Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

ADAPTIVE GLOBAL OPTIMIZATION ALGORITHMS

by
William Phillips

Global optimization is concerned with finding the minimum value of a function where many local minima may exist. The development of a global optimization algorithm may involve using information about the target function (e.g., differentiability) and functions based on statistical models to better the worst case time complexity and expected error of similar deterministic algorithms.

Recent algorithms are investigated, new ones proposed and their performance is analyzed. Minimum, maximum and average case error bounds for the algorithms presented are derived. Software architecture implemented with MATLAB and Java is presented and experimental results for the algorithms are displayed.

The graphical capabilities and function-rich MATLAB environment are combined with the object oriented features of Java, hosted on the computer system described in this paper, to provide a fast, powerful test environment to provide experimental results. In order to do this, matlabcontrol, a third party set of procedures that allows a Java program to call MATLAB functions to access a function such as voronoi() or to provide graphical results, is used. Additionally, the Java implementation can be called from, and return values to, the MATLAB environment. The data can then be used as input to MATLAB's graphing or other functions.

The software test environment provides algorithm performance information such as whether more iterations or replications of a proposed algorithm would be expected to provide a better result for an algorithm. It is anticipated that the functionality provided by the framework would be used for initial development and analysis and subsequently removed and replaced with optimized (in the computer efficiency sense) functions for deployment.

ADAPTIVE GLOBAL OPTIMIZATION ALGORITHMS

by
William Phillips

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Sciences, NJIT

January 2015

APPROVAL PAGE

ADAPTIVE GLOBAL OPTIMIZATION ALGORITHMS

William Phillips

Dr. James M. Calvin, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. Daochuan C. Hung, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Marvin K. Nakyama, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Zhi Wei, Committee Member — Date
Assistant Professor of Computer Science, NJIT

Dr. Antanas Zilinskas, Committee Member — Date
Professor of Informatics, Institute of Mathematics and Informatics, Vilnius, Lithuania

BIOGRAPHICAL SKETCH

Author: William Phillips

Degree: Doctor of Philosophy

Date: January 2015

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2015
- Master of Science in Telecommunications,
  Southern Methodist University, Dallas, TX, 2005
- Master of Software Engineering,
  Seattle University, Seattle,WA, 1985
- Bachelor of Science in Mathematics,
  University of Washington, Seattle, WA, 1979

Major: Computer Science

Presentations and Publications:

William Phillips, "Design Patterns as a Practical Software Engineering Tool," Presentation, Manhattan College, New York, NY. June, 2014

William Phillips, "Adaptive Global Optimization Algorithms," Presentation, Murray State University, Murray, KY. July, 2014

William Phillips, "Design Patterns," Presentation, Fairleigh Dickinson University, Teaneck, NJ. July, 2014

To Jesus Christ and my Mother, Jessie Marie (Brown) Phillips

ACKNOWLEDGMENT

I would like to thank my advisor, James M. Calvin Ph.D, for the direction, support, knowledge and most of all patience. I would like to thank the National Science Foundation for their support in this work. I also want to thank my committee for the time spent in working with me to complete this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Introduction

Optimization has grown out of the field of applied analysis and operations research to meet the need of finding the extremal values of a function. There are many important engineering, scientific and financial applications.

1.2 Problem Statement

The problem addressed in this work is to provide the interested engineer, scientist or financial analyst with a set of performance and error analyzed global optimization algorithms, along with a framework which will assist in the development of additional algorithms. The framework uses well-known design patterns implemented in the Java programming language.

The framework is fully integrated with MATLAB, in that MATLAB mathematical and display functions can be called from the global optimization algorithm as it executes. Values stored by the application can be returned to the MATLAB environment for further processing or display by the MATLAB environment.

1.3 Background

The best approach for finding an extreme value, x^*, where $F(x^*)$ achieves an extreme, on a set D, provided an exact value can not be found by using analytical techniques, is to exploit information about the objective function $F(\cdot)$ in deriving and applying an algorithm to search for $F(x^*)$. For example, if it is known that a harmonic function $F(\cdot)$ only has extremal points on the boundary of D, we know that if $m \leq F(p) \leq M$ holds on the boundary of D it also holds in the interior of D as well. A harmonic

functions is a function $F(\cdot)$ such that if $f \in C''$ in n variables $\sum_{x=1}^{n} F_{x_i x_i} = 0$ everywhere in D. [2]

Another attribute of $F(\cdot)$ that may be exploited is if $F(\cdot)$ satisfies a Lipschitz condition, that is $|F(x_1) - F(x_2)| < L|x_1 - x_2| \; \forall x_1, x_2 \in D$, where L is the Lipschitz constant. This information can be used to increase the efficiency of a passive grid algorithm, an algorithm where information of the objective function is not considered in choosing new points. Zhigljavsky in [21] demonstrates this by increasing the efficiency of the uniform random search algorithm (where the minimum $F(x_n^*) = min\{F(x_i)\}, \; i = 1, \ldots, n$ where $\{x_i\}$ is a set of n uniformly distributed points over D) by making use of the Lipschitz condition on the objective function and a method of Devroye [21]:

Nonuniform Random Covering Algorithm:

Step 1: Set $k = 0, \; f_0^* = \infty, \; Z_1 = \chi$, a set of n uniformly distributed points over D.

Step 2: Set $k = k + 1$.

Step 3: Set x_k to a sample from P_{Z_k}, some probability distribution.

Step 4: Set $f_k^* = min\{f_{k-1}^*, f(x_k)\}$.

Step 5: Set $\eta_i = (f(x_i) - f_k^* + \delta) \; / \; L$ for each $i = 1, ..., k$ and $\delta =$ some small number.

Step 6: Set $Z_{k+1} = \chi \backslash \bigcup_{i=1}^{k} B(x_i, \eta_i, \rho)$ where $B(x_i, \eta_i, \rho)$ is the set $\{z \in X : \rho(x_i, z) \leq \eta_i\}$ and ρ is an acceptable metric.

Step 7: If Z_{k+1} is not empty or other stopping criteria not met, go to Step 2.

This algorithm finds a global minimizer with accuracy δ. In other words, $f_k^* - f \leq \delta$ where k is the last index for which Z_{k+i} in the preceeding algorithm, is empty.

In addition to satisfying a Lipschitz condition, other function attributes that can be exploited include the continuity of the function, the smoothness of the function (how many times it is differentiable), the number of local minima (modality) and the number of variables of $F(\cdot)$.

In solving Global Optimization (GO) problems and developing algorithms it is important to consider the accuracy of the solution provided, the complexity in terms of running time and storage required, and the tools (both hardware and software) used in implementing them.

It should also be noted that the problem of finding the minimum value of $F(\cdot)$ on D is equivalent to finding the maximum because finding the minimum of $F(\cdot)$ is equivalent to finding the maximum of $-F(\cdot)$.

Global optimization algorithms fall into the categories described in the following sections:

1.4 Global Optimization Examples

1.4.1 Groundwater Remediation

One area where global optimization algorithms may be applied is in the case of groundwater remediation, where an area of groundwater has been contaminated. To clean up the site, several points are chosen where the water is pumped out and replaced with clean water. The optimization problem is to select the locations and pumping rates for each well in order to minimize the time required to clean up the site.

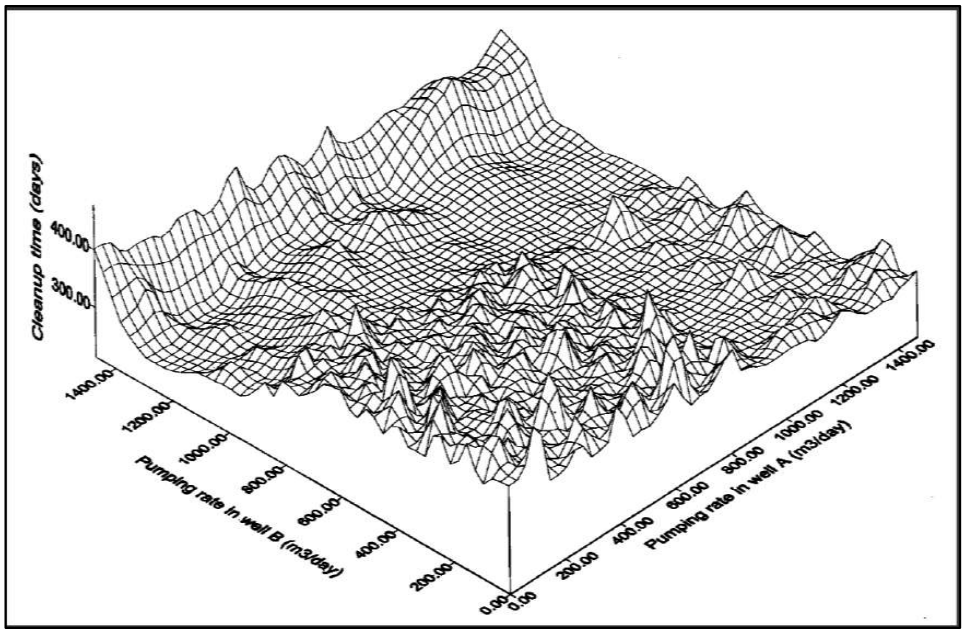

Figure 1.1 A multimodal objective function (from Maskey et al.[18]).

Figure 1.1 shows the performance (cleanup time) as a function of two parameters (pumping rates at different wells) for an aquifer cleanup problem described in [18]. In this case the domain D is the 1000x1000 meter area of the remediation site. The objective function is the two variable function (cleanup time) of the two parameters, pumping rates in wells A and B.

1.4.2 Linear Registration and Motion Correction of Brain Images

Jenkinson in [16] describes an application for global optimization in the field of functional brain image analysis. Here GO algorithms are applied to correct for movement of the target organ (brain) under diagnostic or testing conditions. The requirement for performing this work requires optimizing one of several intensity-based

cost functions. The majority of work has been concentrated on the computational efficiency of the chosen algorithm and less on the quality of the solution. Local optimization algorithms are used.

By using global optimization algorithms such as those described in this paper, the problem of optimization algorithms getting "stuck" at a local extremal value would be solved. The authors of [16] propose the use of multiple-stage algorithms where initially the algorithm performs a coarse measurement on the objective function. Subsequently, more refined algorithms are used to explore promising areas of the domain. The algorithms in the paper combine both global and local information to search for extremal values. However, this multiple stage approach shows promise to further increase the effectiveness of the type of algorithms presented in this thesis.

1.5 Applied Optimization Techniques

Optimization techniques include semidefinite programming, combinatorial optimization, quadratic programming, nonlinear programming, deterministic global optimization, integer programming, algorithms based on radial basis functions, simulated annealing and stochastic programming algorithms. This thesis focuses on adaptive algorithms and presents a number of these algorithms, analyzes their efficiency and complexity, and presents implementations for them.

1.6 Adaptive, Stochastic Global Optimization Algorithms

A stochastic global optimization algorithm is an algorithm that uses probabilistic techniques to determine where within the domain D is the best place to test the objective function for the next extremal value. An adaptive global optimization algorithm uses past information as input to the algorithm to determine where within the domain D to evaluate the objective function.

An example of an adaptive algorithm is the P-Algorithm for univariate functions presented in Chapter 4. The algorithm splits the univariate (one dimensional) domain into segments, and assigns a value to each segment which indicates how likely the extremal value of the function lies within that segment. The value combines local considerations (i.e., how likely the extremal value lies in this segment), along with global considerations (how well has this segment been investigated for the extremal value). Stopping criteria are used to determine when the algorithm should terminate. This can be after a fixed number of iterations or it can be after the algorithm has attained some acceptable error threshold.

As in the case of the well-known deterministic quicksort sorting algorithm, the worst case performance is $O(n^2)$. This occurs if the worst pivot is selected during each iteration. In the average case, whereby each pivot value is randomly selected, the expected performance of deterministic quicksort is $O(n \log(n))$. For adaptive stochastic algorithms, the average case is often of interest and requires analysis using random variables and their distributions.

Bayesian algorithms are of the type described above. Bayesian algorithms compare favorably with other Global Optimization algorithms such as those described above [8]. However, algorithms of this class do have shortcomings. For example, criteria used to select points for evaluation or the memory required to store past information can cause the complexity of the algorithm in terms of execution time or storage to become intractable.

The algorithms presented in this paper rely on the Bayesian approach and are adaptive in that the results of previous computations are retained and used in subsequent decisions about where to select the next evaluation point and use information about the objective function, such as derivatives, to overcome Bayesian limitations. Bounds on convergence rates are established and algorithms are developed to approach or achieve those bounds.

1.7 Test Objective Functions

In order to evaluate the implemented algorithms it is necessary to use test functions. There are libraries of functions in use today. Two such functions that one would expect to encounter are the Rastrigin and Csendes functions. These functions are described in the following sections.

1.7.1 Rastrigin Function

The formula for the d-dimensional Rastrigin function is:

$$F(\vec{x}) = A \cdot d + \sum_{i=1}^{d} x_i^2 - A \cdot \cos(\omega \cdot x_i) \tag{1.1}$$
$$A = 10 \ ; \ \omega = 2 \cdot \pi \ ; \ x_i \in [-5.12, 5.12]$$

A graph of the two-dimensional Rastrigin function is shown in figure 1.2. It has a minimum value of 0 at $x = (0, 0)$. Because of the symmetry and minimum at (0,0), it is sometimes necessary to offset the function in order to avoid inadvertently testing at the minimum or to overcome singularities in algorithms that employ matrix computations (e.g., inversion of matrices).

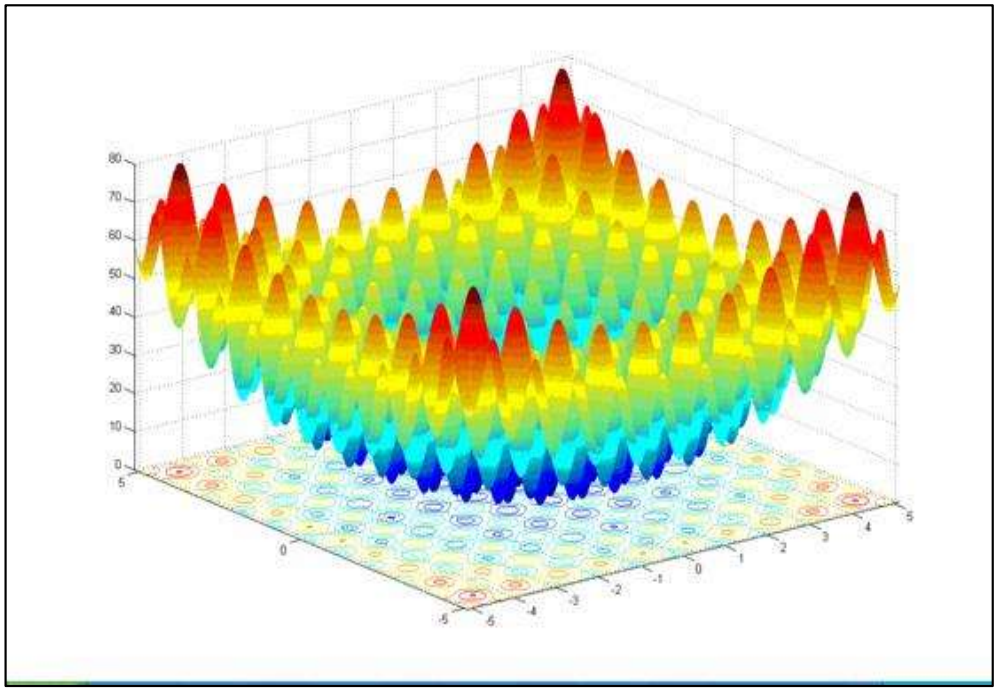

Figure 1.2 Two-dimensional Rastrigin function.

1.7.2 Csendes Function

The formula for the d-dimensional Csendes function is:

$$F(\vec{x}) = \sum_{i=1}^{d} x_i^6 (2 + \sin \frac{1}{x_i}) \tag{1.2}$$
$$x_i \in [-1, 1] \backslash 0$$

A graph of the two-dimensional Csendes function is shown on Figure 1.3. The Csendes function has a countably infinite number of local minima [21]. An accurate algorithm should find a minimum as near to (0,0) as possible. No minimum for this function exists but approaches zero as the x_i approach zero. Algorithms described in this paper are particularly effective minimizing the Csendes function.

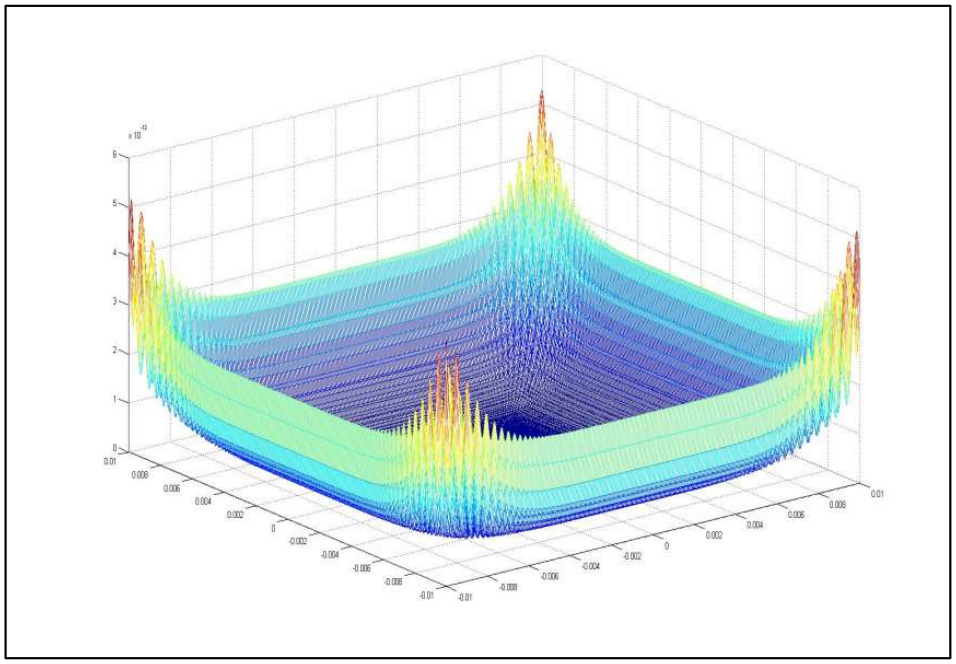

Figure 1.3 Two-dimensional Csendes function.

1.8 Organization of This Thesis

A Global Optimization Algorithm based on the partitioning the domain into rectangles is described in Chapter 2. A Global Optimization Algorithm based on decomposing the domain into triangles is presented in Chapter 3. The P-Algorithm, an algorithm used to approximate the minimum of a multimodal algorithm, is presented in Chapter 4. Partitioning the domain D recursively using a rectangular grid is discussed in Chapter 5, and it concludes with a description and analysis of the Hypercube algorithm. Algorithms based on partitioning the domain D using the Voronoi diagram are presented in Chapter 6. Algorithms based on partitioning the domain D using triangulation are described in Chapter 7.

The Java implementation and Unified Modeling Language (UML) of the algorithms presented in this paper is described in Chapter 8. The framework for the development of Global Optimization algorithms using MATLAB and matlabcontrol to access the MATLAB functions is described in Chapter 9.

CHAPTER 2

ON A GLOBAL OPTIMIZATION ALGORITHM FOR MULTIVARIATE SMOOTH FUNCTIONS

2.1 Problem Statement

The problem of approximating the global minimum f^* of a twice-continuously differentiable function f defined on the d-dimensional unit cube $[0,1]^d$ is considered. The algorithm presented here is similar to a one-dimensional algorithm presented in [4] and a two-dimensional algorithm based on Delaunay triangulations presented in [7]. Those papers presented algorithms with the property that for large enough n (depending on f), the error after n function evaluations is at most

$$c_1 \exp(-c_2\sqrt{n})$$

for c_1, c_2 depending on dimension (1 or 2). While the algorithm presented in [7] was based on a Delaunay triangulation of the domain, the algorithm considered in this paper is based on a rectangular decomposition. Rectangular decompositions have been employed in previous studies of global optimization, for example in [15, 17].

The algorithm in [7] works for arbitrary domains, but the error bound, which required a certain quality triangulation, could only be proved for bivariate functions. Although from a practical point of view, we expect that the algorithm presented in this paper would perform worse than the algorithm of [7], an asymptotic error bound for arbitrary dimension is proven. The main result is that eventually the error is at most

$$c_1(f,d) \exp\left(-c_2(f,d)\sqrt{n}\right),$$

where $c_2(f,d)$ decreases exponentially in dimension d.

In Section 2.5 it is explained why nonadaptive algorithms have error bounds of order $n^{-2/d}$ after a large number n of function evaluations.

2.2 The Algorithm

The algorithm operates by decomposing $[0,1]^d$ into (hyper)-rectangles as follows. Given a current decomposition, choose one of the rectangles (according to the maximal value of a criterion to be defined below) and bisect it along the longest axis by evaluating the function at up to 2^{d-1} midpoints of the longest rectangle edges.

Suppose that an algorithm has evaluated f at n points. Let $s(n)$ denote the number of rectangles in our partition. Each rectangle subdivision (iteration) requires up to 2^{d-1} function evaluations. Therefore, $s(n) \geq n/2^{d-1}$.

Let v_n denote the smallest volume of a rectangle after n iterations. Define

$$q \equiv \frac{3 \cdot 2^{2/3} e^{-1}}{2 \log(2)} \approx 1.27$$

and

$$g(x) = q \cdot d \left(x \log(1/x) \right)^{2/d}$$

for $0 < x \leq 1/2$ and $g(1) = q \cdot d$. Note that g is increasing and $g(x) \downarrow 0$ as $x \downarrow 0$. Let $M_n = \min_{1 \leq i \leq n} f(x_i)$ and denote the error by $\Delta_n = M_n - f(x^*)$.

Let L_n denote the multilinear function that has the same values as f at the vertices of the smallest enclosing rectangle. Note that M_n is equal to the global minimum of L_n. For $1 \leq i \leq n$ set

$$\underline{\rho}_i^n \equiv \frac{|R_i|}{\left(\max_{t \in R_i} L_n(t) - M_n + g(v_n) \right)^{d/2}}, \tag{2.1}$$

$$\overline{\rho}_i^n \equiv \frac{|R_i|}{\left(\min_{t \in R_i} L_n(t) - M_n + g(v_n) \right)^{d/2}}, \tag{2.2}$$

and

$$\rho_i^n \equiv \frac{|R_i|}{\left(L_n(c_i) - M_n + g(v_n)\right)^{d/2}}, \tag{2.3}$$

where c_i is the center of rectangle i. That is, if $R_i = \prod_{k=1}^{d}[a_k, b_k]$, then

$$c_i = \left(\frac{a_1 + b_1}{2}, \frac{a_2 + b_2}{2}, \cdots, \frac{a_d + b_d}{2}\right).$$

Note that the max and min in the denominators of (2.1) and (2.2) are equal to the maximum and minimum of f at the vertices of the rectangle, respectively.

Note that if the smallest rectangle is about to be subdivided, then

$$\rho_i^n \leq \overline{\rho}_i^n \leq \frac{|R_i|}{g(v_n)^{d/2}} \leq \frac{v_n}{(q \cdot d)^{d/2} v_n \log(1/v_n)} = \frac{1}{(q \cdot d)^{d/2} \log(n)},$$

since $v_n \leq 1/n$.

A more formal description of the algorithm follows. In the description, n is the total number of function evaluations to make, i is the current number of rectangles, and j is the iteration number (number of function evaluations that have been made). More formally, the algorithm making at least n function evaluations is:

1. Start with the rectangle $[0, 1]^d$ and with the function values at all 2^d vertices. Let i denote the number of rectangles and j the number of function evaluations. (Initially $i = 1$ and $j = 2^d$.)

2. For each rectangle R_k in the current collection, compute ρ_k^n, keeping track of the rectangle β that has the largest value of ρ_k^n (breaking ties arbitrarily).

3. For the best rectangle R_β, evaluate $f(x_k)$, $k = 1, 2, \ldots, 2^d$, where x_k is the midpoint of the kth edge along the longest dimension (breaking ties arbitrarily). Update the number of rectangles $i \leftarrow i + 1$ and increment j by the number of new function evaluations.

4. Compute v_i, the smallest rectangle volume. If $v_i < v_{i-1}$, then update $g(v_i)$.

5. If $j < n$, return to step 2.

Let $\hat{F} \subset C^2\left([0,1]^d\right)$ denote the set of functions that have a unique global minimizer $x^* \in]0,1[^d$. Denote the matrix of second-order partial derivatives at the minimizer by $D^2 f(x^*)$, assumed positive definite.

Our main result on the convergence rate of the error for this algorithm follows.

Theorem 1 *Suppose that $f \in \hat{F}$. There is a number $n_0(f)$ such that for $n \geq n_0(f)$,*

$$\Delta_n \leq \frac{1}{8} \left\| D^2 f \right\|_{\infty,[0,1]^d} (q \cdot d) \exp\left(-\sqrt{n}\beta(f,d)\right),$$

where

$$\beta(f,d) = \left(\frac{2\Gamma(1+d/2)\sqrt{\det(D^2 f(x^*))}}{(2\pi)^{d/2}(d(d+1))2^{d-1}(2(q\cdot d))^{d/2}} \right)^{1/2}.$$

Note that $\beta(f,d)$ approaches 0 rapidly as d increases.

2.3 Interpolation Error Bounds

The results about interpolation from [20] are used. For a compact set K and $f \in C^2(K)$, define the seminorm

$$\left\| D^2 f \right\|_{\infty,K} \equiv \sup_{x \in K} \sup_{\substack{u_1,u_2 \in R^d \\ \|u_i\|=1}} \left| D_{u_1} D_{u_2} f(x) \right|,$$

where $D_y f$ is the derivative of f in the direction y. This is a measure of the maximum size of the second derivative of f over K.

Lemma 1 *Consider a rectangle $R = \{a_i \leq x \leq b_i\}_{i=1}^d$. The error bound for multilinear interpolation is*

$$\max_{x \in R} |f(x) - L(x)| \leq \frac{1}{8} \sum_{i=1}^{d} (b_i - a_i)^2 \left\| D^2 f \right\|_{\infty,R} = \frac{1}{8} \frac{\sum_{i=1}^d (b_i - a_i)^2}{|R|^{2/d}} \left\| D^2 f \right\|_{\infty,R} |R|^{2/d}.$$

Proof 1 *See [20].*

The quantity

$$\frac{\sum_{i=1}^{d}(b_i-a_i)^2}{|R|^{2/d}}=\frac{\sum_{i=1}^{d}(b_i-a_i)^2}{(\prod_{i=1}^{d}(b_i-a_i))^{2/d}}$$

is a measure of how "bad" the rectangle is; that is, how far it is from cubic. Since by construction the widths are within a factor of 2 of each other, there is some number $1 \leq k \leq d$ and a positive number w such that k of the intervals have width w and $d-k$ have width $2w$, and

$$\frac{\sum_{i=1}^{d}(b_i-a_i)^2}{(\prod_{i=1}^{d}(b_i-a_i))^{2/d}}=\frac{kw^2+(d-k)(2w)^2}{(2^{d-k}w^d)^{2/d}}=\frac{(4d-3k)}{2^{2(d-k)/d}}=\left(d-\frac{3}{4}k\right)2^{2k/d}.$$

For $k=d$ the value d is obtained and for $k=1$ the value $(d-3/4)2^{2/d}$ is obtained. Maximizing

$$d(1-3x/4)2^{2x}$$

over $x \in]0,1[$ gives a maximum value of

$$\frac{3\cdot 2^{2/3}e^{-1}}{2\log(2)}d \equiv q\cdot d \approx 1.27d.$$

So the quality metric ranges from d up to about $1.27d$, and the bound in Lemma 1 is replaced with

$$\max_{x\in R}|f(x)-L(x)| \leq \frac{1}{8}q\cdot d\left\|D^2 f\right\|_{\infty,R}|R|^{2/d}. \tag{2.4}$$

2.4 Proof of Theorem 1

For $\epsilon > 0$, define

$$\mathcal{I}_f(\epsilon) \equiv \int_{[0,1]^d}\frac{dx}{(f(x)-f^*+\epsilon)^{d/2}}.$$

The following lemma generalizes Lemma 3.2 of [7] to arbitrary dimension.

Lemma 2

$$\lim_{\epsilon \downarrow 0} \frac{\mathcal{I}_f(\epsilon)}{\log(1/\epsilon)} = \frac{d(2\pi)^{d/2}}{2\Gamma(1+d/2)} \cdot \left(\det(D^2 f(x^*))\right)^{-1/2} \equiv \alpha(f,d).$$

Proof 2 *Denote the matrix of second-order partial derivatives by $D^2 f(x^*)$, assumed positive definite (and symmetric). Denote the eigenvalues of $D^2(x^*)$ by*

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d > 0.$$

Then by Taylor's theorem (x a column vector),

$$f(x^* + x) - f(x^*) = \frac{1}{2} x^T D^2 f(x^*) x + o(\|x\|^2).$$

Since $D^2 f(x^*)$ is symmetric positive definite,

$$D^2 f(x^*) = V \Lambda V^T$$

for an orthogonal matrix V and diagonal matrix $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_d)$. Then

$$D^2 f(x^*) = V \Lambda^{1/2} \Lambda^{1/2} V^T$$

and

$$f(x^* + x) - f^* = \frac{1}{2} x^T V \Lambda^{1/2} \Lambda^{1/2} V^T x + o(\|x\|^2) = \frac{1}{2} \|Tx\|^2 + o(\|x\|^2),$$

where $Tx \equiv \Lambda^{1/2} V^T x$. Using the fact that x^* is the unique minimizer, there exists a number $c \in]0,1[$ such that for any $\eta > 0$,

$$\int_{[0,1]^d} \frac{dx}{\left(f(x) - f(x^*) + \epsilon\right)^{d/2}} \leq \int_{B_c^d(0)} \frac{dx}{\left((1/2 - \eta)\|Tx\|^2 + \epsilon\right)^{d/2}} + O(1) \qquad (2.5)$$

as $\epsilon \downarrow 0$, *and*

$$\int_{[0,1]^d} \frac{dx}{\left(f(x) - f(x^*) + \epsilon\right)^{d/2}} \geq \int_{B_c^d(0)} \frac{dx}{\left(\left(\frac{1}{2} + \eta\right) \|Tx\|^2 + \epsilon\right)^{d/2}} + O(1), \tag{2.6}$$

where $B_c^d(0)$ *denotes the ball of radius* c, *centered at* 0, *in* R^d. *Using the orthogonality of* V,

$$\begin{aligned}
\int_{B_c^d(0)} \frac{dx}{\left(b \|Tx\|^2 + \epsilon\right)^{d/2}} &= \int_{V(B_c^d(0))} \frac{dx}{\left(b \|\Lambda^{1/2} x\|^2 + \epsilon\right)^{d/2}} \\
&= \int_{B_c^d(0)} \frac{dx}{\left(b \|\Lambda^{1/2} x\|^2 + \epsilon\right)^{d/2}} \quad y_i \leftarrow x_i \left(\lambda_i b/\epsilon\right)^{1/2} \\
&= \int_{E(c,b,\epsilon)} \frac{dy}{\left(\|y\|^2 + 1\right)^{d/2}} \frac{1}{b^{d/2} \left(\prod_{i=1}^d \lambda_i\right)^{1/2}},
\end{aligned}$$

where $E(c, b, \epsilon)$ *is the image of the ball of radius* c *under the map* $x_i \mapsto x_i(\lambda_i b/\epsilon)^{1/2}$. *Therefore,*

$$B^d_{c\sqrt{b\lambda_d/\epsilon}}(0) \subset E(c, b, \epsilon) \subset B^d_{c\sqrt{b\lambda_1/\epsilon}}(0).$$

For arbitrary $c > 0$,

$$\int_{B_c^d(0)} \frac{dx}{\left(\|x\|^2 + 1\right)^{d/2}} = dC_d E_d(c),$$

where

$$C_d = \frac{\pi^{d/2}}{\Gamma(1 + d/2)}$$

and

$$E_d(A) \equiv \int_{r=0}^{A} \frac{dr}{(r^2+1)^{d/2}} = \begin{cases} \frac{1}{2}\log(1+A^2) - \sum_{k=1}^{(d-2)/2} \frac{A^{2k}}{2k(A^2+1)^k}, & d \text{ even}, \\ \log\left(A + \sqrt{1+A^2}\right) - \sum_{k=1}^{(d-1)/2} \frac{A^{2k-1}}{(2k-1)(A^2+1)^{(2k-1)/2}}, & d \text{ odd}. \end{cases}$$

For all $d \geq 1$,

$$E_d(A)/\log(A) \to 1 \tag{2.7}$$

as $A \uparrow \infty$.

The following bounds exist,

$$dC_d E_d \left(c\sqrt{b\lambda_d/\epsilon} \right) \leq \int_{B_c^d(0)} \frac{dx}{\left(b \left\| Tx \right\|^2 + \epsilon \right)^{d/2}} \leq dC_d E_d \left(c\sqrt{b\lambda_1/\epsilon} \right),$$

and

$$\begin{aligned} O(1) + (1/2+\eta))^{-d/2} \left(\prod_{i=1}^{d} \lambda_i \right)^{-1/2} dC_d E_d \left(c\sqrt{(1/2+\eta)\lambda_d/\epsilon} \right) \\ \leq \mathcal{I}_f(\epsilon) \leq O(1) + (1/2-\eta)^{-d/2} \left(\prod_{i=1}^{d} \lambda_i \right)^{-1/2} dC_d E_d \left(c\sqrt{(1/2-\eta)\lambda_1/\epsilon} \right). \end{aligned}$$

Since $\eta > 0$ *is arbitrary, using* (2.7) *completes the proof.*

For the rest of this section a fixed function $f \in \hat{F}$ is considered. Suppose that $n \geq n_1(f)$, where

$$n_1(f) = \inf \left\{ n : n \geq \exp \left(\left\| D^2 f \right\|_{\infty,[0,1]^d}^{d/2} \right) \right\}. \tag{2.8}$$

(Later n is restricted further.)

The following lemma gives an approximation needed to prove the error bound.

Lemma 3 *As* $n \to \infty$,

$$\frac{\sum_{i=1}^{s(n)} \rho_i^n}{\log(1/g(v_n))} \to 1.$$

Proof 3 *Observe that*

$$\begin{aligned} &(f(c_i) - f^* + g(v_n))^{d/2} = \\ &(L_n(c_i) - M_n + g(v_n))^{d/2} \left(1 + \frac{f(c_i) - L_n(c_i)}{|R_i|^{2/d}} \cdot \frac{|R_i|^{2/d}}{L_n(c_i) - M_n + g(v_n)} + \frac{M_n - f^*}{L_n(c_i) - M_n + g(v_n)} \right)^{d/2}. \end{aligned} \tag{2.9}$$

Using (2.4) *the term following term is bounded*

$$\begin{aligned}\frac{|f(c_i)-L_n(c_i)|}{|R_i|^{2/d}}\cdot\frac{|R_i|^{2/d}}{L_n(c_i)-M_n+g(v_n)} &\le \frac{1}{8}qd\left\|D^2f\right\|_{\infty,R_i}(\overline{\rho}_i^n)^{2/d}\\ &\le \frac{1}{8}qd\left\|D^2f\right\|_{\infty,R_i}\left(\frac{1}{(q\cdot d)^{d/2}\log(n)}\right)^{2/d}\\ &= \frac{1}{8}\left\|D^2f\right\|_{\infty,R_i}\left(\frac{1}{\log(n)}\right)^{2/d}\\ &\le 1/8,\end{aligned}$$

since, by definition of n_1 *at* (2.8), $n \ge n_1(f)$ *implies that*

$$\log(n) \ge \left\|D^2f\right\|_{\infty,[0,1]^d} \ge \left\|D^2f\right\|_{\infty,R_i}.$$

Let v_n^* *denote the volume of the rectangle containing* x^*. *Since* $n \ge n_1$, $v_n^* \le 2v_n$. *Then*

$$\begin{aligned}M_n - f^* &\le L_n(x^*) - f(x^*)\\ &\le \frac{1}{8}q\cdot d\left\|D^2f\right\|_{\infty,R_i}(v_n^*)^{2/d}\\ &\le \frac{2^{2/d}}{8}q\cdot d\left\|D^2f\right\|_{\infty,R_i}(v_n)^{2/d}\\ &\le \frac{1}{2}\left\|D^2f\right\|_{\infty,R_i}(q\cdot d)v_n^{2/d}\\ &= \frac{1}{2}\left\|D^2f\right\|_{\infty,R_i}\frac{g(v_n)}{\log(1/v_n)^{2/d}}\\ &\le \frac{1}{2}\left\|D^2f\right\|_{\infty,R_i}\frac{g(v_n)}{\log(n)^{2/d}}\\ &\le g(v_n)/2.\end{aligned} \tag{2.10}$$

Substituting these bounds in (2.9) *gives*

$$\begin{aligned}(f(c_i)-f^*+g(v_n))^{d/2} &\le (L_n(c_i)-M_n+g(v_n))^{d/2}\left(1+\frac{1}{2}+\frac{1}{8}\right)^{d/2}\\ &< (L_n(c_i)-M_n+g(v_n))^{d/2}(2)^{d/2},\end{aligned}$$

and

$$(f(c_i) - f^* + g(v_n))^{d/2} \geq (L_n(c_i) - M_n + g(v_n))^{d/2} \left(1 - \frac{1}{2} + 0\right)^{d/2}$$
$$= (L_n(c_i) - M_n + g(v_n))^{d/2} (2)^{-d/2}.$$

Therefore, for $n \geq n_1(f)$,

$$\int_{[0,1]^d} (f(x) - f^* + g(v_n))^{-d/2} \, dx \leq 2^{d/2} \sum_{i=1}^{s(n)} \rho_i^n$$

and

$$\int_{[0,1]^d} (f(x) - f^* + g(v_n))^{-d/2} \, dx \geq 2^{-d/2} \sum_{i=1}^{s(n)} \rho_i^n.$$

This implies that

$$\frac{1}{2^{d/2} s(n)} \mathcal{I}_f(g(v_n)) \leq \frac{1}{s(n)} \sum_{i=1}^{s(n)} \rho_i^n \leq \frac{2^{d/2}}{s(n)} \mathcal{I}_f(g(v_n)). \tag{2.11}$$

The following bounds exist

$$\frac{1}{s(n)} \sum_{i=1}^{s(n)} \rho_i^n \leq \overline{\rho}^n \leq \frac{1}{(q \cdot d)^{d/2} \log(1/v_n)} \tag{2.12}$$

and

$$\frac{1}{s(n)} \sum_{i=1}^{s(n)} \rho_i^n \geq \frac{1}{3} \underline{\rho}^n \geq \frac{1}{3(q \cdot d)^{d/2} \log(1/v_n)}, \tag{2.13}$$

because the algorithm divides a rectangle into 2 *subrectangles at each step, causing the* ρ_i^n*'s to concentrate between* $\rho^n/2$ *and* ρ^n*. Therefore,*

$$\frac{1}{3 \cdot 2^{d/2} s(n)} \mathcal{I}_f(g(v_n)) \leq \frac{1}{3s(n)} \sum_{i=1}^{s(n)} \rho_i^n \leq \frac{1}{3(q \cdot d)^{d/2} \log(1/v_n)}$$
$$\leq \frac{1}{s(n)} \sum_{i=1}^{s(n)} \rho_i^n \leq \frac{2^{d/2}}{s(n)} \mathcal{I}_f(g(v_n)),$$

using the first inequality in (2.11), (2.12), (2.13), *and the second inequality in* (2.11), *respectively. There is a number* $n_2(f)$ *such that, by Lemma 2,* $n \geq n_2(f)$ *implies that*

$$\frac{1}{2}\log(1/g(v_n))\alpha(d) \leq \mathcal{I}_f(g(v_n)) \leq 2\log(1/g(v_n))\alpha(d).$$

Replace $\mathcal{I}_f(g(v_n))$ *by* $\log(1/g(v_n))$ *(times appropriate factor) to get*

$$\frac{\alpha(d)}{2(d+1)2^{d/2}s(n)}\log(1/g(v_n)) \leq \frac{1}{(d+1)(q\cdot d)^{d/2}\log(1/v_n)} \leq 2\frac{2^{d/2}\alpha(d)}{s(n)}\log(1/g(v_n)).$$

Now replace $\log(1/g(v_n))$ *by*

$$\log(1/(q\cdot d)) + \frac{2}{d}\left(\log(1/v_n) - \log\log(1/v_n)\right)$$

to obtain the bounds

$$\begin{aligned}&\frac{\alpha(d)}{2(d+1)2^{d/2}s(n)}\left(\log(1/(q\cdot d)) + \frac{2}{d}\left(\log(1/v_n) - \log\log(1/v_n)\right)\right)\\ &\leq \frac{1}{(d+1)(q\cdot d)^{d/2}\log(1/v_n)} \leq 2\alpha(d)\frac{2^{d/2}}{s(n)}\left(\log(1/(q\cdot d)) + \frac{2}{d}\left(\log(1/v_n) - \log\log(1/v_n)\right)\right).\end{aligned}$$

Set $x_n = \log(1/v_n)$. *Then*

$$\begin{aligned}\frac{1}{2(d+1)2^{d/2}s(n)}\left(\log(1/(q\cdot d)) + \frac{2}{d}[x_n - \log(x_n)]\right) &\leq \frac{1}{\alpha(d)(d+1)(q\cdot d)^{d/2}x_n}\\ &\leq 2\frac{2^{d/2}}{s(n)}\left(\log(1/(q\cdot d)) + \frac{2}{d}[x_n - \log(x_n)]\right).\end{aligned}$$

This implies that eventually, say for $n \geq n_3(f)$,

$$x_n \geq \left(\frac{ds(n)}{\alpha(d)2^{2+d/2}(d+1)(q\cdot d)^{d/2}}\right)^{1/2},$$

which implies that

$$v_n \leq \exp\left(-\left(\frac{ds(n)}{\alpha(d)2^{2+d/2}(d+1)(q\cdot d)^{d/2}}\right)^{1/2}\right).$$

Using our previous estimate at 2.10,

$$M_n - f^* \leq \frac{1}{8}\left\|D^2 f\right\|_{\infty,[0,1]^d}(q\cdot d)v_n^{2/d}$$
$$\leq \frac{1}{8}\left\|D^2 f\right\|_{\infty,R_i}(q\cdot d)\exp\left(-\left(\frac{ds(n)}{\alpha(d)2^{2+d/2}(d+1)(q\cdot d))^{d/2}}\right)^{1/2}\frac{2}{d}\right).$$

Now $s(n)\geq n/2^{d-1}$*, and so for* $n\geq n_0(f)=\max\{n_1(f),n_2(f),n_3(f)\}$*,*

$$\Delta_n \leq \frac{1}{8}\left\|D^2 f\right\|_{\infty,[0,1]^d}(q\cdot d)\exp\left(-\left(\frac{dn}{\alpha(d)2^{2+d/2}(d+1)2^{d-1}(q\cdot d)^{d/2}}\right)^{1/2}\frac{2}{d}\right)$$
$$= \frac{1}{8}\left\|D^2 f\right\|_{\infty,R_i}(q\cdot d)\exp\left(-\sqrt{n}\beta(f,d)\right)$$

where

$$\beta(f,d)=\left(\frac{2\Gamma(1+d/2)\sqrt{\det(D^2 f(x^*))}}{(2\pi)^{d/2}(d(d+1))2^{d-1}(2(q\cdot d))^{d/2}}\right)^{1/2}.$$

This completes the proof of Theorem 1.

2.5 Nonadaptive Algorithms

In this section we continue to consider the problem of approximating the global minimum f^* of a twice-continuously differentiable function f defined on the unit cube $[0,1]^d$ using only function values. Let us consider the convergence rate that can be obtained with a nonadaptive method; that is, a method that chooses points to evaluate the function independent of the function values.

Denote the eigenvalues of $D^2(x^*)$ by

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d > 0.$$

Consider a sequence of points $\{x_1, x_2, \ldots\}$ that is dense in $[0,1]^d$. This is a fixed sequence independent of f. Define the dispersion of the sequence by

$$d_n = \sup_{x\in[0,1]^d}\min_{1\leq i\leq n}\|x-x_i\|, \quad n\geq 1.$$

There exists a sequence with

$$d_n \leq \sqrt{d}\frac{2}{\log 4}n^{-1/d}$$

for all $n \geq n_1$; see [19], Theorem 6.9. For $f \in \hat{F}$, by Taylor's theorem there is a number $n_2(f)$ such that for $n > n_2$, the error is at most $\lambda_1 d_n^2$. So for $n > \max\{n_1, n_2\}$,

$$\Delta_n \leq \lambda_1 d \left(\frac{2}{\log 4}\right)^2 n^{-2/d}.$$

On the other hand, $d_n = \Omega\left(n^{-1/d}\right)$ ([19], p. 150). In particular, for any sequence,

$$d_n \geq \left(\frac{\Gamma(1+d/2)}{\pi^{d/2}}\right)^{1/d} n^{-1/d}.$$

Therefore, for any nonadaptive algorithm, there exists a function f for which the following lower bound on the error holds:

$$\Delta_n \geq \lambda_d \left(\frac{\Gamma(1+d/2)}{\pi^{d/2}}\right)^{2/d} n^{-2/d}.$$

This means that for any nonadaptive method, for small enough $\epsilon > 0$, about $\sqrt{1/\epsilon^d}$ function values are needed to obtain an error of at most ϵ. We therefore see that for nonadaptive methods, global optimization suffers from a "curse of dimension".

CHAPTER 3

A TWO VARIATE GLOBAL OPTIMIZATION ALGORITHM

3.1 Problem Statement

In this chapter an algorithm for approximating the global minimum of a continuous two variable function f defined on the unit square, $[0,1]^2$ is proposed. The algorithm is a variation on an algorithm described in [7], that decomposes the domain using Delaunay triangulation. An asymptotic upper bound on its approximation error for the case when the objective function is twice continuously differentiable with a unique point of global minimum in the interior of the unit square remains to be established.

Let f be a twice continuously differentiable function defined on $[0,1]^2$. The problem is to approximate the global minimum $M \equiv \min_{0\leq t\leq 1} f(t)$ of f using a number of adaptively chosen function evaluations. While the function is assumed differentiable, the algorithm uses only function values.

At each step of the algorithm the domain will be partitioned into triangles. Initially the unit square is partitioned into four congruent triangles. On each iteration a triangle is selected and is replaced by two triangles created by connecting the midpoint of the longest side to the opposite vertex. All triangles will be similar right triangles, two angles equal to 45 degrees. The area of each triangle will, of course, be equal to $\frac{1}{2}$ the size of the original triangle.

After n iterations there will be n+4 triangles and therefore 2n+4 function evaluations. With each iteration the objective function will be evaluated at the center of each triangle and will therefore be used as input to the algorithm at the next iteration.

The next section describes the details of the proposed algorithm and discusses the expected convergence properties.

3.2 The Algorithm

Let $T_1, T_2, ..., T_{2n+4}$ be triangles partitioning the unit square. For each T_i define ρ_i^n to be a measure, based on a statistical model (such as the one used by the P-Algorithm) of the likelihood that the minimum of the objective function will lie within triangle T_i along with precedence given to larger (unsearched) areas. This is referred to as exploration vs. exploitation. The algorithm should be written so that the implementation of the function ρ_i^n can be easily changed.

The algorithm works as follows. Suppose f has been evaluated at $2n + 4$ points (the domain will have been partitioned into $s(n) = n + 4$ triangles). Define c_i to be the barycenter of the triangle being evaluated, and M_n to be the smallest value observed at any point in the execution of the algorithm. Divide the triangle with the largest ρ_i^n value by connecting the bisection point of the longest side to the opposite vertex.

Evaluate $f(c_i)$ for the two new triangles. If a new M_n value has been discovered or if a new smallest triangle has been formed, recompute ρ_i^n values for all triangles. A separate thread or task could be started to do this as the algorithm continues. As the algorithm progresses and n gets large, M_n would not change much. The triggering of the recomputation of all ρ_i^n values could be made dependent on the value $f(c_i) - M_n$ when $f(c_i)$ is found to be a new minimum value. A similar consideration may be made for finding a smallest triangle.

Algorithm:

Set M = $+\infty$, n = 1;

Step 1: Compute the vertices for each of 4 congruent triangles for the unit square

Step 2: Compute ρ for each triangle

Step 3: Set M = min(M,$f(c_i)$)

Step 4: $n \leftarrow n + 1$

Step 5: Place the triangle on a priority queue keyed by ρ

Step 6: Remove the triangle with maximum ρ from the priority queue

Step 7: Divide triangle into $(\frac{1}{2})$ sized triangles

Step 8: Compute ρ and place each triangle on the priority queue, recomputing all ρ values if required

Step 9: Set M = min(M,$f(c_i)$)

Step 10: if stopping criteria not met go to step 4

Step 11: return M as minimum

If ρ_i^n is defined as:

$$\rho_i^n \triangleq \frac{|T_i|}{(f(c_i) - M_n + v_n(\log(\frac{1}{v_n})))}$$

where $|T_i|$ is the area of the current triangle and $f(c_i)$ denotes the value of the objective function evaluated at the center point of the triangle being evaluated, a convergence rate similar to [7] is anticipated.

Note as the algorithm progresses, every time a new smallest triangle with area v_n or a new minimum value, M_n is found this, to some extent, invalidates the previously

computed ρ values on the priority queue. As the algorithm converges, even if the ρ values are not recomputed, a strategy needs to be developed that will either eliminate the need to recompute these values (by incorporating some additional function of n, M_n or v_n into the ρ computation) or by recomputing all ρ values periodically (i.e., not every time they change). Most likely the best approach shall be to implement a ρ-calculation function as a separate background thread or task that would recalculate ρ whenever a new M_n or smallest triangle is encountered by the algorithm.

3.3 Convergence Analysis

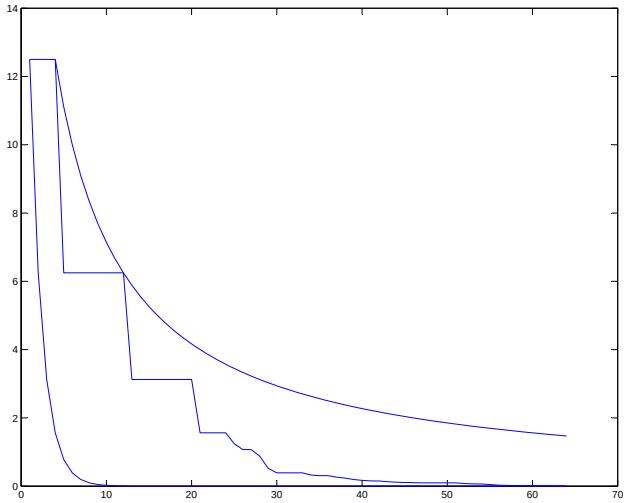

Figure 3.1 Bounds for the sizes of the smallest triangle as a function of n with experimental results

Convergence analysis will proceed along the lines of [7]. v_n must converge faster than the upper bound, $v_n = \frac{1}{n+4}$ as this bound implies that all triangles of a given size are split before continuing. This implies a search which is no better than a grid search. Furthermore, v_n must converge slower than the lower bound $v_n = \frac{1}{2^n}$, as this

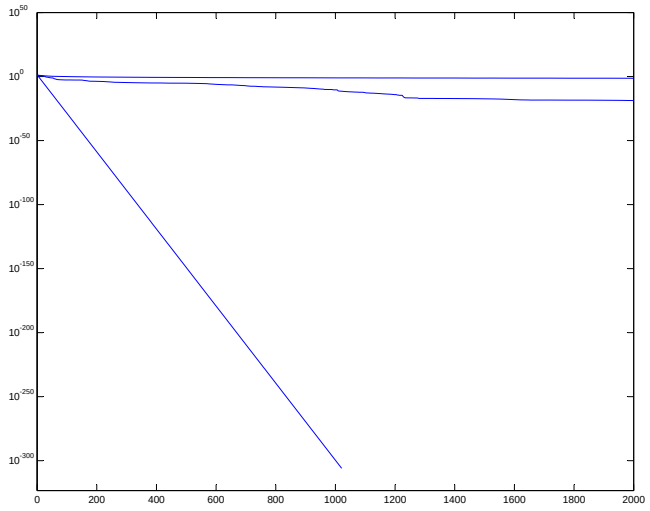

Figure 3.2 Bounds for the sizes of the smallest triangle as a function of n with experimental results (log scale)

implies the search begins with one of the original triangles and splits that triangle into smaller triangles on each iteration thereby not covering some of the domain as $n \to \infty$. These limits are shown graphically in Figure 3.2. Minimum triangle size for the 10x10 square surrounding the origin as the algorithm searches the Rastrigin function is also shown in the figure. Better bounds than those shown in Figure 3.2 need to be derived for v_n. It is conjectured that $v_n \sim \exp(-c\sqrt{n})$.

A similar algorithm using Delaunay triangulation is described in [7]. This algorithm should converge similarly to [7], but subdividing the triangles involves much less computation than performing Delauney triangulation.

At any step n, the number of triangles is given by:

$$s(n) = n + 4.$$

Since all triangles have angle sizes $90°, 45°$, and $45°$, using the quality metric $Q(T)$ defined in [7], newline

$$Q(T) = \frac{(R(T)^2)}{|T|}$$

where $R(T)$ is the radius of the circumscribing circle for the triangle. Note the triangle meets the special case of the Delaunay criteria and that all triangles of the same size meet that same criteria. In other words, if the domain were partitioned into same size triangles, the Delaunay criteria will have been met and each triangle will share that circumscribing circle with one other triangle, forming a square that fits into the circle.

Note that:

$$Q(T) = \frac{(\sqrt{2}a)^2}{\frac{1}{2}a^2} = \frac{2a^2}{\frac{1}{2}a^2} = 4$$

for all triangles formed by this algorithm's partitioning scheme.

Define

$$g(x) = 4x \ \log(1/x)$$

for $0 < x \leq \frac{1}{2}$ and $g(x) = 2$ for $x \geq \frac{1}{2}$. Note g is increasing and $g(x) \downarrow 0$ as $x \downarrow 0$. Let $M_n = \min_{1 \leq i \leq n} f(x_i)$ and denote the error by $\Delta_n = M_n - f(x^*)$ for $n \geq 1$ and $1 \leq i \leq n$ where x^* is the point at which the global minimum occurs.

ρ is a function based on the statistical model:

$$\int_{T_i} \frac{ds}{L_n(s) - M_n + g(v_n)} \qquad 1 \leq i \leq s(n)$$

where L_n is the mapping of the triangle T_i onto the tangent plane of the objective function (i.e., the function being minimized) at the barycenter of the triangle, T_i.

For the algorithm analysis we adapt the above function:

$$\rho_i^n \triangleq \frac{\sqrt{|T_i|}}{f(c_i) - M_n + g(v_n)} \qquad 1 \leq i \leq s(n), \quad g(v_n) = v_n \log(\frac{1}{v_n})$$

The center point of the triangle is chosen instead of averaging the function values evaluated at the vertices of T because, as a general philosophy of global optimization algorithms, evaluation of the objective function is considered to be expensive and should be done only as many times as necessary. For this algorithm, the values evaluated at the vertices are not used to determine the partitioning of the domain. By exploiting information about the second derivative of the objective function, limits for the minimum and maximum values should be obtainable for all x in any triangle, T_i shall be derived.

3.4 Extending the Algorithm to More Than Two Dimensions

A very interesting aspect of this algorithm is the relative ease with which it can be extended into multiple dimensions, along with the potential for not dramatically affecting the convergence rate. The algorithm remains as stated, except a new ρ function is used to have volumes transformed into ρ. The algorithm is front loaded in that the partitioning and initial ρ values are done at initialization time. When hyperpyramids are split by bisecting the longest side and connecting the bisection point to the other vertices, the volume is halved and eventually the resulting two hyperpyramids are similar (and therefore have the same $Q(T)$ value). The most expensive operation will be to find the longest side. This time could be shortened by paying heed as to how the vertices are stored in the record representing the hyperpyramid.

Extending the partitioning scheme proceeds as follows:

Begin with the coordinates of the two-dimensional partitioning:

Table 3.1 Coordinates for Two-Dimensional Partitioning

Triangle	Vertices					
1	0	0	1	0	.5	.5
2	0	1	1	1	.5	.5
3	1	0	0	0	.5	.5
4	1	1	0	1	.5	.5

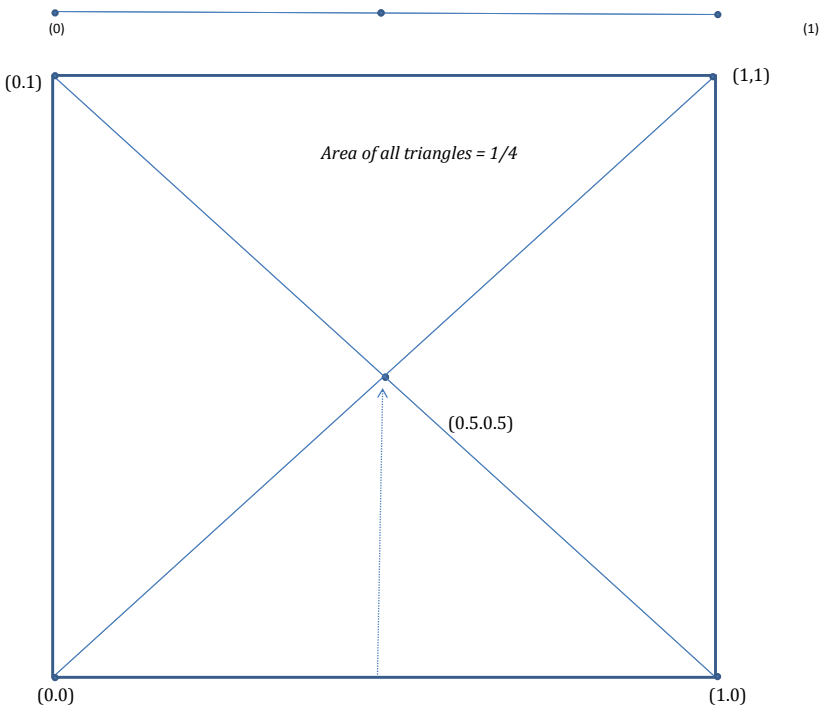

Figure 3.3 Two-dimensional partitioning scheme with first split

There are four triangles in the two-Dimensional case. The three-Dimensional case has six faces of four. In Figure 3.4 the first split yielding similar pyramids is shown. After this split pyramids are similar. Note in the two-Dimensional case triangles are always similar. Experimentation has shown that if initial pyramids' rho value is set to the area (in this case, $\frac{1}{24}$) all of the initial pyramids will be split once. This ensures that no pyramids are unsearched due to ρ increasing too quickly.

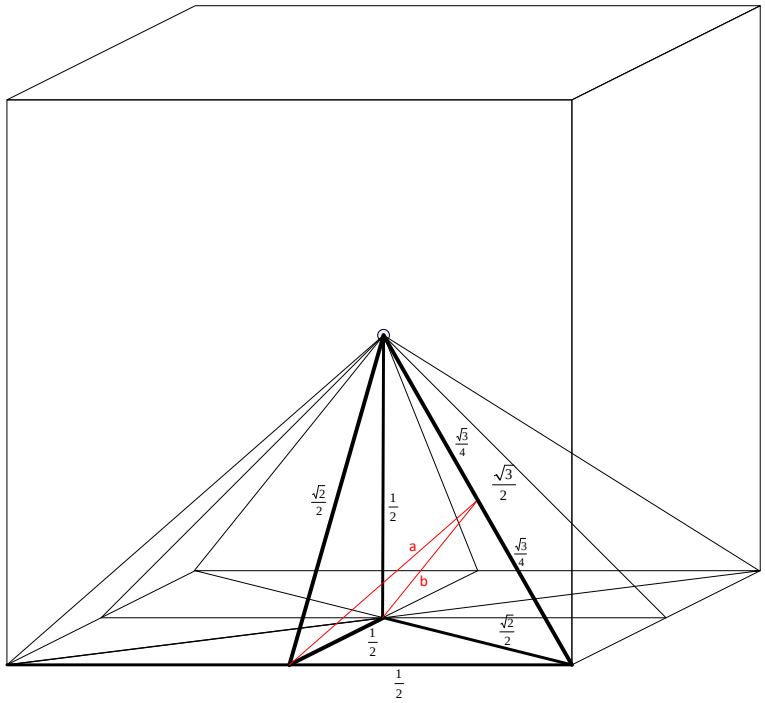

Figure 3.4 Three-dimensional partitioning scheme depicting five splits – fifth split shown in red

To construct the coordinates for the three-Dimensional case six copies of Table 3.1 are required, augmented as shown:

Table 3.2 Coordinates for Three-Dimensional Partitioning – Copy 1

Pyramid	Vertices											
1	**0**	**0**	0	**1**	**0**	0	**.5**	**.5**	0	.5	.5	.5
2	**0**	**1**	0	**1**	**1**	0	**.5**	**.5**	0	.5	.5	.5
3	**1**	**0**	0	**0**	**0**	0	**.5**	**.5**	0	.5	.5	.5
4	**1**	**1**	0	**0**	**1**	0	**.5**	**.5**	0	.5	.5	.5

Table 3.3 Coordinates for Three-Dimensional Partitioning – Copy 2

Pyramid	Vertices											
5	**0**	**0**	1	**1**	**0**	1	**.5**	**.5**	1	.5	.5	.5
6	**0**	**1**	1	**1**	**1**	1	**.5**	**.5**	1	.5	.5	.5
7	**1**	**0**	1	**0**	**0**	1	**.5**	**.5**	1	.5	.5	.5
8	**1**	**1**	1	**0**	**1**	1	**.5**	**.5**	1	.5	.5	.5

Table 3.4 Coordinates for Three-Dimensional Partitioning – Copy 3

Pyramid	Vertices											
9	**0**	0	**0**	**1**	0	**0**	**.5**	0	**.5**	.5	.5	.5
10	**0**	0	**1**	**1**	0	**1**	**.5**	0	**.5**	.5	.5	.5
11	**1**	0	**0**	**0**	0	**0**	**.5**	0	**.5**	.5	.5	.5
12	**1**	0	**1**	**0**	0	**1**	**.5**	0	**.5**	.5	.5	.5

Table 3.5 Coordinates for Three-Dimensional Partitioning – Copy 4

Pyramid	Vertices											
13	**0**	1	**0**	**1**	1	**0**	**.5**	1	**.5**	.5	.5	.5
14	**0**	1	**1**	**1**	1	**1**	**.5**	1	**.5**	.5	.5	.5
15	**1**	1	**0**	**0**	1	**0**	**.5**	1	**.5**	.5	.5	.5
16	**1**	1	**1**	**0**	1	**1**	**.5**	1	**.5**	.5	.5	.5

Extension into four dimensions follows as in the example above. Begin with eight copies of the above 24 element table and augment the table with columns of zeros and ones as above. The area of the initial pyramid is = 1/(number of pyramids) and is halved each time.

Modify ρ to be:

$$\hat{\rho_i}^n \triangleq \frac{|P_i|}{f(c_i) - M_n + g(v_n)}$$

Table 3.6 Coordinates for Three-Dimensional Partitioning – Copy 5

Pyramid	Vertices											
17	0	**0**	**0**	0	**1**	**0**	0	**.5**	**.5**	.5	.5	.5
18	0	**0**	**1**	0	**1**	**1**	0	**.5**	**.5**	.5	.5	.5
19	0	**1**	**0**	0	**0**	**0**	0	**.5**	**.5**	.5	.5	.5
20	0	**1**	**1**	0	**0**	**1**	0	**.5**	**.5**	.5	.5	.5

Table 3.7 Coordinates for Three-Dimensional Partitioning – Copy 6

Pyramid	Vertices											
21	1	**0**	**0**	1	**1**	**0**	1	**.5**	**.5**	.5	.5	.5
22	1	**0**	**1**	1	**1**	**1**	1	**.5**	**.5**	.5	.5	.5
23	1	**1**	**0**	1	**0**	**0**	1	**.5**	**.5**	.5	.5	.5
24	1	**1**	**1**	1	**0**	**1**	1	**.5**	**.5**	.5	.5	.5

where $|P_i|$ is the volume of the ith pyramid and v_n is the value of the smallest pyramid encountered to this point.

3.5 Future Work

Extending the convergence analysis for the two variate case into the three variate case and beyond shows promise. This algorithm could be used with a deployment algorithm to examine a portion of a sub-domain where the total domain is a square, cube etc. Call the class hypercubes. Hypercubes of varying sizes (accompanied by this algorithm) could be used in conjunction with a deployment algorithm to examine more closely areas where the minimum has a greater chance of existing.

Note it is not necessary to store the vertices of the triangles or pyramids for the algorithm presented. When a geometric figure is divided, the next two points can be determined as a function of how many times a division has occurred, and the direction the next points will be located.

By computing a minimizer based on the Taylor expansion and finding the point within the triangle or pyramid (or within the circle or sphere contained within) which minimizes this function, the convergence rate of the algorithm should be increased. The problem is how to modify the further decomposition of the modified grid from that point forward.

This presents a scheme for designing a front-loaded algorithm adaptable to a global minimization problem of any number of variables, limited by the ability to represent the initial decomposition of the domain into hyperpyramids. Recall the growth rate of the domain is a factorially growing function of the number of variables.

CHAPTER 4

P-ALGORITHM

Analysis of adaptive, stochastic algorithms begins with a study of the univarite (one variable) P-Algorithm as presented in [4], [5], and [6]. The P-Algorithm is an adaptive algorithm that approximates the minimum of a univariate function with a convergence rate of $h(\epsilon_n) = \frac{\epsilon_n}{n^2}$ where ϵ_n is a sequence that converges at an acceptable rate to zero (i.e., ϵ_n does not converge to zero too quickly). The P-Algorithm is based on a statistical model that is used to determine where to test the objective function for which the minimum value is sought. The P-Algorithm strikes a balance between thoroughly exploring regions of the objective function where the algorithm determines the minimum is likely to exist and exploring regions that have not been adequately tested.

The P-algorithm adopts a probability model for the objective function and then at each step determines that subset of the domain that maximizes the probability that the next function evaluation will fall below the minimum minus some threshold.

The P-Algorithm divides the domain into sub intervals by computing γ_i^n for each subinterval. γ_i^n is the measure of the likelihood that the minimum value of the objective function will be found in the corresponding subinterval and is the value used to decide which subinterval is to be chosen for the next split. The subinterval with the largest γ_i^n value will be split at (and the minimum value tested for) the location $t_i = x_{i-1}^n + \tau_n(x_i^n - x_{i-1}^n)$, where t_i is the point that maximizes the conditional probability that $\xi(x)$, a stationary Gaussian process with zero mean, unit variance, and a correlation function $r(\cdot)$, will fall below the minimum, M_n, minus some fixed threshold [6].

4.1 Error Analysis

In [6] it is shown that the P-Algorithm converges to $f(x^*)$ for

$$\gamma_i^n = \frac{x_i^n - x_{i-1}^n}{\sqrt{y_{i-1}^n - M_n + \epsilon_n} + \sqrt{y_i^n - M_n + \epsilon_n}},$$

where, for some small positive δ,

$$\epsilon = \frac{n^\delta}{n}$$

and for

$$\tau_n = \frac{1}{1 + \sqrt{1 + (y_i^n - y_{i-1}/(y_i^n - M_n + \epsilon_n))}}$$

Theorem 2 *Let $g(n) \to \infty$ as $n \to \infty$, $g(n) \in o(n)$. The P-Algorithm with $\epsilon_n = \frac{g(n)}{n}$ will converge for any continuous objective function.*

Proof 4 *Theorem 3.1 in [6] states that as long as $n_k \epsilon_{n_k} \to \infty$ the P-Algorithm will converge, where $\{n_k\}$ is a subsequence with the property $\gamma^{n_k} \to 0$* ■

4.2 Convergence Analysis

While the P-Algorithm converges to $f(x^*)$ for the class of functions $g(n) \to \infty \; as \; n \to \infty, \; g(n) \in o(n)$, the convergence rate as described in [6] in terms of the error (i.e., an upper limit on how much the P-Algorithm's minimum differs from the true minimum) can not be determined without analyzing the P-Algorithm for a given $\gamma, \epsilon_n, and \tau_n$.

If $\epsilon_n = \dfrac{\epsilon^\delta}{n}$ (where n is the step number of the P-Algorithm and δ is some small number > 0), and γ as described above, the convergence rate is $O(n^{\delta-3})$ [6].

In [4] the following parameters are used:

$$\gamma_i^n = \frac{2(x_i^n - x_{i-1}^n)}{\sqrt{y_{i-1}^n - M_n + \epsilon_n} \; + \; \sqrt{y_i^n - M_n + \epsilon_n}},$$

where

$$\epsilon = \frac{log(n)}{n}$$

and for

$$\tau_n = \frac{1}{1 + \sqrt{1 + (y_i^n - y_{i-1}/(y_i^n - M_n + \epsilon_n))}}$$

In this instance of the P-algorithm, the error is bounded as $(\dfrac{1}{12})^{\dfrac{1}{4}}$ and the convergence rate is of order $\exp(-c\sqrt{n})$ for some c > 0. Note that in computing γ_i^n it is not necessary to include factor of two in the implementation since the γ_i^n values identify the best line segment to divide by finding the maximum γ_i^n at each step. In other words, multiplying by two will not change which γ_i^n is maximum.

4.2.1 P-Algorithm Pseudocode

Pseudocode for the P-Algorithm is presented below. The values γ_i^n, and τ_n are presented as functions gamma($\cdot$) and tau($\cdot$). Since priority queue operations can execute in log(n) time where n is the number of iterations determined by the stopping criteria, for some implementations of a priority queue, the algorithm represented by the pseudocode executes in Θ(n log n) time [14].

Step 1: Set $M = min(f(x_1), f(x_0))$

Step 2: Set $n = 1$

Step 3: Set $e = \epsilon_n$

Step 4: Set $g = gamma(e, x_0, x_1)$

Step 5: Store (g, x_0, x_1) (g primary key $\rightarrow$ priority queue)

Step 6: Retrieve (Remove from db) (g, x_1, x_0)

Step 7: Set $n = n + 1$

Step 8: Set $e = \epsilon_n$

Step 9: Set $t = tau(e, x_0, x_1)$

Step 10: Set $x_t = x_0 + t(x_1 - x_0)$

Step 11: Set $M = min(M, f(x_t))$

Step 12: Set $g = gamma(e, x_0, x_t)$

Step 13: Store (g, x_0, x_t) (g primary key)

Step 14 Set $g = gamma(e, x_t, x_1)$

Step 15: Store (g, x_t, x_1) (g primary key)

Step 16: If termination condition not met goto Step 7

Step 17: M is the minimum

4.3 P-Algorithm Output

P-Algorithm test points for $f(x) = (x - .2)(x - .5)(x - .7)(x - .9)$, n=10000, $\epsilon_n = \frac{\epsilon^\delta}{n}$, $\delta = .001$ are shown in Figure 4.1. As expected, the function gets full coverage with the region around $f(0.30646561842709186) = -0.00481278069434944$ receiving the most dense coverage. For n=2000000 the minimum value observed is $f(0.3064637439488084) = -0.00481278069540427$.

In this instance, the caller object is MATLAB and values are returned to the MATLAB environment for display.

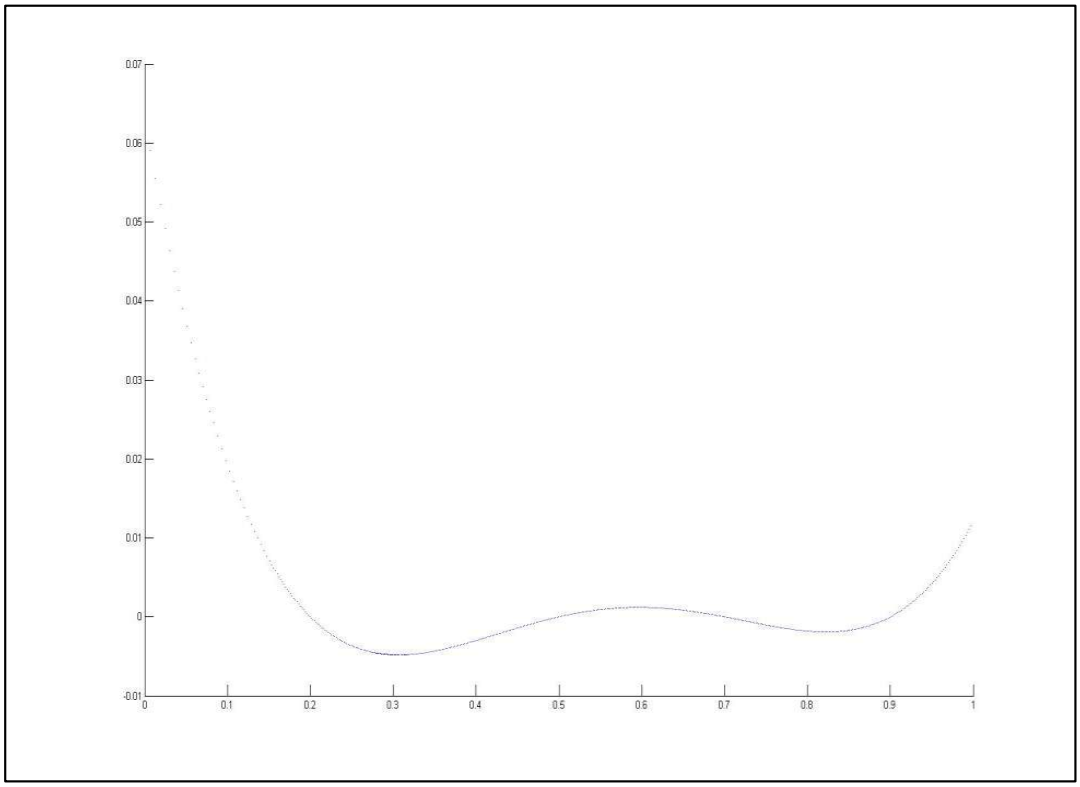

Figure 4.1 P-Algorithm observation points for $f(x) = (x-.2)(x-.5)(x-.7)(x-.9)$.

4.4 Motivation for the P-Algorithm

In Section 6.1 a diagram comprised of line segments is used to cover the domain. By mapping the endpoints of each segment to the endpoints of the P-Algorithm, a multithreadable algorithm can be deployed to find the minimum over all segments of the diagram. By rotation it is possible to map the P-Algorithm to an arc or similar geometric figure, but it is important to ensure the initial points are appropriately placed on the figure. In [21], dividing the input domain into disjoint segments is referred to as stratified sampling. In this description, the idea is to run independent, parallel algorithms on each segment which results in a global minimum estimate, $f(x^*)$ with smaller error.

However, if the initial points are set and the algorithm includes all these initial segments into its computation, the P-Algorithm will still converge to the minimum [3].

CHAPTER 5

η-ADIC GRIDS AND ALGORITHMS

In order to extend the approach taken by the P-Algorithm to multivariate functions, it is necessary to determine how the domain of the objective function shall be covered. For this class of algorithms it is most common to cover the domain by decomposing the domain into a grid of hypercubes (this includes squares and cubes for dimensions two and three, respectively).

5.1 The Grid as a Covering

Zhigljavsky [21] describes measures that provide useful information about the grid, namely ρ-dispersion, discrepancy and compositeness. These values will be defined after the G-Algorithm is described.

5.2 The G-Algorithm

An η-adic grid is formed by decomposing a line, square, cube or hypercube into η^d, η x η x η... x η (d of these, where d is the dimension of the domain being considered) hypercubes. When $\eta = 2$, the G-Algorithm recursively decomposes the domain according to the quadtree scheme. When the termination condition is met, the minimum value is tested for at the barycenter of the resultant hypercube. Stopping criteria could also be some attribute of the present hypercube. The global minimum can be tested for by either testing a point within the hypercube, or another algorithm could be invoked to search for the minimum within that hypercube. In that case, the algorithm can be viewed as a deployment algorithm, deploying a search algorithm on the hypercube.

5.2.1 G-Algorithm Pseudocode

The Pseudocode for the G-Algorithm is presented below:

Start by setting noi = η, M = ∞

Step 1: Algorithm MAlg(upper, lower, f)

Step 2: Set u = upper

Step 3: Set l = lower

Step 4: Set D = $\frac{u-l}{noi}$

Step 5: If stopping criteria met set M = min(M, minimum where tested in hypercube)

Step 6: else

Step 7: Set lt = lower

Step 8: do

Step 9: set ut = lt + D

Step 10: Call MAlg(lt,ut)

Step 11: Set lt = Inc(lt,D)

Step 12 while (lt < u)

Step 13: return M (Minimum is in M)

The purpose of the function Inc($\cdot$) is to increment through all the η^d sub-hypercubes of the hypercube being decomposed. For example, if the cube is being decomposed with $\eta = 2$, the 2^3= 8 cubes would be identified by corner points (0,0,0), (0,0.$\frac{1}{2}$), (0,$\frac{1}{2}$,0), (0,$\frac{1}{2}$,$\frac{1}{2}$), ($\frac{1}{2}$,0,0), ($\frac{1}{2}$,0,$\frac{1}{2}$), ($\frac{1}{2}$,$\frac{1}{2}$,0), ($\frac{1}{2}$,$\frac{1}{2}$,$\frac{1}{2}$). Here lower = (0,0,0), D = ($\frac{1}{2}$,$\frac{1}{2}$,$\frac{1}{2}$), where each element of D is the increment to be applied to the corresponding element of lower as each element is incremented η - 1 times.

5.2.2 G-Algorithm Output

The program output for the G-Algorithm, applied to the two-dimension Rastrigin function, 5 iterations, $\eta = 5$, is the minimum of all observed values at the barycenter of each hypercube:

G-ALGORITHM OUTPUT:
DIMENSION: 2
ITERATIONS: 5
NUMBER OF DIVISIONS AT EACH ITERATION: 5
ESTIMATED MINIMUM = 1.625225806094477E-4
MINIMUM VALUE FOUND AT POINT:
6.399999999999739E-4 6.399999999999739E-4

The above output was generated by the G-Algorithm run for 5 iterations. On each iteration each square is divided into 5 squares. The squares generated when the termination condition is met is measured at the center for the function's minimum.

5.2.3 Motivation for the G-Algorithm

The G-Algorithm has a potential application in the discipline of wireless network security. In [10] a context aware privacy (CAP) system decomposes a grid into squares in the same manner as the G-Algorithm. See Figure 5.1. The map is decomposed until the stopping criteria of a specific population per square is reached. The G-Algorithm may be used to extend the two-dimensional map into a three-dimensional model for use in urban areas where altitude may be of interest in determining locations for the context privacy algorithms used by the CAP system (here $\eta = 2$).

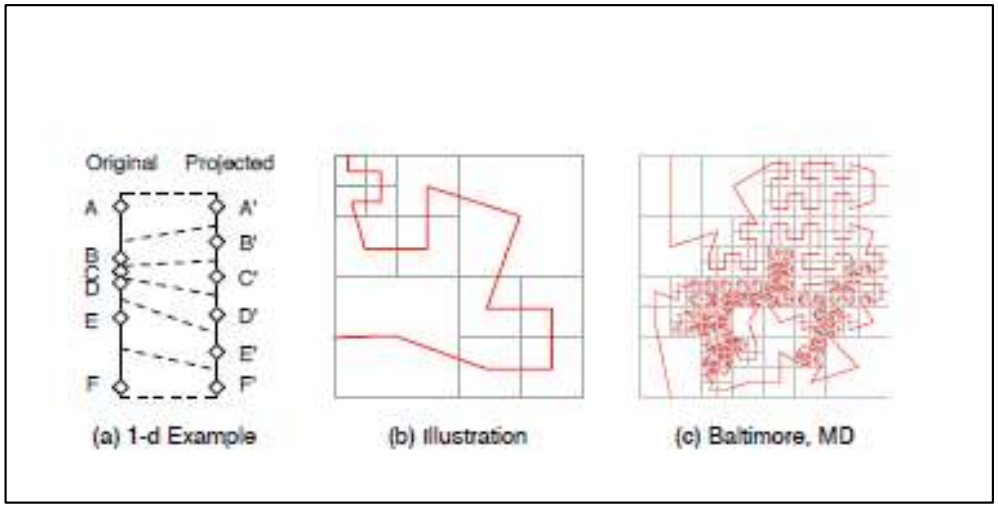

Figure 5.1 Grid decomposition for the context aware privacy system of [10].

5.2.4 Error Analysis of Passive Grids

The G-Algorithm is a passive grid algorithm as classified by [21]. A number of theoretical studies have been done on passive grids. This is because:

a) Passive grids are easily constructed.

b) Passive grids are optimal in some well-defined sense (Zhigljavsky in [21]) .

c) Passive grids are simple to investigate.

d) Passive grids lend themselves to realization on a multithreaded or multiprocessing computer.

e) Passive grids lend themselves to a recursive definition that results in a recursive decomposition of the domain.

The G-Algorithm generates a composite grid. That is, a grid that keeps its features as n (in the case of the G-Algorithm, η and the number of recursion levels) changes. Each grid element is a smaller sized hypercube, similar to all others. Many known grids do not possess this quality [21].

Theorem 3 *The dispersion of the grid Ξ_N, the grid formed by the n hypercubes and barycenters of the G-Algorithm at its termination, defined by*

$$d_\rho(\Xi_N) = \sup_{x\in X} \min_{x_i\in\Xi_N} \rho(x, x_i)$$

where $\rho(\cdot)$ is the Euclidean metric, of a G-Algorithm grid with n recursion levels divided into η sub-cubes on each iteration is $\frac{\sqrt{d}}{2\eta^{(r_L+1)}}$ where d is the dimension, η is the number of intervals each side of the hypercube is divided into and r_L is the lowest recursion level at which a minimum measurement is taken.

Proof 5 *Since minimum measurements are taken at the barycenter of each hypercube,* $\rho(x, x_i)$ *= the Euclidean distance from the barycenter of the hypercube to any corner,* $\sup_{x \in X} \min_{x_i \in \Xi_N} \rho(x, x_i)$ *will be the distance from the barycenter of the largest hypercube to any corner point of the hypercube. Let* r_L *= the recursion level where the G-Algorithm takes it's first minimum measurement. Since the G-Algorithm is a composite grid, each hypercube has the same characteristics as any other; therefore, the first measurement hypercube will be identical to the unit hypercube except it will be* $\frac{1}{\eta^{(r_L+1)}}$ *times the size of the unit hypercube.*

The distance from the barycenter to a corner of a d-dimensional hypercube is $\frac{\sqrt{d}}{2}$*. Therefore the dispersion* $d_\rho(\Xi_N) = \frac{1}{\eta^{(r_L+1)}} \frac{\sqrt{d}}{2}$ ■

While the G-Algorithm is neither adaptive nor stochastic, its study facilitates understanding grid analysis. Furthermore, it is anticipated that the G-Algorithm will be used for preliminary deployment of global optimization algorithms. Upon reaching stopping criteria, other optimization algorithms may be applied to the individual hypercubes. The analysis measures outlined in this section would then be incorporated into the overall (global) error and convergence calculations of the meta-algorithm.

5.2.5 Run Time of the G-algorithm

The run time of the G-Algorithm is $O(\eta^{dr_H})$ where r_H is the highest recursion level defined by the stopping criteria.

5.3 The Quadtree Decomposition Algorithm

The QD-Algorithm extends the grid search by assigning each hypercube a numerical value from a probability model that a minimum may lie within a region, and performing the next search at that location. The search of the domain starts as a square, cube or hypercube (depending on whether the dimension is 2, 3, or >3). At each step the domain is partitioned into smaller (η-adic) hypercubes. The minimum is sampled at the barycenter of each hypercube and that function evaluation value is used to compute the probability value. The hypercube with the highest value is selected for further partitioning into smaller (η-adic) hypercubes. The decomposed hypercube is removed at each step from evaluation. It may be helpful to view the Quadtree Decomposition (QD) algorithm as the union of the P-Algorithm and the G-Algorithm with $\eta = 2$.

The algorithm uses an adaptive quadtree (Gaede and Günther, 1998) [12] decomposition of the domain $[0, 1]^d$. At each step of the algorithm the domain will be partitioned into cubes. The components of the partition will be labeled $(i_1, i_2, \ldots, i_d, k)$ if the component is a cube of side 2^{-k} and vertex nearest the origin at

$$\left(i_1 2^{-k}, i_2 2^{-k}, \ldots, i_d 2^{-k}\right).$$

At each step of the algorithm, a cube is chosen to "split" into 2^d sub-cubes. Thus after n iterations of the algorithm, the domain will be partitioned into $n2^d - n + 1$ sub-cubes. With each iteration, the objective function will be evaluated at the center of each new cube.

After n iterations, let τ_n denote the edge length of the smallest sub-cube, ϵ_n be an appropriate O(n) sequence such that $\epsilon_n \to \infty \; as \; n \to \infty$ (such as log(n) or n^δ) and let M_n denote the smallest observed function value of the first n. For the cube with index i, $1 \leq i \leq n2^d - n + 1$, after n iterations define

$$\hat{\rho}_i^n = \frac{T_i^d}{\left(f(x_i) - M_n + (\epsilon_n \tau_n)^{5d/2}\right)^{2/5}}, \tag{5.1}$$

where T_i is the edge length of the ith cube and x_i is its center. The algorithm splits the cube with the largest value of $\hat{\rho}_i^n$ at each iteration.

5.3.1 Quadtree Decomposition-Algorithm Pseudocode

The Pseudocode for the QD-Algorithm is presented below. Strictly speaking, the Quadtree Decomposition algorithm has $\eta = 2$.

Set M = $+\infty$, $\eta = 2$, hc = initial domain

Step 1: divide hypercube hc into η-adic hypercubes

Step 2: compute ρ^* as defined in 3.1 above and place on priority queue pq, each η-adic hypercube

Step 3: Remove the hypercube with the largest ρ value from the PQ and set to hc

Step 4: while stopping criteria not met go to step 1

Step 5: return M as minimum

*Test for min when ρ is computed.

5.3.2 QD-Algorithm Convergence Analysis

Let $\hat{\rho}^n = \max_{1 \leq i \leq n} \hat{\rho}_i^n$. The algorithm will converge, in the sense that $M_n \downarrow M$, where M is the global minimum, if and only if $\liminf \hat{\rho}^n = 0$, which holds for the described algorithm if f is continuous (it need not be in C^2). Suppose that we are about to create a new smallest cube; then the cube we are subdividing must be (at least tied for) the smallest, so $T_i = \tau_n$ and $\hat{\rho}_i^n \leq 1/\log(n)^2$. Therefore, along the subsequence of times n_k when new smallest cubes are formed, $\hat{\rho}^{n_k} \to 0$. If f is twice-continuously differentiable and has a unique global minimizer in the interior of the domain, then it can be shown that

$$e_n = \exp\left(-\frac{n}{\log(n)}\Theta(1)\right).$$

5.3.3 QD-Algorithm Execution Time

The QD-Algorithm runs in $\Theta(\mathrm{n}\eta^d \log \mathrm{n}\eta^d)$ time where n is the number of iterations determined by the stopping criteria, η is the number of intervals each side of the hypercube is divided and d is the dimension. For each iteration, η^d values are placed on the priority queue which takes log $\mathrm{n}\eta^d$. Removing one entry on each iteration also takes log $\mathrm{n}\eta^d$. For $\eta = 2$ the run time is therefore $\Theta(4n^d log 4n^d)$ [14].

5.3.4 Quadtree Decomposition-Algorithm Output

The search pattern for the Quadtree Decomposition Algorithm for the Rastrigin function is shown on Figure 5.2. The mechanism by which the search values are returned to MATLAB for construction of Figure 5.2 will be covered in Chapter 9 where the Framework for the development of global optimization algorithms is discussed. Notice that the search pattern shows the contour of the objective function, in this case the two-dimensional Rastrigin function.

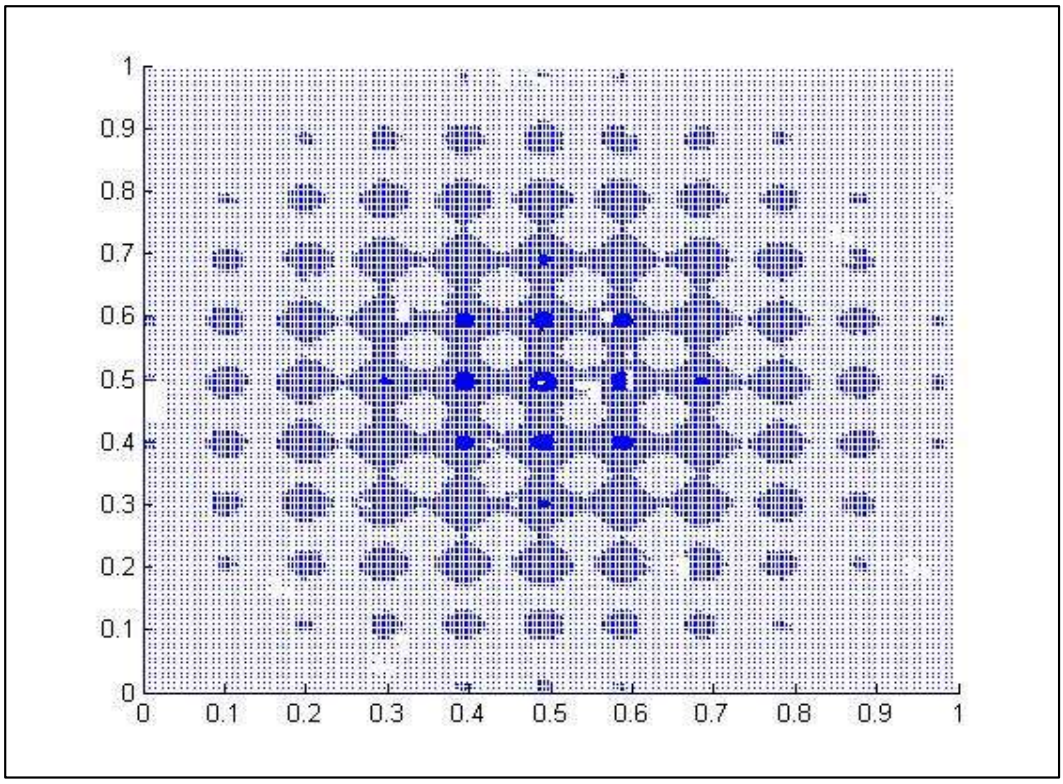

Figure 5.2 Search pattern of the QD-Algorithm for the Rastrigin function.

CHAPTER 6

ALGORITHMS THAT USE THE VORONOI DIAGRAM

Another way to form a grid over a hypercubic domain is to construct a Voronoi Diagram over the region. A Voronoi diagram is determined by a collection of points over some domain. A Voronoi diagram consists of convex, adjacent cells and one interior point such that the cell contains all points in the domain that are closest to the interior point. Each edge of the Voronoi diagram consists of all points equidistant from the interior cell point and the next-closest point in the domain.

An example of a Voronoi Diagram produced from MATLAB forming a grid over the unit square is shown on Figure 6.1. The red starred points are the centers of each Voronoi Cell. The centers of each cell are uniformly distributed over the 1 by 1 unit square.

The diagram on Figure 6.1 is constructed from 500 uniformly distributed points over the unit square. The Voronoi Diagram can be constructed in time $O(n^{\lceil d/2 \rceil})$ [1] where n is the number of points from which the diagram is generated.

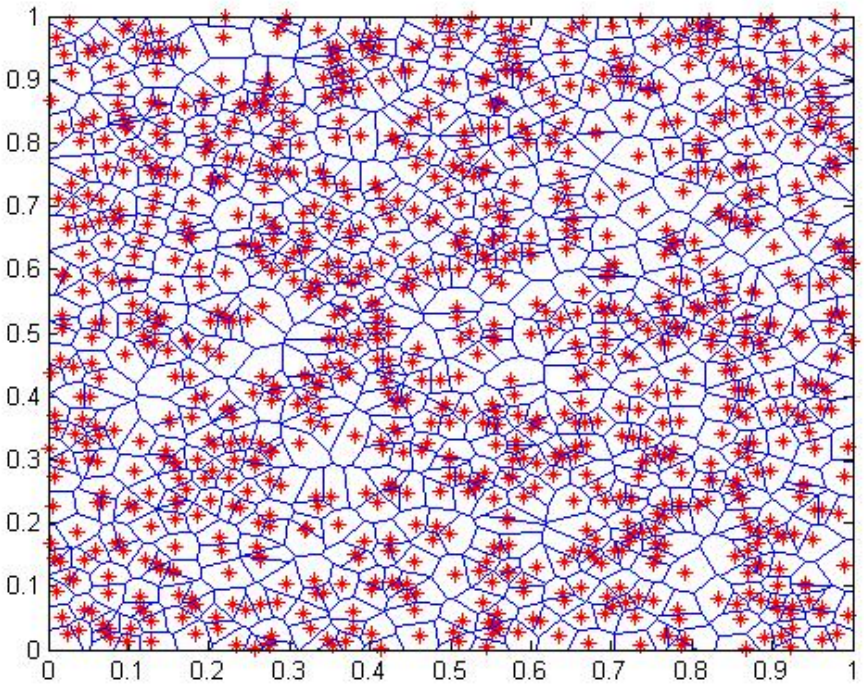

Figure 6.1 A Voronoi diagram.

6.1 The V-Algorithm

The V-Algorithm generates a Voronoi diagram over the domain by covering the domain with a field of uniformly distributed points and using those points to construct the Voronoi Diagram, each point becoming the interior point of the Voronoi diagram. Note the objective function and the Voronoi diagram may be of any dimension. The P-Algorithm is used to determine the minimum value for the objective function over each line segment of the Voronoi Diagram. This algorithm uses an existing algorithm (the P-Algorithm) combined with another (the V-Algorithm) to approximate the global minimum value of an objective function. This algorithm is also the first to use the matlabcontrol package to access a MATLAB function (voronoi()) to provide data (the line segments of the Voronoi diagram) for an algorithm.

6.1.1 V-Algorithm Pseudocode

The Pseudocode for the V-Algorithm is presented below:

Start by setting M = $+\infty$

Step 1: Generate a field of uniformly distributed points over the domain

Step 2: Build a Voronoi Diagram over the domain

Step 3: Set l = next line segment of the Voronoi diagram

Step 4: Set upper and lower bounds of the objective function

Step 5: Build a new P-Algorithm referencing the objective function

Step 6: Set mt = result of P-Algorithm run on line segment l

Step 7: Set M = min(M, mt)

Step 8: If more segments to test go to Step 3

Step 9 : Return M as minimum

6.2 The Multithreaded V-Algorithm

The Multithreaded V-Algorithm takes advantage of the independent nature of the V-Algorithm in that computing the minimum of the Voronoi diagram line segments can be done in parallel. An AlgRunner class is introduced that schedules each P-Algorithm instance as a separate thread. Each thread calls the static, synchronized function processResponse(). ProcessResponse() calls SimpleLog() to output intermediate results, and it is in the processResponse() function that the minimum value and the point where the minimum is observed is kept and made available for retrieval via the getMX() method when the algorithm terminates.

Note that in this algorithm it is necessary to create a new instance of the objective function for each thread. To facilitate that, the ObjSelectionCriteria object used to create the V-Algorithm is passed for use in creating each objective function instance.

6.3 The V-covariance Algorithm

The V-covariance algorithm sequentially constructs a Voronoi Diagram by adding a new point to the Voronoi Diagram at each iteration. In the instance provided here, the Voronoi diagram is recomputed at each iteration.

6.3.1 V-covariance Algorithm Pseudocode

The Pseudocode for the V-Covariance Algorithm is presented below:

Start by setting p = some point in the domain, M = min(p, corners of the domain)

Step 1: Set Voronoi points to the corners of the domain and p

Step 2: Compute the Inverse of the Sigma matrix

Step 3: Build a Voronoi Diagram using the corners of the domain and p

Step 4: Set bestLoc = first point of the Voronoi diagram

Step 5: Set meg = the negative gain of bestLoc (uses Sigma inverse)

Step 6: Set pt = next point of the Voronoi diagram

Step 7: Set ng = negative Gain of pt

Step 8: if (ng < meg)

Step 9: ng = meg

Step 10: bestLoc = pt

Step 11: If more points go to step 6

Step 11: Set M = min(f(bestLoc), M)

Step 12: Add bestLoc to the collection of points to build the Voronoi diagram

Step 13: If stopping criteria not met go to Step 2

Step 14 : Return M as minimum

A mechanism is provided within matlabcontrol to redisplay the Voronoi diagram at each iteration. If this algorithm is deployed, an incremental Voronoi diagram would be used. In other words, a voronoi() function would be found and used that would not rebuild the whole diagram on each iteration but would fix up the existing diagram. This would greatly decrease execution time.

The V-covariance algorithm, instead of using MATLAB functions for matrix operations, uses Java packages from Apache and Mathworks to perform the necessary matrix operations. It should also be noted that these matrix functions, if retained, would ideally be hidden behind the MathEngineFacade. However, they are directly invoked by the V-covariance algorithm to demonstrate the flexibility of the proposed framework. In other words, an algorithm is free to either use the functionality provided but the MathEngineFacade, or the functions there can be immediately replaced by other faster implementations.

For a symmetrical objective function such as the Rastrigin, it is necessary to not pick the first point, p as the center of the domain as this may cause the Sigma matrix to be singular and therefore unable to be inverted.

CHAPTER 7

ALGORITHMS THAT USE TRIANGULATION

Algorithms that divide the domain into triangular regions can be thought of as extensions of the P-Algorithm and Quadtree algorithms. Consider an instance of the P-Algorithm. The P-Algorithm minimizes a univariate function by partitioning the linear domain into smaller line segments based on a probability measure that the next best estimate is in the line segment corresponding to a computed γ value. τ is the point where the division is to take place. For the extension into the multivariate case, the value of τ is selected to simplify the area or volume computation. As in the case of the Bayesian type algorithms presented, the domain segment corresponding to the most promising region will be further divided.

7.1 HP-Algorithm Pseudocode

The Pseudocode for the HP-Algorithm (Hyperpyramid Decomposition) is presented below:

Set M = $+\infty$

Step 1: Compute the hyperpyramids for the unit hypercube (can be scaled)

Step 2: Compute ρ (as in the QC-Algorithm) for each hyperpyramid*

Step 3: Place hyperpyramids on priority queue keyed by ρ

Step 4: Remove the hyperpyramid with maximum ρ from the priority queue

Step 5: Divide hyperpyramid into $(\frac{1}{2})$ sized hyperpyramids

Step 6: Compute ρ* and place each hyperpyramid on priority queue

Step 7: if stopping criteria not met go to step 4

Step 8: return M as minimum

*test for min

7.2 Hyperpyramid Decomposition-Algorithm Output

Figure 7.1 shows the contour graph of the Rastrigin function for a two-dimensional instance of the HP-Algorithm for n=10000 and $\epsilon_n = \frac{n^\delta}{n}$, $\delta = 0.00001$. Note the domain has been scaled to an area 25x the area of the unit square. The point used to decompose the square is (0.2,0.2), (1,1) after scaling. The output is indicative of the search pattern used for the two-dimensional Rastrigin function. This search pattern is also a two-dimensional contour graph of the Rastrgin function where the initial point was chosen at (1,1). This initial selection causes the division into the four triangles shown on Figure 7.1

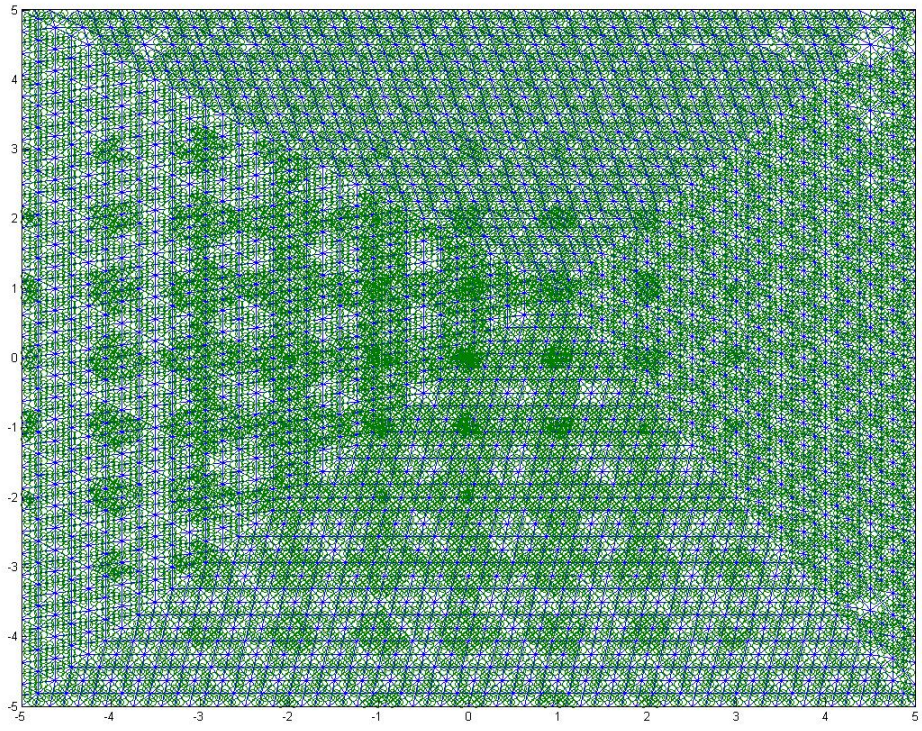

Figure 7.1 Output for the two-dimensional HP-Algorithm.

It should be noted that for the two-dimensional algorithm, if the initial point is in the center of the domain, dividing the triangles by bisecting the largest side always results in similar triangles, $\frac{1}{2}$ the size of the triangle divided. This fact can be used to simplify the area computation, as one only needs to keep track of how many times the initial triangle of area = $\frac{1}{4}$ has been halved. To speed computation, the area of triangle can be precomputed and placed in a table or a function $f(d_i)$ where d_i is the number of times the initial hyperpyramid of size $\frac{1}{p_i}$, where p_i is the number of hyperpyramids in the initial partitioning of the domain.

7.3 Partitioning the Two-Variable Domain

A scheme of partitioning the two-variate domain is shown on Figure 7.2. Here the start point of $(0.5, 0.5)$ is used to initially partition the domain. The initial area of each triangle is 0.25, and if the triangle is halved for each split (splitting by "drawing" a new edge from the midpoint of the longest edge to the free vertex) the area computation involves a division by two – or preferably by a multiplication of 0.5.

The MATLAB generated result of doing this is depicted on Figure 7.1. This indicates where the two-dimensional Rastrigin function was tested for minimum by the implementation of the algorithm. The start point – the point used to initially partition the domain into triangles, was chosen at $(1.0, 1.0)$. The performance of the algorithm is $O(nlogn)$ regardless of whether the fast area computation is used. Here the area was computed using the points of each triangle cell. In fact, the area computation for a convex polygon was used. An industrial strength algorithm would use as selectable implementation for both the splitting and area computation mechanisms and the two should be connected. In other words, an algorithm should be able to interchange them, and when splitting algorithm A_1 is used, area algorithm A_2 should have to be used.

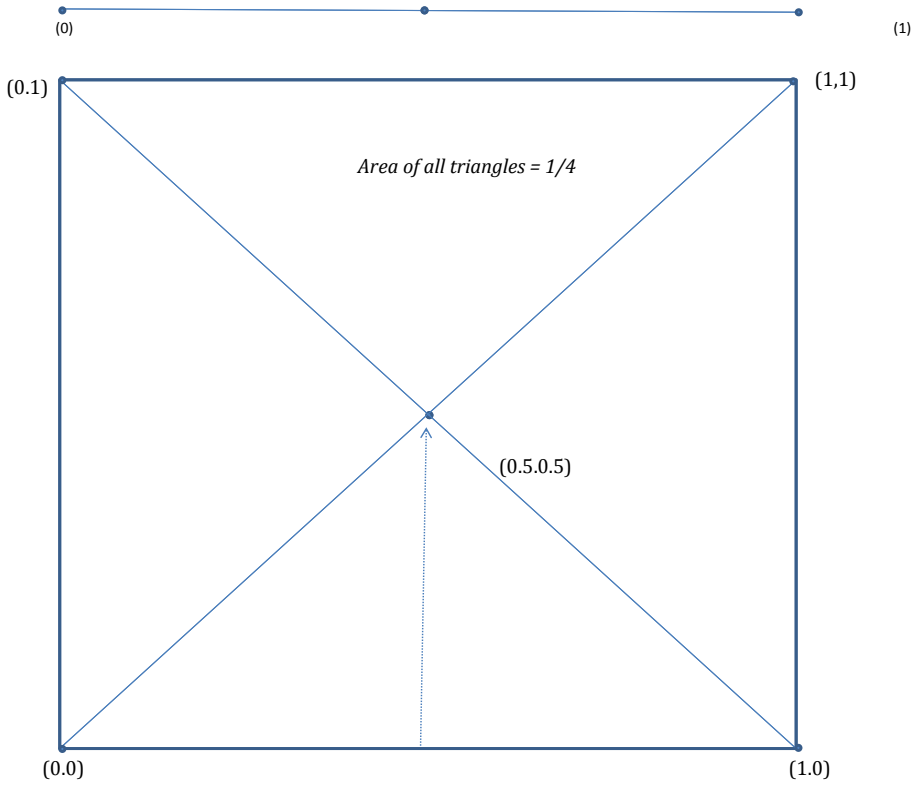

Figure 7.2 Transforming the univariate domain into a two-variate domain.

The processing mimics the P-Algorithm, but in this case a ρ value is computed for each triangle such that the triangle with the largest ρ value is the sub-domain to be halved on subsequent iterations. The triangle should be divided by determining the longest leg of the triangle and bisecting by connecting the midpoint of the longest side to the opposite vertex. This strategy sufficiently addresses the issue of the triangles deteriorating and the triangles eventually will satisfy Delauney criteria. If the start point is chosen as $(0.5, 0.5)$, all triangles meet a special case of the Delauney criteria. The Delauney criterion ensures that no vertex lies within the interior of any of the circumcircles of the triangles.

On Figure 7.2, if the triangle with the largest ρ value were (0,0), (0.5,0.5), (1.0), the triangle would be divided into the two triangles (0,0), (0.5,0.5), (0.5,0) and (0,0.5),

(0.5,0.5), (1,0). The area of the new triangle is 0.25. Note that the point (0.5,0.5) is a vertex of every triangle in the initially partitioned domain. If a point within the rectangle besides (0.5,0.5) were chosen, the coordinates of that point would be placed appropriately.

7.4 Partitioning the Three-Variable Domain

There are many ways to partition a cube into pyramids that will be explored in this section. In [21] Zhigljavski states that stratified sampling dominates the non-stratified approach in finding the optimal value in a global optimization problem. If one is interested in mapping the contour of an objective function, stratification of the domain gives a better image as long as all elements of the partition participate in the algorithm.

Put another way, consider a function that causes the selected Bayesian algorithm to converge rapidly to a minimum. If the domain is partitioned into many pyramids, by the time the algorithm would sample the last of the pyramids, these pyramids may no longer be considered for observation (and rightfully so) by the algorithm. In fact, in the case of a symmetric function, this contributes to the accuracy of the reported minimum but detracts from the display of the contour of the function. This will be demonstrated in the following sections.

7.4.1 Minimum-Pyramid Stratifying Technique

Figure 7.7 depicts stratifying a cube into 4 pyramids. Splitting the pyramids as shown by the red line segments results in pyramids $\frac{1}{2}$ the size of the pyramid being split, giving rise to the simple means of computing the volume of the new pyramid (i.e., multiplying by 0.5). While using a minimum partition tends to ensure all areas of the partition will be explored, partitioning the domain into smaller pyramids has other advantages which will be explored later.

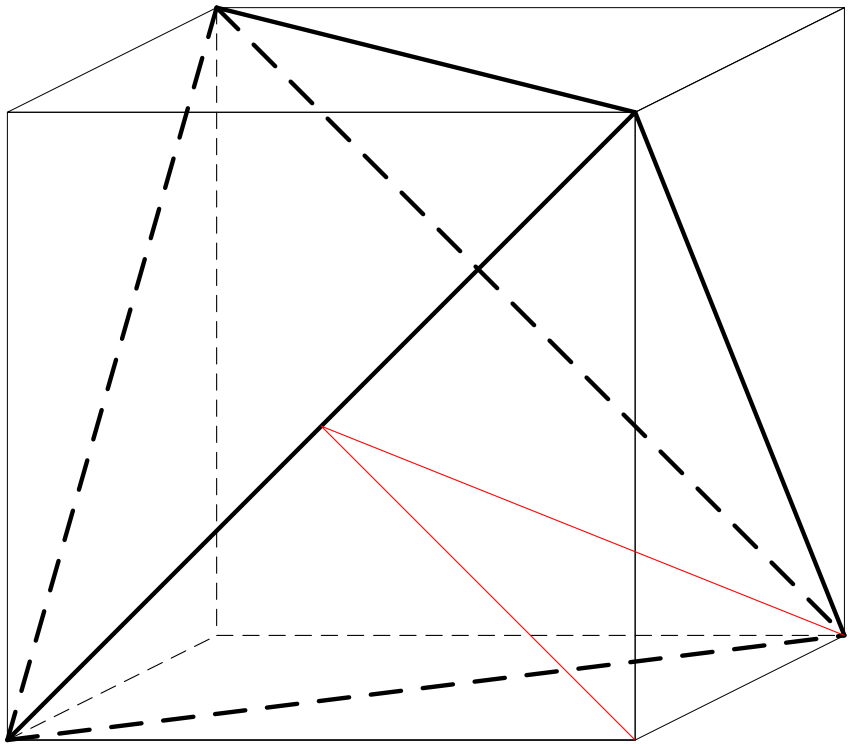

Figure 7.3 Transforming the two-variate domain into a three-variate domain.

In Figure 7.4, the minimum partitioning technique of Section 7.4.1 is shown. One million iterations of the algorithm is performed. The minimum is reported as:

MIN = 0.008371001580814053

AT: -0.001736111111111111 0.003472222222222222 0.005208333333333334

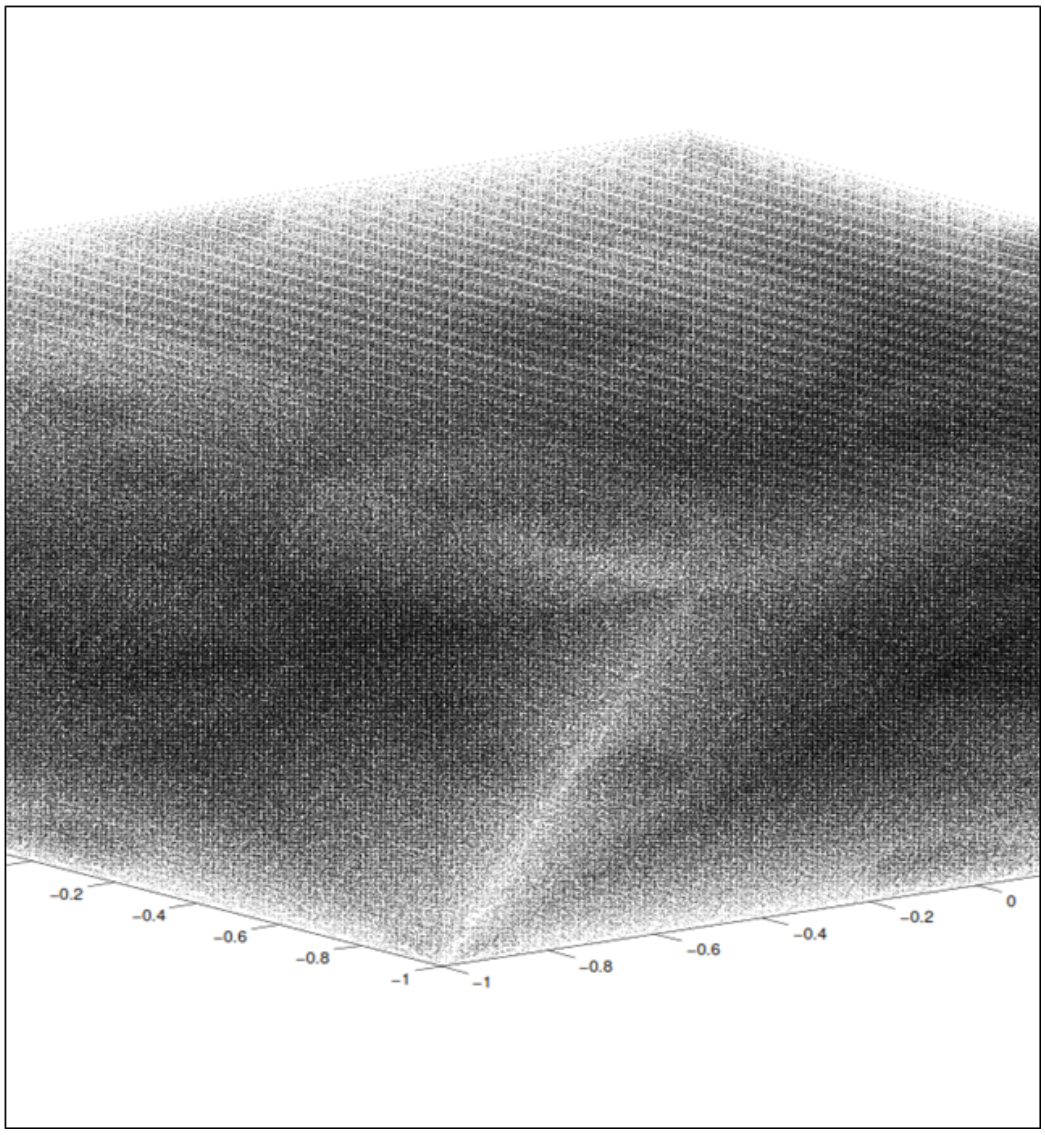

Figure 7.4 1000000 test points for three-dimensional Rastrigin function – minimum partition.

When the diagram is rotated one can see through the side of the cube, the contour (search pattern) of the algorithm:

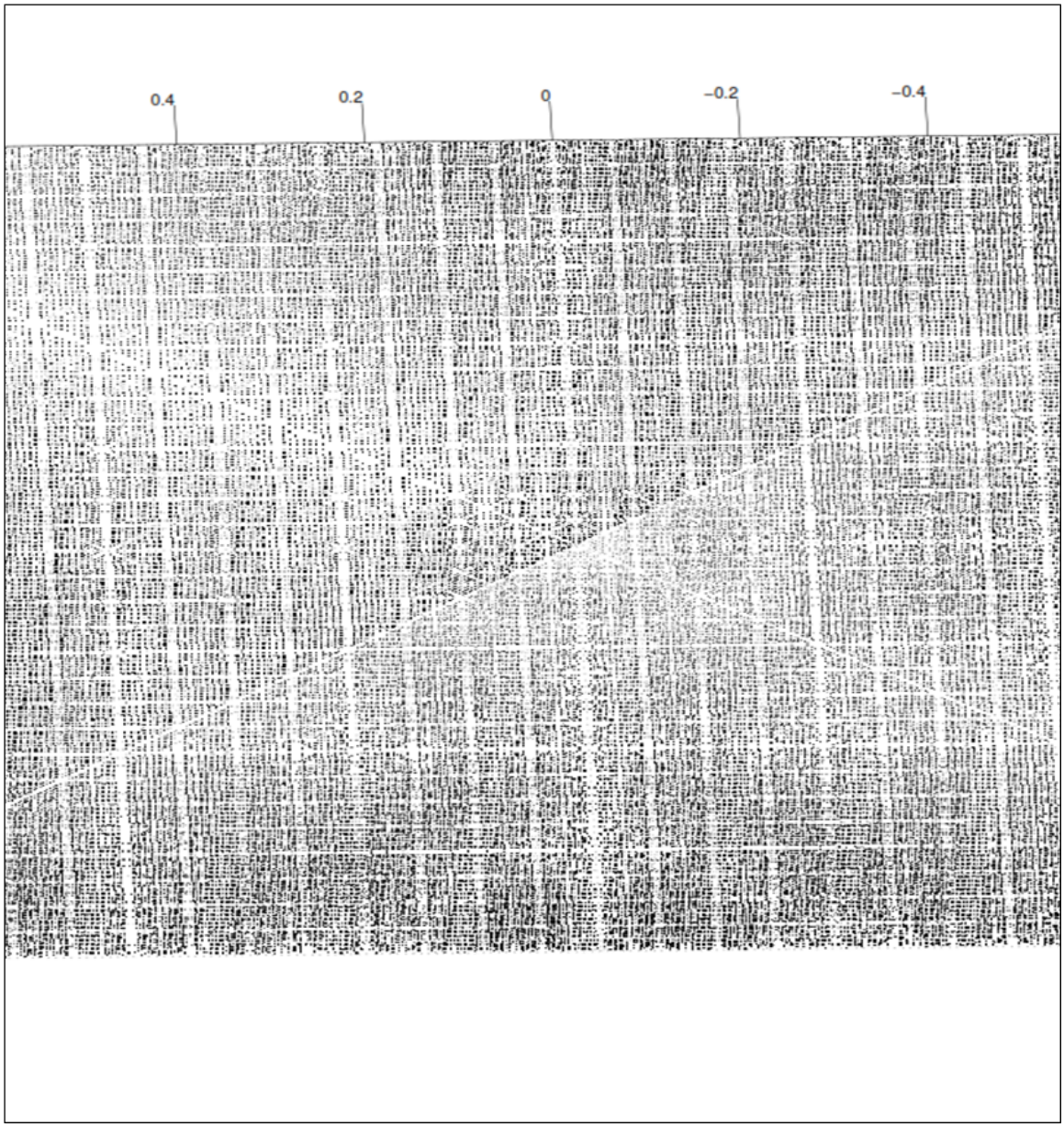

Figure 7.5 1000000 test points for three-dimensional Rastrigin function – minimum partition (rotated).

7.4.2 24-Pyramid Stratifying Technique

The contour (search pattern) of the algorithm reveals a better image of the Rastrigin function (with a poorer minimum) when the domain is partitioned into 24 equal pyramids. The minimum is reported as:

MIN = 0.010761937650649145

AT: -0.006944444444444444 -0.001736111111111111 -0.001736111111111111

The technique of how to partition the cube into 24 pyramids is described in Section 7.4.3; only the line segment connecting the start vertex to the midpoints of the edge segments is eliminated.

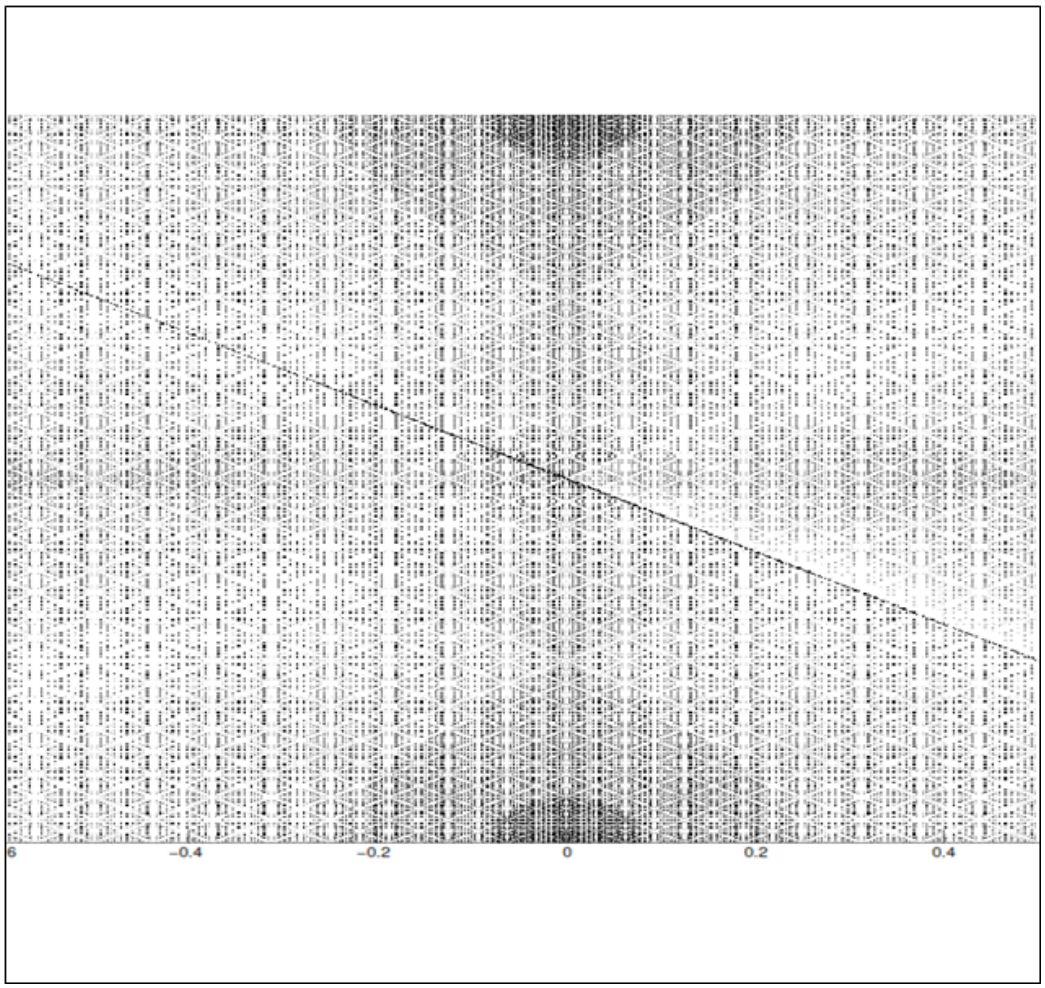

Figure 7.6 1000000 test points for three-dimensional Rastrigin function – 24 pyramid partition (rotated).

7.4.3 48-Pyramid Stratifying Technique

Figure 7.7 depicts transforming the two-dimensional triangulated domain into three dimensions in the style of selecting a start point within the unit cube. The point (0.5,0.5,0.5) (or other point within the unit cube) is added. Six of the constructs shown on Figure 7.7 are created, one for each face of the unit cube, all having in common the point just mentioned. There are eight triangles in the two-dimensional case and 48 in the three-dimensional case. For dimension d, using this method of partitioning the domain, there will be $t_d = 4t_{d-1}d$ hyperpyramids comprising the domain. Solving this recurrence, it is the case that for d dimensions, there are $t_d = 2^n n!$ hyperpyramids comprising a d dimensional domain.

It should be kept in mind that an algorithm using this partitioning scheme computes the $2^n n!$ hyperpyramids initially; the vertices are all combinations of ones and zeros connected to the added point ((0.5,0.5,0.5) in this example. Notice that when a pyramid is split by bisecting the longest side of the pyramid, the areas of the resulting two pyramids are equal.

The HP-algorithm initially partitions the domain as described, and a ρ value for each hyperpyramid is computed and saved (The minimum is tested every time a ρ value is computed). The algorithm proceeds like the P-Algorithm or HC-Algorithm, subdividing each hyperpyramid until stopping criteria is met.

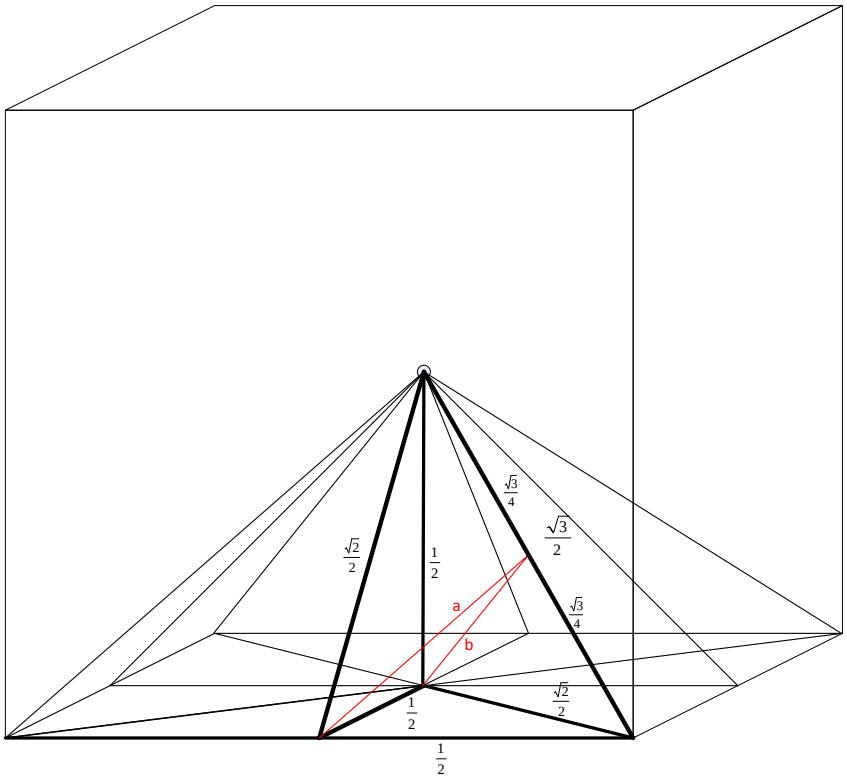

Figure 7.7 Transforming the two-variate domain into a three-variate domain.

Notice that by selecting the longest edge of a pyramid and by splitting the pyramid by drawing the lines from the halfway point to the remaining free vertices, a pyramid is created with the same base and $\frac{1}{2}$ the height. This means the volume of each pyramid is halved at each step. If the start point is chosen in the middle of the cube, the initial volume for each pyramid is $\frac{1}{48}$ the volume of the cube. Also note that the pyramids are similar, and pyramids of the same size satisfy the Delauney criterion. It is most advantageous to select the start point (i.e., the point used to partition the domain as the center point of the hypercube). Moving the limits of the domain is a preferred technique.

Table 7.1 Comparison of Delauney Triangulation vs. HP-Algorithm

Delauney Triangulation	HP-Algorithm
Triangulation moves to center minimum	Slower convergence if minimum near gridline
Goodness can vary especially near borders	All triangles have a goodness of 4
Delauney triangulation computation is expensive	Simple to compute triangulation

7.4.4 The Csendes Test Function

Recall the Csendes function given by:

$$f(x_1, x_2, x_3, ..., x_n) = \sum_{i=i}^{n} x_i^6 (2 + sin\frac{1}{x_i}) \quad x_i \in [-1, 1]\backslash 1,$$

having a countably infinite number of local minima. However, these local minima decrease as $x \to \infty$. This feature makes this algorithm converge quickly. The 48 Pyramid decomposition technique is used on this function. One can see by the Figure 7.8 that the search pattern of the algorithm is more dense towrd the center of the search area.

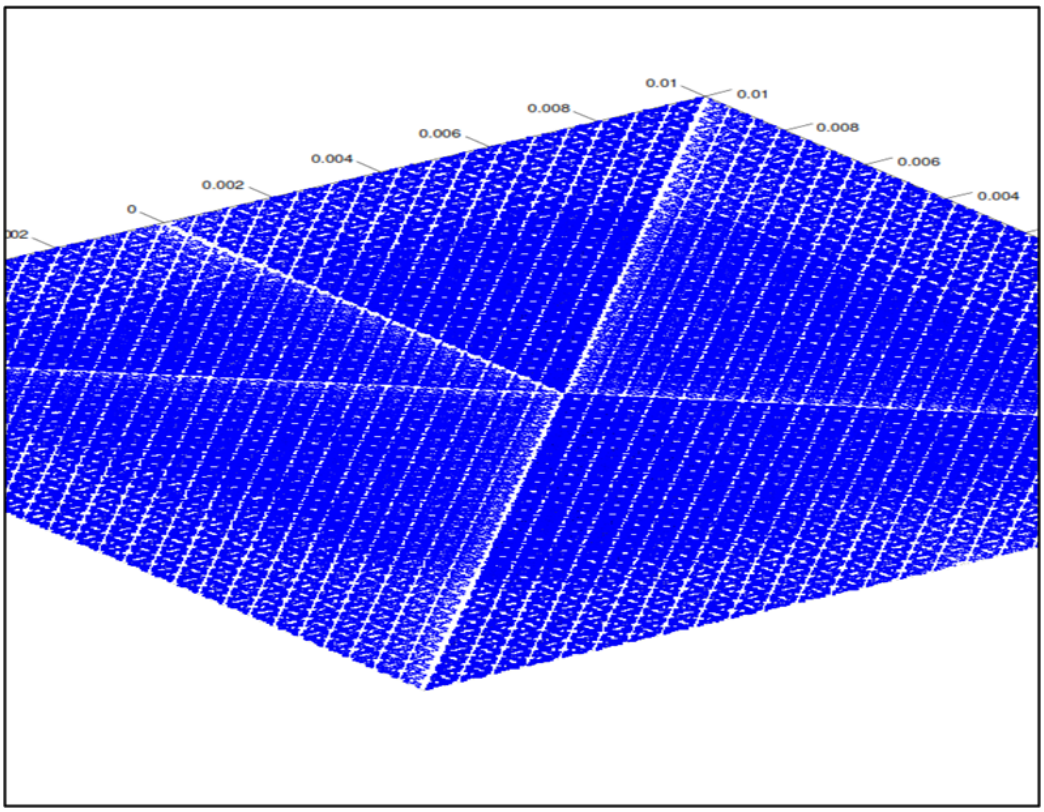

Figure 7.8 The Contour of the three-dimensional Csendes function – 5000000 iterations near the origin.

The minimum was reported on $[-.01, -.01, -.01]...[.01, .01, .01]$ as:

MIN = 3.513488340400819E-24

AT: -1.0416666666666666E-4 6.944444444444446E-5 3.472222222222223E-5.

CHAPTER 8

GLOBAL OPTIMIZATION ALGORITHMS IMPLEMENTATION

8.1 Introduction

The global optimization algorithms presented in this document are described in this section. The algorithms are presented prior to the global optimization framework on which they depend because, as in the case with developing a framework, the entities that the framework produces are based upon the attributes of the algorithms being created. For each of the algorithms, a UML diagram is presented with a realization of how the algorithm objects interact. The reader should be able to map the steps of the algorithm's pseudocode to the realizations presented in this chapter.

8.2 P-Algorithm Implementation

Recall the P-Algorithm finds the minimum value of a univariate function by partitioning the input values (the x-axis) into disjoint line segments that cover the domain. At each iteration of the algorithm, the best segment is determined. The best segment is then split into two segments and replaces the split segment. The algorithm continues executing until the stopping criteria is reached. Again, the stopping criteria can be a given number of iterations or when some error threshold is reached.

In computing a value for the best segment, one internal point of the line segment is used to compute a value to indicate how likely this segment is to contain the global minimum. The point is tested for being the global minimum at each iteration of the algorithm.

8.2.1 P-Algorithm Class Diagram

The Unified Modeling Language (UML)[11] class diagram for the P-Algorithm is shown on Figure 8.1. The framework emits an algorithm object with an objective function (and upper and lower domain bounds) associated with it. When function getMX() is called, the minimum of the function along with the location where the minimum occurred is returned. The algorithm object may be executed by any computer or thread that has access to it. It is worth noting that an algorithm object may create other algorithm objects, typically with a subset of the original domain with either the same or a different algorithm to be used. The framework classes belong to the framework for global algorithms presented in Chapter 9.

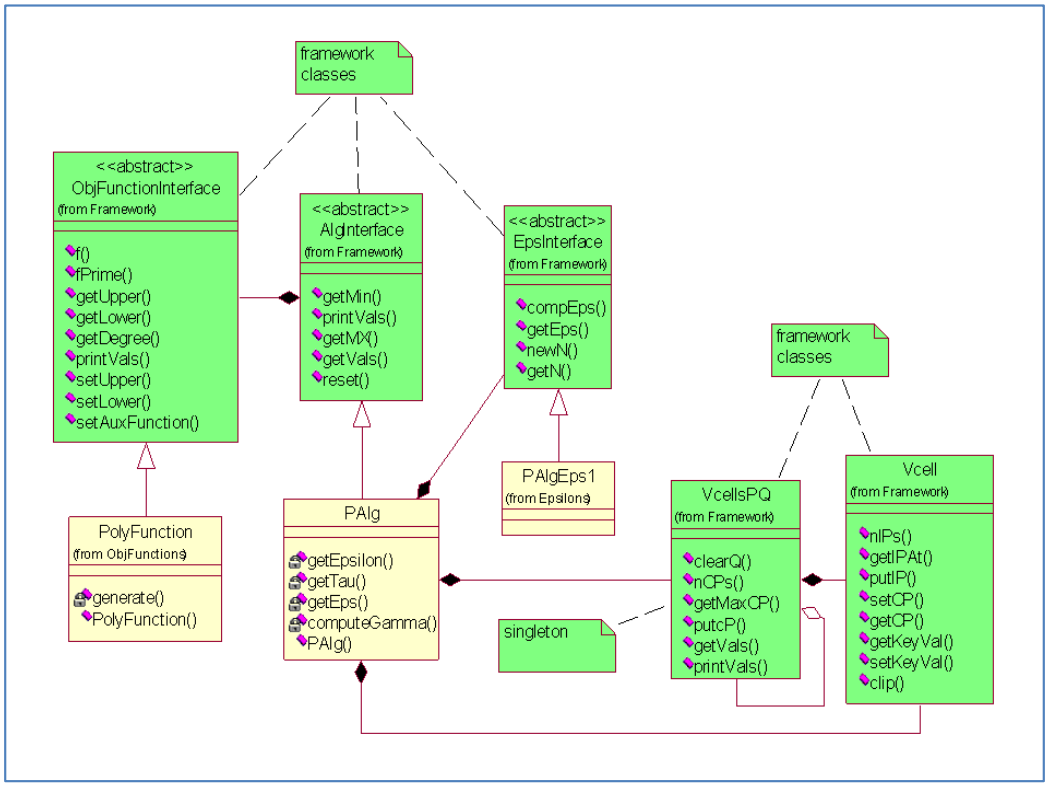

Figure 8.1 UML Class diagram for the P-Algorithm.

8.2.2 P-Algorithm Sequence Diagram

The UML sequence diagram for the P-Algorithm is shown on Figure 8.2. The PAlg object acquires the upper and lower bounds from the objective function (PolyFunction in this instance). Steps 7 through steps 15 of the P-Algorithm pseudocode are executed until the stopping criteria is met. The minimum value and where the value is observed is returned to the caller.

The steps of 8.2 are realized as:

Step 1: The lower and upper bounds and f(lower) and f(upper) are obtained

Step 2: Set n = 1

Step 3: e = epsilon(n) is computed by the internal getEpsilon(n) function which calls PAlgEps1()

Step 4: g is computed by calling computeGamma(e, x0, x1)

Step 5: (g,x0,x1) is stored in the priority queue

Step 6: (g, x1,x0) is retrieved from the priority queue

Step 7: n = n + 1

Step 8: e = epsilon(n) is computed by the internal getEpsilon(n) function which calls PAlgEps1()

Step 9: t = getTau(e, x0 ,x1)

Step 10: xt = x0 + t (x1 - x0)

Step 11: f() is evaluated for xt and M is set to min (M, f(xt))*

Step 12: g is computed by calling computeGamma(e, x0, x1)

Step 13: Store (g,x0,xt) (g primary key)

Step 14 Set g =gamma(e, xt, x1)

Step 15: (g,x0,x1) is stored in the priority queue

Step 16: If termination condition not met goto Step 7

Step 17: M is the minimum

*Note: Here and in all other realizations, f(·) is called with an InputParameters object that encapsulates the correct number of x values for the dimension, the function evaluation at that point, as well as useful functions such as norm(), add(), subtract() etc. The InputParameters class is described in detail in Chapter 9.

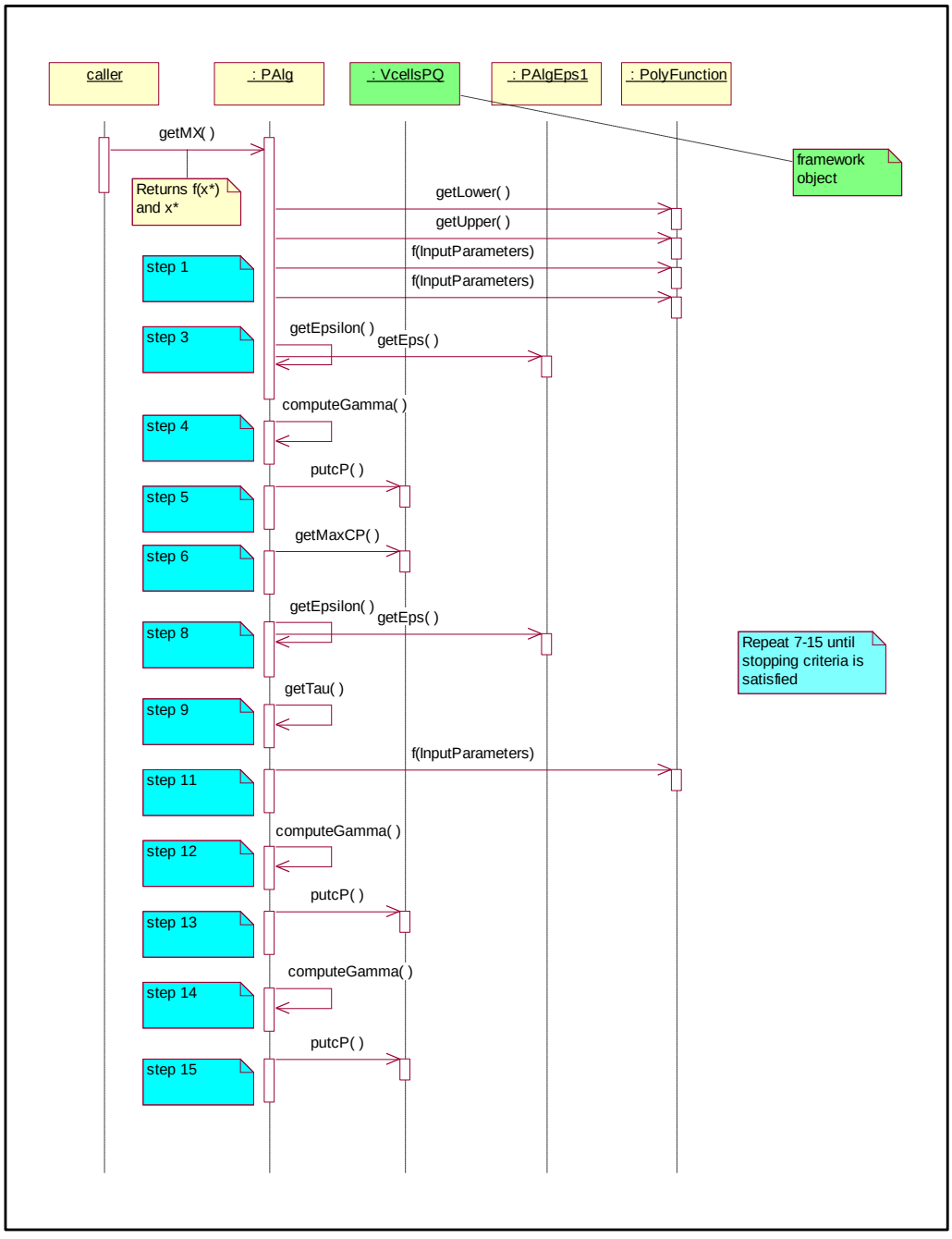

Figure 8.2 UML sequence diagram for the P-Algorithm.

8.3 G-Algorithm Implementation

The G-Algorithm recursively partitions the hypercubic domain (e.g., square, cube, or hypercube) into smaller components. It produces a composite grid over the domain. Recall an example of the city map being partitioned into smaller squares until a target population density is encountered. It is envisioned that this algorithm, instead of testing at the terminal partition, would then deploy another algorithm (e.g the Quadtree Decomposition Algorithm) to explore the area.

8.3.1 G-Algorithm Class Diagram

The class diagram for the G-Algorithm is shown on Figure 8.3. The Rastrigin objective function is used, but any suitable objective function can be minimized. A new G-Algorithm instance is created for each sub-hypercube tested. At the lowest recursion level, the sub-hypercube is tested for minimum value by testing a point or points, or by possibly deploying a different algorithm to search and test the sub-hypercube.

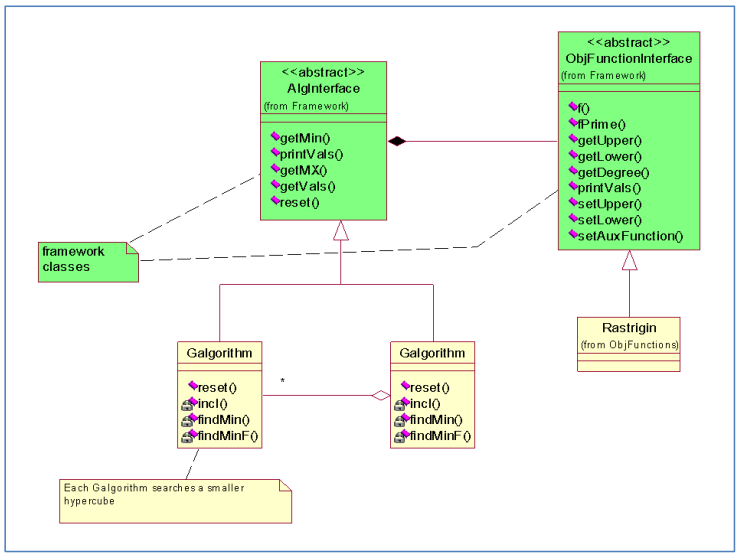

Figure 8.3 UML class diagram for the G-Algorithm.

8.3.2 G-Algorithm Sequence Diagram

The sequence diagram for the G-Algorithm is shown on Figure 8.4. Instances are created recursively until the stopping criteria is met. The objective function is tested at the lowest level of recursion

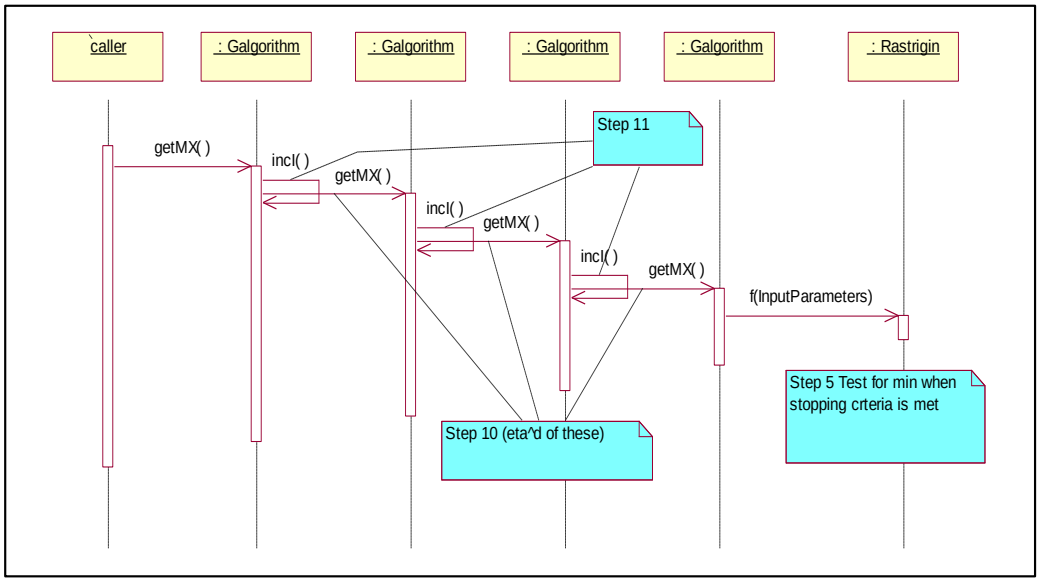

Figure 8.4 UML sequence diagram for the G-Algorithm.

The realization of the G-Algorithm is presented below:

Step 1: caller calls G-Algorithm(upper, lower, f)

Step 2: u = upper

Step 3: l = lower

Step 4: Set D to $\frac{u-l}{noi}$

Step 5: If stopping criteria met call the function to get f(x) Set M = min(M, f(x))

Step 6: else

Step 7: lt is set to lower

Step 8: do

Step 9: ut is set to lt + D

Step 10: Create a new G-Algorithm. Call G-Algorithm(upper, lower, f)

Step 11: lt is set to Inc(lt,D)

Step 12 while (lt < u)

Step 13: return M (Minimum is in M)

8.4 Quadtree Decomposition-Algorithm Implementation

The Quadtree Decomposition Algorithm stratifies (partitions) the initial hypercube into 4 equal areas. It proceeds in a similar fashion as the P-Algorithm in that goodness values are computed for each hypercube. At each iteration, the best hypercube is further decomposed until stopping criteria is reached.

8.4.1 Quadtree Decomposition-Algorithm Class Diagram

The class diagram for the Quadtree (HC) Algorithm is shown on Figure 8.5. The HC Algorithm maintains a priority queue (encapsulated within the HCPoints singleton object) of HCPoint objects. The HCPoint objects contain the information required to split the hypercube, the barycenter of the hypercube and the computed probability

value, ρ. The Rastrigin objective function is used but any suitable objective function can be minimized.

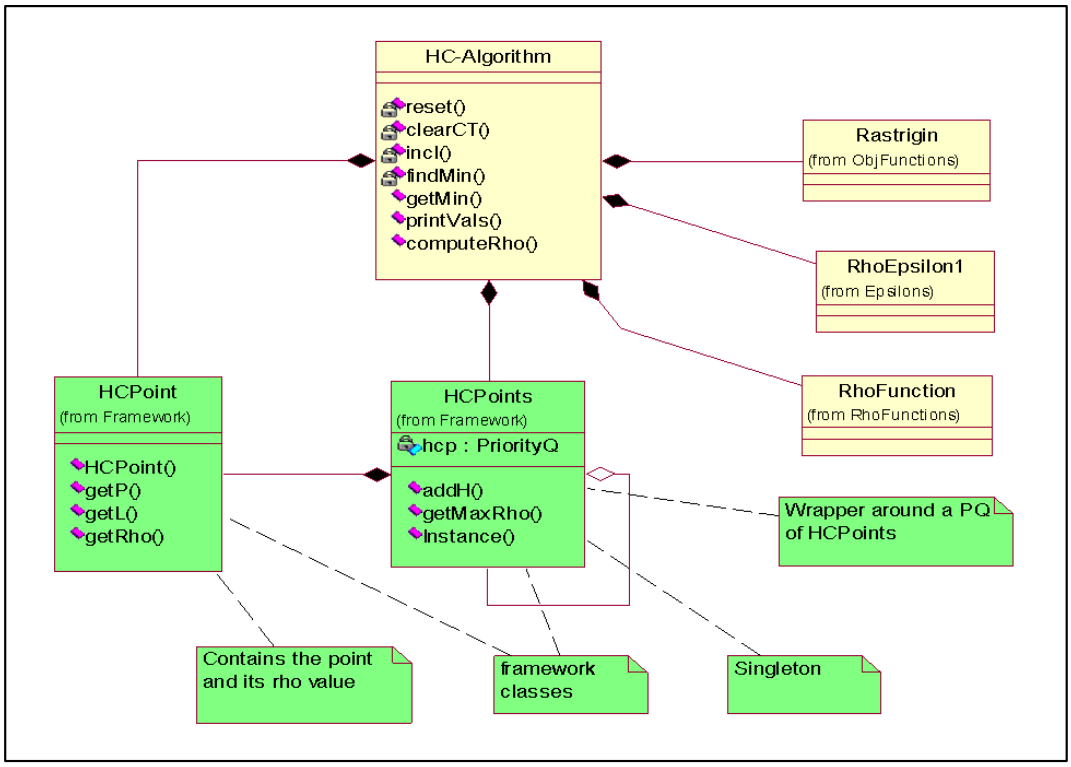

Figure 8.5 UML class diagram for the quadtree decomposition HC-Algorithm.

8.4.2 Quadtree Decomposition-Algorithm Sequence Diagram

The sequence diagram for the HC-Algorithm is shown on Figure 8.6.

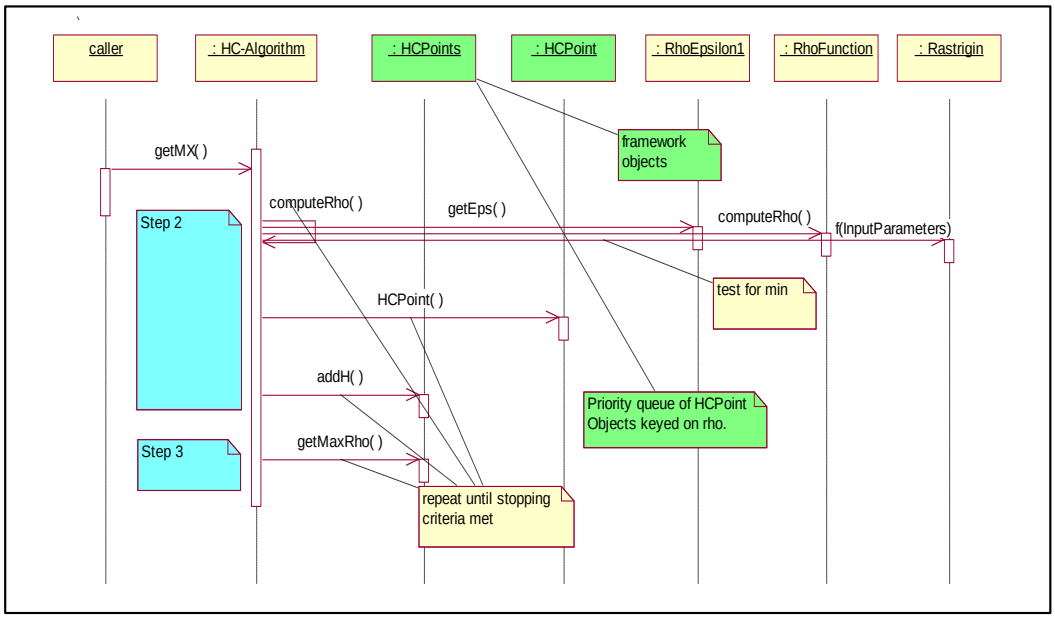

Figure 8.6 UML sequence diagram for the HC-Algorithm.

The realization of the HC-Algorithm is presented below:

Step 0: caller calls HC-Algorithm(hc)

Step 1: hc is divided into sub-hypercubes

Step 2: computeRho() is called which calls RhoEpsilon1() to get epsilon used to call RhoFunction() to compute ρ for each sub-hypercube. addH() is called for each sub-hypercube to place it on the PQ.

Step 3: The hypercube with the max ρ value is removed by calling getMaxRho() and is assigned to hc.

Step 4: If the stopping criteria is not met go to Step 1

Step 5: Return f(x^*) and x^*

8.5 V-Algorithm Implementation

The V-Algorithm generates a number of uniformly distributed points over the domain and uses those points to create a Voronoi diagram covering the domain. The P-Algorithm is executed over each of the line segments comprising the Voronoi diagram. MATLAB is used to create the Voronoi diagram and return it to the Algorithm through the MathFacade component. Note that this mechanism works well for the development and analysis of these algorithms. However, if this or any other algorithm is deployed for industrial use, a faster Voronoi diagram creation function may be found or developed to eliminate execution time problems arising from the application accessing MATLAB through the matlabcontrol proxy interface.

While a grid created in this manner is a passive grid, the P-Algorithm comprises the adaptive, stochastic piece of this algorithm. This algorithm demonstrates how different algorithms may be combined and leads to ensuring that the framework contains the functionality to allow this to occur. Additionally, each P-Algorithm requires no information from the other P-Algorithms execution and these P-Algorithms may be executed independently.

8.5.1 V-Algorithm UML Class Diagram

The UML class diagram for the V-Algorithm is presented on Figure 8.7. Notice that the V-Algorithm uses framework components AlgSelectionCriteria, AlgFactory, AlgInterface and ObjFunctionInterface to create instances of P-Algorithm. In reality, the V-Algorithm and the algorithms presented up to this point are created in a similar fashion. The reason ObjFunctionFactory and ObjSelectionCriteria are not used is because the Objective function that the V-Algorithm was created with is used, only the limits are changed for each execution of the P-Algorithm.

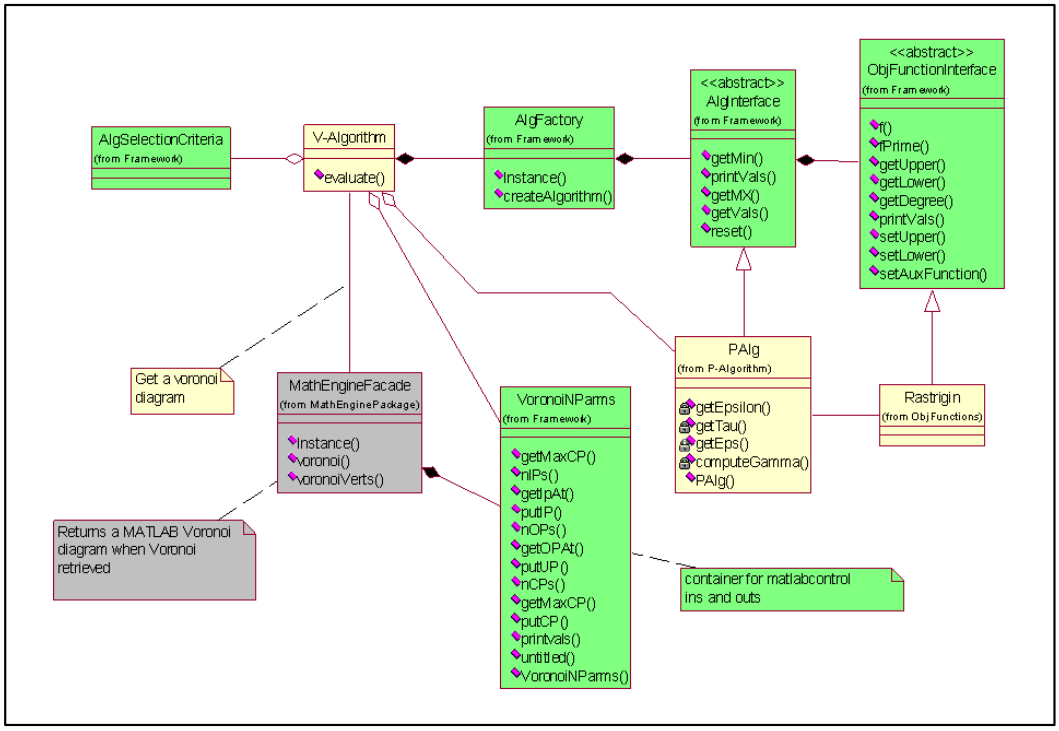

Figure 8.7 UML class diagram for the V-Algorithm.

8.5.2 V-Algorithm UML Sequence Diagram

The realization of the V-Algorithm is presented below:

Step 1: The random function of Java is used to generate a specified number of uniformly distributed points. These are placed in VornoiNParams, the interface object for the voronoi function

Step 2: The voronoi() function of the MathEngineFacade is called. In turn the required matlabcontrol functions are called to retrieve the line segments of the newly created Vornoi diagram

Step 3: Set l = next line segment of the Voronoi diagram

Step 4: setUpper() and setLower are set to the endpoints of the next line segment.

Step 5: Build a new P-Algorithm referencing the objective function

Step 6: Set mt = result of P-Algorithm run on line segment l

Step 7: Set M = min(M, mt)

Step 8: If more segments to test go to Step 3

Step 9 : Return M as minimum

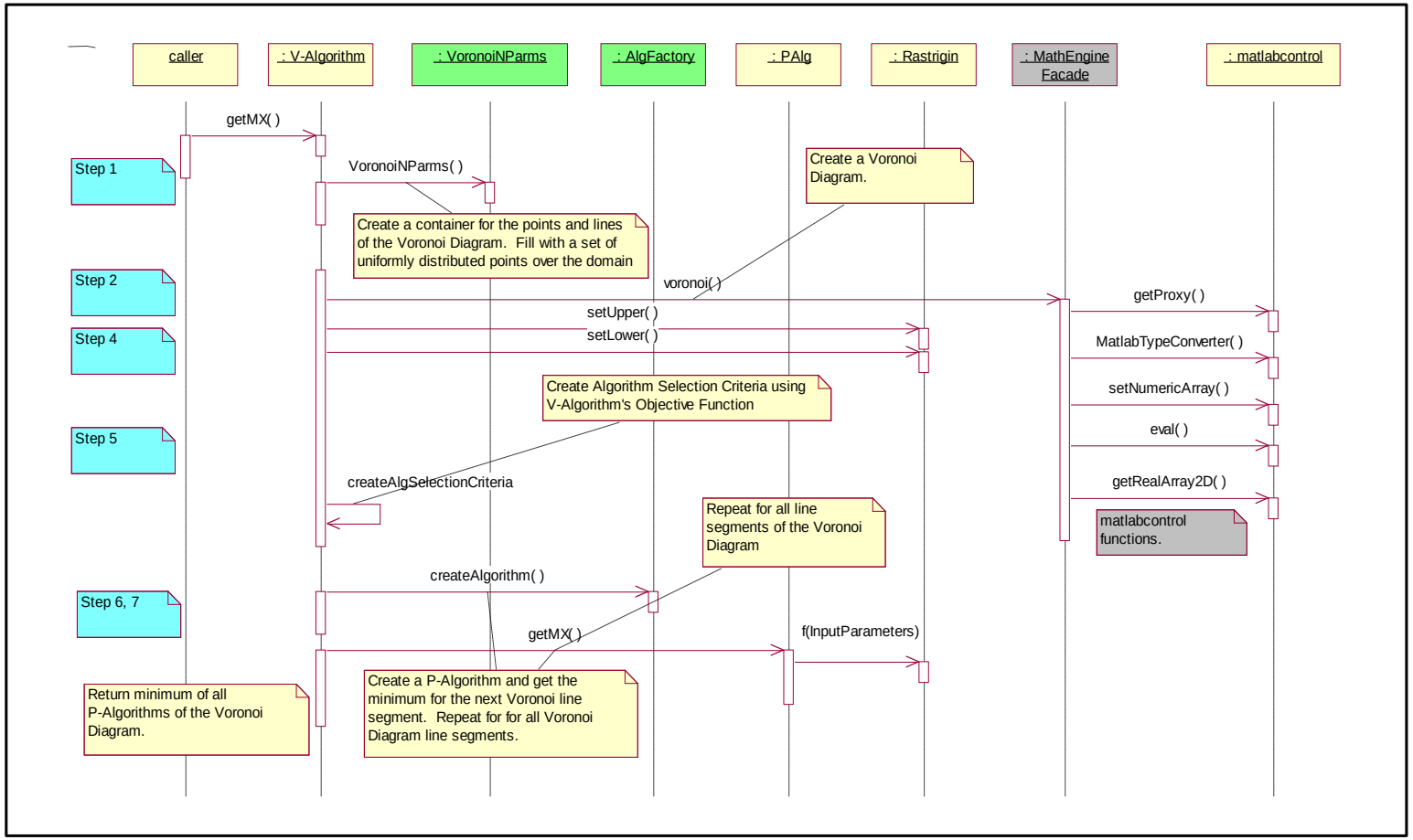

Figure 8.8 UML sequence diagram for the V-Algorithm.

8.6 Multithreaded V-Algorithm Implementation

The Multithreaded V-Algorithm executes each P-Algorithm as a separate thread. The AlgRunner class is introduced to enable algorithms to execute their independent pieces in parallel. AlgRunner contains a synchronized processResponse method that the threads use to report their minimum result.

8.6.1 Multithreaded V-Algorithm UML Class Diagram

The UML class diagram for the Multithreaded V-Algorithm is presented on Figure 8.9. The ObjFunctionSelectionCriteria is required since each tread must have its own instance of the objective function. An AlgRunner class is added to schedule the threads , limited by the java threadpool mechanism. AlgRunner contains the synchronized processResponse() method to capture the minimum reported value of the P-Algorithms.

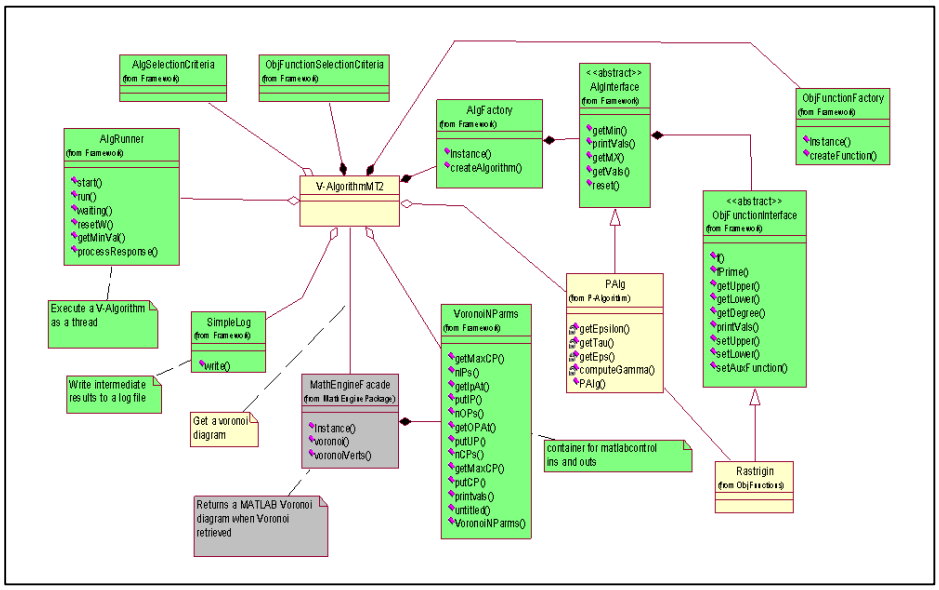

Figure 8.9 UML class diagram for the multithreaded V-Algorithm.

8.6.2 Multithreaded V-Algorithm UML Sequence Diagram

The realization of the Multithreaded V-Algorithm is presented below:

Step 1: The random function of Java is used to generate a specified number of uniformly distributed points. These are placed in VornoiNParams, the interface object for the voronoi function

Step 2: The voronoi() function of the MathEngineFacade is called. In turn the required matlabcontrol functions are called to retrieve the line segments of the newly created Voronoi diagram

Step 3: Set l = next line segment of the Voronoi diagram

Step 4: Use the objFunctionSelectionCriteria to create a new instance of the objective function.

Step 5: Build a new P-Algorithm referencing the objective function

Step 6: Create and run the P-Algorithm thread (log intermediate values). Thread calls processResponse() upon completion to determine minimum.

Step 7: If more line segments go to Step 3

Step 8 : After all threads have completed call getMinVal() to retrieve the minimum and where the minimum occurred. Return M as minimum

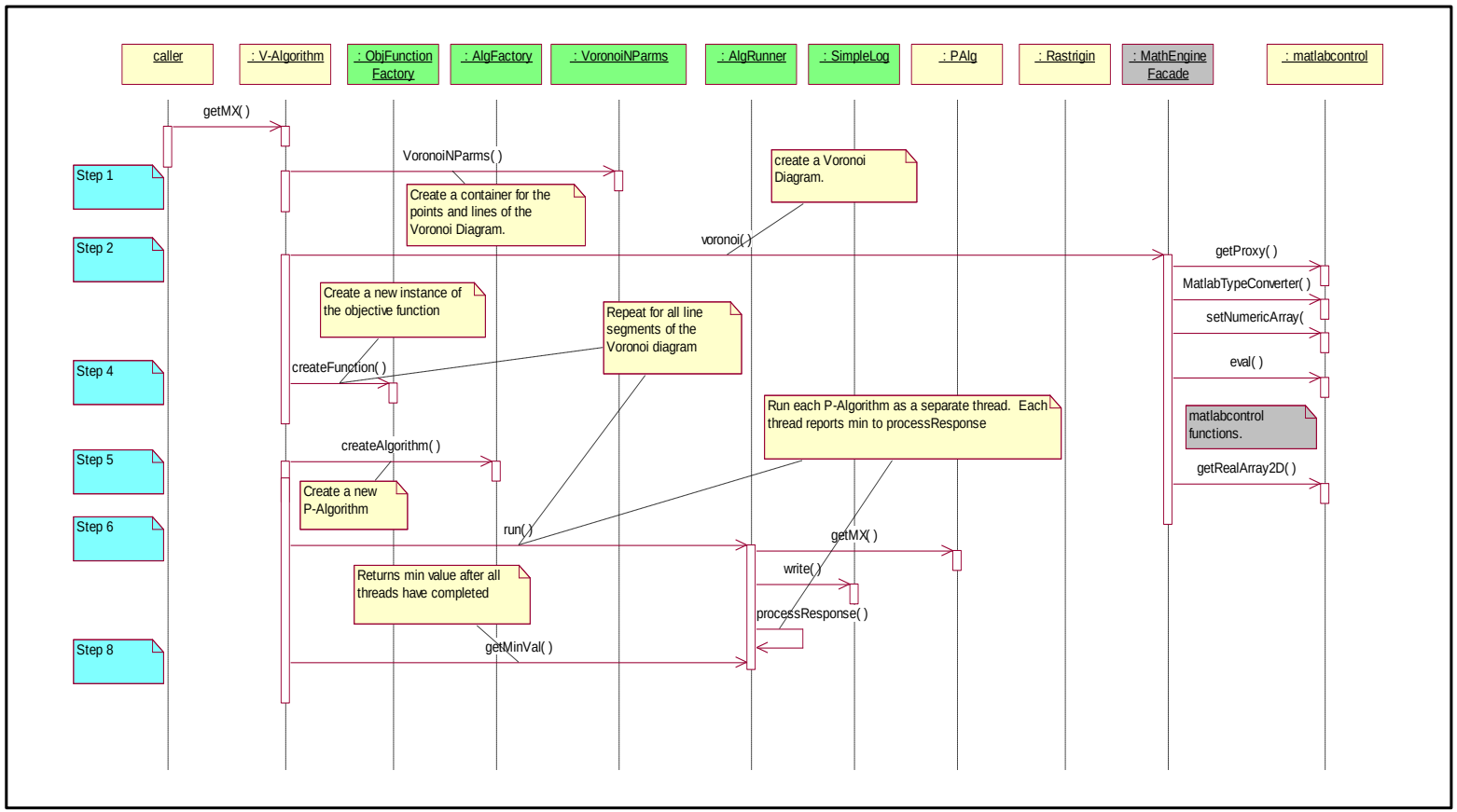

Figure 8.10 UML sequence diagram for the multithreaded V-Algorithm.

8.7 V-covariance Algorithm Implementation

The V-Covariance algorithm retrieves a Voronoi Diagram for a minimal partitioning of the domain. At each iteration, a new point is generated and is used to compute a new Voronoi diagram covering the domain. A way to speed this algorithm would be to develop or find a function that would rebuild the Voronoi diagram by reconstructing the cells modified by adding the new point instead of rebuilding the entire diagram on each iteration.

8.7.1 V-Covariance Algorithm UML Class Diagram

The UML diagram for the V-Covariance algorithm is presented on Figure 8.11. Instead of using matlabcontrol, the V-Covariance class includes matrix computation libraries for Apache and Mathworks directly. An alternative (preferable) approach is to hide the matrix operations behind the MathEngineFacade. This was not done to demonstrate the flexibility of the framework in that an application is not required to use the framework implementations of mathematical functions if others are desired.

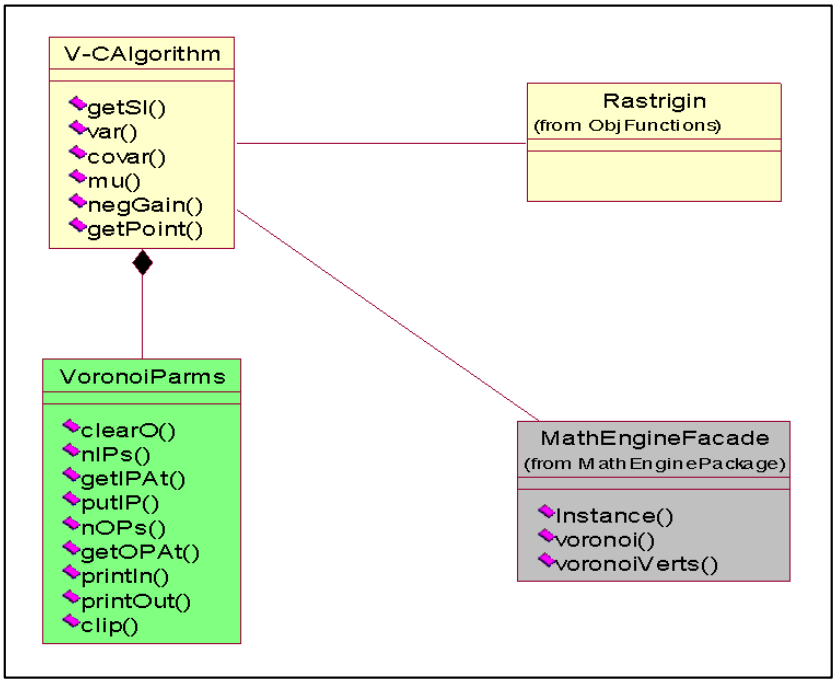

Figure 8.11 UML class diagram for the V-Covariance Algorithm.

8.7.2 V-covariance Algorithm UML Sequence Diagram

The realization of the V-Algorithm is presented below:

Step 1: Set Voronoi points to the corners of the domain and p

Step 2: Call getSI()to compute the inverse of the Sigma Matrix

Step 3: Call voronoi() to build a Voronoi Diagram using the corners of the domain and p

Step 4: Set bestLoc = first point of the Voronoi diagram

Step 5: Compute meg, the negative gain of bestLoc, using Sigma inverse

Step 6: Set pt = next point of the Voronoi diagram

Step 7: Set ng = negGain() of pt

Step 8: if (ng < meg)

Step 9: ng = meg

Step 10: bestLoc = pt

Step 11: If more points go to step 6

Step 11: M = min(f(bestLoc), M)

Step 12: Call putIP() to add bestLoc to collection of points to build the next Voronoi diagram

Step 13: If stopping criteria not met go to Step 2

Step 14 : Return M as minimum

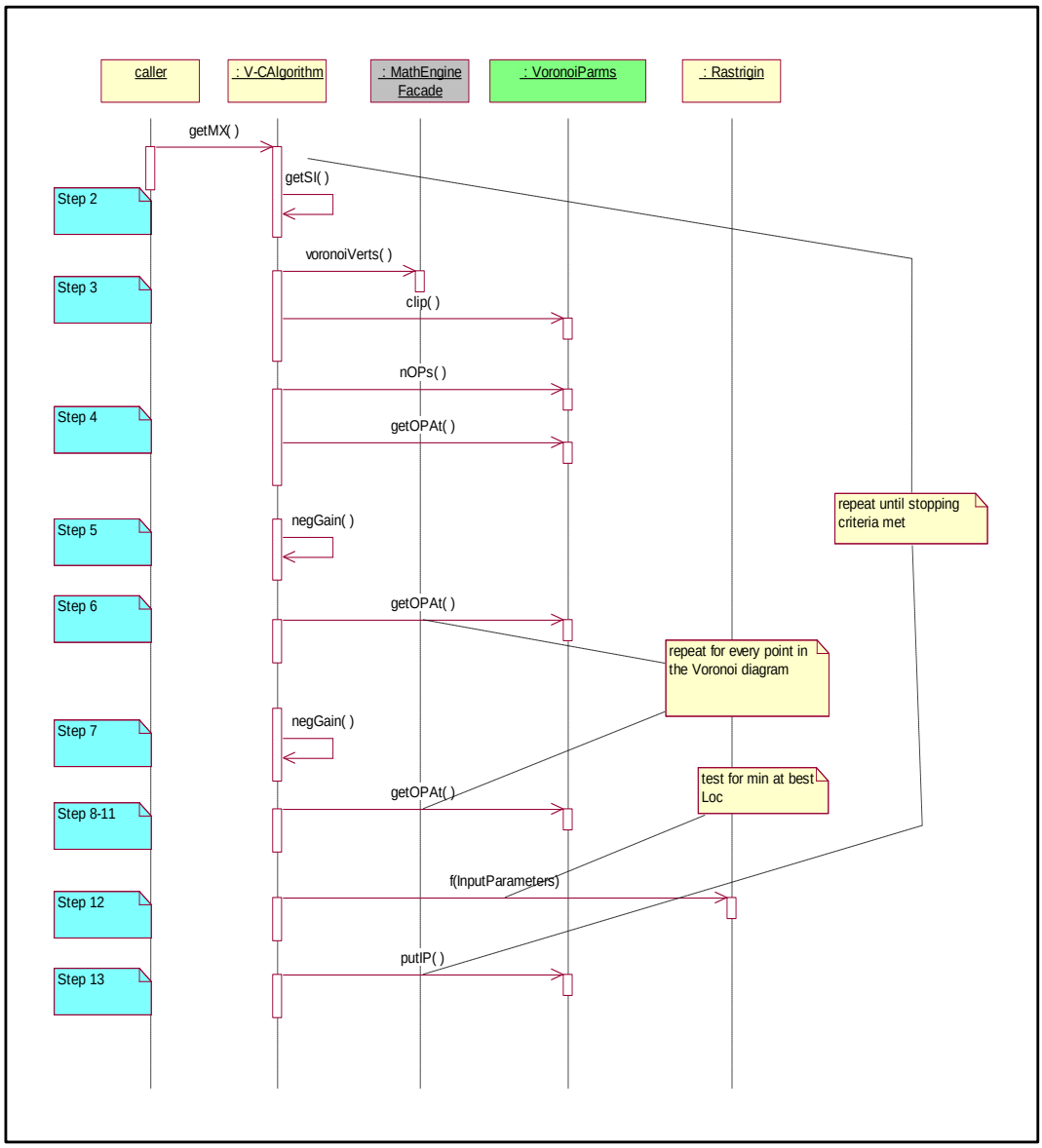

Figure 8.12 UML sequence diagram for the V-Covariance Algorithm.

8.8 HP-Algorithm Implementation

The HP-Algorithm stratifies the hypercubic domain into a number of pyramids. As in the case of the Quadtree Decomposition algorithm, each initial partition is assigned a value which signifies how likely the global minimum resides in that hyperpyramid, while testing for the global minimum at the center of the pyramid. The algorithm runs until stopping criteria is met.

In developing this algorithm, it becomes apparent that the functions which initially partition the domain, perform the split on the hyperpyramid being split (and determine the area) and the functions used in the goodness computation (ρ *or* γ) (reference Abstract Factory or Builder in [13]).

8.8.1 Hyperpyramid Decomposition-Algorithm Class Diagram

The class diagram for the Hyperpyramid Decomposition (HP) Algorithm is shown on Figure 8.13.

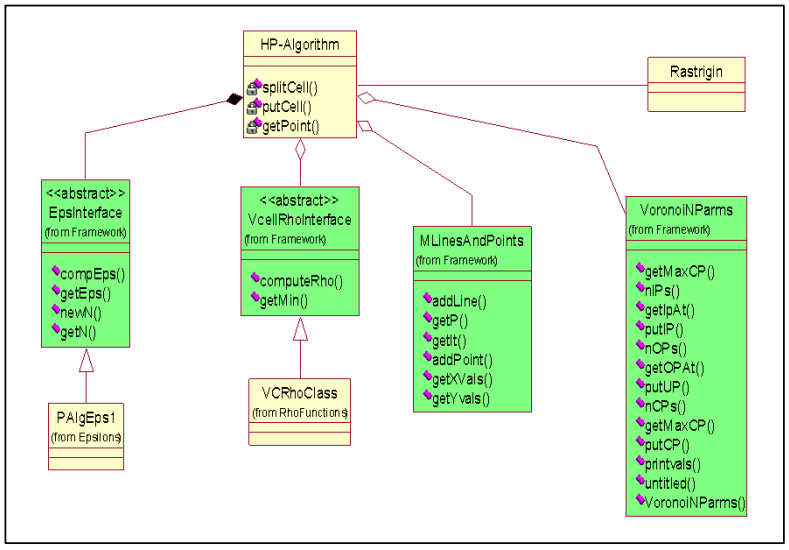

Figure 8.13 UML class diagram for the HP-Algorithm.

8.8.2 Hyperpyramid Decomposition-Algorithm UML Sequence Diagram

The UML sequence diagram for the Hyperpyramid Decomposition (HP) Algorithm is shown on Figure 8.14.

The Realization for the HP-Algorithm (Hyperpyramid Decomposition) is presented below:

Step 1: Compute the hyperpyramids for the unit cube

Step 2: Compute ρ for each hyperpyramid by calling computeRho()

Step 3: Place hyperpyramids on priority queue keyed by ρ by calling putCP()

Step 4: Remove the hyperpyramid with maximum ρ from the priority queue by calling getMaxCP()

Step 5: Divide hyperpyramid into smaller ($\frac{1}{2}$) hyperpyramids by calling splitCell()

Step 6: Place hyperpyramids on priority queue keyed by ρ by calling putCP()

Step 7: if stopping criteria not met go to step 4

Step 8: return M as minimum

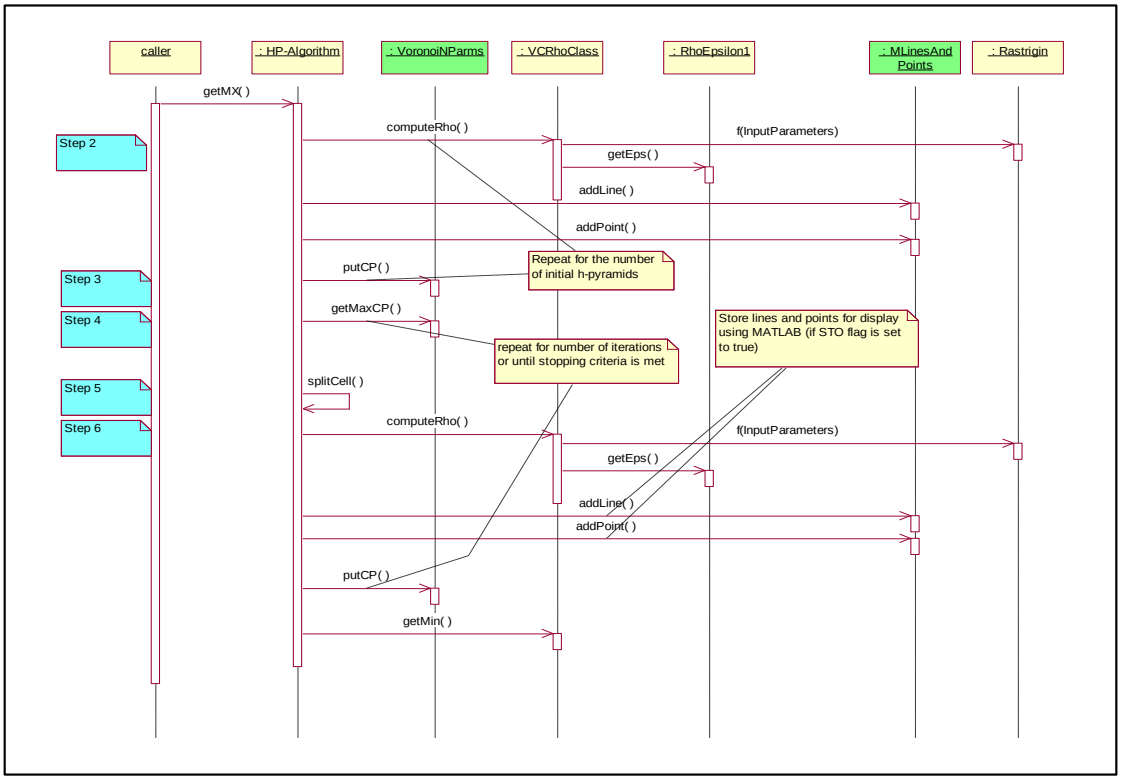

Figure 8.14 UML sequence diagram for the HP-Algorithm.

CHAPTER 9

GLOBAL OPTIMIZATION FRAMEWORK

9.1 Tools for Global Optimization

9.1.1 MATLAB

MATLAB is arguably the most well known and function-rich mathematical application development and display environment in use. The MATLAB environment provides a High Order Language for the implementation of algorithms. There are drawbacks, however. While the MATLAB language does have Object Oriented capabilities, they are not sufficient to implement the design patterns of [13]. Multithreading is possible, but is not as convenient or efficient as the mechanisms provided in the more general purpose High Order Languages.

MATLAB does provide a large assortment of highly reliable mathematical and graphics functions, some of which are used to generate the diagrams of this thesis.

9.1.2 Java

The Java High Order Language was chosen for both the capability to implement the design patterns of [13]; in fact, all of those design patterns are implemented in Java in [9]. Because of this and the threading capabilities built into the Java virtual machine, Java was chosen as the implementation language for this work. It should also be noted (as demonstrated by the graphs displayed in this document) that static Java functions can easily be invoked and results imported into the MATLAB environment. If the algorithms are designed such that an array of doubles are returned, after a Java function is called, the array can be manipulated by the MATLAB environment as if the array was declared and filled in the MATLAB environment. Because of these two

capabilities, Java is the language/virtual machine of choice for the implementation of the optimization algorithms.

9.1.3 matlabcontrol

In addition to returning values to the MATLAB environment, a collection of Java functions packaged into an executable jar file are available for use in calling MATLAB functions from Java. matlabcontrol, a package developed for the automatic grading of Java programs, uses the proxy design pattern to provide a mechanism for a Java program to call and retrieve values from MATLAB functions. While the overhead of calling and retrieving values through the proxy probably precludes using matlabcontrol in production code (equivalent Java functions would be found or developed as in the case of the V-Covariance Algorithm), the interface provided by matlabcontrol is a way to get to the rich and reliable functionality of MATLAB functions. Additionally, one can see the progress of an optimization function in real time as diagrams are reconstructed throughout the execution of the algorithm.

One significant result of this research effort was to add entries into the matlabcontrol compatibility table for the version of the operating system and MATLAB version used.

9.1.4 Object Oriented Software Design (OOSD)

Object Oriented Software Design was used to develop the framework and algorithms. Use cases were not provided as the author provided both the requirements and implementation for the algorithms and framework. Use cases are typically used to bridge this gap. A high level UML is used in that not every variable and every method for a class is defined. If this is desired, code is available and there are many automated tools available that can provide this data by reverse engineering.

Rational Rose 98 was used to create the UML diagrams.

9.1.5 Design Patterns

The following design patterns of [13] are used in the Global Optimization framework:

1. *Singleton – Ensure a class has only one instance, and provide a global point of access to it.* Singleton is used wherever only one instance (object) is required. This is the case for MathEngineFacade, the factories that emit the Algorithm and objective function strategy objects. Singleton is also used for the containers that house the priority queues that occur in many of the algorithms. For multithreaded applications singleton typically provides a synchronized rendezvous point for the multiple threads.

2. *Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.* Strategy is used to encapsulate and provide a common (by inheriting from a common base class), the algorithms and objective functions of the application. The Algorithms and Strategy object are emitted from concrete factories. Emitting the objects from concrete factories enable these concrete factories to be incorporated into a Factory Method or Abstract Factory design pattern later if this is desired.

3. *Facade – Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.* The matlabcontrol interface is implemented behind a singleton facade in order to hide the implementation of the MATLAB functions from the application. In the case of the V-covariance algorithm the matrix functions were implemented in the application itself using Java functions. A preferred approach would be to hide the Java imports of these packages behind the MathEngineFacade. However, by implementing them in the application, it is shown that a developer may choose

to use the functions provided by the framework or not – until perhaps such time as they are built into the framework.

4. *Proxy – Provide a surrogate or placeholder for another object to control access to it.* Proxy is mentioned because this design pattern provides the interface to the matlabcontrol functions. Proxy is used because calls to the MATLAB (in the executable jar file jmi.jar delivered with MATLAB) must be done by a different thread than the caller's. MathEngineFacade can be thought of as a layer on top of the MATLAB control proxy.

9.1.6 Integrated Development Environments (IDEs)

Borland jBuilder 2006 is the IDE used initially to develop the framework. However as versions of Java, MATLAB and matlabcontrol evolved, jBuilder 2006 no longer supported Java. The latest version of jBuilder is now an Eclipse IDE plugin. For this work using the jBuilder Eclipse plugin was not significantly better than using Eclipse directly. Therefore, the project was converted to Eclipse and the Eclipse IDE is the IDE used to develop the framework and examples.

9.2 A Framework for Global Optimization Algorithms

TheUML Class diagram for the framework (as it exists at the time of this writing) is shown on Figure 9.1.

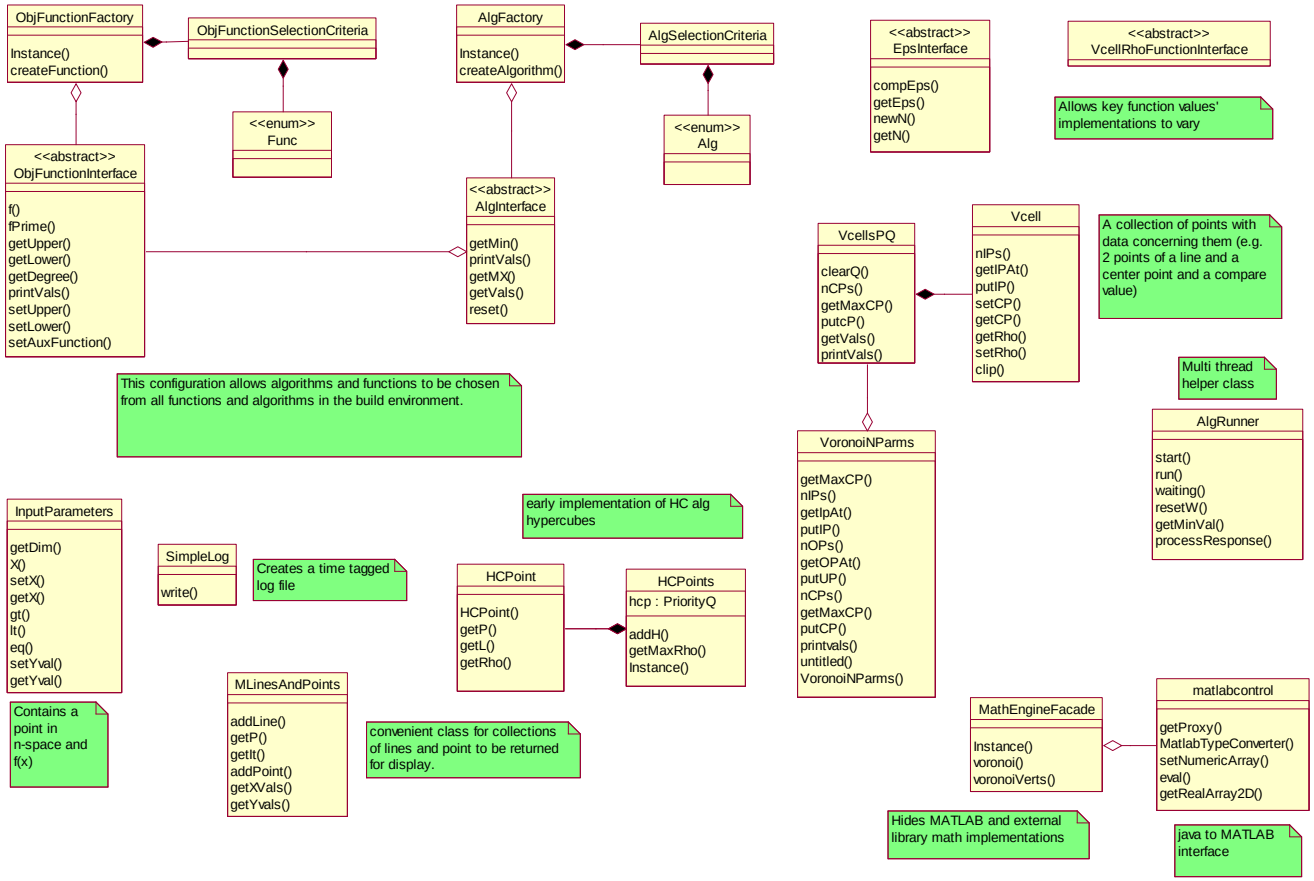

Figure 9.1 Global optimization algorithm framework classes.

9.2.1 Using the Framework to Create an Algorithm

Figure 9.2 shows a sequence diagram for the Global Optimization Framework used to create P-algorithm. The steps are as follows:

Step 1: MINDriver (or MATLAB) invokes a static MIN function to begin

Step 2: MIN builds the ObjFunctionSelectionCriteria for the objective function (in this case, a polynomial function)

Step 3: MIN calls createFunction in the ObjFunctionFactory to create the function Strategy object

Step 4: MIN instantiates an epsilon function

Step 5: MIN builds the AlgSelectionCriteria (including a refernce to the ObjFunction)

Step 6: MIN calls createAlgorithm in the AlgFactory (singleton) to create the algorithm connected to the objective function*

Step 7: MIN calls getMX in the algorithm to determine the minimum

Step 8: P-Alg continues to max γ segment and splits it according to the p-Algorithm to determine the min of the function

Step 9: return M as minimum along with the point where the minimum occurred.

*At this point an object is created that can be executed on this machine or elsewhere.

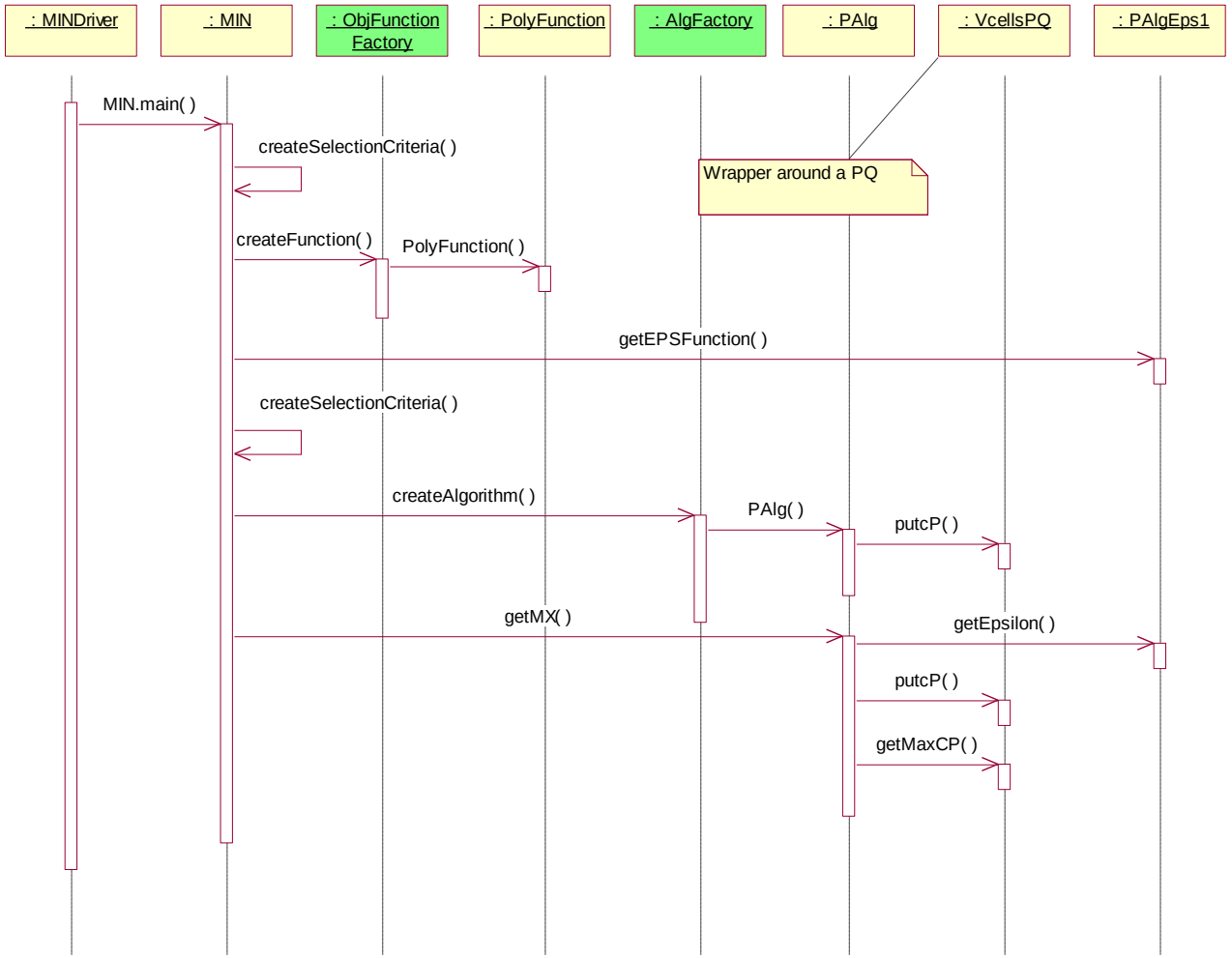

Figure 9.2 Using the framework to create a P-Algorithm.

9.3 Classes of the Framework

A brief description of the classes comprising the Framework for Global Optimization Algorithms.

9.3.1 InputParameters

This class represents a point in n-space. It contains n and a d-sized array of doubles for each coordinate of the point. It also contains utility functions (norm, init, add subtract, etc.) for the points. It contains the y = f(x) value for the point, typically evaluated at the objective function.

9.3.2 Function Creation Classes

These classes are those required for creation of objective functions. The objective function, once created, is emitted from the ObjFunctionFactory as a Strategy object. When algorithms are deployed, it may be advantageous to eliminate the ObjFunctionSelectionCriteria and instead create a separate factory for individual objective functions (e.g., a RastriginFunctionFactory). However, this configuration is preferable for developing and experimenting with global optimization algorithms.

ObjFunctionInterface

ObjFunctionInterface is an abstract class that provides the interface and base class for all objective functions.

ObjFunctionFactory

ObjFunctionFactory is a concrete factory that emits an objective function strategy object of base type ObjFunctionInterface.

ObjFunctionSelectionCriteria

ObjFunctionCriteria contains a Func enumerated type, telling which function is to be created and the input values required to create the function.

Func

An enumerated type that contains one value for each objective function that can be instantiated.

9.3.3 Algorithm Creation Classes

AlgInterface

AlgInterface is an abstract class that provides the interface and base class for all algorithms.

AlgFactory

AlgFactory is a concrete factory that emits an algorithm Strategy object of base type AlgFactoryInterface.

AlgSelectionCriteria

AlgSelectionCriteria contains an Alg enumerated type, telling which algorithm is to be created, and the input values(to include the objective function) required to create the algorithm Strategy object.

Alg

An enumerated type that contains one value for each algorithm that can be instantiated.

9.3.4 Key Function Base Classes

These classes provide interfaces for the important functions that recur in the development of adaptive, stochastic global optimization algorithms.

EpsInterface

This class provides an interface for ϵ_n for the P-Algorithm and others that require an epsilon function that approaches 0 as $n \to +\infty$.

VcellRhoFunctionInterface

This class provides an interface for ρ values used by the HC-Algorithm and others.

9.3.5 Voronoi Diagram Related Classes

These classes are those needed to encapsulate the Voronoi Diagram values and the Priority Queue to return the cell with the maximum rho value.

Vcell

Vcell contains the data pertaining to a Voronoi cell. This includes the points that are the vertices of the voronoi cell along with the rho value.

VcellsPQ

This call encapsulates the priority queue of Vcells.

Voronoi(N)Parms

This class is a container for a Voronoi diagram. It consists of a collection of all cells (returned for the MathEngineFacade).

9.3.6 Hypercube Algorithm Classes

These classes pertain to the HC-Algorithm.

HCPoint

Contains a point that identifies a corner of the hypercube. It also contains the ρ value and barycenter of the hypercube.

HCPoints

Contains the priority queue for the collection of HC points comprising the domain.

9.3.7 Multithreading Class

AlgRunner

AlgRunner contains the functionality to allow algorithms to be multithreaded, provides a synchronized point where threads can report results, and is where the minimum value is tested for and found.

9.3.8 MATLAB Return Values and Other Container Classes

These classes are provided for use in storing and displaying values to the developer.

MLinesAndPoints

This class provides a convenient place to store points and lines to be returned to the MATLAB environment at program completion.

simplelog

This class writes intermediate output results to a log file along with a time stamp.

REFERENCES

[1] *Handbook of Discrete and Computational Geometry, Second Edition (Discrete Mathematics and its Applications)*. Chapman and Hall/CRC, 2 edition, April 2004.

[2] R.C. Buck. *Advanced calculus*. McGraw-Hill, New York,, 1956.

[3] Adam D. Bull. Convergence rates of efficient global optimization algorithms. *Journal of Machine Learning Research*, 12:2879, 2011.

[4] Chen Y. Zilinskas A. Calvin, J.M. An adaptive univariate global optimization algorithm and its convergence rate for twice continuously differentiable functions. *Journal of Optimization Theory and Applications*, 155:628–636, 2010.

[5] J. Calvin and A. Zilinskas. On the convergence of the p-algorithm for one-dimensional global optimization of smooth functions. *Journal of Optimization Theory and Applications*, 102:479–495, 1999. 10.1023/A:1022677121193.

[6] J. Calvin and A. Zilinskas. One-dimensional p-algorithm with convergence rate $o(n^{-3+\delta})$ for smooth functions. *Journal of Optimization Theory and Applications*, 106:297–307, 2000. 10.1023/A:1004699313526.

[7] J.M. Calvin and A. Zilinskas. On a global algorithm for bivariate smooth functions. *Journal of Optimization Theory and Applications*, 2003. To appear.

[8] H.E. Romejin C.G Boender. Stochastic methods. In *Handbook of Global Optimization*, volume 2, pages 829–869. Kluwer Academic Publishers, New York, 1995.

[9] J.W. Cooper. *Java Design Patterns: A Tutorial*. Addison-Wesley, Reading, MA, 2000.

[10] Serhrouchni A. Fayçal M. Cap: a context-aware peer-to-peer system. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'07, pages 950–959, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Scott K. Fowler M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[12] Günther O. Gaede V. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[13] Johnson R. Vlissides J. Gamma E., Helm R. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[14] Tamassia R. Goodrich M. T. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2009.

[15] Neumaier A. Huyer W. Global optimization by multi-level coordinate search. *Journal of Global Optimization*, 14:331–355, 1999.

[16] Brady M. Smith S. Jenkinson M., Bannister P. Improved optimization for the robust and accurate linear registration and motion correction of brain images. *NeuroImage*, 17(2):825–841, October 2002.

[17] Perttunen C.D. Stuckman C.D. Jones, D.R. Lipschitizian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79:157–181, 1993.

[18] Solomatine D.P. Maskey S., Jonoski A. Groundwater remediation strategy using global optimization algorithms. *Journal of Water Resources Planning and Management*, 128:431–440, 2002.

[19] H. Neiderreiter. Random number generation and quasi-monte carlo methods. *Society for Industrial and Applied Mathematics*, 1992.

[20] S. Waldron. Sharp error estimates for multivariate positive linear operators which reproduce the linear polynomials. *In: C.K. Chui, L.L. Schumaker(eds.) Approximation Theory IX*, 1:339–346, 1993.

[21] A. A. Zhigljavsky. *Theory of Global Random Search*, volume 65 of *Mathematics and its Applications (Soviet Series)*. Kluwer Academic Publishers Group, Dordrecht, 1991.