness-of-fit criterion. As a result of these operations, MARS automatically determines the most important independent variables as well as the most significant interactions among them.

The second step of the algorithm is a backward stepwise procedure to remove basis functions that do not contribute significantly to maximizing the least-squares goodness-of-fit criterion. Each basis function represents a disjoint region in the function domain, so removing it just by itself would cause the resulting approximation to have zero value in the corresponding region. Instead, MARS deletes basis functions by merging them into a single (parent) region in roughly the inverse splitting order. This pruning procedure of the basis functions counteracts the problem of over-fitting.

The general MARS model equation is:

$$y = f(x) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(x), \tag{3.2}$$

where M is the number of basis functions in the model, h_m are the basis functions, and β_m are the weights in the sum of the basis functions. A comprehensive description of MARS may be found in [5].

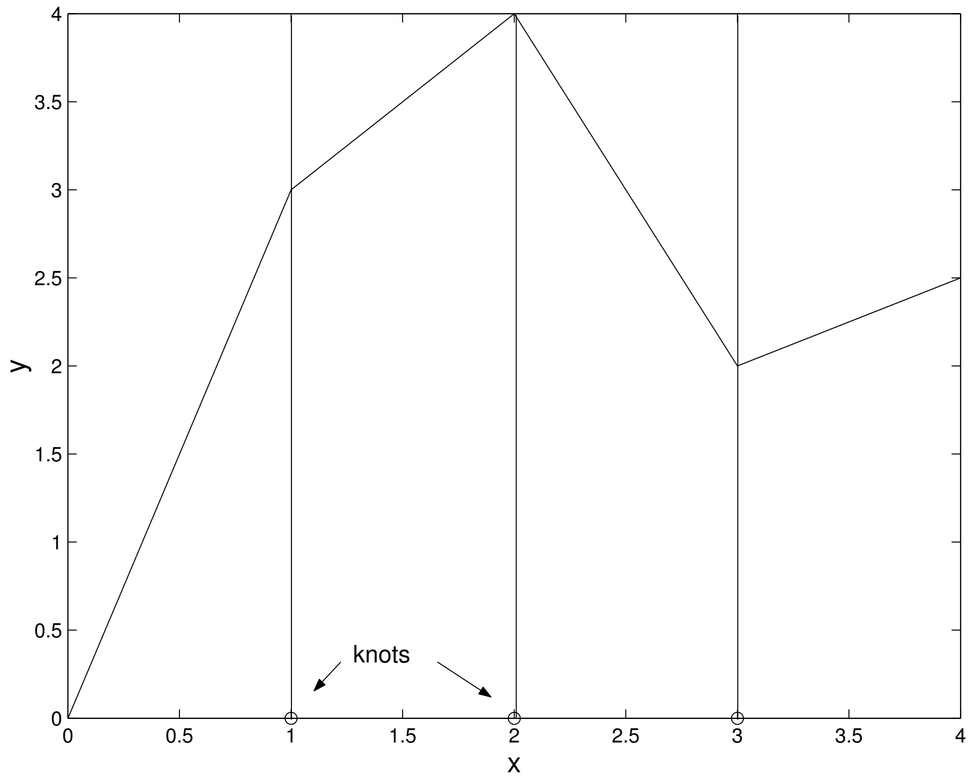


Figure 3.1: A one-dimensional example of the piecewise-linear MARS approach.

3.3 Support Vector Machines

Support Vector Machines (SVMs) are learning methods used for data classification or, as in the case of this study, regression. The sample points, $\mathbf{x}_i \in [0,1]^p$, together with their corresponding y-values, $y_i \in \mathbb{R}$, are known as *support vectors*. The goal is to find a hyperplane approximating the vectors within an error margin of $\varepsilon > 0$ (see Figure 3.2).

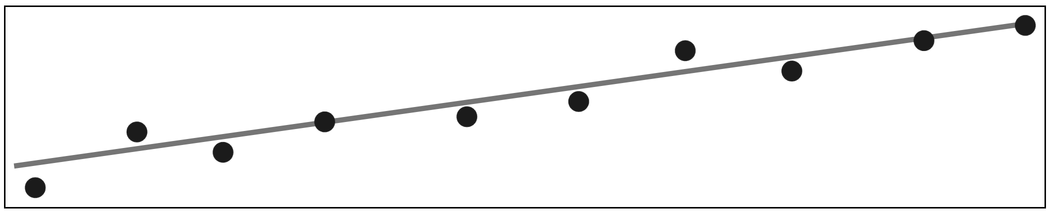


Figure 3.2: A hyperplane approximating the support vectors.

Defining $\mathbf{w}$ as the vector normal to the hyperplane, the program seeks to minimize its norm ($||\mathbf{w}|| = \sqrt{\mathbf{w}^\mathsf{T}\mathbf{w}}$) subject to the constraint

$$\left|y_i - (\mathbf{w}^\mathsf{T}\mathbf{y} + b)\right| \leq \varepsilon,$$

where b is the constant in the equation of the hyperplane,

$$f(\mathbf{y}) = \mathbf{w}^{\mathsf{T}}\mathbf{y} + b.$$

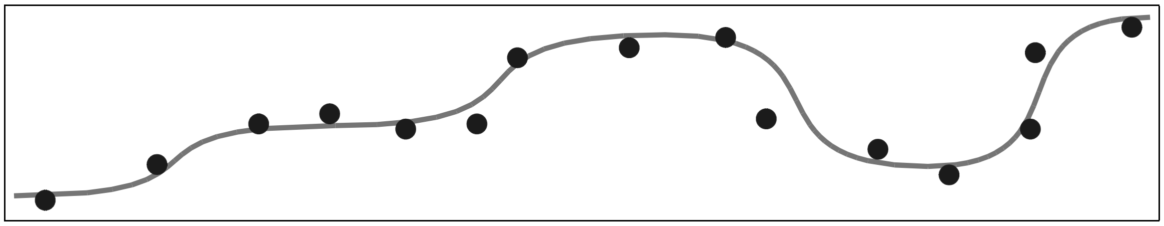


Figure 3.3: A curve approximating the support vectors.

When nonlinear regression is desired (Figure 3.3), the sample points $\{\mathbf{x}_i\}$ are mapped to a *feature space* $\mathfrak{F}$ via a *feature map* $\Psi : [0,1]^p \mapsto \mathfrak{F}$ that corresponds to a *kernel* K, such that $K(\mathbf{x}_i, \mathbf{x}_j) = \Psi(\mathbf{x}_i)^{\mathsf{T}}\Psi(\mathbf{x}_j)$. Since it is only the dot product of the vectors in question that matters, it is customary (and often more convenient) to use the kernel function instead of the feature map itself. The four kernels used by SVM are

- linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i{}^{\mathsf{T}}\mathbf{x}_j$
- polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i{}^{\mathsf{T}}\mathbf{x}_j + r)^d$, where $\gamma > 0$
- radial basis function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2}$, where $\gamma > 0$
- sigmoid: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i{}^{\mathsf{T}}\mathbf{x}_j + r)$.

Here γ, r, and d are kernel parameters.

This project uses Thorsten Joachims' implementation of SVM called `SVM`$^{\texttt{light}}$, which is freely available at `http://svmlight.joachims.org`.

Chapter 4

Test Results for Part I

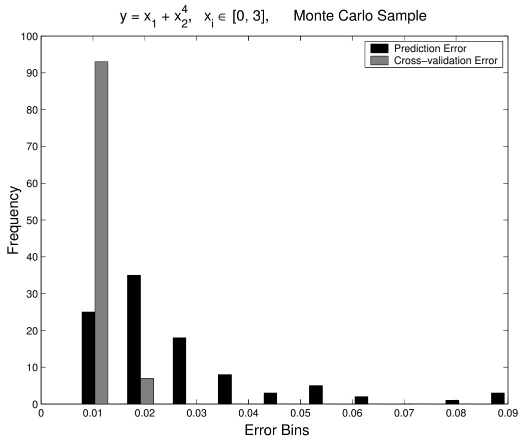
In this chapter, we describe our procedure for testing the four passive sampling methods and two function approximation methods detailed in Chapters 2 and 3. We also present an iterative algorithm for constructing response surfaces that makes use of what we found to be the best techniques. Because we wanted to make sure that we understood how the methods worked for simple cases before testing more complicated cases, we used low-dimensional, well-behaved functions for all of the tests.

4.1 Comparing Sampling Methods

To determine differences in using Monte Carlo, Quasi-Random LP$_\tau$, Maximin Latin Hypercube, and Orthogonal-Array-Based Latin Hypercube samples (see Chapter 2 for descriptions), we generated 361 points with each of the four sampling methods and then constructed response surfaces to approximate the two-dimensional test functions $y = x_1 + x_2^4$ and $y = x_1 + x_1x_2 + x_2$. For both functions, the domain was $[0, 3]^2$. The function approximation method was MARS, and we specified a piecewise-cubic model with 50 basis functions. We also specified interaction levels of 1 for the first function and 2 for the second. These parameters will be further explained in the next section. We used 100 points to compute the prediction error and 100 points to compute the cross-validation error (see Section 3.1 for error definitions).

For a given sample size, the LP$_\tau$ and Maximin Latin Hypercube algorithms produce the same samples each time they are run, but the Monte Carlo and Orthogonal-Array-Based Latin Hypercube methods generate different samples since both of them are based on random numbers. Thus, for each of the latter two methods, we generated 100 samples and calculated average errors. Figures 4.1 and 4.2 show histograms of the prediction and cross-validation errors for the two methods.

Figure 4.3 shows bar graphs of the prediction and cross-validation errors. The Monte Carlo method resulted in the highest prediction error for both test functions. This makes sense since it is a completely random method that does not guarantee well-spaced points. The results for the other three methods are not consistent between the two test functions, so it is difficult to predict how they will work for other functions. We will use LP$_\tau$ for subsequent tests because it resulted in relatively low errors for the two initial tests. This method is also deterministic and therefore has the useful property of always producing the same set of points for a given sample size.



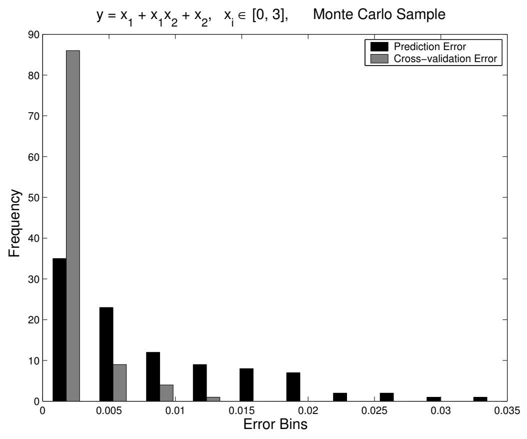
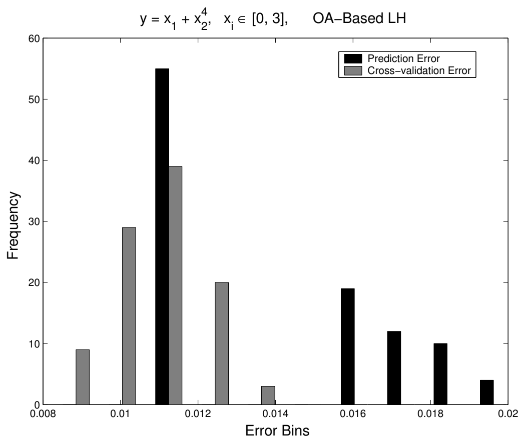


Figure 4.1: Error histograms for Monte Carlo sampling.



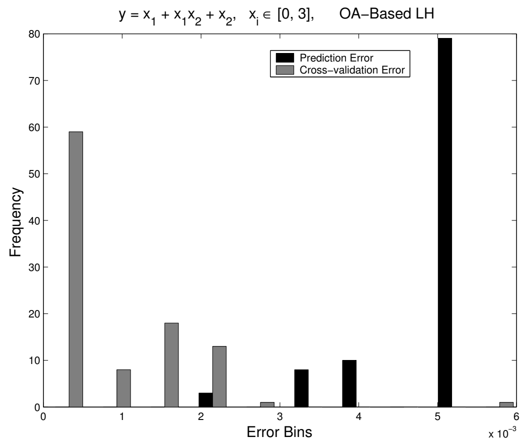


Figure 4.2: Error histograms for OA-Based LH sampling.

4.2 Tuning MARS

The MARS software allows users to control a number of parameters. As mentioned in Section 3.2, MARS involves splitting the function domain into several regions, and then assigning to each region its own basis function. Users can choose the number of basis functions, as well as whether the approximate function should be piecewise-linear or piecewise-cubic. One final parameter that can be varied is the level of interaction among variables. If a function evaluation involves the product of two variables, x_1x_2 or $x_1^2x_2^3$ for example, the interaction level is 2; if a function evaluation involves the product of three variables, the interaction level is 3, and so on. Again, we emphasize that our test functions are low-dimensional because we wanted to understand the

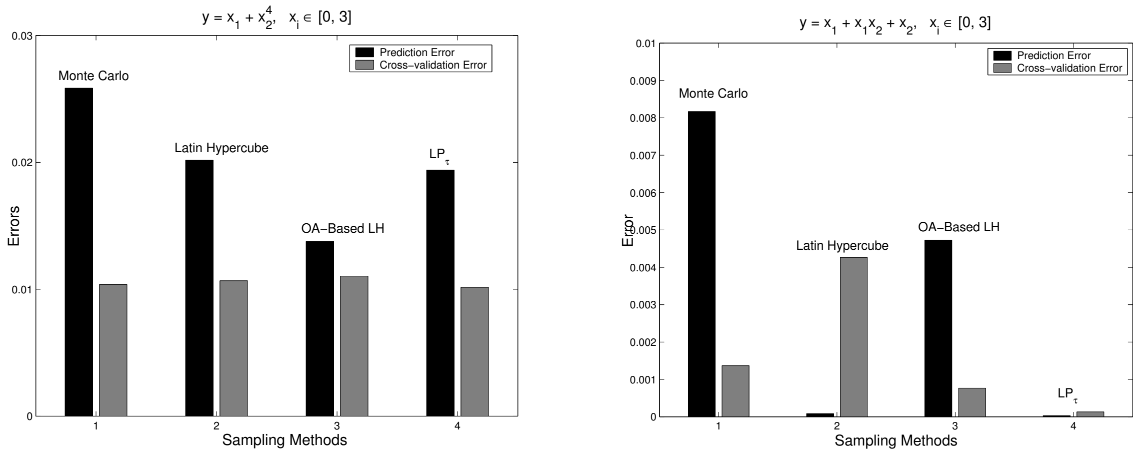


Figure 4.3: Test to compare sampling methods.

behavior of the methods for simple cases. Our conclusions have not been tested in higher dimensions.

4.2.1 Varying MARS Parameter Values

To test the outcome of using different parameter values, we first generated 729 sample points with the LP$_\tau$ method. Then we used 50 points to compute the prediction error and the same number of points for the cross-validation error. Figures 4.4–4.5 show results for the functions $y = x^3$, $y = x^6$, $y = \sin x$, and $y = \sin^2 x$. The domain for the first two functions was $[-2, 2]$, and the domain for the last two functions was $[-\pi, \pi]$. The diamond and square markers represent the prediction and cross-validation errors respectively for the linear model, and the triangular and x-shaped markers represent the prediction and cross-validation errors respectively for the cubic model. From these plots, we notice that on the whole, the piecewise-cubic model has a lower error than the piecewise-linear model. This implies that the piecewise-cubic model is generally more accurate than the piecewise-linear model.

In addition, Figures 4.4–4.5 display a pattern in the number of basis functions required to generate an accurate approximation. Considering that the complexity of the approximation increases with the number of basis functions, we want this number to be as small as possible. The plots show that the error does not decrease significantly between 50 and 100 basis functions. This implies that we likely do not need more than 50 basis functions for a sample of 729 points in order to get an accurate approximation for a one-dimensional function.

4.2.2 Varying Sample Size

We also tested whether or not the number of sample points affected the number of basis functions needed for an accurate approximation. This time, we used only two test functions: $y = x^5$, $x \in [-2, 2]$, and $y = \sin(2x)$, $x \in [-\pi, \pi]$. We specified

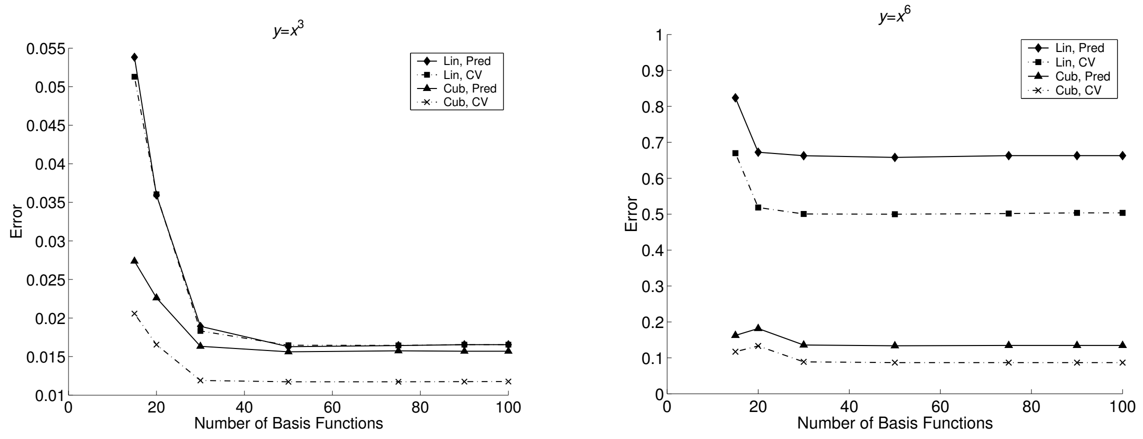


Figure 4.4: Error plots for $y = x^3$ (left) and $y = x^6$ (right).

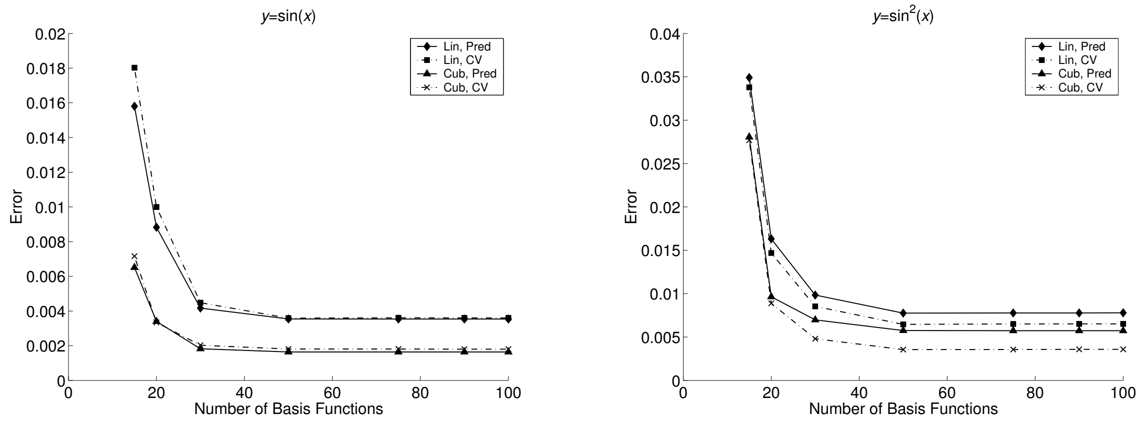


Figure 4.5: Error plots for $y = \sin x$ (left) and $y = \sin^2 x$ (right).

the piecewise-cubic model for all cases. The different markers in the following plots represent different sample sizes, ranging from 169 to 841. As illustrated by Figures 4.6–4.7, the error stabilizes at 40 basis functions, regardless of the function and the sample size. Thus, the test results seem to indicate that for one-dimensional functions, the sample size does not affect the desired number of basis functions.

4.3 Comparing Error Metrics

Note that the cross-validation errors are directly related to the prediction errors. That is, a high prediction error implies a high cross-validation error, and a low prediction error implies a low cross-validation error. Therefore, the cross-validation error appears to be an appropriate indicator of accuracy. However, computing it can be slow because each point selected for cross-validation corresponds to a new function approximation. For instance, using 100 points for cross-validation means 101 function approximations, including the one based on the original training data. In contrast, using 100 points to compute the prediction error does not require more than the single initial function approximation from the training set. For purposes of efficiency, we computed only the prediction error in the remainder of our tests.

Nevertheless, it is important to keep in mind the advantage of the cross-validation error stated in Section 3.1, that it does not require knowledge of the original function. In practice, the original function usually has no explicit formula, and even if it does, evaluating the function would be more expensive than constructing a new approximation. In other words, computing the prediction error in practice, if not impossible, would be even more expensive than computing the cross-validation error. We use the prediction error in our tests only because it takes less time to calculate for our simple test functions.

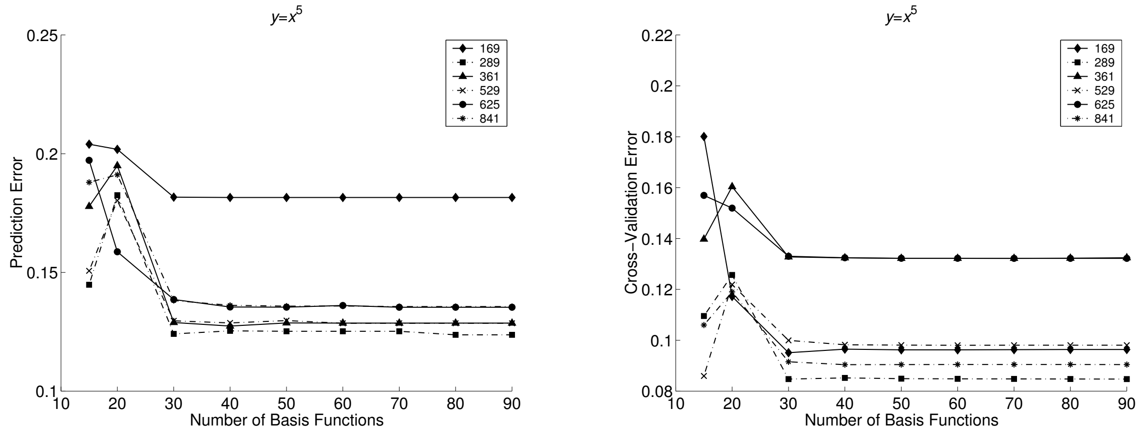


Figure 4.6: Error plots for $y = x^5$.

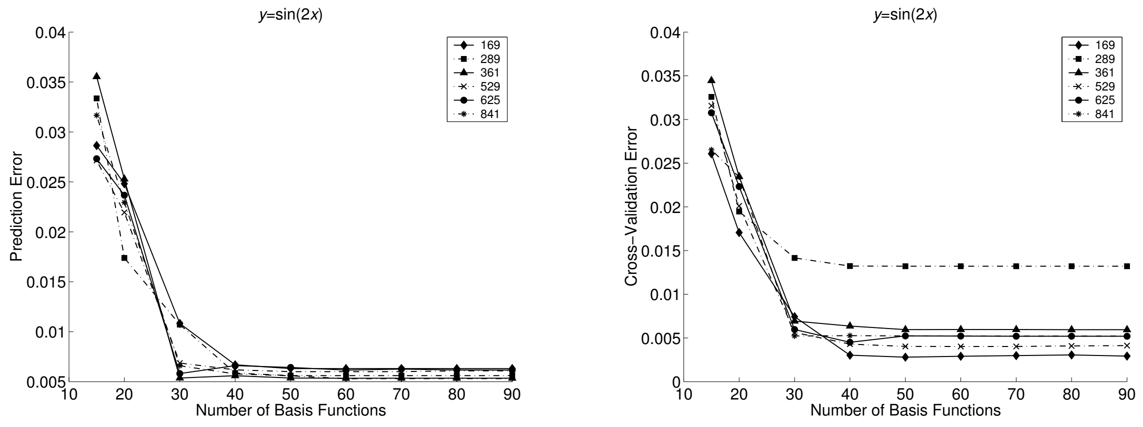


Figure 4.7: Error plots for $y = \sin(2x)$.

4.4 Tuning SVM

The SVM software also has several parameters that may be changed from the default values to produce more accurate response surfaces. As described in Section 3.3, the SVM method makes use of *kernel functions*. We discovered through some initial testing that the polynomial kernel does not work well for non-polynomial functions, but that the radial basis function kernel is fairly versatile. So for the rest of our tests, we used the radial basis function kernel, which has the form $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma ||\mathbf{x}_i - \mathbf{x}_j||^2}$, where $\gamma > 0$. One of SVM's parameters is the value of γ. Two other parameters are w, the width of the tube for regression, and e, a termination criterion.

We computed prediction errors from running SVM with different values for γ, w, and e on two test functions: $y = x^3$, $x \in [-2, 2]$, and $y = \sin x$, $x \in [-\pi, \pi]$. The 90 different cases are presented in Table 4.1.

γ	w	e
5	10^{-1}	10^{-3}
10	10^{-2}	10^{-4}
20	10^{-3}	10^{-5}
	10^{-4}	10^{-6}
	10^{-5}	10^{-7}
	10^{-6}	

Table 4.1: Test values for γ, w, and e.

We generated 729 sample points using LP$_\tau$ for each test function and used 50 points to compute the prediction error. The prediction errors for $\gamma = 5$ are plotted in Figure 4.8. The error curves for $\gamma = 10$ and $\gamma = 20$ have essentially the same shape as for $\gamma = 5$, only the errors are slightly higher. There are actually five different curves in each plot, which correspond to the five different values of e. Since the curves are so close to each other that they are barely distinguishable, we can infer, at least for one-dimensional functions, that the value of e does not matter much. The relationship between the value of w and the prediction error appears almost linear.

4.5 Comparing MARS and SVM

We tested the performance of MARS and SVM for different sample sizes on the same test functions: $y = x^3$, $y = x^6$, $y = \sin(2x)$, and $y = \sin^2 x$. The domain of the first two functions was $[-2, 2]$, and the domain of the last two functions was $[-\pi, \pi]$. For MARS, we changed the number of basis functions depending on the number of sample points (Table 4.2). In general, there should not be more basis functions than sample points because this would mean that at least one of the regions into which the MARS algorithm divides the function domain has no sample points to use in constructing a local approximation. For SVM, we used $\gamma = 4$ for all cases and changed the w and e parameters depending on the function (Table 4.3). The values of w and e were chosen so that running SVM would take no more than a few seconds.

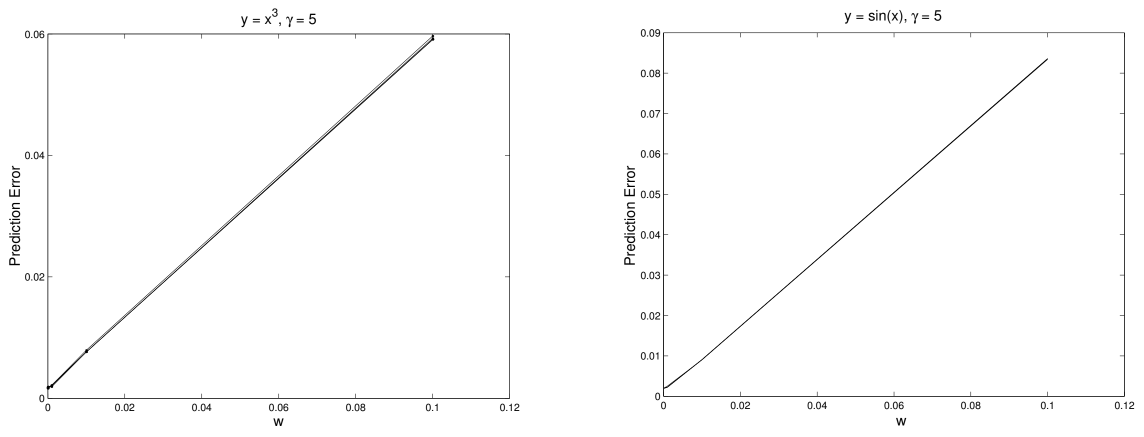


Figure 4.8: Prediction errors for $y = x^3$ and $y = \sin x$, $\gamma = 5$.

number of sample points	25	53	121	289
number of basis functions	11	20	40	50

Table 4.2: Number of basis functions for MARS.

	$y = x^3$	$y = x^6$	$y = \sin(2x)$	$y = \sin^2 x$
w	0.01	0.1	0.001	0.00001
e	0.001	0.001	0.0001	0.0001

Table 4.3: SVM values for w and e.

The subplots in Figures 4.9–4.12 each contain three curves — the exact function, and the two response surfaces produced by MARS and SVM.

Both MARS and SVM create accurate approximations for the $n = 121$ and $n = 289$ cases. However, for $n = 25$ and $n = 53$, MARS performs rather poorly, especially at the endpoints of the domain (see Figures 4.9–4.12). SVM is better for small sample sizes and produces less error at the endpoints. Overall, the prediction errors for SVM are lower than those for MARS, but for the larger sample sizes, SVM takes seconds to run while MARS runs almost instantaneously on an Ubuntu computer with a 3.2 GHz Intel Pentium 4 processor and 2.0 GB of RAM. Figures 4.13 and 4.14 show the prediction errors for the four functions as a function of the number of sample points. For all four functions, the error drops quickly between $n = 25$ and $n = 53$, and becomes nearly flat between $n = 121$ and $n = 289$.

4.6 Conclusions

Our conclusions are based on preliminary results using simple analytic functions. They may serve to illustrate the performance of methods applicable to more complicated functions of interest to LLNL.

- The LP$_\tau$ sampling method is preferable because it creates space-filling samples

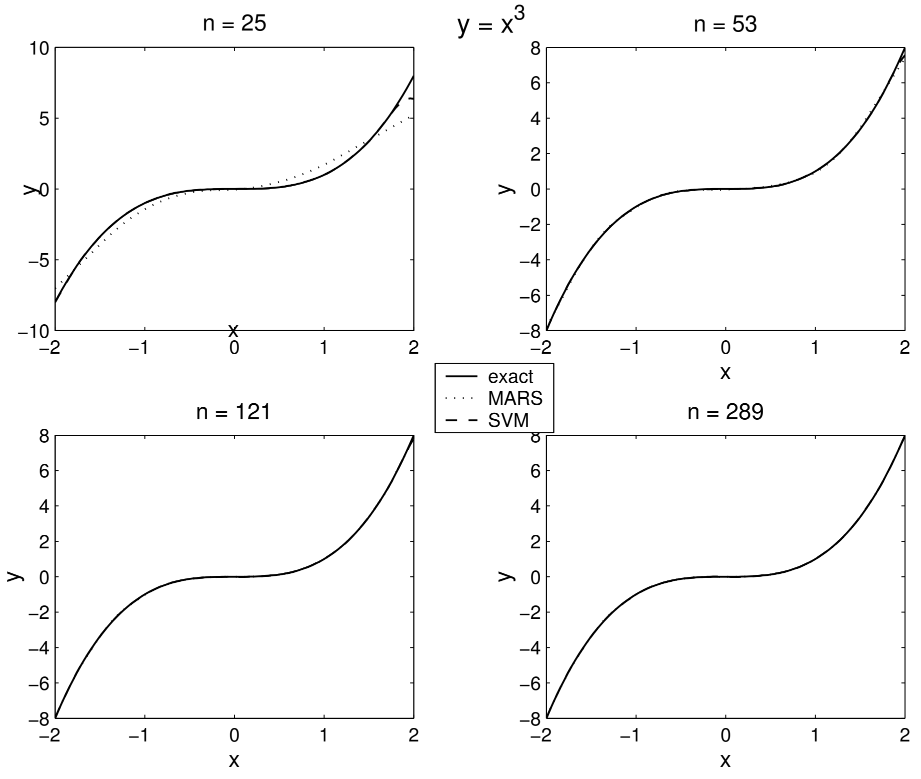


Figure 4.9: MARS and SVM approximations for $y = x^3$.

that result in more accurate function approximations than Monte Carlo, Latin Hypercubes, and Orthogonal-Array-Based Latin Hypercubes.

- SVM is not an easy method to use because there does not seem to be a clear pattern as to which parameter values lead to the most efficient and accurate function approximation. It is difficult for a novice user to know how SVM will behave for a particular function — whether the approximation will take a long time to generate and how large the error will be. MARS generally runs much faster than SVM, and there are not as many parameters the user can modify. However, using the right parameter values, SVM has the potential to generate more accurate approximations than MARS can, especially at the endpoints of the function domain for small sample sizes.

4.7 Iterative Algorithm

In using passive learning sampling techniques, it is difficult to know the number of sample points needed to create a function approximation of a certain accuracy. For example, we might want the error to be no larger than 10% of the maximum absolute value of the function. Larger samples generally mean higher accuracy, but we do not want more points than is necessary. One way to handle this problem is to start with n_0 sample points generated using LP$_\tau$ and run the following iterative algorithm:

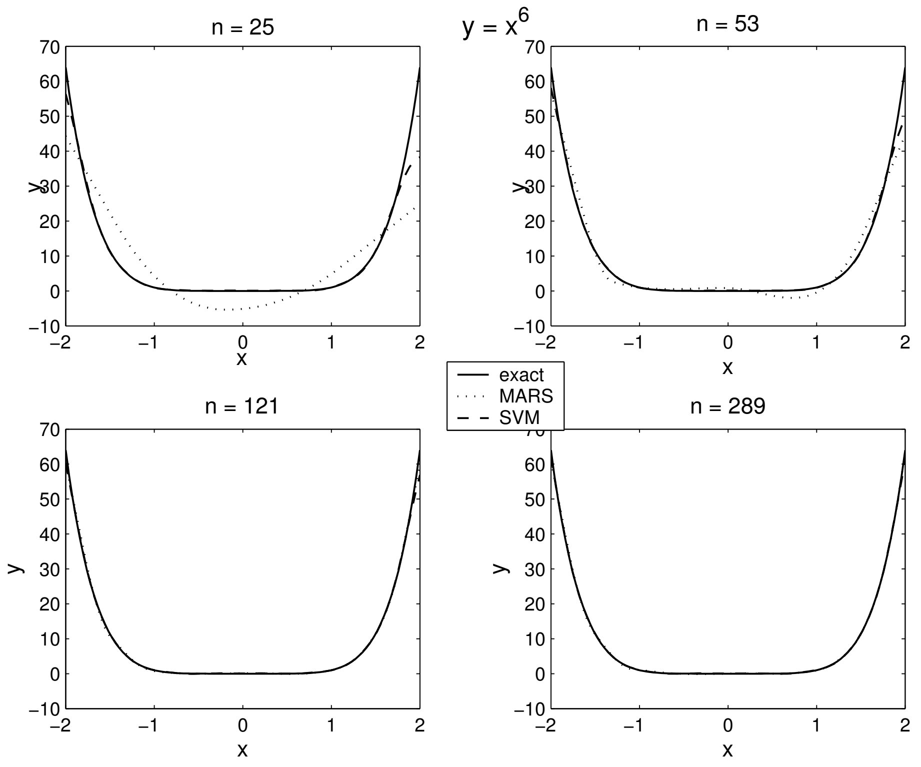


Figure 4.10: MARS and SVM approximations for $y = x^6$.

1. Fit a response surface to the sample using MARS, and compute the cross-validation error.

2. Divide the cross-validation error by the maximum absolute value of the function. (We will call this proportion the "normalized error.")

3. If the normalized error is less than `TOL`, the approximation is accurate enough. Otherwise, add more sample points and repeat steps 1 through 3.

It is appropriate to use the cross-validation error instead of the prediction error because we would like for this algorithm to work in practice. As mentioned in Sections 3.1 and 4.3, exact formulas for the target function are often unknown or extremely expensive to evaluate. Thus, it would not be reasonable to use the prediction error in this case.

For example, we applied the algorithm to the three-dimensional Ishigami function:

$$y = \sin(x_1) + 7\sin^2(x_2) + 0.1x_3^4\sin(x_1), \quad x_i \in [0, 3]. \tag{4.1}$$

One complication for this algorithm is that the number of basis functions should depend on the number of sample points n. For the Ishigami example, we used 25 basis functions for $n \in \{50, 100, 150\}$, and 100 basis functions for $n \geq 200$. We also used $n_0 = 50$, `TOL` $= 0.1$, and added 50 points with each iteration. At $n = 300$, the normalized error was 0.099297.

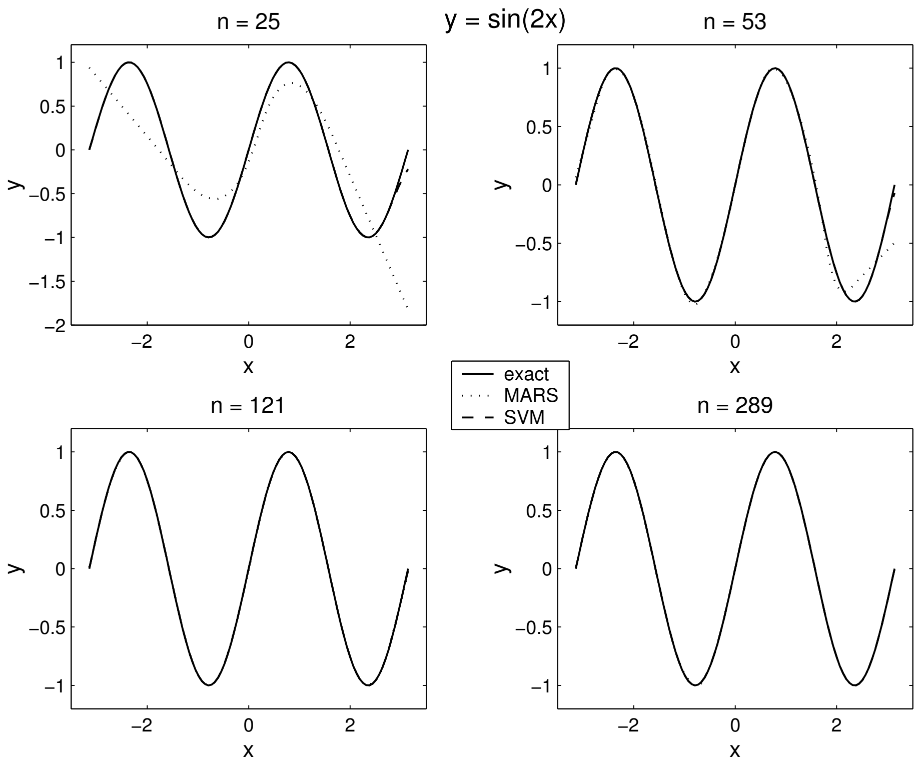


Figure 4.11: MARS and SVM approximations for $y = \sin(2x)$.

The iterative algorithm just described is an *active learning* algorithm because it chooses the optimal number of sample points based on what it has learned from choosing smaller samples. In the next chapter, we will cover more complex techniques for *adaptive sampling* that seek not the optimal number but the optimal placement of sample points within the parameter space.

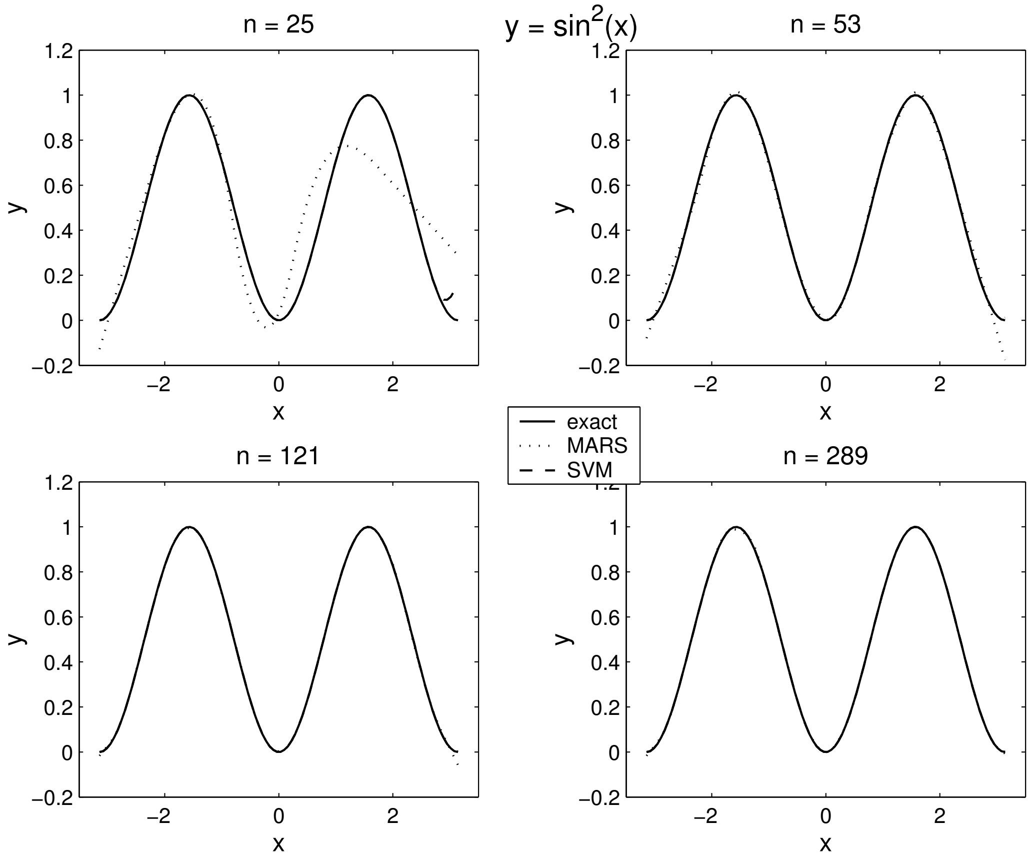


Figure 4.12: MARS and SVM approximations for $y = \sin^2 x$.

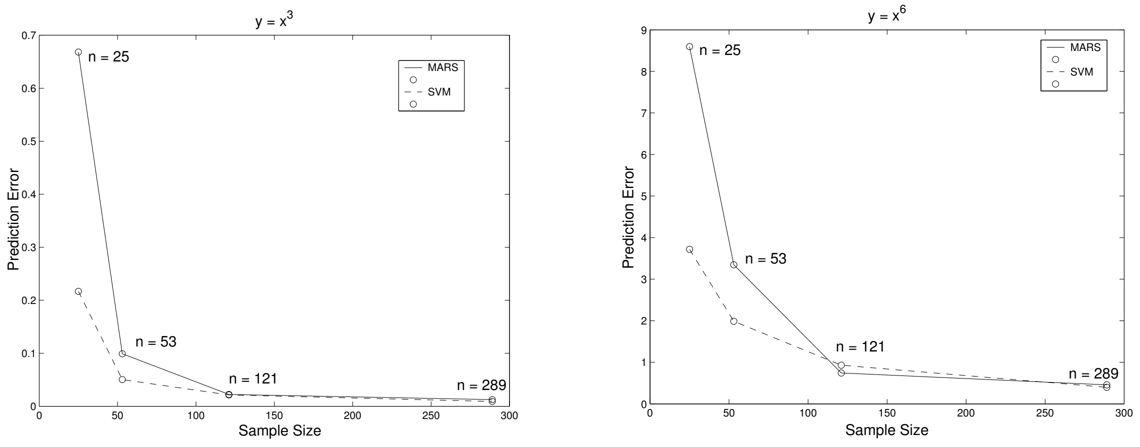
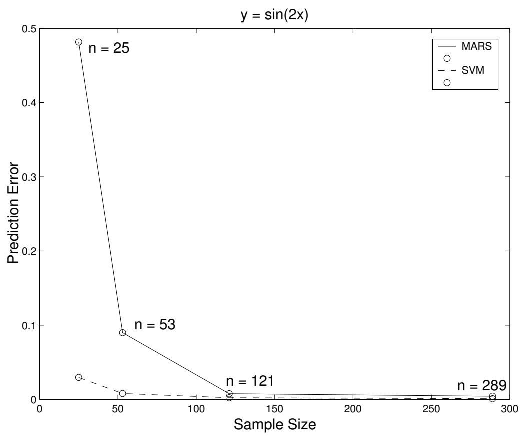


Figure 4.13: Prediction errors for $y = x^3$ and $y = x^6$.



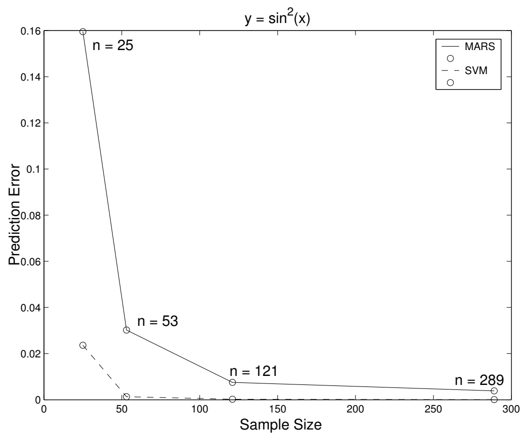


Figure 4.14: Prediction errors for $y = \sin(2x)$ and $y = \sin^2 x$.

Chapter 5

Adaptive Sampling

In this chapter, we begin discussing the second part of this project — adaptive sampling. Although the passive learning techniques discussed in Chapter 2 produce space-filling designs, they sometimes fail to maximize the utility of sample points because they capture redundant information about the function. Adaptive sampling techniques provide a way to judiciously select sample points based on the activity of the function. As illustrated in Figure 5.1, the goal is to place a greater concentration of sample points in areas where the function has more activity and fewer sample points in areas with less activity.

Considering that the exact behavior of the function to be approximated may be unknown, determining how to efficiently distribute sample points within the parameter space presents a major challenge. In practice, it is reasonable to assume that users of function approximation methods know general properties of their target functions. We will begin our study of adaptive sampling methods with an algorithm that generates sets of sample points specifically for monotonic functions, and from there we will develop algorithms that generate sample points for functions with a single minimum. We will end our study of adaptive sampling by investigating an algorithm that targets functions with bounded first derivative.

5.1 The Choose and Learn Algorithm

The first adaptive sampling technique we investigated was Partha Niyogi's Choose and Learn Algorithm (CLA) [11], which can be applied to monotonic functions. A function $f : I \to \mathbf{R}$, where I is an interval, is monotonically increasing in one dimension if for every a and b in I, $a < b$ implies $f(a) \leq f(b)$. It is monotonically decreasing in one dimension if for every a and b in I, $a < b$ implies $f(a) \geq f(b)$. The main idea of the algorithm is to use a worst-case error measure to guide the choice of sample point location. This error measure will be explained in the next section. An outline of the algorithm is provided below.

The Choose and Learn Algorithm (for one-dimensional functions)

1. Let the function be defined on an interval I. Place the first two sample points

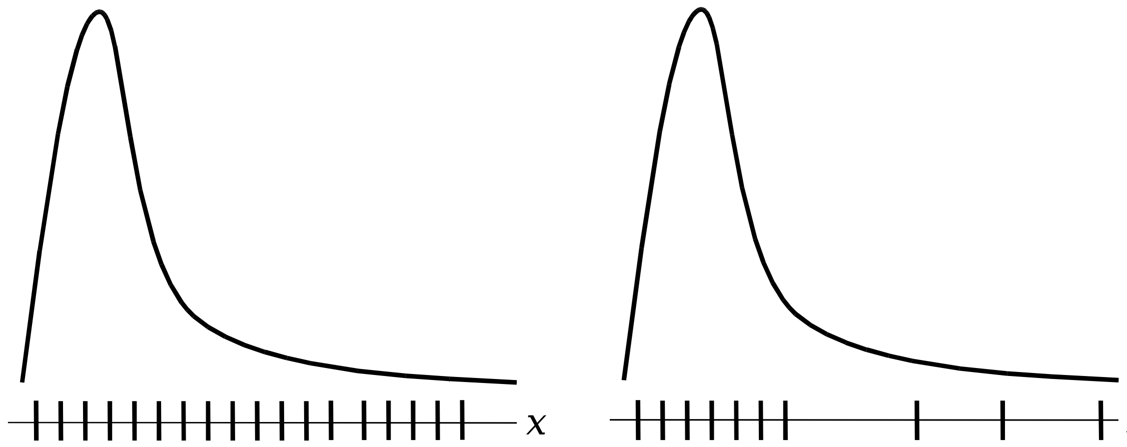


Figure 5.1: 1D comparison of sample point distribution with passive sampling (left) and adaptive sampling (right).

at x_{min} and x_{max}, the left and right endpoints of I respectively.

2. Calculate the error on I.
3. If the error is less than or equal to some fixed tolerance level ε, then terminate the process. Otherwise, divide I into two regions by placing the next sample point at the midpoint of I.
4. Calculate the sum e_D of the errors in all regions.
5. If e_D is less than or equal to ε, then terminate the process. Otherwise, find the region with the largest error and divide it in half by placing the next sample point at its midpoint.
6. Calculate the errors for the newly generated regions and return to step 4.

Figure 5.2 shows a conceptual image of this process after two iterations. This algorithm is adaptive in the sense that with each iteration, it chooses the location of the next sample point based on what it has learned from the data it has seen so far. At every iteration, the algorithm fits a linear approximation to the function in each region. The error in each region is a measure of the greatest possible difference between the linear approximation and the exact function values. We assume that regions in which the function has the largest deviations from the line will have the largest error. It is these regions that will be divided most frequently by the algorithm and consequently will have the greatest number of sample points. By selectively rationing out sample points to the areas that need the most attention, as opposed to uniformly distributing points, the CLA provides a more useful distribution of sample points throughout the parameter space.

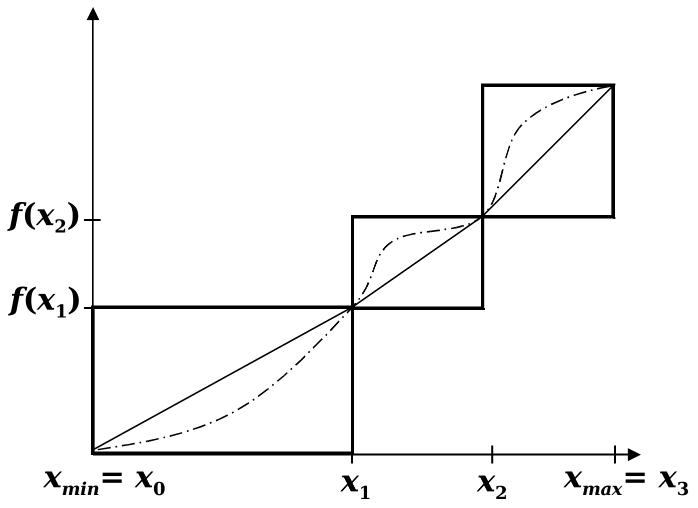


Figure 5.2: CLA applied to a 1D monotonic function after two iterations. The dot-dashed line is the monotonic function, and the solid line is its piecewise-linear approximation.

5.1.1 Implementation of the CLA

For monotonic functions, there are bounds on the range of possible values that the function can take, as indicated by the rectangular boxes in Figure 5.2. The dashed line represents the original function f while the solid line represents the approximating linear function, h, within each interval. It is important to note that because we assume f to be monotonically increasing, the values of f cannot be outside of the solid boxes shown.

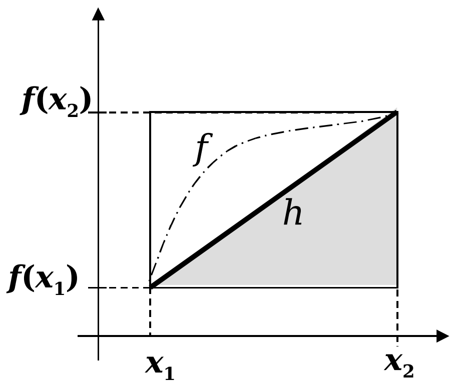


Figure 5.3: Zoomed-in region of domain for which an error is calculated.

Figure 5.3 shows a zoomed-in version of Figure 5.2 that focuses on one region $[x_1, x_2]$, for which we calculate the error e_{C_1}. In general, a region $C_i = [x_i, x_{i+1}]$ has error e_{C_i}, calculated using the following equation (per [11]):

$$e_{C_i} = \left(\int_{C_i} \mid h - f(x_i) \mid^p dx \right)^{1/p} = \frac{1}{(p+1)^{1/p}} (x_{i+1} - x_i)^{1/p} \mid (f(x_{i+1}) - f(x_i)) \mid, \quad (5.1)$$

where $p \geq 1$ is a fixed real number. We used $p = 1$, in which case the error equation simplifies to the formula for the area of a triangle (the shaded section in Figure 5.3):

$$e_{C_i} = \frac{1}{2}(x_{i+1} - x_i) \mid (f(x_{i+1}) - f(x_i)) \mid . \tag{5.2}$$

Since we do not know anything about the behavior of the original function f, except that it increases in each interval $[x_i, x_{i+1}]$, e_{C_i} is the maximum possible error h can have with respect to f. If there are $n + 1$ sample points, then the error e_D over the entire domain is defined by

$$e_D = \left(\sum_{i=0}^{n} e_{C_i}^p \right)^{1/p} . \tag{5.3}$$

Figures 5.4 and 5.5 show the results for the CLA applied to the one-dimensional functions $y = x + \sin x$, $x \in [0, 12]$ and $y = x^3$, $x \in [0, 15]$. The vertical lines represent the location of the sample points selected by the CLA. Clearly the algorithm chooses more points where the function has a steeper slope and fewer points where the function has a flatter slope.

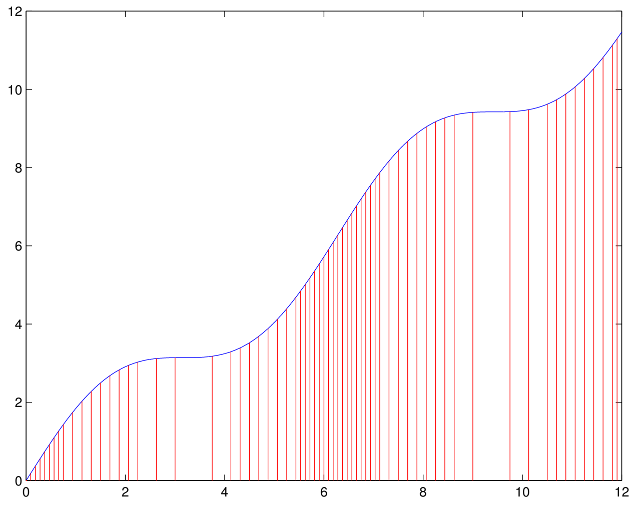


Figure 5.4: CLA applied to $y = x + \sin x$, $x \in [0, 12]$, $\varepsilon = 0.904$.

5.2 The CLA for Two-Dimensional Functions

While implementing the CLA for one-dimensional functions was straightforward, figuring out how to extend it to two dimensions was more complicated, both in terms of keeping track of the different regions and computing the errors in each region. Note that the functions of interest are those that are either monotonically increasing in both variables or monotonically decreasing in both variables. For one-dimensional functions, we calculated the error with respect to a linear approximation h at the endpoints of intervals. In two dimensions, we represent the x_1, x_2 parameter space by

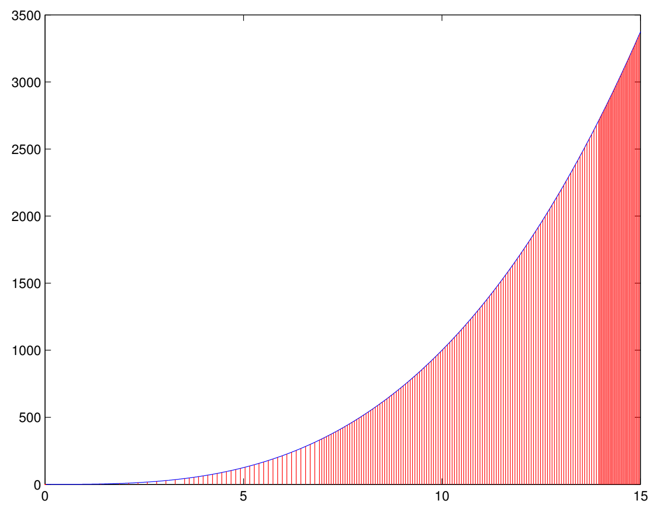


Figure 5.5: CLA applied to $y = x^3$, $x \in [0, 15]$, $\varepsilon = 102.3595$.

a rectangle with corners a, b, c, d as illustrated in Figure 5.6. The function approximation is an affine function $h(x_1, x_2) = \alpha x_1 + \beta x_2 + \gamma$ that has the same values as the original function $f(x_1, x_2)$ at the three corners a, b and d.

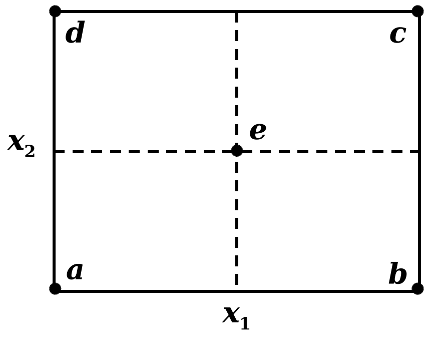


Figure 5.6: Initial parameter space of 2D function.

The error measure is similar to Equation 5.1, only it is based on a double integral over the variables x_1 and x_2. Initially we calculate the error on the rectangular parameter space with corners a, b, c, d. If the error is larger than a fixed ε, then the iterative process begins with locating the center point e of the region and using that point to divide the region into four new subregions (see Figure 5.6). As in the one-dimensional case, we compute the error e_{C_i} of each subregion and then use the sum e_D of the errors, as in Equation 5.3, to determine whether to continue with another iteration or terminate the process.

Figure 5.8 illustrates the application of the CLA for two functions that increase monotonically in two variables, $y = x_1 x_2$ and $y = x_1^2 x_2^2$ (Figure 5.7). As a more dramatic example, Figures 5.9 and 5.10 show two views of a two-dimensional step function, $y = \arctan(40(0.4 - (x_1^5 + x_2^5)^{\frac{1}{5}}))$ — one view from the side and the other from the top. Figure 5.11 illustrates the application of the CLA for this function.

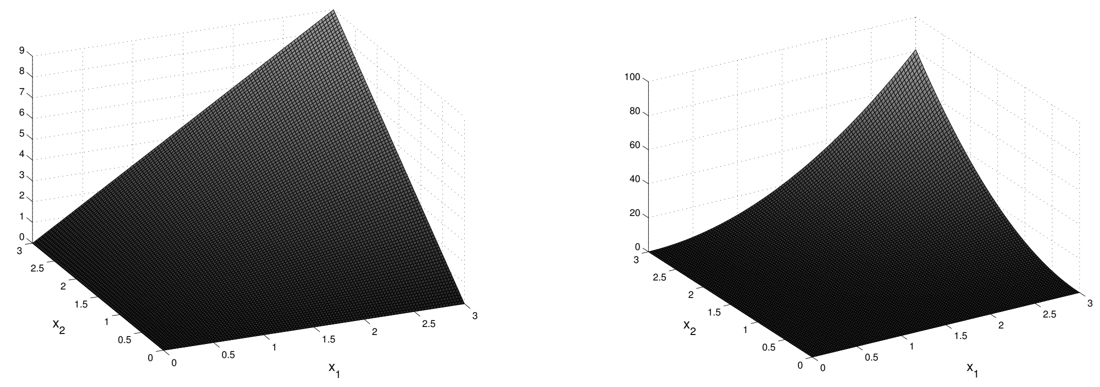


Figure 5.7: $y = x_1 x_2$, $x_i \in [0, 3]$ (left) and $y = x_1^2 x_2^2$, $x_i \in [0, 3]$ (right).

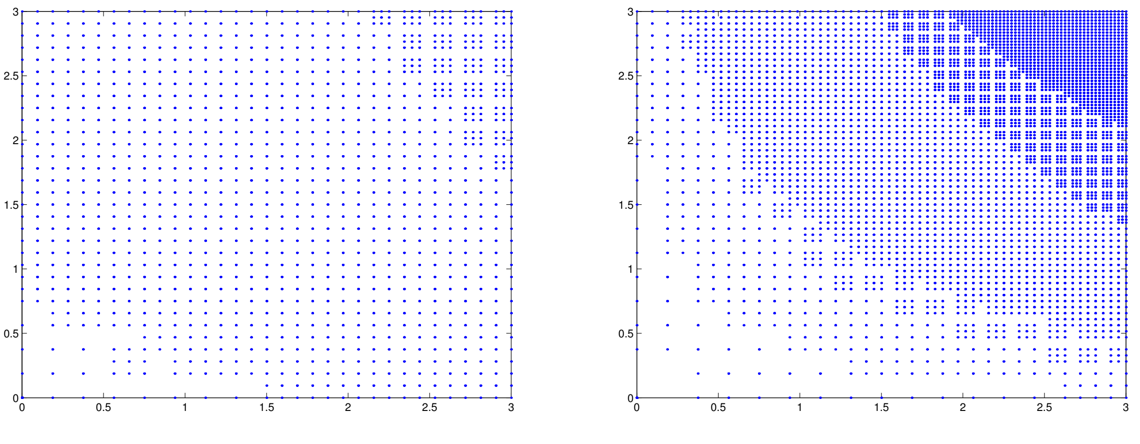


Figure 5.8: Sample points for $y = x_1 x_2$, $\varepsilon = 2.1407$ (left), and $y = x_1^2 x_2^2$, $\varepsilon = 2.15$ (right).

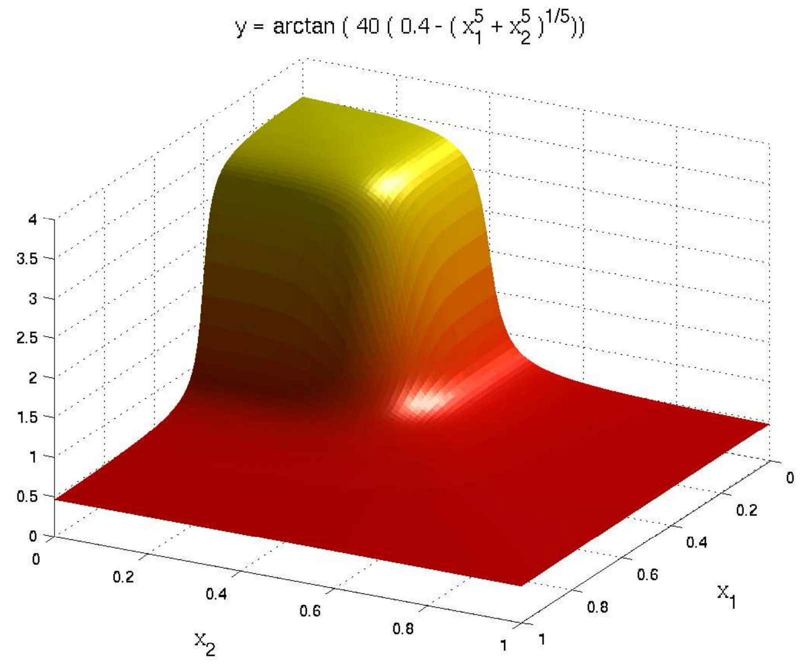


Figure 5.9: $y = \arctan(40(0.4 - (x_1^5 + x_2^5)^{\frac{1}{5}}))$, $x_i \in [0, 100]$ (side view).

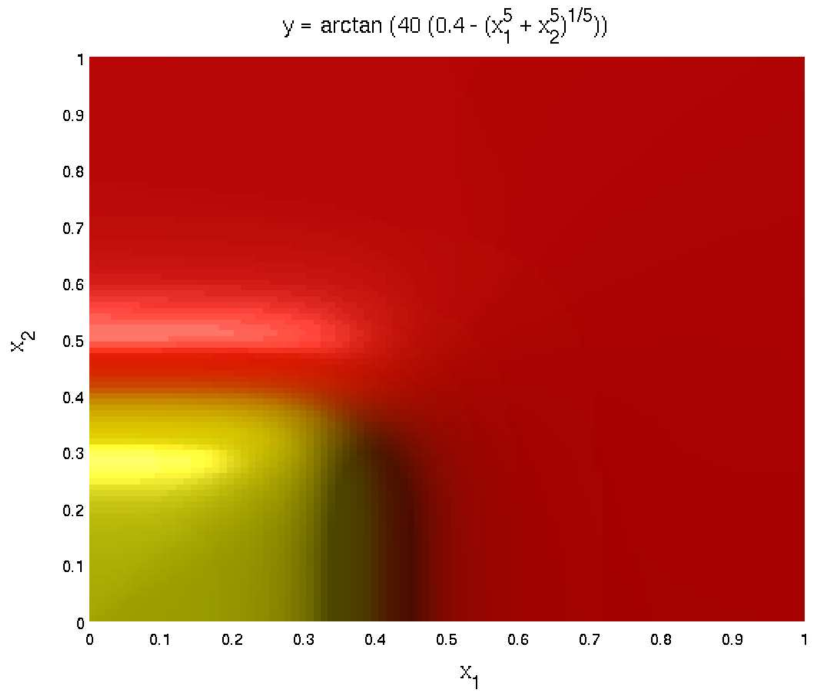


Figure 5.10: $y = \arctan(40(0.4 - (x_1^5 + x_2^5)^{\frac{1}{5}}))$, $x_i \in [0, 100]$ (top view).

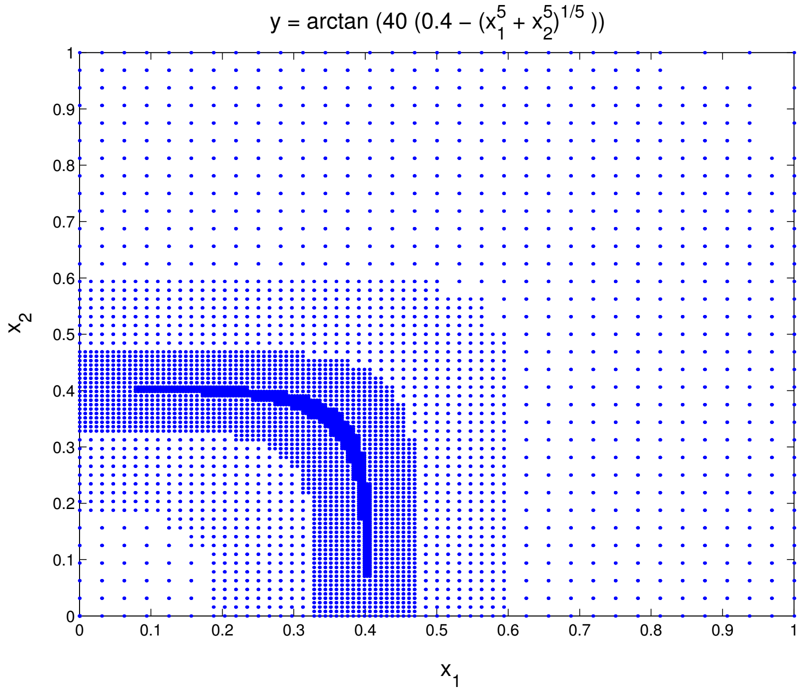


Figure 5.11: Sample points for $y = \arctan(40(0.4 - (x_1^5 + x_2^5)^{\frac{1}{5}}))$, $\varepsilon = 0.001$. Lower-left corner corresponds to the largest value of y in Figure 5.9.

Chapter 6

Adaptive Sampling for Functions with a Single Minimum

Functions with a single minimum are a more complex case for adaptive sampling than monotonic functions. The final part of this project was to develop two sampling algorithms to handle this case.

6.1 Modified Choose and Learn Algorithm

In the previous chapter, we illustrated the application of the original CLA only to monotonically increasing functions. The algorithm works in the same way for monotonically decreasing functions. A function with a single minimum can be divided into two parts, one that is monotonically decreasing and another that is monotonically increasing. Applying the CLA to each part separately produces a sample that again places more points where the function has a steeper slope and fewer points where the function has a flatter slope.

The first step of this *Modified CLA* is finding the approximate location of the minimum in the domain on which the function is defined. This can be accomplished using the following recursive algorithm. First, let the left and right endpoints of the domain be x_{min} and x_{max} respectively. The algorithm returns the endpoints of a smaller interval that contains the minimum.

1. Initialize $a = x_{min}$ and $b = x_{max}$.

2. Find the midpoint of the interval, $x_{mid} := \frac{a+b}{2}$.

3. If $b - a < \delta$, return a and b as the endpoints of the interval containing the minimum.

4. Otherwise, find the corresponding y-values: y_{min}, y_{mid}, and y_{max}.

5. If $y_{min} < y_{mid}$, the minimum must lie in $[a, x_{mid}]$, otherwise we could not have observed an increase in y-values between these two points. In this case, let $b = x_{mid}$ and return to step 2.

6. If $y_{mid} > y_{max}$, the minimum must lie in $[x_{mid}, b]$, otherwise we could not have observed an decrease in y-values between these two points. In this case, let $a = x_{mid}$ and return to step 2.

7. If $y_{mid} \leq y_{min}$, $y_{mid} \leq y_{max}$, or both, we cannot rule out either of the subintervals. In this case, we take $x_{mid\_low} := \frac{a+x_{mid}}{2}$ and $x_{mid\_high} := \frac{x_{mid}+b}{2}$, along with their respective y-values, y_{mid_low} and y_{mid_high}.

8. Re-labeling the points as $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$ in order of increasing x-values, we note that there exists an index i, such that the sequence $\{y_k\}_{k \geq i}$ is monotonically increasing (provided the cardinality of their set is greater than 1) and the sequence $\{y_k\}_{k \leq i}$ is monotonically decreasing. After finding this i, we return to step 2 and focus on the interval $[x_{i-1}, x_{i+1}] \cap [x_0, x_4]$. Note that $(x_{i+1} - x_{i-1}) = \frac{1}{2}(x_4 - x_0)$, so we halve the interval of interest with each recursive step, making the algorithm's running time $O(\log(\delta))$.

The second step of the Modified CLA is to apply the original CLA on two subintervals: $[x_{min}, a]$, on which the function is monotonically decreasing, and $[b, x_{max}]$, on which the function is monotonically increasing.

Our algorithm also offers the user the possibility to determine how many sample points should be created before the algorithm stops. Therefore, the iterative process stops either when the error is less than the threshold, or the number of sample points reaches the maximun number allowed. Since the algorithm partitions the function into two monotonic functions, the number of sample points for each function is determined as follows:

1. Find the length of the original interval.

2. Find the distance from the left endpoint of the original function to the minimum. Note that the function is monotonically decreasing in this interval.

3. Find the distance from the right endpoint of the original function to the minimum. Note that the function is monotonically decreasing in this interval.

4. Find the ratio of the length of each subinterval to the original interval.

5. Use this ratio to assign the corresponding percentage of sample points for each of the two subintervals.

Figure 6.1 illustrates the application of the Modified CLA to the function $y = x^2$ on $[-10, 10]$, for which the single minimum is located at 0. We used $\delta = 0.005$ in the first step to find the approximate location of the minimum and a CLA error tolerance level of $\varepsilon = 7.01$ for both subintervals in the second step.

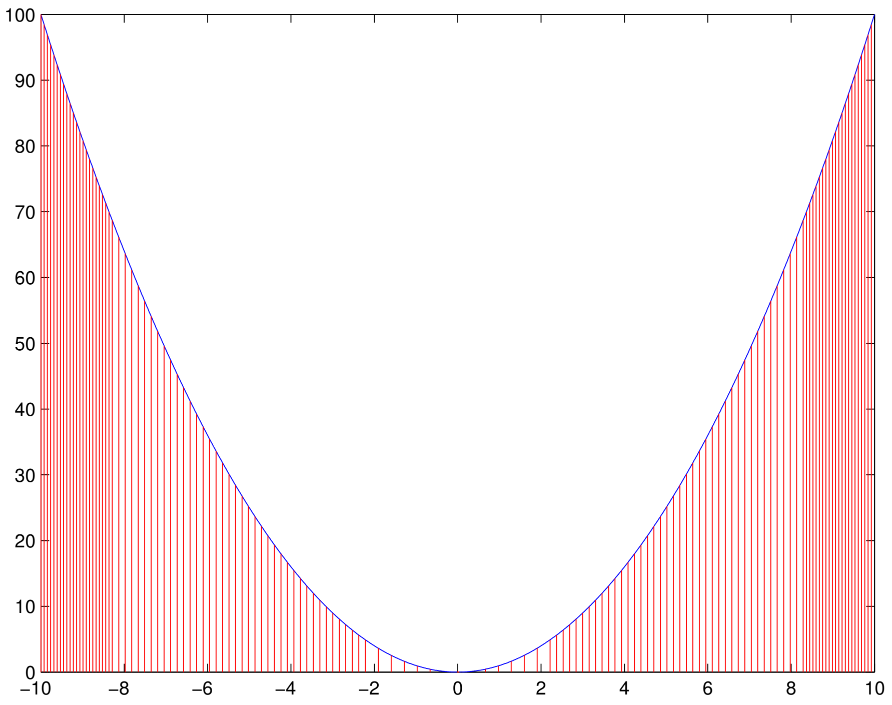


Figure 6.1: Modified CLA applied to $y = x^2$, $x \in [-10, 10]$ with $\delta = 0.005$ and $\varepsilon = 7.01$.

6.2 MARS-Based Algorithm

The second algorithm we implemented was suggested by our industrial advisor, Charles Tong. We call it the MARS-Based Algorithm. Similar to the CLA, the MARS-Based Algorithm is an iterative method that uses the behavior of the function to guide the placement of the next sample point. The key difference between the two methods is that the MARS-Based Algorithm uses standard deviation as the measure of error. The steps of the algorithm are listed below.

1. Select a small number of sample points over $[x_{min}, x_{max}]$ using a passive sampling method (e.g., LP$_\tau$). Add x_{min} and x_{max} to the sample if they are not already present. For convenience, label the sample points $\{x_1, x_2, ..., x_k\}$, where $x_{min} = x_1 < x_2 < \cdots < x_k = x_{max}$. The number of sample points k should be large enough for MARS to be able to generate an approximate function for a user-specified number of basis functions.

2. Generate an approximate function $\hat{f}_k$ using MARS to fit the k sample points.

3. Using a fixed number s of evenly-spaced prediction points, find the mean value of $\hat{f}_k$ on each interval $[x_i, x_{i+1}]$ via the formula

$$m_i = \frac{1}{s} \sum_{q=1}^{s} \hat{f}_k \left(x_i + \frac{q-1}{s-1} (x_{i+1} - x_i) \right), \tag{6.1}$$

and use m_i to compute the standard deviation

$$\sigma_i = \sqrt{\frac{1}{s}\sum_{q=1}^{s}\left(\hat{f}_k\left(x_i + \frac{q-1}{s-1}\left(x_{i+1}-x_i\right)\right) - m_i\right)^2}. \tag{6.2}$$

Store the standard deviation for each interval.

4. Find the interval with the maximum standard deviation, i.e., $[x_{i'}, x_{i'+1}]$ such that $\sigma_{i'} = \max_{1\le i\le k}\sigma_i$. Split this interval in two at its midpoint $x_{k+1} = \frac{x_{i'+1}-x_{i'}}{2}$ and renumber the sample points so that $x_{min} = x_1 < x_2 < \cdots < x_k < x_{k+1} = x_{max}$. (In the implementation of this step, all indices from $i'+1$ to k are increased by 1, and x_k is inserted into the new opening within the array.)

5. Repeat steps 2 – 4 until $\max_{1\le i\le k}\sigma_i < \bar{\sigma}$ or $k = K_{max}$, where $\bar{\sigma}$ and K_{max} are specified by the user.

The MARS-Based Algorithm can be applied directly to both monotonic functions and functions with a single minimum. For example, Figures 6.2 and 6.3 show the application of the MARS-Based Algorithm to $y = x + \sin x$, which is monotonically increasing, and to $y = x^2$, which has a minimum. Figures 5.4 and 6.1 show samples generated by the CLA and Modified CLA for the same two functions.

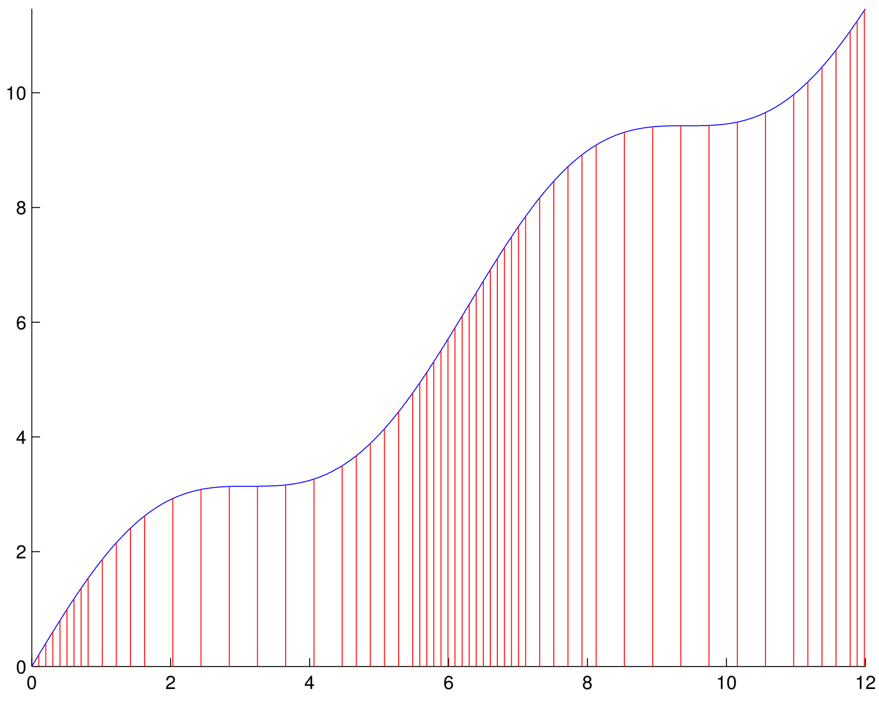


Figure 6.2: MARS-Based Algorithm applied to $y = x + \sin x$, $x \in [0, 12]$, with $\bar{\sigma} = 0.1$.

To test the performance of the two algorithms, we implemented both of them in C. In the next chapter, we compare the effectiveness of the Modified CLA and the MARS-Based Algorithm on one-dimensional functions with a single minimum.

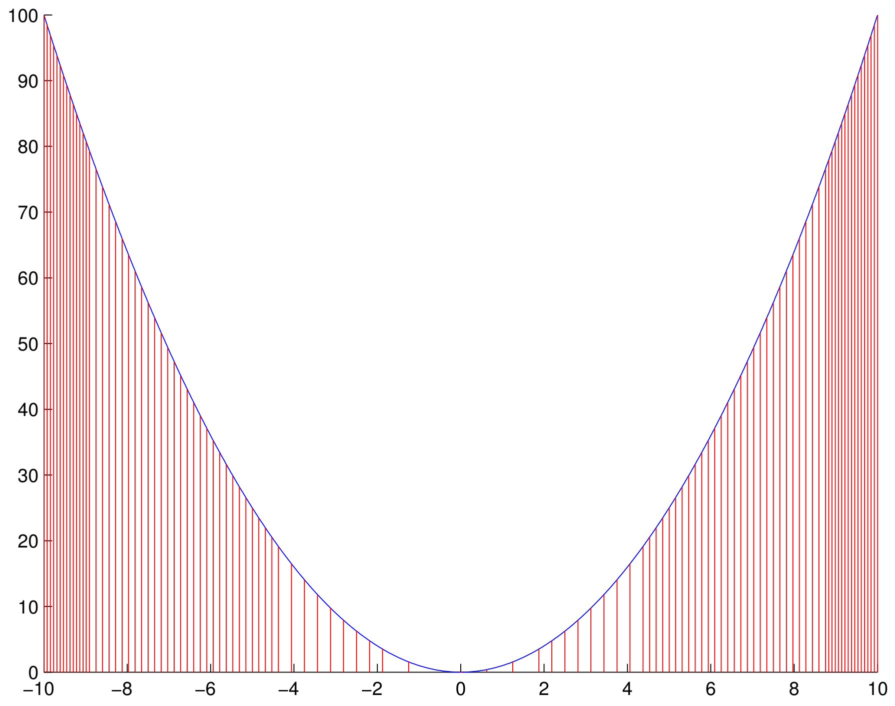


Figure 6.3: MARS-Based Algorithm applied to $y = x^2$, $x \in [-10, 10]$, with $\bar{\sigma} = 0.8$.

Chapter 7

Test Results for Part II

To compare the performance of the Modified CLA and MARS-Based Algorithm, we first applied them to two functions with a single minimum, $y = (x - 0.4)^2 + 0.4$, $x \in [0, 1]$ and $y = \sqrt{|x - 0.6|} + 0.3$, $x \in [0, 1]$, shown in Figure 7.1.

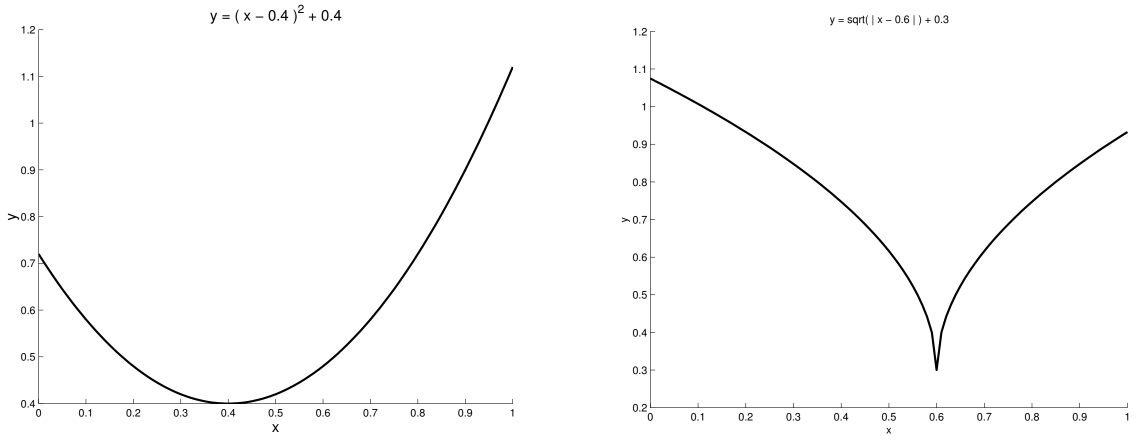


Figure 7.1: $y = (x - 0.4)^2 + 0.4$ (left) and $y = \sqrt{|x - 0.6|} + 0.3$ (right).

We generated samples of size 20, 25, 30, 40, and 50 for both functions using the two algorithms. For the MARS-Based Algorithm, we specified 20 basis functions for every case. For each sample size and each algorithm, we computed a prediction error using 50 points. Figure 7.2 shows these prediction errors.

A third test function was $y = \arctan(50(x^2 - 0.14))$, $x \in [-1, 1]$. Figures 7.4 – 7.6 show the application of the two algorithms to this function for sample sizes of 50, 100, and 200. The curve is the exact function, and the vertical lines represent the placement of the sample points. The plots on the left were all generated using the Modified CLA; the plots on the right were all generated using the MARS-Based Algorithm. In applying the MARS-Based Algorithm, we used 50 basis functions in every case.

After generating the samples, we used the sample points to fit a MARS approximation. Figures 7.7 – 7.9 show the approximate functions that correspond to the samples shown in Figures 7.4 – 7.6. Both algorithms result in an accurate approxima-

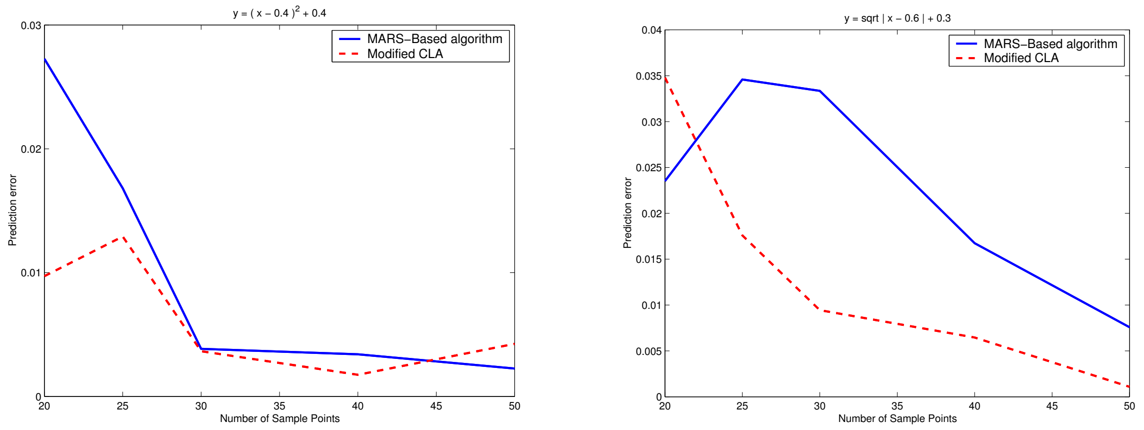


Figure 7.2: Modified CLA and MARS-Based Algorithm prediction errors for $y = (x - 0.4)^2 + 0.4$ (left) and $y = \sqrt{|x - 0.6|} + 0.3$ (right).

tion for 200 sample points, but one major difference is evident for the smaller sample sizes. It appears that, due to differences in the location of the sample points, the MARS-Based Algorithm approximates the function better than the Modified CLA near the minimum, but the Modified CLA approximates the function better than the MARS-Based Algorithm near the endpoints of the interval.

Figure 7.3 shows prediction errors for different samples generated by three different sampling methods: LP$_\tau$, the Modified CLA, and the MARS-Based Algorithm. For this particular test function, $y = \arctan(50(x^2 - 0.14))$, $x \in [-1, 1]$, the errors for the two adaptive sampling methods were comparable to each other and noticeably lower than those for LP$_\tau$, demonstrating the effectiveness of adaptive sampling.

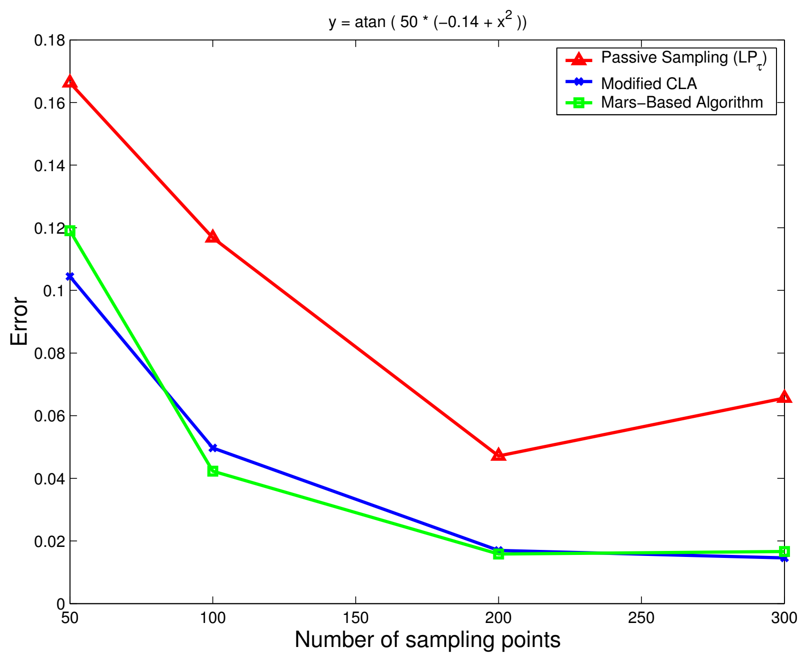


Figure 7.3: Prediction errors for approximating $y = \arctan(50(x^2 - 0.14))$ using different sampling methods.

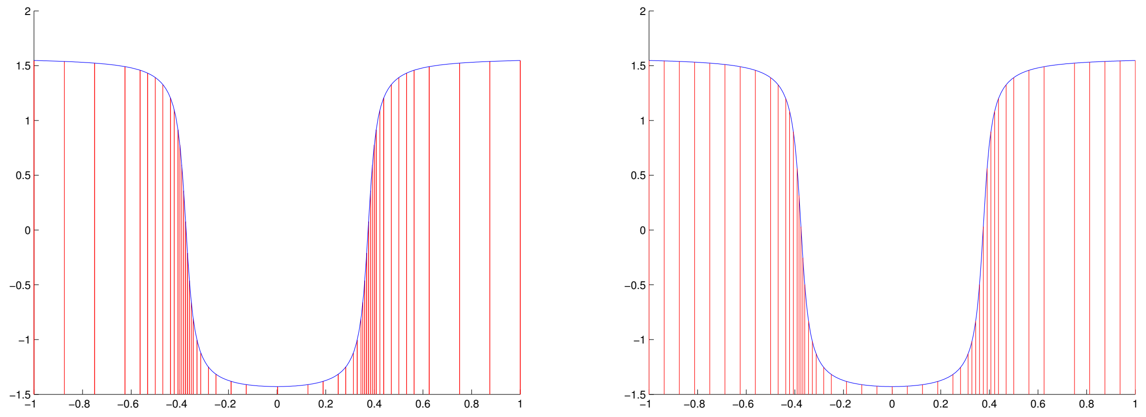


Figure 7.4: Modified CLA (left) and MARS-Based Algorithm (right), 50 points.

Conclusions

Based on the prediction errors from the three tests, samples produced by the Modified CLA and MARS-Based Algorithm result in function approximations with about the same level of accuracy for a given sample size. In terms of implementation for one-dimensional functions, the Modified CLA is simpler, primarily because it does not involve the use of MARS. As discussed in Chapter 4, the MARS software requires the user to choose certain parameter values, such as the number of basis functions, and it is not always obvious which values will work best.

However, extending the algorithms to two-dimensions would be more complicated

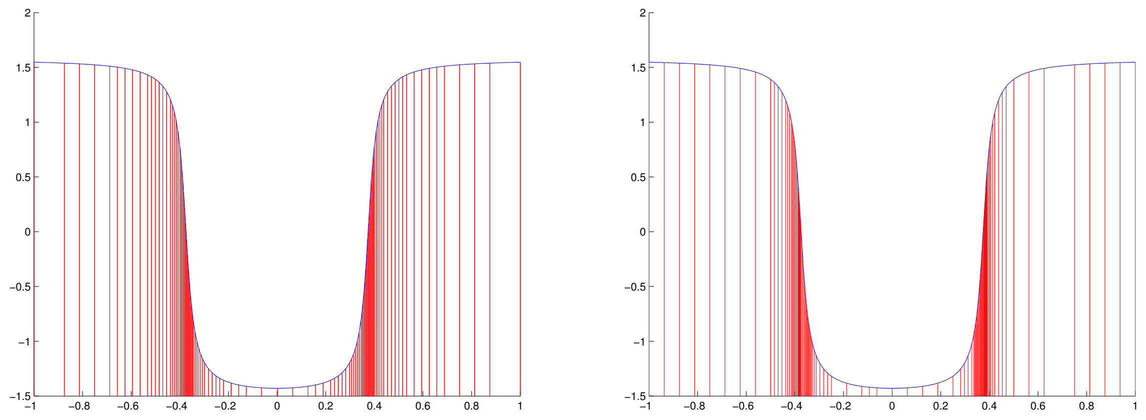


Figure 7.5: Modified CLA (left) and MARS-Based Algorithm (right), 100 points.

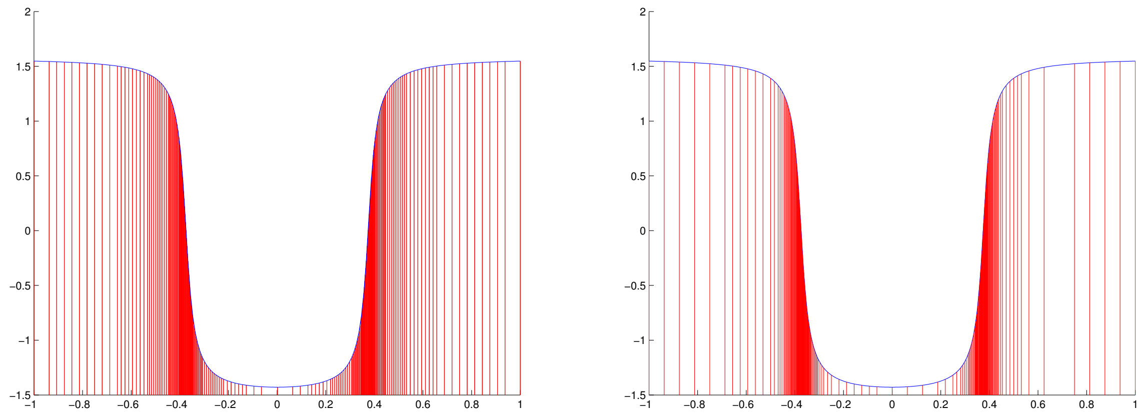


Figure 7.6: Modified CLA (left) and MARS-Based Algorithm (right), 200 points.

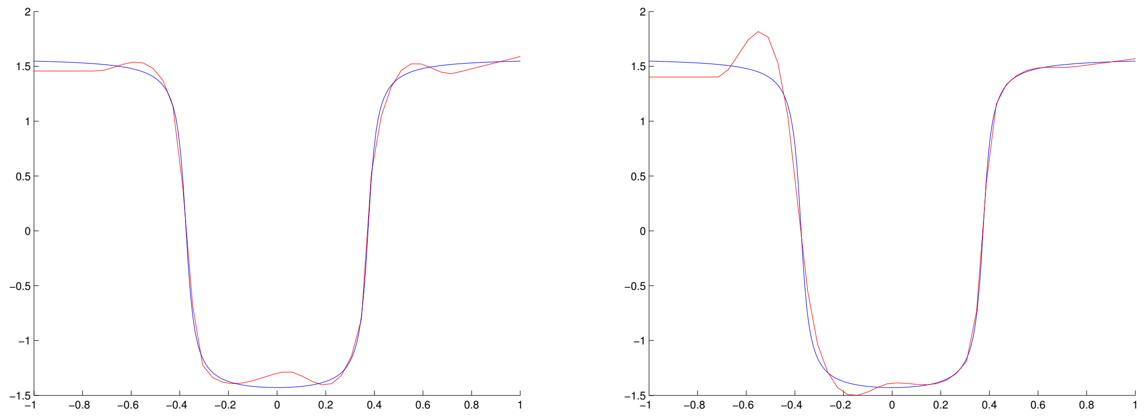


Figure 7.7: Modified CLA (left) and MARS-Based Algorithm (right), 50 points.

for the Modified CLA than for the MARS-Based Algorithm because finding the minimum of a two-dimensional function is not trivial, and because computing the error involves solving a double integral. The MARS-Based Algorithm does not require knowing the approximate location of the minimum, and the standard deviation is not

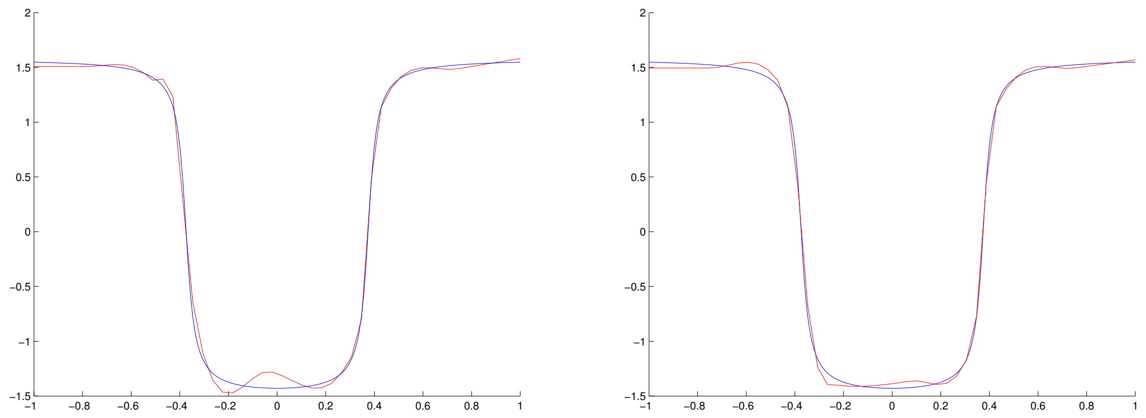


Figure 7.8: Modified CLA (left) and MARS-Based Algorithm (right), 100 points.

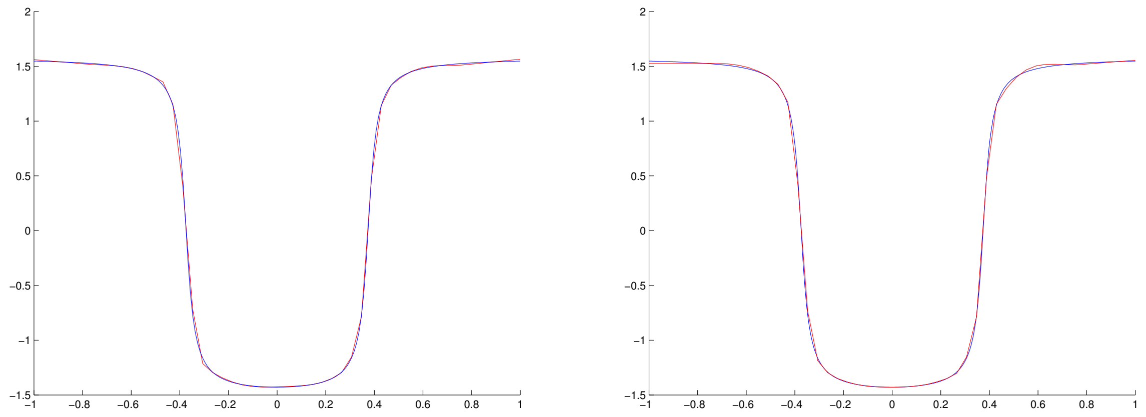


Figure 7.9: Modified CLA (left) and MARS-Based Algorithm (right), 200 points.

difficult to compute for two-dimensional functions.

Chapter 8

Adaptive Sampling for Functions with a Bounded First Derivative

The last class of functions that we considered in this project was functions with a bounded first derivative, i.e., $f : X \to Y$ such that f is differentiable and $|f'(x)| \leq d$ for x in an interval of interest I and some $d \in \mathbb{R}$. Knowing the maximum absolute value d of f' and the value of f at one point, we can also bound the value of f at other points in I because the slope m between any two points in I satisfies $|m| \leq d$. This principle serves as a basis for a variation on the Choose and Learn Algorithm, which we will call CLA-2. The CLA-2 returns a set of sample points using the following steps:

1. Let x_{min} and x_{max} be the endpoints of I. These will be the first two sample points.

2. Evaluate $y_{min} = f(x_{min})$ and $y_{max} = f(x_{max})$.

3. Calculate the error
$$e_I = \frac{1}{4d}(d^2(x_{max} - x_{min})^2 - |y_{max} - y_{min}|^2). \tag{8.1}$$

4. If e_I is greater than some fixed tolerance level ϵ, then divide I into two regions by placing the next sample point at the midpoint of I.

5. Let the $k + 1$ sample points generated thus far be $x_{min} = x_0 < x_1 < \cdots < x_{k+1} = x_{max}$ and $y_i = f(x_i)$. For each region $C_i = [x_i, x_{i+1}]$, $i = 0, 1, \ldots, k$, compute the error
$$e_{C_i} = \frac{1}{4d}(d^2(x_{i+1} - x_i)^2 - |y_{i+1} - y_i|^2). \tag{8.2}$$

6. Compute the total error $e_D = \sum_{i=0}^{k} e_{C_i}$. If $e_D \leq \epsilon$, then terminate the algorithm. Otherwise, find the region with the largest error and divide it in half by placing the next sample point at its midpoint. Repeat steps 4–6.

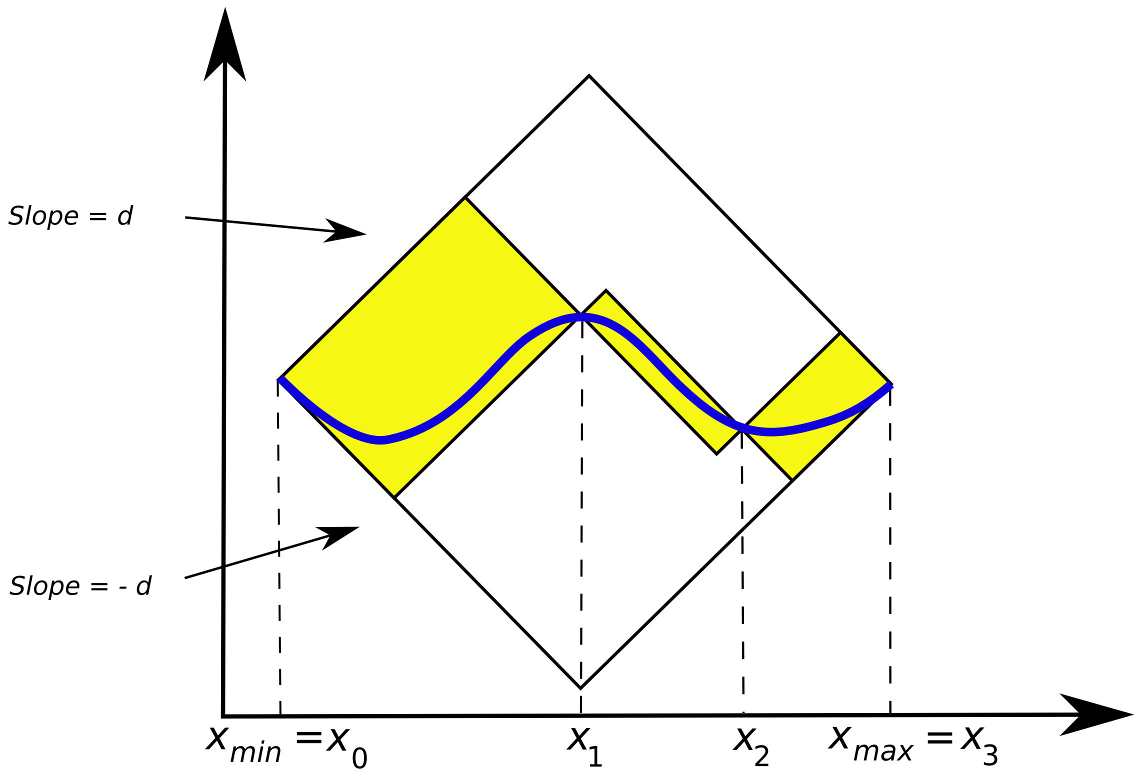


Figure 8.1: First two iterations of the CLA-2 for $y = \sin x$

Figure 8.1 shows an example of the first two iterations of the CLA-2 Algorithm. The shaded parts represent the regions in which the function can exist, bounded by lines with slopes of absolute value equal to the maximum absolute value d of the function's first derivative. The area of each shaded part is the error of the corresponding region, given by Equation 8.2. The idea is that because the function cannot exist outside of the shaded regions without violating the bound on the first derivative, if the area of the shaded regions is small, then there are enough sample points that any approximation to the function should be fairly accurate.

Figures 8.2–8.4 show the application of the CLA-2 to three functions: $y = x^3$, $y = \sin x$ and $y = x \cos x - \sin x$ respectively. An important feature of this algorithm is that it places more sample points where the function has a flat slope than where it has a steep slope. The reasoning behind this behavior is that if the function values at x_i and x_{i+1} are close, then it is possible that the function oscillates on the interval $[x_i, x_{i+1}]$. On the other hand, if $f(x_{i+1})$ is much greater than $f(x_i)$, then considering the bound d on f', the function would have to change steadily over $[x_i, x_{i+1}]$. That is, in the second case, we could reasonably approximate the function by a line, so it is not necessary to place many sample points in that region. Note that this is the exact opposite of how the CLA and MARS-Based Algorithm choose sample points. In the case of CLA, the function was monotonic, so if $f(x_{i+1})$ differed little from $f(x_i)$, that meant that the function could only change little, while if $f(x_{i+1})$ differed a lot from $f(x_i)$ that meant that the function can change in many ways on $[x_i, x_{i+1}]$.

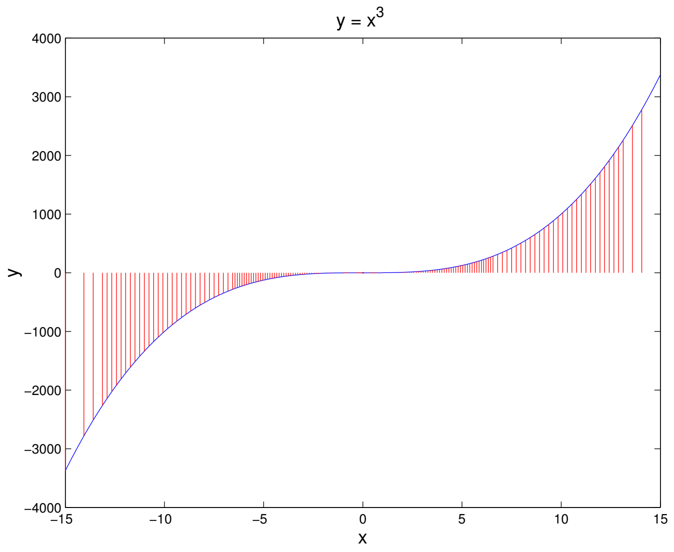


Figure 8.2: $y = x^3$ with $\epsilon = 5$ and $d = 675$, $x \in [-15, 15]$.

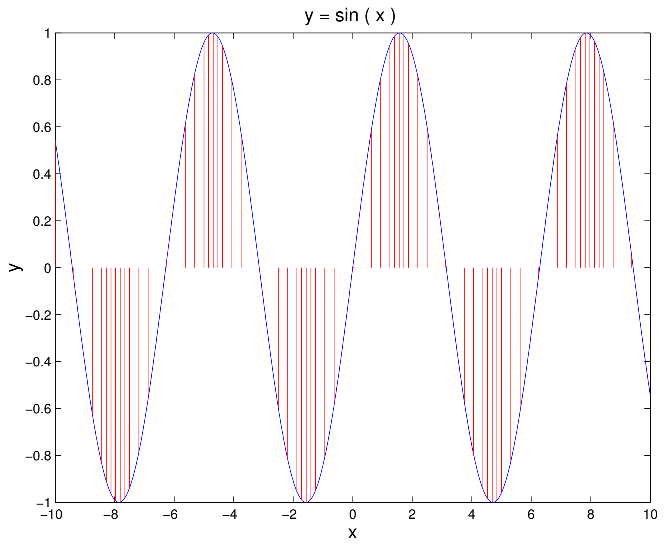


Figure 8.3: $y = \sin x$ with $\epsilon = .02$ and $d = 1$, $x \in [-10, 10]$.

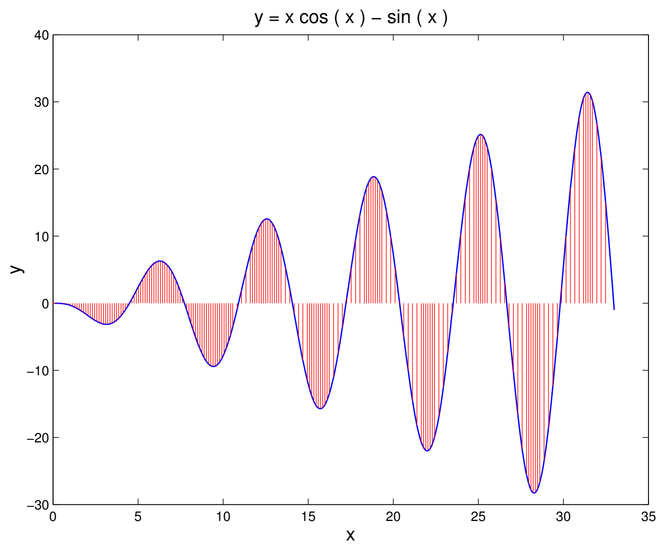


Figure 8.4: $y = x\cos x - \sin x$ with $\epsilon = .5$ and $d = 10.5\pi$, $x \in [0, 10.5\pi]$

Chapter 9

Conclusion

The subject of *response surface methodology* encompasses an enormous variety of techniques to achieve useful experimental designs and accurate function approximations. In this project, we examined four passive sampling methods, two function approximation methods, and four adaptive sampling methods. We obtained code to implement all of these methods, either from an outside source or by programming them ourselves, as shown in Table 9.1.

Method	**Author of Code**	**Language**
Monte Carlo	LLNL team	C
LP$_\tau$	Boris Shukhman	C
Latin Hypercubes	MATLAB (*lhsdesign*)	MATLAB
OA-Based Latin Hypercubes		
Construction of Orthogonal Arrays	Art Owen	C
Driver	LLNL team	C
MARS		
Subroutines to generate response surface	Jerome Friedman	Fortran 77
Driver	LLNL team	C
SVM		
Subroutines to generate response surface	Thorsten Joachims	C
Driver	LLNL team	C
CLA	LLNL team	C
Modified CLA	LLNL team	C
MARS-Based Algorithm	LLNL team	C
CLA2	LLNL team	C

Table 9.1: List of software.

Note that we acquired code to generate an orthogonal array, but we wrote our own code to produce a set of sample points using the orthogonal array. Likewise, we acquired code containing functions to implement MARS and SVM, but we wrote our own programs to call those functions and compute errors.

9.1 Summary of Findings

Regarding passive sampling methods, we found that the LP$_\tau$ method returns a well-spaced sample and will, in fact, always generate the same well-spaced set of points for a given sample size. Its deterministic nature, along with its ability to create a set of sample points for which the subsequent function approximation has relatively high accuracy, made it a more useful method than Monte Carlo, Latin Hypercubes, and Orthogonal-Array-Based Latin Hypercubes.

As for function approximation methods, we determined that even though it is possible for SVM to achieve higher accuracy than MARS, SVM is also more difficult to use. The default values for several of the parameters in the SVM software were not the best for our test functions, but varying the parameter values was problematic because it was not always clear whether changing a value would actually result in a better approximation. Moreover, depending on the function and the combination of parameter values, SVM sometimes ran too slowly to be feasible. The MARS software also had a few parameters to tune, but after some initial tests it was not complicated to predict how the method would respond to modifying their values.

We used MARS exclusively to create function approximations in the second half of this project, in which we investigated different adaptive sampling schemes with a focus on functions with a single minimum. For this special case, we developed two algorithms, the Modified CLA and the MARS-Based Algorithm, that aimed to place more sample points in regions of the function domain on which the function varies greatly. We discovered that the two algorithms produce sets of sample points that are not significantly different in terms of the accuracy of the function approximations generated to fit them. However, there are differences in ease of implementation. The Modified CLA was more straightforward to implement for one-dimensional functions, but would be more difficult to implement for two-dimensional functions.

9.2 Future Research

The tests described in Chapter 4 used only low-dimensional functions. Because of time constraints, we were not able to test more complicated functions, but since function approximation is supposed to be a solution to the problem of evaluating computationally expensive functions, it would be constructive to test the methods with functions of high dimension. Also, the SVM software should be studied more thoroughly to determine if there is a means of choosing optimal parameter values. SVM was originally developed as a learning tool for classification, not regression, but if its regression capability is to be exploited, then there should be more documentation on how to make the best use of the software.

In addition, the adaptive sampling algorithms researched for this project applied to just three classes of functions, those that are monotonic, those with a single minimum and those with a bounded first derivative. Advancing this research includes formulating the algorithms for more complicated functions of higher dimension, and investigating methods to handle functions with other given properties.

Bibliography

[1] P. Bratley and B.L. Fox. Algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):88–100, 1988.

The introduction of this article provided a brief explanation of the mathematics behind the LP$_\tau$ method.

[2] Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

This paper provides an in-depth tutorial of SVM with an emphasis on its application in pattern recognition.

[3] Kai Tai Fang, Runze Li, and Agus Sudjianto. *Design and Modeling for Computer Experiments*. Chapman & Hall/CRC, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487, 2006.

[4] Aly Farag and Refaat M. Mohamed. Regression using support vector machines: Basic foundations. Technical report, Computer Vision and Image Processing Laboratory, Electrical and Computer Science Department, University of Louisville, December 2004.

This paper outlines the use of Support Vector Machines for finding regression.

[5] J.H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991.

This paper describes the existing methodology for multivariate regression modeling and then presents the algorithm for the Multivariate Adaptive Regression Splines (MARS) method.

[6] A.S. Hedayat, N.J.A. Sloane, and John Stufken. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

This book outlines the theory of orthogonal arrays, providing the basic definitions and describing Bush's construction.

[7] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science and Information Engineering, National Taiwan University, 2003.

This paper provides an introduction to SVM, chiefly in relation to its use for data classification.

[8] Thorsten Joachims. `SVM`$^{\texttt{light}}$: Support vector machine. *SVM-Light Support Vector Machine, University of Dortmund, November*, 1999.

This web site offers instructions on running `SVM`$^{\texttt{light}}$.

[9] J.R. Koehler and A.B. Owen. Computer experiments. *Handbook of Statistics*, 13:261–308, 1996.

This article provides a good introduction to the concept of computer experiments and also outlines the Latin hypercube and orthogonal-array-based Latin hypercube sampling methods.

[10] Max D. Morris and Toby J. Mitchell. Exploratory designs for computational experiments. *Journal of Statistical Planning and Inference*, 43(3):381–402, 1995.

This paper discusses the various sampling designs for computer experiments, outlining the maximin criterion used to strengthen Latin hypercubes.

[11] Partha Niyogi. Active learning by sequential optimal recovery. Technical report, Artificial Intelligence Laboratory, Center for Biological and Computational Learning, Massachusetts Institute of Technology, March 1995.

This paper gave an introduction to adaptive sampling and outlined the Choose and Learn Algorithm.

[12] Art B. Owen. Orthogonal arrays for computer experiments, integration and visualization. *Statistica Sinica*, 2(2):439–452, 1992.

This paper provides a detailed outline of orthogonal arrays and their use in computer experiments.

[13] Timothy W. Simpson, Dennis K.J. Lin, and Wei Chen. Sampling strategies for computer experiments: Design and analysis. *International Journal of Reliability and Applications*, 2(3):209–240, 2001.

This paper presents various sampling methods commonly used in computer experiments, including Latin hypercubes.

[14] Boxin Tang. Orthogonal array-based latin hypercubes. *Journal of the American Statistical Association*, 88(424):1392–1397, December 1993.

This paper outlines the nature of orthogonal array-based Latin hypercubes as a sampling method.