Novel Approaches to Creating Robust Globally Convergent Algorithms for Numerical Optimization

by

Joel David Hewlett

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 14, 2010

Approved by:

Bogdan M. Wilamowski, Chair, Associate Professor of Electrical Engineering
Thaddeus Roppel, Associate Professor of Electrical Engineering
Robert Dean, Assistant Professor of Electrical Engineering
Vitaly Vodyanoy, Professor of Veterinary Medicine

Abstract

Two optimization algorithms are presented, each of which seeks to effectively combine the desirable characteristics of gradient descent and evolutionary computation into a single robust algorithm. The first method, termed Quasi-Gradient Directed Migration (QGDM), is based on a quasi-gradient approach which utilizes the directed migration of a bounded set of randomly distributed points. The algorithm progresses by iteratively updating the center location and radius of the given population. The second method, while similar in spirit, takes this concept one step further by using a "variable scale gradient approximation" (VSGA), which allows it to recognize surface behavior on both the large and small scale. By adjusting the population radius between iterations, the algorithm is able to escape local minima by shifting its focus onto global trends rather than local behavior. Both algorithms are compared experimentally with existing methods and are shown to be competitive, if not superior, in each of the tested cases.

Acknowledgments

I would like to thank my parents, Bud and Patsy Hewlett. Without their love and support, I would have likely never begun this endeavor. I would also like to express my deepest gratitude to my advisor, Dr. Wilamowski. His guidance and teaching have been an invaluable resource in my academic development, without which I would have likely never finished.

Table of Contents

List of Figures

Chapter 1

Introduction

The use of optimization algorithms for engineering applications has become quite common in recent years. In fact, the impact of such methods can be seen in nearly every branch of the discipline, and it is not hard to understand why. In a field driven by progress and innovation, optimization algorithms offer a powerful means to that end. While the analytical approach is generally preferred, the fact remains that for many classes of complex problems, the mathematical tools needed to solve them analytically are yet to be developed. In the meantime, optimization has become an attractive alternative which effectively spans the gap between theory and innovation, making it a staple in the modern practice of engineering.

As of late, a rift has begun to develop within the field of optimization, giving rise to two very distinct schools of thought. The net effect of this split is that the large majority of optimization algorithms now come in one of two flavors, generally referred to as gradient descent and evolutionary computation. To put it plainly, a method is either fast, or it is powerful. Although this distinction is not necessarily valid for every case, it is a widely accepted generalization nonetheless. Each method has its own niche, and comparison of any two is highly subjective on the basis of application. Still, the use of optimization algorithms continues to see rapid growth in a number of diverse fields ranging from robotics [6] and computational intelligence [7][8][9][10] to wireless transmission [11][12] and digital filter design [13][14]. With this continued increase in interest and application, the demand for a newer and more versatile approach has become evident.

Still, despite the abundance of literature and attention garnered by the field in recent years, the discipline appears to have reached an impasse. This is due in large part to the fact that while a number of algorithms have been devised, the middle ground remains largely

unexplored. That is to say that the polarization of the field with respect to gradient and evolutionary methods has resulted in very little effort spent in combining the two. While some attempts have been made at bringing the two sides together [15][16][17], most have experienced rather limited success. It is the purpose of this work to help further this cause, with the ultimate goal being a single robust algorithm which encompasses the strengths of both methods, making itself useful over a wider range of problems.

1.1 Thesis Outline

An overview of numerical optimization is offered in Chapter 2. This includes a light review of some relevant concepts and terminology, as well as a brief look at the workings of some commonly used algorithms. Two novel algorithms are introduced in Chapters 3 and 4, which comprise the body of the work. The first algorithm, presented in Chapter 3, seeks to combine some of the strengths of the gradient and evolutionary approaches in a simple but effective manner. The details of the algorithm are presented along with some experimental data. A comparison of the algorithm's performance with one of the most popular methods in current use is also included. In Chapter 4, the details of the second algorithm are discussed. Though similar in principle to the first, the second algorithm offers an even more powerful and robust alternative. The algorithm is compared with a number of competing methods, and is shown to have equal if not superior performance over a varied assortment of test cases. Finally, the thesis concludes with some closing thoughts in Chapter 5.

Chapter 2

Numerical Optimization

Engineering is the practice of analyzing, designing, and manipulating physical systems. This is achieved by way of detailed mathematical models, which are used to approximate the behavior of their physical counterparts to the most accurate degree possible. In most cases these models can be obtained analytically, however, occasionally there arise situations in which certain attributes of a physical system are hidden, and cannot be directly measured or observed. Consequently, the resulting lack of necessary information prohibits the use of a purely analytical solution. Instead, the modeling of such a system requires a more heuristic approach, which is the realm of numerical optimization.

In the most general sense, numerical optimization is the process of manipulating the variables of a given function in order to achieve an optimal desired output. The ultimate goal is to obtain a unique set of values which produce the best possible output with respect to the predetermined ideal. In the case of modeling, it is the errors, or residuals, between the measured outputs of the physical system and its corresponding model that are of interest. Here, the objective is to minimize the mean square value of the error, which is a function of the model coefficients. Doing so yields a unique set of coefficients which offer the most accurate representation of the model's physical counterpart. Thus numerical optimization offers a powerful heuristic alternative for acquiring the desired set of coefficients, even when an analytical solution is unattainable.

It is important to note that similar heuristic methods are used to solve problems in nearly every discipline imaginable. Therefore, while system identification and modeling may represent the predominant applications of optimization within the field of engineering, similar processes are utilized in nearly every facet of applied science. In fact, the average person unknowingly employs heuristic problem solving methods on an almost day-to-day

basis; a fact which emphasizes the power of such methods for overcoming situations in which detailed knowledge of a specific system may be lacking.

A familiar example of this is the 3-band equalizer on a home stereo system. The system has 3 inputs, corresponding to the sliders or knobs which control the individual bands, and 1 output in the form of the sound being produced by the speakers. What is interesting about this example is that despite having no practical understanding of the internal workings of the system, the average person is nonetheless capable of achieving near optimal sound quality by ear alone. What may only appear to be a simple attempt at achieving the most faithful reproduction of one's favorite record, is in reality the heuristic process of solving a 3-dimensional optimization problem. And it is this same powerful ability to essentially do more with less that makes the study of optimization so attractive.

Although optimization algorithms are, by definition, processes of trial and error, they should not be confused with simple brute-force methods which employ primitive techniques such as blind or random searches. On the contrary, the study and practice of numerical optimization rests quite heavily on a thorough and rigorous mathematical foundation. Naturally, to cover the subject in it's entirety would far exceed the practical scope of this thesis. Thus the following chapter is intended only as a supplement to the material presented later, and is by no means a comprehensive introduction to the subject of optimization. In keeping with this objective, only the most fundamental concepts and terminology are covered. However, should a more comprehensive treatment be desired, the reader is directed to [24], which represents an excellent technical reference on the subject of optimization.

2.1 Mathematical Formulation of Optimization Problems

Mathematically speaking, optimization is the process of searching for an extremum x^* of some function $f(x)$. Ideally, given some initial value x_0, the algorithm will converge to the function's global minimizer or maximizer, depending on the desired case (for simplicity's sake, the term optimization will be considered synonymous with minimization[1] from here

[1]Note that maximizing a function f is equivalent to minimizing $-f$

on). The general mathematical formulation for an optimization problem can be expressed as

$$\min_x f(x) \tag{2.1}$$

where $x \in \mathbf{R}^n$ with $n \geq 1$. The function $f : \mathbf{R}^n \to \mathbf{R}$ is commonly known as the *objective function*, or simply the *objective*.

2.2 The Anatomy of an Algorithm

An optimization algorithm, as the name implies, is an iterative process which seeks to minimize some objective $f(x)$. At each iteration k, the goal is to generate an *iterate*, x_{k+1}, such that $f(x_{k+1}) < f(x_k)$, with the expectation that as k increases, $x_k \to x^*$. While there are numerous methods by which these iterates may be generated, most numerical methods can be expressed in the form

$$x_{k+1} = x_k + \sigma_k \tag{2.2}$$

where $\sigma_k \in \mathbf{R}^n$ is known as the *step*. Looking at (2.2), it is clear that σ_k is responsible for determining how the algorithm progresses from one iterate to the next, which ultimately defines the algorithm itself. Thus it is the way in which this step is generated that separates one algorithm from another.

The step σ is nothing more than a vector extending from one iterate to the next, which can be uniquely expressed by its length and direction. Thus, in a sense, the generation of the step can be seen as the two fold process of a) determining the best direction in which to proceed and b) deciding how far to travel in the direction chosen. In general, so long as σ faces "downhill," and the distance traveled is sufficiently short, there are an infinite number of possible combinations which will result in convergence. However, the rate of convergence depends quite heavily on the length and direction of the step. Thus convergence alone is not sufficient. A well designed algorithm should not only guarantee[2] local convergence, but it

[2]This guarantee does not necessarily apply to cases in which the objective function is non-smooth, and therefore not continuously differentiable.

should do so in as efficient and effective a manner possible. Therefore, the choice of search direction and step length are of utmost importance to an algorithm's success.

2.2.1 Common Search Directions

The most obvious of all search direction is the gradient direction $-\nabla f_k$, also known as the direction of *steepest descent*. As the name implies, $-\nabla f_k$ represents the direction in which the objective function $f(\mathbf{x})$ decreases most rapidly, making it a natural choice. Though sometimes referred to as the gradient direction, the steepest descent direction actually points in the opposite direction of the true gradient, which corresponds to the direction of greatest *increase*. The true *gradient* is, by definition, the vector of all partial first derivatives of the objective function, or more formally,

$$\nabla f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{bmatrix}, \text{ where } \mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{2.3}$$

In optimization, where the goal is to find the point at which a given function yields the lowest possible value, knowing the direction in which that function changes most rapidly is of obvious value, making the gradient a particularly useful tool.

Another commonly used search direction is the *Newton direction*. Though it is more difficult to compute than the gradient, it provides significantly more information about the local behavior of the objective, which in turn yields a much higher rate of convergence. This is achieved through the use of a second-order model $m_k(\sigma_k)$ of the objective function, derived from its Taylor series approximation

$$f(\mathbf{x}_k + \sigma_k) \approx f_k + \sigma_k^T \nabla f_k + \frac{1}{2}\sigma_k^T \nabla^2 f_k \sigma_k \triangleq m_k(\sigma_k) \tag{2.4}$$

From this, the Newton direction is defined as the vector σ_k which minimizes the quadratic model $m_k(\sigma_k)$. Assuming for now that $\nabla^2 f_k$ is positive definite, it is possible to solve for σ_k

by setting (2.4) equal to zero. Doing so yields the following explicit formula for the Newton direction:

$$\sigma_k = -(\nabla^2 f_k)^{-1} \nabla f_k. \tag{2.5}$$

Whereas the steepest decent direction revolves around the computation of the gradient, the Newton direction relies primarily on the calculation of the *Hessian* $\nabla^2 f_k$, which is a matrix containing all second partial derivatives of the objective function. That is,

$$\nabla^2 f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\delta^2 f}{\delta x_1^2} & \frac{\delta^2 f}{\delta x_1 \delta x_2} & \cdots & \frac{\delta^2 f}{\delta x_1 \delta x_n} \\ \frac{\delta^2 f}{\delta x_2 \delta x_1} & \frac{\delta^2 f}{\delta x_2^2} & \cdots & \frac{\delta^2 f}{\delta x_2 \delta x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta^2 f}{\delta x_n \delta x_1} & \frac{\delta^2 f}{\delta x_n \delta x_2} & \cdots & \frac{\delta^2 f}{\delta x_n^2} \end{bmatrix}, \text{ where } \mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{2.6}$$

As alluded to earlier, the computation of the Hessian can be a rather expensive operation, which is one of the Newton method's major drawbacks. Still, the benefits of this approach are typically considered to outweigh the costs.

In addition, a number of strategies have been devised which serve to reduce the computational requirement by replacing the true Hessian matrix with a close approximation that does not need to be fully reevaluated at each iteration. Search directions which operate in this way are commonly referred to as *quasi-Newton* methods. Some of the more common implementations include the Broyden, Fletcher, Goldfarb and Shanno (BFGS) algorithm, and the closely related Davidon, Fletcher and Powell (DFP) algorithm.

Another attractive attribute of many quasi-Newton methods is that they can be reformulated to operate on the inverse of the approximated Hessian instead of on the approximation itself, which alleviates the burden of performing costly matrix inversions when solving (2.5). This ability can prove especially valuable on problems with high dimensionality.

2.2.2 Choosing A Step Length

While the Newton and steepest decent directions can both be shown to guarantee a reduction in the value of the objective[24], this is only true for sufficiently small step lengths. This is due to the fact that the reliability of the corresponding linear and quadratic models diminishes as a function of distance. While there are a number of ways of selecting proper step lengths, in general, most methods follow one of two basic strategies.

Line Search Strategies

The first of the two strategies is known as the *line search* method. In this approach, the search direction is chosen first, which yields a one-dimensional sub-problem in which the objective is treated as a function of the step length, which is equivalent to solving

$$\alpha_k = \min_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{d}_k), \quad \alpha > 0 \tag{2.7}$$

where d_k is the search direction and α is a scalar which controls the length of the step. In practice, it is not practical to provide an exact solution to (2.7), and in fact, an approximate solution is generally sufficient for convergence. The most common approach is to generate a finite set of trial steps and choose the value α_k which yields the greatest reduction in the objective $f(\mathbf{x})$. The generation of the trial steps can be a rather complex process in and of itself, and is often subject to one or more sufficient decrease conditions which are used to determine if a given step produces an adequate improvement in the objective.

Trust Region Strategies

The second major strategy for handling step length is know as the *trust region* approach. For this approach, the step is chosen as the solution to a constrained sub-problem in which the objective function $f(\mathbf{x})$ is replaced by some approximate model $m_k(\mathbf{x})$. The resulting

sub-problem is written

$$\sigma_k = \min_{\sigma} \; m_k(\mathbf{x}_k + \sigma), \text{ where } \mathbf{x}_k + \sigma \text{ lies within the trust region.} \tag{2.8}$$

Usually, the trust region is spherical in shape, which requires the chosen step to satisfy the constraint $||\sigma|| \leq r_k$, where r_k is the radius of the constraining region. If the solution to the constrained sub-problem does not produce sufficient improvement in the objective, the trust region is adjusted accordingly and the process is repeated.

Typically, a quadratic model is chosen for $m_k(\mathbf{x})$, making most trust region strategies Newton or quasi-Newton in nature. Therefore, in most cases, m_k is of the form

$$m_k(\mathbf{x}_k + \sigma) = f_k + \sigma^T \nabla f_k + \frac{1}{2}\sigma^T B_k \sigma \tag{2.9}$$

where B_k is either the true Hessian or some approximation to it.

Combining (2.8) and (2.9), while also introducing the constraint imposed by the spherical trust region, results in the following reformulation of the trust region sub-problem,

$$\sigma_k = \min_{\sigma} \; f_k + \sigma^T \nabla f_k + \frac{1}{2}\sigma^T B_k \sigma, \quad \text{s.t. } ||\sigma|| \leq r_k \tag{2.10}$$

It may be shown [24] that the solution σ_k^* to the constrained sub-problem in (2.10) must satisfy

$$(B_k + \lambda I)\sigma_k^* = -\nabla f_k \tag{2.11}$$

for some scalar $\lambda \geq 0$. This characteristic is particularly useful since it implies

$$\sigma_k^*(\lambda) = -(B_k + \lambda I)^{-1} \nabla f_k \tag{2.12}$$

which reduces σ to one-dimensional function of λ. As with α for line search methods, it is not practical to seek an exact value for λ. Instead, an approximate value of lambda is obtained in much the same way as the step length α is chosen in a line search. An interesting

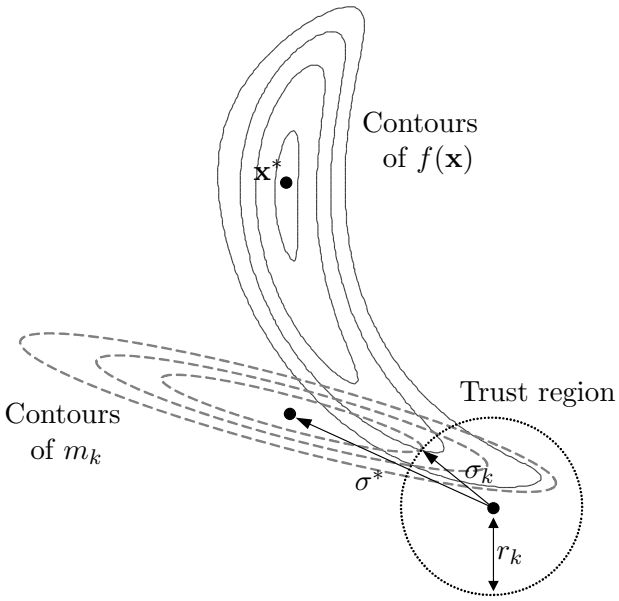

Figure 2.1: Comparison of the trust-region and Newton steps

characteristic of the trust region approach is the fact that the choice of the trust region radius effects not only the step length, but the direction of the search as well. An example of this can be seen in Fig. 2.1. Here, the vector σ^* represents the Newton step, whereas σ_k represents the constrained trust-region step. Notice that as the radius of the trust region decreases, the direction of the step converges toward the gradient direction. Therefore, the trust-region step is essentially a linear combination of the gradient and Newton directions.

Chapter 3

Quasi-gradient Optimization Using Directed Migration

Gradient based search methods are known to be very efficient, especially for cases in which the surface of the solution space is relatively smooth, with few local minima. Unfortunately, in cases where the search space lacks this relative uniformity, gradient methods become easily trapped in the local minima commonly found on rough or complex surfaces. Methods of evolutionary computation (including genetic algorithms), on the other hand, are much more effective for traversing these more complex surfaces, and are inherently better suited for avoiding local minima. However, like the gradient based approach, evolutionary computation has its own significant weakness. This stems from the fact that despite its reliability, solutions are often not optimal. Furthermore, both methods are known to converge very slowly [2] [3][5].

The objective behind this chapter is to take advantage of both methods by combining the desirable characteristics of each. Unlike standard evolutionary computation, populations are generated using the gradient, which is not directly calculated but is instead extracted from the properties of the existing population. Several similar approaches have been undertaken along this path [1] [16][17][2][3][4], but the method which is proposed here has less computational complexity and is more suitable for on-line hardware implementation. Simple computations are repeated with every iteration, and the gradient is updated simply by subtracting the coordinates of the best solutions from the current and previous populations. The steepest decent method, which is the most commonly used gradient method, tends to approach the solution asymptotically, which results in a much slower rate of convergence. By comparison, the proposed method converges much more rapidly.

3.1 Proposed Method

The proposed method is a hybrid algorithm which offers both the relative speed of gradient descent and the methodical power of evolutionary computation. Like the latter, this hybrid algorithm relies on a randomly generated population of initial points. It also shares the advantage of being an exclusively feed-forward process. What separates this approach from standard methods of evolutionary computation is the way in which the successive populations are generated. This is where the proposed method borrows more from the gradient based approach. The hybrid approach relies on the calculation of a "rough" gradient using the individual errors associated with a given population. The minimum of these errors is determined and an entirely new population is generated about the corresponding point. This offers a more directional approach to the generation of successive populations. Thus this approach can be viewed as a sort of migration rather than recombination, for which it is termed Quasi-Gradient Directed Migration (QGDM). The result is an algorithm which converges much more rapidly than the combinational approach commonly associated with genetic algorithms, while at the same time reducing the risks presented by local minima.

3.1.1 Detailed Description

The QGDM method is best represented as a four step process. Although the algorithm technically involves only three steps per iteration, an additional step is required to begin the process.

Step 1: **Choosing a Starting Point and Radius** - Let f be a function defined in $\mathbf{R}^n$. Choose an initial center point c and radius r. An initial minimum m must also be selected. For the first iteration, let $m = c$.

Step 2: **Generating a Random Population** - With n being the number of points per population, define a set of vectors $V = \{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$ such that

$$\mathbf{v}_{ik} = r_k X\mathbf{Y} + c_k \; for \; i = 1...n \tag{3.1}$$

where X is a random number ranging from 0 to 1, $\mathbf{Y}$ is a normalized random vector, and k is the current iteration. Using (3.1) creates a set of random vectors whose members are centered about c_k with magnitudes ranging from 0 to r_k. Therefore $\mathbf{v}_{ik}$ represent positional vectors of the points p_{ik} which lie within the defined region.

Step 3: **Approximation of the Gradient** - Evaluate $f(p_{ik})$ for $i = 1...n$. If $\forall i(\min(f(p_{ik})) > f(m_k))$, repeat step two. Otherwise, let m_{k+1} be p for which $f(p) = \min(f(p_{ik}))$. An approximate gradient vector $\mathbf{g}_k$ can then be defined by taking the difference between m_{k+1} and m_k.

$$\mathbf{g}_k = m_{k+1} - m_k$$

Step 4: **Creating a New Center and Radius** - The new center should be shifted slightly so that the following population will lie in the in the general direction of the approximated gradient. Using

$$c_{k+1} = \alpha\mathbf{g} + m_k \; for \; \alpha \geq 1$$

allows the size of the shift to be controlled by α. If no shift is desired, α can be set to 1, in which case $c_{k+1} = m_{k+1}$. In order to ensure that m_{k+1} lies within the new region, it is necessary that $r_{k+1} \geq \|c_{k+1} - m_{k+1}\|$. This can be set using

$$r_{k+1} = \beta\|c_{k+1} - m_{k+1}\| \; for \; \beta \geq 1$$

Once r_{k+1} and c_{k+1} are determined, steps two through four are repeated until $f(m)$ is within the desired tolerance.

The two dimensional example case in fig. 1 illustrates the process through four iterations. For this particular example, $\alpha = 2$ and $\beta = 3/2$. This can be clearly seen by the way r_k, represented by the large circles, perfectly bisects $\mathbf{g}_{k-1}$ for each iteration. Also, notice that for the first iteration, the gradient vector $\mathbf{g}$ extends from c to m, whereas in all subsequent iterations $\mathbf{g}$ is defined from m_k to m_{k+1}.

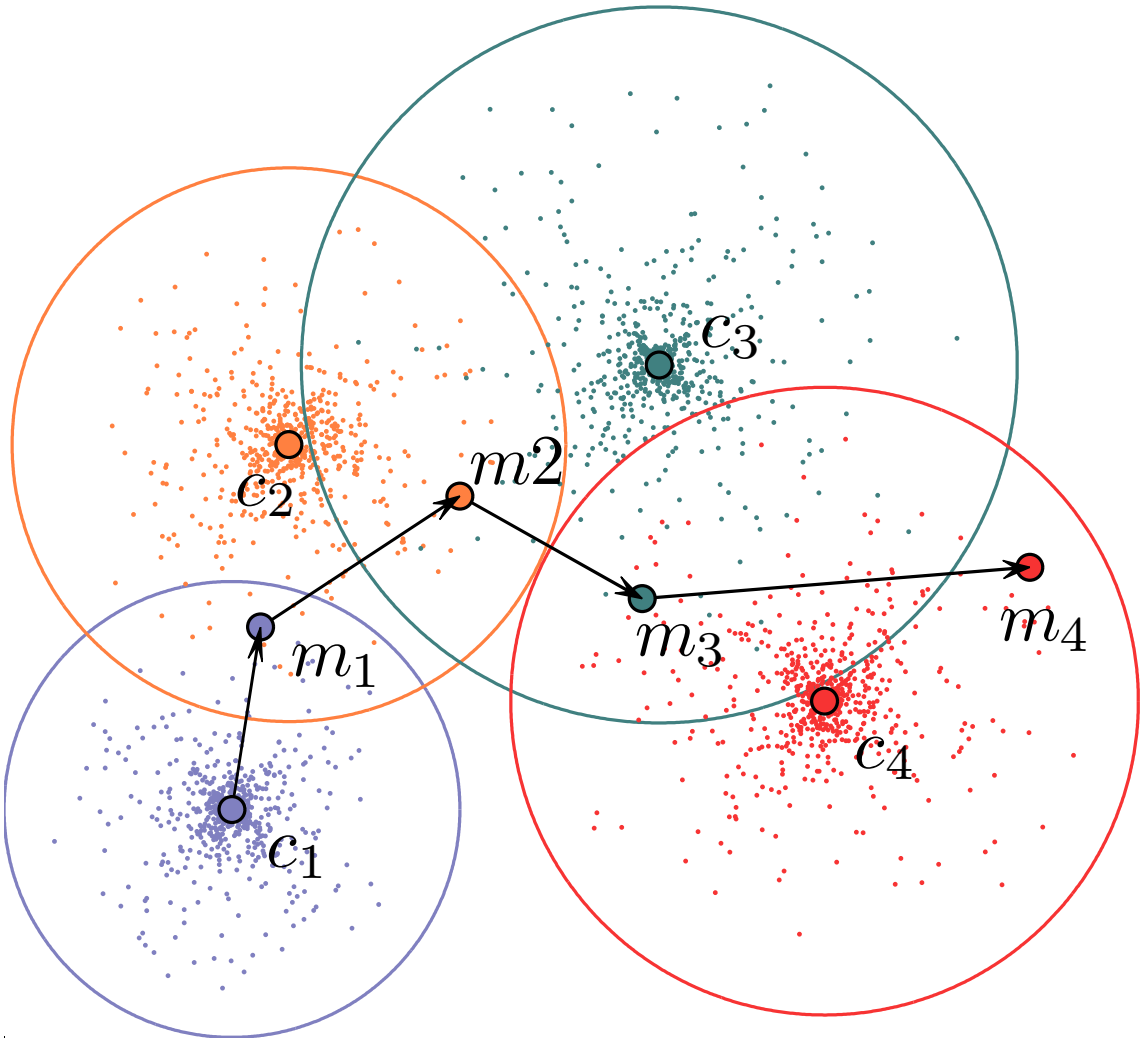

Figure 3.1: Four iterations in two dimensions

3.2 Possible Modifications

Spherically shaped regions may not be optimal when generating successive populations. Using conical regions which extend in the direction of the approximate gradient might greatly increase the speed of convergence. The following is just one of many similar modifications.

3.2.1 Conically Shaped Regions

It is desired that the vector $\mathbf{v}$ be used as the axis of symmetry. This is best done by switching to polar coordinates which only require the zenith angle ϕ and radial component ρ in order to define the region's orientation. This is an especially useful way of describing a hyperconical region since it is valid for any subspace of $\mathbf{R}^n$. The drawback of this method is that it requires that the axis of symmetry lie along the zenith axis. In order to extend this method to cases using arbitrary axes of symmetry, it is necessary to change to a basis U whose zenith axis is in the direction of $\mathbf{v}$. The region can then be defined using the new set of coordinate axes. With this, representing a point or vector within the defined region in terms of the standard basis E can be done quite easily by changing bases from U back to E using a simple transition matrix.

Defining a new basis

The process described below first requires the formation of a new basis using the n-dimensional axial vector $\mathbf{v}$. To do this, an additional $n-1$ linearly independent vectors are required. The first step is to devise a generalized method for generating this set of vectors.

1. **Generating an ordinary basis**

   In order to generate an ordinary basis, it is first necessary to acquire the proper tools for determining linear independence. The following three theorems are particularly useful.

Theorem 3.1 *Let* $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ *be* n *vectors in* $\mathbf{R}^n$ *and let*

$$\mathbf{x}_i = (x_{1i}, x_{2i}, ..., x_{ni})^T$$

for $i = 1, ..., n$. *If* $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$, *then the vectors* $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ *will be linearly independent if and only if* $\mathbf{X}$ *is singular.*

Theorem 3.2 *An* $n \times n$ *matrix* $\mathbf{A}$ *is singular if and only if*

$$\det(\mathbf{A}) = 0$$

Theorem 3.3 *Let* $\mathbf{A}$ *be an* $n \times n$ *matrix.*

(i) *If* $\mathbf{A}$ *has a row or column consisting entirely of zeros, then* $\det(\mathbf{A}) = 0$.

(ii) *If* $\mathbf{A}$ *has two identical rows or two identical columns, then* $\det(\mathbf{A}) = 0$.

Combining these three theorems yields a particularly useful result.

Corollary 3.4 *Let* A *be an* $n \times n$ *matrix. The column space of* A *forms an ordinary basis for* $\mathbf{R}^n$ *if and only if*

(i) A *contains no rows or columns consisting entirely of zeros.*

(ii) *No two rows or columns of* A *are identical.*

The conditions from Corollary 3.4 can nearly always be satisfied by taking the standard basis E and simply replacing $\mathbf{e}_1$ with $\mathbf{v}$. The only case in which this will not result in an ordinary basis of $\mathbf{R}^n$ is when $v_1 = 0$, which can be easily remedied by replacing e_{i1} with $(v_1 + 1)$ for $i = 2, ..., n$. This method will work for any $\mathbf{v}$ except $\mathbf{0}$. To summarize,

(a) Let $E = \{\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n\}$. Replace $\mathbf{e}_1$ with $\mathbf{v}$.

(b) Replace e_{i1} with $(v_1 + 1)$ for $i = 2, ..., n$.

The resulting set of vectors (S) is an ordinary basis for $\mathbf{R}^n$ with $\mathbf{s}_1 = \mathbf{v}$.

Although this method does produce a basis for $\mathbf{R}^n$ relative to $\mathbf{v}$, it still lacks one essential characteristic. In order for the vector lengths and angles of separation to be preserved when changing bases, S must be orthogonal. Clearly, S is not an orthogonal basis, however this can be remedied using the Gram-Schmidt Orthogonalization Process.

2. **The Gram-Schmidt Process**

The The Gram-Schmidt Process is a method for orthogonalizing a set of vectors in an inner product space, most commonly the Euclidian space $\mathbf{R}^n$. The GramSchmidt process takes a finite, linearly independent set $V = \{v_1, ..., v_n\}$ and generates an orthogonal set $V' = \{u_1, ..., u_n\}$ that spans the same subspace as V.

Theorem 3.5 (The Gram-Schmidt Process) *Let* $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ *be a basis for the inner product space V. Let*

$$\mathbf{u}_1 = \left(\frac{1}{\|\mathbf{x}_1\|}\right)\mathbf{x}_1$$

and define $\mathbf{u_2}, ..., \mathbf{u_n}$ *recursively by*

$$\mathbf{u}_{k+1} = \frac{1}{\|\mathbf{x}_{k+1} - \mathbf{p}_k\|}(\mathbf{x}_{k+1} - \mathbf{p}_k) \quad for \quad k = 1, ..., n-1$$

where

$$\mathbf{p}_k = \langle \mathbf{x}_{k+1}, \mathbf{u}_1 \rangle \mathbf{u}_1 + \langle \mathbf{x}_{k+1}, \mathbf{u}_2 \rangle \mathbf{u}_2 + ... + \langle \mathbf{x}_{k+1}, \mathbf{u}_k \rangle \mathbf{u}_k$$

is the projection of $\mathbf{x}_{k+1}$ *onto Span*$(\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_k)$. *The set*

$$\{\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n\}$$

is an orthonormal basis for V.

By applying Theorem 3.5 to the ordinary basis S, orthogonal basis S' is obtained, which spans the same subspace as S, who's first element $\mathbf{u}_1$ shares the same axis as $\mathbf{s}_1$, and therefore lies in the direction of the initial vector $\mathbf{v}$. Thus the net result is an orthonormal basis S' which allows for orthonormal transformations with the standard basis E.

Defining a Hyperconically bounded region

With the basis S', defining a hyperconically bounded region about $\mathbf{v}$ is a trivial matter. Using hyperspherical coordinates, the region can be described using only the radial component ρ and the zenith angle ϕ. Any point obtained by manipulating the other angular components will lie on the boundary of this region. For a region described by a maximum radial component ρ_{max} and a maximum zenith angle ϕ_{max}, any point satisfying $0 \leq \rho \leq \rho_{max}$ and $0 \leq \phi \leq \phi_{max}$ is guaranteed to lie within the desired region. All that is required now is to perform an orthonormal transformation from S' back to E.

Changing bases from S' to E

Changing bases is a relatively simple matter. A transition matrix from S' to E can be found with relatively little effort. However, before it can be used to for changing the basis of a point within the predefined region, it's coordinates must be converted from a spherical to cartesian representation. Thus some generalized method for making this conversion is needed.

Hyperspherical to cartesian coordinate conversion

A coordinate system has now been defined for n-dimensional Euclidean space, which is analogous to the spherical coordinate system defined for 3-dimensional Euclidean space. These coordinates consist of a radial coordinate ρ, and $n-1$ angular coordinates $\phi_1, \phi_2, ..., \phi_{n-1}$. If x_k are the Cartesian coordinates, then

$$x_k = \begin{cases} \rho \cdot \cos(\phi_k) & for\ k = 1 \\ \rho \cdot \prod_{i=1}^{k-1} \sin(\phi_i) \cdot \cos(\phi_k) & for\ k = 2, ..., n-1 \\ \rho \cdot \prod_{i=1}^{k-1} \sin(\phi_i) & for\ k = n \end{cases} \tag{3.2}$$

This offers a generalized method by which an n-dimensional vector or point in hyperspherical coordinates may be converted into it's cartesian representation. Once the conversion has been made, the given point or vector may be freely converted from one basis to the other using nothing more than a simple transition matrix.

Creating a transition matrix from S' to E

Theorem 3.6 (Transition matrices) *Let $\{\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n\}$ be an ordered basis for $\mathbf{R}^n$, and let $\mathbf{c}$ be a coordinate vector with respect to $\{\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n\}$. To find the corresponding coordinate vector $\mathbf{x}$ with respect to $\{\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n\}$, simply multiply the matrix $U = [\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n]$ by $\mathbf{c}$.*

$$\mathbf{x} = U\mathbf{c}$$

Summary

A step-by-step summary of the combined process.

Step 1: Using $[\mathbf{v}, \mathbf{e}_2, ..., \mathbf{e}_n]$, create a new basis for $\mathbf{R}^n$ by replacing e_{i1} with v_1+1 for $i = 2, ..., n$. Call the resulting matrix S and refer to it's column space as $\{\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\}$.

Step 2: Using $S = \{\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\}$, let

$$\mathbf{u}_1 = \left(\frac{1}{\|\mathbf{s}_1\|}\right)\mathbf{s}_1$$

and define $\mathbf{u_2}, ..., \mathbf{u_n}$ recursively by

$$\mathbf{u}_{k+1} = \frac{1}{\|\mathbf{s}_{k+1} - \mathbf{p}_k\|}(\mathbf{s}_{k+1} - \mathbf{p}_k) \quad for \quad k = 1, ..., n-1$$

The resulting matrix $[\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n]$will form an orthonormal basis for $\mathbf{R}^n$ which is denoted as U.

Step 3: Define a hyperconical region about $\mathbf{u}_1$ by choosing a maximum radial component ρ_{max} and a maximum zenith angle ϕ_{max}.

Step 4: Generate vectors within the defined region by choosing ρ and ϕ such that $0 \leq \rho \leq \rho_{max}$ and $0 \leq \phi \leq \phi_{max}$.

Step 5: Let $\mathbf{x}$ be one of the vectors chosen in step four. Convert $\mathbf{x}$ from hyperspherical to cartesian coordinates using (3.2).

Step 6: Express $\mathbf{x}$ with respect to $[\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n]$ using the U as a transition matrix.

$$\mathbf{x}' = U\mathbf{x}$$

This newly generated vector $\mathbf{x}'$ has two very important characteristics, which make it particularly useful. They are:

(i) The length of $\mathbf{x}'$ is no greater than ρ_{max}

(ii) The angle of separation between $\mathbf{x}'$ and the original vector $\mathbf{v}$ is no greater than ϕ_{max}

Controlling the Defined Region's Shape

It might be desirable to have ρ diminish as ϕ increases. This can be done by expressing ρ as a function of ϕ.

$$\rho(\phi) = \rho_{max}\left[1 - \left(\frac{\phi}{\phi_{max}}\right)^{\beta}\right]$$

The variable β can be adjusted to control the shape of the given region. As β becomes larger, the shape of the region becomes increasingly conical. Fig. 2 offers a graphical representation of how β effects the region's shape.

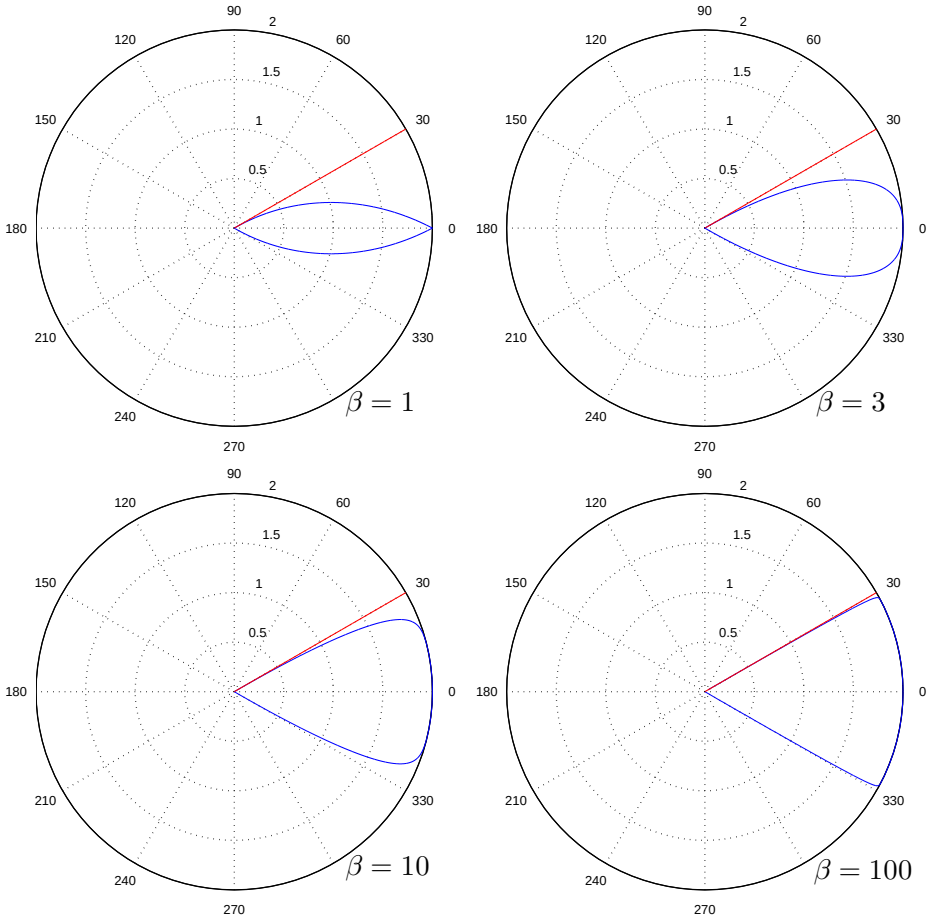

Figure 3.2: Search region shapes with $\rho_{max} = 2,\ \phi_{max} = 30°$

3.2.2 A Modified Population Distribution

One of the dangers of using a uniform population distribution is its susceptibility to local minima. If the algorithm gets into a low-lying area which is larger than the current maximum population radius, the algorithm will be permanently stuck. In order to avoid this scenario, occasional points must be plotted outside the desired maximum radius.

One effective and efficient way of doing this is to use a population density which decreases exponentially as a function of the radius. This can be achieved by replacing $r_k X$ in (3.1) with

$$\frac{1}{X + \frac{1}{r_k}} - \frac{r_k}{r_k - 1} \tag{3.3}$$

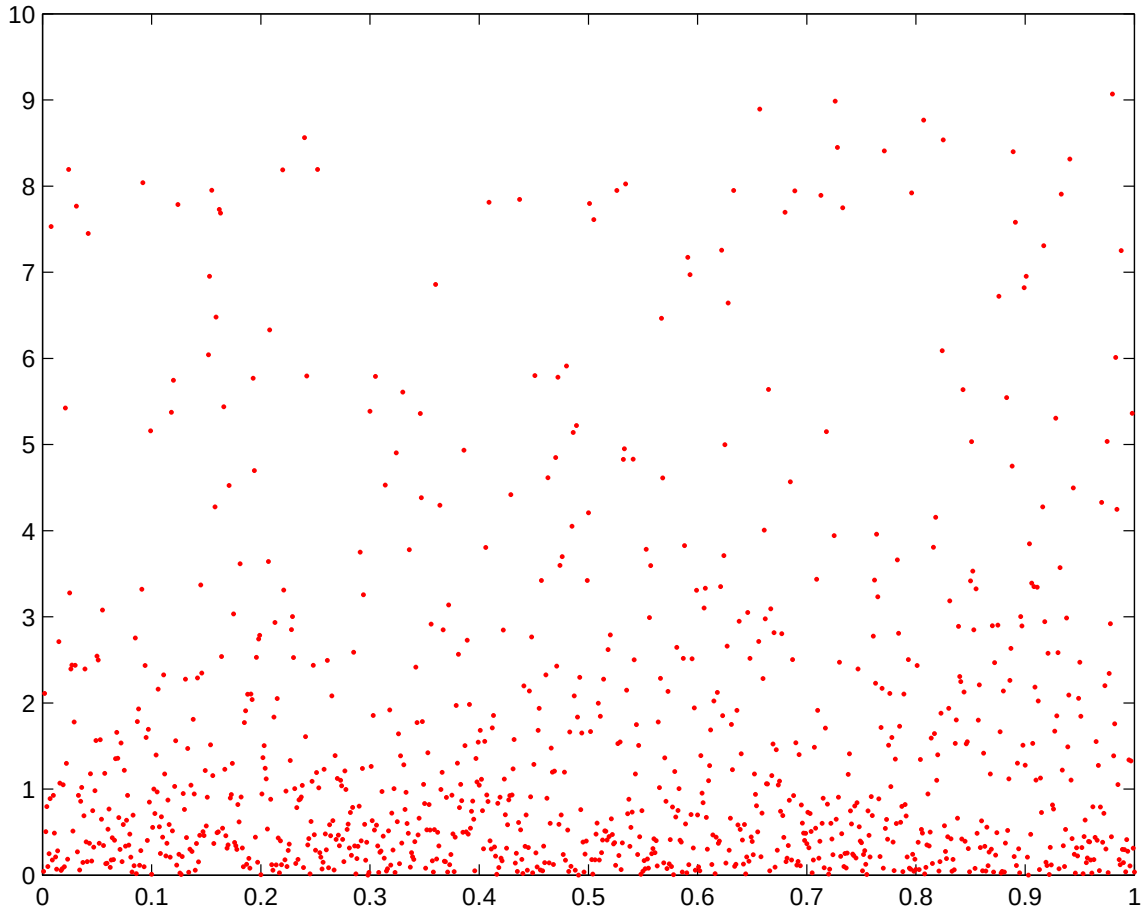

Figure 3.3: A Plot of 1000 random points using (3.3)

which has a range of

$$\left(0, \frac{r_k}{1+\frac{1}{r_k}}\right)$$

Although the upper bound for $\|v_{ik}\|$ is no longer r_k, for larger values of r_k,

$$\frac{r_k}{1+\frac{1}{r_k}} \approx r_k$$

Fig. 3.3 shows how (3.3) effects density.

3.3 Experimental Results

Two variations of the algorithm were used for testing. The first set of tests were performed using the standard algorithm as described in section **??**. The second set used the modified population distribution from section 3.2.2. The algorithms were then tested using population sizes of 5, 10, and 20 points per iteration. Statistical Data was compiled for both algorithms using 100 runs for each population size.

Due to the common practical application of optimization in the field of artificial neural networks, initial tests were made using the algorithm for training a simple two-layer network. The network was then retrained for comparison using error back-propagation, which is the most popular training algorithm in current use. Statistical data and error plots are included for each test. All figures and statistics were taken for successful runs only, with the exception of success rate, which was defined as the ratio of successful runs to non-convergent ones. From the results, it is clear that both algorithms perform comparably to EBP, with the modified version being by far the most consistent.

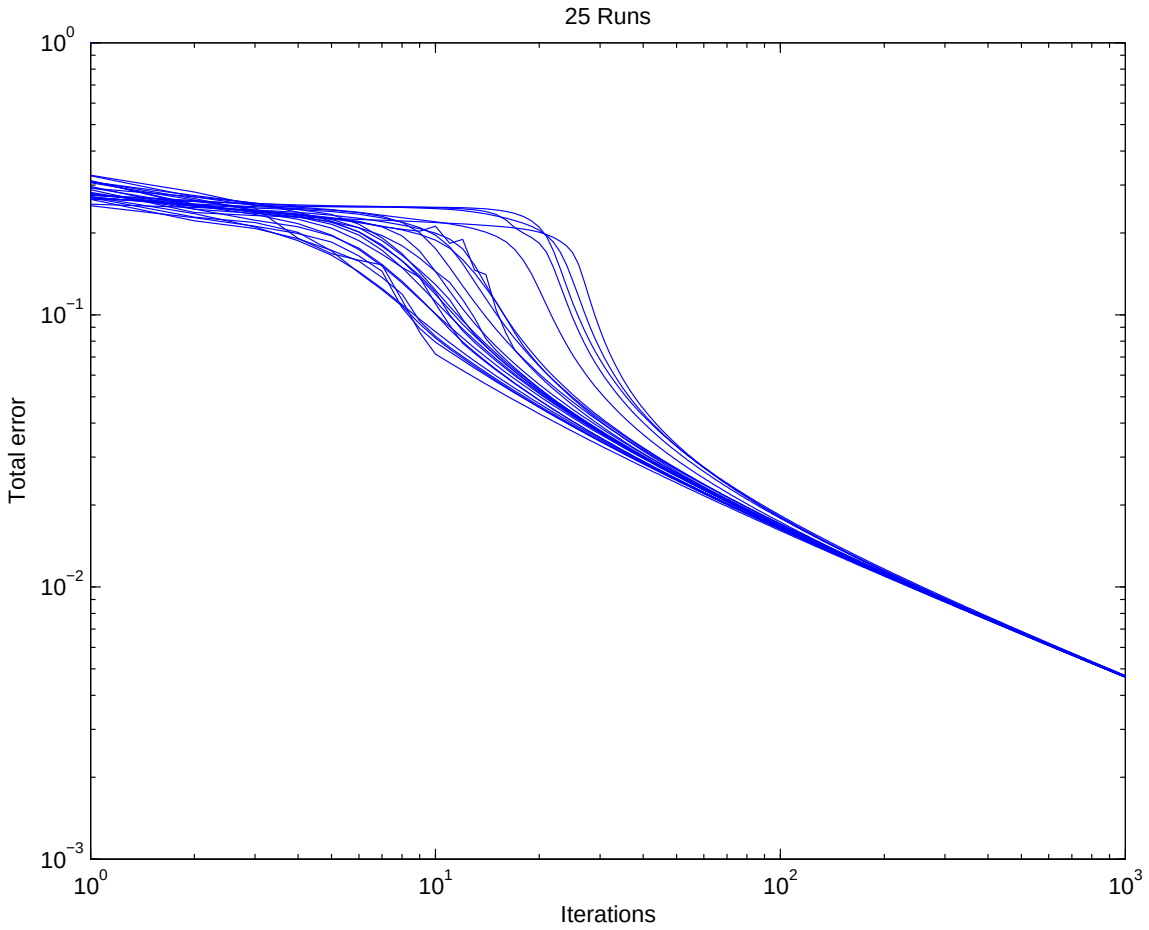

Figure 3.4: Error back propagation using $\alpha = 0.5$

Error back propagation

Population	20	
Gain	5	
Learning constant	0.5	
Min run	138	ite
Max run	259	ite
Ave. run	152	ite
Standard deviation	15	ite
Success rate	66	%

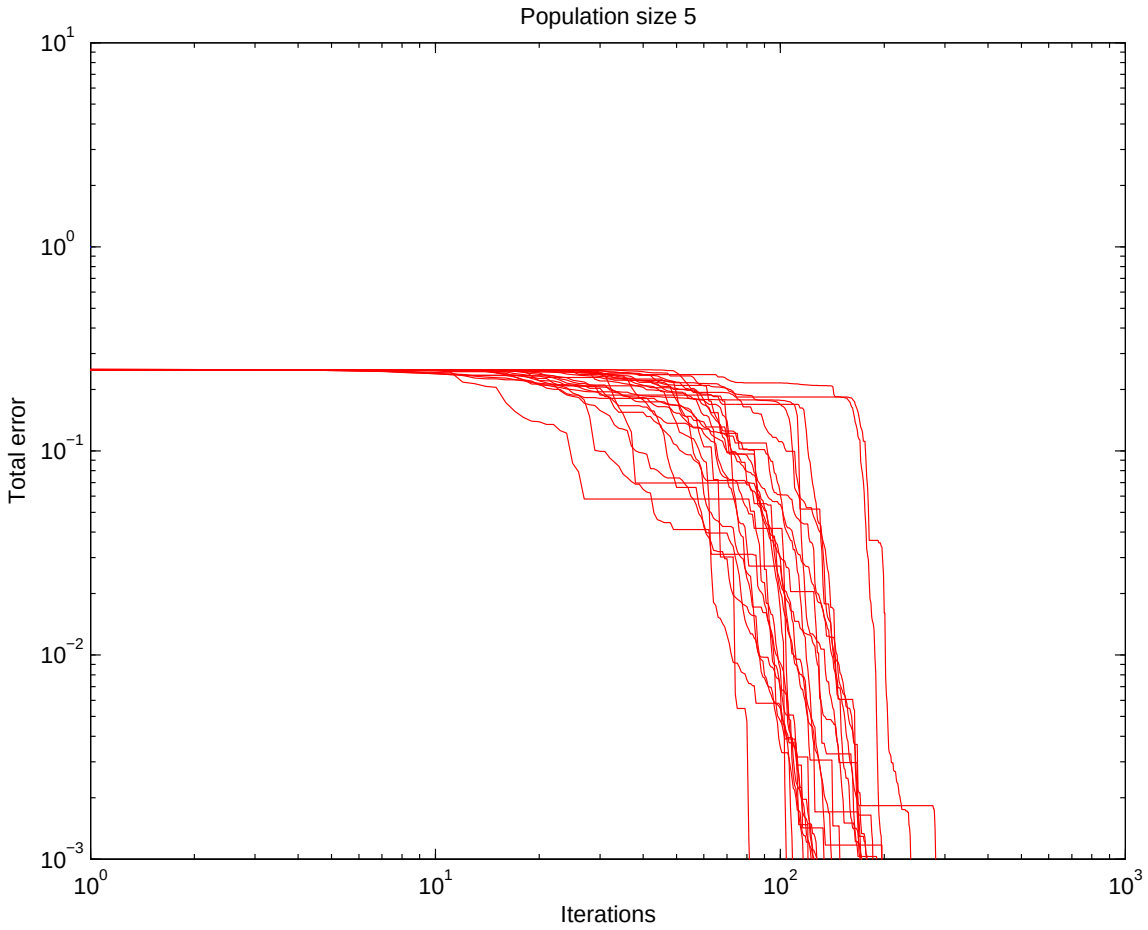

Figure 3.5: QGDM algorithm using modified distribution with k=0.1, p=5

Modified FF algorithm

Population	5	
Initial radius	5	
Total runs	100	
Min run	57	ite
Max run	376	ite
Ave. run	105	ite
Standard deviation	47	ite
Success rate	82	%

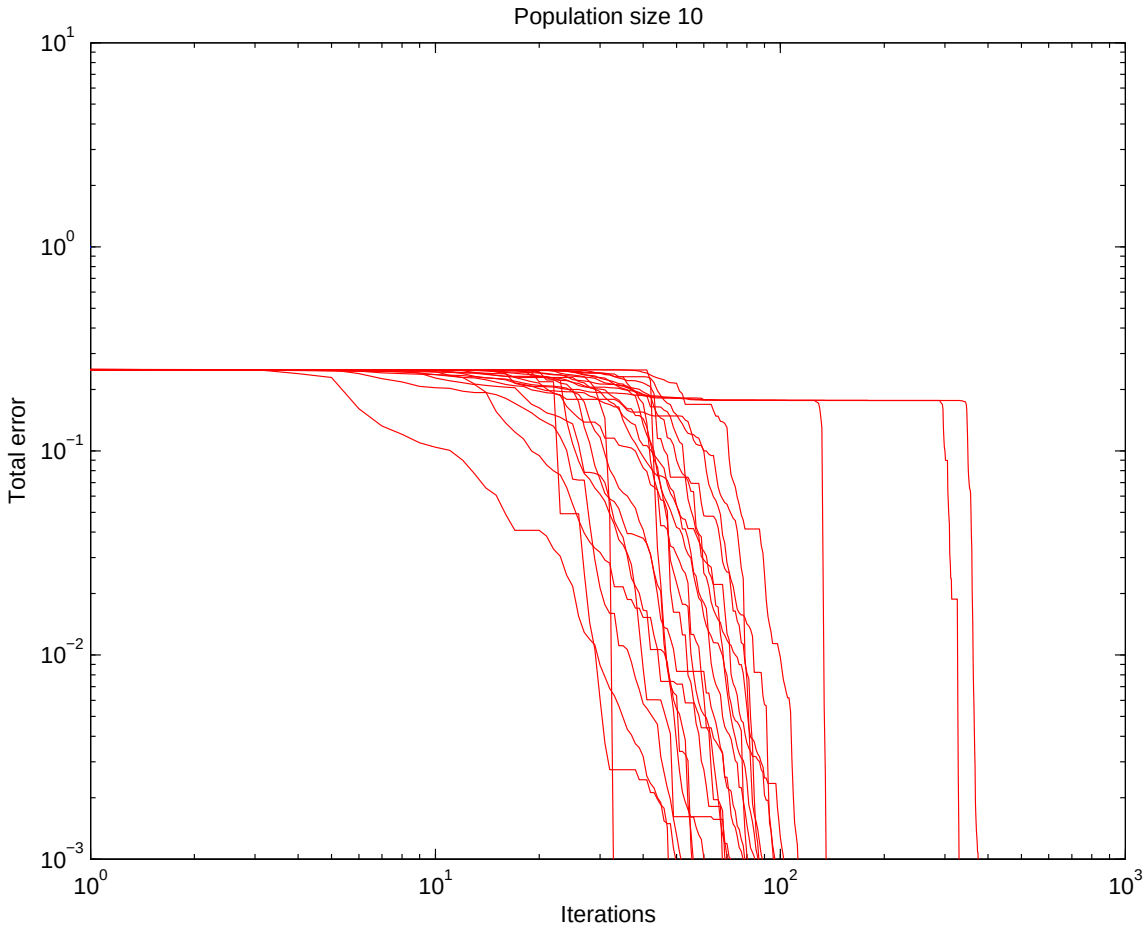

Figure 3.6: QGDM algorithm using modified distribution with k=0.1, p=10

Modified QGDM algorithm

Population	10	
Initial radius	5	
Total runs	100	
Min run	21	ite
Max run	302	ite
Ave. run	67	ite
Standard deviation	45	ite
Success rate	81	%

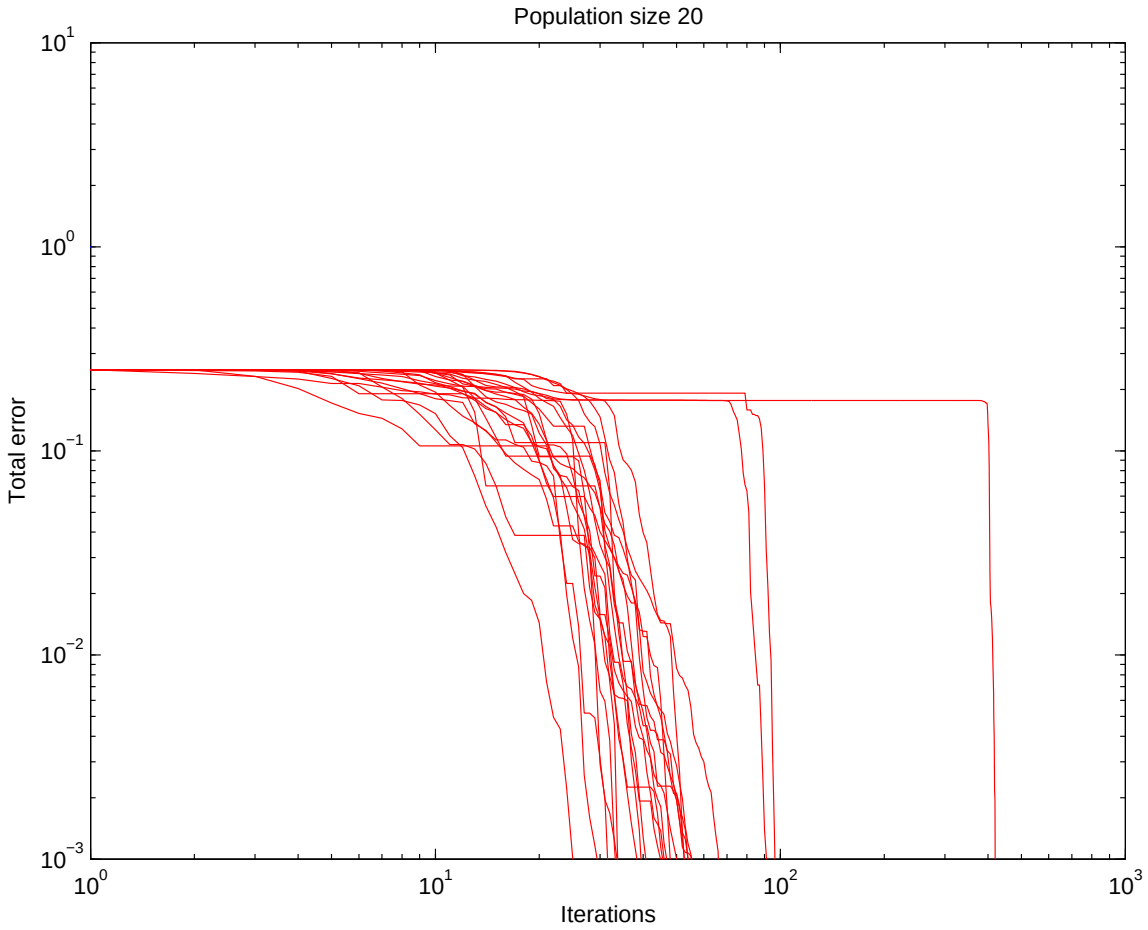

Figure 3.7: QGDM algorithm using modified distribution with k=0.1, p=20

Modified QGDM algorithm

Population	20	
Initial radius	5	
Total runs	100	
Min run	17	ite
Max run	84	ite
Ave. run	36	ite
Standard deviation	11	ite
Success rate	81	%

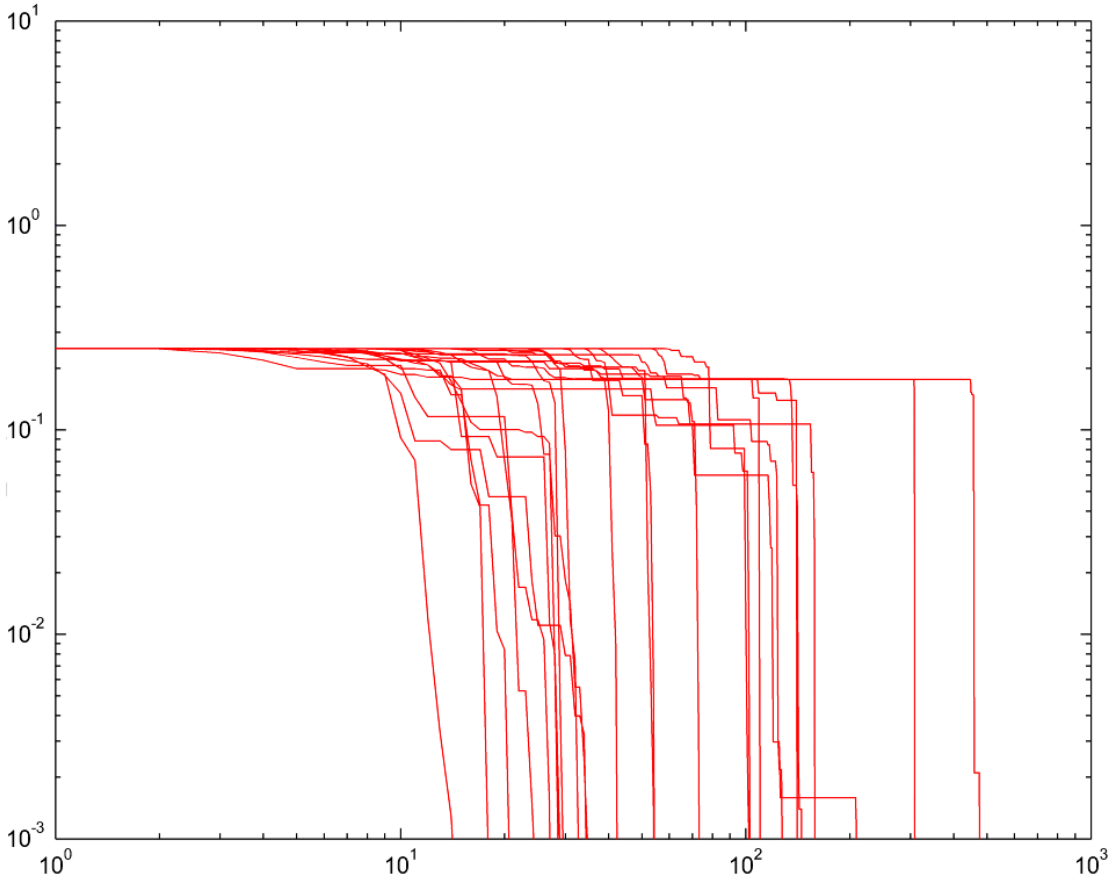

Figure 3.8: QGDM algorithm using $\beta = 1.5$, $\alpha = 2$

Standard QGDM algorithm

Population	5	
Min run	17	ite
Max run	301	ite
Ave. run	82	ite
Success rate	55	%

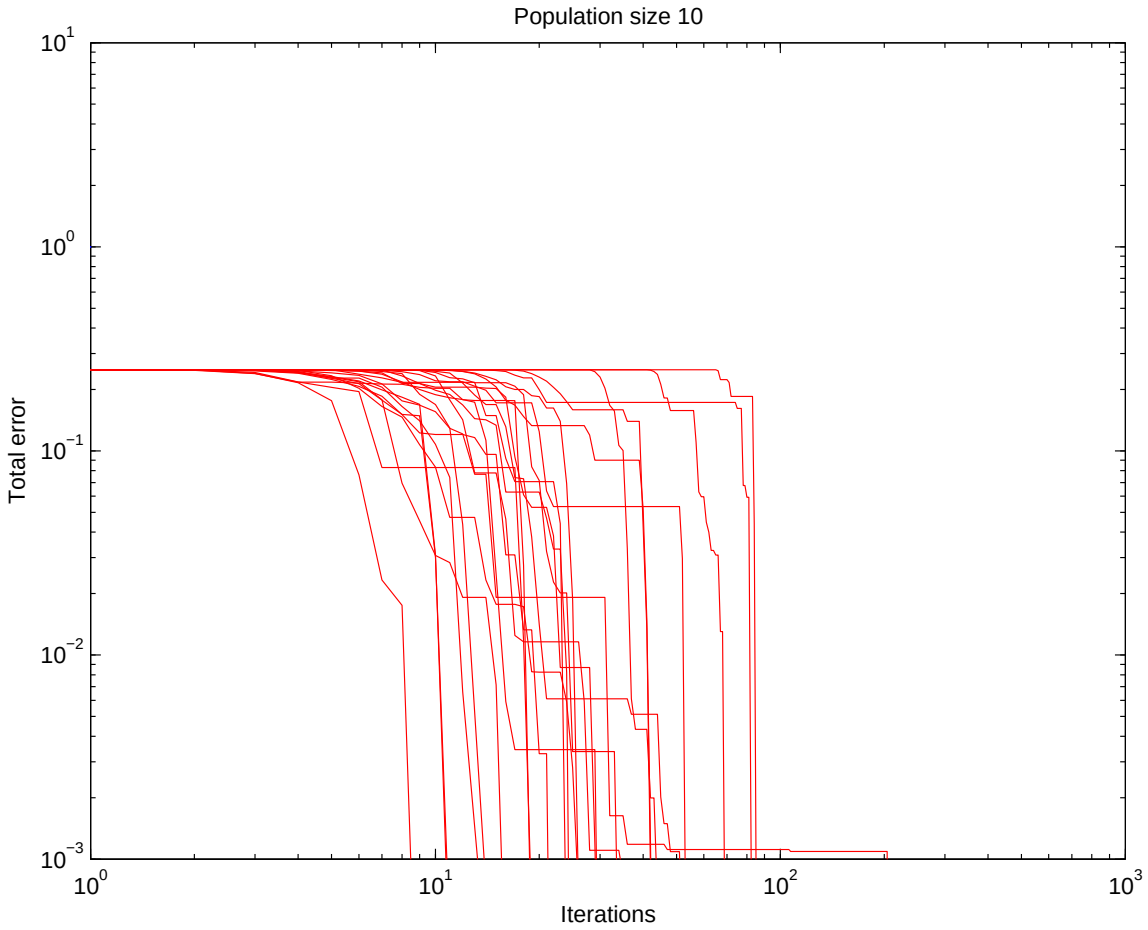

Figure 3.9: QGDM algorithm using $\beta = 1.5$, $\alpha = 2$

Standard QGDM algorithm

Population	10	
Min run	11	ite
Max run	159	ite
Ave. run	34	ite
Success rate	66	%

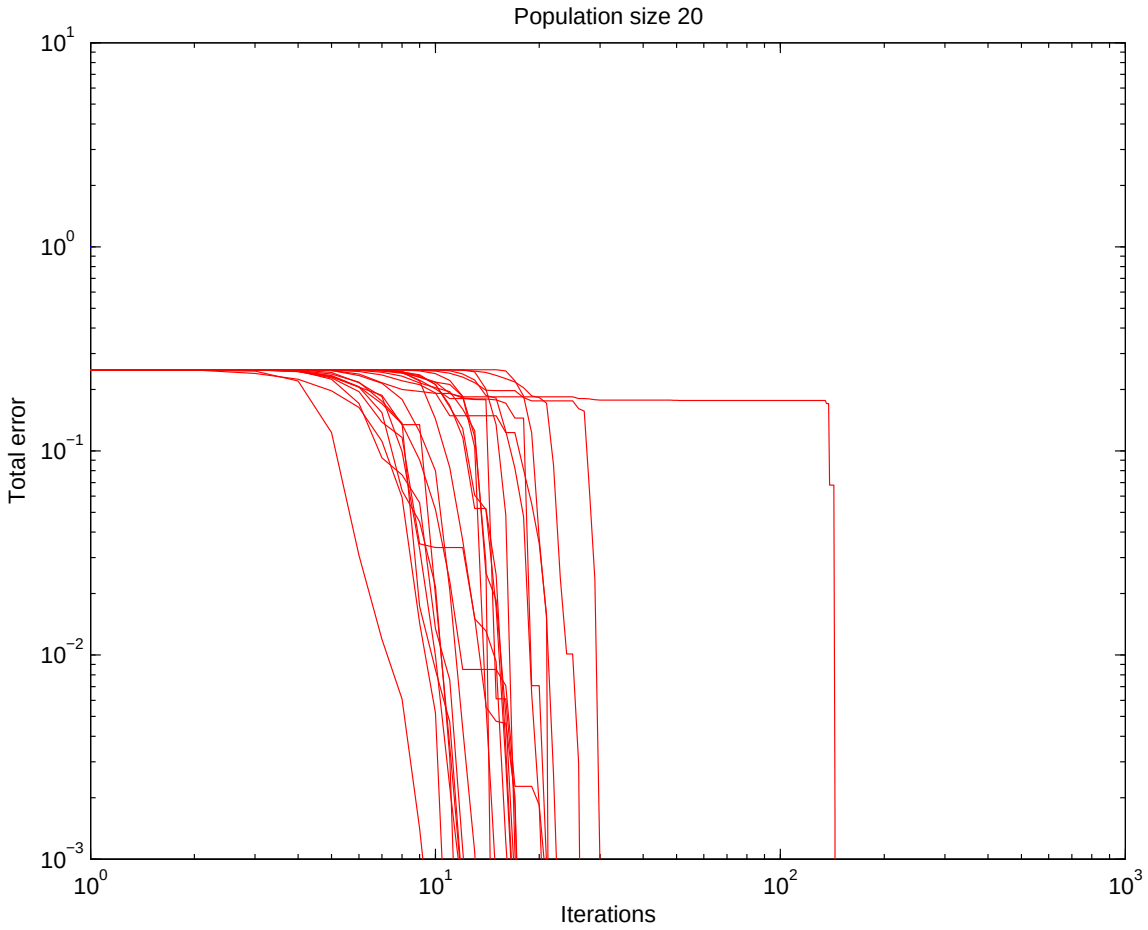

Figure 3.10: QGDM algorithm using $\beta = 1.5$, $\alpha = 2$

Standard QGDM algorithm

Population	20	
Min run	9	ite
Max run	197	ite
Ave. run	29	ite
Success rate	70	%

Chapter 4

Variable Scale Gradient Approximation

Gradient methods are often compared to a ball rolling along some surface. Neglecting momentum and assuming the ball is infinitely small, it will follow the path of steepest descent exactly. This is a powerful illustration which, consequently, also highlights the method's fundamental flaws. With little effort one can visualize any number of surfaces on which the ball might become permanently trapped before ever reaching its desired destination. Take, for example, a flight of stairs as in Fig. 4.1. Even though the behavior of the surface is relatively simple, it is clear that no matter where the ball on the left is placed, it will never find its way to the foot of the stairs. Though the staircase clearly exhibits a global trend which may be determined with only a limited amount of information, the ball is still incapable of proceeding. The advantage of this illustration is that it makes the solution quite obvious: use a bigger ball. Though changing the size of the ball won't change the gradient of the surface, it will make the ball more sensitive to larger scale behavior, allowing it to successfully descend the stairs. After all, all of the relevant information for descending a staircase can be found at the corners of the individual steps. While steps may be an essential part of a staircase, from the perspective of minimization, they are nothing more than noise. Furthermore, by actively allowing the ball to expand and contract as it descends, it is possible to navigate on as large or small a scale as necessary. This is the basic principle by which the proposed method operates, and from which it derives its name: Variable Scale Gradient Approximation (VSGA).

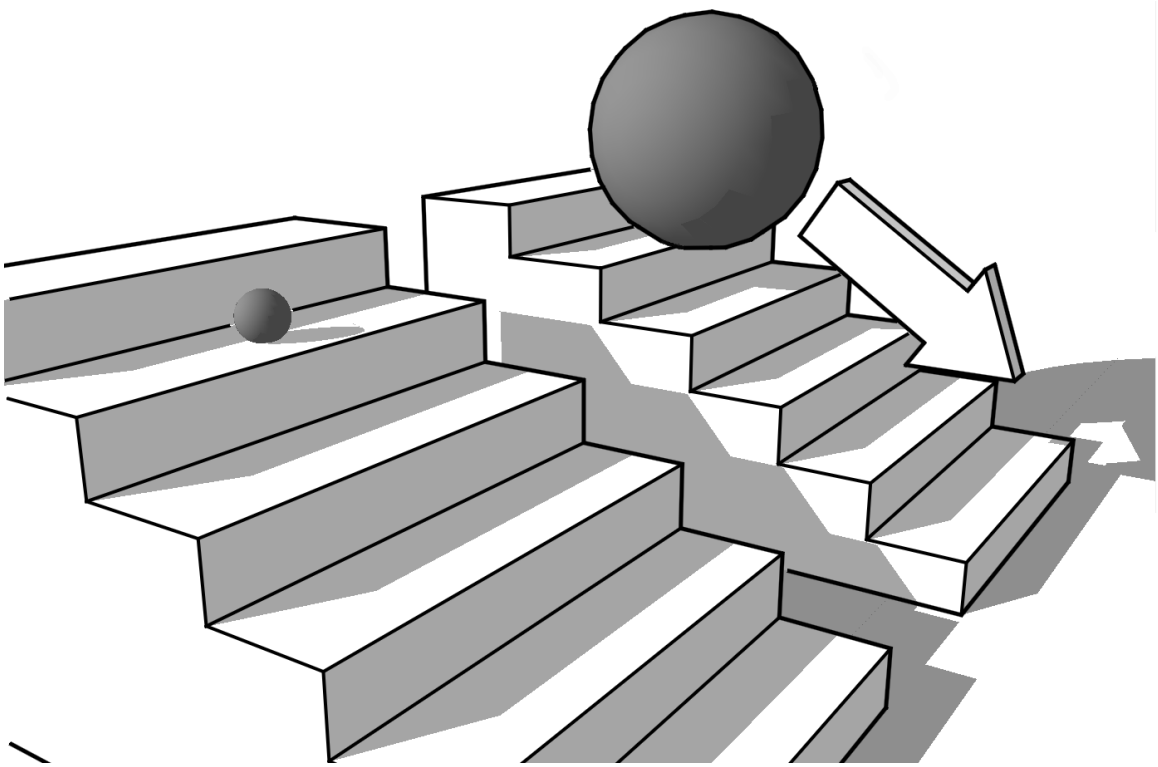

Figure 4.1: Ordinary gradient methods (left) are easily trapped on complex or piecewise constant optimization surfaces. The proposed method (right) is able to overcome these obstacles by expanding the radius of its population-based gradient approximation.

4.1 Proposed Method

The key feature of the proposed method is the way in which the gradient is approximated. During each iteration, the algorithm generates a "population" of points within a spherically bounded region. A portion of the generated set lies on the region's perimeter, and it is these members of the population which are used in the approximation of the gradient. Next, an additional point is generated in the direction of the approximated gradient and added to the current population. Finally, the fittest member of the population is compared to the current best solution, and the population's center and radius are updated accordingly. The basic flow of the algorithm is presented in Fig. 4.2.

4.1.1 Approximating the gradient

As previously noted, the gradient approximation is the defining feature of the proposed method. The generalized process for computing it is presented in a step-by-step manner.

Let $x_0 \in \mathbf{R}^n$ be the center of a population P with radius r, and let $y = f(x_0)$ be the value of the objective function $f : \mathbf{R}^n \rightarrow \mathbf{R}$ at x_0 .

1. First, generate a set of n random vectors $\{\Delta x_1, \Delta x_2, ..., \Delta x_n\}$, each of length r, and define an $n \times n$ matrix,

$$\Delta X = \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \end{bmatrix}$$

2. Now, let $x_i = \Delta x_i + x_0$, and define a vector $y = [y_1 ... y_n]^T$ where

$$y_i = f(x_i) \tag{4.1}$$

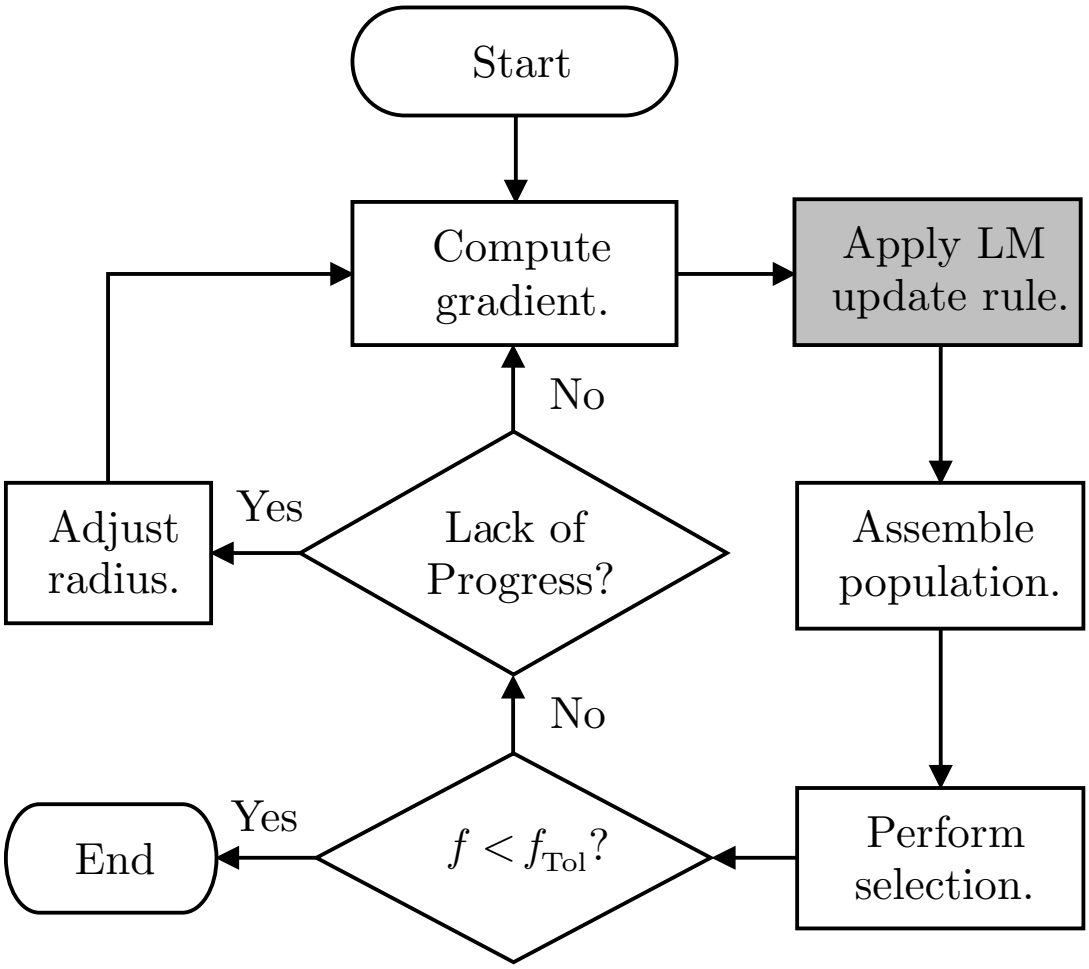

Figure 4.2: A simplified flow chart depicts the basic operation of the proposed algorithm.

for $i = 1, ..., n$. It should be clear from 4.1 that each value y_i corresponds to the value of the objective function at a randomly generated point x_i, which lies on the perimeter of the population.

3. Finally, compute the gradient approximation using

$$\nabla f = \Delta X^{-1} \cdot (y - y_0) \tag{4.2}$$

It may appear, and understandably so, that a significant leap has been made between steps 2 and 3. So for the sake of clarity, a short derivation of 4.2 is included. Let f be an n-dimensional scalar function. A linear approximation of f with respect to a point $x_0 = (x_{01}, x_{02}, ..., x_{0n})$ may be found using the first two terms of the Taylor series expansion.

$$\begin{aligned} f(x) &\approx f(x_0) + \tfrac{\delta f}{\delta x_1} \cdot \Delta x_1 + \tfrac{\delta f}{\delta x_2} \cdot \Delta x_2 + \cdots + \tfrac{\delta f}{\delta x_n} \cdot \Delta x_n \\ &= f(x_0) + \nabla f|_{x_0} \cdot \Delta x \end{aligned} \tag{4.3}$$

Solving this equation for the gradient yields

$$\nabla f|_{x_0} \approx \frac{\Delta f}{\Delta x} = \frac{f(x) - f(x_0)}{x - x_0} \tag{4.4}$$

By generating a point x in the vicinity of x_0, it is possible to obtain a numerical approximation of $\nabla f|_{x_0}$ using 4.4. However, for $n > 1$, 4.4 is an invalid expression since it implies that Δx is a vector, and therefore has no inverse. Thus, in order to generalize 4.4), f is evaluated for a set of n linearly independent points, allowing Δx and Δf to be replaced by the $n \times n$ matrix ΔX and the $n \times 1$ vector of corresponding function values Δf. This leads to the generalized form

$$\begin{aligned} \nabla f_{n\times 1} &\approx \begin{bmatrix} \Delta x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ \Delta x_{n1} & \cdots & x_{nn} \end{bmatrix}^{-1}_{n\times n} \begin{bmatrix} f(x_1) - f(x_0) \\ \vdots \\ f(x_n) - f(x_0) \end{bmatrix}_{n\times 1} \\ &\approx \Delta X^{-1} \cdot \Delta f \end{aligned} \tag{4.5}$$

which is equivalent to 4.2.

4.1.2 A modified LM update rule

Once the gradient is computed, an additional point x_{n+1} is generated in the direction of the approximation. The proposed method uses a modified version of the Levenberg-Marquardt (LM) algorithm [19] update rule for this purpose. For a scalar valued function $f : \mathbf{R}^n \rightarrow \mathbf{R}$, the LM update rule may be written

$$x_{n+1}^{(k+1)} = x_0^{(k)} - (\nabla f \cdot \nabla f^T + \mu I)^{-1} \nabla f \cdot y_0^{(k)}, \tag{4.6}$$

where k is the current iteration, and μ acts as a damping factor which reduces oscillation by actively controlling the step size. One who is familiar with the LM algorithm may notice that the Jacobian has been replaced by the gradient in 4.6. This is because the Jacobian is defined as the matrix of all first-order partial derivatives of a vector valued function. Thus, for a scalar valued function, the Jacobian is, by definition, the transpose of the gradient. Although the proposed method may be extended to handle vector valued functions, the version presented here is intended for use with scalar valued problems. Thus the Jacobian is replaced by the gradient.

This is not the only modification. For the proposed implementation, the rule is also modified to account for the population radius r. This is done by adding a term which ensures that the length of the update step will be no smaller than r. This modification is needed to ensure that when the algorithm encounters local minima, the step size will exceed the radius of the current population, thus providing greater diversity. The modified version of the rule is written

$$x_{n+1}^{(k+1)} = x_0^{(k)} - (\nabla f \cdot \nabla f^T + \mu I)^{-1} \nabla f \cdot y_0^{(k)} - \frac{\nabla f}{\|\nabla f\|} \cdot r^{(k)} \tag{4.7}$$

The added term in 4.7 is simply a vector of length r in the direction of ∇f.

It is important to note that the algorithm presented here, using the modified version of the LM update rule, is but one of many possible implementations. In other words, the shaded box in Fig. 2 may be replaced with any number of existing gradient methods without compromising the underlying principles. The only requirement is that the gradient be calculated in the described fashion. In fact, the proposed method was specifically designed with this sort flexibility in mind. Therefore, the update rule may be readily exchanged with another second order method such as the BFGS method [20] or the closely related Davidon Fletcher Powell algorithm, which has been shown to outperform LM in certain applications [21].

4.1.3 Assembling the population

The population P is generated in two parts whose combined size is denoted by the parameter λ. The primary population,

$$A = \{x_1, ..., x_{n+1}\}, \tag{4.8}$$

is created in the two previous steps, and is of size $n + 1$, while the secondary population,

$$B = \{x_{n+2}, ..., x_\lambda\}, \tag{4.9}$$

consists of a set of m points which are randomly distributed within the region defined x_0 and r. Here, the value of m is a user defined parameter, and may be assigned any non negative integer value including zero. This leads to the following definitions for λ and P,

$$\lambda = n + 1 + m, \tag{4.10}$$

$$P = A \cup B = \{x_1, ..., x_\lambda\}. \tag{4.11}$$

While the inclusion of B in P can provide a noticeable increase in the rate of convergence when applied to relatively complex optimization surfaces, in general, it is sufficient to let $m = 0$.

4.1.4 The selection process

Once P has been constructed according to 4.11, its members are ranked with respect to fitness. Next, the fittest member $x_{opt} \in P$ is chosen, and the following conditional, also shown in Fig. 2, is evaluated:

IF $f(x_{opt}) < f_{Tol}$ **THEN**

Terminate algorithm.

ELSE IF $f(x_{opt}) \geq f(x_0)$ **THEN**

Adjust population radius r.

ELSE

$x_0 = x_{opt}$.

END IF

where $f_{Tol}$is the maximum acceptable value for the objective. Updating in this way essentially recycles the information used in the approximation of the gradient, which would otherwise be thrown out. This process also helps to guarantee the algorithm's stability since it is equivalent to running two methods in parallel. That is, if the generated step does not result in a lower value of the objective, the method will resort to selecting the best of the points used in the gradient approximation. In this way, it is still possible for the method to proceed. Furthermore, because the current best solution is included in the selection process, it is guaranteed that the value of the objective will not increase from one iteration to the next. Therefore, while the instability of the update rule may affect the algorithm's convergence, the algorithm itself will remain stable as a result of selection.

4.1.5 Adjustment of the population radius

Any number of radius update rules may be devised, however the version presented here is perhaps the simplest. The method uses a fixed step size Δ to modify the radius over a bounded user defined interval $[r_{min}, r_{max}]$, and is subject to the following conditions:

IF $r \geq r_{max}$ **THEN**

$r^{(k+1)} = r^{(k)} + \Delta$.

ELSE

$r^{(k+1)} = r_{min}$.

END IF

where k is the number of the current iteration. Although this method is simple, it proves to be adequate. The two features which make the method so basic are the use of a resetting value of r, and the constant step-size Δ. The constant step size serves the purpose of increasing the population radius when the algorithm encounters local minima, while the increase in r causes the gradient approximation to become less sensitive to local behavior, thereby allowing it to escape the traps introduced by complex behaviors. Conversely, if greater accuracy of approximation becomes necessary for local search, the algorithm is still able to proceed once r is reset.

4.2 Test Functions

Four unique functions were used for testing. Each function exhibits features deemed relevant for the purpose of comparison. All four functions are presented in generalized form, with n being the dimension of the search space.

4.2.1 Test Function 1

$$T_1(x) = \sum_{i=1}^{n} \left(\frac{x_i}{4}\right)^4, \; x_i \in [-10, +10] \tag{4.12}$$

The first test function, T_1, has a simple convex, continuous, parabolic surface with a minimum of $T_1(x) = 0$ located at the origin. The function is used for two primary purposes. First, it offers a fair comparison of the proposed method with some of the more common gradient methods. Second, it highlights the strengths of such methods, which are superior to evolutionary methods when applied to problems of this type. Also, because the function is fourth order, it will highlight the difference between algorithms of higher and lower orders.

4.2.2 Test Function 2

$$T_2(x) = \sum_{i=1}^{n} \left(\frac{\lfloor x_i \rfloor}{4} \right)^4, \; x_i \in [-10, +10] \tag{4.13}$$

The second test function, inspired by the illustration in Ch. 4, is of an identical form to that of T_1, except that the variables have been floored in order to make it piecewise constant. T_2 has a minimum value of 0 for all $x = [x_1 ... x_n]$ which satisfy $x_1 \in [0, 1)$ for $i = 1...n$. The features of this function are useful for testing the hypothesis made in Ch. 4. If the proposed method operates as intended, there should be little difference in performance between the T_1 and T_2.

4.2.3 Test Function 3

$$\begin{aligned} f(x) &= \tfrac{1}{2} \sqrt{\sum_{i=1}^{n} \lfloor x_i \rfloor^2}, \; x_i \in [-100, +100] \\ g(x) &= \tfrac{x}{4} + (1 - \cos(\pi x)) \cdot (\tanh\left(\tfrac{x}{4}\right) - 1)^2 \\ T_3(x) &= g \circ f \end{aligned} \tag{4.14}$$

Like T_2, T_3 is also piecewise constant, but with the addition of local minima, which is the most common challenge faced by gradient methods. Though evolutionary methods may also become susceptible to these traps if population diversity becomes too low, they much better suited for this type of problem. T_3 has a minimum value of 0 for all $x = [x_1 ... x_n]$ which satisfy $x_i \in [0, 1)$ for $i = 1...n$.

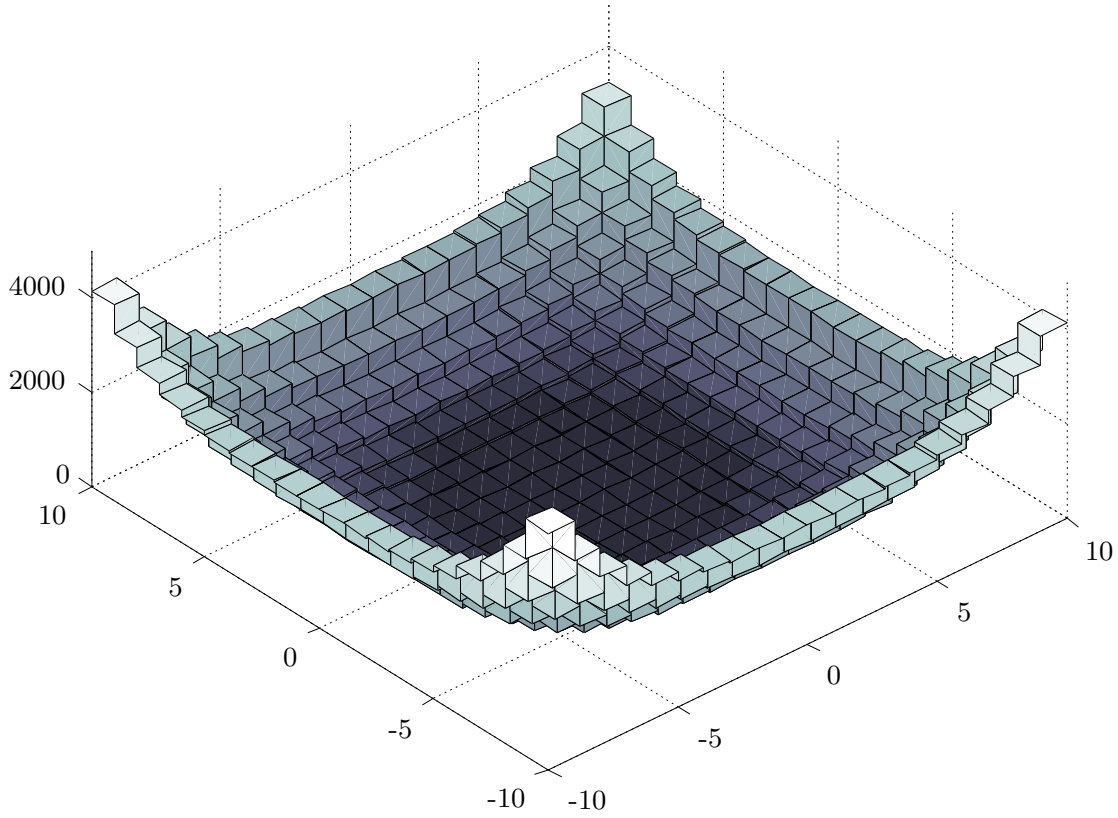

Figure 4.3: Test Function 2

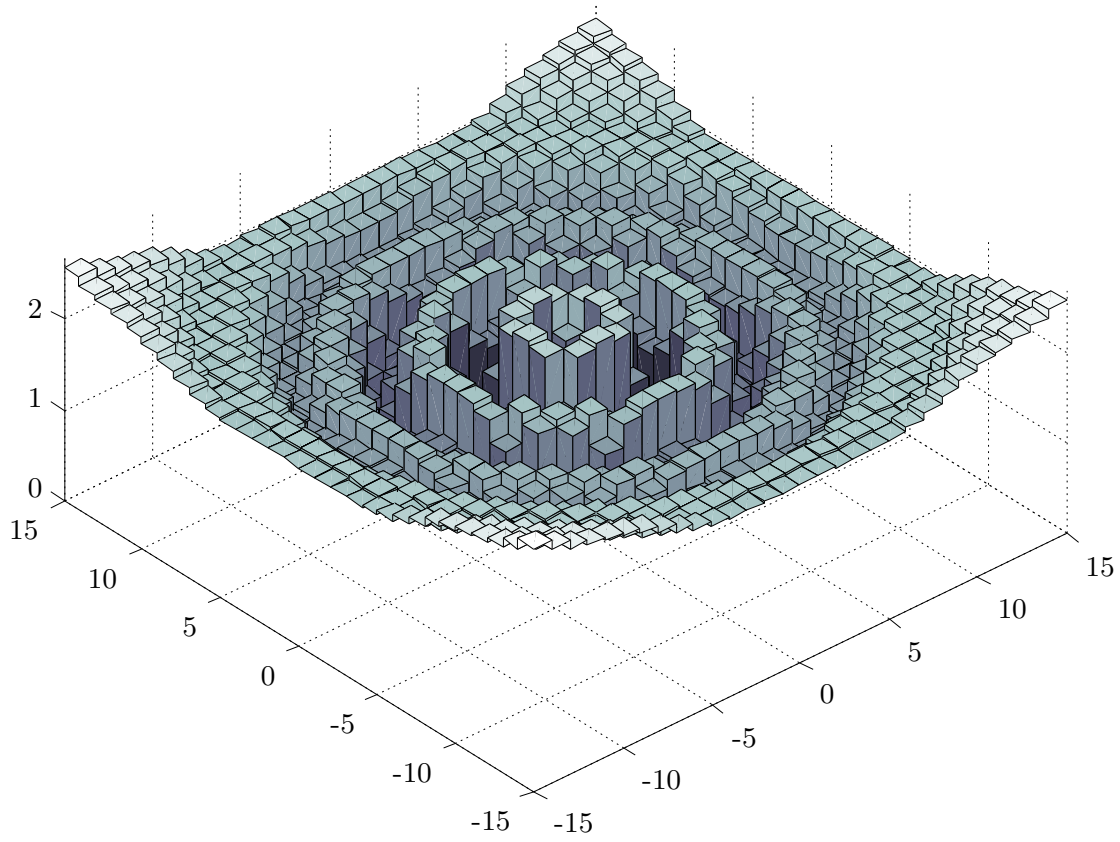

Figure 4.4: Test Function 3

Algorithm	Parameters	T_1	T_2	T_3	T_4
MSD	α	1	1	1	1
LM	μ_0	0.1	0.1	0.1	0.1
SA-ES	μ, λ, σ_0	3,12,1	3,12,1	8,16,10	8,16,50
CMA-ES	μ, λ, σ_0	3,12,1	3,12,1	8,16,10	8,16,50
PM	$m, r_{min}, r_{max}, \Delta$	$0, 10^{-16}, 1, 1$	$0, 2, 6, 2$	$4, 2, 6, 2$	$0, 10^{-6}, 12, 3$

Table 4.1: A list of algorithm parameters used in testing

4.2.4 Test Function 4

$$T_4(x) = \frac{1}{2}\sum_{i=1}^{n} \left(x_i^2 + \tan(x_i)^2 - 10 \cdot \cos(2 \cdot x_i) + 10\right),\ x_i \in [-100, +100] \tag{4.15}$$

T_4 is the most extreme of the test functions. Though it is piecewise continuous, each of the regions of continuity is a deep local minimum. Thus, gradient methods are only capable of minimizing T_4 on a local basis. Due to the large number of local minima, evolutionary methods which rely on modified mutation strength for accelerated convergence may also be susceptible to entrapment. The surface of T_4 is shown in Fig. 4.5. In reality, the "walls" which separate the local minima in Fig. 4.5 are infinitely high. The minimum of T_4 is 0, and is located at the origin.

4.3 Experimental Results

The proposed method is compared with four well known algorithms. It should be noted that the purpose of the comparison is not to show that the proposed method is superior to all algorithms over all problems; clearly that is not the case. Instead, the goal is to show that while the algorithm shows the high rate of convergence and efficient local search characteristics of a second order gradient method, it is also capable of minimizing complex classes of problems which are usually associated with evolutionary methods. Thus the evolutionary methods used for comparison were selected on the basis of these same qualities. The following is a list of the compared methods.

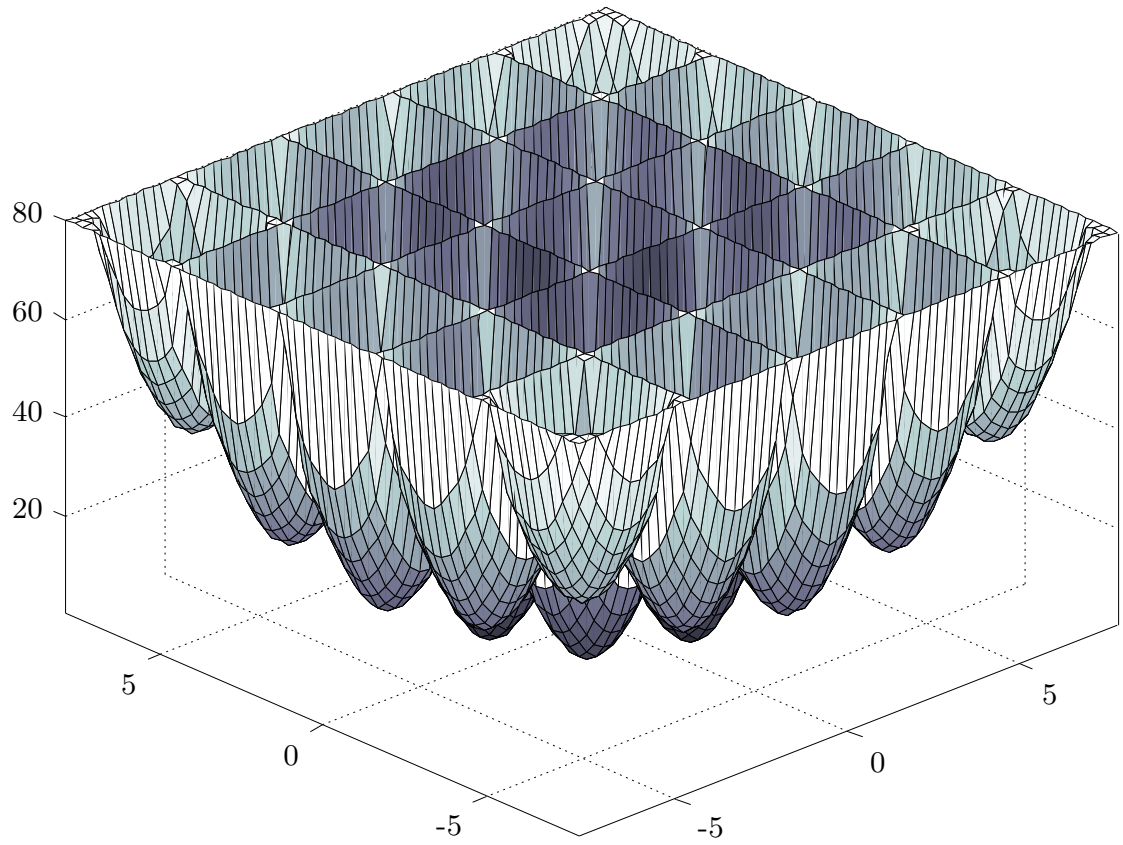

Figure 4.5: Test Function 4

Algorithm	Test Function			
	T_1	T_2	T_3	T_4
MSD	100%	FAILURE	FAILURE	FAILURE
	5477.98	FAILURE	FAILURE	FAILURE
LM	100%	FAILURE	FAILURE	FAILURE
	43.6	FAILURE	FAILURE	FAILURE
SA-ES	100%	100%	100%	48%
	214.48	426.28	355.72	6475.7
CMA-ES	100%	100%	100%	67%
	186.68	138.32	772.24	815.61
PM	100%	100%	100%	100%
	46.3	28.72	148.21	382.36

Table 4.2: Comparison of success rate and mean function evaluations

1. MSD: The Method of Steepest Descent is perhaps the most well known of all gradient methods. MSD is a simple first order method which, as the name implies, employs a user defined step size to proceed in the direction of "steepest descent." The method was chosen for its high degree of stability as well as its reputation as the standard gradient method.

2. LM: The Levenberg Marquardt algorithm, described briefly in Section III-C, is regarded as one of the fastest gradient methods available. The method was chosen as a benchmark among second order algorithms. Furthermore, the gradient portion of the proposed method uses an update rule directly inspired by the LM algorithm, making the method especially relevant for comparison.

3. -SA-ES: Self-Adaptive Evolution Strategy [22], a member of the larger family of algorithms known as Evolutionary Strategies [22], is a powerful evolutionary method which uses self-adapted mutation strength for optimal convergence as well as an accelerated local search. The method was chosen as a strong representative of the power of evolutionary methods.

4. CMA-ES: Covariance Matrix Adaptation Evolutionary Strategy [23], also a member of the family of Evolutionary Strategies, is an evolution path related technique which uses search space information in a highly efficient manner, making it exceptionally fast with respect to other evolutionary methods.

5. PM: The proposed method.

The performance of each algorithm on each of the test functions was evaluated over a series of 100 simulation runs using the algorithm parameters listed in Table 4.1. Each algorithm was then evaluated with respect to rate of success as well as the mean number of function evaluations per solution. The tabulated results are presented in Table 4.2, which represents the case $n = 2$ for all four functions. For each of the test functions, success was defined by two conditions:

1. $f(x) < 10^{-6}$

2. No more than 10^5 total evaluations of the objective.

The fields in Table 4.2 highlighted in bold font denote the top performers for each of the test cases. One may notice the similarity between the proposed method and the LM algorithm with regard to the first test function. Notice that for this particular function, the behaviors of the two algorithms are nearly identical. This validates the earlier hypothesis that, in the absence of local minima, the two methods should behave in a similar manner. Note too the striking difference in the rate of convergence between the two second order gradient methods when compared to the first order method of steepest descent. Another point of interest is the performance of the proposed method on the second test. It was suggested in Section IV-B that, assuming proper performance, little difference should be observed between T_1 and T_2. This is confirmed by the results in Table 4.2. In fact, the algorithm actually performs better! This behavior is a result of the increased population radius coupled with the modified LM update rule, which, when used together, increase the rate at which the proposed method traverses the optimization surface.

Finally, notice the similarity between the proposed method and that of CMA-ES with respect to T_3 and T_4. In fact, for T_4, the similarities are striking. When applied to these more complex surfaces, the behavior of the algorithm changes drastically, and yet there seems to be little, if any, effect on the level of performance. What is most striking about the results in Fig. 4.6 is the change in behavior between T_1 and T_4. As the nature of the test functions becomes more complex, the behavior of the proposed method changes from a second order gradient method to one which bears a striking resemblance to that of evolutionary computation!

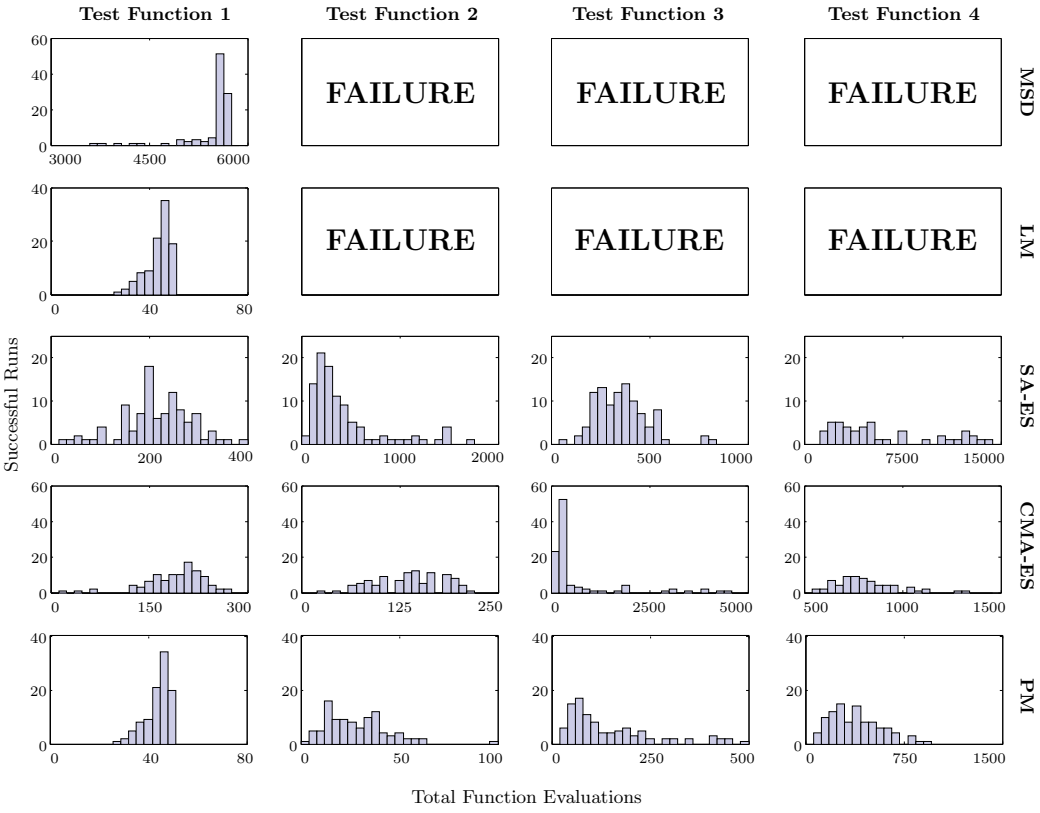

Figure 4.6: A Graphical summary of the test results

Chapter 5

Conclusion

In this thesis, two novel algorithms for numerical optimization were presented. In both cases, the objective was to modify the general form of the traditional first and second order approaches in order to include global solutions to complex problems, while also retaining desirable local behavior commonly associated with such approaches. This was motivated by the fact that traditional gradient based approaches are susceptible to entrapment in local minima, and will only produce global convergence if the initial guess happens to lie within the global minimizer's region of attraction. Conversely, while newer methods such as evolutionary computation are quite adept at obtaining global solutions, they are very inefficient with respect to local convergence, making them prohibitively time consuming for many applications.

The first approach, presented in Chapter 3, borrows some from evolutionary computation through the use of successive populations of pseudo-randomly generated test points. However, instead of using the mechanisms of combination and random mutation to generate successive populations, the proposed method uses the cumulative information contained within each generation to "migrate" from one point to the next. This migratory behavior allows the algorithm to progress in a way which is similar to gradient methods, while the use of randomly distributed populations make it possible to overcome the challenges presented by local minima. The latter is done by actively controlling the population's radius, allowing test points to venture beyond the region of local attraction. Not only does this method require very little computational effort, but it was also shown to outperform a common first order method for the purpose of training a neural network.

In Chapter 4, a method was presented by which nearly any first or second order algorithm can be modified to overcome the risk of entrapment. This was done through the use of

a variable scale gradient approximation (VSGA), which effectively controls the sensitivity of the search direction with respect to the small and large scale behaviors of the objective. Like its predecessor, the VSGA is extracted using the information contained within a pseudo-randomly generated set of test points. The difference, however, is that the VSGA direction yields a far more accurate model of the objective function's behavior with respect scale. Thus, as the radius of the population increases, the VSGA becomes less sensitive to local behavior, allowing it to overcome the challenge of local minima. Conversely, when the radius of the population is quite small, the VSGA becomes a very close approximation to the true gradient, making it quite efficient in local searches as well. To confirm this, the algorithm was tested using a series of test functions chosen to typify some of the most commonly faced difficulties. The performance of the proposed method was also compared with a set of competing algorithms which included gradient as well as evolutionary methods. It was than shown from the results that the proposed method was not only able to emulate both classes of algorithms based on the behavior of each function, but it was also shown to out-perform the compared methods in terms of rate of success and average number of evaluations of the objective.

Bibliography

[1] H. A. Abbass, A. M. Bagirov, J. Zhang, "The discrete gradient evolutionary strategy method for global optimization," *Congress on Evolutionary Computation - CEC'03* Volume 1, 8-12 Dec. 2003 pp. 435 - 442 Vol.1

[2] M. Manic and B. Wilamowski, "Random Weights Search in Compressed Neural Networks using Overdetermined Pseudoinverse," *Proc. of the IEEE Interantional Symposium on Industrial Electronics - ISIE'03* Rio de Janeiro, Brazil, June 9-11, 2003, pp. 678 - 683.

[3] M. Manic and B. Wilamowski "Robust Neural Network Training Using Partial Gradient Probing," *IEEE International Conference on Industrial Informatics - INDIN'03* Banff, Alberta, Canada, August 21-24, 2003, pp. 175 - 180.

[4] J. Y. Wen, Q. H. Wu, L. Jiang, S.J. Cheng, "Pseudo-gradient based evolutionary programming,",*Electronics Letters* Volume 39, Issue 7, 3 April 2003 pp. 631 - 632

[5] B. Wilamowski "Neural Networks and Fuzzy Systems," *The Microelectronic Handbook* CRC Press -2006 chapters 124.1 to 124.8.

[6] Derenick, J. C.; Spletzer, J. R., "Convex Optimization Strategies for Coordinating Large-Scale Robot Formations," Robotics, IEEE Transactions, vol.23, no.6, pp.1252-1259, Dec. 2007

[7] Seok-Beom Roh, W. Pedrycz, Sung-Kwun Oh, "Genetic Optimization of Fuzzy Polynomial Neural Networks," Trans. on Industrial Electronics, vol. 54, no. 4, pp. 2219-2238, Aug. 2007.

[8] L. dos Santos Coelho, B. M. Herrera, "Fuzzy Identification Based on a Chaotic Particle Swarm Optimization Approach Applied to a Nonlinear Yo-yo Motion System," Trans. on Industrial Electronics, vol. 54, no. 6, pp. 3234-3245, Dec. 2007.

[9] S.-K. Oh, W. Pedrycz, H.-S. Park, "A New Approach to the Development of Genetically Optimized Multilayer Fuzzy Polynomial Neural Networks," Trans. on Industrial Electronics, vol. 53, no. 4, pp. 1309- 1321, Aug. 2006.

[10] Faa-Jeng Lin, Po-Kai Huang, W.-D. Chou, "Recurrent-Fuzzy-Neural-Network-Controlled Linear Induction Motor Servo Drive Using Genetic Algorithms," Trans. on Industrial Electronics, vol. 54, no. 3, pp. 1449-1461, June 2007.

[11] Grimaccia, F.; Mussetta, M.; Pirinoli, P.; Zich, R.E., "Genetical Swarm Optimization (GSO): a class of Population-based Algorithms for Antenna Design," Communications and Electronics, 2006. ICCE '06. First International Conference on , pp.467-471, 10-11 Oct. 2006

[12] Grimaccia, F.; Mussetta, M.; Pirinoli, P.; Zich, R.E., "Optimization of a reflectarray antenna via hybrid evolutionary algorithms," Electromagnetic Compatibility, 2006. EMC-Zurich 2006. 17th International Zurich Symposium on , pp. 254-257, 27 Feb.-3 March 2006

[13] Jinn-Tsong Tsai, Jyh-Horng Chou, Tung-Kuan Liu, "Optimal design of digital IIR filters by using hybrid taguchi genetic algorithm," Trans. on Industrial Electronics, vol. 53, no. 3, pp. 867- 879, June 2006.

[14] Yang Yu; Yu Xinjie, "Cooperative Coevolutionary Genetic Algorithm for Digital IIR Filter Design," Industrial Electronics, IEEE Transactions on , vol.54, no.3, pp.1311-1318, June 2007

[15] Salomon, R., "Evolutionary algorithms and gradient search: similarities and differences," Evolutionary Computation, IEEE Transactions on , vol.2, no.2, pp.45-55, Jul 1998

[16] M. Bundzel and P. Sincak, "Combining Gradient and Evolutionary Approaches to the Artificial Neural Networks Training According to Principlesof Support Vector Machines," IEEE International Joint Conference on Neural Networks - IJCNN'06, pp. 2068 - 2074, 16-21 July 2006.

[17] X. Hu; Z. Huang; Z. Wang "Hybridization of the multi-objective evolutionary algorithms and the gradient-based algorithms," Congress on Evolutionary Computation - CEC'03, vol. 2, pp. 870 - 877, 8-12 Dec. 2003

[18] J. Hewlett, B. Wilamowski, and G. Dundar "Merge of Evolutionary Computations with Gradient Based Method for Optimization Problems" ISIE 07 IEEE International Symposium on Industrial Electronics, Vigo, Spain, June 4-7, 2007.

[19] D. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," SIAM Journal on Applied Mathematics, vol. 11, pp. 431-441, 1963.

[20] J. Nocedal and S. J. Wright, Numerical Optimization. 2nd ed., New York: Springer, 2006, pp. 136-143.

[21] Abid, S.; Mouelhi, A.; Fnaiech, F., "Accelerating the Multilayer Perceptron Learning with the Davidon Fletcher Powell Algorithm," Neural Networks, 2006. IJCNN '06. International Joint Conference on, pp.3389-3394

[22] H.-G. Beyer, H.-P. Schwefel, "Evolution Strategies: A comprehensive introduction," Natural Computing, vol. 1, pp. 3-52, 2002.

[23] N. Hansen, S. D. Mller, P. Koumoutsakos, "Redusing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES)," Evolutionary Computing, vol. 11, no. 1, pp. 1-18, 2003.

[24] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer-Verlag, 2006.

Appendices

A.1 QGDM.m

```
function [WW1,WW2,OP,ser,ite,k]=QGDM(fun,IP,DP,WW1,WW2,k1,k2,emax,nmax,nnp,r1);
% [WW1,WW2,OP,ser,ite,k]=QGDM(fun,IP,DP,WW1,WW2,k1,k2,emax,nmax,nnp,r1) - erorr backprop for 2 layers
% uses: versize2, augm, forw1l, oerror, fsbip, fsli, fsuni, newwt3
%
%    Arguments:
%           fun - activation function 'fsbip'
%           IP - [np,ni] matrix of inputs (not augmented).
%           DP - [np,no] matrix of desired outputs
%           WW1 - [nh,ni]  weight matrix for the first layer
%           WW2 - [no,nh]  weight matrix for the second layer
%               ni - number of inputs
%               nh - number of hidden neurons
%               no - number of outputs
%               np - number of patterns
%                 not working for unipolar
%           k1- neuron gain at net=0 for first layer
%           k2- neuron gain at net=0 for second layer
%           emax - max normalized error
%         nmax - max number of iterations
%           nnp - number of points per population
%           r1 - starting radius
%    Returns:
%           WW1 - [nh,ni]  weight matrix for the first layer
%           WW2 - [no,nh]  weight matrix for the second layer
%           OP - [np,no] matrix of outputs
%           ser  - vector of normalized errors as function of iterations
%           ite - number of total iterations
if nargin ~= 11  error('Wrong number of arguments');   end;
[ni,no,np,nh]=versize2(IP,DP,WW1,WW2);
IP1=augm(IP);
step=1;
je=0;
c1=WW1;
c2=WW2;
r=r1;
k=0;
%---------------Calculate network output (OP2)-----------------%
   [NET1,OP1,GAIN1] = forw1l(fun,IP1,WW1,k1);                  %
   IP2=augm(OP1);                                              %
   [NET2,OP2,GAIN2] = forw1l(fun,IP2,WW2,k2);                  %
%-------------------------------------------------------------%
%-----------------Calculate initial error (e)-----------------%
e = oerror(DP,OP2);                                            %
%-------------------------------------------------------------%
for ite=1:nmax
  [WW1,WW2,r,e,OP,c1,c2]=newwt3(WW1,WW2,r,e,DP,IP1,fun,k1,k2,c1,c2,nnp);
  if (ite/step == round(ite/step)) | (ite==1)
     je=je+1;
     ser(je)=e;
     %disp(sprintf('ite=%5d  error=%12.10f',ite,e));
   end;
   %WW1
```

```
   %WW2
if e<emax return; end;
end;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ww12,ww22,r,e1,OP,c1,c2]=newwt3(WW1,WW2,r,e,DP,IP1,fun,k1,k2,c1,c2,nnp);
%----------return initial state if e2 is never less than e1----------%
OP=0;                                                                  %
e1=e;                                                                  %
ww12=WW1;                                                              %
ww22=WW2;                                                              %
for i=1:nnp                                                            %
%-----------------Create random weights (wwa,wwb)--------------------%
    wwa=2*rand(size(WW1))-1;                                           %
    wwb=2*rand(size(WW2))-1;                                           %
    magww=sqrt(sum(sum(wwa.^2)')+sum(wwb.^2));                         %
    rr=r*rand;                                                         %
    wwa=rr*wwa/magww+c1;                                               %
    wwb=rr*wwb/magww+c2;                                               %
%------------------feed-forward calculation (OP2)--------------------%
   [NET1,OP1,GAIN1] = forw1l(fun,IP1,wwa,k1);                          %
   IP2=augm(OP1);                                                      %
   [NET2,OP2,GAIN2] = forw1l(fun,IP2,wwb,k2);                          %
%-----------------------Calculate error (e2)-------------------------%
e2 = oerror(DP,OP2);                                                   %
%-----------------------Update current status------------------------%
if e2<e1                                                               %
    e1=e2;                                                             %
    OP=OP2;                                                            %
    ww12=wwa;                                                          %
    ww22=wwb;                                                          %
end;                                                                   %
    end;                                                               %
%---------------------Update center and radius----------------------%
if e1<e                                                                %
    r=(3/2)*sqrt(sum(sum((ww12-WW1).^2)')+sum(sum((ww22-WW2).^2)')); %
    c1=2*(ww12-WW1)+WW1;                                               %
    c2=2*(ww22-WW2)+WW2;                                               %
end;                                                                   %
%-----------------------------------------------------------------------%
return;



%******************MY CODE ENDS HERE***************************



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [NET,OP,GAIN] = forw1l(fun,IP,WW,k);
% [NET,OP,GAIN] = forw1l(fun,IP,WW,k) - forward computation for one layer
% uses:
%
%    Arguments:
%           fun - string with the name of activation function
%           IP  - np*ni matrix of input vectors (in augumented space)
%           WW  - no*ni  weight matrix.
%               ni - number of inputs
%               no - number of outputs
%               np - number of patterns
```

```
%            k - gain of neurons at net=0
%    Returns:
%           OP   - np*no matrix of actual outputs.
%           NET  - np*no matrix of net values
%           GAIN - np*no neuron gains

if nargin ~= 4  error('Wrong number of arguments');   end

NET=IP*WW';
[OP,GAIN]=feval(fun,NET,k);
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [e,EE] = oerror(DP,OP);
% [e,EE] = oerror(DP,OP);  error computation for one layer
% uses:
%
%    Arguments:
%           OP  - [np,no] matrix of outputs
%           DP  - [np,no] matrix of desired outputs
%    Returns:
%           e   - total error
%           EE  - [np,no] error matrix (d-o)

if nargin ~= 2  error('Wrong number of arguments.');   end;
sprintf('test  error ');
[np,no]=size(DP);
  EE=DP-OP;
  e = sqrt(sum(sum(EE.*EE)))/(2*no*np);
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [IP] = augm(IP);
% [IP] = augm(IP) create augmented input matrix
% uses:
%
%    Arguments:
%           IP -  [np,ni] matrix
%    Returns:
%           IP  - [np,ni+1] matrix with +1 on last comumn

if nargin ~= 1   error('Wrong number of arguments.');   end;
[np,ni]=size(IP);
IP(:,ni+1)=ones(np,1);
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [o,gain] = fsbip(net,k)
% [o,gain] = fsbip(net,k) - bipolar sigmoidal function
% uses:
%
%    Arguments:
%           net - input variable
%           k  - gain at net=0
%    Returns:
%           o - output value
%           gain - neuron gain

if nargin > 2    error('Wrong number of arguments.');  end
  o = 2*ones./(ones+exp(-2*k*net))-ones;
  gain = k.*(1-o.*o);
```

```
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [o,gain] = fslin(net,k)
% [o,gain] = fsemlin(net,k) - semi linear function
% uses:
%
%    Arguments:
%           net - input variable
%           k  - gain at net=0
%    Returns:
%           o - output value
%           gain - neuron gain

if nargin > 2    error('Wrong number of arguments.');  end
  o = k*net;
for i=1:length(o)
  if o(i)<0
    o(i)=0;
  end;
end
  gain = k;
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [o,gain] = fsuni(net,k)
% [o,gain] = fsuni(net,k) - unipolar sigmoidal function
% uses:
%
%    Arguments:
%           net - input variable
%           k  - gain at net=0
%    Returns:
%           o - output value
%           gain - neuron gain

if nargin > 2    error('Wrong number of arguments.');  end
if nargin == 1  k=1;  end
%k=4*k;
[nr,nc] = size(net);
o = ones./(ones+exp(-k.*net));
gain = k.*(1-o).*o;
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ni,no,np,nh]=versize2(IP,DP,WW1,WW2);
% [ni,no,np,nh]=versize2(IP,DP,WW1,WW2)-verification of matrix sizes for two layer backpropagation input
% uses:
%
%    Arguments:
%           IP  - np*ni matrix of input paterns (not in augmented space)
%           DP  - np*no matrix of desired patterns
%           WW1  - weight matrix for first layer (for augmented space)
%           WW2  - weight matrix for second layer (for augmented space)
%       network structure is extracted form weights matrixes
%    Returns:
%           ni - number of inputs (in augmented space)
%           no - number of outputs
%           np - number of patterns
%           nh - number of hidden neurons
```

```
if nargin ~= 4    error('Wrong number of arguments.');  end
[npi,ni]=size(IP);
[npo,no]=size(DP);
[nh,niw1]=size(WW1);
[now,niw2]=size(WW2);
if npi~=npo   npi, npo, error('Number of input and output patterns differs');     end
if ni~=niw1-1 ni, niw1, error('WW1 does not match input paterns'); end
if no~=now    no, now,  error('WW2 does not match output paterns'); end
if nh~=niw2-1 nh, niw2, error('WW2 does not match hidden neurons'); end
np=npi;
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.1 VSGA.m

```
function [x0,y0] = vsga(obj,x0,y_max,k_max,r_lim,delta,m)
%VSGA   Optimizes objective functions with complex surfaces.
%   VSGA uses a "(V)ariable (S)cale (G)radient (A)pproximation" to solve
%   problems of the form:
%
%                               min  f(X)
%                                x
%
%   where X and the values returned by f can be scalars or vectors.
%
%   [X0,Y0] = VSGA(OBJ,X0,Y_MAX,K_MAX,R_LIM,DELTA,M) returns the minimizer X0
%   and its corresponding objective value Y0.
%
%       Input arguments:
%           OBJ   [handle] : Handle of the objective function
%           X0    [1xn]    : Initial starting point
%           Y_MAX [scalar] : Maximum allowable tolereance for Y0
%           K_MAX [scalar] : Maximum number of iterations
%           R_LIM [1x2]    : Radius limits [r_min,r_max]
%           M     [scalar] : Secondary population size
%
%   For details of the algorithm's operation, see
%
%       Hewlett, J.D.; Wilamowski, B.M.; Dundar, G., "Optimization Using a
%       Modified Second-Order Approach With Evolutionary Enhancement,"
%       Industrial Electronics, IEEE Transactions on , vol.55, no.9,
%       pp.3374-3380, Sept. 2008
%
%   OBJ - The objective function is defined as a separate MATLAB function.
%   For example,
%
%       function [y]=testT1(x)
%       y=((.25*x(1))^4+(.25*x(2))^4);
%       return
%
%   Example:
%
%       [x0,y0] = vsga(@testT2,[100,100],1e-6,100,[1e-6,3],1,10);
%
%   minimizes testT2.

% Joel Hewlett, Mar. 2009
% Auburn University Department of Electrical and Computer Engineering
% $Revision: 1.0 $  $Date: 2009/03/27 13:42:26 $

%---------------------------------------%---------------------------------------%
%   Initialize constants/variables      %
%---------------------------------------%
output = ['%1.8e <== Total Error ',...
'%1.1e <== radius %d <== iteration'];
y0 = feval(obj,x0);                     % Initial value of the objective
y_best = [y0,zeros(1,k_max-1)];         % Holds value of y0 for each iteration
```

```
r = r_lim(1);                           % Set initial radius equal to r_min
I = eye(length(x0));                    % nxn identity matrix
mu = 0.1;                               % Learning rate for LM
mut = mu;                               % Stores mu when radius changes
chk = 0;                                % Set when radius is growing
k = 1;                                  % Iteration counter
rcount = 0;
%---------------------------------------%---------------------------------------%
%   Begin Main Loop                     %
%---------------------------------------%
while k <= k_max,
    k = k + 1;
%---------------------------------------%
%   Compute Gradient                    %
%---------------------------------------%
    [grad,x_opt] = popgrad(x0,obj,r,m); % Caculate gradient approximation
    y_opt=feval(obj,x_opt);
    Hessian = grad'*grad;
    if rcond(Hessian)<0.5               % If Hessian is not well conditioned,
        Hessian=0*Hessian;              %   set it equal to 0.
    end
    count = 0;                          % Counter for LM mu update
    temp = y0;
%---------------------------------------%
%  Apply Update Rule                    %
%---------------------------------------%
    while (1),
        step=((Hessian+mu*I)\grad'*y0)';% Levenberg Marquardt step
        dir_step=step/sqrt(step*step'); % Diversity offset
        x_np1=x0-(step+r*dir_step);
        y_np1=feval(obj,x_np1);
        if chk == 0;
        if y_np1<=temp
            temp = y_np1;
            if mu>1e-50, mu=mu/10; end;
            break;
        end;
        if mu<1e+50, mu=mu*10; end;
        end
        count = count+1;
        if count>2, break; end;
    end;
%---------------------------------------%
%   Assemble Population                 %
%---------------------------------------%
    P=[x_opt;x0;x_np1];
    y=[y_opt,y0,y_np1];
%---------------------------------------%
%   Perform Selection                   %
%---------------------------------------%
    ndex=find(y==min(y));               % Find index of best y value
    y0=y(ndex(1));                      % Update y0
    x0=P(ndex(1),:);                    % Update x0
    y_best(k) = y0;
    disp(sprintf(output,y0,r,k));       % Display progress to the command window
    if y0 < y_max                       % Check if y0 is sufficiently small
        break;
    end
%---------------------------------------%
%   Check Progress/Adjust Radius        %
%---------------------------------------%
    if (y_best(k) == y_best(k-1))       % Check progress
        chk=1;
        if r >= r_lim(2)                % If r has reached its upper limit,
```

```
            r=r_lim(1);                 %   reset r.
        else
            r = r + delta;              % Update r
        end
        mu = 0.1;
    else
        if chk == 1;
           mu = mut;                    % Restore mu
        else
           mut = mu;                    % Store current mu
        end
        chk=0;
    end
end
%---------------------------------------%---------------------------------------%
%   Plot y0 as function of k            %
%---------------------------------------%
semilogy(y_best)
xlim([1 k]);
xlabel('k');
ylabel('y0');
title('Run Summary')
return
```

B.2 popgrad.m

```
function [grad,x_opt] = popgrad(x0,obj,r,m)
%POPGRAD   Calculate a population based gradient approximation.
%   GRAD = POPGRAD(X0,OBJ,RADIUS,M) returns the gradient approximation of
%   OBJ evaluated at the point X0, using a population radius of RADIUS.
%
%   [GRAD,X_OPT] = POPGRAD(X0,OBJ,RADIUS,M) also returns the vector X_OPT,
%   which corresponds to the member of the generated population with the
%   lowest value for the objective function OBJ. If M = 0, X_OPT is simply
%   chosen from the N points used to calculate GRAD. If M > 0, M randomly
%   generated points within the region defined by X0 and RADIUS are also
%   considered.
%
%       Input arguments:
%           X0      : 1xn vector
%           OBJ     : function handle
%           RADIUS  : scalar (>0)
%           M       : scalar (>=0)
%
%   OBJ - The objective function is defined as a separate MATLAB function.
%   For example,
%
%       function [y]=testT1(x)
%       y=((.25*x(1))^4+(.25*x(2))^4);
%       return
%
%   Example:
%
%       [g,x] = popgrad([10 10],@testT1,1e-6,10);
%
%   finds the the values of g and x for the testT1 using a population
%   radius of 1e-6 and a secondary population size of 10.

% Joel Hewlett, Mar. 2009
% Auburn University Department of Electrical and Computer Engineering
% $Revision: 1.0 $  $Date: 2009/03/27 10:07:26 $

%---------------------------------------%---------------------------------------%
```

```
%   Generate Population (P)             %
%---------------------------------------%
n = length(x0);                         % Size of primary population
A = 2*rand(n)-1;
A = A./(ones(n,1)*sqrt(sum(A'.^2)))'*r;
A = A+ones(n,1)*x0;                     % Primary population (A)
if m > 0
    B=(2*rand(m,n)-1);
    B=B./(ones(n,1)*sqrt(sum(B'.^2)))'*r;
    B=B + ones(m,1)*x0;                 % Secondary population (B)
    P=[A;B];                            % Combined population (P)
else
    P=A;                                % Combined population (P)
end
%---------------------------------------%
%   Evaluate Population Fitness (y)     %
%---------------------------------------%
y0 = feval(obj,x0);
lambda = n+m;                           % Size of combined population
for i = 1:lambda,
    y(i) = feval(obj,P(i,:));
end
%---------------------------------------%
%   Approximate Gradient (grad)         %
%---------------------------------------%
dy = y(1:n)-y0;
dx = P(1:n,:)-ones(n,1)*x0;
grad = (pinv(dx)*dy')';
%---------------------------------------%
%   Perform preliminary selection       %
%---------------------------------------%
y(end+1) = y0;
P(end+1,:)= x0;
index = find(y==min(y));
x_opt = P(index(1),:);
return
```